

- [54] **DATA COMPRESSION AND DECOMPRESSION SYSTEM**
- [72] Inventors: **Paul A. D. De Maine; Gordon K. Springer**, both of State College, Pa.
- [73] Assignee: **Research Corporation**, New York, N.Y.
- [22] Filed: **Sept. 15, 1969**
- [21] Appl. No.: **857,707**

R.W. Bemer, Data Compression System, IBM Tech. Disc. Bull. Vol. 3, No. 8, Jan. 1961

Primary Examiner—Raulfe B. Zache
Assistant Examiner—Sydney R. Chirlin
Attorney—Robert S. Dunham, P. E. Henninger, Lester W. Clark, Gerald W. Griffin, Thomas F. Maran, Howard J. Churchill, R. Bradlee Boal, Christopher C. Dunham, Robert Scobey and Henry T. Burke

- [52] U.S. Cl.444/1
- [51] Int. Cl.G06f 7/06
- [58] Field of Search340/172.5

[57] **ABSTRACT**

A high speed, multistage, compressor-decompressor system for processing arbitrary bit strings by reversibly removing redundant information. Alphanumeric information is processed by Type 1 compression which involves removing patterns of contiguous bytes and replacing each removed pattern by decompression information which takes considerably less storage space, and Type 2 compression which involves removing individual redundant bytes and constructing a bit map identifying the location of the removed bytes. Numerical information is processed by a compression technique involving truncation, recursive differencing, sequence removal, packing, and then utilizing the Type 1 and Type 2 compression which are used in conjunction with alphanumeric information. The information which is to be compressed is arranged in strings of bytes and any information defining removal of redundant information from a string is kept together with the string. As a result, each string is self-defined in the sense that it contains all information needed to decompress that string.

[56] **References Cited**

UNITED STATES PATENTS

3,237,170	2/1966	Blasbalg et al.	340/172.5
3,273,130	9/1966	Baskin et al.	340/172.5
3,289,169	11/1966	Marosz	340/172.5
3,310,786	3/1967	Rinaldi et al.	340/172.5
3,413,611	11/1968	Pfeutze	340/172.5
3,422,403	1/1969	Webb	340/172.5
3,490,690	1/1970	Apple et al.	340/172.5 X
3,535,696	10/1970	Webb	340/172.5

OTHER PUBLICATIONS

P.A.D. De Maine, B. A. Marron, and K. Kloss, The Solid System II; Numeric Compression The Solid System III; Alphanumeric Compression Nat. Bureau of Standards Technical Note 413, Aug. 15, 1967

25 Claims, 38 Drawing Figures

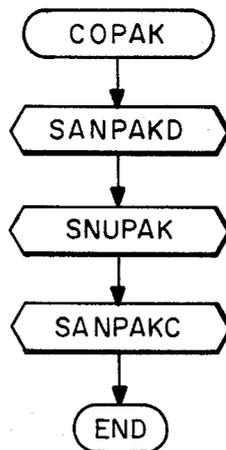
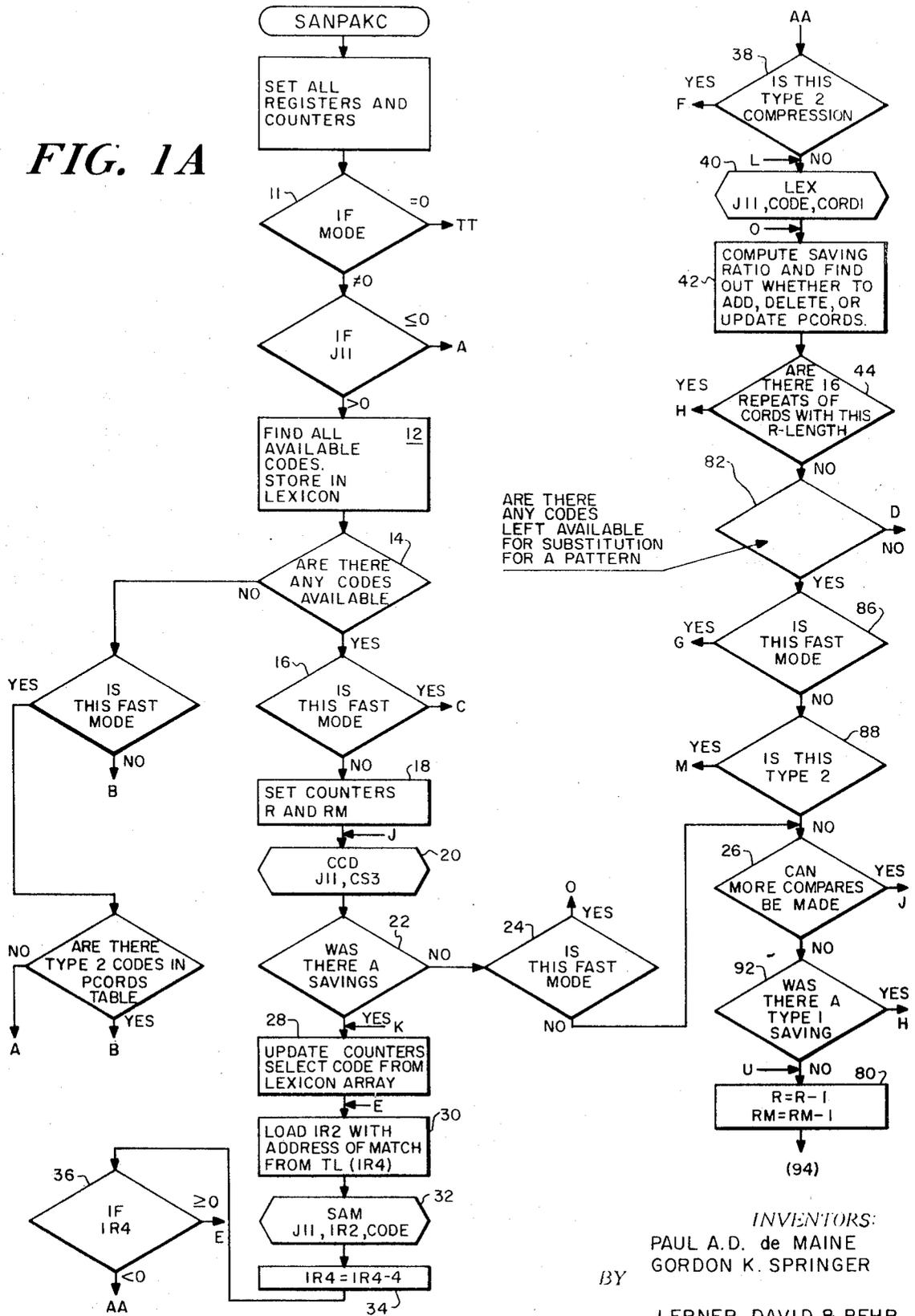
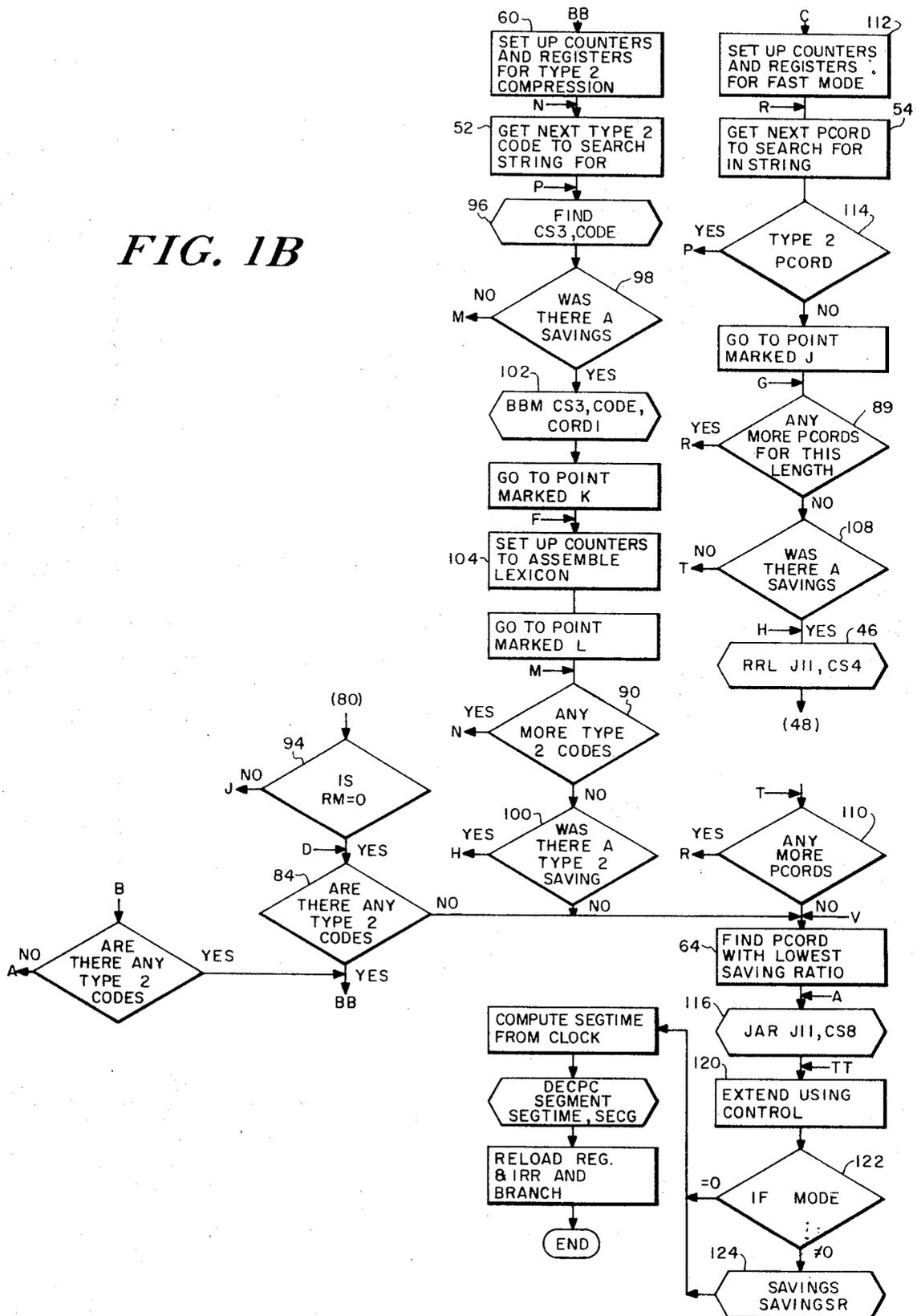


FIG. 1A



INVENTORS:
 PAUL A.D. de MAINE
 GORDON K. SPRINGER
 BY
 LERNER, DAVID & BEHR
 ATTORNEYS

FIG. 1B



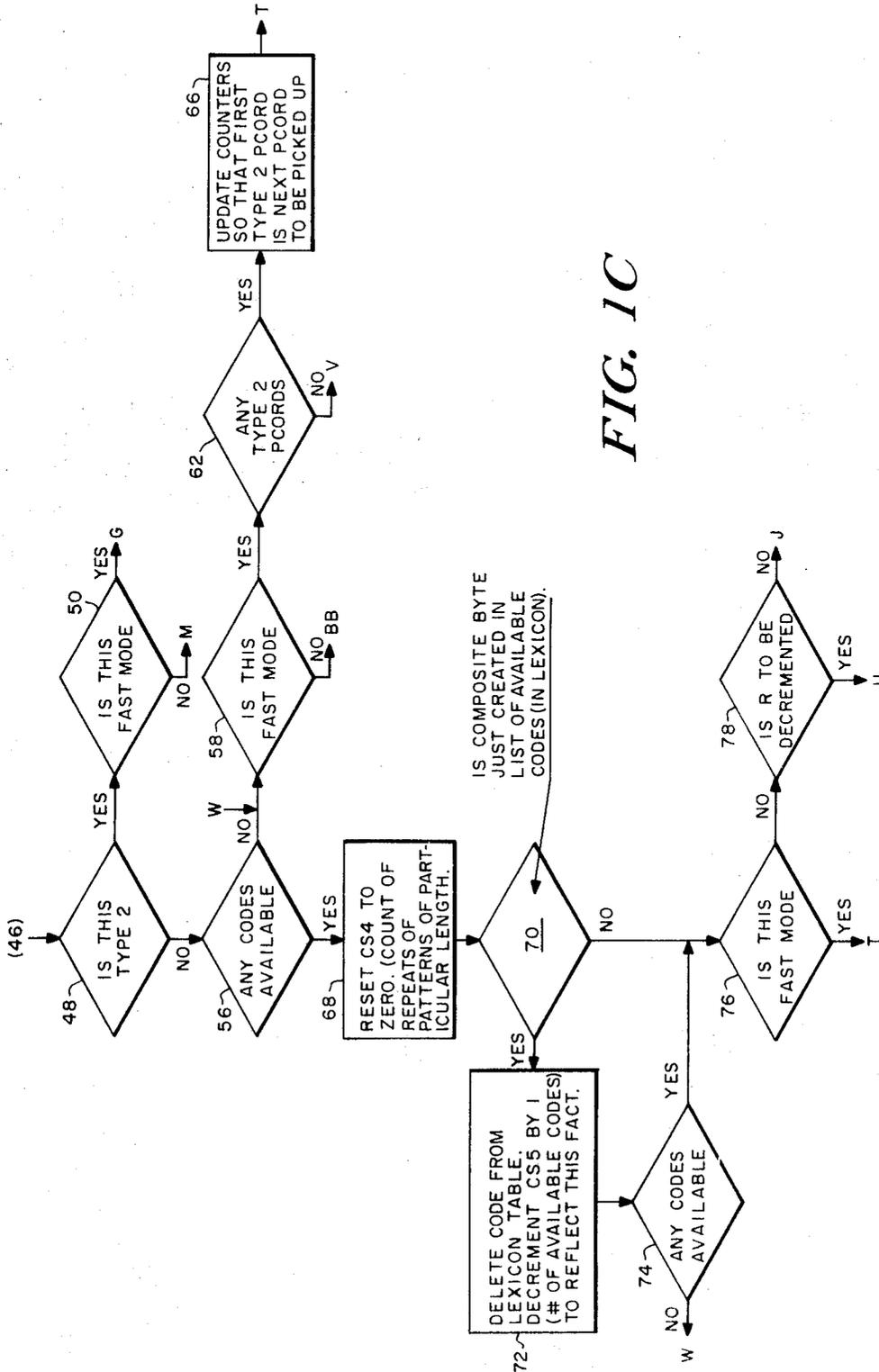
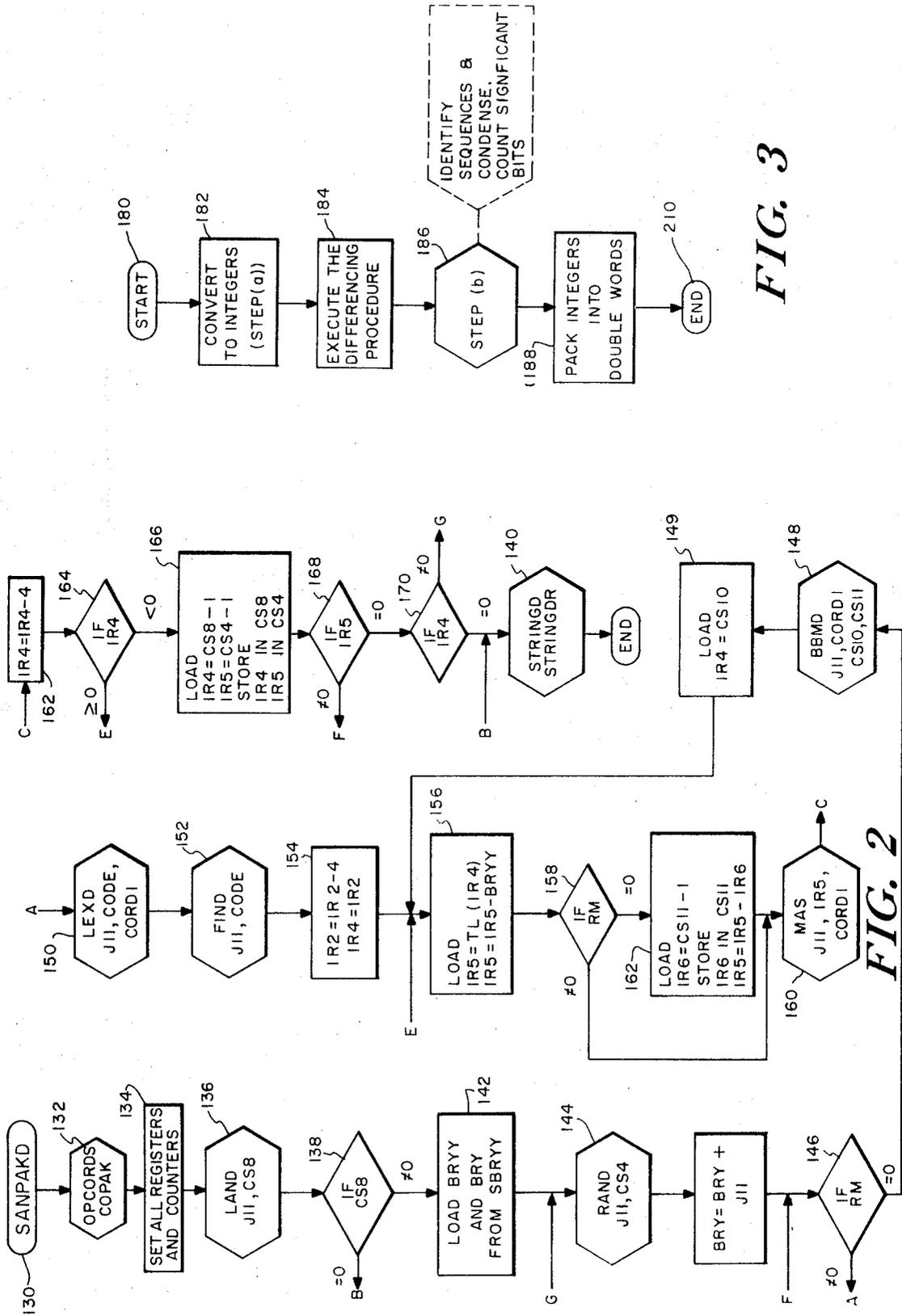


FIG. 1C



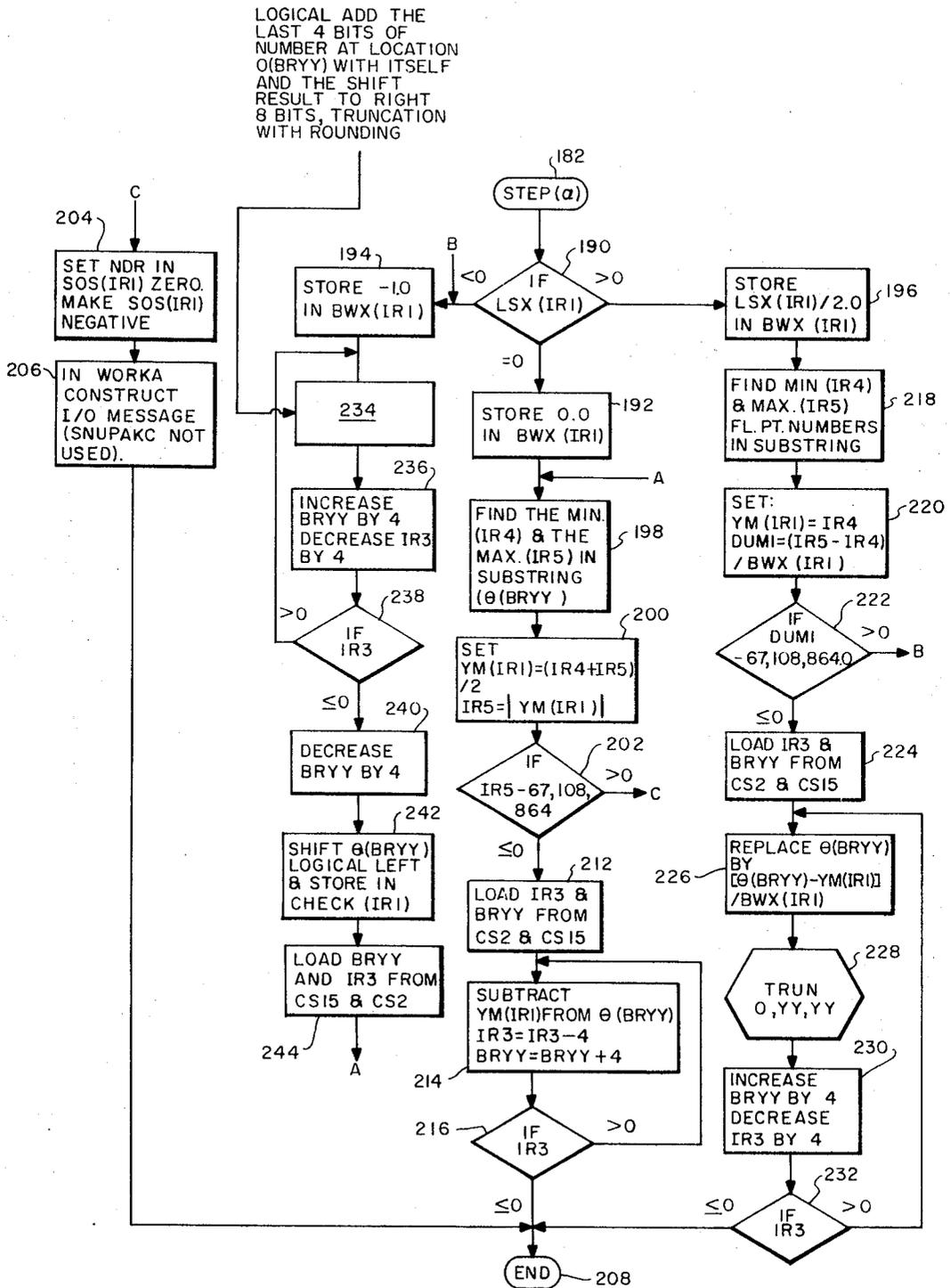


FIG. 4

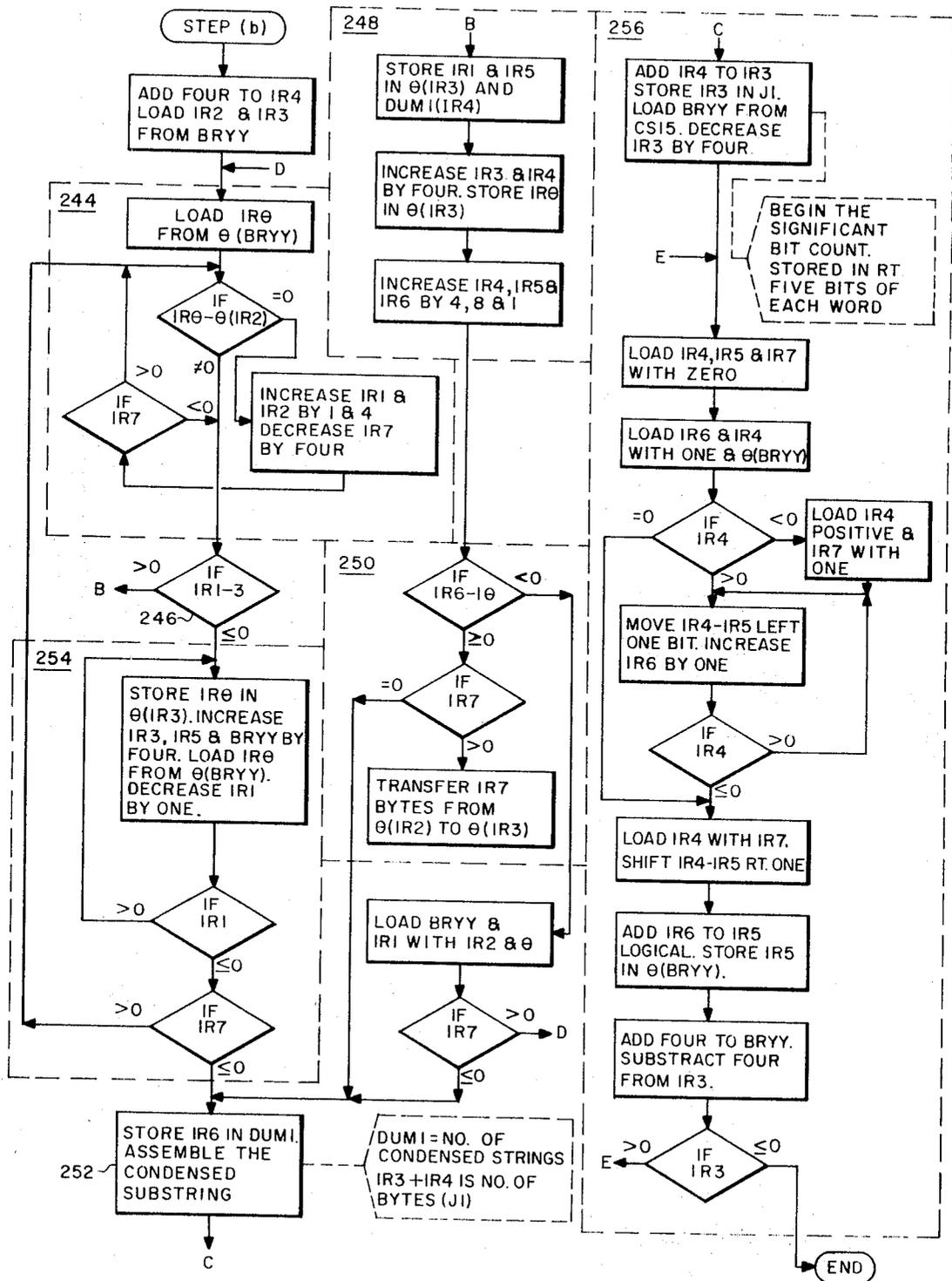


FIG. 5

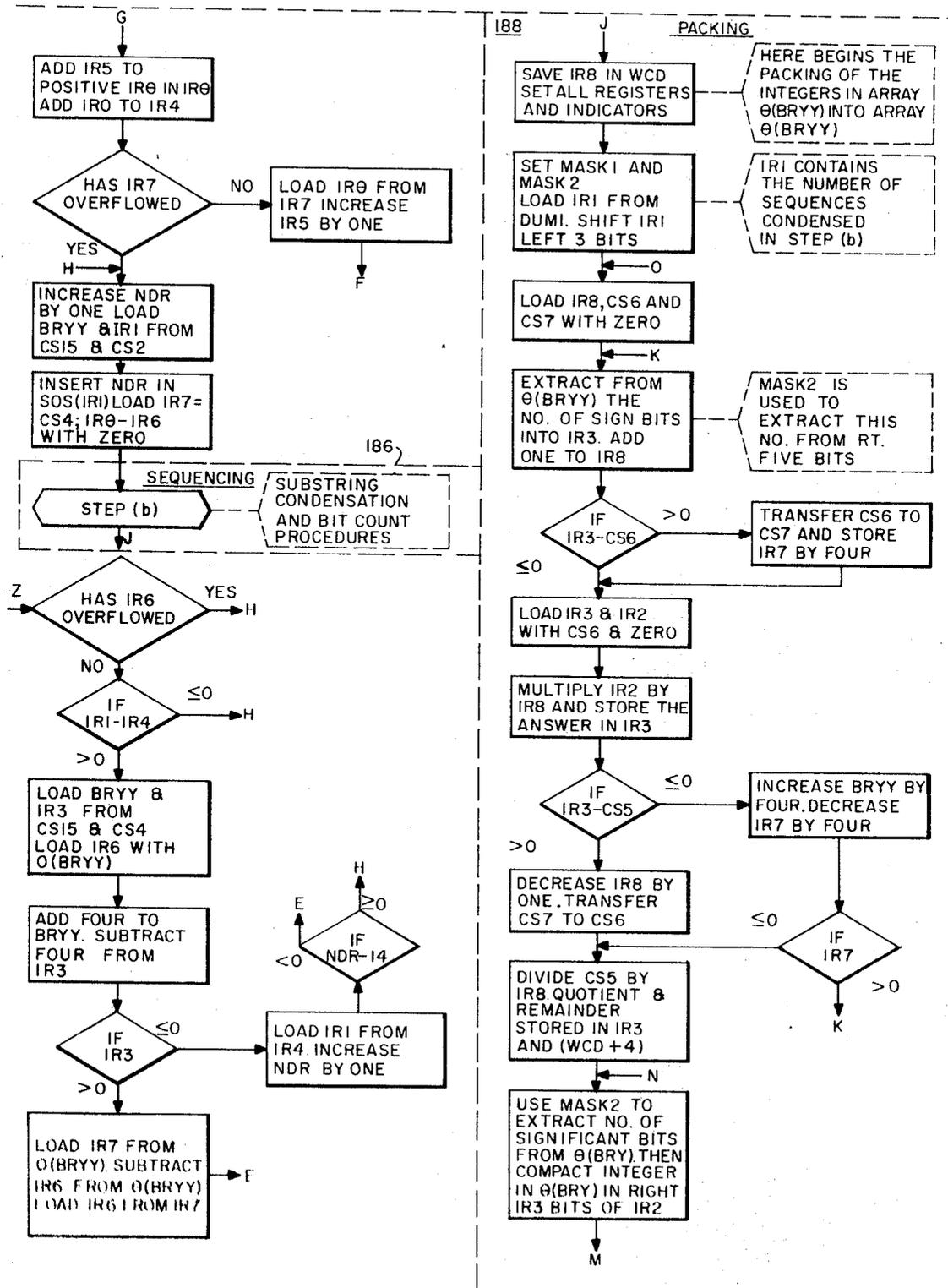


FIG. 6B

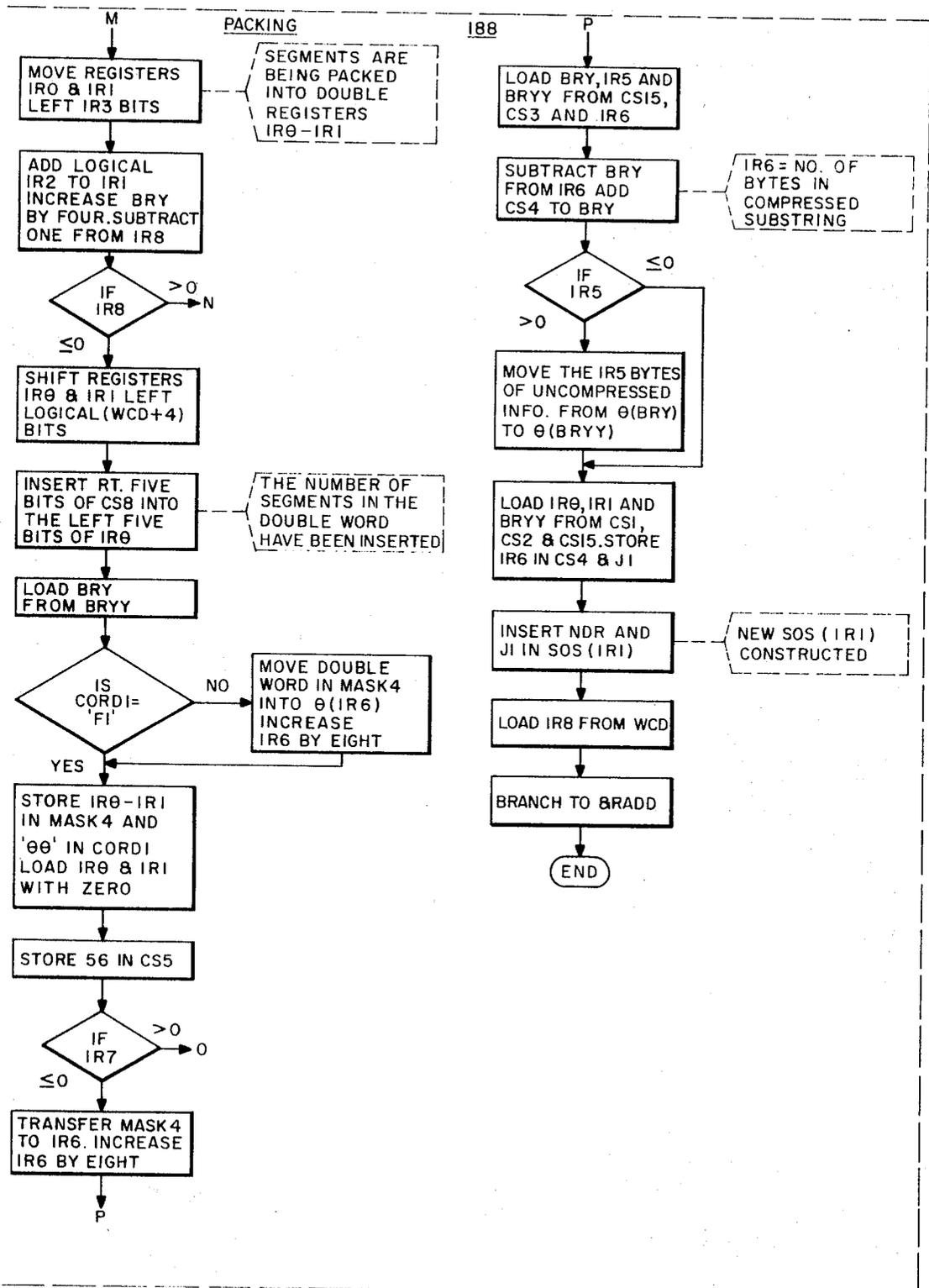


FIG. 6C

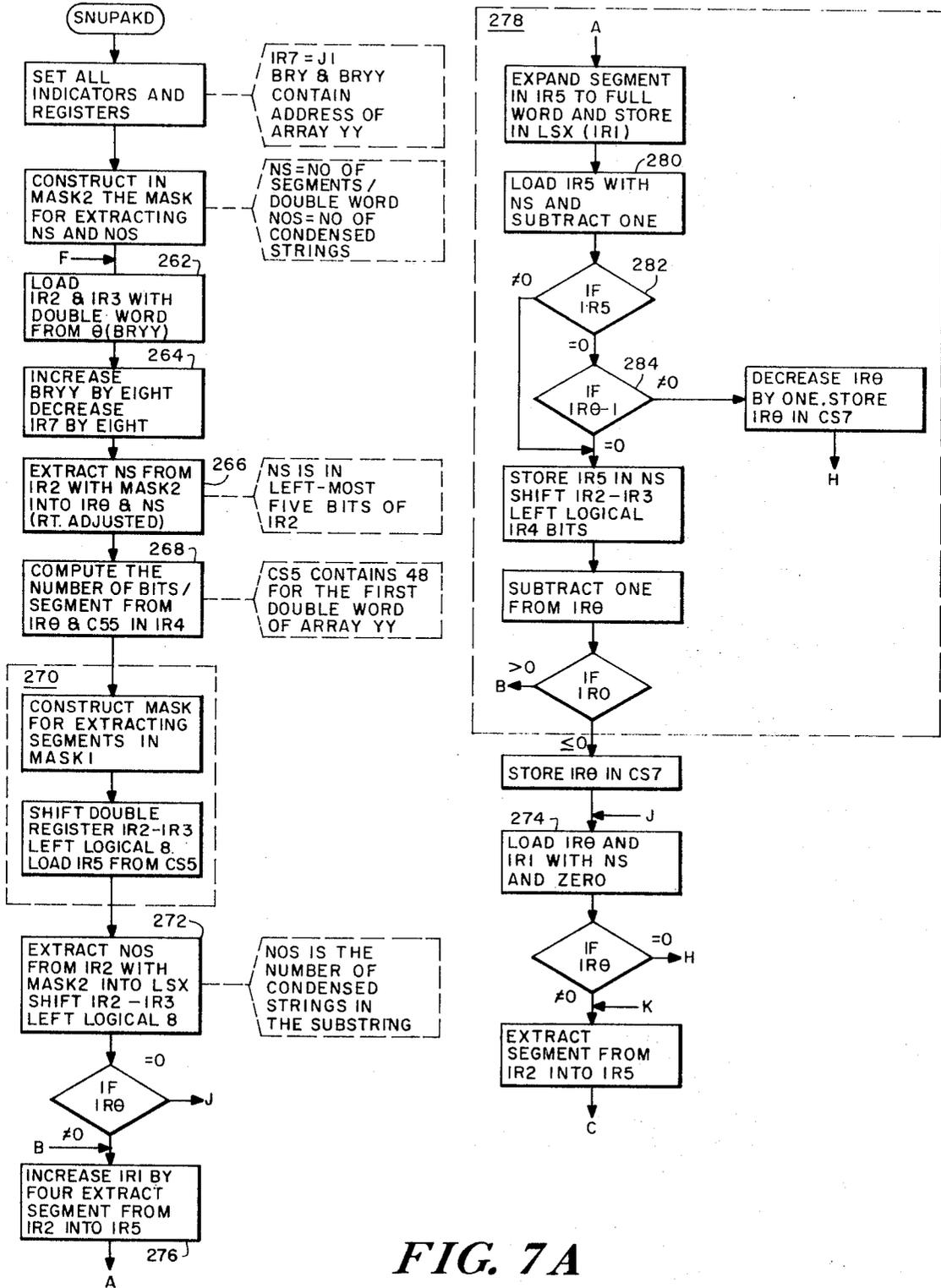


FIG. 7A

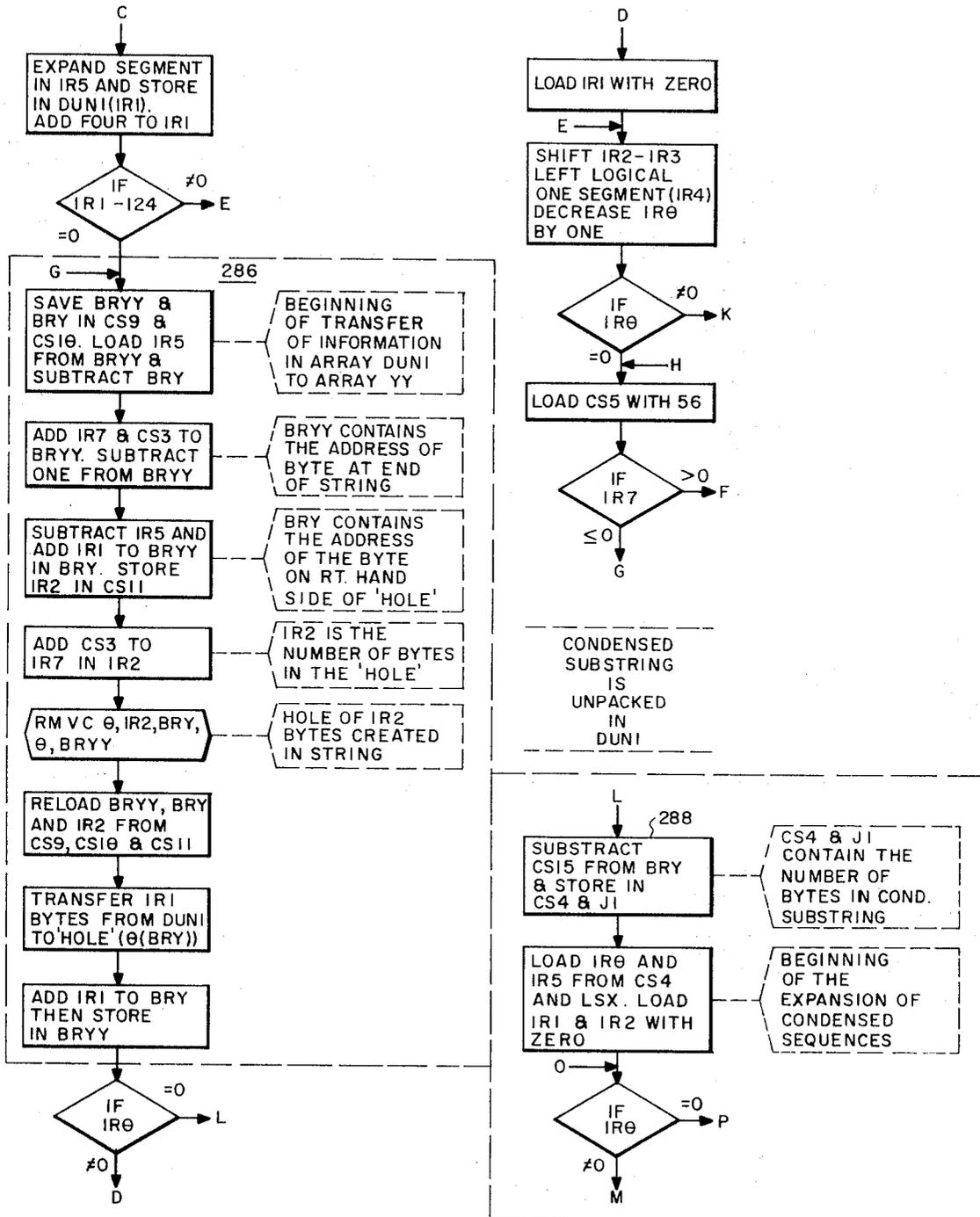


FIG. 7B

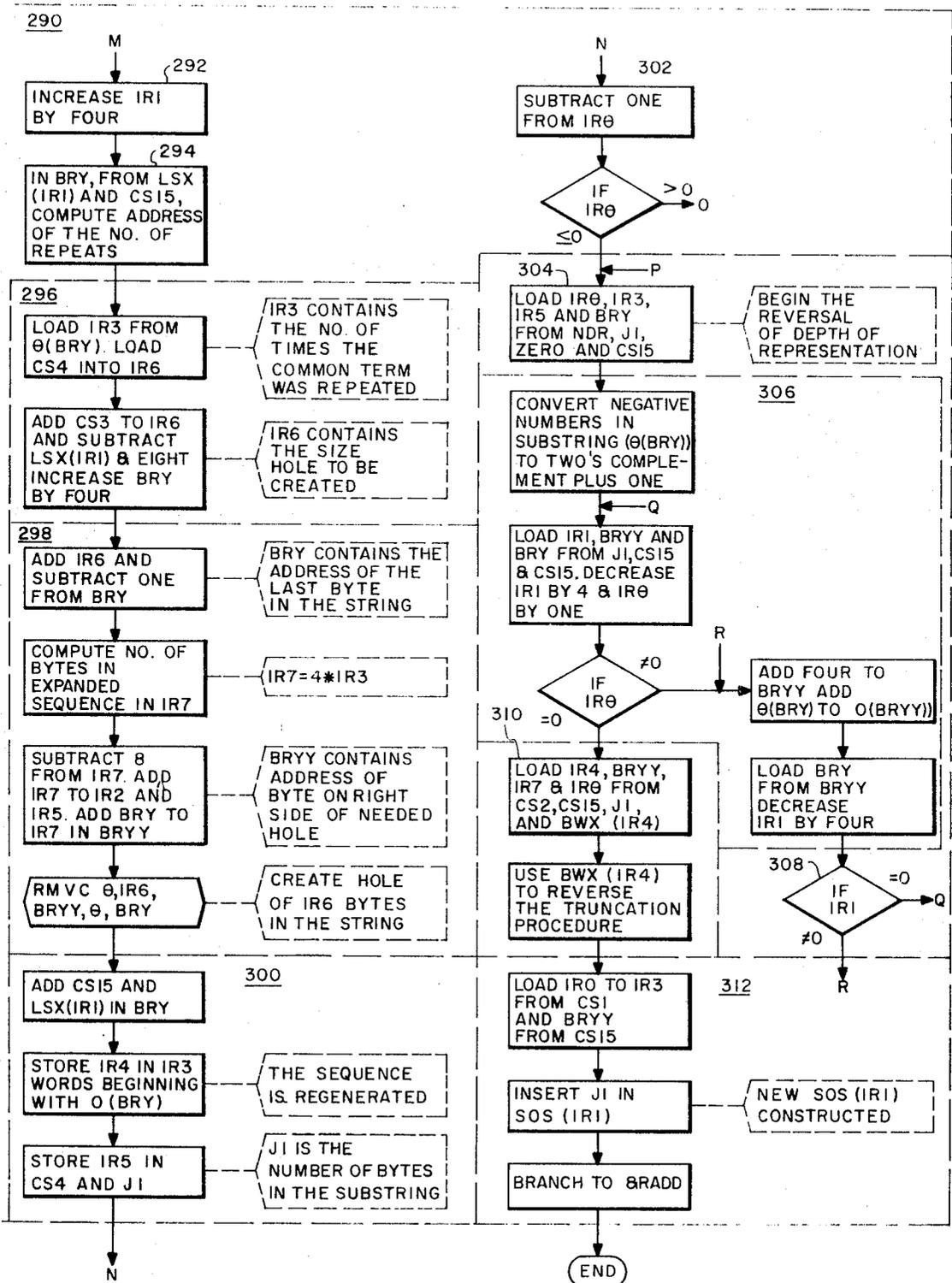


FIG. 7C

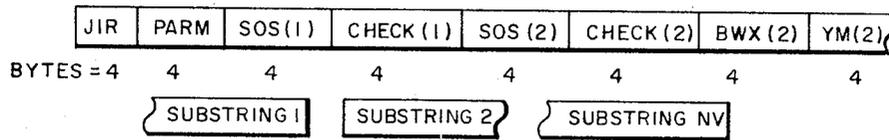


FIG. 8

SOS	LSX	FORMAT-TYPE	COPAK COMPRESSOR OPTION
<0	<0	X	USE SANPAKC ONLY
<0	≥0	A	USE SANPAKC ONLY
=0	=0	I	USE SNUPAK (1) THEN SANPAKC
=0	>0	E AND/OR F	USE SNUPAK (2) THEN SANPAKC
=0	<0	E AND/OR F	USE SNUPAK (3) THEN SANPAKC
>0	-	A	DON'T USE SNUPAK OR SANPAKC

FIG. 9

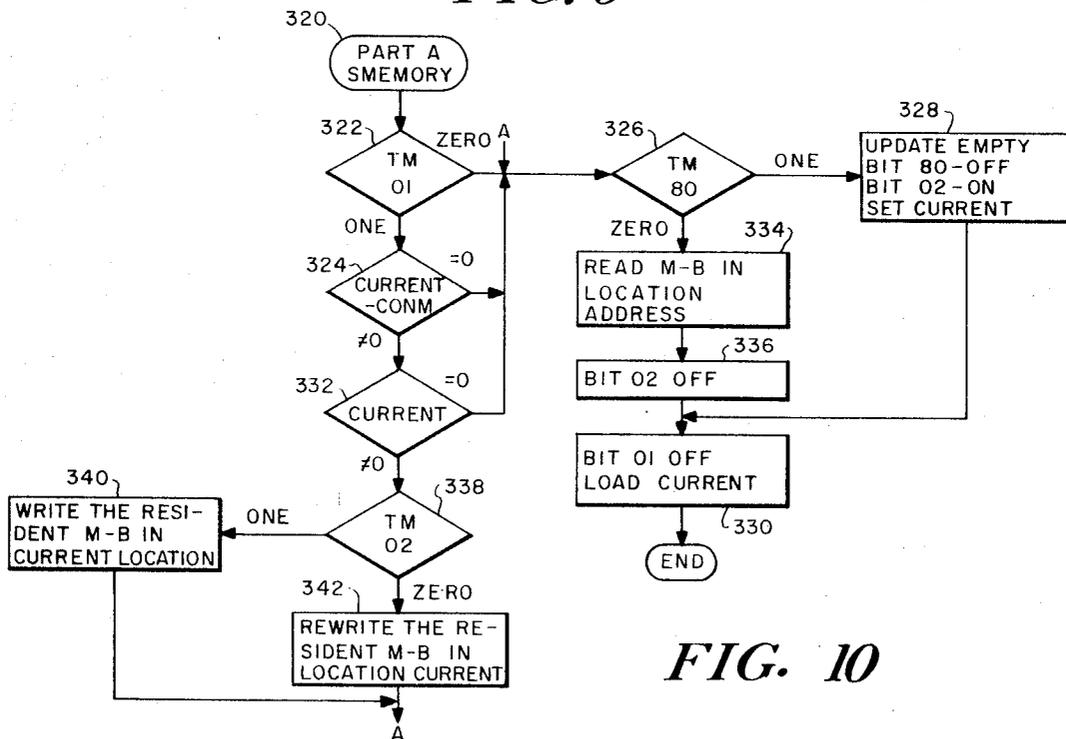


FIG. 10

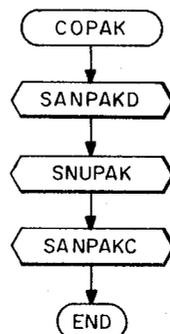


FIG. 11

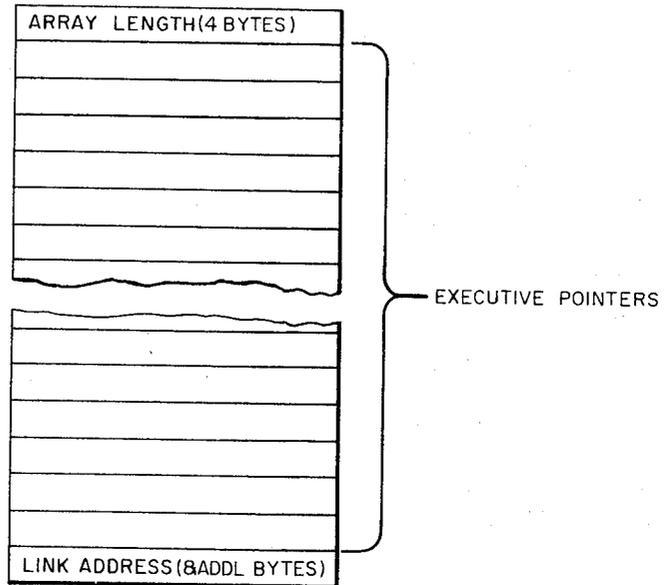


FIG. 12

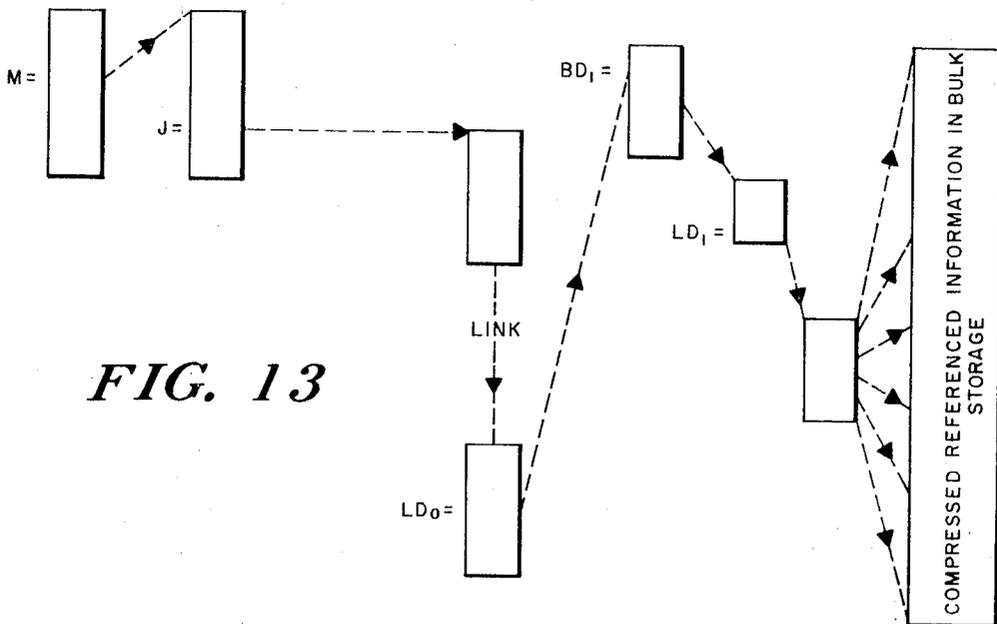


FIG. 13

MA

JA(M)

LD(φ)

BD(1)

LD(1) RFILE

MAIN FILE

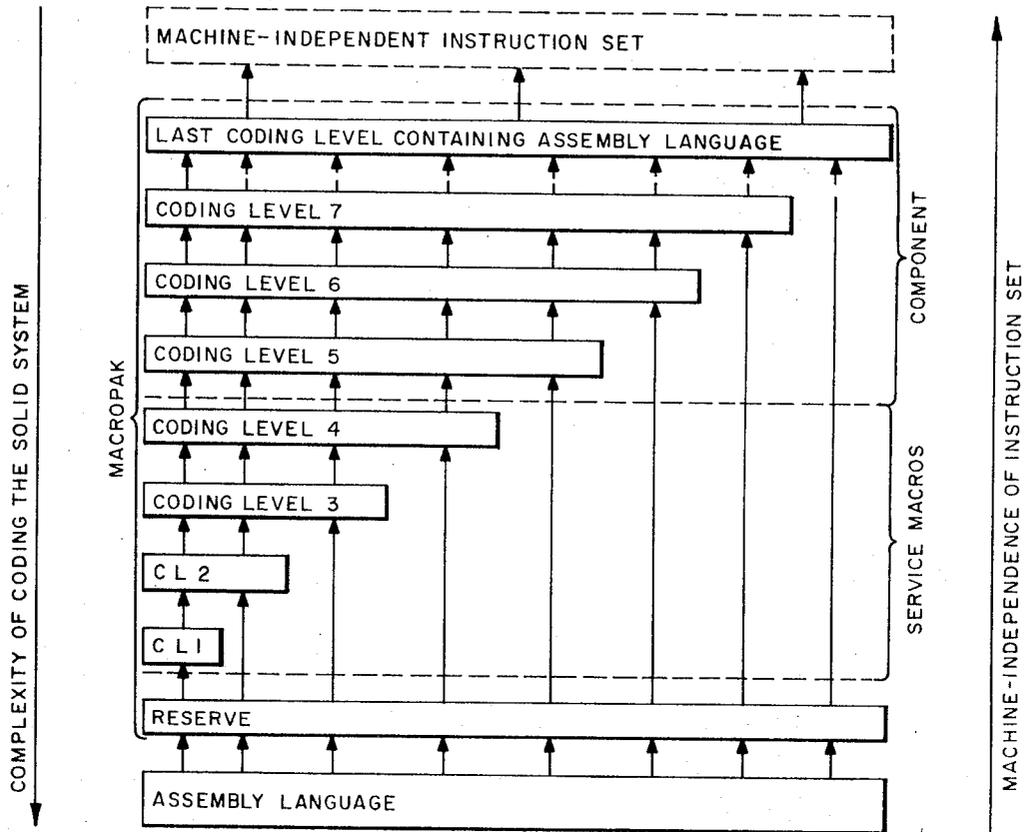


FIG. 14

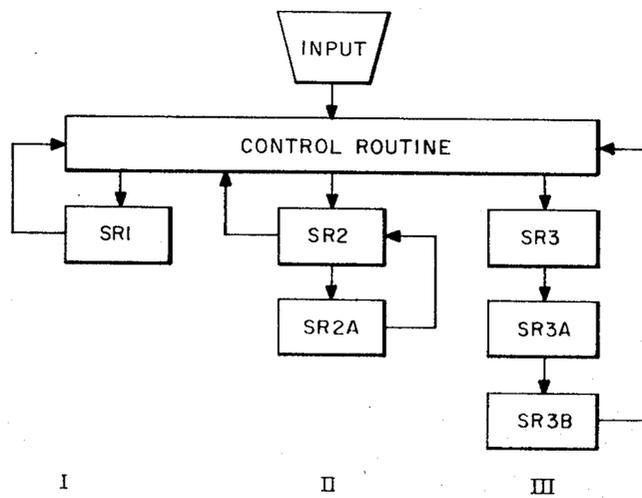


FIG. 15

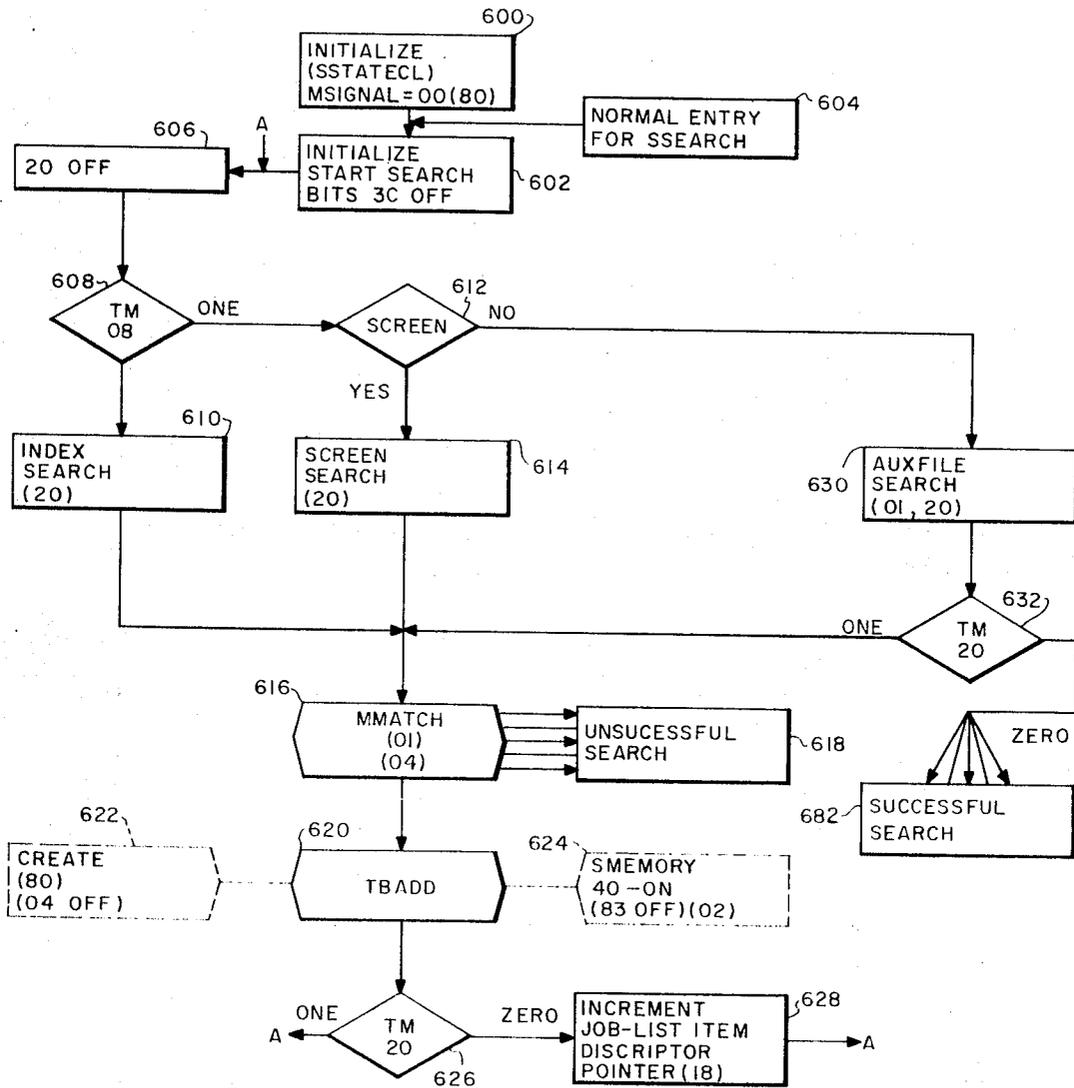


FIG. 16

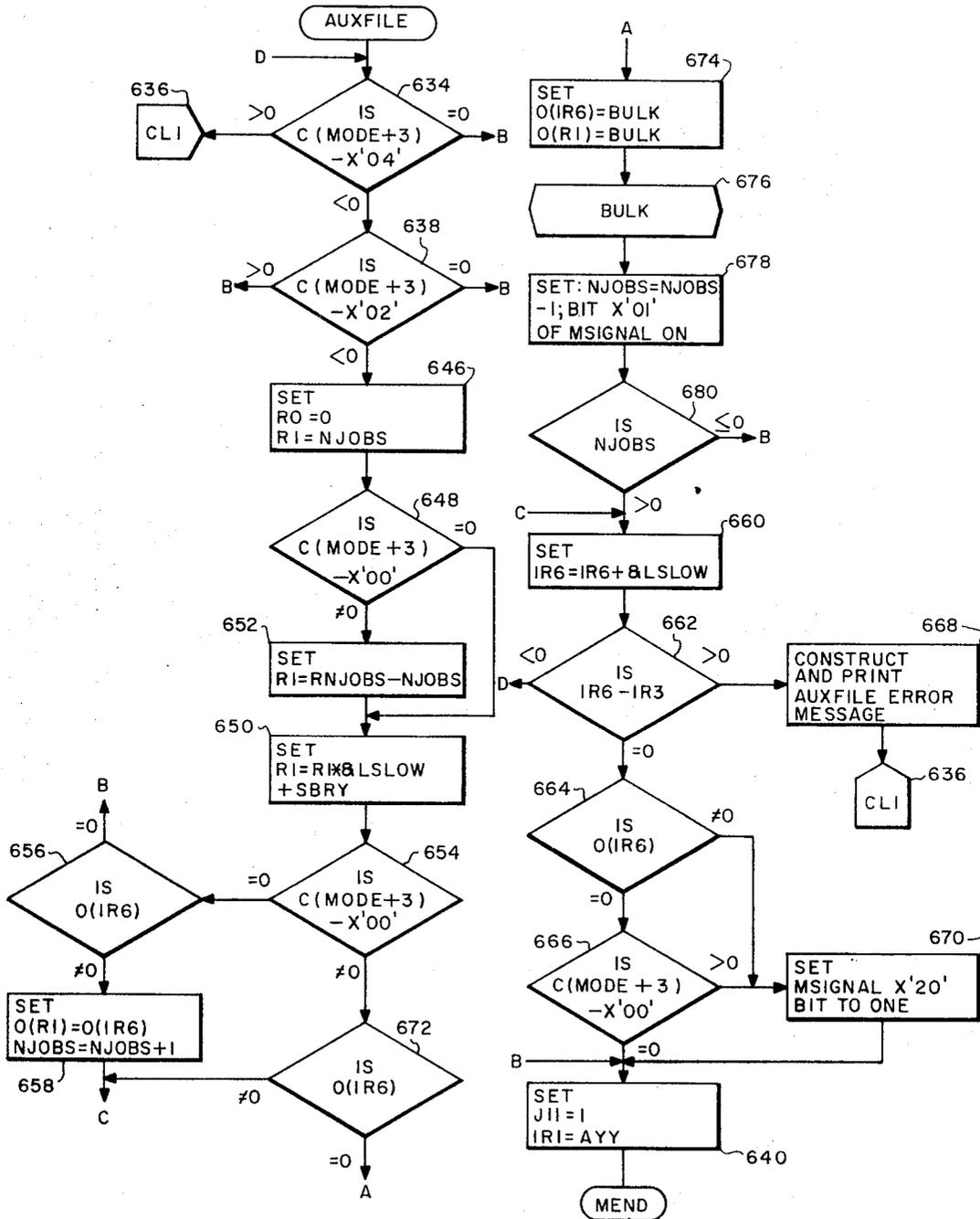


FIG. 17

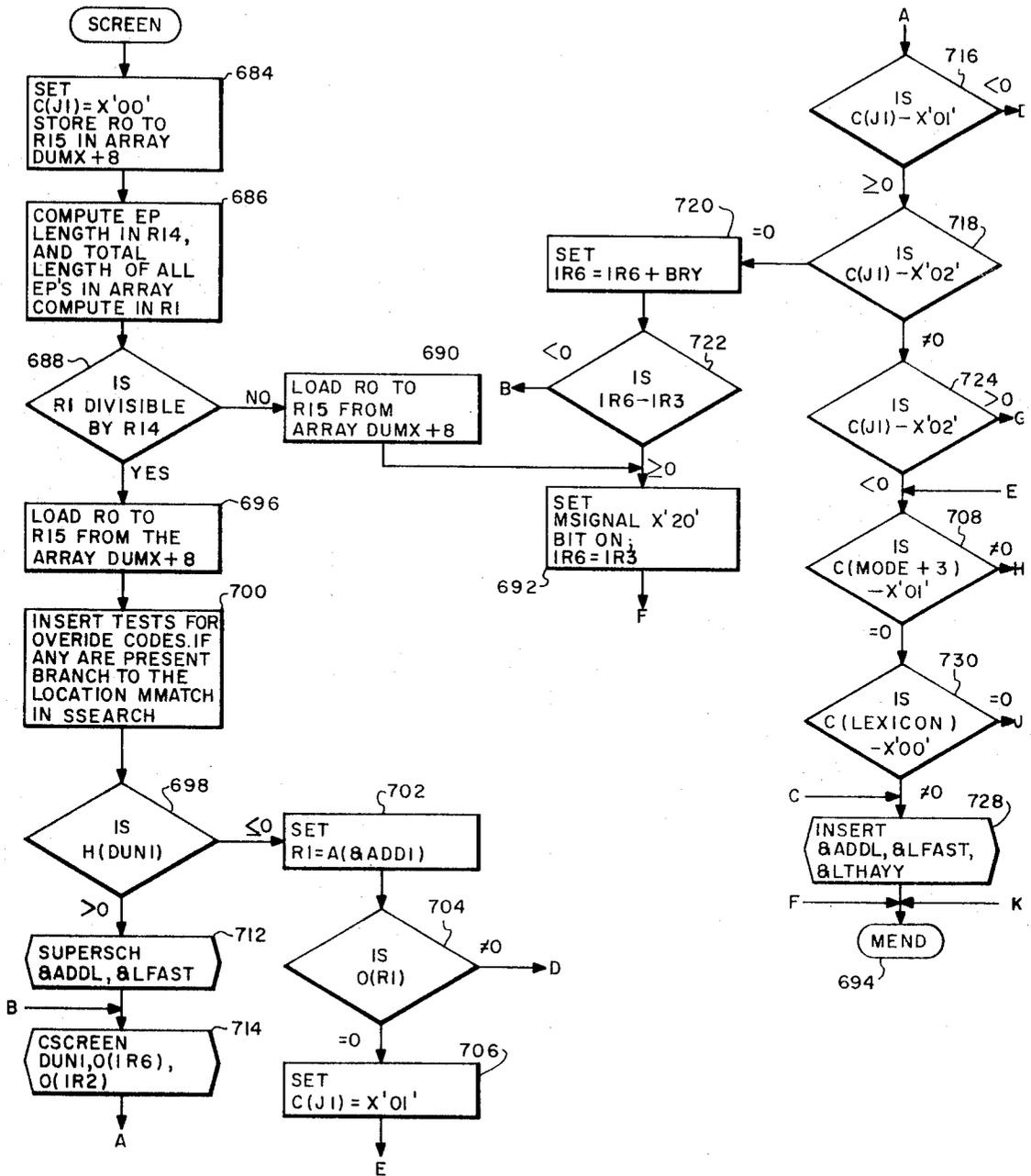


FIG. 18A

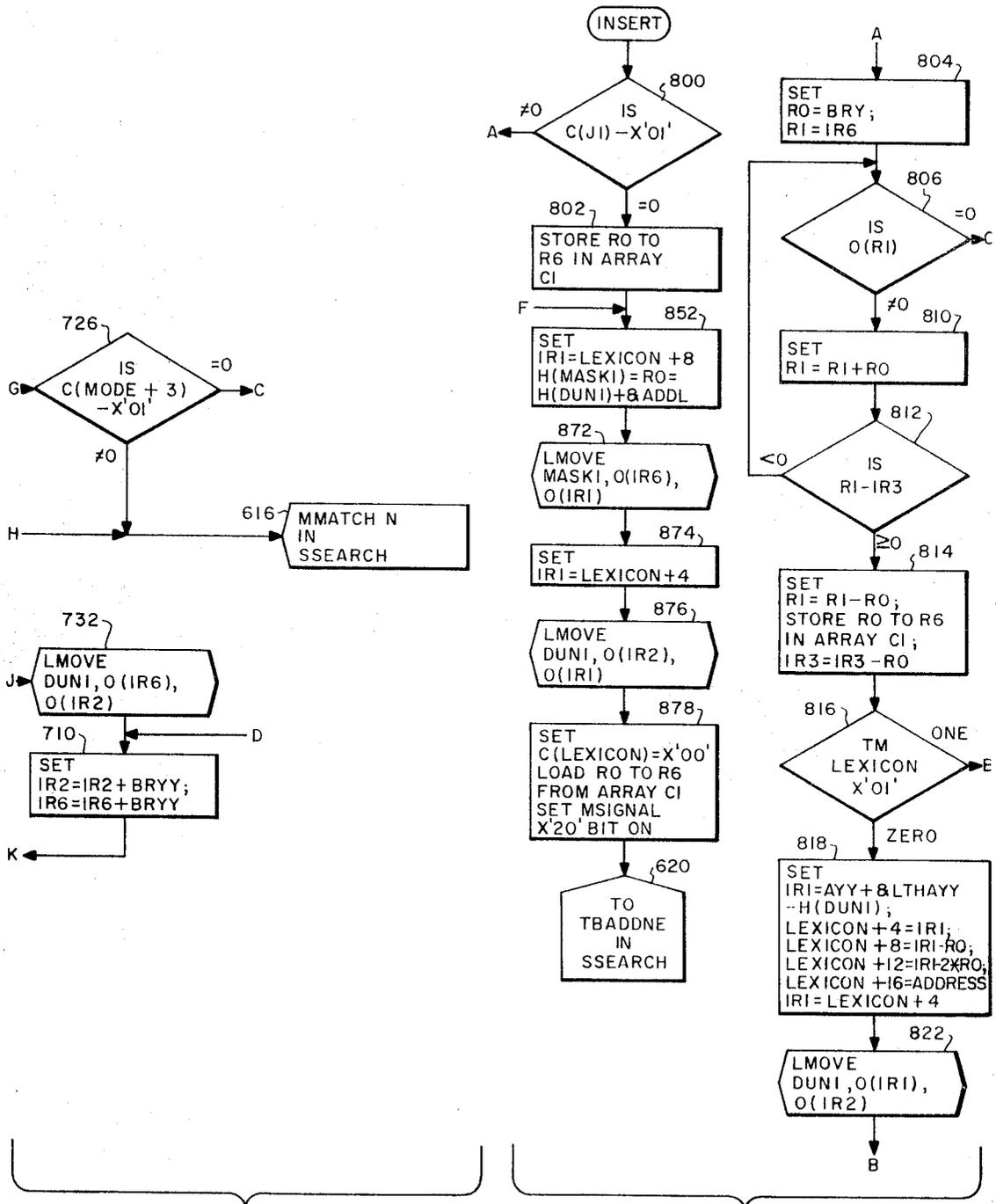


FIG. 18B

FIG. 20A

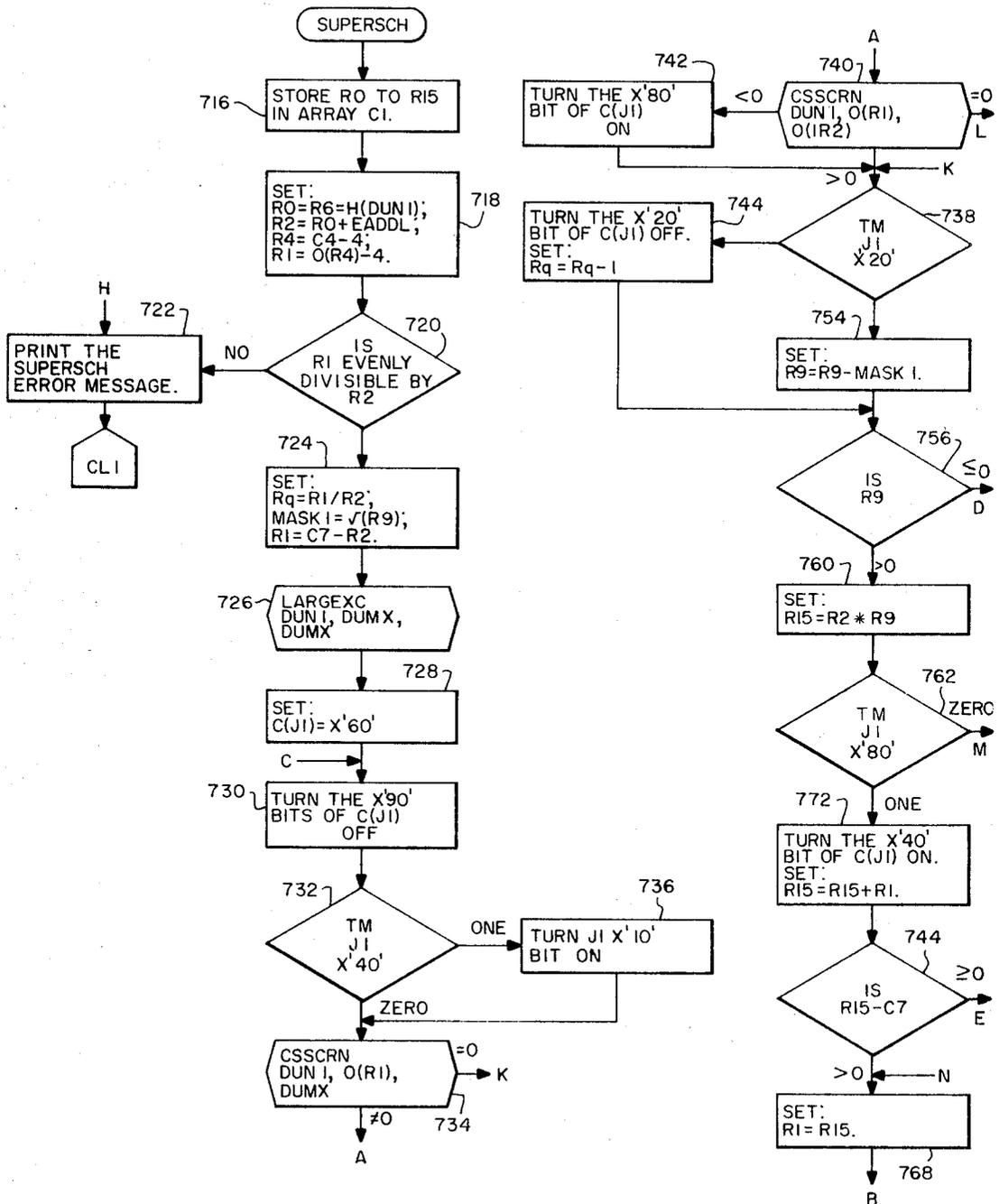


FIG. 19A

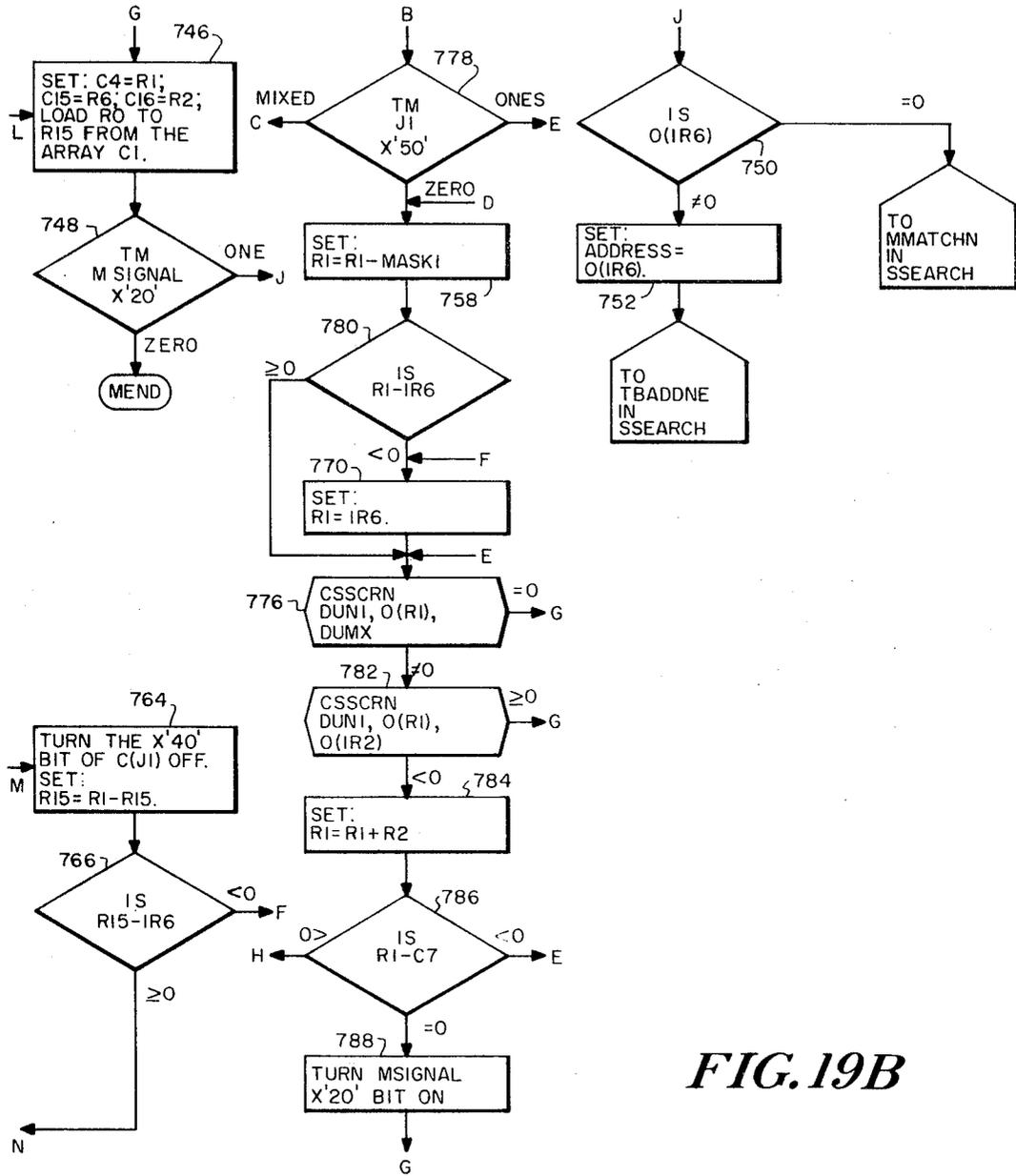


FIG. 19B

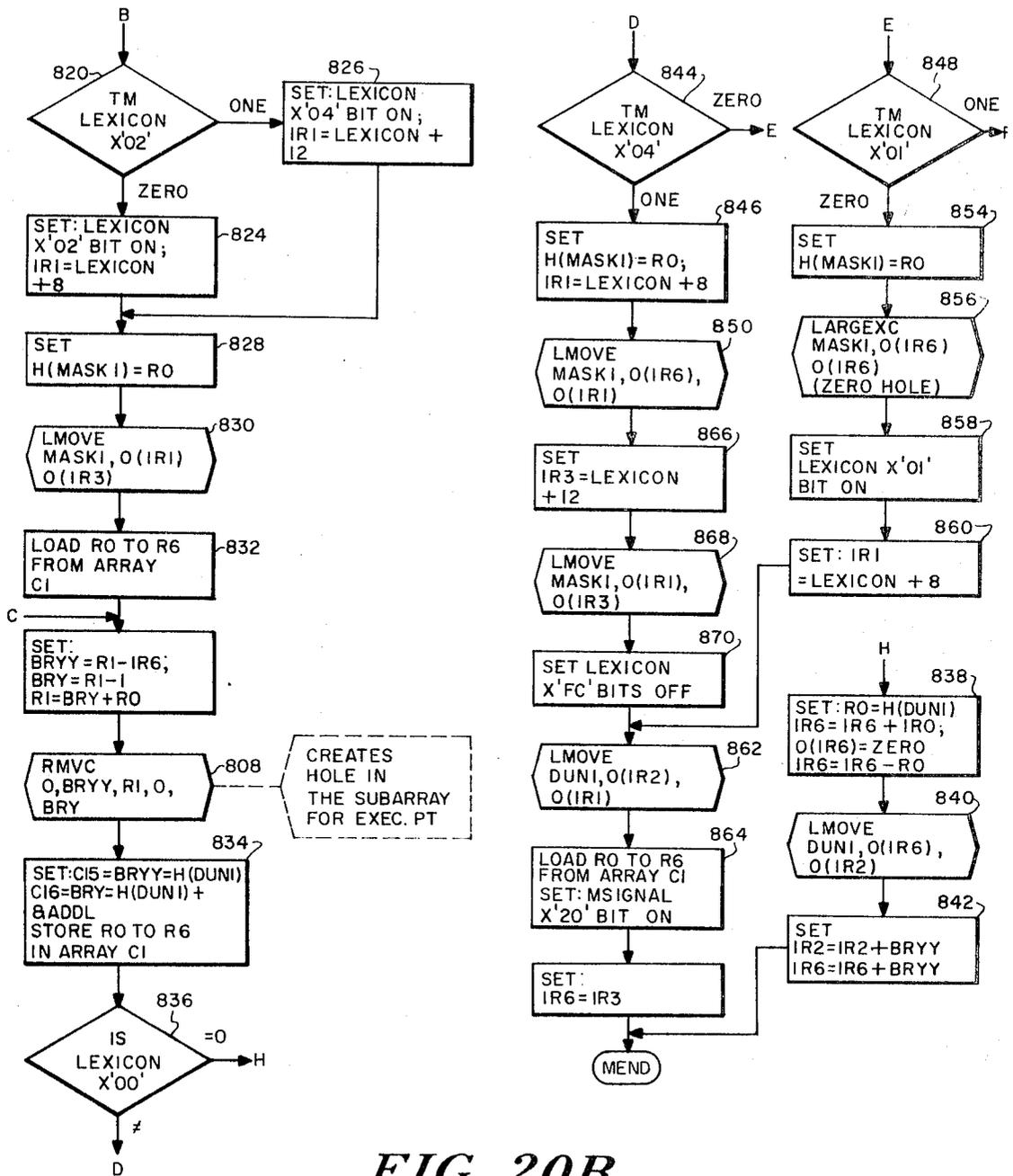


FIG. 20B

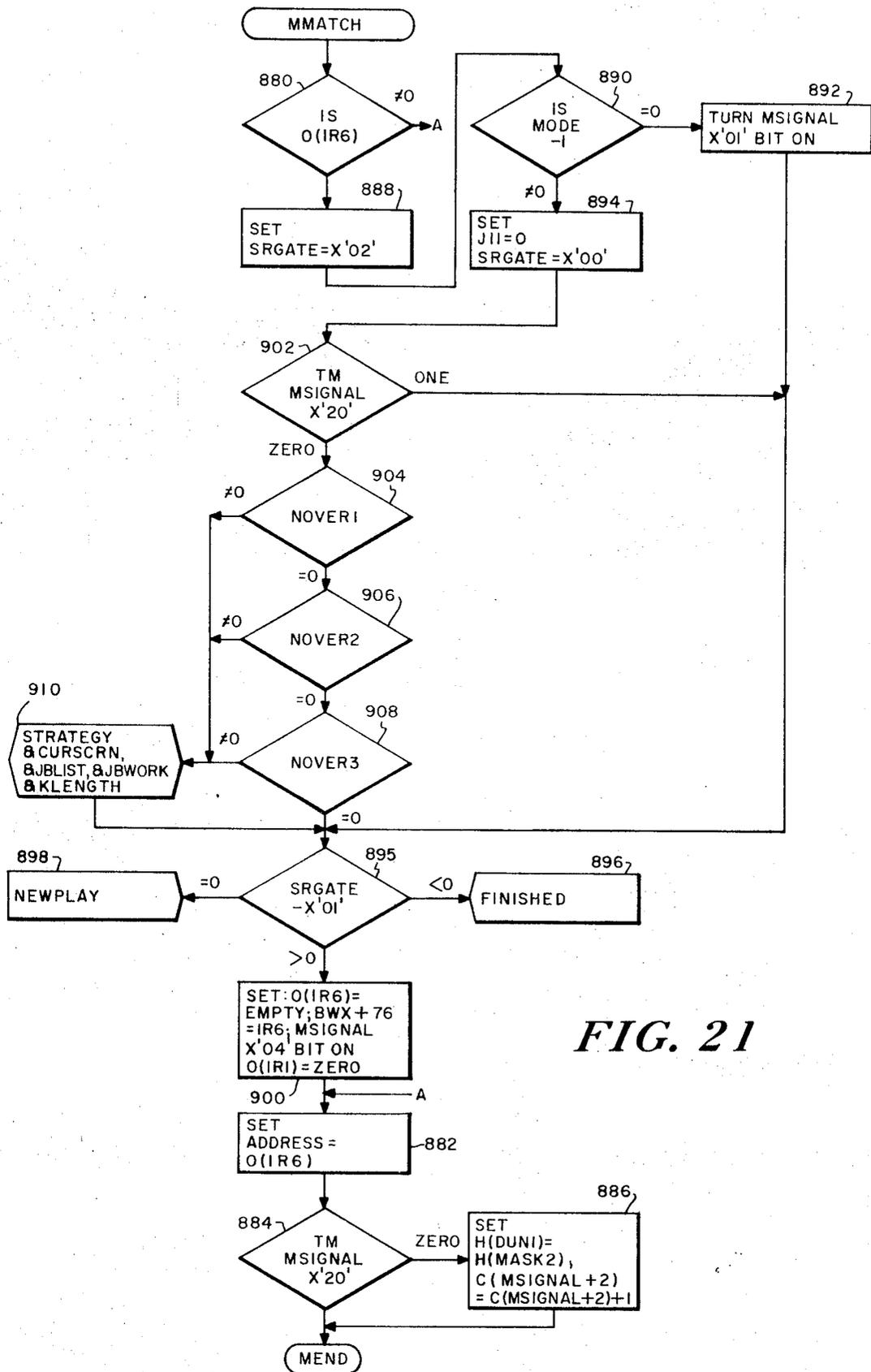


FIG. 21

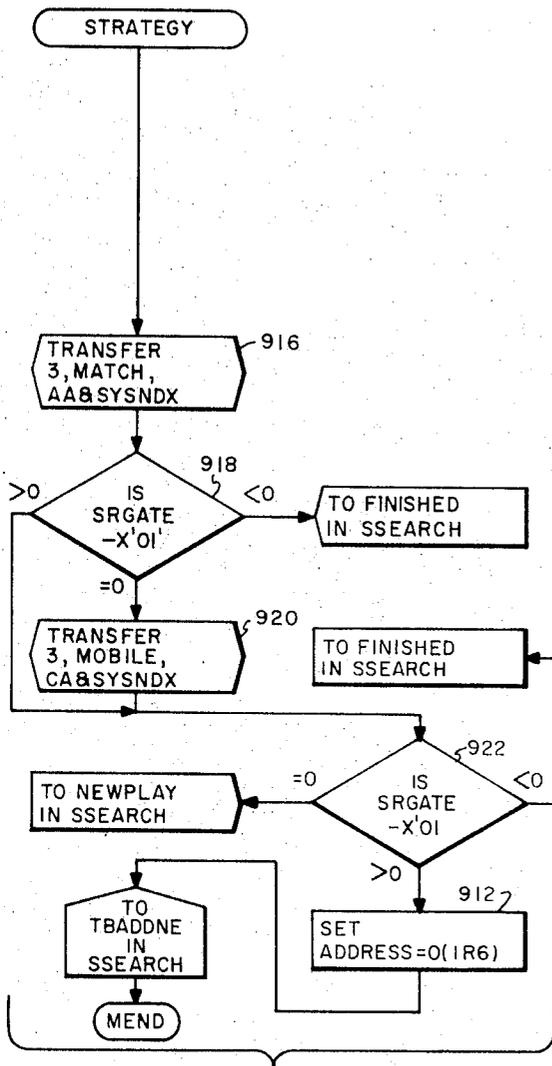


FIG. 22

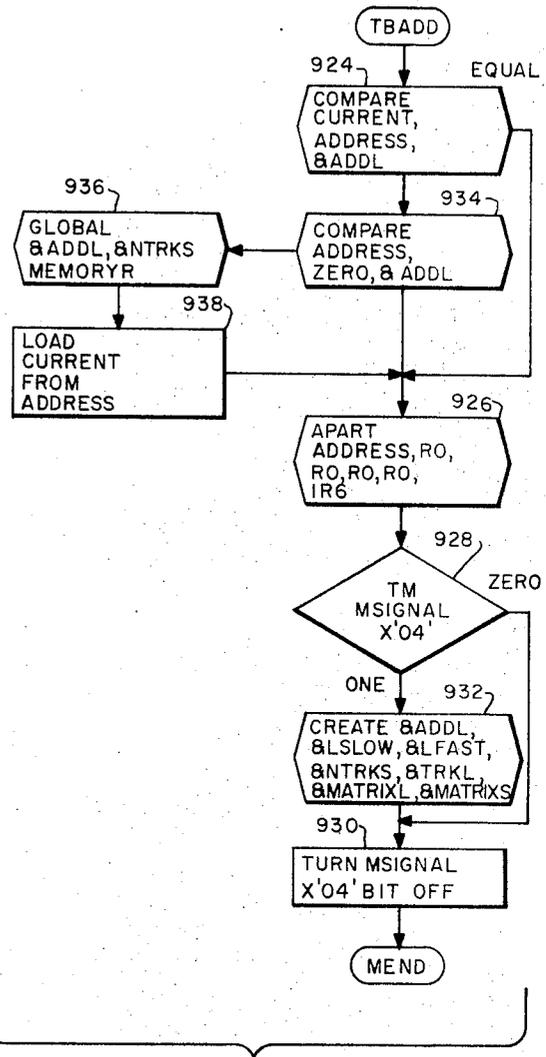


FIG. 23

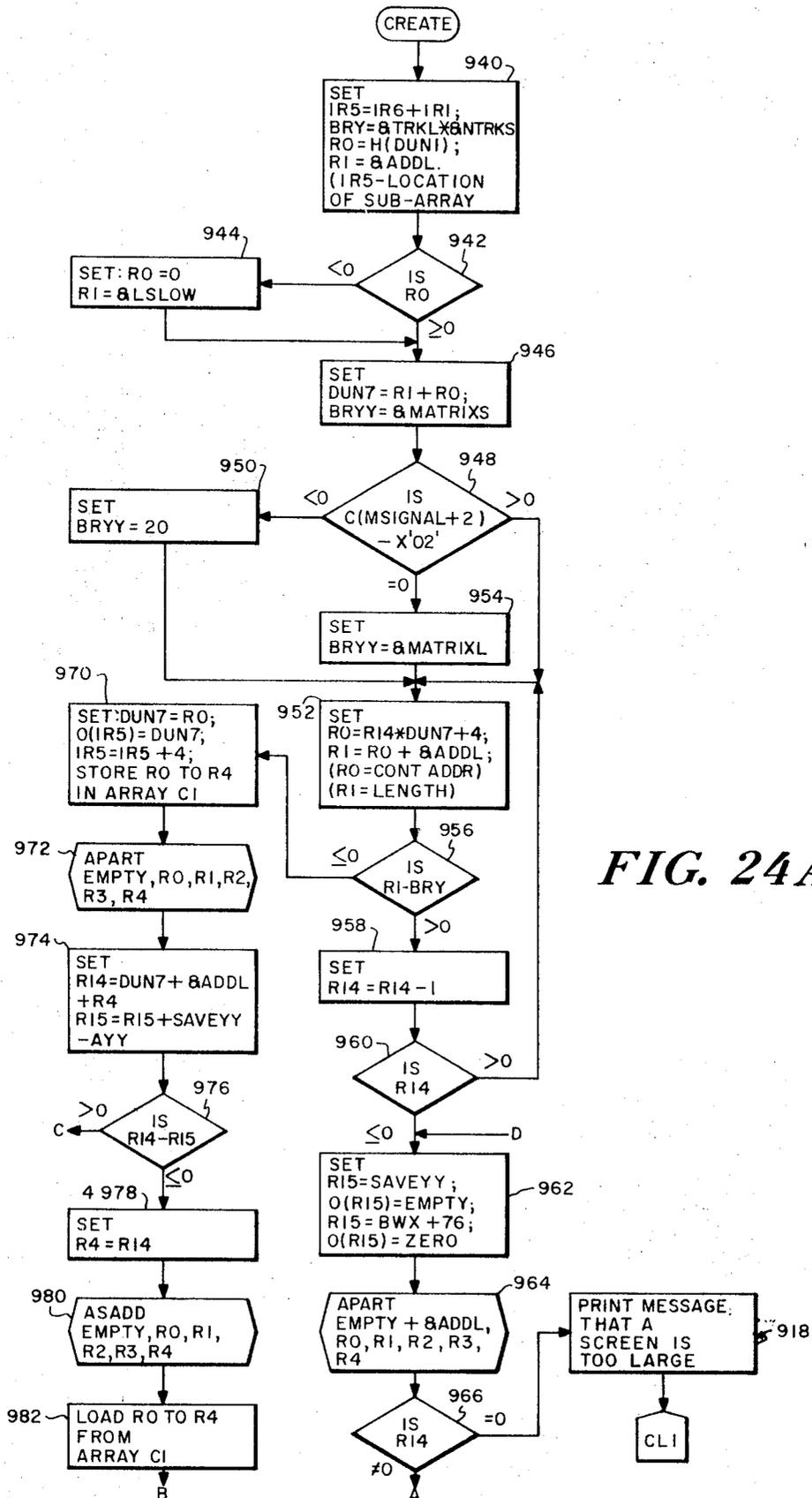


FIG. 24A

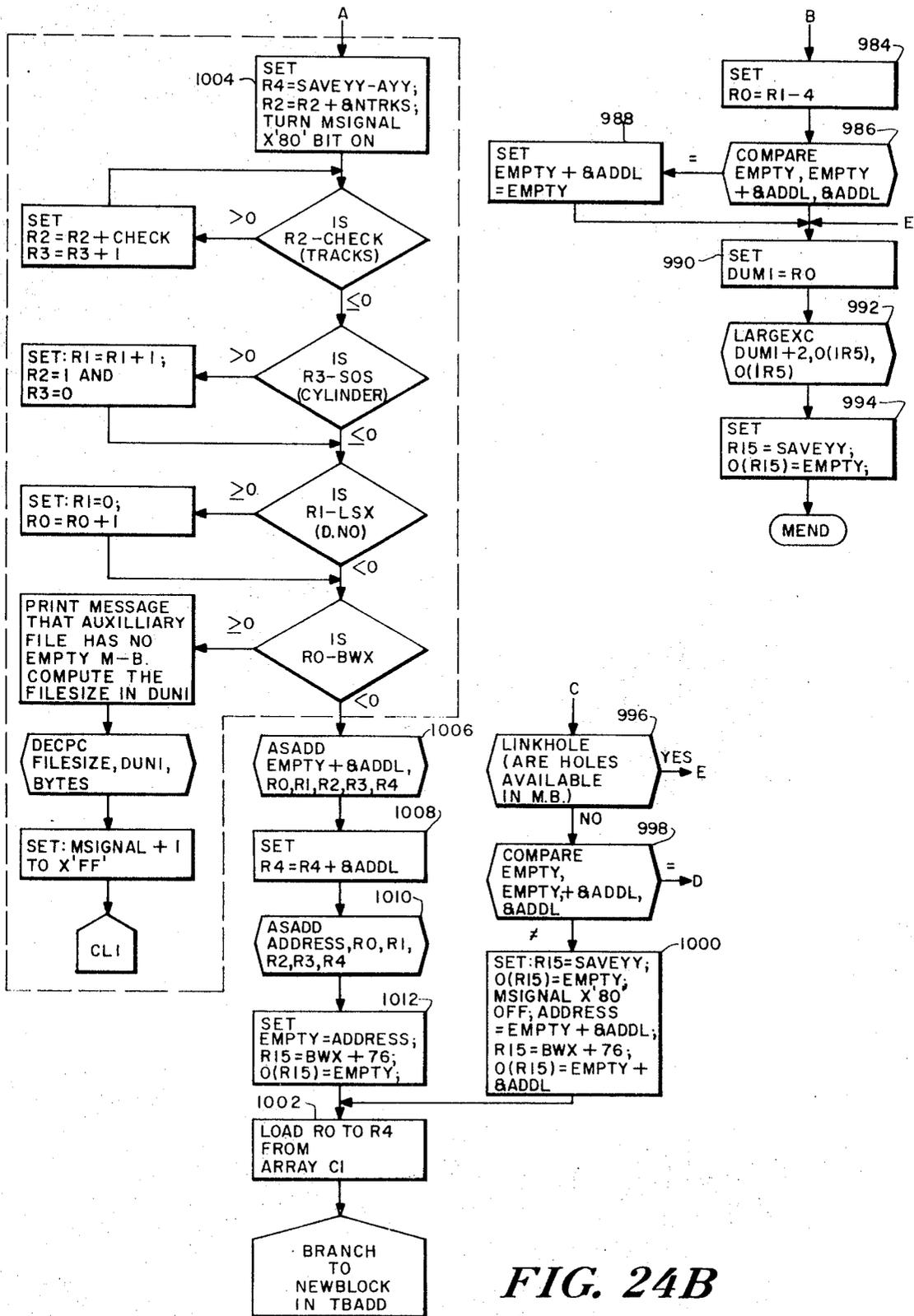


FIG. 24B

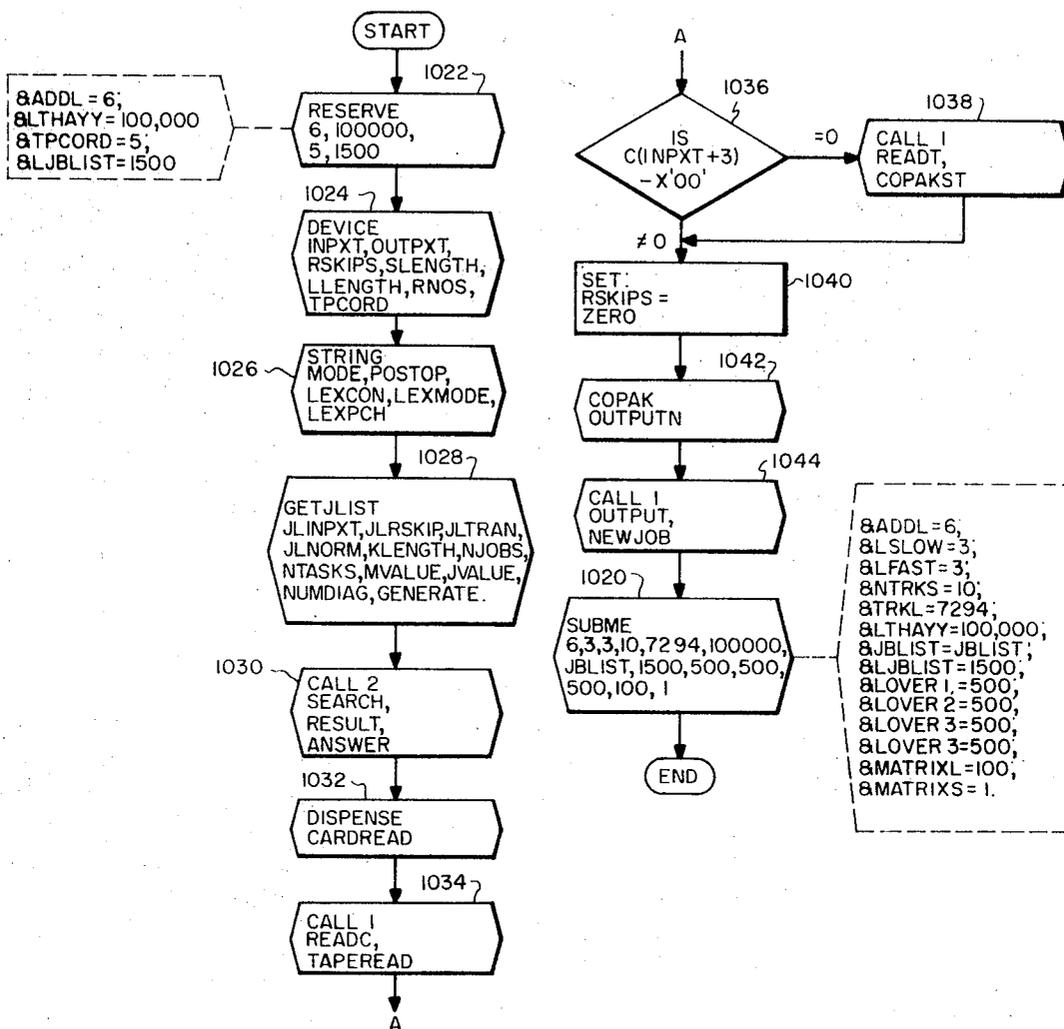


FIG. 25

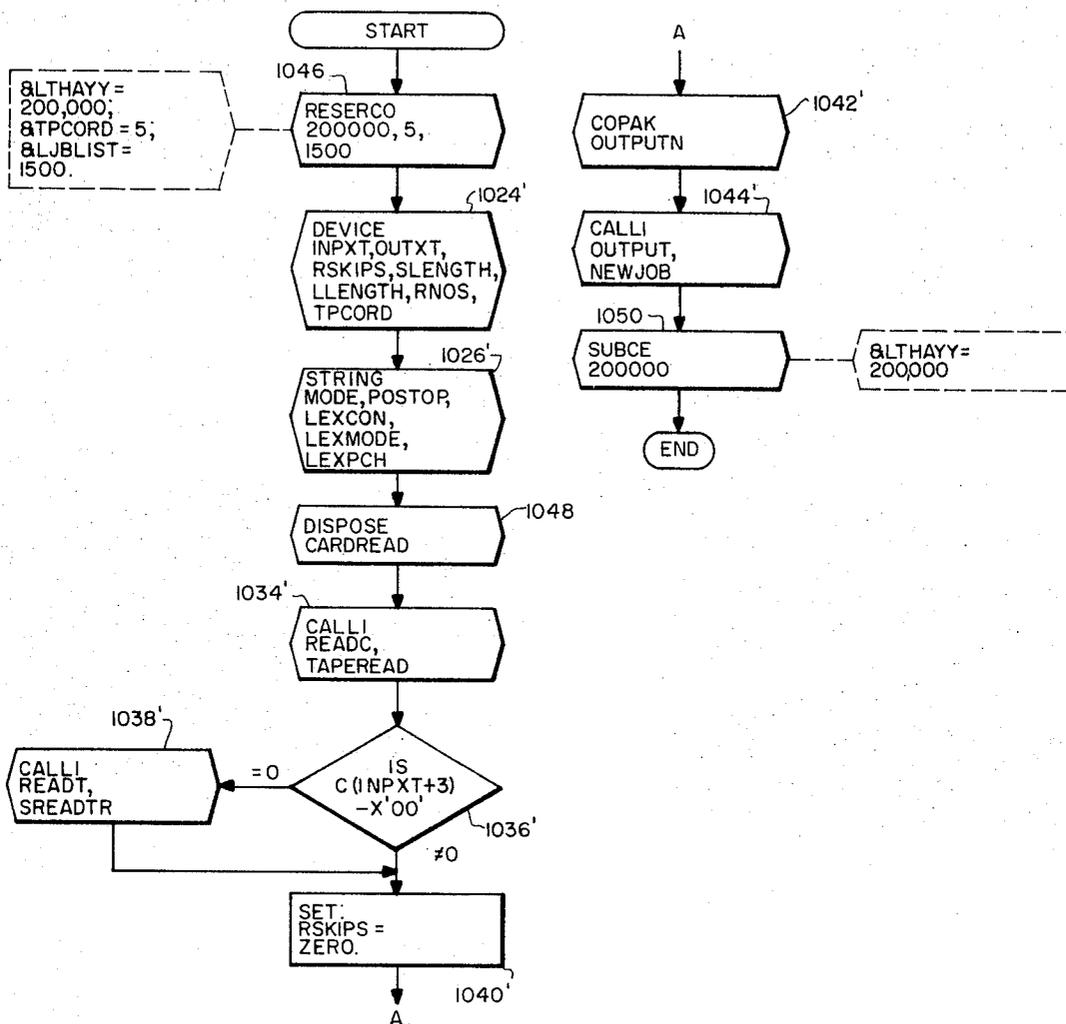


FIG. 26

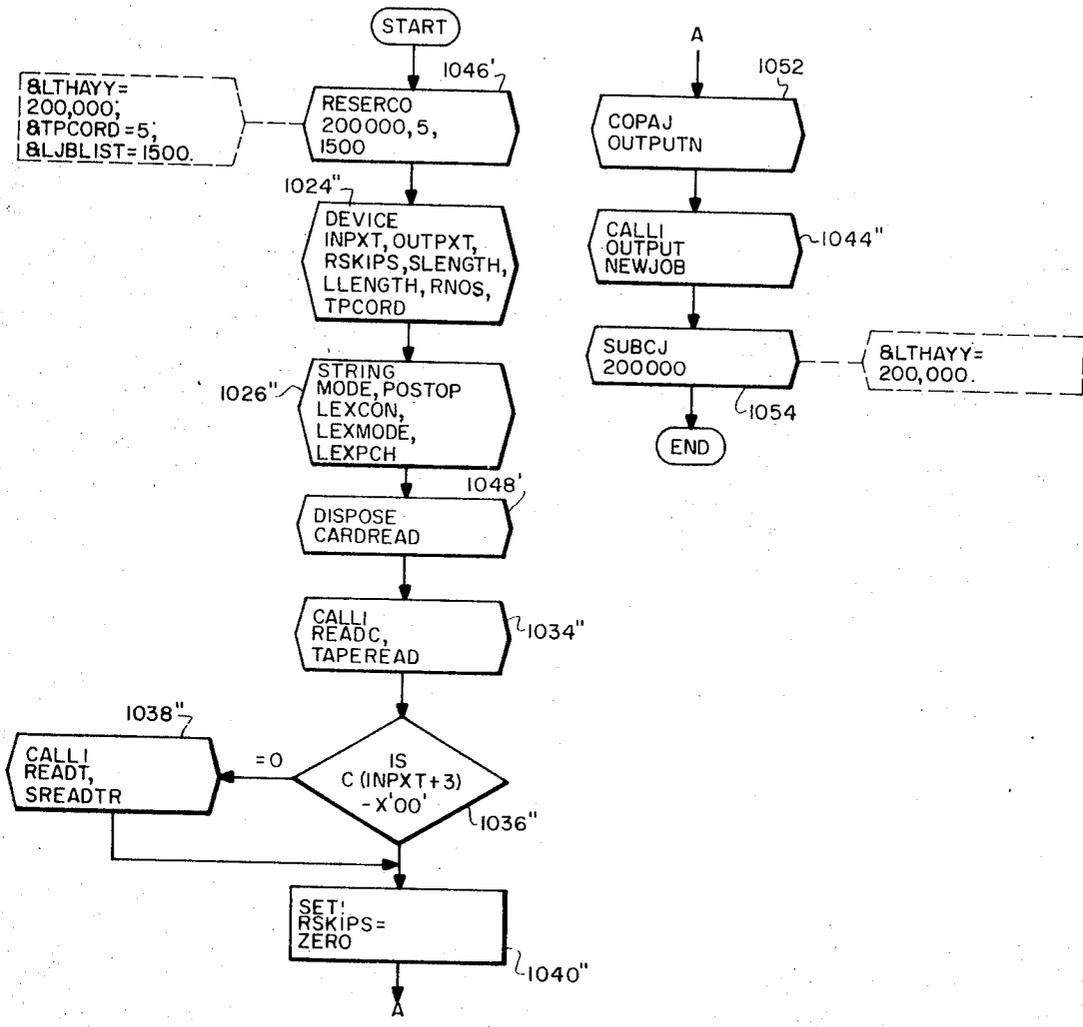


FIG. 27

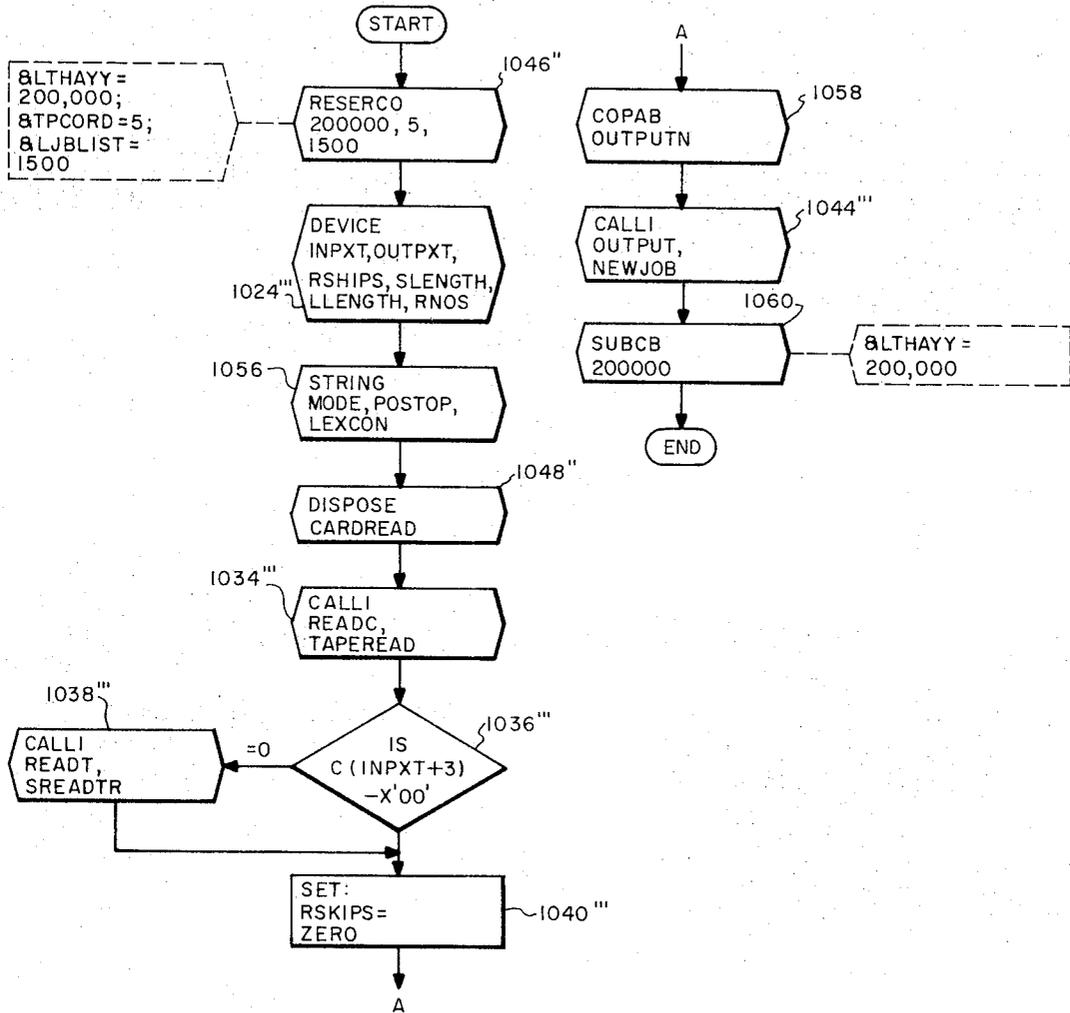


FIG. 28

DATA COMPRESSION AND DECOMPRESSION SYSTEM

TABLE OF CONTENTS

Abstract of the disclosure
 Background of the invention
 Summary of the invention
 Brief description of drawings
 Detailed description
 Preface
 A. Mathematical basis of the SOLID System
 B. OVERVIEW
 Structure of the file
 Description of Figure 12
 Description of Figure 13
SYSTEM PARAMETERS
 A. Structure of JOBLIST ITEM
 B. Address structure
 C. Computer organization of AUXILIARY FILE
 D. Definitions of parameters
 Design philosophy
 Design of the SOLID System
 Description of Figure 14
 Contents of MACROPAK
 Reserve
 Service Macros
 Components
 C. The CONTROL routine
 Description of Figure 15
 Description of MACRO-INSTRUCTION in MACROPAK
 A. Reserve
 Table 1 (System Parameter)
 Special Service-Macros
 "JUNK" Type-Instructions
 "RESERVE" Type-Instructions
 "SUBMP" Type-Instructions
 B. Service-Macro
 (a) General Service-Macro
 Table 2 (General Service Macro Instructions)
 1. Arithmetic Operations
 2. Calling Procedures
 3. Input/Output Calling Procedures
 (i) Card/Printer Operations
 (ii) Type Operations
 4. Extended IBM and Message Operations
 5. String Movement Instructions
 (b) Special Service/Macros
 1. Initializing Operations
 2. Input Operations
 3. Compressor
 Table 3 (Decompressor/Compressor Components)
 (i) Calling Procedures
 (ii) Alphanumeric
 Table 4 (Alphanumeric Special Service Macro)
 (iii) Numeric
 4. Translation Package
 5. Transformation Package
 6. Normalization Package
 7. Retrieval Package
 8. Global Memory
 9. Mobile Canonicalization Package
 C. Components
 Table 5 (Levels and Classes of Components)
 1. Initializing Components
 2. I/O Components
 (i) Basic Components
 (ii) Special Components
 3. Compressor
 4. Translation Package
 5. Transformation Package
 6. Normalization Package
 7. Retrieval Package
 8. Global Memory
 9. Mobile Canonicalization Package
RETRIEVAL PACKAGE
 Overview
AUXILIARY FILE
 Part A
 Part B
 Description of Figure 16
 Search Procedures
 Description of Figure 17
 Description of Figure 18
 Description of Figure 19
 Description of Figure 20
 Description of Figure 21
 Description of Figure 22
 Description of Figure 23
 Description of Figure 24
GLOBAL MEMORY
 Description of Figure 10
COPAK COMPRESSOR
 Definition and Commands
 Overview of the COPAK Compressor
 Description of Figure 11
SANPAKC (Introduction)
 Detailed Description
 Descriptions of Figures 1A, 1B and 1C
SANPAKD (Introduction)
 Detailed description
 Description of Figure 2
NUPAKC (Introduction)
 Description of Figure 3
 Description of Figure 9
 Detailed description
 Description of Figure 4
 Description of Figure 5
 Description of Figure 6
NUPAKD
 Description of Figure 8

Description of Figures 7a, 7b and 7c
CONTROL PROGRAMS
 A. Overlay Structure
 (i) IBM 360/40
 (ii) IBM 360/87
 B. CONTROL Program for the SOLID System
 Description of Figure 25
CONTROL PROGRAM COPAKCO
 Description of Figure 26
CONTROL PROGRAM COPAKAN
 Description of Figure 27
CONTROL PROGRAM COPAKNU
 Description of Figure 28
 Program listings of MACRO Instructions in alphabetical order
 Claims

15 A program for a general purpose digital computer for storing, retrieving or updating and purging a collection of items whose individual members have been assigned descriptor sets. The program comprises first translating the assigned descriptor sets into a special digital linear informational representation form. Within core storage of the digital computer a first index column array (MARRAY) is provided. MARRAY consists of subarrays each having EXECUTIVE POINTERS. The EXECUTIVE POINTERS in the index array contains only an address of length &ADDLY bytes. The EXECUTIVE POINTERS are arranged in the MARRY subarray seriatim in accordance with the associated M value and the addresses contain the beginning address of a JARRAY subarray.

20 A plurality of subarray JARRAYS are provided in core storage which JARRAYS each contain EXECUTIVE POINTERS including the next descriptor in the JOB-LIST ITEM and the address of the next subarray to be checked. The EXECUTIVE POINTERS in the column JARRAY subarrays are arranged seriatim in accordance with the value of the JARRAY EXECUTIVE POINTERS descriptors.

25 The address portion of the JARRAY EXECUTIVE POINTER points a subarray in a memory block resident in core. Said memory block subarrays are similar to said JARRAY subarray.

30 The final column arrays (RFILE) in the memory block have addresses which point to the address in the bulk storage of the digital computer where the collection of items may be stored, retrieved, updated or purged.

35 At the end of each subarray in the MARRAY, JARRAY and RFILE subarrays, a link address is provided where the search may be continued when a particular column subarray is filled. Core storage also maintains composite addresses, namely EMPTY which gives the exact location in core where the next newly created memory block is to be stored, with the fast memory address (FMADD of EMPTY) containing the beginning address of a transient portion of the core storage array or the beginning of the memory block; CURRENT the address in core storage where the memory block normally resides, with the fast memory address portion of CURRENT (FMADD of CURRENT) being the relative address of the first byte of the resident memory block that is not part of an existing subarray, and; ADDRESS which is an address extracted from a subarray during a search and which may be a link address pointing to a continuance of the subarray; an address extracted from an EXECUTIVE POINTER, or zero if

40 the subpath is missing. Additionally, there are provided in storage various indicator elements of which one is called MSIGNAL which will give a continuous updated indication of the status of the current search.

45 After initializing the system, the program searches each descriptor in the JOB-LIST ITEM starting in the MARRAY and continuing in the JARRAY through the memory block until, in retrieval, one finds in RFILE an address or addresses in bulk storage where the information to be retrieved can be found or, in storage, an address where information can be stored. Provision is made for the automatically updating storage, for eliminating certain descriptors by using overrides, and for transferring memory blocks back and forth into core with a minimum time loss.

50 The information in bulk storage is in compressed form and is decompressed only after having been retrieved or prior to

storage. The compressor and decompressor takes two forms; one is an alphanumeric compressor and decompressor (SANPAK) and the other is numeric compressor and decompressor (SNUPAK).

The compressor and decompressor can be either in the form of a program for a general purpose digital computer or may be a hard wired special computer.

The alphanumeric compressor operates to compress a string of digital signals by first scanning a segment of the string on a byte to byte basis. A table (LEXICON) is provided in the core storage of the digital computer. It has 256 byte positions. Each byte position corresponds to the 256 bit configurations possible in a single byte of information. In each of the appropriate byte positions of the table is stored the number of times particular bit configurations appear in the segment. Those bit configurations that do not appear in the segment are segregated as possible Type 1 code bytes. If there are Type 1 code bytes then the segment is scanned in multiple byte segments, byte by byte to determine if there are any common groupings of bytes for which a Type 1 code byte may be substituted. If a common multi-byte segment is found, a Type 1 code byte is substituted in their place in the string and the string is closed up. At the head of the string is placed the common multi-byte segment for which substitution has been effected, the Type 1 code byte substituted, and the number of times that the Type 1 code byte was substituted. This information is used upon decompression of the string. The common multi-byte segments (PCORDS) may also be kept in a special table called the PCORD TABLE and, where certain PCORDS are expected to be found in a given information, it may not be necessary to scan the string byte by byte with different common multi-byte segments, but the string may be scanned with one of the PCORDS from the PCORD TABLE so as to speed up the compression process. The PCORD TABLE is continuously being updated with the amount of saving achieved with particular PCORDS so that only those PCORDS which achieve a saving may be used in successive compression steps.

Additionally, where this type of compression has been utilized to its fullest, or where it cannot be used because there are no Type 1 codes available, Type 2 compression is effected. In this type of compression, where a particular byte appears more than 34 times in the first 256 bytes in the string, these common bytes are removed from the string and a bit map is placed at the head of the string to show where the bytes have been removed.

The numeric compressor compresses digital numeric strings by converting the strings of numeric information into integers, eliminating floating point exponents, differencing successive integers seriatim, placing at the head of the string a number indicating the number of differencing procedures, condensing identical sequences in the string and placing information at the head of the string showing the place where the identical sequences have been condensed and packing all of the substring integers into double words in an optimal fashion.

The search procedure and the compression techniques are all integrated into a single storage, retrieval and updating and purging system which has been called the SOLID SYSTEM.

BACKGROUND OF THE INVENTION

The invention is in the field of data storage and retrieval systems and particularly in the field of large scale systems of this nature. It also relates to data compression and decompression systems.

In designing a large computer oriented data storage and retrieval system it is desirable that the final product meet the following design and performance specifications:

- i. Storage, retrieval, updating and purging tasks must be accomplished as fast as possible.
- ii. The system must be independent of the information base.
- iii. Components should be capable of being coded independently of all the other components in the system.
- iv. Programming a particular modification of a fully implemented scheme or combining equivalent hardware com-

ponents to meet the varied needs of users, should be as simple as possible.

- v. The system should be open ended to provide for future modifications.
- vi. The coded system should be as free of machine dependence as possible to provide for easier translation to other computer configurations.

Because size and scope would prohibit writing such system as a single program or creating a one piece hardware embodiment, the system is organized on a component by component basis. Each component should perform a single task in the overall scheme. For example, one component can handle card input; another the output; and so on for each separate task the system will perform.

To simplify recoding of a large system for another computer, it is essential that a higher language be used or developed. The present higher languages (e.g., FORTRAL, ALGOL, COBOL, SNOBOL, COMMIT, etc.) are not suitable for coding large retrieval or indexing systems because they do not have the bit and byte manipulation capabilities that are essential for efficient machine coding. A large system should, therefore, have associated with it an open ended higher language, such as ALLOCATE, which can grow as the system is implemented. Thus, a fully implemented System can be coded in a machine independent higher language that can provide the basis for a retrieval language. The macro language provided in the IBM System 360 can provide a starting point for the language ALLOCATE.

Each component can be coded in the macro language. The central concept is thus one of extensively nested macros incorporated into the assembly language processor of the computer. In this way the normal operations of the assembly language are extended with macro instructions that perform the special operations needed. In the IBM 360 System the assembly language is called BAL. A programmer can add, delete, or ignore certain components of the system to suit specific needs. This design allows for the unrestricted growth of the system and the retrieval language (ALLOCATE) by adding new components to the system macro library.

Translation of a System defined in this manner to other computer configurations can be greatly simplified by the use of the component type system. For example, it is possible to translate directly to FORTRAN IV by a suitable translator. The translation can be performed component by component rather than by trying to rewrite an entire system. Moreover, since the components are independent of one another, only those components needed for the particular application of SOLID need be translated. The necessity for programming around deleted components becomes unnecessary.

In regard to the data compression part of the invention, the need for effective compressors is obvious because it is always desirable to reduce the number of information indicia required to represent information of given content without affecting the information content. Special recoding techniques that save storage or transmission time, such as the "SQUOZE encoding" developed for the Share 709 System, and the PREST scheme for the IBM 7094 have been developed in the past and are in use but both are tied inseparably to the particular data base or to a particular hardware. It is desirable therefore to have a data compressing system which is completely independent of the data base so as to have unrestricted general purpose use and which in addition meets the following design objectives:

- i. By compression, increase the amount of information that can be stored in mass storage or on magnetic tapes and other peripheral devices.
- ii. Increase the rate of transmission either from "slow" to "fast" memory or between receiver/transmitter stations by transferring compressed information.
- iii. Automatically decompress the compressed information when it is needed either by a computer (in fast-memory) or by users of the system.

- iv. Error checking procedures which will insure that errors in the transmitted compressed information will be found either before or during compression.
- v. Increase the efficiency of the computer system by decreasing the time required for search and/or fetch operations.

SUMMARY OF THE INVENTION

The invention is in a data management system for manipulating large amounts of information. In a particular form, the invention resides in a data storage and retrieval system which, once it has associated a set of descriptors to an item of information of any size, is independent of the actual data base of that item of information. That item of information can then be stored in bulk memory, in compressed form, and the set of descriptors associated with it can be stored, retrieved, updated or purged within a fixed time which is independent of: the number of sets of descriptors maintained by the invented system, the actual size of a particular set of descriptors, or the type of search, retrieval, updating or purging operation carried out. The advantage of this fixed time for manipulating descriptors derives from the fact that they are manipulated independently of the items of information in bulk storage, and that an efficient novel manner of manipulating sets of descriptors has been provided.

In particular, a very small portion of each set of descriptors is kept in the fast memory of a computer incorporated into the invented system such that only a small part of that fast memory is occupied even though the total storage space required for all sets of descriptors may exceed the fast memory capacity many times. The remaining portions of all sets of descriptors are organized in memory blocks of which one is at all times in fast memory but all others are kept in virtual memory. The system ensures, through the use of "continuance tables" which come into use before a search associated with a particular set of descriptors goes into a memory block, that the search will be completed within a single memory block. Thus, for any number of memory blocks needed for a particular great number of sets of descriptors, a search should involve a transfer of only one block from virtual memory to fast memory. The size of the bulk storage space occupied by the information with which the descriptor sets are associated thus also has no effect on the search speed.

Storage space in fast and in virtual memory is utilized efficiently in that the need for reserving specific blocks of memory for a particular use has been avoided. The invented system utilizes arrays and subarrays which have no fixed location and which can vary in size as needed in a manner not requiring the intervention of a user of the system, but controlled in an optimum manner by a system control package. Further, any available spaces within a memory block which have been vacated by purged sets of descriptors are used for creating new descriptor paths before attempting to locate previously unused memory space. The control package for overseeing the use of these vacated spaces operates by linking each vacated space to another such space to create a continuous chain, such that only the beginning of it need be kept track of. Once it is determined that a new descriptor set is to be stored in a particular memory block, the control package need only locate the start of this chain of vacated spaces and then insert appropriate descriptor information in the first available locations along the link which can take that information. The newly occupied spaces are then deleted from the link but the rest of the link, if any, closes again around the deleted spaces.

Storage space and retrieval time for the bulk storage information are optimized because, before entering bulk storage, the bulk information may be compressed into self-defining strings such that each string has associated with it all information needed for decompressing it. When compressed information is taken out of bulk storage, it can be decompressed without referring to any additional information associated with that particular string but stored elsewhere.

While the compressor-decompressor portion of the invention is of great importance to the efficiency of the data management system referred to above, it is also of great utility in any situation where data compression-decompression may be desirable, such as in communication between various combinations of peripheral devices, computer systems and subsystems and communication networks.

BRIEF DESCRIPTION OF THE DRAWINGS

FIGS. 1A, 1B and 1C illustrate a flow diagram of a macro SANPAKC used in alphanumeric compression.

FIG. 2 is a flow diagram of a macro SANPAKD used in alphanumeric decompression.

FIG. 3 is a generalized flow diagram of a macro SNUPAKC used in numeric compression.

FIG. 4 is a flow diagram of step *a* of the macro SNUPAKC of FIG. 3.

FIG. 5 is a flow diagram of step *b* of the macro SNUPAKC of FIG. 3.

FIG. 6A, 6B and 6C illustrate an expanded flow diagram of the macro SNUPAKC of FIG. 3.

FIGS. 7A, 7B and 7C illustrate a flow diagram of a macro SNUPAKD used in numeric decompression.

FIG. 8 shows the information at the head of a string prior to its being supplied to the decompressor SNUPAKD of FIG. 7.

FIG. 9 is a table showing input format code and its meaning with respect to the type of compression in the string.

FIG. 10 is a flow diagram of Part A of a macro SMEMORY which is in a GLOBAL MEMORY component of the invented system.

FIG. 11 is a simplified flow diagram of a macro COPAK for use in compression-decompression.

FIG. 12 is a diagrammatic showing of a typical array in storage.

FIG. 13 is a diagrammatic showing of the manner in which a JOB-LIST Item is searched in an AUXILIARY FILE.

FIG. 14 is a diagram illustrating the hierarchical arrangement in the design selected for the invented system.

FIG. 15 is a diagrammatic showing of CONTROL routines utilized in the invented system.

FIG. 16 is a flow diagram showing the status of information called MSIGNAL and used in the retrieval package of the invented system.

FIG. 17 is a flow diagram showing a macro AUXFILE, used in search procedure of a file called RFILE.

FIG. 18A and 18B are a flow diagram of a macro called SCREEN and used in search procedure.

FIGS. 19A and 19B are a flow diagram of a macro SUIPERSCH used in search procedure.

FIGS. 20A and 20B are a flow diagram of a macro called INSERT.

FIG. 21 is a flow diagram of a macro called MMATCH.

FIG. 22 is a flow diagram of a macro called STRATEGY.

FIG. 23 is a flow diagram of a macro called TBADD.

FIGS. 24A and 24B are a flow diagram of a macro called CREATE.

FIG. 25 is a flow diagram for a CONTROL package called SOLIDE.

FIG. 26 is a flow diagram for a CONTROL package called COPAKCO.

FIG. 27 is a flow diagram for a CONTROL package called COPAKAN.

FIG. 28 is a flow diagram for a CONTROL package called COPAKNU.

Preface

In order to facilitate initial orientation into this sophisticated and multifaceted invention, the detailed description begins with a brief exploration of the mathematical basis of associating sets of descriptors with items of information for the purpose of creating a versatile and flexible data management system. Under the heading following that, an illustrative

example is given of the invention as a data storage and retrieval system using a particular normalized form of these descriptors to access information items stored in large scale bulk storage. Once the cooperation between various portions of the invented system is indicated by means of the illustrative example, these portions are explained in great detail and particularity, with emphasis on their interrelation. The detailed description concludes with a portion devoted to a data compressing and decompressing system used in conjunction with the invented data management system. The data management system is referred to below as SOLID, and the data compressing and decompressing system is referred to as COPAK.

A. Mathematical Basis of the SOLID System

Suppose that a particular document has associated within it the following nine descriptors or designators:

A, B, C, D, E, F, G, H, and I.

To simplify matters, we shall suppose that there are no Type 2 over-rides. Perhaps, at this time, a simple description of the need for Type 2 over-rides should be presented. There are four codes for descriptors or designators which are reserved. These codes are as follows:

0, 1, 2 and 3.

The last three are called Type 1, Type 2, and Type 3 over-rides respectively. The 0 code for a designator indicates that there is no information at that point in the information representation.

In the retrieval mode, the Type 1 override code indicates that any non-zero designator at that particular point in the information representation is to be accepted for retrieval purposes. In the storage mode, that is when the information representation is being stored, the Type 1 over-ride code is used to create a new class with a 1 as a designator. It is thus possible to create a new non-specific class by substituting a 1 for one or more of the designators in an information representation.

The Type 2 over-rides are substantially similar to the Type 1 over-rides in the retrieval mode except that the Type 2 over-ride indicates that any zero or non-zero designator in the Type 2 over-ride position is acceptable.

When utilized in the storage mode, a Type 2 override code for a designator in an information representation means that the existing designator must be replaced by another designator which can be found in a separate table in storage (table AOVER2R).

The translation routine, used for an entire collection of documents, arranges the designators A, B, C, D, E, F, G, H, and I in the form of a label as follows:

$$\text{LABEL} = \begin{vmatrix} A & D & F \\ G & B & E \\ I & H & C \end{vmatrix} = (\text{ABC/DE/F//GH/I}) \dots (1)$$

Here the linear form is simply a shorthand way of writing the square array. / and // are inserted for clarity. LABEL is called an information representation (IR) of the particular designator-set. Its elements are divided into the following three categories or levels of disclosure.

Kernels: These lie on the principal diagonal (i.e., A, B, and C).

CI Connectors: Lie above the principal diagonal (i.e., D, E, and F).

CII Connectors: Lie below the principal diagonal (i.e., G, H, and I).

In an information representation, IR, the designators cannot be reclassified by an transformation. This means that IR's can be manipulated by any transformation rules that leave the designators in their assigned levels of disclosure. The mathematics of the transformation rules and normalization are shown and fully discussed in Appendices I and II of the book "Information Retrieval: a Critical View" in the article by P.A.D. deMaine and B. A. Marron, "The SOLID System I. A Method for Organizing and Searching Files." The book was

edited by Schecter and was published by the Thompson Book Company in Washington, D. C. in 1967.

The elements in an IR can be assigned numbers, from a look-up table for each level of disclosure, in the translation step. However, as was discussed previously, four numbers have been assigned special meanings, namely Type 1, Type 2 and Type 3 over-rides and, of course, the zero which means no information for a designator.

Information representations can be expanded by replacing a single kernel by a new IR. Contraction occurs when an information representation is replaced by an IR with fewer kernels. These properties of expansion and contraction permit the reclassification of documents without having to reorganize the file. Reclassification may be desirable because either the original designator-set did not adequately describe the referenced information, or, due to natural growth, new subclasses must be created. Thus these information representations permit the uninhibited growth of a retrieval system without incurring redundancy or obsolescence.

The status of an information representation with respect to expansion or contraction is indicated by the first bit-map (B_1) thus:

$$B_1 = (M/m_1, m_2, \dots, m_M/L/1_1, 1_2, \dots, 1_L) \dots (2)$$

Here M is the number of nested representations in the IR; m_1 is the number of kernels in the ith nested IR with the basic IR; L and 1_i refer to the Type 2 over-rides. The first bit-map for the above example, 1, is $B_1 = (1/3/0)$. Here we have assumed that the kernels are single information representations and that no Type 2 override codes are present. The second bit-map (B_2) is the binary projection of the linear form of LABEL. For example, if all nine designators in the LABEL are not zero, B_2 is:

$$\left. \begin{aligned} B_2 &= (111/11/1//11/1) \text{ binary} \\ &= (7/3/1//3/1) \text{ in decimal ten} \end{aligned} \right\} \dots (3)$$

The second bit-map (B_2) is constructed from the information representation in its square-array form. The linear form of LABEL is compressed by eliminating all zero designators. Terminal zeros in B_2 are omitted. For example, if E, G and I in (1) are zero, then:

$$\left. \begin{aligned} \text{LABEL} &= (\text{ABC/D/F//H}) \\ B_1 &= (1/3/0) \\ B_2 &= (7/2/1//1) \end{aligned} \right\} \dots (4)$$

It should be noted that the original information representation can be constructed from LABEL and B_2 .

The MOBILE CANONICALIZATION is used to transform the information representation and its bit-maps to any of its equivalent forms, i.e., another form for 4 is:

LABEL (BAC/DE//H); $B_1 = (1/3/0)$; $B_2 = (7/3/0//01)$. One of these equivalent forms (the NORMAL FORM) is unique for each and every designator set. This means that there is a unique "information path" associated with each and every descriptor-set, no matter what its source. Thus the SOLID System is independent of the information base. It is possible, to avoid normalization, if that is desired, and to use the unnormalized information representation. This situation occurs when the designators must remain in a particular order because of the nature of the information itself. This might occur, for example, where designators in the three classes (Kernels, CI, and CII Connectors) have been assigned the same codes, and their places in the information representation indicates the type of designator.

In the SOLID System the information in (4) is combined thus:

$$\text{JOB-LIST ITEM} = (1/3//\text{ABC}/2/\text{D}/1/\text{F//H}/\text{H}^*) \dots (5)$$

The first two numbers are the values for m ($=1$) and $J=m_1, m_2, m_3, \dots, m_M=3$. ABC is the principal diagonal of the information representation. The remaining numbers are the other diagonals of the second bit-map (B_2) and LABEL alternated. The asterisk (*), added to the last non-zero LABEL diagonal, indicates that the path is terminated. The field separators are inserted for clarity. They are not present in the machine

representation. For a general information representation which contains M nested representations and $J = m_1 m_2 m_3 \dots m_M$ kernels 5 is replaced by 6:

$$\text{JOB-LIST ITEM} = (m/J/LO_0/BD_1/LD_1/\dots/BD_{1-j}/LO_{1-j}^*) \quad (6)$$

Here BD_i is the decimal ten value of the binary-bit projection of the i^{th} diagonal of the IR, whose compressed form is LD_i (i.e., no zeros). The range of i can be expressed as follows: $(1-j, \leq i \leq (j-1))$. LD_{1-j}^* means that the compressed diagonal LD_{1-j} terminates the JOB-LIST ITEM. Diagonals of B_2 and LABEL are numbered beginning with the principal diagonal; through the diagonals with CI connectors; then through diagonals with CH connectors.

The diagonals of the second bit-map (B_2) are binary bit projections of the associated diagonals of the information representation. Each bit in the B_2 diagonal indicates the zero (bit off) or non-zero (bit on) status of a particular designator in the information representation. The actual machine method of representing the B_2 diagonals is based on the fact that the basic data-unit in the IBM 360 system, a byte, contains eight bits. Thus in a single byte of a B_2 diagonal the status of up to eight designators in the associated IR diagonal can be recorded in the SOLID System each B_2 diagonal is left adjusted to a byte boundary. This means that a particular B_2 diagonal begins at the left-most bit in a particular byte and continues, across byte boundaries, if necessary, until all the bits needed to indicate the zero or non-zero status of all the designators in the associated IR diagonal have been recorded. The used bits in the last byte of a particular B_2 diagonal will be left adjusted in the byte and the unused bits (of the original eight) are set to zero or turned-off. For example, if one takes the principal diagonal in (1), its B_2 diagonal would contain a single byte with the eight bit binary-number 11100000. This would be the decimal ten number 224. If both designators in the LD_1 diagonal of the IR (1) are not, zero, then its single byte BD_1 diagonal would contain the binary number 11000000 or the decimal ten number 192. It is understood that where the IR diagonal has more than eight designators it may be necessary to use two or more bytes in the B_2 diagonal, to record the zero and non-zero status of the designators. By using the above left adjusted method for B_2 diagonals, the binary representations will follow without the need for any additional calculation by the computer.

As an example, suppose that the elements in the linear form of LABEL are from an assigned descriptor set that has been rearranged to the square array form as follows:

$$\text{LABEL} = \begin{pmatrix} A & E & \theta & I \\ \theta & B & \theta & \theta \\ \theta & P & C & F \\ \theta & \theta & Q & D \end{pmatrix} = (ABCD/EF/I/PQ)$$

The first bit map is $B_1 = (1/4/0)$
The second bit map is $B_2 =$

$$= \begin{pmatrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix} = (240/160/0/128/96)$$

The second bit-map diagonals are binary bit projection of LABEL diagonals, left adjusted to a byte boundary. The principal B_2 diagonal, in this case 240, is not present in the JOBLIST ITEM.

The information in LABEL, B_1 , and B_2 are combined thus:

$$\text{JOB-LIST ITEM} = (1/4/ABCD/160/EF/0/128/I/96/PQ^*)$$

The asterisk added to diagonal PQ signifies this is the terminal link in the information path. The Translation Package produces the JOB-LIST ITEM as a self-defined string thus:

3	M	m_1	6	ABCD	3	160	4	EF	3	128	3	I	3	96	4	PQ*
---	---	-------	---	------	---	-----	---	----	---	-----	---	---	---	----	---	-----

Bytes 2 2 1 2 4 2 1 2 2 2 1 2 1 2 1 2 2

The first two bytes contain the length of the JOB-LIST ITEM. Each diagonal is preceded by two bytes which define its length. The compressed referenced information is stored in bulk storage at an automatically assigned location.

m_1 utilized only one byte. This means that a single representation cannot have more than 255 kernels. It is understood that the parent IR, which can have up to M representations nested in it, may have as many as M times 255 kernels. The two byte length of M limits its maximum value to 65,535. Thus, without considering the storage limitations, the parent IR can have a maximum of between 255 ($M=1$) and 14,211,415 ($M=65,535$) kernels. It is unlikely that these limits will ever be attained in applications of the SOLID System.

15 Overview - An Illustrative Configuration of the SOLID System In Data Storage and Retrieval

The JOBLIST items defined under the previous heading are utilized in a system such as that shown in block form in FIG.

16. Although the operation of the configuration of FIG. 16 is described in much greater detail later on in the specification, it is believed that a brief and qualitative preview of it at this time will help illuminate and unify understanding the operation, interrelation and cooperation of the various portions of the invention particularized below.

20 In one specific example in reference to FIG. 16, stages 600 and 602 initialize all appropriate registers in a computer system which is used in the embodiment of the invention and which has fast and virtual memories.

25 Then, a JOBLIST item, which may have been stored or created at stage 604, is used during the processing at stage 602. A portion of the JOBLIST item is examined at stage 606 where it may be determined that it is an index. Control then goes to stage 610.

30 In stage 610, a control procedure explained in much greater detail below carries out a search through an array in memory associated with indexes to locate information which may be associated with the particular index of the current search. If such information exists, it would be in the form of an EXECUTIVE POINTER. One of the following three situations may occur:

a. An EXECUTIVE POINTER is found and control goes to stage 616 which contains a control package which is called MMATCH and is explained in detail further in the specification;

b) An EXECUTIVE POINTER is not found in the searched array or possible extensions of it. If the system is in retrieval mode, the control package at stage 616 determines that further search procedures are unnecessary and aborts the search. If the search had been for a descriptor or screen in the JOBLIST item in a screen array (SCREEN SEARCH), and the JOBLIST item currently being used contained overrides, the control package MMATCH would call, through its macro called STRATEGY another control package called MOBILE CANONICALIZATION PACKAGE. The contents and function of the macros and packages mentioned above are defined in detail in the description below. If the system is in a storage or updating mode then steps which are explained in detail further in this specification are taken to create a new searching path for the JOBLIST item currently in use; and

c) If the memory array reserved for the EXECUTIVE POINTERS which are being searched is full then control again goes to stage 616 containing the control package MMATCH, but with an indication that more space for the array may be needed.

35 Stage 620 contains a control package called TBADD which assumes control as directed by portions of the MMATCH control package of stage 616. The control package TBADD which

is a complex and highly efficient control for determining if any transfer of memory blocks between fast and virtual memory may be required at a particular stage of using a JOBLIST ITEM. It should be noted that limiting all search procedures associated with one JOBLIST item to a single memory block is an important facet of the invention because that fixes the maximum time required for a search associated with a JOBLIST. One memory block is always in fast memory; a number of memory blocks may be in virtual memory. If a search is limited to a single memory block then only one block need be transferred from virtual to fast memory. A search thus will take the same maximum time whether there are two or N memory blocks in the system. Because of this provision the search time is considered independent of the file size.

If a new memory block is required, then the control package TBADD in Stage 620 transfers control to another control package called SMEMORY which saves, if necessary, the memory block which had been in fast memory up to this time and replaces it by a new one obtained from virtual memory. If only a combination of the memory space defined for a particular array is needed, the TBADD package transfers control to a control package called CREATE which performs the needed operations.

Once all operations associated with a portion of the JOBLIST ITEM currently being used have been completed, the control package TBADD transfers control to a stage 626 where it is determined if there are any more portions left of the JOBLIST ITEM. If there are, procedures similar to the one referred to above is repeated for each such portion, until a location is reached in a memory array called RFILE which is for addresses of information stored in bulk storage.

In addition to the INDEX SEARCH and the search through RFILE (AUXFILE) mentioned above, there are also operations involving search procedures using the JOBLIST ITEM between the INDEX SEARCH and AUXFILE called SCREEN SEARCH.

The function and operation of the control packages mentioned above, as well as the function and operation of portions thereof, has been explained in detail under the subsequent headings. Particular attention is directed to those individual steps or points in the control packages and portions thereof described below which relate to transferring or accepting control from another control package or a portion thereof.

STRUCTURE OF THE FILE

Storage is divided into the MAIN and AUXILIARY FILES. The MAIN FILE contains the referenced information which is stored in bulk storage (i.e., tapes, data-cells, etc.). The AUXILIARY FILE contains information paths which terminate in locations that contain addresses of the referenced information in the MAIN FILE. Job-List items, like those in (5) and (6), are used to trace, modify, or create information paths in the AUXILIARY FILE. The fully automatic COPAK compressor, discussed hereinafter, is used to substantially reduce the storage requirements of the MAIN FILE.

The AUXILIARY FILE consists of a maze of fully self-organizing column-arrays that are associated with an index (M), and screens (J , LD_0 , BD_1 , LD_1 , ...) that appear in the Job-List, (6). The column arrays are:

- a. A single column array, MA, which is associated with the number of nested representations (M). Here M_m is the maximum value of M in the system.
- b. M_m column arrays ($JA(I)$), with the range of I being between $1-I-M_m$. The M th of these [$JA(M)$] is associated with the J screen ($m_1, m_2, m_3, \dots, m_M$) for the M nested representations.
- c. With each element of column array $JA(M)$ there are associated two families of column arrays ($BD(I)$ and $LD(I)$). $BD(I)$ and $LD(I)$ are associated with the configurations of the I th diagonals, for B_2 and LABEL in the Job-List item (see 6).

For each B_2 , "I" begins with the first positive diagonal, proceeds through all positive diagonals and then all

negative diagonals, ending with the last non-zero diagonal. For each LABEL, "I" begins with the principal diagonal then proceeds as for B_2 .

- d. The RECORD FILE (RFILE) contains the addresses in bulk storage of the referenced information.

Each column array has the structure shown in FIG. 12. The first four bytes if the array contains its length. This is followed by entries or elements, called executive pointers, and, in the last position, the link or continuance address for the array. The link address is $\&ADDL$ bytes long, and it contains the locations where the particular kind of column-array (viz MA, JA, or $LD(I)$, etc) is continued or extended. There are two kinds of elements or EXECUTIVE POINTERS. One kind, which is used exclusively in the MA array or its extensions, contains only an address of length $\&ADDL$ bytes. The second kind of EXECUTIVE POINTER is used in the arrays associated with screens. It contains a screen (e.g., J in 6) and a composite address of length $\&ADDL$, bytes. The System Parameter $\&ADDL$, whose value can be changed to meet storage requirements, will be discussed hereinafter.

If the column array, shown in FIG. 12, is for the index M , the elements or EXECUTIVE POINTERS are stored seriatum with respect to the associated M value, and the addresses contain the beginning address of a JA array which is associated with the particular M value. Since the EXECUTIVE POINTERS are arranged seriatum, one need only go to position (M times $\&ADDL$ plus four) in the MA array to find the address of the JA array that is associated with the particular M value.

At the end of the array in FIG. 12 is the $\&ADDL$ long continuance address, which contains the location of another array where the array is extended (or continued) for other values of, for example, M . This seriatum arrangement continues over as many arrays as are required to store the different EXECUTIVE POINTERS. Thus the system is totally expandable, as the number of different M values does not in anyway affect the system. It is possible to store as many EXECUTIVE POINTERS, each associated with a different M value, in the MA array(s) as the system is capable of storing.

All those arrays associated with screens differ from the MA, or index array, in that each EXECUTIVE POINTER in the arrays contains two parts. That is, the first element is a particular value of the particular screen (viz J , LD_0 , BD_1 , etc.) that is associated with the array in question. The second part contains

the address of the array that is associated with the next screen where the search should continue. Further, within each array, the EXECUTIVE POINTERS are arranged in numerical order with the lowest value for the screen in the first position. For example, if the screen J_6 is equal to 1, then the corresponding EXECUTIVE POINTER would be in the first position in array JA, and the address would contain the location of the array that is associated with screen LD_0 , where the search continues. It should be noted that the screen J cannot be zero and that the search is aborted if it is. It should also be noted that all the elements or EXECUTIVE POINTERS in a particular array (shown in FIG. 12) are the same length (=screen address length). The length of a principal array, that is the total length of all EXECUTIVE POINTERS (=screen $\&$ address) between the initial four bytes, which indicates the array length, and the link address is a variable System Parameter ($\&MATRIXL$), that is discussed hereinafter. Of course, each screen array has a link address for an extension array, as discussed above. Also the EXECUTIVE POINTERS are arranged in numerical order over an array and all of its extensions.

Now column-arrays are created when they are needed for inserting the missing links or sub-paths (i.e., the elements in column-arrays) that will define a new "information path." The length of each newly created array is determined by the value of $\&MATRIXL$.

Each element in the column arrays is called an EXECUTIVE POINTER which contains the beginning address of another column array. Elements in arrays that are associated with the diagonals B_2 and LABEL contain a screen (VIZ. J , LD_0 , etc.), in addition to the address. Within each column

array and its extensions the elements are ordered according to the numerical value of the associated screen (for diagonals J , LD_0 , etc.) or the index M . The element in the column array that is associated with the last non-zero diagonal in the JOB-
LIST item (see (5) and (6)) contains a screen (i.e., LD_{1-j} *) and the address of a sub-array of RFILE. The sub-array of RFILE contains the address in bulk storage of the compressed
referenced information.

The values of M , J , LD_0 —(see (6)) are used to trace an “information path” through the maze of arrays of RFILE. For fixed values of M , each different configuration for each diagonal is entered only once and then only if it occurs. Since arrays are created in unoccupied storage areas only when they are needed, there is minimal movement of data. Moreover, because of the “paths” are essentially independent of each other, the time needed for a search is not altered by increasing or decreasing size of the file. Also, because only non-duplicate subpaths are stored, the AUXILIARY FILE will require substantially less storage than conventional files require. Further, because the entire search is accomplished in core, the search is extremely fast in providing the exact address in bulk storage where the desired compressed referenced information is stored.

FIG. 13 is a diagrammatic showing of the manner in which a JOB-LIST item is used to trace an “information path” in the AUXILIARY FILE. The value of M in the JOB-LIST item locates the EXECUTIVE POINTER in the array MA which points to the particular screen array $JA(M)$ associated with that value of M . In the screen array $JA(M)$, the screen J (as found in the JOB-LIST item) is used to locate a second EXECUTIVE POINTER which will point to the particular screen array, $LD(\phi)$, that is associate with the JOB-LIST entries J and M . In the array $LD(\phi)$, the screen LD is used to locate an EXECUTIVE POINTER which points to the next array, $BD(1)$. It should be noted that in this case, in searching through the first column array or $LD(\phi)$, there was no EXECUTIVE POINTER found for the value of screen LD in the JOB-LIST item. However, when one came to the bottom of the first column array of $LD(\phi)$ a link address was given, indicating that the search should be continued in the first extension column array of $LD(\phi)$. In this extended column array of $LD(\phi)$ there was an EXECUTIVE POINTER pointing to the next column array, $BD(1)$, which contained the screen LD_0 . In $BD(1)$, the next EXECUTIVE POINTER with BD_1 as a screen was found. This continues until, finally, the array $LB(1)$ is found where the screen LD_1 * indicates that the JOB-LIST ITEM is ended and the EXECUTIVE POINTER then points to a place in RFILE where the address(es) in bulk storage of the referenced information can be found. These address(es) in bulk storage are used to fetch the referenced information that was requested by the JOB-FILE item. It should be noted that in the search to reach an address in RFILE, one has been working solely with core storage information. Once the RFILE address has been located, the time necessary to obtain the particular information will be determined by the characteristics of the device on which the bulk storage information is stored. For example, if the bulk storage information is on a disk, the time it takes to bring the referenced information into core storage will include the disk access and transfer times. It should be understood that, there is no need to search the MAIN FILE because the bulk storage address(es), which are found in the RFILE array(s), are the exact location(s) of the requested information. It is also understood that the information in the bulk storage or MAIN FILE is, within the context of this system, stored in compressed form, and that it will be decompressed by the COPAK decompressor in the core-storage. The COPAK compressor will be discussed hereinafter.

Retrieval operations in the SOLID System are illustrated in FIG. 13 and they have just been described. It should be noted that if, during the search of the AUXILIARY FILE, no EXECUTIVE POINTER can be found in a particular array, then this means that the requested referenced information is not in

the MAIN FILE or the bulk storage. If this situation occurs in the retrieval mode then the search is discontinued and the user is advised that the information is not in the system. If this occurs in the storage mode, then a new subpath is created by inserting new EXECUTIVE POINTERS in those arrays which do not have them. This creation of a new information path continues through the AUXILIARY FILE until the RFILE is reached and the new bulk storage address has been allocated of the compressed referenced information. Thus, later in the retrieval mode, the same JOBLIST item will trace the newly created information path and locate the new referenced information. Thus, the system expands, by itself, independently of the user. Further, it should be noted that the new items are stored without in any way effecting any of the other information previously stored in the system. Thus, the system can automatically expand until all the allocated storage has been used. Moreover, there is no duplication of information in the AUXILIARY FILE because, in the storage mode, EXECUTIVE POINTERS are added and new arrays are created only if subpaths, defined by addresses in EXECUTIVE POINTERS, cannot be found. The expansion of the arrays in the AUXILIARY FILE, which occurs when new information is stored, is entirely independent of the user. Storage areas for newly created or extended arrays are automatically allocated by the computer, and is not in any way controlled by the user. Thus, because all retrieval and storage operations are fully automatic, the user has no concern whatever with the actual machine structures of either the AUXILIARY or the MAIN FILES. The actual way in which the computer uses the AUXILIARY FILE will be described next. However, the computer method of organizing the AUXILIARY FILE, which is fully described hereinafter, must first be briefly discussed.

In the computer, the AUXILIARY FILE is divided into two parts. One part, which is permanently resident in core storage, contains the column arrays that are associated with the index M and the screen $J(=m_1, m_2-m_1)$. This part is generated or read from cards in the RESERVE macro-instruction when the SOLID System is initialized. It is automatically generated when the system is used for the first time. The Second part of the AUXILIARY FILE contains all those column arrays associated with the remainder of the screens in the JOB-LIST ITEM (i.e., J , LD_0 —, BD_{1-j} , LD_{1-j} *) and the bulk storage address. This part is divided into memory blocks which are stored on disks in the virtual memory. They are transferred to core storage by the Global Memory component (SMEMORY) whenever they are needed. The size of the memory blocks determines the efficiency of the SOLID System, because as the memory block size increases the average search time decreases, since the memory blocks will be transferred less frequently. Each information path is restricted to a single memory block.

In retrieval operations the “continuance tables” will be used by the permanently resident part of the AUXILIARY FILE to select the memory block that might contain the request path, described by a JOB-LIST ITEM. The Global Memory Component, SMEMORY, decides either that the selected memory-block is already core-resident, in which case it is not transferred, or transfers it from virtual memory to core. Thus, a maximum of one memory-block is transferred for each JOBLIST item or request. In storage operations (i.e. “continuance tables” will be used to ensure that no “information path” will extend over more than one memory-block, which guarantees that a maximum of one memory block will be transferred for each request. One System Parameter (<HAYY), which is described hereinafter, sets the memory-block size. It should be understood that in normal applications of the SOLID System, the memory-block size will be large enough so that there will be a high probability that many requests can be answered from a resident memory-block and, consequently, there will infrequent transfers of memory-blocks from the virtual memory (i.e.. disks) to core. During the storage cycle, if the particular storage path crosses more than one memory block, a program can be used to transfer

that path to a single memory block so as to avoid the problem of crossing memory blocks during a retrieval cycle.

The system contains many composite addresses that are used during storage and retrieval operations to insure that memory blocks are correctly positioned either in core or in storage, and for enabling the machine to know where information is to be stored or retrieved at any particular time.

Composite addresses have two parts. The first part, which is called the slow memory address, specifies a location on a peripheral device like disks, drums, magnetic tape, data-cells, etc. It is used when memory-blocks or referenced information is to be transferred to or from core memory. The second part of the composite address is called the fast memory address, and it specifies a location in core-memory. Now there are two such composite addresses at the head of the permanently core-resident part of the AUXILIARY FILE, which contains the slow memory address in virtual memory (e.g. disks) where the next new memory-block can be stored. The first such composite address is called EMPTY. The slow memory address part of EMPTY will be used and updated when a new memory block is created in core-storage, and it is the location in virtual memory where the newly created memory-block can be stored by the Global Memory component (SMEMORY). The fast memory address in EMPTY is the location in core storage where the new memory block can be created. This location is always immediately after the permanently resident part of the AUXILIARY FILE.

The second composite address at the head of the permanently core-resident part of the AUXILIARY FILE is called BULK. It also has a slow memory address and a fast memory address part. The slow memory address part of BULK contains the bulk storage location where newly compressed referenced information can be stored. This part of BULK is used and updated in the storage mode when new "information paths" are created or another bulk storage address is added to an existing RFILE subarray. This occurs in the storage mode when new referenced information is added to the MAIN FILE. The fast memory part of BULK contains the location in core storage where the new uncompressed referenced information is found. In the SOLID System the address of the uncompressed information is located in the full word named LB-YYY. The COPAK compressor, which is executed after new bulk storage (or slow memory) addresses as assigned, compresses the information in the location specified by the fast memory address and transfers it to the bulk storage location specified by the recently assigned slow-memory-address.

During the initialization of the SOLID System, which occurs in the macro-instruction RESERVE, the permanently core-resident part of the AUXILIARY FILE is either generated by the macro-instruction MJARRY, or it is read from cards. This card-deck is punched by the Global-Memory component (SMEMORY) during the termination procedure. SMEMORY will be described later in this disclosure. During the initialization, the first two composite addresses, which precede the M-J arrays are loaded and the third composite address, which follows immediately after the M-J arrays, is made equal to the hexadecimal number FFFFFFFF. These three composite addresses, which at this point are in the principal data-array (YY), are transferred to the locations EMPTY, BULK, and CURRENT. Three other composite addresses (CORD1, (EMPTY+&ADDLY), and ADDRESS) also plays a significant role in the SOLID System. Two of these, (EMPTY+&ADDL) and CORD1, are used to ensure that machine or operation errors will not damage the AUXILIARY FILE. The last composite address, ADDRESS, is used to trace or create the information paths in the AUXILIARY FILE. The roles played by these six composite addresses (EMPTY, BULK, CURRENT, (EMPTY + &ADDL), ADDRESS, and CORD1) are described next.

The SSEARCH component used the information in the JOBLIST ITEM to trace (retrieval or storage) or create (new storage) information paths in the AUXILIARY FILE in core-storage. The composite address BULK is used at the end of

storage operation, when the RFILE array is reached, to assign bulk storage locations for new compressed referenced information. After each use BULK is updated to show the location of the next available space in the bulk storage. In the termination procedure, which is executed by SMEMORY, BULK is stored in its assigned location at the head of M-J arrays and the card deck for the next initialization of the SOLID System is punched. Of course, this new card-deck will also contain the new value of EMPTY.

For our present purpose we will suppose that a search is not discontinued because the sub-path cannot be found or created.

SSEARCH begins each task by setting (EMPTY +&ADDL) equal to the address EMPTY, and then it searches the MA array, which resides permanently in core, for the EXECUTIVE POINTER associated with the M value in the JOBLIST item. If a new sub-path is being executed then a new EXECUTIVE POINTER will be constructed from the address EMPTY and correctly inserted in the array. The address part of the located (or constructed) EXECUTIVE POINTER is placed in ADDRESS. In the next step, which occurs in the TBADD macro-instruction, the slow memory address parts of CURRENT and ADDRESS are used to determine whether or not the request address (ADDRESS) points to a location in core memory. The three alternatives are:

- i. If the slow memory address parts of CURRENT and ADDRESS are equal, then the fast memory address part of ADDRESS points to a location in the resident memory block. In this case a transfer between the core and virtual memory does not occur.
- ii. If the slow memory address part of ADDRESS is zero, then the fast memory address part points to a location in the permanently resident part of the AUXILIARY FILE. In this case, which occurs only if the located EXECUTIVE POINTER is in array MA, a transfer of memory-blocks does not occur.
- iii. If the slow memory address parts of CURRENT and ADDRESS are not equal and that for ADDRESS is not zero, then ADDRESS points to a memory block that is not resident in core. In this complex situation the Global Memory component uses the slow parts of addresses CURRENT and ADDRESS, and the MSIGNAL signal byte to decide what course of action should be taken. Full details of the various procedures that are executed by the RETRIEVAL and GLOBAL MEMORY PACKAGES are given hereinafter.

From the foregoing discussion it should be recognized that in the SOLID System the management and organization of both the AUXILIARY and the MAIN FILES is automatic in every respect. It should be further recognized that the system can be started anew or restarted when there is no information about memory-blocks in core storage and/or in the virtual memory. Moreover, fully automatic safety procedures, which are discussed hereinafter, protect the files from all machine and operator hazards. Finally, because storage is allocated automatically, the growth of the system is bound only by the total storage capacities of the machine that is being used. Thus, the entire system does not depend in any sense upon the amount of information that is stored, whether it is zero or billions on bytes.

SYSTEM PARAMETERS

There are 14 parameters in the SOLID System which must be set before the system is compiled. Properly selected values for these parameters insure that usage of core-storage and the performance of the SOLID System will be optimal. Nine of the 14 parameters can be reset, by recompiling the system, at any time. If the other five parameters (&ADDL, &LSLOW, &LFAST, &ENTRKS, and &TRKL) are reset, the AUXILIARY FILE must be regenerated from scratch.

Six parameters are used to define the eight principal data-arrays for the entire SOLID System. One of these

(<HAYY) is the amount of core-storage that can be used by both the AUXILIARY FILE and the COPAK Compressor. Two parameters name (&JBLIST) and define the length (&LJBLIST) of the data-array that is used for storing the JOBLIST ITEMS that are produced by the TRANSLATION PACKAGE from the descriptor-sets. &LJBLIST is also the length of a work-array (JBWORK). Three parameters (&LOVER1, &LOVER2, and &LOVER3) specify the lengths of the five arrays that are used to store information about the three over-ride codes (Type 1 (1); Type 2 (2); and Type 3 (3)).

Two variable parameters are associated with the COPAK Compressor One of these, <HAYY, which is also associated with the AUXILIARY FILE, has already been mentioned. The other parameter (&TPCORD) is used to optimize the performance of the alphanumeric component of the COPAK Compressor.

Eight parameters are associated with various aspects of the AUXILIARY-FILE. One of these (<HAYY) has already been mentioned. Four of the eight parameters (&ADDL, <HAYY, &NTRKS, and &TRKL) are concerned exclusively with the way in which the AUXILIARY FILE is transferred between core-storage and virtual memory (i.e. disks). The last four variable parameters (&LSLOW, &LFAST, &MATRIXL, and &MATRIXS) together determine the length of each newly created column array. Two parameters (&MATRIXL and &MATRIXS), and information about the over-ride codes, determine the way in which the arrays that are associated with screens are searched.

The roles played by these fourteen System Parameters are discussed next.

A. Structure of JOBLIST ITEMS.

In the SOLID System a TRANSLATOR component, which is called by the TRANSLATION PACKAGE, rearranges the previously assigned descriptor-sets to the particular linear form that is used to trace the "Information Paths" through the part of the AUXILIARY FILE in core-storage. A new TRANSLATOR component must be recoded for each new collection of items that are to be stored in the SOLID System.

The linear form (JOBLIST ITEM) for an information representation with M nested representations and

$$K \left(= \sum_{i=1}^M m_i \right)$$

Kernels is:

$$JOBLIST \text{ ITEM} = (M/m_1 m_2 \dots m_m / LD_0 / BD_1 / LD_1 / \dots / BD_{1-j} / LD_{1-j}^*) \cdot (7) \text{ In the computer 7 is expanded to 8.}$$

	LJBI	M	$m_1 m_2 \dots m_m$	LL ₀	LD ₀	LB ₁	BD ₁	LL ₁	LD ₁	LB ₂	BD ₂	LB _{1-j}	BD _{1-j}	LL _{1-j}	LD _{1-j} *	...
Bytes	2	2	M	2	(LL ₀ -2)	2	(LB ₁ -2)	2	(LL ₁ -2)	2	(LB ₂ -2)	2	(LB _{1-j} -2)	2	(LL _{1-j} -2)	

LJBI is the number of bytes in the JOBLIST ITEM (up to and including the last screen, LD_{1-j}*). Each screen is preceded by a half-word (two bytes) which contains its length plus two. For example (LL₀-2) is the length of screen LD₀.

The screen that is associated with the "zero" or principal diagonal of the second bit-map (B₂) is omitted from the computer representation, (8). Screens associated with terminal zero on empty diagonals in both the second bit-map (B₂) and LABEL (i.e. LD₀, LD_{1-j}, BD_{1-j}, LD_{1-j}) are omitted from the representation, 8. B₂ diagonals in 8 are left-adjusted to a full byte boundary. Thus the JOBLIST ITEM 5 should be written:

$$JOBLIST \text{ ITEM} = (1/3/ABC/128/D/128/F/64/H^*)$$

In the computer this representation is expanded to 9 thus:

	29	1	3	5	ABC	3	128	3	D	3	128	3	F	3	64	4	H*	...
Bytes	2	2	1	2	3	2	1	2	1	2	1	2	1	2	1	2	2	

In 9 there are 29 bytes in the JOBLIST ITEM. The first screen, J=3, is used to compute the rank

$$\left(K = \sum_{i=1}^M m_i = 3 \right)$$

of the information representation (IR). The first B₂ diagonal entered in 9 is the screen 128. This screen and the K value together indicate that the associated diagonal of the IR, whose screen is D, actually is Dφ. The asterisk in the last screen (H*) indicates termination of the information path. The terminal zero or empty screens associated with B₂ and LABEL have been omitted.

JOBLIST ITEMS like 9 are constructed from assigned descriptor-sets by the TRANSLATOR components in the data-array &JBLIST (=JBLIST), which is &LJBLIST bytes long. Random JOBLIST ITEMS can be generated for test purposes. The TRANSLATOR components also extract information about over-rides from the assigned or generated descriptor-sets and stores it in the five over-ride arrays (AOVER1, AOVER2, AOVER2R, AOVER3, AOVER3R), whose lengths are defined by the parameters &LOVER1, &LOVER2, and &LOVER3. The TRANSLATOR components can produce more than one JOBLIST ITEM. Moreover, the JOBLIST may be automatically changed by adding or deleting items during retrieval or updating of the AUXILIARY FILE. The information that is generated by the TRANSLATOR components (i.e. JOBLIST ITEMS, over-ride tables, etc) is used to trace (for retrieval), modify (for updating), create (for new storage), or purge "information paths" in the AUXILIARY FILE.

The variable parameters &JBLIST, &LJBLIST, &LOVER1, &LOVER2, and &LOVER3 that are associated with the TRANSLATION PACKAGE can be changed at any-time. However, in this case, the CONTROL routine and three components (SMEMORY, SRESULT, and SSEARCH) might have to be recompiled.

The structure of the composite address used in the SOLID System are discussed next.

B. Address Structure:

Three system parameters (&ADDL, &LSLOW, and &LFAST) sets the number of bytes in the addresses that are stored in the AUXILIARY FILE. The elements in the column-arrays in the AUXILIARY FILE are EXECUTIVE POINTERS, each of which contain a single address. In the "path" tracing procedure, the address part of an EXECUTIVE POINTER is the location of the next column array that is to be created (in a new storage or updating act) or searched (for retrieval). The elements of the RFILE sub-arrays, which are

the terminal location of the "information paths," contain the Bulk Storage Address (BSA) of the referenced information.

In the current version of the SOLID System the structure of the component addresses is:

D	DNO	TRK	CYLN	FMADD
---	-----	-----	------	-------

Bits: 4 4 6 10 24

Here:

D is a code (0 to 15) which specifies a particular type of peripheral storage device (viz., disk, magnetic tape, data-cell, drum, etc.)

DNO is a code (0 to 15) which specifies a particular device of type D.

TRK (0 to 63) is the track where the information begins. CYLN (0 to 1023) is the cylinder where the information begins.

(Note if D specifies magnetic tape then the record is stored in the two bytes occupied by TRK and CYLN).

FMADD is the beginning location in core-memory where the information will reside.

The lengths of the slow (D,DNO,TRK,CYLN) and fast (FMADD) parts of the composite addresses are set by the variable parameters &LSLOW and &LFAST respectively. The total length of the composite address is set by the parameter &ADDL. In one program the values of &LSLOW, &LFAST, and &ADDL, are 3, 3, and 6 bytes respectively. Assignments of the device type code (D) for the AUXILIARY FILE and RFILE address need not be the same. In one programs D=0 means a magnetic tape drive (for Bulk Storage Addresses) or an IBM 2314 disk drive (for AUXILIARY FILE Addresses).

Three service-macros (APART, ASADD, and COMPARE) disassemble, assemble, and compare addresses of the above type. If any of the above three System Parameters (&ADDL, &LSLOW, or &LFAST) are changed, these three macros must be recoded, and the AUXILIARY FILE must be regenerated. In one of the version of the SOLID System, one IBM 2314 disk pack is used to store the AUXILIARY FILE. This disk has been assigned D=0 and DNO=0. The compressed "referenced" or "bulk" information is stored on magnetic tapes which have been assigned D=0 and DNO = 0, 1,-, 15.

C. Computer Organization of the AUXILIARY FILE

In the SOLID System the AUXILIARY FILE is divided into two parts. One part, which is permanently resident in core-storage, contains the column arrays that are associated with the prime index M and the screens beginning with J(=m₁m₂-m_m) (see (8)). This part is generated by the service-macro MJARRAY, which is called by the initializing component SS-TATECL, when the SOLID System is used for the first time. Thereafter, it is read from cards by SSTATECL in the macro RESERVE at the start of each job-stream. A new card-deck is punched as the final act in each job-stream.

The second part of the AUXILIARY FILE contains all those column arrays that are associated with the remainder of the screens in the JOBLIST ITEMS (i.e., LD₀, BD₁, LD_{1,-}, LD_{1,j}*) and the Bulk Storage Addresses (BSA), which are assigned when they are needed. This part is divided into memory-blocks, which are stored on disks in virtual memory, and are transferred to core-storage by the Global Memory component (SMEMORY) whenever they are needed. The size of the memory-blocks determines the efficiency of the SOLID System. As the size of the memory-blocks increase, the average search-time decreases because the memory-blocks will be transferred less frequently. Complete information paths are restricted to a single memory-blocks. Thus, because a maximum of one memory-block is transferred per query, search-time is virtually independent of the AUXILIARY FILE size.

Each memory-block is prefaced by a composite address (of length (&ADDL) whose fast-memory address, FMADD (of length &LFAST), is its first unused byte. This information is used when new sub-paths are to be created. The slow-memory part of the composite addresses (of length &LSLOW) contains the beginning location in virtual memory where the memory-block will be stored. The first part of the AUXILIARY FILE, which resides permanently in core-storage, is prefaced by two composite addresses. The first of these is associated with the first unused byte in the last memory-block. This information is used to create new memory-blocks or to extend blocks that are full. The second composite address is the Bulk Storage Address that is used to assign slow-memory locations for new referenced information. Its fast-memory part, FMADD (of length &LFAST) contains the location in core-storage when the compressed referenced information will reside.

Three system parameters (<HAYY, &NTRKS, and &TRKL) together determine the size of every memory-block. &TRKL is the length of a single record in virtual memory. &NTRKS is the number of records of length &TRKL in each memory-block. <HAYY is the number of bytes in the principal data-array (YY), in core-storage. This array, YY, must contain the permanently resident part of the AUXILIARY FILE (i.e., arrays associated with M and J); a single memory-block; and approximately two strings of uncompressed referenced information. In one example of the program for the SOLID System &TRKL=7294; &NTRKS=10; and <HAYY=100,000. If these three parameters are changed once then the service-macros APART, ASADD and COMPARE must be changed, and the system started from scratch.

Two system parameters (&MATRIXL and &MATRIXS) determine the lengths of all newly-created column-arrays. These two parameters, which can be reset at any time are defined in the next section, D.

D. Definitions of Parameters

In this section, D, the fourteen system parameters of the SOLID System are defined.

&ADDL: The number of bytes in the composite addresses that are used in the AUXILIARY FILE.

&LSLOW: The length (in bytes) of the slow-memory part of the composite address.

&LFAST: The number of bytes in the fast-memory part of the composite address.

NOTE: &ADDL, &LSLOW, and &LFAST are associated with the service macros APART, ASADD, and COMPARE. These macros must be changed if &ADDL, &LSLOW or &LFAST is altered.

&JBLIST: (=JBLIST): The name of the array that is used for storing the JOBLIST ITEMS that are produced by the TRANSLATOR COMPONENTS (see Section A).

&LJBLIST: The number of bytes in the JOBLIST array, &JBLIST, and in the JOBLIST work-array, JBWORK.

&LOVER1: The length (in bytes) of the three principal over-ride arrays (AOVER1, AOVER2, and AOVER3) that are used for storing information about Type 1, Type 2, and Type 3, over-rides. Two additional over-ride arrays used by the TRANSLATOR components. AOVER1, AOVER2, and AOVER3 normally contain information about the specific locations (in the JOBLIST array, &JBLIST) of the designated types of over-ride codes.

&LOVER2: The length of the second, Type 2 over-ride array (AOVER2R). This array normally contains information for updating the "information paths" (in the AUXILIARY FILE) and the compressed referenced information (in the Bulk Storage).

&LOVER3: The number of bytes in the second Type 3 (or gate) over-ride array (AOVER3R), which normally contains the two gates for each Type 3 over-ride whose location is in array AOVER3. &LOVER3, should be about the same length as &LOVER2, and twice the length of &LOVER1.

<HAYY Designates the length of the principal data-array (YY) of the SOLID System. YY should be large enough to include the permanently resident portion of the AUXILIARY FILE, which holds the M-J arrays in about 1080 bytes; a memory-block (= &NTRKS* &TRKL bytes); and two strings of uncompressed reference information. In one program <HAYY is set equal to &NTRKS* &TRKL plus 20,000. In large applications of the SOLID System (viz., a national retrieval network) it is anticipated that the value of <HAYY will be set between 1,000,000 and 15,000,000.

&MATRIXL is the number of elements in the column-arrays that are associated with the screen of the principal diagonal of the information representations. In the JOBLIST ITEM 9 this is ABC.

&MATRIXS: The number of elements (e.g. EXECUTIVE POINTERS) in the column arrays that are associated with all screens other than the principal one. In JOBLIST ITEM (8) these are: $J(=m_1, m_2 - m_M)$, BD_1 , LD_1 , etc. At present &MATRIXS is also the number of Bulk Storage Addresses (BSA) in the RFILE sub-array which terminates the "information paths." However, the RFILE structure can be changed so that it can be accessed independently.

&NTRKS: The number of records in virtual memory that are in each memory-block (see Section C). In our program the record length (&TRKL) is set equal to the IBM 2314 disk track length (=7294 bytes). Thus &NTRKS is the number of tracks needed to store a single memory block.

&TPCORD: The number of permanent cords that are to be used by the alphanumeric compressors in the fast mode.

This is one of two parameters which together determine the optimum throughput rate for the COPAK compressor &TRKL: The number of bytes in each record in virtual memory. In one program &TRKL is set equal to the track length of the IBM 2314 disks (7294).

DESIGN PHILOSOPHY

In designing a large computer oriented system it is essential that design and performance specifications for the completed system be set up. The design goals for the SOLID System are given next.

- i. Storage, retrieval, updating, and purging tasks must be accomplished as fast as possible.
- ii. The system must be independent of the information base.
- iii. Components of the SOLID System should be capable of being coded independently of all the other components in the system.
- iv. Programming of the fully implemented scheme, to meet the varied needs of users, should be as simple as possible.
- v. The system should be open ended to provide for future innovations.
- vi. The coded system should be as free of machine dependence as possible to provide for easier translation to other computer configurations.

Because the size and scope prohibit writing the system as a single program, it was decided to write the system on a component by component basis. Each component performs a single task in the overall scheme. For example, one component handles card input; another the output; and so on for each separate task the system will perform.

To simplify recoding of a large system for another computer, it is essential that a higher language be used or developed. The present higher languages (e.g., FORTRAN, ALGOL, COBOL, SNOBOL, COMIT, etc.) are not suitable for coding large retrieval or indexing systems because they do not have the bit and byte manipulation capabilities that are essential for efficient machine coding. The SOLID System has associated with it an open ended higher language, ALLOCATE, which grows as the system is implemented. Thus, the fully implemented SOLID System will be coded in a machine independent higher language that can provide the basis for a retrieval language. The macro language provided in the IBM System 360 provides a starting point for the language ALLOCATE.

Each component of SOLID is coded in the macro language. The central concept is one of extensively nested macros incorporated into the assembly language processor of the computer. In this way the normal operations of the assembly language are extended with marco instructions that perform the special operations needed for SOLID. In the IBM 360 System the assembly language is called BAL. A programmer can add, delete, or ignore components of the system at any time. This design allows for the unrestricted growth of the system and the retrieval language (ALLOCATE) by adding new components to the system macro library.

Translation of the SOLID System to other computer configurations is greatly simplified by the use of the component type system. For example, it is possible to translate the SOLID System directly to FORTRAN IV by a suitable translator. The translation will be performed component by component rather than by trying to rewrite the entire system. Moreover, since the components are independent of one another, only those components needed for the particular application of SOLID need be translated. The necessity for programming around deleted components becomes unnecessary.

Design of the SOLID System

The design of the SOLID System is based on the concept of a system which contains two subsets of instructions. In one subset are all the assembly language instructions. The second subset, which is entered in the macro library (SOLID.MACLIB) of the System 360 Assembly Language Processor (ALP), contains all of the components of the SOLID System and certain selected service macros. Components are entered as macro subroutines with their own USING controls so that they may be placed anywhere in the system. Independently compiled components are stored in a partitioned data set, SOLID.LOAD.

Since both subsets of instructions are processed by the same compiler, the programmer can code in any arbitrarily selected combination of assembly language and macro instructions. In the remainder of this discussion the terms: "level of coding" or "coding level" refer to one such arbitrarily selected combination of instructions. At every coding level, instructions can be added, deleted, or over-riden. New Instructions can be programmed at a previously defined coding level. In its final form, the SOLID System, which will be coded at the highest level (in the language ALLOCATE), is independent of the machine used. As the coding level is lowered, the proportion of instructions needed from the first subset (assembly language) is increased and the programming becomes more difficult. The language ALLOCATE can have less than fifteen instructions, drawn from the two subsets mentioned above.

The two part design that has evolved for the SOLID System consists of the various components and service macros, MACROPAK, and a control routine (CONTROL). The control routine, which is coded in the evolving higher language ALLOCATE, assigns tasks to the various components of the SOLID System when a search, storage, compression or decompression job is executed.

The service macros in MACROPAK perform specialized tasks such as bit, byte and string manipulation which are necessary for an information system. Also included in this group are macros used for the input/output operations; the calling procedures for branching from the control routine (CONTROL); and other specialized macros. The components are coded in a hierarchical fashion with extensive nesting. The kind of hierarchical arrangement that has been achieved is illustrated in FIG. 14.

Each of the coding levels indicated in FIG. 14 designates an arbitrarily selected combination of assembly language (first subset) and macro (second subset) instructions. The pseudo-operations used in each of these arbitrarily selected coding levels are defined in MACROPAK and are themselves coded in mixtures of instructions from any of the lower coding levels. In a hierarchical scheme of this kind, the difficulty of coding the system is decreased as the coding level is raised. This occurs because both the proportion of assembly language instructions and the need for a specific knowledge of the contents of RESERVE are decreased.

The open ended, two part design just described permit optimum machine language coding while giving rise to a "machine independent" language that greatly simplifies the task of recoding the SOLID System for a new configuration. For the System 360 for example this design has naturally led to the full utilization of the System 360 macro language facility. In some configurations it may be necessary to extend the as-

sembly language processor so that those instructions that are not defined in the machine instructions set can be handled. Moreover, some or all of the macro instructions in the second subset may suggest ways in which existing hardware can be modified or they may influence the design of fourth generation 5 machines. In this connection the automatic multi-stage COPAK compressor can be realized in the form of a small fast computer with an equivalent hardware set.

Contents of MACROPAK

The macro-instructions in MACROPAK can be classified as follows:

- i. Reserve: There are twenty-two heavily nested macro-instructions which together contain the data declarations, system parameters, and status controls for the SOLID System and its stand-alone subsystems. Only two of these 22 instructions actually appear in the different CONTROL routines.
- ii. Service-Macros: macro-instructions which simulate useful but unavailable "instructions", including certain rather elaborate operations.
- iii. Components: useful macro-subroutines too large or complex to be viewed as single operations.

The 151 entries in one of the versions of MACROPAK are stored in SOLID.MACLIB, which is concatenated with the macro-library of the System 360 Assembly Language Processor (ALP). The contents and functions of entries in the three classes (Reserve, Service Macros, and Components) are described below:

Reserve:

The first and last executable instructions in each CONTROL routine is a "RESERVE" type and a "SUBMP" type of macro-instruction respectively. MACROPAK contains two "RESERVE" (RESERCO and RESERVE) and eight "-SUBMP" macro-instructions. Different combinations of these two kinds of instructions are used in the CONTROL routines for the SOLID System and its stand-alone subsystems. The fourteen System Parameters defined earlier, which are used to tailor the SOLID System to the machine configuration, are primarily associated with the "RESERVE" and "SUBMP" types of instructions.

The "RESERVE" type of macro-instruction is located immediately after the START instruction of the CONTROL routine. It establishes addressability; defines all global constants, counters, DCB or format statements, work arrays, and registers; and initializes the CONTROL routine. It also contains the instructions for opening and closing all those input/output devices that are used in the SOLID System for communication purposes and storing the referenced information. These various functions are performed by one of three "JUNK" type instructions (JUNK, JUNKC, JUNKR), which contain six of seven special macro-instructions that perform the identified special tasks (e.g. define constants). One of these special macro-instructions (DEVICES) calls the component OPENSHUT, which executes all opening and closing instructions, and the component SACTION, which is designed to execute all error-correcting procedures for the COPAK compressor. The CONTROL routine for the SOLID System is initialized in the component SSTATECL, which is called from the RESERVE macro-instruction. JUNK and JUNKC are slightly modified versions of JUNKR. JUNK is used to separately compile components see Components). JUNKC is used in RESERCO for the stand-alone subsystems.

The fully expanded "JUNK" type instruction contains more than 300 items whose significance must be understood if the SOLID System is to be coded entirely in assembly language. However, at the highest coding level (i.e., the language ALLOCATE) the significance of only the twenty-seven input/output commands need be fully understood.

The "SUBMP" type of instruction appears immediately before the END instruction of the CONTROL routines. Its func-

tion is to locate the literal pool; define the principal data array and specify their lengths; dummy addresses of unused components; and positions of the OPENSHUT component at compilation time. The principal data-array (YY) is specified in "-SUMBP" instructions. The two JOBLIST arrays (JBLIST and JBWORK) and five over-ride arrays (AOVER1, AOVER2, AOVER3, AOVER2R, and AOVER3R) are specified in the WORKAREA macro-instruction, which appears in certain of the "SUMBMP" type of instruction.

Service Macros:

In one version there are 98 entries in MACROPAK that are classified as part of the service macros.

For the most part, the macros classified under this heading performs special tasks and are extensively nested within components see Components). Among these entries are macros which will truncate a floating point number to fixed-point (TRUNC); convert a fixed point number to floating point (CONVE); move any string of information left (LMOVE) or right (RMOVE=RMVC) by a designated number of bytes; and several other macros which perform the specialized bit, byte, and string manipulative tasks needed for the SOLID System.

All the macros dealing directly with the input/output devices such as the card reader, card punch, printer, disks, and tape units are also classified as part of the service macros. The macros for reading and writing magnetic tape also perform the tasks of blocking or deblocking the information. At the highest coding level (i.e., the language ALLOCATE) card input, tape input and the entire output-operation(s) are single macro-instructions.

Other macros of special note here are nine macros which facilitate branching between the components.

Components:

Components are macro-instructions that contain their own USING controls for establishing addressability. They can be compiled in the control routine, or they may be separately compiled in named CSECTS. Compiled CSECTS are link-edited and stored in the partitioned data-set, SOLID.LOAD. The calling procedures are discussed herein after.

In one version there are 31 components of the SOLID System. The following three components initialize the system at the indicated times. OPENSHUT, which is called by the DEVICES macro-instruction in the "JUNK" type instructions, performs the tasks of opening and closing input/output devices at the beginning and end of each job-stream. SSTATECL, which is called by the RESERVE macro-instruction, initializes the CONTROL routines for the SOLID System at the beginning of each job-stream. SCOMMAND, which is called from the control routines, initializes the system at the beginning of each new job and before each use of the COPAK compressor.

Four of the components handle the input/output for the SOLID System (SJOBLIST, SREADC, SREADT, and SOUTPUT). These entries use the input/output service macros and another three components (SPRINT, SPUNXH, and SREID). Performance data for each job (component SRESULT) and for the COPAK Compressor (service macro SAVINGS in SANPAKC) are printed. In production runs this information would not be required.

There are six components which are associated with the COPAK Compressor. One of these (SACTION) which is called from the "JUNK" type instruction, is intended to process the error-correcting procedures for the several forms of the COPAK Compressor. Six service-macros handle the task of setting-up the strings of referenced information and setting the status controls for the three principal compressor components, (SANPAKC, SANPAKD, and SNUPAK). The remaining two components, (SNAPAKJ and SNUPAB) are variants of SANPAKD and SNUPAK that are used in the separate alphanumeric and numeric stand-alone compressors.

Eleven components are used by the TRANSLATION PACKAGE, which generates JOBLIST ITEMS in their normal forms. The supervisory component (SJOBLIST), which also handles input, uses three service-macros (JLITEM, TRANS-LATE, and NORMFORM) to call the random JOBLIST generator component (SGENITEM); the five translator components (STLATOR1, STLATOR2, STLATOR3, STRATOR4, and STLATOR5); and the normalization component (SNORMAL). The fully implemented SNORMAL component will use the three transformation rules, which are to be coded in components SCYCLIC, SREFLECT and SXCHANGE. These transformation rules will be used in the RETRIEVAL PACKAGE also. SGENITEM can be used to generate the random JOBLISTS that are needed to evaluate the performance of the SOLID System. The five translator components will actually perform the task of extracting (if necessary) and rearranging the descriptor-sets to the JOBLIST ITEM form. A new translator component must be coded for each new application of the SOLID System. There are provisions for incorporating up to 255 different translator components.

Five components are used by the RETRIEVAL PACKAGE, which duplicates the tracing, creating, purging and modification of "information paths" described earlier. One component (SRESULT), which is mentioned above, prints performance data for the RETRIEVAL PACKAGE. The supervisory component (SSEARCH) uses two service-macros (MMATCH and TBADD) to call the MOBILE CANONICALIZATION PACKAGE and the Global Memory component (SMEMORY). The Global Memory transfers the memory-blocks of the AUXILIARY FILE between core-storage and virtual-memory whenever they are needed. Two new components (SMATCH and SMOBILE) and the three transformation rules (SCYCLIC, SREFLECT and SXCHANGE) are used in the MOBILE CANONICALIZATION PACKAGE. The supervisory component of the RETRIEVAL PACKAGE, SSEARCH, has ten variable parameters which together designate the number of elements in each column array, and optimize the searches.

C. The CONTROL Routine.

The CONTROL routines are used to "thread" the retrieval, storage, updating, purging or compression problem through the various parts of the SOLID System. CONTROL is coded exclusively in the higher language ALLOCATE. In one version the CONTROL routine for the entire SOLID System contains sixteen statements. Five of these are ALP instructions. The other eleven instructions are taken from the second subset of macro-instructions. Thus the CONTROL routines can be easily changed to meet user needs by simply adding or deleting single instructions. This facility, and the fourteen System Parameters mentioned earlier permit the translation of the SOLID System to a particular machine configuration or application.

There are seven service macros which facilitate the branching among components. The three types of control that can be achieved by single macro-instructions are illustrated in FIG. 15. Five service macros (CALL1, CALL2, CALL3, CALL4, and CALL5) permit branching from the CONTROL routine to a component and the eventual return to any pre-assigned address in CONTROL. For example, the pseudo-operation; CALL1, SEARCH, RESEARCH passes control to the SSEARCH component and then returns it to the address RESEARCH in the CONTROL routine. The TRANSFER and GLOBAL service-macros permit branching between components.

The first and last executable statements in the CONTROL routines are the "RESERVE" and "SUBMP" type of macro-instruction respectively. Any allowed assembly language and service-macro instruction can be inserted between these two statements. The following program will read a record of com-

pressed information from magnetic tape and print it in hexadecimal format.

```

SOLIDIO      START      0
              RESERIO   600
              MVC       MODE(4),ZERO
              REIDT     LIST,JII
              LA        BRY,JII
              PRINT     B,LIST,O(BRY)
              B         CL1
LIST          DS        150F
              SUBIO     SOLIDIO
              END

```

The information is read into the array LIST, which can contain up to 600 bytes. JII contains the record length.

RESERIO and SUBIO are minor modifications of the RESERCO and SUBCE macro-instructions respectively.

DESCRIPTION OF MACRØ-INSTRUCTIONS IN MACROPAK

The two-part open-ended design that has evolved for the SOLID System consists of MACRØPAK, which contains the second subset of instructions, and a control routine (CONTROL). This control routine, which operates under the O/S system, assigns the specific tasks to the individual components of the SOLID System. It is easily changed to include new hardware or for special applications of parts of the SOLID System.

MACRØPAK, which is entered in the macro-library (SOLID.MACLIB) of the System 360 Assembly Language Processor (ALP), contains entries which are identified in the Reserve, Service-Macro, or Component classes (see above). The individual macro-instructions are described below. Listings of the macro-instructions are given in the Appendix.

In the following, the macro-instruction name (e.g., SKIPP) precedes the prototype statement thus: SKIPP - (&J SKIPP &LC). A blank left field in the prototype statement means that there is no location variable.

A. Reserve

Twenty-two entries in MACROPAK are identified in the "Reserve" category. There are three "JUNK", two "RESERVE", and eight "SUBMP" types of instructions. The roles played by these three kinds of macro-instructions has already been described.

Seven of the 22 entries can be viewed as special service-macros for the "JUNK" type instruction. Another two entries, ENTRANCE and WORKAREA can be regarded as special service-macros for the SUBMP type instructions. These nine "special service-macros" of the Reserve category perform tasks like defining global constants; beginning and terminating jobstreams; declaring formats; specifying relocatable addresses and entry points; designating registers, counters and gates; and defining work areas for the SOLID System.

Two of the three "JUNK" type instructions are used in the two "RESERVE" type instructions. The third "JUNK" type instruction is a special variant that is used to separately compile components of the SOLID System. Only one of the "RESERVE" type instructions, and one of the eight "SUBMP" type instructions appears in each CONTROL routine. Three components (OPENSHUT, SACTION and SS-TATECL) are associated exclusively with the "RESERVE" type of instruction. Two of these (OPENSHUT and SACTION) are called from one of the special service-macros for JUNK type instructions, DEVICES. The third component, SS-TATECL, is called only by the macro-instruction RESERVE. The components are described in Section C of this Chapter.

The fourteen System Parameters, defined elsewhere in this disclosure, are the variable parameters for the "RESERVE" and "SUBMP" type macro-instructions. The System Parameters are briefly described in Table I.

Table 1. Brief descriptions of System Parameters for the solid system. Definitions are given in the section on "System Parameters."

System Parameter	Description
&ADDL	Length of composite addresses.
&JBLIST &LFAST	Name of the JOBLIST array (JBLIST) Length of fast-memory part of composite addresses
&LJBLIST	Length of the &JBLIST array.
&LOVER1 &LOVER2	Length of the principal over-ride arrays Length of the second Type II over-ride array
&LOVER3	Length of the second Type III over-ride array.
&LSLOW	Length of slow-memory part of composite addresses
<HAYY &MATRIXL	Length of the principal data-array Number of entries in sub-arrays of the principal screen
&MATRIXS	Number of entries in sub-arrays of the secondary screens
&NTRKS	Number of seconds in each memory-block
&TPCORD	Number of permanent cords to be used in fast mode
&TRKL	Length of records in memory-blocks

The 22 macro-instructions in the Reserve category are described next.

Special Service-Macros:

1. CONSTANT-(CONSTANT)
Global constants and some error messages are defined in CONSTANT. This macro-instruction appears only in the three "JUNK" type instructions.
2. DEVICES-(&J DEVICES)
The DEVICES macro loads the second (BR3=11) and third (BR4=12) USING registers for the CONTROL routines. It calls the component OPENSHT, which executes the IBM OPEN and CLOSE system macro-instructions for the communication (e.g. Card Reader and Punch; Printer) and bulk storage (e.g. magnetic tapes) devices. DCB statements for these devices are given in the INOUT special service macro of the Reserve category (see below). DEVICES prints messages about the bulk-storage devices and it calls the components SACTION and SMEMORY. After executing the termination procedures (at the end of each job-stream) control is returned to the IBM Operating System O/S).
3. ENTRANCE-(ENTRANCE)
ENTRANCE is used in six of the eleven "SUBMP" type instructions to specify that the relocatable addresses for all components are in the main-stem of the designated CONTROL routines. These CONTROL routines are actually compiled in the so-called extended forms (see Section C). The remaining five CONTROL routines, whose "SUBMP" type instructions do not contain ENTRANCE, are executed with planned overlays.
4. INOUT-(INOUT)
The DCB or format statements for communication (e.g. Printer, Card Reader and Punch) and bulk storage (e.g. Magnetic Tapes) devices are specified in INOUT. The current allocation of resources is as follows:

DCB Name	Device	Assigned Purpose
MASTER PRINT	Card-Read Print	Read 80-columns of a card. Prints 132 bytes, preceded by the control character.
5 PUNCH TAPEIND	Card-Punch Tape Input tape containing uncompressed referenced information	Punches 1 columns on a card.
TAPEINC	Tape Input tape containing compressed referenced information.	
10 TAPEOTC	Tape Output tape for the COPAK Compressor.	
TAPEJB	Tape Input tape containing the descriptor-sets.	

15 If new DCB's are added, then the DEVICES macro must be changed also. Opening and closing operations, and DCB's for the Global Memory component, SMEMORY, are specified in the DCBMEM marco-instruction, which is executed in SMEMORY.

20 5. MODADI-(MODADI)
The relocatable addresses for the 31 components, five over-ride arrays, and two JOBLIST arrays (JBLIST and JBWORK) are specified in the MODADI macro-instruction by V-type addresses. The address of the principal data-array, YY, is also specified. MODADI is sued in the JUNIO, JUNKC and JUNKR macro-instructions. All new V-type and A-type addresses for the SOLID System must be specified in both MODADI and MODADX.

30 6. MODADX-(MODADX)
Except for the EXTRN declarations of all A-type addresses, the MODADX and MODADI macro-instructions are identical. MODADX is used only in the macro-instruction JUNK, which is used to separately compile the individual components as named CSECTS.

7. SAVEAREA-(SAVEAREA)
Global storage-areas for saving registers in the SOLID System are specified in the SAVEAREA macro-instruction. Some registers are also assigned names. The five COSAVEX (X=1,2,3,4 or 5) arrays are used by the TRANSFER calling instruction. If more than five levels are to be used, new arrays must be defined (see Section C).

8. STORAGE-(STORAGE &TPCORD,&LJBLIST)
The STORAGE macro-instruction allocates storage for the input/output commands, indicators, counters and gates; and for the composite addresses used by the RETRIEVAL PACKAGE. The permanent cords table (PCORDS), and other arrays used by the COPAK compressor are also allocated in STORAGE. The System Parameters &TPCORD and &LJBLIST are mentioned in Table 1, above.

9. WORKAREA-(WORKAREA &LOVER1,&LOVER2 ,&LOVER3,&LJBLIST)
The lengths of the five over-ride arrays (AOVER1,AOVER2,AOVER2R, AOVER3, and AOVER3R) and the two JOBLIST arrays (JBLIST and JBWORK) are assigned in WORKAREA, which is a special service-macro for the "SUBMP" type of instruction. These eight arrays have been assigned relocatable addresses (in MODADI and MODADX). The absolute machine address for the beginning of each array is found in the location specified in the MODADI and MODADX macro-instructions. For example the word AAOVER1 contains the absolute machine address of array AOVER1.

"JUNK" TYPE INSTRUCTIONS

70 Two of the four "JUNK" type instructions (JUNKC and JUNKR) are actually service-macros for the two "RESERVE" type instructions. The third, JUNK, is used to separately compile the components of the SOLID System for the overlay forms of the CONTROL routines. The two System Parameters that are associated with "JUNK" type instructions specify the

number of permanent cords (&TPCORD) and the length of the two-JOBLIST arrays (JBLIST and JBWORK), &LJBLIST. The values for these parameters are stored in locations PCGATE and LJBLIST in the STORAGE macro-instruction.

10. JUNK-(JUNK &TPCORD,&LJBLIST)

JUNK is used as a DSCET to separately compile the components of the SOLID System. It uses the MODADX macro-instruction.

11. JUNKC-(JUNKC &TPCORD,&LJBLIST)

JUNKC is used in the RESERCO macro-instruction, which appears in the six CONTROL routines for stand-alone operation of the COPAK compressor and its components.

12. JUNKR-(JUNKR &TPCORD,&LJBLIST)

JUNKR is the service macro for RESERVE. The two CONTROL routines for the SOLID System, SOLIDE and SOLIDO, contain the RESERVE macro-instruction.

"RESERVE" Type Instructions

The three "RESERVE" type instructions are the initializing instructions in different CONTROL routines. The four System Parameters associated with these instructions have been defined in Table 1. Each "RESERVE" type instruction executes the IBM SAVE instruction, which saves all registers, and establishes the addressability of the entire CONTROL routine with registers 10, 11 and 12. These registers have been named BR1, BR3 and BR4 respectively. A brief description of each "RESERVE" type instruction is given below:

13. RESERCO-(RESERCO <HAYY,&TPCORD,&LJBLIST)

RESERCO is the initializing macro-instruction for the six CONTROL routines for the stand-alone COPAK compressor and its components. The addresses SAVEYY, SBRY and SBRY are computed and the permanent cords table, PCRODS, is set to zero.

14. RESERVE-(RESERVE &ADDL,<HAYY,&TPCORD,&LJBLIST)

RESERVE initializes the CONTROL routines for the entire SOLID System (SOLIDE and SOLIDO). The actual initialization of registers, M-J arrays, and the permanent cords table (PCRODS) occurs in the component SSTATECL, which is called from RESERVE.

"SUBMP" Type Instructions

The "SUBMP" type instruction is the last executable statement in the CONTROL routines. Its function is to locate the literal pool; to dummy the relocatable addresses of unused components; to specify and position the eight principal arrays; and to position components compiled in the main stem. There are eight "SUBMP" type instructions in MACROPAK. Six of these (SUBCB, SUBCBO, SUBCE, SUBCO, SUBCJ, and SUBCJO) are associated with the six CONTROL routines for the stand alone compressors. Two (SUBME and SUBMO) are used in the CONTROL routines SOLIDE and SOLIDO respectively. A "SUBMP" type instruction ending with an 0 is used in the CONTROL routine that is to be executed as a planned overlay. "SUBMP" type instructions without the end 0 are used for the so-called extended forms of the CONTROL routines. (see below)

Thirteen of the 14 System Parameters are associated with the "SUBMP" type of instruction. These parameters have been defined in Table 1. The eleven "SUBMP" type instructions are described next.

15. SUBCB-(SUBCB <HAYY)

SUBCB and RESERCO are used in the CONTROL routine for the extended form of the stand-alone numeric compressor-decompressor (COPAKNU).

16. SUBCBO-(SUBCBO <HAYY)

SUBCBO is a variant of SUBCB which is used for the planned overlay form of the stand-alone numeric compressor-decompressor (COPAKNUO).

17. SUBCE-(SUBCE <HAYY)

SUBCE and RESERCO are used for the extended form of the stand alone combined compressor-decompressor, COPAKCO. This compressor-decompressor contains both the numeric and alphanumeric part.

18. SUBCJ-(SUBCJ <HAYY)

The extended form of the stand-alone alphanumeric compressor-decompressor, COPAKAN, uses the macro-instructions RESERCO and SUBCJ.

19. SUBCJO-(SUBCJO <HAYY)

SUBCJO is the special variant of SUBCJ that is used in the planned overlay form of the CONTROL routine COPAKAN, which is called COPAKANO.

20. SUBCO-(SUBCO <HAYY)

SUBCO is a special variant of SUBCE. It is used in the planned overlay CONTROL routine COPAKCOO.

21. SUBME-(SUBME &ADDL,&LSLOW,&LFAST,&NTRKS,&TRKL,<HAYY &JBLIST,&LJBLIST,&LOVER1,&LOVER2,&LOVER3 ,&MATRIXL,&MATRIXS)

SUBME and RESERVE are used in the extended form of the CONTROL routine for the entire SOLID System, SOLIDE. This CONTROL routine executes all 31 components. The 13 System-Parameters associated with SUBME have been defined in Table 1.

22. SUBMO-(SUBMO &LOVER1,&LOVER2,&LOVER3,&LJBLIST)

SUBMO is used for the overlay form of the CONTROL routine for the entire SOLID System, SOLIDO.

A "RESERVE" type macro-instruction is the second instruction in the control routine. A "SUBMP" type macro-instruction always precedes the the END statement.

B. Service-Macros

The 98 service-macros in the current version of MACROPAK can be identified as either General or Special Service-Macros. The 37 General Service-Macros are needed for most information processing systems. Thus they can be regarded as basic operations which are not in the Assembly Language Processor (ALP) instruction set. The 64 Special Service-Macros execute the special bit, byte and string manipulative operations used in the SOLID System.

In the following discussion an "address" means either a named location or a singly subscripted variable. The two register form of IBM addressing (i.e., D2(X2,B2)) is not allowed.

a. General Service-Macros

The thirty-seven General Service-Macros are identified in five classes (see Table 2). The General Service-Macros can be used anywhere in the SOLID System since all registers used by these macros are preserved. Seven macro-instructions, in Class 4, are obvious extensions of existing IBM 360 ALP instructions. The remaining 30 General Service Macros must be viewed as vital operations for the SOLID System. Twenty-one of these, in Classes 2 and 3, are calling procedures which are used to manage all I/O and transfers during execution. The last nine macro-instructions, in Classes 1 and 5 (see Table 3), perform vital arithmetic and string movement operations on data.

The hardware implementation of many of the General Service Macros, either singly or in special combinations, can very substantially increase the already very impressive performance of the SOLID System.

The 37 General Service Macros are discussed below.

TABLE 2.

Classes of the General Service Macro Instructions

Class	Macro Name	Macro No.	Class	Macro Name	Macro No.
1	CONVE	23		REID4	42
	DICE	24		REID5	43
	FACE	25		REID6	44
	TRUNC	26		REID7	45
2	CALL1	27		REID8	46
	CALL2	28	3(ii)	REIDJB	47
	CALL3	29		REIDT	48
	CALL4	30		WRITE	49
	CALL5	31			
	GLOBAL	32	4	CSSCRN	50
	STSR	33		DECPC	51
	TESTL	34		DUMADD	52
	TRANSFER	35		HEXPC	53
				LARGEXC	54
3(i)	PRINT	36		SKIPP	55
	PUNSH	37		TALE	56
	PUNXH	38	5	LMOVE	57
	REID1	39		RMOVE	58
	REID2	40		RMVC	59
	REID3	41			

1. Arithmetic Operations

23. CONVE-(&J CONVE &FROM,&TO)

The integer number in address &FROM is converted to a normalized (short) floating-point number and stored in address &TO. All registers are unchanged.

24. DICE-(&J DICE &NOBITS,&FROM,&TO)

A random integer number with the number of bits specified in &NOBITS is produced in the full-word location &RANDNO (right adjusted). &ODDNO contains the multiplicand, and it is updated after each use of DICE. If &ODDNO contains zero, a starting 32 bit odd number is constructed from the "Time of Day". Eleven global words (in STORAGE) are reserved exclusively for storing the random starting numbers. They are initialized to zero. These words are GENERATE, ODDNO, and ODDNOX (with X=1,2,-9). All registers are unchanged.

NOTE: DICE uses a high-speed multiplicative-type generator to set &RANDNO bit by bit. &NOBITS cannot equal zero or exceed 32. An error message is printed and all operations are terminated if &NOBITS contains zero.

25. FACE-(&J FACE &NFACES,&RANDNO,&ODDNO)

Face is a special variant of DICE which produces a random integer number (in &RANDNO) that lies in the range from 0 to that specified in &NFACES. For example, if &NFACES contains 3, &RANDNO will contain 0, 1 or 2. All registers are unchanged.

NOTE: If &NFACES contains zero an error message is printed and all operations are terminated.

26. TRUNC-(&J TRUNC &FROM,&TO)

The normalized (short) floating point address &FROM is truncated to an integer number and stored in address &TO. All registers are unchanged.

NOTE: If the truncated number contains more than eight hexadecimal digits all operations are terminated.

2. Calling Procedures

Special calling procedures are used in the SOLID System to branch between the CONTROL routine and the components or to branch among the components themselves. The three types of calling procedures have already been illustrated (FIG. 15). The Macro-instruction CALL1 (see below) is the first type. TRANSFER is the second type. GLOBAL is a special version of TRANSFER which is used exclusively for calling the Global Memory component, SMEMORY. CALL2,CALL3,CALL4, and CALL5 is the third type of calling procedure.

There are two macro-instructions (STSR and TESTL) which are associated with the seven special calling instructions (see Table 3, Class 2). TESTL is nested in the CALLX(X=1,...,5, and TRANSFER macro-instructions. The STSR instruction is the first executable instruction in each component. Together the TESTL and STSR macro-instructions permit the use of V-type relocatable addresses (in MODADI and MODADX) for all components of the SOLID System. With a CALLX(X=1,-5) instruction the USING and BRANCH registers are altered. No register is changed by using either the GLOBAL OR TRANSFER instructions.

In the following macro-instructions &NAME, &NAME1, etc. are the relocatable addresses (V-type) of the designated components. In the SOLID System the name of a component is its relocatable address preceded by an S(i.e., ANPAKC (relocatable address); SANPAKC (Name)). &ALINST is the return address in the CONTROL routine. &RETURN is the return address in the CONTROL routine or in the component in which the call is issued.

27. CALL1-(&J CALL1 &NAME,&ALINST)

After the branching to the component S&NAME (entry point &NAME) control is returned to location &ALINST in the CONTROL routine. The USING(&UR) and BRANCH(&RR) registers specified for component S&NAME are altered.

28. CALL2-(&J CALL2&NAME1,&NAME2,&ALINST)

Components S&NAME1 and S&NAME2 are executed before control is returned to location &ALINST in CONTROL.

29. CALL3-(&J CALL3 &NAME1,&NAME2,&NAME3,&ALINST)

The components with relocatable addresses &NAME1,&NAME2 and &NAME3 are executed before returning to the address &ALINST in the CONTROL routine.

30. CALL4-(&J CALL4 &NAME1,&NAME2,&NAME3,&NAME4,&ALINST)

The four components whose relocatable addresses are given in the instruction are executed before returning to address &ALINST in the CONTROL routine.

31. CALL5-(&J CALL5 &NAME1,&NAME2,&NAME3,&NAME4,&NAME5,&ALINST)

The indicated five components are executed before control is returned to address &ALINST.

32. GLOBAL-(&J GLOBAL &ADDL,&NTRKS,&RETURN)

This instruction can be used anywhere in the SOLID System to call the Global Memory component, SMEMORY. The return address (&RETURN) can be in the CONTROL routine or anywhere in the component where the call is issued. All registers are unchanged. &ADDL is the length of the composite addresses, and &NTRKS is the number of tracks or records in each memory-block (see Table 1).

33. STSR-(&J STSR &UR,&RR,&DUMMY)

This instruction, which is the first executable statement in each component, defines addressability and restores registers 0, 1, 14 and 15 which are used in V-type addressing. &UR and &RR are the using and branch registers for the component. &DUMMY is DUMMY (for separate compilation of components) or SOLID (if the component is positioned by the SUBMP service-macro). Further details are given in the subsection on Components, Section C.

34. TESTL-(&J TESTL &NAME)

&NAME is the relocatable address of the component (viz. the relocatable address of component SANPAKC is ANPAKC). This instruction appears in the calling procedures for components (viz. CALLX, TRANSFER, and GLOBAL) and the input/output routines (see below). Together with the STSR instruction, TESTL ensures that the registers 0, 1, 14, and 15 will

be restored after executing a branch and link instruction to a V-type address.

35. TRANSFER-(&J TRANSFER &N,&NAME,&RETURN)

TRANSFER is the so-called global calling procedure (see Section C). &N is the assigned level of the component S&NAME. &RETURN is defined above. If a TRANSFER instruction is issued within a component, then &N must be greater than the assigned level of the component. All registers are unchanged.

3. Input/Output Calling Procedures

Assembly Language Processor (ALP) input/output packages for tapes and cards are a part of the SOLID System. They are global calling procedures and can be used anywhere in the SOLID System. The DCB statements for these packages are in the INOUT macro-instruction (see Reserve). Thus, these DCB statements can be changed without recompiling the individual input/output components. Here two forms of the calling procedures are noted.

i. Card/Printer Operations

The print (PRINT), read (REIDY, Y=1,2,-8) and punch (PUNXH) operations in the SOLID System are extremely versatile. A single variable (&FORMAT) completely designates the formats (i.e., A, I, J, B, E, F, mixed (e.g., IFB), or X). If certain kinds of errors are made in these instructions, the operation is aborted and an appropriate error message is printed. The addresses in these operations can be either named locations or singly subscripted variables. The two-register form of IBM addressing (i.e., $D_2(X_2, B_2)$) is not permitted.

36. PRINT-(&J PRINT &FORMAT,&FORM,&TO)

All four-byte words (for numerical data) or bytes (for alphanumeric data) between the addresses &FROM and &TO+4 are printed in the fields and formats designated by &FORMAT. The first address (&FROM) must be located on a word boundary. If &FROM is greater than &TO the operation is aborted with the printed message: ADDRESSING ERROR. All registers are unaltered.

37. PUNSH-(&J PUNSH &FORMAT,&FROM,&TO)

PUNSH is a special form of the PUNXH operation which punches (on cards) the information between addresses &FROM and &TO in the X or column binary format. Unlike PUNXH, PUNSH does not have an associated component. No register is changed.

38. PUNXH-(&J PUNXH &FORMAT,&FROM,&TO)

Information between addresses &FROM and &TO+14 is punched on cards in the field(s) and formats designated by &FORMAT. Format options are A, I, B, E, F, mixed (e.g., IEF) or X. The address &FROM should be located on a single word boundary. Output obtained from the PUNXH operation can be read with the REIDY (Y=1,2,-8) operation. If &FROM is greater than &TO the PUNXH operation is aborted with the printed message: ADDRESSING ERROR. All registers are unaltered.

39 to 47 REIDY, with Y = 1,2,-8.

There are eight separate card-read instructions with the form:

&J REIDY &FROMAT,&W1,&W2,-&WY

&W1, &W2, -&WY are the addresses of the single variables or arrays where the information is to be stored in the designated format, &FORMAT (see Chapter V). The REIDY operation can be used to simultaneously load up to eight separate locations or arrays. The formats can be A, I, B, J, E, F, mixed or X. At the end of each completed card-reading operation the total number of bytes that were stored is in location JII. If any address (&W1, &W2, etc.) is in protected storage (i.e., in the O/S system or CONTROL routine) or the &FORMAT conflicts with the number of variables (Y)

the REIDY operation is aborted with an appropriate error message. No register is changed.

ii. Tape Operations

In the SOLID System there are two tape-read (REIDT and REIDJB) and on tape-write (WRITE) macro-instructions which are used in the input (SREIDT), job-list (SJOBLIST) and output (SOUTPUT) components. The DCB statements associated with these three macro-instructions are in the INOUT macro-instruction. &ADDRESS is the location in memory where the reading (from tape) or writing (on tape) is to begin. The location &JII contains either the number of bytes read into memory (for REIDT and REIDJB) or the number of bytes that are to be written on tape (for WRITE). Registers are unchanged in these operations. In their present form, all tape DCB's have a block-size of 3000 bytes and two buffers. Variable length records (up to 3000 bytes) are used for all tapes. However, there is no limit for the value in location &JII.

48. REIDJB-(&J REIDJB &ADDRESS,&JII)

This instruction uses the tape DCB statement TAPEJB (see INOUT in the Appendix). This tape normally contains the index data that is to be translated (or rearranged) to the JOB-LIST ITEM form which is used to trace the information path through the AUXILIARY FILE. The DCB name is TAPEJB and the DDNAME = COPAK8. No register is changed.

49. REIDT-(&J REIDT &ADDRESS,&JII)

In this macro-instruction the retrieval command (i.e., MODE = 0 (retrieve) or \neq 0 (store or update)) determines which of two tapes will be read. If MODE = 0 compressed bulk information, which will be decompressed by COPAK, is read with the TAPEINC DCB. For MODE \neq 0 bulk (or referenced) information is read with the TAPEIND DCB. This information will be processed by the COPAK compressor and stored on the device indicated by the output command, OUTPXT (see Chapter V). The DDNAMEs of TAPEINC and TAPEIND are COPAK5 and COPAK4 respectively. No register is changed.

50. WRITE-(&J WRITE &ADDRESS,&JII)

The &JII bytes of information beginning the &ADDRESS are written on the tape with DCB TAPEOTC. The DDNAME of the TAPEOTC DCB is COPAK6. All registers are unaltered.

4. Extended IBM and Message Operations.

There are seven macro-instructions in Class 4. Four of these (CSSCRN, DUMADD, LARGEXC, and SKIPP) are extensions of existing IBM 360 ALP instructions. The remaining three (DECPC, HEXPC, and TALE) are used to print messages. Brief descriptions of these Class 4 general Service Macros are given below.

51. CSSCRN-(&J CSSCRN &HLGTH,&ADD1,&ADD2)

This instruction compared the number of bytes in the half-word &HLGTH of the items beginning in locations &ADD1 and &ADD2. &HLGTH can contain any positive integer number up to 65,535. If &HLGTH contains zero the first 256 bytes are compared. Condition code settings are the same as those for the IBM CLC (Compare Logical Character) instruction. The registers are not changed by the CSSCRN instruction.

52. DECPC-(&J DECPC &NAME,&LOC,&DISCR)

This instruction prints a message about the decimal-ten number in &LOC thus: &NAME=&LOC &DISCR. If &DISCR=SECS, the number in &LOC is printed as a fixed-point number with two decimal places. Otherwise &LOC appears as an integer number. If &DISCR=RATE then BYTES PER SEC. is printed for &DISCR. &NAME and &DISCR are strings of up to 20 alphanumeric characters. No register is altered by the DECPC instruction.

53. DUMADD-(&J DUMADD &BRANCH,&DUMMY)

This instruction generates the IBM 360 ALP statement:
&DUMMY EQU &BRANCH

The address that is to be dummied, &DUMMY, is set equivalent to the address &BRANCH. The DUMADD macro-instruction is used extensively in the "SUBMP" type instructions of the category Reserve.

54. HEXPC-(&J HEXPC &NAME,&LOC,&LENGTH)

In this instruction, a string of hexadecimal characters, whose length (in bytes) is in the full word &LENGTH is printed thus:

&NAME=HEXADECIMAL STRING

The descriptive label (&NAME) can be any string of alphanumeric characters up to 19 bytes long. If &LENGTH contains a number greater than 65 or less than one, it is set to 65. The HEXPC instruction does not change any register.

55. LARGXC-(&J LARGXC &N-BYTES,&TO,&FROM)

LARGXC is the extended form of the IBM XC (Exclusive Or Character) instruction. The half-word &N-BYTES contains the number of bytes, (up to 65535) beginning in the address &FROM, that are to OR'ed into the string beginning in address &TO. If &NBYTES contains zero LARGXC is not executed. If it contains a negative half-word number, then the first 256 bytes are OR'ed. All registers are unchanged.

56. SKIPP-(&J SKIPP &LC)

&LC pages are skipped on the printer. All registers are saved. The instruction: SKIPP 1 skips one page.

57. TALE-(&J TALE &DASH,&MESSAGE,&NOTIMES)

This instruction prints the message: START OF NEW &MESSAGE.,&NOTIMES times. &DASH is the background in which the message is printed. Thus TALE ASTERICK,SEGMENT, 2 will print (extended over 133 characters):

```
*****START      OF      NEW      SEG-
MENT*****
*****START      OF      NEW      SEG-
MENT*****
```

All registers are unchanged.

5. String Movement Instructions

There are three service-macros which greatly facilitate the movement of strings of bytes in core-storage. In the first two instructions (LMOVE and RMOVE), the halfword &NOBYTES contains the number of bytes that are to be moved, beginning in address &FROM, to begin in location &TO. The addresses cannot be doubly subscripted in the RMOVE and LMOVE instructions. In all three macro-instructions all the registers are saved.

58. LMOVE-(&J LMOVE &NOBYTE,&TO,&FROM)

If &NOBYTES \leq 0 the LMOVE instruction is not executed. The maximum number of bytes that can be moved is 65,535. LMOVE is the extended form of IBM MVC instruction, which moves the bytes in strings from left to right. No registers are changed.

59. RMOVE-(&J RMOVE &NOBYTES,&TO,&FROM)

This instruction is similar to LMOVE except that the bytes in the strings are moved from right to left. Thus RMOVE can be used to displace a string (of length &NOBYTES) right. &TO and &FROM are the addresses of the leftmost bytes. No registers are changed.

60. RMVC-(&J RMVC &D1,&IR,&B1,&D2,&B2)

In this special form of the RMOVE macro-instruction &D1(&B1) is the address of the last byte in the string's new location. &D2(&B2) is the address of the last byte in the string that is to be moved. &D1 and &D2 are displacements. &B1, &B2 and &IR are registers. &IR contains the number of bytes which are to be moved. If &IR contains zero or is negative the RMVC instruction is not executed.

The reverse move macro-instructions (RMOVE or RMVC) are much slower than the LMOVE or MVC instructions. The

speed of the LMOVE and RMOVE instructions primarily determine the speed of the compressor and decompressor components of COPAK. These two instructions can perform best when hardware implemented.

b. Special Service-Macros

There are 64 macro-instructions entered in MACROPAK that are exclusively associated with the 31 components of the SOLID System. Two of these, BEGINS and MJARRAY, are used to initialize various aspects of the entire SOLID System. Another three (DEVICE, GETJLIST, and STRING), which use the General Service-Macro Input calling procedures, handle input for the SOLID System and its subsystems.

One special service-macro, SAVINGS, computes both the percentile savings and the thrupt rate for the COPAK Compressor. The remaining 58 special service macros perform the special bit, byte, string, and search operations that are an essential part of the SOLID System.

The last 58 special service macros, the String Movement and Arithmetic Operations (see Section B (a)), to a very substantial degree, determine the speed of compression, decompression, retrieval, storage, purging, and updating. The principal service-macros can be hardwired in the SOLID System.

In this Section, B(b), the 64 special service-macros are discussed in terms of the nine groups of components (see Section C). For example, there are two components, OPENSHT AND SSTATECL, which initialize the SOLID System. Also, there are six components in the COPAK compressor that used the twenty-nine compressor special service macros. In general all the special service macros have been designed to perform highly selective tasks at particular locations in the SOLID System. Registers, arrays, gates, and counters are changed to achieve the designed purpose. They cannot be used outside their designated environments.

1. Initializing Operations

61. BEGINS-(&J BEINGS)

This instruction, which appears near the beginning of the SSEARCH component, is the initializing routine for the global memory (SMEMORY) component. The physical characteristics of the virtual memory devices (i.e., the number of cylinders, tracks/cylinder, and the number of devices) are recorded in arrays CHECK, SOS, BWX, and LSX.

62. MJARRAY-(&J MJARRAY &ADDL)

MJARRAY is used in the principal initializing component, SSTATECL. It generates the M-J arrays when the SOLID System is used for the first time. The MJARRAY macro-instruction is executed if NEW-FILE appears on the first card in the data-deck. &ADDL is the number of bytes in the composite addresses (see Table 2).

2. Input Operations

Three macro-instructions (DEVICE, GETJLIST, and STRING) are used in the CONTROL routines to read (from cards) twenty-three of the 27 input commands for the SOLID System and its subsystems. Here the functions of the three macro-instructions will be discussed.

63. DEVICE-(&J DEVICE &INPUT,&OUTPUT,&SKIPS,&SLENGTH,&LLENGTH,&RNOS,&CORDS)

The DEVICE and STRING (see below) macro-instructions are special calling procedures for executing the SCOMMAND component. With DEVICE the seven device commands are read from a single card. The default options for each device command are executed in the component SCOMMAND.

64. GETJLIST-(&J GETJLIST &JLNPXT,&JLR-SKIP,&JLTRAN,&JLNORM,&KLENGTH,&N-JOBS,&NTAKS,&NVALUE,&JVALUE,&NUM-DIAG,&GENERATE)

GETJLIST is a special calling procedure for executing the

component SJOBLIST which is the supervisory component for the TRANSLATION PACKAGE. The first eight commands, which are associated exclusively with the control and use of the TRANSLATORS, are read from a single card. The last five commands are associated exclusively with the random generation of JOBLIST ITEMS by Monte Carlo Generators. These five commands are read (from a single card) only if the input command &JLNPXT=16.

65. STRING-(&J STRING &MODE,&POSTOP,&LEXCON,&LEXMODE,&LEXPCH)

STRING is a special calling procedure for executing the second half of the input component SCOMMAND. The five string commands are read from a single card.

3. Compressor

There are 29 special service macros associated with the COPAK Compressor, which has ten components (see Section C). Four of the special service macros (COPAB, COPAJ, JIMP1, and SINCOP) and two components SANPAKJ and SNUPAK are special variants that are used in the stand-alone Numeric, COPAKNU, and Alphanumeric, COPAKAN, versions of COPAK. The relationships between the components of the compressor are discussed hereinafter. Here it is sufficient to note the following:

- i. Three components handle input (SREADC and SREADT) and output (SOUTPUT) for the compressor part of the SOLID System.
- ii. Two components SCOMMAND and SSTATECL set up the strings of information for processing by the COPAK compressor. SSTATECL also initializes the AUXILIARY FILE.
- iii. The actual compression/decompression (of compressed information) is done by one special service macro (COPAB, COPAJ, or COPAK), which calls the components when it is used. Table 3 shows the relationships between the special service macro and the compressor/decompressor components they call.

TABLE 3.

Decompressor/compressor components combined means both Numeric and Alphanumeric Information.

Service-Macro	Components called	Type of information ¹
COPAB	SNUPAB	Numeric.
COPAJ	SANPAKJ, SANPAKC	Alphanumeric.
COPAK	SANPAKD, SNUPAK, SANPAKC	Combined.

¹ Handled by Compressor/Decompressor.

Strings of information are divided into segments, which are handled by the alphanumeric components (SANPAKC, SANPAKJ and SANPAKD). Segments are divided into substrings which are processed by the numeric components SNUPAB and SNUPAK. Detailed descriptions of the composition of the strings before and after compression will be found elsewhere.

The 32 special service macros associated with the COPAK compressor are identified in the following three categories:

Calling Procedures: Three macro-instructions (COPAB, COPAJ and COPAK) are used to call the compressor/decompressor components (see Table 3.)

Alphanumeric: Sixteen macro-instructions are associated exclusively with the three alphanumeric components (SANPAKC, SANPAKD and SANPAKJ).

Numeric: Ten special service macros are associated exclusively with the two numeric components (SNUPAB and SNUPAK).

A description of the 32 special service macros in their three categories begins next.

i. Calling Procedures:

A single macro-instruction in each CONTROL routine is

used to execute the compressor/decompressor parts of the several different forms of the COPAK compressor.

66. COPAB-(&J COPAB &RADD)

This special service macro uses the CALL1 instruction for calling the stand-alone numeric component (SNUPAB). After executing SNUPAB control returns to address &RADD in the CONTROL routines (COPAKNU and COPAKNUO). The COPAB Macro-instruction is associated exclusively with the "SUBMP" type instructions SUBCB and SUBCBO.

67. COPAJ-(&J COPAJ &RADD)

COPAJ uses the CALL2 instruction for calling SANPAKJ then SANPAKC for the stand-alone alphanumeric compressors (COPAKAN and COPAKANO). &RADD is the return address in the CONTROL routines. COPAJ is used only when the "SUBMP" type instructions SUBCJ and SUBCJO are used

68. COPAK-(&J COPAK &RADD)

COPAK uses the CALL3 instruction to call SANPAKD, SNUPAK, then SANPAKC (in that order) before returning control to the address &RADD in the CONTROL routines. COPAK is used in the CONTROL routines for the SOLID System (SOLIDE and SOLIDO) and for the stand-alone combined compressor (COPAKCO and COPAKCOO). Four "SUBMP" type instructions (SUBCE, SUBCO, SUBME, and SUBMO) are associated with COPAK.

ii. Alphanumeric

The distribution of the sixteen Special Service-Macros among the several components is shown in Table 4.

TABLE 4.

Distribution of the Sixteen Alphanumeric Special Service Macros among the indicated components.

SANPAKC	SANPAKD	SANPAKJ	SOUTPUT
BBM	BBMD	BBMD	PPCORDS
CCD	FIND	FIND	
FIND	LAND	TEMP1	
JAR	LEXD	LAND	
LEX	MAS	LEXD	
RRL	OPCORDS	MAS	
SAM	RAND	OPCORDS	
SAVINGS		RAND	

Two macro-instructions, OPCORDS and PPCORDS, organize and punch the permanent cords table (PCORDS) respectively. The PCORDS table is used by SANPAKC when it operates in the fast mode, (as described hereinafter). Thruput rates and percentile savings for the Compressors are computed in SAVINGS, which is executed at the end of SANPAKC. In production situations SAVINGS should be removed. The JIMP1 macro-instruction is used only in the stand-alone alphanumeric compressor (COPAKAN and COPAKANO). It performs the substring manipulations that are normally executed in the numeric component (SNUPAK). The remaining twelve special service macros perform highly specialized bit, byte, and string manipulative operations.

69. BBM-(&J BBM &JII,&CODE,&CORD1)

The BBM macro searches the first &JII bytes of a string for the repeated occurrences of the high order byte in &CODE. A bit-map is constructed to denote the relative positions in the string where the repeated byte occurs. The resulting bit-map and its length are stored beginning at the location &CORD1. All registers are preserved by this macro.

70. BBMD-(&J BBMD &JII,&CODE)

BBMD disassembles the bit-map which was constructed by the BBM macro. The associated code and bit-map are removed from the head of the string. The code is stored in &CODE. The addresses of the occurrences of the code in the string are determined by disassembling the bit-map and are stored in the array TL, which is defined in JUNKR. &JLL is reduced by the number of bytes in the bit-map minus two and the string is moved

left the same number of bytes. All registers are preserved by this macro.

71. CCD-(&J CCD &JII,&NBB)

The CCD macro searches a string for the repeats of a R-byte pattern which is located &NBB bytes from the beginning of a string. Only the &JII-&NBB bytes following the pattern in the string are searched. The addresses of the repeats of the pattern are stored in the array TL. This macro also determines whether or not a savings can be made by comparing $N*(R-1)$ with $R+2$. N is the number of repeats of the pattern. If $N*(R-1) > R+2$, then a savings can be made. Registers 0 to 3 are altered in this macro.

72. FIND-(&J FIND &JII,&CODE)

The FIND macro searches the first &JII bytes of a string for the single byte which is contained in &CODE. The addresses where the byte occurs are stored in the array TL. Registers 0 to 3 are altered by the FIND macro.

73. JAR-(&J JAR &JII,&NR)

The JAR macro combines the three low order bytes of &JII and the low order byte of &NR into a single machine word. &JII is increased by four bytes before the word is constructed. This composite control word is then inserted at the head of the string addressed by the register BRY Y. This control word is used to decompress the compressed strings of information.

74. JIMP1-(&J JIMP1 &RADD)

JIMP1 contains the numeric special service macros SOSCODE, ENDSS, and STRINGA, which are normally executed in the component SNUPAK (see (iii) Numeric below). JIMP1, which appears at the end of component SANPAKJ (see Section C), is used for the stand-alone alphanumeric compressor, COPAKAN and COPAKANO.

75. LAND-(&J LAND &JII,&NR)

The LAND macro removes the four bytes of control information placed at the head of the string by the JAR macro. The four bytes are then disassembled into two full words (&JII and &NR). &JII is decreased by four to its original value. &JII and &NR occupy three and one bytes in the composite control word. Registers 0, 1 and 2 are changed by this macro.

76. LEX-(&J LEX &JII,&CODE,&CORD1)

This macro places the high order byte in &CODE and the R-bytes in &CORD1 at the head of a string. The string is moved to the right R+1 bytes to accommodate the addition. &JII is increased by the number R+1. Register 2 is altered by this macro.

77. LEXD-(&J LEXD &JII,&CODE,&CORD1)

The LEXD macro removes the first R+1 bytes of a string storing the first byte in &CODE and the following R bytes in &CORD 1. The string is moved to the left R+1 bytes. &JII is reduced by the number R+1. Registers 0 and 1 are changed by this macro.

78. MAS-(&J MAS &JII,&NBB,&CORD1)

The MAS macro creates an R-byte opening in a string beginning &NBB byte from the head of the string. The R-bytes contained in &CORD1 are then substituted for the single byte which was located &NBB bytes from the head of the string. &JII is increased by R-1. Registers 0, 1 and 2 are changed by this macro.

79. OPCORDS-(&J OPCORDS &RADD)

In this macro-instruction, which appears at the beginning of the SANPAKD component (see Section C), some indicator messages are printed and the timing of the compression and decompression steps is begun. In production runs, the statements which perform these functions would be removed. The principal purpose of OPCORDS is to select those &TPCORD permanent cords with the highest saving ratios. &TPCORD is one of fourteen variable parameters which is set before SOLID is compiled. &RADD is the beginning location in the SANPAKD component of the string disassembly macro-instruction (STRINGD).

80. PPCORDS-(&J PPCORDS &RADD)

The PPCORDS macro-instruction appears at the beginning of the SOUTPUT component. &RADD is the first instruction is SOUTPUT which follows PPCORDS. The principal function of this macro is to print and punch (in column binary) the table of permanent cords (PCORDS) if the LEXMODE command is zero.

81. RAND-(&J RAND &JII,&REPEATS)

The RAND macro removes a single composite byte from the head of string. The left four bits of the byte are incremented by one and stored in &REPEATS. The right four bits of the byte are stored in RM. RM is incremented by one and stored in R. &JII is reduced by 1. Registers 0 to 3 are changed by this macro.

82. RRL-(&J RRL &JII,&REPEATS)

The RRL macro constructs a composite byte which is to be added to the head of a string. The left four bits of the byte contain the number in &REPEATS decreased by one. The right four bits of the byte contain the number in RM. The composite byte is inserted at the head of the string. &JII is increased by 1. Registers 0, 1 and 2 are altered by this macro.

83. SAM-(&J SAM &JII,&ENBB,&CODE)

SAM substitutes the single byte contained in &CODE for the R-bytes which are located &NBB bytes from the head of a string. The trailing bytes of the string are moved left R-1 to eliminate the spaces created during the substitution. Registers 0, 1 and 2 are changed by the SAM MACRO.

84. SAVINGS-(&J SAVINGS &RADD)

The percentile savings in storage and the thrupt rate expressed in bytes/second of uncompressed information) are computed in SAVINGS, which appears at the end of SANPAK. Both the percentile savings and thrupt rate are printed if $MODE \neq 0$ (storage and update modes). Just the thrupt rate is printed if $MODE=0$ (retrieval). SAVINGS would be omitted during production runs. &RADD is a dummy address.

iii. Numeric

Two of the ten special numeric service macros (CONSTRCT and EXTRAKT) assemble and decompose the segment control word (PARM), which contains the segment length and the number of substrings. Another two macro-instructions (SOSCODE and ENDSS) ensure that the compressor (NUPAKC) and decompressor (NUPAKD) parts will process each segment of information one substring at a time. SOSCODE also redefines the state-of-substring control word, SOS. Of the remaining three macro-instructions, two (STRINGA and STRINGD) assemble and decompose compressed substrings respectively. The macro-instruction COPAKEND, which appears in the output component (SOUTPUT), executes the post-operation commands.

85. CONSTRCT-(&J CONSTRCT &PARM,&III,&NV)

The four byte composite status-of-segment control word, &PARM, is constructed with &JII in the leftmost three bytes and &NV in the rightmost byte. &JII is the total number of bytes in the segment of information. &NV is the number of substrings in the segment. &JII and &NV are four byte words. The CONSTRCT operation leaves all registers unchanged.

86. COPAKEND-(&J COPAKEND)

The COPAKEND macro-instruction, which appears at the end of component SOUTPUT, executes the post-operation commands POSTOP(=LJ), NJOBS (number of bulk-storage items), and NTASKS (number of ITEMS in Joblist). These commands are:

LJ<0; increment LJ by one and, if LJ=0, set SWITCH=0 and LEXPCH=1. This means that for the next segment of information the alphanumeric compressor (SANPAKC) will operate in the fast mode and the PCORDS table will not be punched.

Decrement NJOBS by one. If NJOBS >0 control goes to the location CARDREAD in the CONTROL routine, where the next string is read.

If, after decrementation, NJOBS <0, then NTASKS is examined. For NTASKS <0 control goes to the location LOOKFILE in the CONTROL routine. Normally the RETRIEVAL PACKAGE is entered at LOOKFILE. For NTASKS = 0 control goes to the macro-instruction DISPENSE or DISPOSE at location ANSWER in the CONTROL routine. Other options of the POSTOP command are executed in DISPENSE and DISPOSE.

87. ENDSS-(&J ENDSS &RADD)

The ENDSS macro-instruction appears in the SNUPAK and SNUPAB components after the SOSCODE, NUPAKC, and NUPAKD macro-instructions. In ENDSS the new four byte status-of-substring composite control word (SOS) is constructed. During decompression the absolute error check for each decompressed substring occurs. If an error is detected, control passes to location RTRANMIT in the macro-instruction DEVICES (see Section A), where the error procedure component (SACTION) is called (see Section C). If no errors are detected, and some substrings are still to be processed, control returns from ENDSS to location &RADD in SNUPAB or SNUPAK.

88. EXTRAKT-(&J EXTRAKT &JII,&NV,&PARM)

This macro-instruction is the reverse of the CONSTRCT instruction. The number of bytes in the segment (&JII) and the number of substrings (&NV) are extracted from the four byte composite status-of-segment control word, &PARM. &JII and &NV are four byte words. All registers and &PARM are unchanged.

89. NUPAKC-(&J NUPAKC &RADD)

NUPAKC is the numeric compressor macro-instruction which is used in numeric compressor components, SNUPAB and SNUPAK. After a substring is processed by NUPAKC, control goes to location &RADD in SNUPAB or SNUPAKC for execution of the ENDSS macro-instruction.

90. NUPAKD-(&J NUPAKD &RADD)

NUPAKD is the numeric decompressor instruction in the component SNUPAK. After processing a substring by NUPAKD, control is returned to the ENDSS instruction in SNUPAK or SNUPAB at location &RADD. NUPAKD reverses the steps executed in NUPAKC.

91. SINCOP-(&J SINCOP &RADD)

SINCOP is used in the numeric stand-alone compressor component (SNUPAB). It contains the macro-instruction STRINGD and those parts of component SANPAKD that are needed to process segments of information. Information about the segments is printed and timing of compression/decompression is begun in SINCOP. &RADD is a dummy address.

92. SOSCODE-(&J SOSCODE &RADD)

The SOSCODE macro-instruction appears near the beginning of the SNUPAK component (see Appendix). Its principal purpose is to initialize the substrings for processing by either the compressor or decompressor parts of the numeric compressor component, SNUPAK or SNUPAB. The sign and NDR, which occupies the rightmost four bits of the status-of-substring commands, are modified to divert control to either the macro NUPAKC or to component SANPAKC. &RADD is the instruction in SNUPAK which follows SOSCODE.

93. STRINGA-(&J STRINGA &RADD)

This macro is used near the end of the SNUPAK and SNUPAB components (see Appendix). Its principal function during decompression is to check that processing by the numeric decompressor part of SNUPAK yielded the correct number of bytes for the segment. Disagreement leads to the printing of an ap-

propriate error message and control passes to the location RTRANMIT, for processing by the error-procedure component (SACTION).

During compression STRINGA inserts the information that is needed to decompress and check the substrings at the head of the segment of compressed information.

&RADD is the location of the instruction which follows STRINGA in the components SNUPAK or SNUPAB.

94. STRINGD-(&J STRINGD &RADD)

This macro extracts the control information that was inserted at the head of the segment by STRINGA during compression. This control information is used to check the alphanumeric decompression (just completed in SANPAKD) and is used by the NUPAKD macro-instruction to decompress the segment, one substring at a time. Additional error checks are made on each substring (in ENDSS) and also on the segment after processing by SNUPAK (in STRINGA). &RADD is the return address in the component SANPAKD or SANPAKJ.

4. TRANSLATION PACKAGE

The TRANSLATION PACKAGE is a subsystem of SOLID that produces normalized JOBLIST ITEMS from the assigned descriptor sets or generates them with random number generators. The JOBLIST ITEMS are used to trace, create, purge, or update the information paths in the AUXILIARY FILE.

The package consists of eleven components and eleven special service macros. One special service macro (GETJLIST), which has been described above (see macro 68, calls the TRANSLATION PACKAGE supervisory component (SJOBLIST) from the CONTROL routines SOLIDE and SOLIDO. SJOBLIST reads the thirteen TRANSLATION PACKAGE commands and executes them. SJOBLIST performs the following six functions:

- i. Read TRANSLATION PACKAGE commands from cards.
- ii. Read the assigned descriptor-set from the designated input device, or
- iii. Generate JOBLIST ITEMS with random number generators.
- iv. Translate the assigned descriptor-sets to the JOBLIST form.
- v. Read the override data from cards.
- vi. Normalize the JOBLIST ITEMS produced in functions iii and iv. The last function, vi, is not executed if the command JLNORM equals zero.

One component (SGENITEM), which is called from SJOBLIST by the special service macro JLITEM, generates the random JOBLIST ITEMS. Another six special service macros (BITTHROW, JBLISTI, KERTHROW, LBLTHROW, MJTHROW and SQUEEZE) are associated exclusively with the component SGENITEM.

The special service macro TRANLATE calls the five TRANSLATOR components from SJOBLIST. There are provisions for including up to 255 different TRANSLATORS. The TRANSLATOR components, which rearrange the assigned descriptor-sets to the JOBLIST form, must be coded for each new collection of items.

The override data is read (from cards) by a single special service macro (OVERRIDE) in SJOBLIST.

The NORMALIZATION PACKAGE consists of four components (SNORMAL, SCYCLIC, SREFLECT, and SXCHANGE) and one special service macro, NORMFORM. However, three of these components (the TRANSFORMATION PACKAGE) are also used by the MOBILE CANONICALIZATION PACKAGE. NORMFORM is used in SJOBLIST to call the normalization supervisory component, SNORMAL.

Seven of the ten special service macros that are to be described below require an understanding of the JOBLIST ITEM structure, which follows.

	LJBI	M	m ₁ m ₂ . . . m _M	LD ₀	LD ₁	LD ₂	LD ₃	LD ₄	LD ₅
Bytes=	2	2	M	2	(LD ₀ -2)	2	(LD ₁ -2)	2	(LD ₅ -2)

LJBI is the number of bytes in the JOBLIST ITEM. M is the number of nested information representations, which have ranks $m_1, 2, m_3, \dots, m_M$. LD₀ is the principal diagonal of the Information Representation (IR) of rank

$$r = \sum_{i=1}^M m_i$$

and LD₀-2 is length. BD_i and LD_i (i ≠ 0) are the left adjusted second bit-map (B₂) diagonal and the associated LABEL diagonal respectively. (LB_i-2) and LL_i-2 are the lengths of the screens BD_i and LD_i. A JOBLIST item terminates with an asterisk attached to the last non-zero diagonal of LABEL. A JOBLIST consists of NTASKS JOBLIST items, whose total length is found in JLL. A Bit Map Item consists of a Bit-Map Head (LB_i) and a Bit-Map Screen (BD_i) Information Representation Items consists of a I.R. Head (i.e. LL_i) and an I.R. Screen (i.e. LD_i). Bit-Map and I.R. elements are the bits and elements in their screens.

95 BITTHROW-(&J BITTHROW &NUMBITS,&BITMAP)

This macro-instruction generates a pseudo-random Bit Map Item, which contains a two-byte Bit-Map Head and the Bit-Map Screen. The Bit-Map Head contains the length of the Bit-Map Screen plus two. &NUMBITS contains the number of elements (i.e. bits) in the screen. &BITMAP is the beginning address of the Bit-Map Item. All registers are unchanged.

96 JBLISTI(&J JBLISTI &JBLIST,&MVALUE,&JVALUE,&NUMDIAG)

JBLISTI constructs a pseudo-random JOBLIST item, in the array &JBLIST(=JBLIST). Location &MVALUE contains the stipulated value of 'M'. &JVALUE contains the maximum value for each of the m_i (1 ≤ i ≤ M) values in the first screen (see above). &NUMDIAG contains the number of Bit-Map Item - I.R. Item pairs that are to be constructed. No register is altered.

Note: JBLIST quits constructing the Bit Map Item - I.R. Item pairs when the JOBLIST item is within 16 bytes of the end of the &JBLIST array. The length of this array is in location LJBLIST. The truncated JOBLIST items are processed normally.

97 JLITEM-(&J JLITEM &JBLIST,&MVALUE,&JVALUE,&NUMDIAG)

This macro-instruction appears in the supervisory component (SJOBLIST) and it calls the random JOBLIST ITEM Generator Component, SGENITEM. Registers 2 through 5 are loaded with the addresses of the four variables, in that order. If the &JBLIST array is less than 256 bytes long, an error message is printed and execution is terminated. The four variable parameters (&JBLIST, &MVALUE, &JVALUE and &NUMDIAG) are defined above.

98 KERTHROW-(&J KERTHROW &JBLIST) (i.e.

KERTHROW constructs a random principal diagonal for the information representation i.e., LD₀ above) and stores it in the array JBLIST at the location &JBLIST. The principal diagonal, which is used as the second screen in the JOBLIST item, has no 'zero' I.R. elements. Registers 5 through 15 are not changed.

99 LBTHROW-(&J LBLTHROW &JAYBITS,&BITMAP)

This macro-instruction produces a single I.R. Item that is associated with the Bit Map Item which begins in location &BITMAP. &JAYBITS contains the number of I.R. elements that are to be generated. All registers are unchanged.

100 MJTHROW-(&J MJTHROW &JBLIST,&M,&JAY)

MJTHROW assigns to 'm' the value of &M and then generates M random values of m_i (see above) whose values lie between one and &JAY. &JBLIST is the

starting address in the array JBLIST of the current JOBLIST ITEM. &M contains assigned numbers of nested representations (M). &JAY contains the maximum rank for the M nested representations. If &M ≤ 0 or &JAY ≤ 1 an error message is printed and execution is terminated. All sixteen registers are unchanged.

101. NORMFORM-(&J NORMFORM &JBLIST,&NTASKS,&JLL,&KLENGTH)

This macro-instruction is used in the TRANSLATION PACKAGE supervisory component (SJOBLIST) to call the principal NORMALIZATION PACKAGE component (SNORMAL). The four variable parameters of NORMFORM are dummy parameters intended to indicate the key information that is needed by SNORMAL. &JBLIST(=JBLIST) is the beginning address of the JOBLIST. &NTASKS (=NTASKS) contains the number of items in the JOBLIST. &JLL (=JLL) contains the total length of the JOBLIST. &KLENGTH (=KLENGTH) contains the number of bytes in each element of the I.R. NORMFORM does not change any register.

102. OVERRIDE-(OVERRIDE)

The OVERRIDE macro-instruction is executed in component SJOBLIST when control returns from TRANSLATE. Information about the number of Type 1, Type 2, and Type 3 override codes (which is automatically collected by the TRANSLATORS) is used in OVERRIDE to read updating information from cards. Registers 3 and 4 are changed.

103. SQUEEZE-(&J SQUEEZE &NUMBER,&ADDRESS)

SQUEEZE removes the 'zero' or 'empty' elements from a single I.R. Screen and modifies the I.R. Head accordingly. &NUMBER CONTAINS THE NUMBER OF I.R. Elements in the I.R. Screen. &ADDRESS is the beginning address of the associated Bit Map Item. All registers are unaltered.

104. TRANLATE-(&J TRANLATE &ARRAY,&NTASKS,&JLL,&KLENGTH)

This macro-instruction is used in the supervisory component of the TRANSLATION PACKAGE, SJOBLIST, to call the five TRANSLATOR components (STLATORX, with X=1,2,...5). These TRANSLATORS rearrange the assigned descriptor-sets to the JOBLIST form. Timing of searches begins, and information about the TRANSLATOR is printed in TRANLATE. The translator gate, (TGATE) which is a command read in SJOBLIST, determines which of the five TRANSLATORS will be used. There are provisions in TRANLATE for incorporating up to 255 different translators. When control returns from the selected TRANSLATOR component, NOVER1, NOVER2 and NOVER3 contain the number of occurrences of Type 1, Type 2 and Type 3 override codes. The locations of these override codes in the JOBLIST is located in the three primary override-arrays. This information is used by the OVERRIDE macro-instruction in the component SJOBLIST to read update information from cards. The variables are the dummy parameters defined for NORMFORM.

5. Transformation Package

The TRANSFORMATION PACKAGE consists of three components (SCYCLIC, SREFLECT, AND SXCHANGE). The transformation Package is used by both the TRANSLATION and RETRIEVAL PACKAGES.

6. Normalization Package.

At this time the NORMALIZATION PACKAGE consists of one empty shell component (SNORMAL) and its calling

macro, NORMFORM (see Translation Package Above). The normalization Package will use the Transformation Package and it will be executed in the supervisory component SJOBLIST after the TRANSLATORS have produced JOBLISTS (see 4. Translation Package). New special reserve macros will be incorporated here when they are implemented.

7. Retrieval Package

The RETRIEVAL PACKAGE will automatically retrieve, store, purge, or update the AUXILIARY FILE and/or the MAIN FILE with information produced by the Translation Package and, if necessary, the MOBILE CANONICALIZATION PACKAGE also.

The RETRIEVAL PACKAGE consists of 16 special service macros, two components (SSEARCH and SRESULT), and the MOBILE CANONICALIZATION and GLOBAL MEMORY PACKAGES. Another two special service macros (BEGINS and MJARRAY) and one component (SS-TATECL) initialize the RETRIEVAL Package (see 1. Initializing Operations above). They will not be considered further here.

The GLOBAL MEMORY PACKAGE is called from the SSEARCH component by the TBADD macro-instruction whenever a memory-block is to be transferred between core-storage and virtual memory. The calling procedure, GLOBAL, has been described in (2. Input Operations) SRESULT, which prints search performance data, is called from the CONTROL routines SOLIDE and SOLIDO after each use of the RETRIEVAL PACKAGE. In production situations SRESULT will not be used.

The structure of the composite addresses, that are used in the AUXILIARY FILE to reference the memory-blocks, is:

	D	DNO	TRK	CYLN	FMADD
No. of bits	4	4	6	10	24

D and DNO specify the type of device and its number respectively. TRK, CYLN, and FMADD are the track, cylinder, and fast-memory addresses (relative to the starting point) respectively. If a magnetic type is specified, then the sixteen bits of TRK and CYLN specify the record number.

Three of the 16 special service macros (APART, ASADD, and COMPARE), which are used throughout the RETRIEVAL and GLOBAL MEMORY Packages, are used to decompose, assemble, and compare the component parts of the composite addresses. Another macro-instruction, BULK, updates the Bulk Storage Composite Address (also called BULK) after each new allocation of storage for referenced information. The macro-instruction DISPENSE is executed in the CONTROL routines SOLIDE and SOLIDO when control returns from components SSEARCH and SRESULT. Its primary function is to execute the post-operation commands which determine whether or not the COPAK compressor will be used. DISPOSE is a special form of DISPENSE that is used in the stand-alone compressor CONTROL routines. The macro-instruction LINKHOLE, listed as instruction (115) below, ensures that unused storage in memory-blocks, released during purging operations, will be efficiently searched. The remaining nine of the 16 special service macros are associated exclusively with the heavily nested SSEARCH component.

105. APART-(&J APART &ADDRESS,&RD,&RD-NO,&RTRK,&RCYLN,&RFMADD)

APART extracts the five component parts from the composite address (in location &ADDRESS) into the designated register. If &RTRK and &RCYLN are the same register, a magnetic tape is specified, and both registers contain the record number. The composite address, in location &ADDRESS, and all other registers are unchanged.

106. ASADD-(&J ASADD &ADDRESS,&RD,&RD-NO,&RTRK,&RCYLN,&RFMADD)

This macro-instruction assembles the composite address in location &ADDRESS from its five component parts, in the designated registers. If &RTRK and &RCYLN are the same register, a magnetic tape is specified, and the record number is placed in the twelve bits normally occupied by RTRK and CYLN (see above).

107. AUXFILE-(&J AUXFILE &LSLOW,&ADDL)

AUXFILE is executed in the SSEARCH component whenever the terminal location of an information path has been found or created. The terminal locations are in RFILE, and they contain the address(es) of the compressed reference information. AUXFILE inserts (storage or updating), deletes (purging), or collects (retrieval) the bulk-storage address(es) in the RFILE sub-arrays of the AUXILIARY FILE. New composite bulk storage addresses are assigned and the macro-instruction BULK is executed in AUXFILE. The two System Parameters (&SLOW and &ADDL) are the lengths of slow and fast and the entire composite address respectively. In our program &SLOW=3 and &ASSL=6.

Note: A record of all the transactions completed in AUXFILE for each request is kept in NJOBS(=number of addresses) and in a 360 byte sub-array at the high address end of the principal data array, YY. Thus, at this time 360/&SLOW=120 is the limit of the number of transactions that can be completed for each storage, update, purging, or retrieval task. Another subsequent macro-instruction, which is to be inserted immediately after the fifth last instruction of AUXFILE (see Appendix), will execute the options whenever NJOBS>120. In the storage and update modes NJOBS (set in the TRANSLATION PACKAGE) should be less than 121. The bulk storage addresses in the sub-array at the high address end of array YY is transferred to the Joblist work array (whose address is in location AJBWORK) at the end of the SSEARCH component. This information is used by the COPAK compressor and the output component, SOUTPUT.

108. BULK-(&J BULK)

This macro-instruction is executed in AUXFILE after each new assignment of a bulk storage address (called BULK). It increments the record number in the composite address BULK by one.

Note: In one present form BULK assumes that the compressed referenced information is to be stored on a single magnetic tape - BULK must be altered before production begins.

109. COMPARE-(&J COMPARE &ADDL,&ADD2,&ADDL)

The first four parts (D, DNO, TRK and CYLN) in the composite addresses &ADD1 and &ADD2 are compared. &ADDL is the number of bytes (viz. six, in the addresses). The COMPARE instruction sets the condition code in the PSW OF THE System 360. &ADDL is a System Parameter which is set at compilation time.

110. CREATE-(&J CREATE &ADDL,&LSLOW,&LFAST,&NTRKS,&TRKL,&MATRIXL,&MATRIXS)

The CREATE MACRO, which is called by TBADD in the SSEARCH component, is entered whenever a new sub-array is to be created in a memory-block. The new sub-array may be needed to define a new subpath or it may be needed to extend an RFILE subarray. CREATE which is closely interconnected with the global memory component (SMEMORY), via the TBADD instruction, is extremely complex. Without this special service-macro the generalized retrieval system which we have devised would not exist. The seven variable names (of CREATE) are System Parameters that have been fully defined elsewhere in this disclosure. If &ADDL, &SLOW, &LFAST, &NTRKS or &TRKL are changed, the AUXILIARY FILE must be started from scratch. &MATRIXL and &MATRIXS can be changed at any time.

111. CSCREEN-(&J CSCREEN &HLGTH,&ADD1,&ADD2)

CSCREEN is used in the SCREEN macro-instruction whenever the screen portion of an EXECUTIVE POINTER is to be compared with the corresponding

screen in the JOBLIST item. &ADD1 is the beginning location of the EXECUTIVE POINTER screen. &ADD2 is the address of the JOBLIST item screen. &HLGTH is a half-word which contains the screen length. All registers are unchanged.

The byte JI is used to indicate the comparison status thus: JI=00; the screens are equal.

JI=01; the first screen (&ADD1) is zero (i.e., the location is empty).

JI=02; the first screen (&ADDL) is less than the second (&ADD2).

JI=04; the first screen is greater than the second.

112. DISPENSE-(&J DISPENSE &RADD)

DISPENSE is used in the CONTROL routines SOLIDE and SOLIDO immediately after the SSEARCH and SRESULT components are called. It is executed after each use of SSEARCH and again after each use of the COPAK Compressor, when it is called from COPAKEND in SOUTPUT. DISPENSE executes the Post-Operation (LJ), NTASKS, and NJOBS commands. Register 1 is altered. &RADD is the entry location (in the CONTROL routine) for reading input for the compressor.

113. DISPOSE-(&J DISPOSE &RADD)

DISPOSE is used in place of DISPENSE in the CONTROL routines for the six stand-alone compressors. The primary difference between DISPENSE and DISPOSE is the way they execute the NTASKS and NJOBS commands.

114. INSERT-(&J INSERT &ADD1,&LFAST,<HAYY)

INSERT is used in the macro-instruction SCREEN (see below) to insert new executive pointers in their correct positions in subarrays in the AUXILIARY FILE. It is executed only when an EXECUTIVE POINTER must be moved from a location in any subarray. &ADDL, &LFAST, and <HYY are System Parameters.

115. LINKHOLE-&J LINKHOLE &EPLNGTH,&FEMPTY)

LINKHOLE is used in the CREATE macro-instruction. It is executed only when a memory-block is completely used, and before a new memory-block is created. Its designed purpose is to reuse vacant storage areas in the resident memory-block that might have been released during purging operations. Implementation instructions have been incorporated in the macro (see Appendix). EPLNGTH contains the length of the new Executive Pointer. &FEMPTY is the address of the first available subarray in the memory-block. All available subarrays are chained together via their link addresses.

116. MMATCH-(&J MMATCH &NOVER1,&NOVER2, &NOVER3,&CURSCRN,&JBLIST,&JBWORK,&KLENGTH)

This macro-instruction is executed in the SSEARCH component whenever a screen or index, which corresponds to a subpath, is not found. It is also executed during retrieval operations if there are override codes present in the JOBLIST items. If a storage operation is being performed, the signal, MSIGNAL, is set to indicate that an insertion is to be made in the resident memory-block and, after exiting from MMATCH, the insertion is made. The screening procedure, the CREATE macro, and the global-memory component (SMEMORY) are tied together by the MSIGNAL and JI (see CSCREEN) multi-bit signalling system. This signalling system polices the resident memory-block and notifies CREATE whether or not arrays are to be created. It also notifies SMEMORY what procedure is to be executed when a new memory-block is needed or when the job-stream is to be terminated. In retrieval operations MMATCH aborts the search, if no overrides are present, passes control to the MOBILE CANONICALIZATION PACKAGE via its calling procedure,

STRATEGY. The last four variable names to MMATCH specify information needed in STRATEGY (see below). The screen procedures, MMATCH, and the MOBILE CANONICALIZATION PACKAGE are tied together by the SRGATE multi-bit signalling system.

&NOVER1=NOVER1 contains the number of Type 1 override codes.

&NOVER2=NOVER2 contains the number of Type 2 override codes.

&NOVER3=NOVER3 contains the number of Type 3 override codes. &CURSCRN,&JBLIST,&JBWORK, and &KLENGTH are dummy variable names for STRATEGY which are explicitly defined in the STRATEGY macro (see Appendix).

117. SCREEN-(&J SCREEN &ADDL,&LFAST,&ADD1,<HAYY)

The SCREEN macro-instruction is executed in the SSEARCH Component whenever a screen is sought, (in the resident memory-block,) in one of the column arrays, or its extensions. After initializing the counters and registers, if no over-ride codes are present in the JOBLIST item, the location in the array where the executive pointer should be is determined by the SUPERSCH macro-instruction. If overrides are present, control will go directly to MMATCH, for processing by the MOBILE CANONICALIZATION PACKAGE. If the executive pointer selected by SUPERSCH contains the sought screen (i.e. the sub-path has been found) the tracing procedure (through TBADD) continues. If the two screens are not identical, one of the following can occur:

1. A retrieval job is terminated via the MMATCH macro with a message that the search was unsuccessful.
2. If a vacancy exists, control again goes to MMATCH for insertion of the new executive pointer, which defines a new subpath, and the tracing of the information path continues.
3. If no vacancy exists a hole is created for the new executive pointer and (2) occurs. The hole creating procedure is as follows: If the column-array has a vacant location then all executive pointers greater than the one to be inserted are moved and (2) occurs. If the array is filled, the last executive pointer (EP_L) is saved; a hole is created; the new executive pointer is inserted; the continuance or extension array is found (via the link-address or created (via TBADD), and CREATE); and SCREEN is used to search the new array for the location to insert EP_L . This procedure is repeated until all the executive pointers in an array and its continuances are arranged in increasing order, then the tracing (or creation) of sub-paths continues in the normal manner. The procedure that will be used to ensure that the information paths do not cross memory-blocks, thus eliminating costly additional accesses to the virtual memory, is somewhat analogous to the hole creating procedure that has just been described. However, in this case, the Automatic purging (still to be implemented), hole-closing, and hole-creating capability of SCREEN must all be used. Implementation of this capability will require new service macros for SCREEN and TBADD. The four variable names in the Screen prototype statement are System Parameters, defined earlier.

118. STRATEGY-(&J STRATEGY &CURSCRN,&JBLIST,&JBWORK,<HAYY)

This macro instruction appears in the MMATCH macro. It is the entry macro for the MOBILE CANONICALIZATION PACKAGE. (see below). STRATEGY is executed if any overrides are present. It interacts closely with SCREEN and TBADD via the multi-bit

SRGATE command. &CURSCRN, &JBLIST, &JBWORK, and <HAYY are dummy variables which are fully explained in the STRATEGY macro in the Appendix.

119. SUPERSCH-(&J SUPERSCH &ADDL,&LFAST) 5

SUPERSCH is executed in the SCREEN macro-instructions. It is executed if no override codes are present. However, it can be entered from the macro-instruction STRATEGY. SUPERSCH does a partition search of an array to find where the new executive pointer should be located. The two System Parameters, &ADDL and &LFAST, have already been defined.

120. TBADD-(&J TBADD &ADDL,&LSLOW,&LFAST,&NTRKS,&TRKL,&MATRIXL,&MATRIXS) 15

The link address (for continuance or extension arrays) or address in the executive pointer obtained by SCREEN is stored in ADDRESS, and control goes to the TBADD macro. The peripheral equipment addresses, which occupy the three left bytes of ADDRESS and CURRENT, are compared in TBADD to determine if the required memory-block (designated by ADDRESS) is resident in core. If it is not in core the global-memory component (SMEMORY) fetches it. If the create bit of the MSIGNAL signal byte is on, TBADD directs CREATE to create a new column-array with &MATRIXL or &MATRIXS locations for executive pointers. The variables for TBADD are defined in the section on System Parameters.

8. Global Memory

The fully implemented GLOBAL MEMORY consists of one component (SMEMORY and two special service macros (DCBMEM and GLOBAL). The calling procedure (GLOBAL), which has already been described above, is used in the TBADD macro-instructions. GLOBAL MEMORY transfers memory-blocks between core-storage and virtual memory. The Global Memory component, SMEMORY, contains its own DCB or format statements, which are specified in the macro-instruction DCBMEM.

121. DCBMEM-(&J DCBMEM)

This macro-instruction appears at the end of the SMEMORY components. It contains DCB or format statements together with the IBM OPEN and CLOSE instructions for all the peripheral devices that can be used by the GLOBAL MEMORY. If new peripherals are added the initializing macro-instruction BEGINS, which appears in the SSEARCH component, must be altered. However, except for the recompilation of SSEARCH and SMEMORY, no other changes are necessary. One IBM 2314 disk is currently used for the virtual memory. Branching to the location CLOSE terminates the job-stream. The virtual memory peripheral devices are opened (in DCBMEM) when SMEMORY is used for the first time. The DCB name is GLOBAL1 and its DDNAME is COPAK7.

9. Mobile Canonicalization Package

This package consists of the calling procedure STRATEGY, which is described above, and two components (SMATCH and SMOBILE), which are referred to below. The aforementioned MOBILE CANONICALIZATION PACKAGE can also use the TRANSFORMATION PACKAGE.

Component SMATCH will determine whether or not mismatches (in MMATCH) are solely due to the presence of one or more override codes. Component SMOBILE will be executed after SMATCH only if the NORMALIZATION PACKAGE was used (to obtain NORMAL FORMS). The design objectives for the various parts of the MOBILE CANONICALIZATION PACKAGE have been fully discussed in the article by P.A. D. deMaine and B.A. Marron, "The SOLID System I. A Method for Organizing and Searching Files." in the book "Information Retrieval: A Criti-

cal View." The book is edited by G. Schechter and was published by the Thompson Book Company of Washington, D.C. in 1967.

C. Components

Components are macro-instructions that are stored in the macro-library, SOLID.MACLIB, which have their own using statements to define addressability. In the components it is assumed that data about information that is being processed and the information itself will be found in certain locations and arrays whose addressability is established in the CONTROL routine. Except for this restriction, the components may be viewed as independent subprograms or subroutines. They can be separately compiled (as named CSECTS) for use in planned overlays (see CONTROL PROGRAMS), or they may be compiled with the CONTROL routine by inserting their ENTRY and prototype statements into the 'SUBMP' type-instruction. This flexibility in the use of components facilitates the implementation or modification of components, and, with planned overlays, makes it easy to fit the SOLID System onto small 360 configurations. Moreover, with this flexibility the SOLID System can be spread over several partitions in a single computer or over several computers in a network to further improve its already impressive performance.

The calling procedures are used to branch between the CONTROL routine (viz. main-stem) and the components or among the components themselves. The three different types of calling procedures are illustrated in FIG. 15, and they are described in Section B. The points that are to be emphasized in this section, C, will be illustrated with the calling procedures CALL1 (prototype: &J CALL1 &NAME,&ALINST) and TRANSFER (prototype: &J TRANSFER &N,&NAME,&RETURN), and the alphanumeric compressor component (prototype: SANPAKC &NAME, &UR, &RR, &DUMMY). &NAME is the relocatable address of the component, obtained by dropping the first S from the component's name. &ALINST and &RETURN are return addresses. &UR and &RR are the USING and branch registers for the component. &RR can be any register except &UR or 10, 11, and 12, which are used to establish addressability in the CONTROL routine. The choice of the base or USING register (&UR) is generally restricted to 8, 9, and sometimes 7 (see below). &DUMMY designates whether the component was separately compiled as a named CSECT (&DUMMY=DUMMY) or if it was compiled in the CONTROL routine by inserting the prototype statement in the "SUBMP" type instruction (&DUMMY=SOLID). If &DUMMY is equal to DUMMY, the separately compiled component is stored in the module-load library, SOLID.LOAD, for use in planned overlays.

The instruction:

```
CALL1 ANPAKC,RANPAKC
```

executes the branch-and-link to the component (SANPAKC) and, on completion of the component's task, returns control to the location RANPAKC. Because the CALLX (X=1,2,-5) instruction changes the values of registers &UR, &RR and others they can only be safely used if the return address (&ALINST=RANPAKC) is in the CONTROL routine. In general, they cannot be used to branch between two components.

The TRANSFER instructions can be used to branch between the CONTROL routine and components or among components. In this instruction all registers are unchanged. Thus the return address, &RETURN, can be either in the CONTROL routine or in the component where the instruction was issued. This greater versatility has been achieved by assigning levels to every component. The level of the requested component is incorporated into the TRANSFER instruction. For example:

```
TRANSFER 1, ANPAKC,RANPAKC
```

means that control is to be transferred to component SNAPAKC at the first level. The return address RANPAKC can be in the CONTROL routine or it can be in another com-

ponent (where the TRANSFER was used). The rules for using the TRANSFER instructions are given next.

- i. There are, in the given form, five levels available. This number can be increased by adding to the COSAVE&N (&N=1,2,-5) arrays in the macro-instruction SAVEAREA.
- ii. While every component has been assigned a specific level, new levels can be arbitrarily reassigned. However, a level assigned to the preceding members of a chain of components cannot be used. This means that with a CALL1 and five successive TRANSFER instructions control can be transferred from the CONTROL routine through five components. The return address for each TRANSFER instruction can be in the CONTROL routine or in the component which contains it.
- iii. Components which have been assigned the level "GLOBAL" are not subject to the restrictions in (ii). They have their own special calling procedures, which can be used anywhere in the SOLID system. PRINT,PUNXH,REIDX(X=8) and GLOBAL are such calling procedures.

The levels assigned to each component are shown in Table 5.

TABLE 5.

Levels and Classes of Components. The assigned levels are used in the TRANSFER instruction (see Text). The classes are the categories in this Section, C.

Component	Number	Class	Level	Component	Number	Class	Level
OPENSHTUT	129	1	0	SJOBLIST	145	4	1
SSTATECL	130	1	1	STLATOR1	146	4	2
SPRINT	131	2(i)	GLOBAL	STLATOR2	147	4	2
SPUNXH	132	2(i)	GLOBAL	STLATOR3	148	4	2
SREID	133	2(i)	GLOBAL	STLATOR4	149	4	2
SCOMMAND	134	2(ii)	1	STLATOR5	150	4	2
SOUTPUT	135	2(ii)	1	SCYLIC	151	5	3
SREADC	136	2(ii)	1	SREFLECT	152	5	3
SREADT	137	2(ii)	1	SXCHANGE	153	5	3
SACTION	138	3	1	SNORMAL	154	6	2
SANPAKC	139	3	1	SRESULT	155	7	2
SANPAKD	140	3	1	SSEARCH	156	7	1
SANPAKJ	141	3	1	SMEMORY	157	8	GLOBAL
SANPAB	142	3	1	SMATCH	158	9	2
SNUPAK	143	4	2	SMOBILE	159	9	2
SGENITEM	144	4	2				

The thirty-one components of the SOLID System (see Table 5) are identified in nine classes or categories. Two (OPENSHTUT and SSTATECL) may be viewed primarily as components which initialize the SOLID System. One of these (OPENSHTUT) opens and closes the peripheral device (except virtual memory). Three (SPRINT, SPUNXH, and SREID) of the seven I/O components perform the basic operations of reading or punching cards, and printing. The other four components (SCOMMAND, SOUTPUT, SREADC and SREADT) are used to communicate with the user. They, together with components SJOBLIST and SRESULT, handle the input and output for the SOLID System and its subsystems.

The six components in Class 3 are exclusively associated with the COPAK compressor subsystem. One of these six (SACTION), which is called from the macro-instruction DEVICES (see Section A), is executed when a decompression error is detected. Another two components, SANPAKD and SNUPAK, are the special forms of SANPAKJ and SNUPAK that are used in the stand-alone alphanumeric and numeric compressors.

The TRANSLATION PACKAGE contains seven components (Class 4). The supervisory component, SJOBLIST, also reads the translation Package commands from cards and information about descriptor-sets. The TRANSFORMATION (Class 5) and NORMALIZATION (Class 6) packages contain three and one components respectively.

The RETRIEVAL PACKAGE (Class 7) contains two components, SRESULT and SSEARCH. The component SRESULT prints information after each use of the SSEARCH component. In production situations SRESULT can be omitted.

The GLOBAL MEMORY (Class 8) and MOBILE CANONICALIZATION (Class 9) packages contain one and two components respectively.

In the remainder of this section, C, short descriptions of the 31 components are given.

1. Initializing Components

122. OPENSHTUT-(OPENSHTUT)

This level 0 component is always compiled in the main stem (viz. CONTROL ROUTINE). It is positioned by the "SUBMP" type instruction (see Section A). It opens (at the beginning of each jobstream) and closes (during termination) the DCB's specified in the INOUT macro-instruction. All peripheral devices, other than those in the virtual memory, must be specified in INOUT and OPENSHTUT. OPENSHTUT, which is called in the macro-instruction DEVICES, uses registers 8 and 9 for USING and Branching.

123. SSTATECL-(SSTATECL &NAME,&UR,&RR,&ADDL<HAYY,&DUMMY)

SSTATECL is called from the RESERVE macro-instruction at the start of each job-stream. It initializes addresses used in SSEARCH and generates or reads (from cards) the permanently resident part of the AUXILIARY FILE, which contains the array associated with the prime index, M, and the screen J(=m₁,

m₂m₃-m_M). If the first card of the data-deck is NEW-FILE then the MJARRY macro-instruction generates the M-J arrays.

LEVEL=1

&NAME is STATECL

&UR can be registers 8 or 9

&RR can be any register except &UR, 10, 11 or 12.

&ADDL is the number of bytes in the composite addresses which preface a memory-block.

<HAYY is the number of bytes in the principal data-array, YY.

2. I/O Components

SPRINT, SPUNXH and SREID are the basic card/printer I/O components for the SOLID System. SCOMMAND reads (from cards) both the device and string commands. SOUTPUT is the principal output component. It handles the compressed and decompressed referenced information. SREADC and SREADT read in the referenced information that is to be compressed or decompressed.

i. Basic Components

There are two output (SPRINT and SPUNXH) and one input (SREID) components in the SOLID System which are used to print, punch cards and read cards. These components, which are called in the SOUTPUT and SREAD components, can be used (with the PRINT, PUNXH, and REIDX macro-instructions) on a stand-alone basis. They are extremely versatile, and can be used anywhere in the SOLID System.

124. SPRINT-(SPRINT &NAME,&UR,&RR,&DUMMY)
the PRINT service-macro calls the component SPRINT,
which prints the requested information in the
designated format(s) on the printer. The DCB, PRINT,
is specified in the INOUT macro-instruction. 5

Level=GLOBAL

&NAME is PRINT

&UR must be register 9; &RR can be any register except
9-12.

125. SPUNXH-(SPUNXH &NAME,&UR,&RR,&DUM-
MY) 10

This component is called by PUNXH (see Section B).

The DCB, PUNXH, is specified in INOUT (see Section
A).

Level=GLOBAL

&NAME is PUNXH.

&UR must be register 9.

&RR can be any register except 9, 10, 11, or 12.

126. SREID-(SREID &NAME,&UR,&RR,&DUMMY) 20

This component is called by the eight REIDX (X=1,2,-8)
service-macros. It reads information from cards with
the DCB named MASTER (specified in INOUT).

Level=GLOBAL

&NAME is REID.

&UR must be 9

&RR can be any register except 9, 10, 11 or 12.

ii. Special Components

127. SCOMMAND-(SCOMMAND
&NAME,&UR,&RR,&DUMMY) 30

This component has two parts, both called from the
CONTROL routines. The first part (&NAME=CO-
MANDD) reads the device commands from cards
(calling instruction is DEVICE). The second part
(&NAME=COMANDS) is called by STRING, it reads
the string commands. The default options for all input
commands are set in SCOMMAND.

Level=1

&NAME=COMMAND, COMANDD or COMANDS.

&UR must be 8

&RR can be any register except 9, 10, 11 or 12.

128. SOUTPUT-(SOUTPUT
&NAME,&UR,&RR,&DUMMY) 45

This output package for the COPAK compressor prints,
punches, or writes on tape the information strings that
are processed by COPAK. A format code, which is
stored with the compressed string of information, is
used to print or punch the output in the entry format
type.

Level=1

&NAME is OUTPUT

&UR can be register 8 or 9

&RR can be any register except &UR, 10, 11, or 12.

129. SREADC-(SREADC &NAME,&UR,&RR,&DUM-
MY) 50

The SREADC component reads control information and
the substrings of data on cards that are processed by
COPAK. The PCORDS table is read and checked in
SREADC. The status-of-substring control words (SOS)
are modified and the status-of-segment control word
(PARM) is constructed. The input commands are
printed at the end of the SREADC component.

Level=1.

&NAME is READC.

&UR is register 8 or 9.

&RR can be any register except &UR, 10, 11, or 12.

130. SREADT-(SREADT &NAME,&UR,&RR,&DUM-
MY) 70

This component reads the compressed and/or decom-
pressed information from magnetic tape. Compressed
information is read with the DCB named TAPEIND
(DDNAME=COPAK4) and uncompressed informa-
tion is read with the DCB named TAPEINC (DD-

NAME=COPAK5). DCB's are specified in the INOUT
macro-instruction.

Level=1

&NAME is READT

&UR can be either 8 or 9.

&RR can be any register except &UR, 10, 11 or 12.

3. Compressor

The stand-alone numeric (COPAKNU), alphanumeric
(COPAKAN), and combined (COPACOKO) Compressors
use different combinations of the six compressor components.
The combined compressor, which is also used in the CON-
TROL routines SOLIDE and SOLIDO, consists of two
alphanumeric (SANPAKC and SANPAKD) and one numeric
(SNUPAK) component. COPAKAN contains SANPAKC and
SANPAKJ, which is a modified form of SANPAKD. COPAK-
NU contains the modified form of SNUPAK, which is called
SNUPAB. The SACTION component is called from the main
stem whenever decompression errors occur.

131. SACTION-(SACTION &NAME,&UR,&RR,&DUM-
MY) 25

This component is called from the macro-instruction
DEVICES in the "Reserve" type instruction (see Sec-
tion A). Control passes to SACTION from the decom-
pressor parts of the compressors whenever an error is
found. Currently SACTION prints appropriate error
messages and terminates the job-stream. Error correct-
ing procedures and/or retransmission requests should
be handled by SACTION.

Level=1

&NAME is ACTION

&UR can be any register other than 0, 1, or 10-15.

&RR can be any register other than &UR, 10, 11 or 12.

132. SANPAKC-(SANPAKC
&SNAME,&UR,&RR,&DUMMY) 35

This component compresses the strings of information by two
recursive bit-pattern methods in one of two anodes. In the
SLOW-MODE the recursive bit-patterns that are used are ob-
tained from the string itself, and those bit patterns which yield
savings are stored in a table, PCORDS. In the FAST-MODE
only those bit-patterns in the PCORDS table are used. The
three input commands associated with SANPAKC (LEX-
CON, LEXMODE, AND LEXPCB) provide the following op-
tions:

- i. Build a new PCORDS by compressing the first X
strings of information in the SLOW-MODE, then
process all subsequent strings in the FAST-MODE.
- ii. Read in PCORDS and operate exclusively in the
FAST-MODE.
- iii. Extend the PCORDS table by processing the first X
strings in the SLOW-MODE and then switch to the
FAST-MODE.

Level=1.

&NAME is the relocatable address ANPAKC.

&UR — the base or using register — can be 8 or 9.

&RR — the branch register — can be any register except
&UR, 10, 11 or 12. In our programs &RR=1.

&DUMMY is discussed above.

133. SANPAKD-(SANPAKD
&NAME,&UR,&RR,&DUMMY) 65

This component first decompresses strings compressed by
SANPAKC then disassembles its decompressed strings
for processing by the decompressor part of the numeric
compression package (SNUPAK). Since all the infor-
mation that is needed to decompress the strings is
stored in the strings themselves, no additional data is
needed.

Level=1.

&NAME is the relocatable address ANPAKD.

The restrictions on &UR, &RR and &DUMMY for SAN-
PAKC apply for SANPAKD also.

134. SANPAKJ-(SANPAKJ &NAME,&UR,&RR,&DUM-
MY) 75

SANPAKJ is the modified form of SANPAKD that is used in the stand-alone alphanumeric compressors (COPAKAN). It contains the macro-instruction JIMPI, which performs those substring operations which are normally executed in SNUPAK.

Level=1

&NAME is the relocatable address ANPAKD.

&UR,&RR, and &DUMMY have the specifications given for SANPAKC and SANPAKD.

135. SNUPAB-(SNUPAB &NAME,&UR,&RR,&DUM- 10
MY)

SNUPAB is a modified form of SNUPAK, that is used in the stand-alone numeric compressor (COPAKNU). SNUPAB contains the macro-instruction STRINGD, which is normally executed at the end of SANPAKD.

Level=1.

&NAME is the relocatable address NUPAK.

&UR can be 8 or 9

&RR can be any register except &UR, 10, 11 or 12.

&DUMMY is set equal to SOLID if SNUPAB is compiled in the main-stem (viz. extended form). It is set equal to DUMMY if SNUPAB is compiled separately as a named CSECT.

136. SNUPAK-(SNUPAK &NAME,&UR,&RR,&DUM- 25
MY)

SNUPAK is the numeric compressor-decompressor package in COPAK. It processes strings of information one substring at a time. Compression is accomplished by the four step procedure: truncation, differencing, sequencing, and packing. There are three truncation methods, two of which are automatic. If savings cannot be achieved compression is terminated without loss of information.

Level=1

&NAME is NUPAK.

&UR can be register 8 to 9.

&RR can be any register except &UR, 10, 11, or 12.

4. Translation Package

The TRANSLATION PACKAGE consists of seven components and the NORMALIZATION PACKAGE, which uses the TRANSFORMATION PACKAGE. One of the seven components, SJOBLIST, can be regarded as the supervisory routine for the entire Translation Package. The macro-instruction GETJLIST calls SJOBLIST from the CONTROL routines SOLIDE AND SOLIDO.

SJOBLIST is also the input component for the Translation Package. It reads the Translation Package commands; generates or reads descriptor-sets; reads over-ride information; rearranges descriptor-sets to the JOBLIST item form; and normalizes the JOBLIST items. Random JOBLIST items are produced by the component SGENITEM. Five TRANSLATOR components (STLATORX, with X=1,2,-,5), which are called by the special service macro TRANSLATE, convert the descriptor-sets to their JOBLIST item form. There are provisions for incorporating up to 255 TRANSLATORS. JOBLIST item are converted to their NORMAL FORMS by the Normalization Package, which is called in SJOBLIST by the special service macro NORMFORM.

The TRANSLATION PACKAGE components are briefly described next:

137. SGENITEM-(SGENITEM 65
&NAME,&UR,&RR,&DUMMY)

This component generates JOBLIST items in the array stipulated by its calling procedure (JLITEM), which appears in the component SJOBLIST. SGENITEM uses the information in registers 2, 3, 4 and 5 that were loaded in JLITEM.

Level=2

&NAME is the relocatable address GENITEM.

&UR can be register 8 or 9.

&RR can be any register except 2, 3, 4, 5, &UR, 10, 11 or 12

&DUMMY has been defined above.

138. SJOBLIST-(SJOBLIST &NAME,&UR,&RR,&DUM-
MY)

SJOBLIST is the supervisory component for the TRANSLATION PACKAGE. Its functions have been briefly described above.

Level=1.

&NAME is JOBLIST

&UR can be either 8 or 9

&RR can be any register except &UR, 10, 11, or 12.

139.

to

STLATORX, with X=1,2,3,4, or 5

143.

There are five TRANSLATOR components which have prototype statements like: STLATOR1 &NAME,&UR,&RR,&DUMMY. The first, STLATOR1, is reserved for the AGISAR Translator, which rearranges automatically, extracted data from N-dimensional graphs (or pictures) to the JOBLIST form. New Translators must be coded for each new collection of items. There are provisions in the special service macro TRANLATE, which appears in component SJOBLIST, for incorporating up to 255 Translators.

Level=3.

&NAME is TLATOR1 or TLATOR2 or TLATOR3 or TLATOR4 or TLATOR5.

&UR can be 8 or 9.

&RR can be any register except &UR, 10, 11 or 12.

5. Transformation Package

35 The TRANSFORMATION PACKAGE consists of three components whose design purposes are to execute the CYCLIC SHIFT, REFLECTION, and the INTERCHANGE Transformation Rules. These components can be used by both the NORMALIZATION and MOBILE CANONICALIZATION packages.

144. SCYCLIC-(SCYCLIC &NAME,&UR,&RR,&DUM-
MY)

The CYCLIC component will execute both the left and right cycle shifts. Entry information needed in CYCLIC is defined in the component SNORMAL.

Level=3

&NAME is CYCLIC

&UR can be 8 or 9

&RR may be any register except &UR, 10, 11 or 12.

145. SREFLECT-(SREFLECT
&NAME,&UR,&RR,&DUMMY)

This component can be called from SNORMAL or SMOBILE. It will execute the Reflection Rule.

Level=3

&NAME is REFLECT

&UR, &RR and &DUMMY are the same as for component SCYCLIC

146. SXCHANGE-(SXCHANGE
&NAME,&UR,&RR,&DUMMY)

SXCHANGE executes the kernel Interchange Rule.

Level=3

&NAME is XCHANGE

&UR, &RR and &DUMMY have been specified for SCYCLIC.

6. Normalization Package

The NORMALIZATION PACKAGE contains one component, SNORMAL, which is called by the NORMFORM macro-instruction in component SJOBLIST. SNORMAL can use the TRANSFORMATION PACKAGE (see above) to obtain the NORMAL FORMS of JOBLIST items that are produced by the TRANSLATORS.

147. SNORMAL-(SNORMAL
&NAME,&UR,&RR,&DUMMY)

This component is called by NORMFORM in the component SJOBLIST after the TRANSLATORS have been executed. Full details of its assigned role in the SOLID System are given in the publication by P.A.D. deMaine and B.A. Marron, mentioned earlier.

Level=2.

&NAME is NORMAL

&UR may be 8 or 9

&RR can be any register except &UR, 10, 11, or 12.

7. Retrieval Package

The RETRIEVAL PACKAGE contains two components, SRESULT and SSEARCH, and it uses the GLOBAL MEMORY and MOBILE CANONICALIZATION PACKAGE, which uses the TRANSFORMATION PACKAGE. In its present form the Retrieval Package can handle explicit retrieval and storage questions. With minor changes, in the MMATCH and AUXFILE macro-instructions, it would handle the explicit purging and updating tasks also. The fully implemented retrieval package is able to handle any kind of explicit, implied (or non-explicit), or browsing question.

The SSEARCH component and the GLOBAL MEMORY and MOBILE CANONICALIZATION PACKAGES are extensively interrelated together. They must be regarded as the Central Core of the SOLID Retrieval System. The Global Memory Component (SMEMORY), which is called from the TBADD macro-instruction in SSEARCH, transfers the memory-blocks between core-storage and the preselected storage devices. The SSEARCH component automatically traces and/or creates the information paths in the resident memory-block. The MOBILE CANONICALIZATION PACKAGE, which is called from the MMATCH macro-instruction, is used in retrieval operations if override codes are present. It makes possible the automatic implied (or non-explicit), fragment, or browsing searches. The SRESULT component, which is called from the CONTROL routine after completion of a search, prints results obtained by the SSEARCH component. SRESULT can be changed to collect statistics on the performance of the retrieval system. Hard-copy of the stored bulk-information is produced by the principal output component, SOUTPUT.

The complex independence of SSEARCH and the two packages mentioned are described elsewhere in this disclosure.

148. SRESULT-(SRESULT &NAME,&UR,&RR,&L-SLOW,&JBLIST,&DUMMY)

The SRESULT component is called from the CONTROL routine after the SSEARCH component has been executed. In its present form it constructs and prints an obvious message like: REQUESTED INFORMATION APPEARS ON PRINTER. It also prints the request JOB-LIST item and the BULK address(es) assigned or retrieved in RFILE. These address(es) are normally the location of the compressed referenced information in the bulk storage. SRESULT can be modified to collect and analyze performance data for the SOLID System.

Level=2

&NAME is RESULT.

&UR is register 8

&RR can be any register except &UR, 10, 11, and 12

&SLOW is the length of the slow position of composite addresses.

&JBLIST(=JBLIST) is the Joblist array whose address is in AJBLIST

&DUMMY has been defined.

149. SSEARCH-(SSEARCH &NAME,&UR1,&UR2 &RR,&ADDL,&LSLOW,&LFAST,NT RKS,&TRKL,<HAYY,&JBLIST,&LB JLIST,&MATRIXL,&MATRIXS,&DUMMY)

This component used the information in the JOB-LIST item(s) to retrieve (MODE=0), store (MODE=1), update or purge (MODE=2,3 or 4) items in the AUX-

ILIARY FILE and compressed referenced information in BULK STORAGE. Except for providing the retrieval command, MODE all operations are fully automatic. Thus information sub-paths are automatically traced (MODE=0, 2, and 3) and/or created (MODE=1, 2 or 3) and/or purged (MODE=2 and 3) in fast memory with the JOB-LIST item information. The Global Memory component (SMEMORY) ensures that the correct memory-block is resident in core-storage. SMEMORY also updates the memory-blocks of the AUXILIARY FILE in the virtual memory (see SMEMORY, above). Protection feature in SSEARCH ensure that the AUXILIARY FILE (in virtual memory) will never be altered by coding, operator of machine errors.

The MOBILE CANONICALIZATION PACKAGE, which handles implicit and intersecting file questions, is called from the Service-Macro MMATCH.

Purging and updating operations in the AUXILIARY FILE will be executed in MMATCH. Thus the basic form of the SSEARCH component will never be altered.

Level=1

&NAME - the relocatable address is SEARCH.

&UR1 and &UR2 are the two USING registers (7 and 8) which establish addressability in the SSEARCH component.

&RR - the branch register - can be any register except 7, 8, 10, 11 or 12).

&DUMMY is defined above. The ten System Parameters were defined earlier in this disclosure.

8. Global-Memory

The GLOBAL MEMORY PACKAGE consists of one component (SMEMORY) and its calling procedure (GLOBAL), which has already been discussed. The DCB or format statement(s) and the opening and closing instructions for the Global Memory are specified in the macro-instruction DCB-MEM, which is found at the end of component SMEMORY. If more storage is allocated then DCBMEM must be altered and the macro BEGINS, which is used in SSEARCH, must be changed. The Global Memory Package is fully described elsewhere in this disclosure.

150. SMEMORY-(SMEMORY &NAME,&UR1,&UR2,&RR,&ADDL,&INTRKS&TRKL,&JBLIST,&DUMMY)

This component must be positioned by the SUBMP macro-instruction in the CONTROL routine at compilation time. SMEMORY supervises the transfer of the memory-blocks (in the AUXILIARY FILE) between core storage and the designated virtual memory devices. The two parts of SMEMORY accomplish the following:

Part A: (Relocatable Address MEMORY) This part is entered from the TBADD macro-instruction in the SSEARCH component if a new memory-block is needed. If necessary the currently resident memory-block is rewritten at its assigned location (in virtual memory) and then the requested memory-block is fetched from virtual memory. New memory-blocks, which can be created in core-storage, are automatically assigned storage areas in virtual memory).

Part B: (Relocatable address SAVEFM) This part is entered immediately before the Operating System of the 360 regains control to terminate the job. If the resident memory-block has been altered in any storage, updating or purging operation it is stored at the assigned location in virtual memory. It is suggested that the IBM O/S job terminating routine be modified to include a final call to Part B.

The service macro DCBMEM, which appears at the end of SMEMORY, specifies the DCB, OPEN and CLOSE instructions for SMEMORY. The macro-instruction BEGINS, which is used in SSEARCH, specifies the device numbers.

Level=GLOBAL
 &NAME is MEMORY.
 &UR1 and &UR2 are two base registers (7 and 8).
 &RR — the branch register - can be any register except 7,
 8, 9, 10, 11 or 12. 5
 &ADDL is the length of the composite addresses.
 &ENTRKS is the number of tracks (or records) in a
 memory-block.
 &TRKL is the length of the track (or record).
 &JBLIST(=JBLIST) is the Joblist array whose address is 10
 in location AJBLIST.
 &DUMMY is defined above.

9. Mobile Canonicalization Package

The MOBILE CANONICALIZATION PACKAGE consists 15
 of a service macro (STRATEGY), which appears in
 MMATCH, and two components, SMATCH and SMOBILE.
 The component SMOBILE will use the TRANSFORMATION
 PACKAGE to rearrange previously normalized JOBLIST
 items that contain override codes. The component SMATCH 20
 will determine if mismatches occur only because overrides are
 present. SMATCH and SMOBILE will also perform the inter-
 secting type of search.

151. SMATCH-(SMATCH &NAME,&UR,&RR,&DUM-
 MY) 25

The information needed by the SMATCH macro has
 been given in the macro-instruction STRATEGY (see
 Appendix).

Level=2

&NAME is MATCH

&UR can be 7,8, or 9 30

&RR can be any register except registers 7-12.

152. SMOBILE-(SMOBILE &NAME,&UR,&RR,&DUM-
 MY) 25

SMOBILE is executed after SMATCH and then only if a
 mismatch could not be resolved. Information required
 by SMOBILE is stipulated in the macro STRATEGY
 (see Appendix).

Level=2

&NAME is MOBILE

&UR can be 7, 8 or 9. 30

&RR can be any register except registers 7-12

&DUMMY is defined above.

RETRIEVAL PACKAGE

Overview:

The AUXILIARY FILE can be viewed as a maze or net-
 work that automatically grows, contracts, or is modified to ex-
 actly fill the indexing needs for each and every application of
 the SOLID Retrieval System. Each path through the maze is
 unique and terminates in a location which contains the ad-
 dress in bulk storage where the compressed referenced informa-
 tion is stored. The JOBLIST items, which are produced by
 the Translation Packages from the assigned descriptor-sets,
 are used to trace or create the subpaths that together define an
 information path. New subpaths are created only when they
 are needed during a storage or updating assignment. Subpaths
 are eliminated during purging and in some updating opera-
 tions. The "length" of a path (or search) is determined solely
 by the number of decisions that are made while tracing the
 path, not by the operation (i.e. storage or retrieval or purging
 or updating) that will be performed at the bulk storage ad-
 dress. 50

In many respects this scheme is analogous to a telephone
 network. Each telephone number can be viewed as a unique
 description (viz. JOBLIST item) of a path from the sub-
 scriber's substation to another substation in the network. A
 telephone call made from the subscriber's substation will be
 aborted if any link (i.e. subpath) of the path does not exist. In
 the SOLID System the prime index, M, and the screen, J are
 analogous to an "area code", and the other screens are
 descriptions of the intermediate substations that are to be 55

linked for the telephone call. The analogy with a telephone
 network breaks down when the following facets of the SOLID
 System are considered.

- Unlike telephone numbers, which are somewhat arbitrar-
 ily assigned to each subscriber, the JOBLIST items ac-
 tually describe both the path and the referenced informa-
 tion. This means that assignment of "idiot numbers", like
 those in the National Compound Registry, are quite un-
 necessary.
- When the proposed new components are implemented,
 the Retrieval Package will have a capability for "b-
 rowsing", which has no parallel in telephone networks.
- Unlike telephone networks, whose new substations must
 be created at quite rigidly prescribed locations, the
 SOLID System creates new paths or substations wherever
 storage is available.

The AUXILIARY FILE is divided into two parts. One of
 these parts, which resides permanently in the computer, is as-
 sociated with the prime index M and the screens J. The second
 part is divided into memory-blocks and is stored in virtual
 memory. The Retrieval Package uses the JOBLIST items
 produced by the Translation Package and a single input com-
 mand, MODE, to automatically execute all tracing, creating
 and purging operations in core storage. A global memory
 component, SMEMORY, transfers the memory blocks
 between virtual memory and core storage when they are
 needed. The Continuance Tables will be used to restrict all
 paths within single memory-blocks. This will ensure that each
 explicit storage, purge, retrieval, or update request can be ex-
 ecuted with, at most, the transfer of one memory-block.

AUXILIARY FILE

It has already been noted that the AUXILIARY FILE is di-
 vided into two parts. Part A, which resides in core storage, is
 generated by the macro MJARRAY or it is read from cards by
 the SSTATECL component when the file is initialized. Part B
 is divided into memory blocks that are stored in fast-slow or
 visual memory. 35

The principal data array (YY) is divided into three portions
 as follows:

- The first portion contains that part of the AUXILIARY
 FILE which will reside permanently in core storage (i.e.,
 Part A below). 40
- The second, transient portion must be large enough to
 hold one memory-block.
- The third portion is used to manipulate the strings of
 referenced information that are stored or retrieved in the
 bulk storage. 45

Memory blocks are transferred to the transient portion of
 core storage by the global memory component (SMEMORY)
 whenever they are needed for tracing or creating new "infor-
 mation paths". The two parts of the AUXILIARY FILE are
 discussed next: 50

Part A:

Part A contains one sub-array associated with the prime in-
 dex, M, and five subarrays which are associated with the
 screen J. It is prefaced by two composite addresses, EMPTY
 and BULK. The first four items in EMPTY together give the
 location in virtual memory where the next newly created
 memory block is to be stored. The fast memory address por-
 tion of EMPTY (FMADD) contains the beginning address of
 the transient portion of the data array, YY. The first four
 items in BULK together give the location in bulk storage
 where compressed referenced information is stored. The fast
 memory address portion of BULK (FMADD) specifies the
 core-location of the referenced information. 55

Normally, the first input item for the SOLID System is a
 card deck with Part A punched in column binary. When the
 SOLID System is used for the first time, this card-deck is
 replaced by a single card that contains the word NEWFILE. 60

This generates the initializing information for Part A. Thereafter, if part A was changed, a new card deck is punched at the end of each job-stream. The information on the first two cards and last two cards is used to check the card deck at input time.

Part B:

Each of the memory-blocks is prefaced by a composite address, CURRENT. The first four items in CURRENT disclose the location in virtual memory where the memory-block normally resides. The fifth item, FMADD, is the relative address in the principal data array (YY) where a new sub-array can be created. Thus FMADD is the location of the first byte in the resident memory-block that is not a part of an existing sub-array.

Description:

The translated JOBLIST item which is stored in the array &ARRAY, is used by the Retrieval Package to trace (retrieval and old storage) or create (new storage) the information path in the AUXILIARY FILE. This is accomplished with the aid of three addresses (EMPTY, CURRENT, and ADDRESS) and eight bit indicators in a single byte (MSIGNAL). The composite addresses EMPTY and CURRENT initially preface Part A and the resident memory block of the AUXILIARY FILE respectively. ADDRESS is extracted from a subarray during the search. It is either the link address, pointing to an extension or continuance of the subarray, or an address extracted from an EXECUTIVE POINTER. The position of the EXECUTIVE POINTER will disclose the prime index or screen in the JOBLIST item. If a subpath is missing, ADDRESS contains zero. In this case the MMATCH macro instruction either completes the construction of a new EXECUTIVE POINTER, thus creating a new subpath, or it aborts the search.

The eight bits of MSIGNAL are used by the retrieval package to indicate the status of the AUXILIARY FILE with respect to the current search. The signal system is discussed next.

Signal System (MSIGNAL):

The meanings that are assigned to each of the eight bits in MSIGNAL are given next.

MSIGNAL BIT ON (HEXA-DECIMAL)	Instruction
80	A new memory-block is to be created.
40	The search component has been used before.
20	ADDRESS contains a link address.
10	Tracing with screen J has been completed.
08	Tracing with index M has been completed.
04	A new subarray is to be created.
02	The resident memory-block is new.
01	The resident memory-block has been changed.

The 40 bit is turned off when the system is initialized. This occurs at the start of each job-stream in the SSTATECL component after the card-deck of Part A, has been read. Three bits (01, 02 and 80) are used by the Global Memory component (SMEMORY) to save the resident memory-block and to fetch a new one. Two bits (04 and 20) indicates the status of the sub-array whose beginning address is in ADDRESS. The last two bits (08 and 10) are used to indicate the type of executive pointer (e.g., with or without screen) in the subarray that is to be searched next.

The role played by MSIGNAL is shown in FIG. 16. A step-by-step description follows:

Step a

At the start of the job-stream the bits in MSIGNAL are turned off. This occurs in the SSTATECL component at stage 600. If CURRENT=FFFFFFFF, bit 80 is turned on. This means that no memory blocks are present in the AUXILIARY FILE.

Step b

All bits in MSIGNAL except 80, 40, 02 and 01 are turned off at stage 602. The index registers which point to M in the JOBLIST item (in array &ARRAY) and to the sub-array associated with M are initialized, and at stage 602, the composite address EMPTY is saved at CORD1. If the 40 bit is off, this address will also be stored in location EMPTY+&ADDL and the 40 bit turned on. If EMPTY and EMPTY+&ADDL do not contain the same address at the end of the job-stream a new card deck will be punched for Part A. Step b at 604 is the normal entry point to the retrieval package.

Step c

At stage 606 the 20 bit in MSIGNAL, which is used indicated that an extension or link sub-array is to be fetched or created, is turned off.

Step d

Bit 08 is inspected at stage 608. If it is one, then either the JOBLIST item index points to a screen, or a search of an RFILE array, which contains bulk storage address(es) of compressed referenced information, is indicated. The length of the unused part of the JOBLIST item is used at stage 612 to differentiate these situations. If the MSIGNAL 08 bit is zero, then the JOBLIST item index points at M, and an index search of the subarray MA is completed at stage 610.

Step e

The sub-array in core storage is searched at stage 614 for an EXECUTIVE POINTER which contains the screen in the JOBLIST item. The following situations can occur:

1. An EXECUTIVE POINTER is found. The address portion is loaded into ADDRESS and control goes to MMATCH at stage 616. In this case, MSIGNAL is not altered.

2. The subarray is full, so one of its extensions or continuances must be fetched or created. In this case the "-continuance bit", 20, is turned on and the link address is loaded in ADDRESS.
3. An EXECUTIVE POINTER which contains the screen in the JOBLIST item cannot be found in the sub-array or in its extension(s). If MODE=0 (i.e., retrieval) ADDRESS is set to zero. In the storage or updating modes (MODE>0) a hole is made at the correct spot in the subarray and the screen is stored in it (left adjusted). ADDRESS is set to zero.

Step f

In the MMATCH macro instruction at stage 616, ADDRESS is compared to zero. If it is zero and MODE=0 (i.e. retrieval) the search is aborted as unsuccessful at stage 618. It should be noted that the MOBILE CANONCALIZATION PACKAGE is called by the macro STRATEGY from MMATCH whenever a mismatch (i.e. ADDRESS is zero) occurs in the retrieval mode and there are over-rides present. If MODE=1 (i.e. storage), the composite address CURRENT is stored in the correct location in the subarray, and both the 01 and 04 bits in MSIGNAL are turned on. This completes the construction of the new EXECUTIVE POINTER and also indicates that the resident memory-block has been changed. If ADDRESS is not zero the next subpath has been found and MSIGNAL is not changed.

Step g

TBADD at stage 620 is the most complex and powerful instruction in the Retrieval Package. It uses the information in the composite addresses CURRENT and ADDRESS plus bits 01, 02, 04 and 80 in MSIGNAL to accomplish the following:

1. If a new memory block is required, then the global memory component (SMEMORY at stage 624) saves, if necessary, the resident memory block and fetches a new one. Bits 01, 02, and 80 in MSIGNAL are used by SMEMORY.
2. If the 04 bit is "on", the CREATE macro instruction at 622 creates a new subarray, beginning in the location specified by the core address portion of ADDRESS. The composite address CURRENT is recomputed at stage 626 and, if necessary, the address EMPTY is recomputed also. The create bit (04) is turned off at the end of CREATE.

Step h

At stage 626 the continuance bit 20 of MSIGNAL indicates whether or not the JOBLIST item index is to be incremented and bits 10 and 08 are to be altered. This procedure is executed until either the search is aborted (in MMATCH) or the search ends successfully at stage 682 after retrieval or insertion of the bulk storage addresses in the RFILE subarray of the resident memory-block. These tasks are performed by the AUXFILE macro instruction at stage 632. It turns on MSIGNAL 01 bit if a bulk storage address is inserted in RFILE. The 20 bit is turned on if a continuance of the RFILE subarray has to be fetched or created.

After a successful search, the bulk storage address(es) are used to store (MODE=1), update (MODE=4), or retrieve (MODE=0) the compressed referenced information in the MAINFILE. The retrieved referenced information is decompressed by COPAK before it is disseminated.

Search Procedures:

The subarrays in core-storage are searched for EXECUTIVE POINTERS in one of three different ways (see FIG. 16). These are:

Indexes

The EXECUTIVE POINTERS that are associated with the prime index (M) are stored in the subarray at the relative address indicated by the M value. A zero at the relative address means that no EXECUTIVE POINTER has been inserted.

RFILE

The bulk storage address is stored in the first vacant element of the RFILE sub-array or its extension (or continuances). A single macro-instruction, AUXFILE, is responsible for searching and maintaining the RFILE sub-arrays. Retrieved or newly assigned bulk storage addresses are stored in the high address end of the principal data array, YY. The beginning address is found in location SBRY. The number of retrieved or stored addresses is in location NJOBS. An unsuccessful retrieval is indicated by setting JII to zero.

The RFILE subarrays are created at the first available location in core-storage when they are needed. The number of bulk storage addresses that can be stored in a particular RFILE sub-array is determined by the System Parameter &MATRIXS.

The AUXFILE macro-instruction is executed at stage 630 of FIG. 16 whenever the residual or unused length of the JOBLIST item is less than zero. This occurs after the beginning address of the terminal RFILE array has been found. AUXFILE uses the retrieval command, MODE, and, if need be, the bulk storage address (BULK), to retrieve or store addresses of the compressed referenced information in RFILE.

The retrieval command, MODE, has been assigned the following five meanings:

MODE	MEANING
0	Retrieve
1	Store or Update
2	Change items in the AUXILIARY FILE
3	Purge paths from the AUXILIARY FILE
4	Purge, replace and add compressed referenced information to the MAINFILE.

The AUXFILE these five commands mean that bulk storage addresses are to be retrieved, stored, purged or changed in RFILE. The three types of override codes (Type 1, Type 2, and Type 3), which are automatically inserted in the JOBLIST items during the translation of descriptor-sets, have different meanings for each of the five values of MODE. These are as follows:

MODE	Override	Meaning of Over-ride
60	0	Accept any non-zero value for the designated descriptor or element
	2	Accept any value, zero or non-zero, for the designated element.
	3	Accept any value for the designated descriptor that lies in the range specified in array AOVER3R.
70	1	Create normal paths with the specified override. If such paths exist the inverted file searches will not be executed during retrieval.
	2	
2	1	Replace specified element(s) by a "1".

- 2 Replace specified element(s) by a "2".
- 3 Replace specified element(s) by a "3".
(For the MODE=2 update the Normalization Package will not be used).
- 3 Purge the information path specified in JOBLIST from the AUXILIARY FILE.
- 2 Replace the element that is specified in the JOBLIST item by the element in the array AOVER3R.
- 3 Use the information in array AOVER3R to construct alternate paths for the path described by the JOBLIST item.
- 4 Purge the referenced information items whose bulk storage address(es) are given in array AOVER1 from the MAIN FILE. Add the compressed referenced information whose storage addresses are given in array AOVER2. Replace the items in MAINFILE as specified in array AOVER3.

In its present form the SOLID System can process explicit retrieval (MODE=0) and storage (MODE=1) requests which do not use the NORMALIZATION or MOBILE CANONICALIZATION PACKAGES. The full potential of the SOLID System, which permits the use of all five MODE options and overrides, include utilization of the NORMALIZATION and MOBILE CANONICALIZATION PACKAGE instructions and the MMATCH and AUXFILE macro-instructions are modified slightly. In the following discussion it will be seen that the AUXFILE macro-instruction has been designed so that branches for the remaining three MODE options (2, 3, and 4) can be very easily incorporated. The flow-chart for AUXFILE (FIG. 17) is discussed next.

In FIG. 17, the operation starts at stage 634 wherein the question is asked "is MODE greater than, equal to, or less than 4?" If the answer is that MODE is greater than 4, then the operation is terminated at stage 636, because meanings have not yet been assigned for MODE greater than 4. If MODE is less than 4, control goes to stage 638. If MODE is equal to 4 control passes to stage 640.

At stage 638, the question asked is "is MODE equal to, less than, or greater than 2?" If MODE is greater than or equal to 2, then, control again goes to stage 640. Stage 640 is operative to handle the conditions when MODE is equal to 2, 3 or 4.

If MODE equals one or zero, the program continues to stage 646 and registers R0 and R1 are loaded with zero and NJOBS respectively.

At this point the significance of two counters, NJOBS and RNJOBS, must be understood. Both counters are initialized in the TRANSLATION PACKAGE, which is executed before the RETRIEVAL PACKAGE is used. For retrieval operations NJOBS and RNJOBS are both set equal to zero. NJOBS are used to count the number of bulk storage addresses retrieved from the RFILE array by AUXFILE. For storage operations both NJOBS and RNJOBS are set equal to the number of items of compressed referenced information that are to be stored for the particular JOBLIST item. Thus, if MODE=1, NJOBS cannot be less than one. NJOBS is incremented (retrieval) or decremented (storage) each time a bulk storage address is retrieved or inserted in the RFILE arrays. The bulk storage addresses are also recorded in a sub-array of the principal data array, YY, that begins at the location stored in SBRY. SBRY is set during the initialization of the SOLID System. NJOBS and RNJOBS are used to compute the actual locations in the sub-array of YY where each bulk storage address is recorded.

In the following discussion of stage 648 it is assumed that this is the first time that stage 646 is executed for a particular JOBLIST item. It will be understood that R1 is incremented (retrieval mode) or decremented (storage mode) in cycles through or within AUXFILE. At stage 648, a determination is made as to whether the system is operating in the retrieval or the storage mode. If it is in the retrieval mode then the program continues to stage 650 and, the location where the bulk storage address is to be recorded is computed in the register R1 from the System Parameter &LSLOW, SBRY, and R1 itself. It should be noted that before this computation R1 contains the number of bulk storage addresses that have been recorded in the subarray of YY to this point. &LSLOW is the length of each bulk storage address.

If, at stage 648, the system is operating in the storage mode, then control goes to stage 652, wherein register R1 is set equal to the difference between RNJOBS and NJOBS. As already noted above this difference (in R1) is the number of bulk storage addresses that have been assigned so far for the particular JOBLIST item. From stage 652 the program goes to stage 650, wherein the new R1 is computed as described above.

After stage 650 has been executed control goes to stage 654, wherein the question asked is "is MODE zero?". At this stage it should be noted that the location of the first element in array RFILE is recorded in register IR6 and that register IR3 contains the location of the continuance address of the RFILE array.

If MODE=0 at stage 654 then control goes to stage 656 and the question is asked "is the element of RFILE specified in register IR6 zero?" If the answer at stage 656 is "it equals zero" control then goes to stage 640 and the exit operations of AUXFILE are executed. If the answer at stage 656 is "it is not zero" then control goes to stage 658. Therein the bulk storage address in the RFILE array is recorded in the subarray YY at the location specified in register R1, and NJOBS is incremented by one. From stage 658 the program branches to stage 660 and register IR6 is incremented by the value &LSLOW, which is the length of elements (viz., Bulk Storage Address(es)), in the RFILE array. At the next stage, 662, the contents of registers IR6 and IR3 are compared. If IR6 equals IR3, control goes to stage 664, wherein the question is asked "is the continuance address of the RFILE array zero?" If the continuance address is zero, control passes to stage 666 where the question asked is "is MODE equal or greater than zero?" If MODE equals zero, the retrieval mode is indicated and control goes to stage 640 to begin the AUXFILE exit operations. If, at stage 666, MODE is greater than zero, then the retrieval mode is indicated, and the program goes to stage 670, where the MSIGNAL 20 bit is turned on. From stage 670 control goes to stage 640, exits from AUXFILE. If, at stage 664, the continuance address is not zero, then the program goes directly to stage 670 as described above. The MMATCH and TBADD macro-instructions use the MSIGNAL 20 bit, which was turned on at stage 670, to fetch or create an extension of the RFILE array. They will be fully discussed hereinafter.

It should be noted that the MSIGNAL 20 bit is interrogated after each use of AUXFILE. If it is "one" control goes to the MMATCH macro-instruction (at stage 616) and the extension of the RFILE array is fetched (retrieval or storage) or created (storage) at stages 620. Eventually control returns to the AUXFILE macro via stages 626, 606, 608 and 612 of FIG. 16.

If, at stage 662, IR6 is greater than IR3, something is wrong, and the program goes to stage 668, where an error message is printed before exiting from the system at stage 636. If IR6 is less than IR3 control returns to the first stage in AUXFILE (634) where the next cycle through AUXFILE begins.

If the answer to the question asked at stage 654 was "MODE is not zero," then the storage mode is indicated and control goes to stage 672. At stage 672 the question asked at stage 656 is again posed. If the answer is "it is not zero" then control goes to stage 660 and the previously described operations are executed. If the answer at stage 672 is "it is zero",

this means that a zero element has been found in the RFILE array, and control goes to stage 674. At stage 674 the new bulk storage address (in BULK) is stored in the RFILE and the YY array, at the locations specified by registers IR6 and R1 respectively. At this point it should be noted that the new bulk storage address that has just been assigned will be used to store compressed referenced information. COPAK will compress the reference information then store it in the MAIN-FILE at the assigned bulk storage addresses.

At the next stage, 676, of the AUXFILE macro, the address BULK is updated by the macro-instruction BULK. Thus the address BULK now contains the next location in the MAIN FILE that can be assigned for storing compressed referenced information. From stage 676 the program goes to stage 678, wherein NJOBS is decremented by one, and the MSIGNAL 01 bit is turned on. It should be noted that at this point NJOBS contains the number of bulk storage addresses that must still be assigned for the JOBLIST item, and the 01 MSIGNAL bit now indicates that the resident memory-block has been changed.

At stage 680 the question is asked "is NJOBS greater than zero?" If the answer is "it is greater than zero" then control goes to stage 660, and the previously described operations are executed. If the answer is "it is not greater than zero," which is identical with the answer "equal zero or negative", then control goes to stage 640.

At stage 640 JII and IR1 are set equal to one and AYY respectively. Later in the SSEARCH component JII=1 indicates that a successful search has been performed and that the new assigned (storage mode) or retrieved (retrieval mode) bulk storage addresses will be found in the YY array, beginning in the location specified in SBRY. At the end of the SSEARCH component the information in this part of array YY is transferred to a work array, JBWORK, and, for the storage mode, NJOBS is set equal to RNJOBS. The information in NJOBS and in array JBWORK are used by the COPAK compressor, which is discussed hereinafter. The register IR1 has been reset to point to the beginning of that part of the AUXILIARY FILE that resides permanently in core-storage.

It should also be noted that in stage 678 the MSIGNAL 01 bit was turned on. This bit will signal the GLOBAL MEMORY component (SMEMORY), when it is called later by the TBADD macro-instruction (see stage 620, FIG. 16), that the resident memory-block must be updated in virtual memory. GLOBAL MEMORY is discussed later in this disclosure.

FIG. 18 is a flow diagram of the program steps in the SCREEN macro-instruction. SCREEN is used (at stage 614 in FIG. 16) whenever the MSIGNAL 08 bit is "on" and the residual length of the JOBLIST item, which is recorded in the half-word DUN1 is not less than zero. These conditions occur when the screens in the JOBLIST item (viz., J, LD₀, BD₁, etc) are being used to trace or create subpaths. Before beginning the discussion about the SCREEN macro instruction, it should be understood that there are two steps involved. First, an array whose EXECUTIVE POINTERS have the same length screen as the one in the JOBLIST item must be located. Second, the located array must be searched for the element where an EXECUTIVE POINTER with the JOBLIST item screen should be located. Of course, the located element may contain zero, the desired EXECUTIVE POINTER, or another EXECUTIVE POINTER. Once the location, where the EXECUTIVE POINTER should be, is found, then a decision can be made about the course of action that should be taken.

In the AUXILIARY FILE all the arrays that are associated with a particular screen (say LD₀) and a particular value for the preceding screen or index (in our case the screen J) are linked one to the other by their continuance addresses. Within each array, all EXECUTIVE POINTERS have the same length screen. However, two different linked arrays can have EXECUTIVE POINTERS with different length screens.

At stage 684 in FIG. 18 the byte JI is set to zero and all registers, R0 to R15, are stored in an array which begins at DUM1+8. These registers are saved because they are needed

if the attempt to locate an array which can be searched is unsuccessful. The byte JI has been initialized for the SUPERSCH macro-instruction, which is executed later in SCREEN, and is fully described hereinafter.

From stage 684 the program goes to stage 686, wherein the length of the EXECUTIVE POINTER, whose screen is in the JOBLIST item, is computed in register R14. Thus R14 contains the screen length plus &ADDL, which is the address length. Also, at stage 686 the total length of all the EXECUTIVE POINTERS in the array that are to be searched is stored in register R1. This length is found in the first four bytes of the array. At the next stage, 688, the question is asked "is register R1 exactly divisible by R14?" If the answer is "no" this means that the array is not to be searched, because the length of its screens differ from the length of the screen in the JOBLIST item. In this case control goes to stage 690 wherein all the registers, R0 to R15, are reloaded from the array DUMX+8, then the program goes to stage 692. At stage 692 the MSIGNAL '20' bit is turned "on" and register IR6 is set equal to register IR3, then the program exits from SCREEN at stage 694. At this point it should be noted that both registers IR3 and IR6 now contain the location of the array's continuance address. This information, in register IR6, and the 20 MSIGNAL bit are used by the MMATCH macro-instruction (Stage 616, FIG. 16) to eventually fetch or create an extension of the array or to abort the search, as described for MMATCH hereinafter. It should also be noted, from the discussion for FIG. 16, that if a search is not aborted at stage 616 control will eventually return to SCREEN via stages 606, 608, and 612.

If, at stage 688 of FIG. 18, the answer was "yes", this means that the array can now be searched. At stage 696 all registers, R0 to R15, are reloaded from the array DUMX+8, wherein they were stored at stage 684. At the next stage, 700, a series of programmatic steps, in a conventional manner, will determine whether or not override codes are present in the JOBLIST item. If overrides are present, control would go to the MMATCH macro-instruction (stage 616, FIG. 16) for eventual processing by the macro instruction MOBILE CANONICALIZATION.

From stage 400 control goes to stage 698 wherein the question is asked "is the half-word DUN1 greater than zero?". At this point it should be noted that the half-word DUN1, which is set in the initialization step at stage 602 of FIG. 16, contains the length of the JOBLIST item screen. If, at stage 698, the answer is "not greater than zero" control goes to stage 702 and the register R1 is loaded with the address of &ADD1, which is the address of the EXECUTIVE POINTER in the array that is being searched. At this point it must be noted that there is only one EXECUTIVE POINTER in the array, because the screen length is zero. At the next stage, 704, the question is asked "is the EXECUTIVE POINTER zero?" If it is, the program goes to stage 710, wherein the registers IR2 and IR6 are both incremented with register BRY. At this point it should be noted that register BRY contains the length of the JOBLIST item screen, IR2 the location of the JOBLIST stem screen in JOBLIST, and IR6 the location of the EXECUTIVE POINTER in the array that was reached. Thus, the incremented registers IR2 and IR6 contain the location of the next screen (in JOBLIST) and the address part of the EXECUTIVE POINTER. The program next goes to stage 694 and then exits from SCREEN to the MMATCH macro-instruction at stage 616 in FIG. 16, as described above.

If, at stage 704, the answer was "it is zero" then control goes to stage 706 wherein the byte JI is set equal to one, which indicates that a vacant element has been found in the array. At the next stage, 708, the question is asked, "is MODE equal to one?". If the answer is "it is one" this signifies that a retrieval operation is being performed, and the program goes to stage 730. At stage 730 the question is "is the byte LEXICON zero?". If the answer is "it is zero", then at stage 732 the LMOVE macro instruction is used to move the JOBLIST item screen (specified in register IR2) to the EXECUTIVE

POINTER position in the array as specified in register IR6. It should be noted that stage 710 is executed only in the storage mode, and that the LMOVE operation, just described, actually constructs the first parts of a new EXECUTIVE POINTER in the correct position in the array. The address part of this partially constructed EXECUTIVE POINTER will be inserted at stage 616 of FIG. 16, wherein the MMATCH macro-instruction is executed. At that time the MSIGNAL 01 bit, which signifies whether or not the resident memory block has been altered, will be turned on, and subsequently, in stage 620 of FIG. 16, a new array will be created in the resident memory block.

Control goes from stage 732 to stage 710 and thence exits at stage 694, in the manner described previously.

Before continuing the discussion, an understanding of the roles played by SUPERSCH (stage 712), INSERT (stage 728), and the byte indicator LEXICON is needed. SUPERSCH actually performs the task of locating the position in the AUXILIARY FILE where an EXECUTIVE POINTER with the JOBLIST item screen should be. If the location is occupied by an EXECUTIVE POINTER with a different screen and the storage mode is indicated, then a vacancy must be created at the particular spot so that the new EXECUTIVE POINTER can be inserted in its correctly ordered position in the array and its continuances. The hole creating process can involve both the movement of EXECUTIVE POINTERS, within an array, and the transferral of EXECUTIVE POINTERS across an array. These two tasks are accomplished in a complex manner by the INSERT macro-instruction, which will be discussed hereinafter. INSERT uses the LEXICON byte as an indicator. At stage 730 of FIG. 18 the LEXICON byte indicates the status of the transferral of EXECUTIVE POINTERS between arrays.

Now at stage 730 of FIG. 18 the question asked was "is the byte LEXICON zero?" If the answer is "it is zero" this means that a vacancy exists and that there are no EXECUTIVE POINTERS being transferred across arrays. In this case control goes to stage 732 in the manner described earlier. If, at stage 730, the answer is "it is not zero", then a hole must be created in the array at the location specified in register IR6. This task is accomplished by the INSERT macro instruction (at stage 728), wherein the LEXICON byte is also altered and, if necessary, a transferred EXECUTIVE POINTER is inserted. Next control goes to stage 694 to begin the sequences of operations stipulated at stages 616, 620, etc. in FIG. 16. It should be noted that the LEXICON byte indicator is used after the TBADD macro-instruction (stage 620, FIG. 16) to return control, if necessary, to the SCREEN macro at stage 684. After the transferral of EXECUTIVE POINTERS has been completed the new EXECUTIVE POINTER is constructed in the hole that was created by the INSERT. This insertion occurs as the final step in the INSERT macro-instruction.

The remaining stages of FIG. 18, which begin at stage 712, are discussed next.

Stage 712 is executed when the screen length, which is recorded in the half-word DUN1, is greater than zero. In this case control passes from stage 698 to stage 712, wherein the SUPERSCH macro-instruction is executed. SUPERSCH finds the location in an array or its extensions (or continuances) where the EXECUTIVE POINTER with the JOBLIST item screen should be. In SUPERSCH, which is fully discussed hereinafter, the continuance 20 bit of MSIGNAL can be turned on, and new arrays are created or fetched by passing directly from SUPERSCH to the MMATCH (stage 616) and TBADD (stage 620) macro-instructions, as shown in FIG. 16. In both these cases, if the search operation is not aborted in the MMATCH macro-instructions, control returns to SUPERSCH via stages 684 through 698 in FIG. 18. Thus when control goes from SUPERSCH (stage 712) to stage 714; the register IR6 specifies exactly where the EXECUTIVE POINTER with the JOBLIST screen should be.

At stage 714 the CSCREEN macro-instruction is used to compare the screen in the EXECUTIVE POINTER to zero and to the screen in the JOBLIST item, and set the byte JI with one of the following codes:

JI	Meaning
00	The two screens are equal
01	The EXECUTIVE POINTER screen is zero
02	The EXECUTIVE POINTER screen has a lower numerical value than the JOBLIST item screen.
04	The EXECUTIVE POINTER screen has a higher numerical value than the JOBLIST item screen.

At stages 716, 718 and 724 of the SCREEN macro-instructions the specific number in byte JI is determined. If byte JI is zero, the program goes from stage 716 to stage 710, and the exit procedures of SCREEN are executed. If, at stage 718, JI is found to be 02 control goes to stage 720, and the register IR6 is incremented by register BRY, which contains the EXECUTIVE POINTER length. At the next stage, 722, the question is asked, "is register IR6 less than register IR3?" If the answer is "register IR6 is not less than register IR3" control goes to stage 692 wherein the MSIGNAL 20 bit is turned on and register IR6 is set equal to register IR3. At this point register IR6 contains the continuance address of an array that must be fetched or created, so control goes to stage 694 and then exits from SCREEN, as described above. If, at stage 722, it was found that register IR6 is less than register IR3, this would mean that there is at least one EXECUTIVE POINTER whose screen is greater than the screen in the JOBLIST item, and control goes to stage 714, which has been described previously. It should be recalled that all the equal length EXECUTIVE POINTERS are ordered within each array and its extensions (or continuances) with the lowest screen in the first position of the array. The operations in the branch, which begins at stage 720 and goes through stage 722 to stage 714, have been included to ensure that correct location is found by SUPERSCH (at stage 712).

If, at stage 724, it is determined that JI contains the number 04, then control goes to stage 726, wherein the question is asked "is MODE equal to zero?" If MODE is zero, the retrieval mode is indicated and the program exits from SCREEN directly to the MMATCH macro-instruction (at stage 616 in FIG. 16), wherein the unsuccessful search is aborted. If, at stage 726, it is found that MODE is not zero (i.e. the storage mode is indicated) then control goes to stage 728 and the INSERT macro is executed, in the manner described earlier. If, at stage 724, JI is found to contain zero, then the sequence of steps that begins at stage 730 is executed in the manner described earlier.

In FIG. 19, the SUPERSCH macro instruction is shown in flow diagram form. As was stated with respect to the description of FIG. 18, the SUPERSCH macro execution started after completion of stage 698, where it is determined:

a. the particular array that is to be searched has EXECUTIVE POINTERS whose length is a multiple of the contents of register R14, as it was determined that R1 was exactly divisible by R14; R9 by dividing R1 by R2. Next, the square root of the number of EXECUTIVE POINTERS in the array, which is in register R9, is computed and stored at the location MASK1. If the square root is not an exact number, it is truncated without rounding. Thus, an integer number is always stored in MASK1. Also, at stage 724, the absolute address of the last EXECUTIVE POINTER in the array is computed in register R1. This is done by subtracting the EXECUTIVE POINTER length, which is in register R2, from the absolute continuance address, which is stored in location C7.

After completion of stage 724, control goes to stage 726 wherein the macro-instruction LARGEXC is performed. The LARGEXC macro instruction is an extended form of the IBM basic assembler instruction XC. In this LARGEXC macro-instruction the number of bytes specified in the half-word

DUN1 of the array DUMX are set to zero. Because the half-word DUN1 contains the screen length, this means that a zero screen has been constructed in array DUMX. After completion of stage 726, control goes to stage 728, wherein the byte JI is set equal to the hexadecimal number 60.

At this stage it should be understood that the SUPERSCH macro-instruction operates by moving back and forth along the array in jumps equal to the square root of the number of EXECUTIVE POINTER positions in the array, as recorded in MASK1. Once a subblock, of the array of length MASK1, where the screen might be located, is determined it is searched, one EXECUTIVE POINTER at a time. The left-most four bits of byte JI, which are initialized at the start of the SCREEN macro-instruction (see stage 684 of FIG. 18), are used to control the directional movements of the register-pointer during execution of SUPERSCH. These four bits of the byte JI have been assigned the following meanings in SUPERSCH.

- i. The 80 bit indicates the direction in which the register pointer R1 must be changed. If the 80 bit is "on" (i.e., one) then R1 must be increased. If the 80 bit is "off" (i.e., zero) then R1 must be decreased.
- ii. The 40 bit indicates the last direction of change for the EXECUTIVE POINTER register R1. That is, if the 40 bit is "on", the last direction of change of R1 was positive, and if "off", the last direction of change was negative (i.e., R1 was decreased).
- b. further, it has been determined if the screen in the JOBLIST and the screen in the array are greater than zero. As the system enters SUPERSCH, the first stage of the operation is accomplished at stage 716. At stage 716, registers R0 to R15 are stored in the array C1. Then, in the next succeeding stage 718, certain registers are set.

First, both registers R0 and R6 are set equal to the screen length recorded in the half-word DUN1.

Register R2 is set equal to the EXECUTIVE POINTER length by adding the screen length, set in R0, and the composite address length &ADDL, which is a System Parameter.

Register R4 is loaded with the absolute machine address of the head of the array. This is accomplished by taking the address in location C4, which is the address of the first available EXECUTIVE POINTER in the array, and subtracting therefrom four bytes. These four bytes at the head of the array contain the relative continuance address. Thus the value in register R4 will be the absolute address of the start of the array.

Register R1 is then set to the length of the EXECUTIVE POINTERS in the array. This is accomplished by subtracting four bytes from the relative continuance address, found at the address recorded in register R4. The four bytes are subtracted from the relative continuance address because that address is relative to the head of the array.

After completion of stage 718, the control is transferred to stage 720 wherein the question is asked "is R1 evenly divisible by R2?". Actually, this question should always be yes as the same question has been asked at stage 688 in the screen macro. However, a check has made at stage 720 as to whether the length of the EXECUTIVE POINTERS in the array, as recorded in register R1 are divisible by the EXECUTIVE POINTER length recorded in register R2. If the answer is "no", then the program would continue at stage 722 to print out an error message for the SUPERSCH macro. If the answer at stage 720 is "yes", the program continues to stage 724.

At stage 724, certain parameters are computed. First, the number of EXECUTIVE POINTERS in the array is computed in register

- iii. The 20 bit of the byte JI is used to indicate whether or not the first direction of change of register R1 has occurred. If it is "on", this means that the register R1 is to be changed for the first time. Of course, the 20 bit "off" means that register R1 has already been changed at least once.
- iv. The 10 bit is used to record the last status of the 40 bit of the JI byte.

Now, at stage 728, the 40 and 20 bits of byte JI were turned on. This means, with respect to the 40 bit, that the EXECUTIVE POINTER register was last increased. This occurred when the position of the last EXECUTIVE POINTER in the array was computed in Register R1 at stage 724. With respect to the 20 bit, this means that the register R1 has not been changed from its initial setting.

At stage 730 the 80 and 10 bits are turned off, and control goes to stage 732, wherein the question is asked "What is the value of the 40 bit in byte JI?" If the 40 bit is one, then the 10 bit is turned on at stage 736 and the program goes to stage 734. If, at stage 732, the 40 bit is zero, control goes directly to stage 734. Thus at stage 734 the status of the 40 bit has been preserved in the 10 bit. At stage 734, the macro-instruction CSSCRN is executed. The macro-instruction CSSCRN is an extended form of the IBM basic assembly language instruction CLC. At stage 734, the question is asked "is the screen at the location indicated in register R1 (the last EXECUTIVE POINTER in the array at this stage time) equal to zero or a number other than zero?". If the screen is equal to zero, then control goes to stage 738. If the screen is a value other than zero, control goes to stage 740. At stage 740, another CSSCRN macro instruction determines if the screen, whose address is recorded in register R1, is equal to, less than, or greater than the screen whose address is recorded in register IR2. It would be remembered that the screen whose address is recorded in register IR2 is the screen in the JOBLIST item, and that the screen whose address is recorded in register R1 is the screen which we are looking at in the array. If the screen in the array is greater than the screen in the JOBLIST item, then we must go backward in the array to find the correct location and, accordingly, control goes directly to stage 738. If the screen in the array is less than the screen in the JOBLIST item, then control goes to stage 742, wherein the 80 bit in the JI byte is "turned on." This indicates that the register R1 should be increased to find the location in the array where the EXECUTIVE POINTER should be. From stage 742 the program goes to stage 738.

At stage 738, the 20 bit in the JI byte is checked. If the 20 bit is one, then, at stage 744, the 20 bit of the JI byte is turned off, and the register R9 is decremented by one. This enables us to return to the first EXECUTIVE POINTER in the array when the new value of R1 is computed from R9. Control goes from stage 744 directly to stage 756.

Before discussing what happens at stage 738, when the 20 bit is zero, a discussion must be made of what happens at stage 740 when the screen in the array is equal to the screen in the JOBLIST item. When this occurs, we have found an EXECUTIVE POINTER which has the JOBLIST screen, and control goes to stage 746 where termination of the SUPERSCH operation will be effected. At stage 746, the location C4 has recorder therein the address in register R1. C15 has stored therein the screen length recorded in register R6 and, C16 has stored therein the EXECUTIVE POINTER length recorded in register R2. Then registers R0 to R15 are loaded from the array C1. Next control goes to stage 748 wherein the MSIGNAL 20 bit is checked. If the MSIGNAL 20 bit is on, the address specified by register R6 is a continuance address. If the MSIGNAL 20 bit is off, then the SUPERSCH procedure has been completed.

If, at stage 748, the MSIGNAL 20 bit was "on", control goes to stage 750 wherein the question is asked "is the continuance address whose location is recorded in register IR6 equal to zero?" If the continuance address is equal to zero, then control goes directly to the MMATCH macro in SSEARCH (see stage 616 in FIG. 16).

If, at stage 750, the continuance address specified by register IR6 is not zero, then it is stored in the location ADDRESS, and control goes directly to the TBADD macro-instruction in SSEARCH (see stage 620 in FIG. 16).

TBADD macro will fetch a new array, and control will eventually be returned to the SUPERSCH macro instruction for a search of the new continuance array.

At stage 738, if it was determined that 20 bit of JI was zero, meaning that this is at least the second time that control has come through stage 738, then the register R9 is reset to the value R9 minus the contents of MASK1 the square root of the total number of EXECUTIVE POINTERS. After completion of stages 754, or stage 744, control would pass directly to stage 756 wherein a check is made as to whether R9 is equal to or less than zero, this means that the subblock search has been completed and control passes directly to stage 758. However, it is in the best interest of this discussion to discuss the result when R9 is greater than zero and the subblock search must be completed and then return to stage 758 at a later time in this discussion.

When R9 is greater than zero, the program passes control to stage 760 wherein a register R15 is set equal to the contents of register R2 times the contents of register R9 or the exact number of bits which must be moved to look at a new EXECUTIVE POINTER in the array. After completion of stage 760, control passes to stage 762 wherein the JI 80 bit is checked. If the 80 bit is zero, this means we must move backwards and if the 80 bit is one this means that we must move forward.

If the 80 bit is zero, control passes to stage 764 wherein the 40 bit of the JI byte is turned off and R15 is reset to the address in R1 minus the value set in R15. This, in effect, sets the new EXECUTIVE POINTER address in the array which is to be checked. Then, at stage 766, a check is made as to whether this new address is greater than or equal to the address in register IR6 which is the first address in the array. If, the address in R15 is greater than or equal to the address in register IR6, the first address in the array, control would pass to stage 768 wherein register R1 would now be set at the address in register R15. If the answer at stage 766 had been less than zero, then control would have passed back to stage 700 for purposes of accomplishing a step search of the EXECUTIVE POINTERS in the sub-block. This type of search will be discussed later.

If the 80 bit of the JI byte at stage 762 is "one," this means that the EXECUTIVE POINTER index, R1, must be increased. Thus register R15 is set equal to register R1 plus the increment (which is in register R15) and the 40 bit of byte JI is turned on to indicate that the movement is forward. At stage 774 the question is asked "is this new value of register R15 greater than the continuance address which is recorded in storage C7?" If the address in register R15 is equal to or greater than the address in location C7, control goes to stage 766 to execute a series of instructions that will be discussed later. However, if the address in register R15 is less than the continuance address (recorded in location C7), control goes to stage 768 wherein the address recorded in register R15 is then transferred to location R1.

When the operation at stage 768 is completed the program goes to stage 778 wherein the question is asked "Are the 40 and 10 bits of the JI byte both on, both off, or mixed?" If they are mixed, then the subblock search continues by returning control to stage 730. If they are both on, this means that there has been two successive forward going steps and, accordingly, control goes directly to stage 776. If they were both off, this means there have been two successive backward going steps and control goes to stage 758 wherein the register R1 is decreased by the subblock length, which is in MASK1.

Control then goes to stage 780 wherein the question is again asked "is the new value in register R1 equal to or greater than the address of the first EXECUTIVE POINTER that is recorded in location IR6?" If that is the case, control goes directly to stage 776. If the new address in register R1 is less than the address in register IR6, control goes to stage 770, wherein the address in register R1 would be set equal to the value of IR6. This assures that the address in register R1 will not be less than the address of the first EXECUTIVE POINTER in the array. From stage 770 control goes to stage 776 wherein the macro-instruction CSSCRN is executed. There, the question is asked whether the screen of the present EXECUTIVE POINTER whose address is recorded in register

R1 is equal to zero?" If it equals zero, the SUPERSCH termination procedure that start at stage 746 is executed. If at stage 776, the answer is "not zero" control goes to stage 782. (see above). At stage 782 the CSSCRN macro-instruction is used to ask the question: "is the screen whose address is recorded in register R1 less than the screen whose address is recorded in register IR2 (the JOBLIST item)?" If the answer is "not less than", then control again goes to stage 746 to begin the SUPERSCH termination procedure. It should be noted here that if the two screens are equal, the exact location in the array has been located, and no new EXECUTIVE POINTER will be inserted. However, if, during a storage mode, the screen whose address is in register R1 is greater than the JOBLIST item screen, whose address is in register IR2, then the INSERT macro-instruction of FIG. 20 will eventually be used to create a hole and insert a new EXECUTIVE POINTER.

If the answer at stage 782 is "less than" control goes to stage 784 wherein register R1 is incremented by the EXECUTIVE POINTER length, which is in register R2.

Then, control moves to stage 786 wherein the question is asked "is register R1 greater than, less than, or equal to the continuance address that is recorded in location C7?" If register R1 is greater than the continuance address, control goes to stage 722, and therein indicates that there is an error in SUPERSCH. If register R1 is equal to the continuance address, then control goes to stage 788 wherein the MSIGNAL 20 bit is turned "on", thus indicating that a continuance of the array must be fetched. Control goes from stage 788 to stage 746 wherein the SUPERSCH termination procedure begins (see above).

If, at stage 786, register R1 is less than the continuance address, control returns to stage 776. This cycle (through stages 776, 782, 784, and 786) is repeated until control goes to stage 746 (from stages 776 or 782) or either of stages 722 or 788 (from stage 786). Thus, in the SUPERSCH macro, we have either found the exact location of an EXECUTIVE POINTER or found the exact location where the new EXECUTIVE POINTER is to be inserted, or we have determined that an extension (or continuance) of the array must be fetched or created and searched.

FIG. 20 is the flow diagram for the macro-instruction INSERT. By the time the program reaches the macro-instruction INSERT, at stage 428 of FIG. 18, certain things have occurred. First, the particular EXECUTIVE POINTER recorded in register IR6 has been defined, and, additionally, it is known whether, at that address, the space is empty or filled. Further, if the space is filled, we know that the screen in the array is greater than the screen in JOBLIST. Additionally, we know whether we are in the retrieval or storage mode. We know that it is in the storage mode. Further, we have the JI value set in CSCREEN; that is, the value of the JI byte is either 01 or 04 meaning that the screen in the array is zero or that the screen in the EXECUTIVE POINTER in the array is higher than the screen in JOBLIST.

The first stage of the macro INSERT is stage 800 wherein the question is asked "is JI equal to 01?". If the byte JI is 01, then this means that the screen in the array is equal to zero.

If the value of the screen is zero, the program continues to stage 802. If JI is not 01, then the program continues to stage 804. For purposes of discussion we will assume that the 04 bit of JI is "on" and that we are proceeding to program stage 804. At stage 804 two registers are set. The first register R0 is set at the length of the EXECUTIVE POINTER which is stored in the register BRY. Register R1 is set to the address of the EXECUTIVE POINTER in the array, which is in register IR6. After these two registers are set, the program continues to stage 806 wherein the question is asked "is the EXECUTIVE POINTER in the array whose address is in register R1 vacant or zero?" If it is vacant or zero then the program would continue to stage 808.

If, at stage 806, the EXECUTIVE POINTER whose address is in register R1 is not zero, then the program would continue

through a loop defined by stages 810 and 812 until a vacant or zero below the starting address is found in the array. First, if the EXECUTIVE POINTER is not zero, at stage 810, register R1 is reset to one EXECUTIVE POINTER length beyond the address set in IR6, by adding register R0 to register R1. Then, at stage 812, the determination is made as to whether the continuance address has been reached. If it has not been reached, the program returns to stage 806 and a determination is made as to whether at that new R1 the EXECUTIVE POINTER is zero. If the EXECUTIVE POINTER was zero, then control goes to stage 808. If the EXECUTIVE POINTER at the new address in R1 is filled, the program would continue through stages 810 and 812 until, either, (a) an empty EXECUTIVE POINTER location would have been found, or, (b) the absolute continuance address in register IR3 would be found. If the absolute continuance address in register IR3 is found, the program will continue to stage 814. At stage 814, the following registers would be set:

- a. R1 is reset to the address of the last EXECUTIVE POINTER in the array by subtracting the value of R0 from R1. Then,
- b. Registers R0 to R6 are stored in array C1; and
- c. The value of register IR3 is decreased by R0.

After completing these steps, the program continues to stage 816 where a determination is made as to whether the 01 bit in LEXICON is on.

At this time, perhaps a discussion of the LEXICON array and its use in INSERT is desirable. The 16 bytes of the LEXICON array are used by the INSERT macro. In the first byte, which is set in SSEARCH, the last four bits are significant. The 01 bit is used to signify whether an address has been determined for LEXICON in the YY array storage. Further, when the 01 bit is on, there is an indication that in addition to the address in the YY array having been determined, the screen in JOBLIST has been stored in the high end of the YY array and the address of the screen is stored beginning at LEXICON plus 4 bytes. At stage 816, if the 01 bit is zero, then the program would continue to stage 818. If the 01 bit has been on, the program would have continued to stage 820. At stage 818 certain steps are taken:

- a. the register IR1 is set at a value equal to the beginning address of the main storage array, which is stored in AYY during the initialization by SSTATECL;
- b. the System Parameter <HAYY, which is the length of the main storage array YY, is added to IR1. This would bring us to the end of the YY array.
- c. From this value is subtracted the half word DUN1, which is the length of the screen in JOBLIST.
- c. Then, this address IR4 is stored in LEXICON plus 4 bytes.
- e. In LEXICON plus 8 bytes is stored the address IR1 minus R0. R0 contains the length of the EXECUTIVE POINTER in the array.
- f. In LEXICON plus 12 bytes is stored IR1 minus 2 times R0.
- (g) In LEXICON plus 16 is stored ADDRESS, which is the composite address of the next available array which will be requested in CREATE.

Then the program continues to stage 822 wherein the LMOVE macro is used to move the screen from JOBLIST and store it at the location described by register IR1 at the end of the array YY. This saves the JOBLIST screen in storage for its eventual insertion in the correct position in the array.

Then, as with the case of the LEXICON bit being turned "on", the program continues to stage 820.

At stage 820 a check is made to determine whether the 02 bit of LEXICON is turned on. If the 02 bit is zero, then, at stage 824, the 02 bit in LEXICON is turned on and register IR1 is loaded with the address in LEXICON plus 8. If the 02 bit were on a stage 820, the LEXICON 04 bit would have been turned on at stage 826 and register IR1 would have been loaded with the address at location LEXICON plus 12. In this case, it means that there is already an EXECUTIVE

POINTER defined by the address in LEXICON plus 8 and, therefore, it is necessary to load the new EXECUTIVE POINTER in the address defined at location LEXICON plus 12.

5 After completion of the steps at either 824 or 826, control goes to stage 828 wherein the half word MASK1 is set equal to the length of the EXECUTIVE POINTER, which is in register R0. Then control is transferred to stage 830, wherein the left move macro instruction LMOVE is executed and the EXECUTIVE POINTER in the array, defined by register IR3, is stored at location in register IR1 in the YY array. Then registers R0 to R6 are loaded from array C1 at stage 832. These registers were saved during the transfer process at stage 814.

15 At stage 808, the right move macro RMVC is executed. For this macro, certain registers are set. Register BRYY, the number of bytes which have to be moved within the array, is set equal to register R1 minus register R6. Register BRY set equal to register R1 minus 1. Register R1 is set equal to register BRY plus register R0. BRY now contains the end location of the old array.

It should be understood that R1 contains either the address of the last byte in the last EXECUTIVE pointer in the array, or alternatively, if this stage 808 had been reached directly from stage 806, the address of the last byte in the first vacant EXECUTIVE POINTER in the array.

25 The right move macro RMVC moves the EXECUTIVE POINTER starting at the address IR6 to the end of the array one EXECUTIVE POINTER length, leaving a hole in the sub array for the new EXECUTIVE POINTER whose screen is in JOBLIST and whose address will be inserted in MMATCH. The last EXECUTIVE POINTER in the array must be stored and, in fact, was stored previously, in the save area of the YY array and the address where said EXECUTIVE POINTER was stored is recorded at either LEXICON plus 8 or LEXICON plus 12.

30 At the completion of the right move macro RMVC at stage 808, the program continues to stage 834 wherein location C15 and register BRYY are both set equal to the value of the half-word DUN1, which is the length of the screen in JOBLIST. Location C16 and register BRY are set equal to the value of the half word DUN1 plus &ADDL. Thus BRY and C16 now contain the length of the EXECUTIVE POINTER which will be inserted in the array. Next registers R0 to R6 are stored in array C1.

40 Then, the program continues to stage 836 wherein the question is asked "is the LEXICON byte zero?" This would have occurred only if the program stage 808 had been entered directly from stage 806. If this was so, the LEXICON array would not be used because no continuance is required; and, accordingly, the program then continues directly to stage 838 wherein certain operations would be accomplished before exiting from the program.

45 At stage 838, R0 is set to the value of the half-word DUN1, IR6 is changed to the IR6 value plus R0. IR6 now contains the location of the place in the array where an address must be added to the screen from JOBLIST to form a new EXECUTIVE POINTER. The address at IR6 is, of course, set to zero. Then IR6 is reset to the screen address in the array by subtracting therefrom the value R0.

60 After completing these steps, the control goes to stage 840 wherein another LMOVE macro is executed. The screen in JOBLIST is moved to the address designated by IR6 in the array.

65 After completing stage 840, IR2 is reset to equal IR2 plus BRYY or the address of the next screen in JOBLIST. IR6 is reset to IR6 BRYY or the position in the array where the address must be placed adjacent to the screen just taken from JOBLIST. This is accomplished at stage 842. After completion of stage 842, the program is completed.

75 Returning to stage 836, we will consider the possibility when LEXICON is not equal to zero. Then, the program continues at stage 844 wherein a determination is made whether the 04 bit in LEXICON is zero or one. If the 04 bit is zero, that

means that the location defined by the address in LEXICON plus 12 byte is empty. If the 04 bit is one, then the location defined by the address in LEXICON plus 12 is filled. It should be understood that when the location in the YY array is filled whose address is in LEXICON plus 12 bytes then, necessarily, the location determined by the address in LEXICON plus 8 bytes is also filled and two EXECUTIVE POINTERS are in place. If the 04 bit is on, the program continues to stage 846. If the LEXICON 04 bit is zero, the program continues to stage 848.

At stage 846, the half word MASK1 has stored therein the contents of register R0. R0 contains the EXECUTIVE POINTER length. Then register IR1 is loaded with the address in LEXICON plus 8 bytes.

If the 04 bit of LEXICON were zero, as was stated previously, the program would have continued at stage 848 wherein the 01 bit of LEXICON would be checked. If the 01 bit in LEXICON were one, this would have meant that there were no more continuances and control would have gone to stage 852 and thence in the same manner that will be discussed with respect to stage 802. However, for our purposes, we will discuss the operation when stage 848 indicates that the 01 bit of LEXICON is zero. At stage 854 the half-word MASK1 is loaded with the EXECUTIVE POINTER length, which is in Register R0. Then, control goes to stage 856 wherein the macro LARGEXC is executed. This instruction merely zeros or erases the EXECUTIVE POINTER which was in the hole identified by the address recorded in register IR6 IR6 has recorded therein the address of the hole where the screen of JOBLIST is to be inserted. Thus, at stage 856, this hole has been erased and cleared for a later insertion

After completing the step at stage 856, control is transferred to stage 858 wherein the LEXICON 01 bit is turned on. Then, at stage 860 the address in LEXICON + 8 is recorded in register IR1. Then, the program continues to stage 862. At stage 862, another LMOVE macro instruction is executed and the screen of the EXECUTIVE POINTER located by the address in register IR1 is moved into the JOBLIST in place of the screen originally therein. The screen originally therein, of course, has been saved in the YY array at the address recorded in LEXICON + 4.

After completion of the operation at stage 862, control goes to stage 864 wherein the registers R0 to R6 are loaded from the array C1 and the MSIGNAL 20 bit is turned on. After completion of stage 864, it will be obvious that the insertion operation has not been truly completed by the operation above described, but the program will continue, as shown in FIG. 16 through MMATCH stage 616, TBADD macro 620, and stage 626 back to the start of SCREEN at stage 614. This will recycle and, when it comes through for the second time, the LEXICON 04 bit will be on. When the LEXICON 04 bit is on, control would have come through stage 846 to stage 850. At stage 850 an LMOVE macro-instruction would have been executed to insert the EXECUTIVE POINTER recorded in the YY array at the address in LEXICON plus 8 bytes into the opening whose address is recorded in register IR6

After completing stage 850, control is transferred to stage 866 wherein register IR3 is loaded with the address in LEXICON + 12. Then, at stage 868, another LMOVE macro is executed to transfer the EXECUTIVE POINTER in the YY array determined by the address in LEXICON + 12 to the place in the YY array determined by the address in LEXICON + 8 bytes. After completing stage 868, control goes to stage 870 wherein the 08 and 04 bits in the LEXICON byte are turned off. Then, control goes to stage 862 wherein the screen stored in the YY array whose address is at LEXICON + 8 bytes is transferred to JOBLIST. Then control passes through stage 864 as discussed previously.

If the LEXICON 04 bit was zero and the LEXICON 01 bit was one, the control goes to stage 852, as discussed earlier. Similarly, when the 01 bit of the JI byte was on, control also goes from stage 800 to stage 852. In stage 852, the register IR1 has recorded therein the address in LEXICON + 8 bytes. The

half word MASK1 has recorded therein the value of R0, which is equal to the value in the half word DUN1 plus &ADDL, the EXECUTIVE POINTER length. From stage 852 control is transferred to stage 872 wherein another LMOVE macro instruction is executed and the EXECUTIVE POINTER stored in the YY array at the address defined in LEXICON + 8 is inserted in the array at the address recorded in IR6. Then, at stage 874, IR1 register is reset to LEXICON + 4 bytes. At stage 876, the screen whose location was determined by the address in LEXICON + 4 bytes is transferred back to the JOBLIST.

After completing stage 876, control goes to stage 878 and certain parameters are set. First, the byte LEXICON is set to zero; the registers R0 to R6 are loaded from array C1, and the MSIGNAL 20 bit is turned on. After completing stage 878, control is transferred to stage 620 in FIG. 16.

It must be understood that the hole created in the array will be filled when control returns to SCREEN, after cycling through SSEARCH, at stage 730 and goes to stage 732. The LMOVE macro in stage 732 will insert the correct screen in the hole left in the array.

In FIG. 21, there is shown the flow diagram for the MMATCH macro. In the first stage in MMATCH a determination is made as to whether the address associated with the EXECUTIVE POINTER in the array, as determined by the address recorded in register IR6, is zero or a value other than zero. IR6 has been set by any one of the three stages in INDEX search 610, screen search 614 or AUXFILE search 630 to point at the address of the EXECUTIVE POINTER in the array or to the continuance address. If the address specified by register IR6 is not zero then there is no mismatch; and, therefore, control should go to TBADD. Thus, MMATCH will be bypassed and control will go directly to TBADD. The output of stage 880 is connected to stage 882 wherein the address specified in register IR6 is recorded in the location ADDRESS.

Then the program continues to stage 884 wherein the question is asked "is the MSIGNAL 20 bit zero or one?". If the MSIGNAL 20 bit is "one", indicating that one must access or create a continuance, the program goes directly to the TBADD macro. If the MSIGNAL 20 bit is off or zero, the program continues to stage 886. There the half word MASK2 value is stored in the half word DUN1. As we noted previously, the half word DUN1 contains the screen length and it now contains the screen length of the next screen in JOBLIST.

Further, the third byte of MSIGNAL (or MSIGNAL + 2) bytes is incremented by to indicate that this is the first screen in JOBLIST. Each time MMATCH is executed at stage 884 and the MSIGNAL 20 bit is 0 the MSIGNAL + 2 byte is incremented by one. After completing stage 886 the control goes to the TBADD macro at stage 620 in FIG. 16. The programmatic stages 880, 882, 884 and 886 are not, physically within the MMATCH program listing, but they have been show in the flow diagram for purposes of clarity.

After a determining at stage 880 that the address specified by register IR6 is zero, control is transferred to stage 888 where SRGATE is set equal to 02. SRGATE is a special gate which has three different settings. If SRGATE is 02, this means that the search was successful and the search may be continued. If SRGATE is 01, this means that the search must be reexecuted and that the JOBLIST will have been rearranged. This occurs in the macro-instruction STRATEGY. If the SRGATE is 00, this means that the search has failed and it must be terminated This is accomplished by exiting to the instruction FINISHED in SSEARCH. The next stage of MMATCH is at stage 890 where the question is asked "is MODE equal to one?" If MODE is equal to one, then the search is in the storage mode. If MODE is not equal to one, then a retrieval or one of the three update modes operations is being executed. Thus, if the answer at stage 890 is that MODE is one, control goes to stage 892. If the answer at stage 890 is that MODE is not equal to one, the control is transferred to stage 894. At stage 892 the MSIGNAL 01 bit is turned on, in-

dicating that the resident memory block has been changed. This information will be used by the TBADD macro and the Global Memory component (SMEMORY) if a new memory block is required.

After completing step 892, control goes to stage 895 and SRGATE is checked. If the SRGATE is zero, then this means that the search is finished and it is terminated through stage 896 at FINISHED. If SRGATE is 01, this means that the JOBLIST was rearranged in the STRATEGY macro and a new search must be started by going through stage 898 to the location NEWPLAY in the SSEARCH component. If, the SRGATE is in fact 02 at stage 895 then control goes to stage 900 wherein the composite address EMPTY is stored in the array at the address defined by register IR6. This completes the constructions of the new EXECUTIVE POINTER in the array. Additionally, the MSIGNAL 04 bit is turned on and the location at the head of the M array where address EMPTY is normally stored is zeroed. It should be remembered that the M and J arrays are permanently resident in core. Also, at storage 900, the address in register IR6 is recorded in the storage location BWX plus 76. This information may be required by the CREATE macro and the SMEMORY component.

At stage 894 the register IR6 contains the address of a vacant or zero location in the array. This means that the retrieval and update has been unsuccessful and the override code information must be used to determine whether or not the search must be aborted. It should be noted that the three remaining options for MODE (namely 2, 3, and 4), which were discussed earlier, can be carried out by appropriate program steps between stages 890 and 894.

If, at stage 890, it was determined that the search was not in the storage mode, the program would have continued to stage 894 wherein the counter JII is set to zero. SRGATE is also set to zero. This information indicates that the search has been unsuccessful. This information can, of course, be overridden if in the next succeeding states 904, 906 and 908 control goes to STRATEGY and STRATEGY determines that, in fact, the search can be successful.

After completing the operation at stage 894, control goes to stage 902 wherein the question is asked "is the MSIGNAL 20 bit zero or one?". The MSIGNAL 20 bit indicates whether or not a link (or continuance) address is to be inserted at the foot of the array. If a link address is being considered the answer is one, and control goes directly to stage 895. Because the SRGATE has been set at zero the unsuccessful search will be aborted through stage 896. If, at stage 902, the answer was "zero," then the MSIGNAL 20 bit was not turned on and control goes to stage 904 wherein the question is asked "are there any Type 1 over rides?" If the answer is "no", then control goes to stage 906 wherein the question is asked "are there any Type 2 over rides?". If the answer at this stage is "no", the program would have continued to stage 908 where again the question would be asked "are there any Type 3 over rides?" If the answer at stage 908 is "no", control goes to stage 895 and, since the SRGATE is zero, the unsuccessful search would be aborted through stage 896 at location FINISHED in the SSEARCH component.

If at any one of the stages 904, 906 or 908, Type 1, Type 2, or Type 3 overrides were found, control is transferred to stage 910 and the macro-instruction STRATEGY is executed. Type 1, Type 2 and Type 3 overrides are introduced in the assigned descriptor-sets and they are counted and their locations are noted in the translators when these descriptor-sets are rearranged to their JOBLIST item forms. This occurs before the search procedure begins. Thus, if there is Type 1, Type 2, or Type 3 overrides are present in the assigned descriptor-sets the macro-instruction strategy will be used.

The flow diagram for the STRATEGY macro-instruction is shown in FIG. 22. In the first stage, 916, a special TRANSFER macro-instruction transfers control to the special component SMATCH. In SMATCH there will be accomplished, automatically, the inverted or intersecting file type search.

At stage 918, the question is asked, "what is the value of of SRGATE?". If SRGATE is two, then control goes directly to stage 922 wherein the question is again asked "is SRGATE equal to, less than, or greater than one?". If SRGATE is zero, at stage 918, the control passes to FINISHED in the SSEARCH macro. This would terminate the search.

If SRGATE is one, at stage 918, the following conditions have probably occurred. The screen in JOBLIST might have been A1BD. If, in searching the screen array, SMATCH at stage 916 determines that there is a ABCD screen in the array, the SMATCH would have set the SRGATE so that the MOBILE CANONICALIZATION routine would be executed at stage 920. SMATCH would not have set the SRGATE at one if, during the translation stage, the JOBLIST had not been normalized.

At stage 920, control is transferred to the macro-instruction MOBILE wherein the MOBILE CANONICALIZATION package is executed on the JOBLIST item. At the end of the SMOBILE program, control is transferred back to the stage 922 in STRATEGY. MOBILE CANONICALIZATION can be defined as a strategic rearrangement of the JOBLIST item which might effect matching. If the JOBLIST item can be rearranged so as to achieve matching, then, SMOBILE, will do it.

After completing stage 920, control goes to stage 922 wherein again the question is asked "is SRGATE equal to, less than or greater than one?". If less than one, the unsuccessful search is terminated at the location FINISHED in SSEARCH. If SRGATE is equal to one control goes to the location NEWPLAY in SSEARCH. This occurs when new permitted arrangements of the JOBLIST item were effected in SMOBILE. If SRGATE is greater than one, the matching procedure executed in SMATCH and or SMOBILE) has disclosed the existence of an acceptable information subpath. In this case, a stage 912, the composite address in location 0(IR6) is loaded into ADDRESS and control goes to stage 924 in the TBADD macro.

The flow diagram for TBADD is shown in FIG. 23. In TBADD, the first stage (924) contains a COMPARE macro-instruction which compares the slow address parts of the composite addresses CURRENT and ADDRESS. If these two slow memory addresses are equal, this means that the requested memory-block is already resident in core-memory. It should be remembered that CURRENT contains the virtual memory address of the resident memory block, and ADDRESS contains the virtual memory address of the requested memory-block. The second part of ADDRESS, which is called the fast-memory address, specifies the beginning address of the requested array when the requested memory-block is core-resident.

Now to return to FIG. 23, if at stage 924 the slow address parts of CURRENT and ADDRESS are equal then the requested memory-block is already resident in core and control goes to stage 926. There, the APART macro-instruction extracts the fast-memory part of ADDRESS and stores it in register IR6 is checked. Next control is transferred to stage 928 and the 04 bit of the MSIGNAL byte is checked. If the 04 bit of MSIGNAL is off, control goes to stage 930. The 04 bit of MSIGNAL indicates whether or not the requested sub array exists in core-memory. If the sub array already exists the MSIGNAL 04 bit is off. If the subarray does not exist, then the MSIGNAL 04 bit is one and, a new subarray must be created by the CREATE macro at the address specified in register IR6.

If the MSIGNAL 04 bit is on, control goes to stage 932 wherein the macro instruction CREATE is executed. The operation of the macro-instruction CREATE will be more fully discussed hereinafter (see FIG. 24). However, for purposes of a simplified description, CREATE first checks to see whether it is possible to fit a new subarray in the resident memory block. If there is not enough space in the resident block CREATE calls the Global Memory Component (SMEMORY), which writes the resident memory-block in virtual memory and then creates a new resident memory-block.

All the composite addresses are updated. Control then goes to state 930 wherein the MSIGNAL 04 bit is turned off. After completing stage 930, control is transferred to stage 626, as shown in FIG. 16.

If there is room in the memory block for the subarray, CREATE simply creates the subarray, updates the composite addresses in the memory block, then control goes to stage 930.

If, at stage 924, it was determined that the slow memory portions of CURRENT and ADDRESS were not equal, then control passes to stage 934 wherein the slow portion of ADDRESS would be compared with zero. If ADDRESS was equal to zero, this would mean that a memory block is not required because the fast memory address in ADDRESS specifies a location in the permanently core-reident part of the AUXILIARY FILE. Accordingly, control passes immediately to stage 926. If the slow-memory part of ADDRESS is not equal to zero at stage 934, then control passes to the macro GLOBAL at stage 936. GLOBAL calls the Global Memory component (SMEMORY), which supervises the memory-blocks of the AUXILIARY FILE. GLOBAL insures that the resident memory-block will be updated in virtual memory and it fetches (or creates) the new memory-block whose address in in ADDRESS. After GLOBAL has completed its operations, the composite address CURRENT is set equal to ADDRESS. This is accomplished at stage 938. This means that request address (ADDRESS) is now also stored in CURRENT.

After completing stage 938, control goes to stage 926 in the manner discussed previously.

In FIG. 24, there is shown the flow diagram for the CREATE macro. In the first state, 940, initializing information is computed. First, register LR5 is set equal to the sum of the values in register IR6 and IR1. This sum in register IR5 is the absolute machine address of the head of the array that is to be created. Register IR6 contains the relative address of the head of the array that is to be created within the resident memory block. Register IR1 contains the base address of the memory block.

Register BRY is set equal to $\&TRKL$ times $\&NTRKS$ or the memory block size. $\&TRKL$ and $\&NTRKS$ are system parameters with $\&TRKL$ being the number of bytes per track and $\&NTRKS$ being the number of tracks per memory block. Register R0 has recorded therein the half word DUN1, which is the screen length. Register R1 has recorded therein $\&ADDL$, which is the address length for the array. It is now necessary to determine, first, how much storage is required for the array so that it will then be possible to determine whether there is sufficient room in the memory block for the new array to be created. Thus, the first step is to proceed to stage 942 where the question is asked "is R0, which contains the screen length, greater than zero, zero, or less than zero?" At stage 944 register R1 will be set at $\&LSLOW$, the slow memory address length which is a system parameter used in AUXFILE, and Register R1 is set to zero.

If register R0 is equal to or greater than zero, control goes directly to stage 946. Control also goes from stage 944 to 946. At stage 946, register DUN7 has recorded therein the sum of registers R1 and R0. In the case of control coming from stage 944, DUN7 will contain $\&LSLOW$. If register R0 was equal to or greater than zero, then DUN7 will contain the sum of $\&ADDL$ and the screen length. This is the EXECUTIVE POINTER length for the array. Additionally, at stage 946, register BRY is loaded with $\&MATRIXS$, which is a system parameter indicating the number of EXECUTIVE POINTERS in a secondary array. Secondary arrays are those associated with the screens BD_1 , LD_1 , BD_2 , . . . , and the bulk storage addresses.

The value of the byte (MSIGNAL+2) determines which kind of array is to be created. At stage 948, the question is asked "does the MSIGNAL plus 2 byte have two, more than two, or less than two therein?" If there is less than two in the MSIGNAL +2 byte, then, at stage 950, BRY is set to 20. If there is less than 2 in the MSIGNAL plus 2 byte this means an

array associated with index M or screen J is being created. In this case the length of the array is arbitrarily set at twenty EXECUTIVE POINTERS. If the MSIGNAL + 2 byte is greater than two, then the present value of $\&MATRIXS$ is correct and control goes directly to stage 952. If the MSIGNAL + 2 byte contains two, then control goes to stage 954 wherein BRY would be set to equal to $\&MATRIXL$. $\&MATRIXL$ is a system parameter indicating the number of EXECUTIVE POINTERS which would fill the primary array, which is associated with the screen LD_0 .

At stage 952, the register BRY (the number of EXECUTIVE POINTERS in the array is multiplied times DUN7 (the EXECUTIVE POINTER length) and four bytes are added thereto then the total is stored in register R0. At this point the register R0 contains the relative continuance address in the array that is to be created. The extra four bytes are added to account for the location at the head of the array where the relative address of the continuance address is stored. Then, register R1 is set equal to register R0 plus $\&ADDL$, which is the total length of the array that is to be created. $\&ADDL$ is added in order to provide space for the continuance address, which will be added at the end of the array. Control then goes to stage 956 where the question is asked "is register R1 (the length of the array), greater than BRY (the size of the memory block?)". If the answer is yes, then there is something wrong in the system which must be checked. First, the program would proceed to stage 958 wherein the register R14 is decreased by 1. This means that the length of the array will be decreased by one EXECUTIVE POINTER length. Then control goes to stage 960 where the question is asked "is R14 greater than zero?". If this is the case, as it should be, control returns to stage 952 to repeat stage 952 and 956. This byte will continue until either the memory block size is greater than the array size, as determined at stage 956, or the array size is zero. If the register R14 is zero, control goes to stage 962 where certain registers are saved. The exact operation at stage 962 and the succeeding stage 964 will be discussed with respect to another phase of the macro-instruction CREATE. However, suffice to say that control eventually goes to stage 966 wherein the question is asked "is register R14 equal to zero or a value other than zero?". Since, at stage 969 it was determined that register R14 was zero, control goes to stage 968 where a message is printed that a screen is too large for the memory block, and termination of the SOLID System would begin at location CL1 in the CONTROL routine.

If the array length is less than or equal to the memory block length then control is transferred from stage 956 to stage 970. At stage 970, the DUN7 is loaded with register R0. This occurs because, if the array length had been decreased through stages 958 and 960, then DUN7 will have changed. The value in DUN7 is then stored in the array at the location designated by register IR5. Thus the value of IR5 is increased by 4 bytes. Thus, IR5 now points at the first EXECUTIVE POINTER or element in the array that is to be created. At the final step of stage 970 the registers R0 to R4 are stored in the array C1.

Next, control goes to stage 972 where the EMPTY is decomposed into its five parts in registers R0, R1, R2, R3, and R4. At this stage, register R4 contains the address of the first unused byte in the memory block. Control then passes to stage 974 where register R14 is set equal to the sum of DUN7 plus $\&ADDL$ plus register R4. Thus register R14 contains the relative address of the last byte in the new array that is to be created in core-memory. Register R15 is set equal to R14 plus SAVEYY minus AYY. In this equation, the original R15 was the length of the memory block, computed in BRY, SAVEYY is the absolute machine address of the beginning of the resident memory-block and AYY is the absolute machine address of the beginning of the M and J arrays. Thus register R15 now contains the relative address of the last byte in the memory block. At stage 976, a determination is made as to whether or not R14 is greater than R15. If R14 is greater than R15, then this means that the new array cannot be created in the resident memory-block. If R14 is less than or equal to R15,

control goes to stage 978 where the register R4 is set equal to register R14. From stage 978 control goes to stage 980 where the macro instructions ASADD performs the task of updating EMPTY with the new value in register R4. After completing stage 980, control goes to stage 982 where registers R0 to R4 are loaded from array C1. Next, control is transferred to stage 984 where register R0 is set equal to register R1 minus 4. Since register R0 originally contained the relative address of the continuation address and register R1 contains the length of the array, register R0 now contains the length of the array minus four bytes.

From stage 984 control goes to stage 986 where contain safety checks are completed. First, the slow memory portion of address EMPTY is compared with the slow memory portion of address EMPTY + &ADDL. If they are equal, this means that no memory block is to be created and, accordingly, at stage 988, EMPTY is set equal to EMPTY thus updating the fast portion of EMPTY. If the two are not equal at stage 986, control goes to stage 990 wherein the contents of register R0 are stored in the DUM1. Then, at stage 992, the macro-instructions LARGEXC is performed. In LARGEXC, the entire array except for the first four bytes are zeroed. At this point the new array has been created. After completing stage 992, control goes to stage 994 where register R15 is set equal to SAVEYY, the absolute address where the CURRENT address is recorded in the memory-block. Next, the updated composite address EMPTY is stored in the location specified by register R15. After completing stage 994, control returns to the TBADD macro.

The more difficult problem occurs when, at stage 976, it is found that there is insufficient space in the resident memory block to create the new array. When this occurs, control goes directly to stage 996.

At stage 996, the macro-instruction LINKHOLE (defined previously as macro 115) determines whether or not there are any unused arrays in the memory block which can be used for the new array. If the answer is yes, control would go immediately to stage 990, because the array already exists in the memory block. If the answer is no, control goes to stage 998 where a COMPARE macro-instruction compares the slow memory address portions of EMPTY and (EMPTY + &ADDL). If they are equal, this means we have not yet computed the slow memory address of the new memory block that is to be created. If they are not equal, this means that the slow memory address of the new memory block that is to be created has been computed and, accordingly, control goes directly to stage 1000 and, thence, to stage 1002. If a new slow memory address had not been computed, stage 1002 will be reached at the end of the branch which begins at stage 962. For purposes of clarity, it is assumed that the parts of EMPTY and (EMPTY + &ADDL) were not equal at stage 998. At this point the slow memory address of the new memory block has been computed and, accordingly, at stage 1000, register R15 is loaded with the address in SAVEYY, which is the address of the location at the foot of the M and J arrays. Then, the EMPTY address is stored at the foot of the MJ array. Next the MSIGNAL 80 bit is turned off to indicate that the request for a newly created memory-block is being executed. Also, the request address (ADDRESS) is set equal to EMPTY + &ADDL. Register R15 is then updated to (BW_X+76). (BW_X+76) contains the address in the resident memory block where the new composite address was inserted. (see stage 900 in FIG. 21). Then, in this location in the last memory block, address EMPTY + &ADDL is inserted. Thus, the old memory block has now been updated by inserting the new value of the address of the new array. From stage 1000 by inserting control goes to stage 1002. However, before discussing the operation at stage 1002, let us consider the case when, at stage 998, the slow memory addresses of EMPTY and (EMPTY + (ADDL)) were equal. In this case, control goes to stage 962 where register R15 is loaded with the address in SAVEYY, which is the address of the composite address at the foot of the M and J arrays. Then, EMPTY is stored in the location at the foot of the

M and J arrays. Further, R15 is loaded with the address contained in (BW_X+76), and the composite address in the resident memory block that is specified in register R15 is set to zero.

From stage 962 control goes to stage 964 wherein the macro instruction APART separates the five components of the composite address EMPTY + &ADDL and records them in the five registers R0 to R4. Then, at stage 966 the question is asked "is register R14 equal to or greater than zero?" If R14 is equal to zero, the abort procedure which begins at stage 968 is executed. If R14 is not zero control goes to stage 1004. For purposes of clarity, stage 1004 has been shown as a block grouping which will effect the following:

Stage 1004 is a series of steps which compute the five components for the new composite address for the memory block that is to be created. These five components are as follows:

1. &RD which is the device type number (disc. tape. data cell, etc.) which is recorded in register R0.
2. &RDO which is the device number recorded in register R1.
3. &RTRK which is the beginning track on the device recorded in register R2.
4. &RCYLN which is the beginning cylinder on the device recorded in register R3.
5. &RFMADD which is a relative fast memory address in core where there is space to create an array.

Because a new memory-block is being created &RFMADD is set equal to the address of the foot of the M and J arrays, plus &ADDL. If the device is a tape rather than a disk, it is not necessary to have both track and cylinder numbers, but only a record number and, accordingly, &RTRK and &RCYLN together contain a single record number recorded in registers R2 and R3. If, in computing the new components for the composite address of the new memory block, it is determined that there is, in fact, insufficient equipment to store the new memory block; for example, one has run out of disk memory and there is no other available virtual memory, then a message is printed to notify the machine operator that he needs to obtain additional storage devices for virtual memory. This abort procedure is completed through the Global Memory component SMEMORY. The resident memory-block is saved and the operator is notified that additional devices must be obtained. Additionally, the system also advises the operator as to what type of devices are to be preferred. The declared universe of the system, what is the amount of devices available to the system, is defined in a single macro-instruction called BEGINS.

From the block of stages 1004 control goes to address (EMPTY+&ADDL) is assembled from the five registers R0 to R4. Then R4 (= &RFMADD) is incremented by &ADDL at stage 1008. It should be understood that in stage 1004 the register R4 contained the value in SAVEYY and it is necessary to increment it by the amount &ADDL to get the relative address of the first byte where the new array can be created. After completing stage 1008, control goes to stage 1010, where the composite address is ADDRESS updated with the new value in register R4. EMPTY is set equal to ADDRESS at stage 1012. Additionally, the EMPTY address is inserted in the resident memory block at the address in (BW_X+76) as was done with respect to stage 1000.

From stage 1012 control is transferred to stage 1002 wherein registers R0 to R4 are loaded from the array C1 and then control goes to stage 936 in the TBADD macro. This return to GLOBAL in TBADD saves the resident memory block. After completing the save operation in SMEMORY, control returns from the TBADD macro-instruction to CREATE again. In CREATE, at stage 976, a determination is made as to whether there is sufficient space to create the new array in core and, accordingly, it does so at stages 976, 980, 982, 984, 986, 988, 990, 992 and 994 and then exits from CREATE.

GLOBAL MEMORY

The GLOBAL memory component (hereinafter called SMEMORY), transfers memory blocks between the AUXILIARY FILE, which can be located on any combination of devices, and core storage. The AUXILIARY FILE must be separated by definition from bulk storage. That is, information which is utilized to address information in the bulk storage will be found in the AUXILIARY FILE or in core storage. The AUXILIARY FILE is, normally, placed on the disk storage of the computer and aids in finding the address of a particular group or segments of information. The DCB (macro-instructions of the IBM 360 which define the characteristics of the data set on a peripheral storage device) and read/write instructions of each new device that is made a part of the GLOBAL MEMORY are incorporated in the DCBMEM macro-instruction, which is used in SMEMORY. The storage capacity of each new device must be given in the macro BEGINS. This information will be used by the computer to assign new memory blocks when all previously assigned devices are full. Thus, by modifying SMEMORY the GLOBAL MEMORY is easily extended to include new storage devices when they are added. SMEMORY notifies the operator when the GLOBAL MEMORY is full. Because the existing storage is not altered in any way when the GLOBAL MEMORY is extended, this component (SMEMORY) permits the simultaneous growth of the hardware and retrieval systems.

There are two parts, A and B, of the SMEMORY component. Part A supervises the AUXILIARY FILE while the information paths are being traced or purged or created or updated. Part B is entered when the job stream is terminated. Its function is to save (if necessary) the resident memory block and to punch (if necessary) the first part of the AUXILIARY FILE. This punched card deck, which contains the M and J subarrays, will preface the input deck for the next job stream.

It should be noted that the M and J subarrays, although part of AUXILIARY FILE, are always in core storage. The M and J subarrays are in core storage because all information paths start with these subarrays and, thus, it is possible to save considerable time by avoiding the necessity for fetching information from disk storage to start these paths.

At every step in the retrieval package, safety procedures are executed which assure that the memory block in the AUXILIARY FILE will never be damaged by program, input, operator or machine errors. Only the physical breakdown of the virtual memory hardware components can damage the AUXILIARY FILE. No attempt will be made to describe the numerous safety procedures. The two parts of SMEMORY are described next.

The flow chart of Part A of SMEMORY is shown in FIG. 10. Before explaining the flow chart, it should be understood that the input data for SMEMORY contains a composite address whose slow-memory part specifies the location of a memory block in the virtual memory storage. The fast memory part of the composite address specifies the location of the requested information when the memory-block resides in core-storage. The composite address is normally six bytes long and it is used to determine the course of action of GLOBAL MEMORY. For example, in the search procedure it is necessary that the machine believe, at all times, that the information it is looking for is in core storage. It is a purpose of GLOBAL MEMORY to find the information no matter where it may be and transfer it to core storage whenever it is needed. The composite address discussed above contains, in its first three bytes, information relating to non-core storage. The first four bits, designates the type of device of non-core storage where the memory-block can be found. For example, the size permutation and combinations of the first four bits will include codes for tape storage, disks, drum storage, etc. This will key the next four bits to know on which particular one of a possible 16 different units of tape, disk, or drum storage the memory-block can be found. The next 16 bits are divided into six bit

and ten bit sections which together designate an address on the particular storage element. For example, if the storage element is a disk, the next sixteen bits contain the track number on the first six bits and the cylinder number on the next ten bits. If the storage element is a tape, then the 16 bits together specify the record where the memory block begins.

The remaining three bytes in the composite address contains the core address where the particular information can be found when the memory-block resides in core.

It should be noted that there is one byte of information known as MSIGNAL, which is continually being updated, and it specifies the type of action that is to be taken by the Global Memory Component (SMEMORY). If the right most bit of MSIGNAL is a "one", this means that the resident memory block or the permanently resident part of the AUXILIARY file has been altered by inserting a new EXECUTIVE POINTER and by creating a new subarray. Another portion of storage which is checked by SMEMORY is the composite address called CURRENT. If CURRENT equals FFFFFFFF (a condition which is placed into CURRENT in SSTRATECL when there is no AUXILIARY FILE) the AUXILIARY FILE does not exist, and a new memory block will have to be created in core-storage. When CURRENT equals zero, that means that there is no resident memory block, and one will have to be fetched from the virtual memory or created in core-storage. These values of CURRENT are set when the AUXILIARY FILE is initialized at the beginning of each job stream.

In one version of the SMEMORY component, the virtual memory was on IBM 2311 disks which had two distinctly different write modes (i.e., new write and rewrite). The 02 or next to last, rightmost bit of the MSIGNAL byte designates which of the two write modes is to be used to store the resident memory block at the location in virtual memory specified by the slow-memory part of the composite address CURRENT.

The flow diagram for the first part of SMEMORY is shown in FIG. 10. The first step in SMEMORY is to check the rightmost of 01 bit of MSIGNAL at stage 322. If the MSIGNAL 01 bit is zero, this means that the memory block was not changed and, therefore, there need be no read out into virtual memory. If the MSIGNAL 01 bit was one, then there must be a readout into virtual memory.

If the MSIGNAL 01 bit is "one", the next step is to check the address CURRENT to determine whether or not there is an AUXILIARY FILE, by subtracting from CURRENT, FFFFFFFF (CONM.). If there is no AUXILIARY FILE, and the answer is, therefore, "zero", then control continues to stage 326 wherein a determination is made as to whether one wishes to create a memory block in core or whether one wishes to read it from virtual memory. This determination is made by testing the MSIGNAL 80 bit which, if it is "one" indicates that you wish to create a new memory block in core and, if it is "zero", indicates that you wish to read a memory block from virtual-memory into core. If the MSIGNAL 80 bit is "one" then the next step is to execute stage 328 wherein the machine updates the component address EMPTY, a position in core storage which contains the composite address of the next available position in virtual memory where a new memory-block can be stored. This EMPTY address is set in CURRENT. Of course, the MSIGNAL 80 bit is turned off and the MSIGNAL 02 bit is turned on to indicate that this newly created memory-block has never been written into virtual memory. When the time comes to write this memory-block into virtual-memory the 02 bit of MSIGNAL will indicate that the new write operation must be used.

After completing step 328, control goes to stage 330 wherein the MSIGNAL 01 bit is turned off to indicate, that at this point, there has been no modification of the new memory block that is now resident in core.

If, at stage 324 the answer was something other than zero, control would have been transferred to stage 332. There CURRENT is compared to zero. If CURRENT were zero, then the procedure set forth with respect to stages 326, 327

and 330 would have been executed in substantially the same manner. However, there is one variation that could occur. That is, if a stage 326 it was determined that an existing memory block was to be read into core-memory from the virtual memory address specified in ADDRESS. In this case the MSIGNAL 80 bit would have been zero and control would have gone to stage 334, where the memory block which had been requested during the SEARCH procedure is read into core storage. It should be understood that in the TBADD macro in the SSEARCH component it was determined that the resident memory block is not the correct memory block for a particular search and, in fact, a different memory block was requested by SSEARCH. After the steps at stage 334 are completed the MSIGNAL 02 bit is turned off to indicate that the new memory block has been read from peripheral storage.

It thus should be noted that when the MSIGNAL 02 bit is off, it indicates that a rewrite procedure must be used when the resident memory-block is transferred back to its specified location in virtual-memory. If the MSIGNAL 02 bit is "on" this would indicate that the resident memory-block is new and the new write procedure must be used to transfer it to the AUXILIARY FILE. After completing stage 336, control goes to stage 330 wherein the MSIGNAL 01 bit is turned off, meaning that the new resident memory block has not yet been changed, then CURRENT is now loaded with the address of the memory block taken from ADDRESS.

If CURRENT is not zero, it specifies where the resident memory-block should be stored in the virtual memory. It must be understood that the memory block is core-memory, which originally came from the virtual memory or it was created, has been modified before SMEMORY was called. The virtual memory must be updated with this changed resident memory-block before a new memory-block is transferred to core-memory or created. Thus, if there is an address in CURRENT, control goes to stage 338 wherein the MSIGNAL 02 bit is again reviewed to determine whether this is a new write or a rewrite procedure. If it is a new write procedure, then the resident memory block has not been transferred to virtual memory before. Thus, the resident memory block must be written in the virtual memory for the first time. Its previously assigned virtual memory location, is found in the slow memory part of the composite address CURRENT.

If this is a new write procedure, then the MSIGNAL 02 bit is "one" and the program continues to stage 340 wherein the memory block in core is written for the first time into the virtual memory at the location specified by CURRENT. If the MSIGNAL 02 bit is zero a rewrite procedure is used. Thus, control goes to stage 342 wherein the rewrite procedure transfers the memory block from core back to its address in the virtual-memory that is specified in CURRENT. From stage 342, control goes to stage 326. It should be noted that the entire purpose of SMEMORY is to give to SSEARCH or a similar programmatic procedure a new memory block whenever it is needed and take care of the procedural functions that are necessary to preserve the core-resident memory block. The stages 322, 324, 332, 338, 340 and 342 have taken care of this procedural function. At stage 326, a determination is made as to whether a new memory block is to be created in core storage or whether a memory block is to be read from virtual memory into core storage. If a new memory block is to be created in core storage, then control is transferred to stage 328. If the required memory block already exists it is read into core storage at stage 334.

COPAK COMPRESSOR

COPAK is a high-speed, multistage, compressor-decompressor software package that can be used to compress arbitrary bit-strings by reversibly removing redundant information. Decompression occurs without losing a single significant binary-bit of the original string. Except for minimal commands, both the compressor and decompressor parts of COPAK are fully automatic. COPAK operates independently of both the data-base and the information-content.

COPAK can be used for supervision of bulk storage and for transmission of data in communications and computer networks. A more effective role can be achieved by implementing COPAK on a small, high-speed, low-cost, specially designed dedicated computer. This unit could be interfaced with computer/communication networks or used on a stand alone basis for compressing and decompressing information. As such, it is highly useful as a buffer-converter between various combinations of computer systems and input-output devices. Careful considerations indicate that this low-cost unit could have a throughput between three and thirty times faster than COPAK on the IBM 360/67. The throughput on the 360/67, inclusive of both input and output times, lies between 40K and 900K BAUDS, with the optimum near 550K BAUDS.

COPAK has been described in detail hereinunder as a machine process in the form of a combination of a computer software package and a general purpose digital computer of adequate capacity and versatility. In fact, the COPAK package described hereinunder has been utilized in conjunction with general purpose machines such as IBM 360/67 and 360/40. It is noted that when carrying out COPAK, general purpose machines perform a specialized task and only those components of the warehouse of components contained therein which are ordered and organized by the COPAK act, as controlled by COPAK. In effect then, the combination of COPAK and a general purpose machine becomes a special purpose digital computer. Alternatively, the flow diagrams and the program steps and instructions described hereinunder in detail comprise a teaching of combining existing hardware components, such as those used in the general purpose machines mentioned above, under control of COPAK, to arrive at a special purpose computer carrying out COPAK. The process of so combining existing components as dictated by COPAK is an engineering task for one of ordinary skill in the art and does not involve inventive efforts. Although COPAK is usually referred to as "software package" hereinunder, its function as a special purpose machine when combined with a general purpose digital computer should remain clear.

The communications and computer industries have placed great emphasis on engineering research which can increase the "efficiency of networks" by increasing the channel capacity or speed (of transfer) or by reducing the proportion of redundant signals. In recent years the storage capacities of peripheral devices (like disks, drums, data-cells, tapes and cards) have been enormously increased by advances in engineering technology. Some special recoding techniques that save storage and/or lower transmission costs have been widely used. However, these special techniques are of limited usefulness because they apply to particular devices and/or they are not independent of the data base. There appears to be no report of a major effort to devise general software packages that can increase the information content per unit of the information itself. Such packages could artificially increase the storage capacities of existing facilities and lower transmission costs.

To be of more than transient usefulness these general software packages should meet as many of the following specifications as possible.

- i. with a minimal number of commands the software packages should be capable of handling any binary coded information. This means that compression and decompression must be independent of the data-base or the information content.
- ii. Compressed information should be automatically decompressed back to the original whenever it is needed.
- iii. For communications networks the rate of compression should not be less than the rate of transmission. The decompression rate (at a receiver station) should not be slower than the rate of compression.
- iv. There must be checks to ensure that errors in the compressed information will be detected before or during decompression.
- v. The effectiveness of the proposed package for increasing the capacity of existing storage devices will be deter-

mined by several factors. Some of these are: the access and transfer times of the peripheral equipment; the speed of decompression; the frequency that the particular information is used. Obviously, infrequently used information can be highly compressed to release storage that would not normally be available.

To be fully effective in both storage and communications applications the general software packages should have adjustable parameters which would permit the user to stipulate the maximum amount of time that can be devoted to compressing (or decompressing) information.

The COPAK compressor meets the five specifications just listed. The computer speed and two variable parameters determine the rate of both compression and decompression. On the IBM 360/67 COPAK compresses information at rates of 40,000 to 900,000 BAUDS. Decompression is at least one and a half times faster.

Definition and Commands

The two parts of COPAK (Compressor and Decompressor) each have two stages (SNUPAK and SANPAK). The COPAK compressor handles the information as strings, segments and substrings. A string of information can be divided into non-equal segments. Segments can be sub-divided further into substrings. The lengths of strings, segments and substrings is a user option. The numeric stage (SNUPAK), which can process any information designated as binary coded numbers, handles segments of information at the substring level. The alphanumeric stage (SANPAK) handles strings of information at the segment level. As described hereinunder, COPAK processes one segment per string (i.e., string = segment) with each segment containing between one and twenty substrings.

The Device Command LLENGTH, which must have a value less than 256, specified the number of bytes in the label or key which may preface each string. This information is not processed by COPAK. The leftmost LLENGTH bytes are removed from the first segment and the shortened segment is processed by COPAK. The structure of the stored composite string is:

LLLENGTH	Label or Key	Processed New Segment
1 Byte	LLLENGTH Bytes	

The String Command MODE determines which part of the COPAK compressor is to be used, i.e., MODE=0, decompress; MODE ≠ 0, compress. Three other String Commands (LEXCON, LEXPCH and LEXMODE) are associated exclusively with the alphanumeric compressor stage (SANPAKC) of COPAK. They will be discussed later.

The three Substring Commands (NV, SOS and LSX) are used extensively in the numeric stages (SNUPAK) of both parts of COPAK. NV, which is entered once for each segment, is the number of substrings (maximum 20). One SOS command and LSX command are entered for each substring. Together they determine the entry format-type of the substring and the path that is to be taken through the compressor parts of COPAK. The entry format-type for each substring is stored in the compressed segment as a four-bit format code. This is used to produce hard copy when the segments are retrieved. Format codes used are: A=1; I=2; E or F=3; X=4 (printed in the hexadecimal format (B)). Here X is the IBM 360 column binary. The substring commands are not entered if MODE=0 (i.e., for retrieval).

Overview of the COPAK Compressor

In the compression mode (MODE ≠ 0) the compressor parts of COPAK construct a completely self-defined string which contains the label or key; format codes; string structure

(i.e., segments and substrings retain their identities); and sufficient information to ensure that errors will be detected during decompression. This information, exclusive of the label, is normally less than 24 bytes per segment. It is added even if there is no actual compression. The decompressor parts of COPAK unscramble the self-defined string to obtain the identical original information. The error-checks are executed during decompression. If an error is found, control goes to a location where error-correcting procedures and/or retransmission commands can be executed.

The status of each segment is recorded in a four-byte work area (PARM) which is updated whenever the segment is altered. The structure of PARM is:

Number of Bytes in Segment	Number of Substrings
3 Bytes	1 Byte

The status of each substring in a segment is indicated by a four-byte word (SOS) which is updated whenever the substring is altered. The substring composite control words (SOS) contain four items of information thus:

Sign	Number of Bytes in Substring	Format Code	NDR
1 Bit	23 Bits	4 Bits	4 Bits

Here NDR is the "depth of representation" that is computed in the differencing procedure (of NUPAKC).

A flow diagram for the COPAK compressor is given in FIG. 11. As a segment of information enters the computer its status-of-substring control words (SOS) are changed, to the form shown above, and the status-of-segment control word (PARM) is constructed. At this stage the sign of SOS and the values of both NDR (in SOS) and LSX together determine how a substring will be handled by the compressor part of SNUPAK. After some preliminary processing of the segment in SANPAKD it is processed, one substring at a time, by SNUPAK. In this step the SOS composite words are updated and a new status-of-segment control word (PARM) is constructed. The sign of the first SOS and the number of bytes in the segment emerging from SNUPAK are transferred to JII, which is the temporary control variable for SANPAKC. In the final step of SNUPAK the control words (PARM and SOS), check information, and other data needed by the decompressor part of SNUPAK are inserted at the head of the segment.

The information in JII is used by the alphanumeric compressor part (SANPAKC) to decide whether or not compression of the newly defined segment is to be attempted. If compression occurs in SANPAKC, information, which is used by SANPAKD during decompression, is inserted at the head of the segment. In the final step of SANPAKC, four bytes of control-information are inserted at the head of the segment. The label or key information is then inserted preceeding the control-information at the head of the segment, (see Device Commands). The structure of the four byte word of control-information is:

Number of Bytes in Segment Emerging from the Alphanumeric Compressor (SANPAKC)	NL
3 Bytes	1 Byte

Here NL is the number of redundant bit-patterns removed by SANPAKC.

A single string command suffices to bring about decompression of a stored segment. When this command, `MODE=0`, is used the label or key is first removed from the head of the segment. Then the four bytes of control information are extracted and the following steps are executed:

Step a. The compressed segment is decompressed with the alphanumeric decompressor (SANPAKD) if `NL ≠ 0`.

Step b. The control information that was inserted after processing by the SNUPAK compressor is extracted.

Step c. The substrings of information are decompressed one at a time by the decompressor part of SNUPAK.

Step d. The label or key, previously removed, is replaced at the head of the decompressed string.

Error-checks occur at every step of this decompression procedure. Thus a segment with `N` substrings has $(N+2)$ absolute error checks. Also, there are an additional 15 error-checks which are made during the decompression by SANPAKD and SNUPAK. Moreover, the conventional CHECKSUM can be used as an additional error check. If errors are found, the decompression is aborted and control goes to location RTRANSMIT, were error-correcting and retransmission procedures can be utilized.

The compressor parts of COPAK have incorporated fail-safe procedures which prevent the inadvertent destruction of information. For example, if SNUPAK is told to compress text or binary information as integers it will abort and change the processing commands to execute SANPAKC without destroying the data.

SANPAKC

INTRODUCTION

SANPAKC is the Macro instruction used for alphanumeric compression of information within the COPAK system.

DETAILED DESCRIPTION

The data on which SANPAKC operates is in alpha-numeric form, in strings of units. In the embodiment described hereinunder the units are conventional 8-bit bytes. It should be clear however, that SANPAKC, as well as the complete COPAK package, can be equally applicable to machine using units other than bytes.

Two distinct types of compression are carried out consecutively. Each may be carried out either in Fast Mode or in Slow Mode.

In Type 1 compression, the string is searched for identical patterns of two or more contiguous units. If such identical multi-unit patterns are found, they are deleted from the string and decompression information which takes less space but has sufficient information content for subsequent decompression of the string to its original form is added to the strings.

In the Slow Mode of Type 1 compression, the scan for identical multi-unit patterns is carried out by comparing a pattern of several contiguous units of the string with all other patterns in the string of like size. In the Fast Mode, this is carried out by comparing previously chosen patterns which are believed to occur often with patterns of like size in the string.

In Type 2 compression, which is executed after the completion of the Type 1 compression, the compressed string is scanned for individual units which occur more than a certain number of times. If such units are found, they are deleted from the string and decompression information is added to the string, but only if the length of the decompression information is less than the length of the deleted information.

As a brief qualitative description of a particular example of carrying out Type 1 compression in the Slow Mode, a string of 1,000 bytes is scanned such that the numerical value of each byte is used to address a 256-byte table in which each location corresponds to a unique one of the 256 possible combinations of the eight binary bits of each byte of the string and each location of the table acts as a counter for the number of times it has been addressed. After the last byte of the 1,000 byte string has been used as an address in this manner, the table is ex-

amined for locations which have not been addressed. The address values of these locations, if any, are stored consecutively in one area of LEXICON table and are called Type 1 codes. It will be appreciated that these Type 1 codes represent bytes which are not present in the 1,000 byte string. Additionally, the address values of the locations of the 256-byte table which have been addressed more than a certain number of times, for example, more than 34 times, are stored in another area of the LEXICON table and are called Type 2 codes. These Type 2 codes represent bytes which occur very often in the 1,000 byte string and are likely candidates for deletion. Next, a pattern of contiguous bytes from the string, for example, the first 12 bytes, is compared with all other patterns of the same format in the string. Identical patterns found in this manner are deleted from the string and are replaced by a Type 1 code from the LEXICON table, but only if actual saving in string length would result from this process and only if a unique Type 1 code is available for each group of like patterns. The same Type 1 code followed by the 12 bytes of the deleted pattern is inserted at the beginning of the string for later use in decompression. The process is repeated for different patterns of contiguous bytes for as long as there are unused Type 1 codes and for as long as saving in length of the string can be achieved. When a pattern has been found to occur several times in the string and has been deleted therefrom, it is stored in a PCORDS table which contains patterns likely to occur often in similar strings. A savings ratio is associated with that pattern to indicate the degree of compression achieved by the use of that pattern.

In Slow Mode of Type 2 compression, a portion of the compressed string, for example, a portion of 256 consecutive bytes, is examined for redundancy of particular individual bytes anywhere in the portion. If a particular byte selected from the Type 2 code in the LEXICON table is still found to occur in that portion more than a certain number of times, a 256 bit map is constructed in which each bit location corresponds to a byte of the examined portion of 256 bytes. The bit map serves as a record of the byte position in which the particular byte was found. The redundant bytes are then deleted, the string is closed in to take up the vacated space and the bit map together with the value of the deleted byte is added to the string after the size of the bit map is minimized. The value of a deleted byte and the savings ratio associated with it may be added to the PCORDS table.

In the Fast Mode of carrying Type 2 compression, the portion of 256 bytes from the string is checked for the occurrence of bytes selected not from the LEXICON table but from previously stored bytes in the PCORDS table.

In both modes of both Type 1 and Type 2 compressions, continuous track is kept of various string characteristics for the purpose of insuring complete reconstruction of the compressed string and for the purpose of providing adequate error detection features. A more detailed explanation of the SANPAKC compression, with particular reference to the flow diagrams in the drawings, can be found below.

The flow chart of SANPAKC is given in FIGS. 1A, 1B and 1C. The first step performed in the Macro SANPAKC is to initialize all the registers and counters in that portion of the computer which is being used for alphanumeric compression. The next program instruction at step 11, is to check whether the MODE Command is set equal to "0" or not. "0" means that no compression is desired, and "1" means that compression is desired. The "0" value would occur when the system was in a retrieval mode and therefore, compression would not be required. If the machine was in the storage mode (`MODE ≠ 0`) compression might be desirable. The next step in the program is to determine if the variable JII is greater than zero. If JII is equal to or less than "0", that means that no compression is desired. If it is greater than "0" then compression is desired. The only way that JII would be negative is by setting it to a negative value prior to entering SANPAKC indicating that compression, by SANPAKC, is not desired. Therefore, even though the computer was in the store mode, one could prevent

compression of the information. If JII is positive, the program begins compression at stage 12 in the flowchart of FIG. 1A. Although the flowchart shows various stages in the program, it is understood that this is just a means of designating a group of steps to be completed at a particular point in time. The program listing is IBM Assembly language and is set forth at the end of the written description. At stage 12, the program initiates the steps of finding all available codes and then storing the available codes in a location named LEXICON. Thus, this step is achieved as follows:

a. A thousand byte string of information is scanned one byte at a time starting from the first byte. The numerical value of the first byte is used to address a location in a table of 256 byte positions corresponding to the 256 different bit configurations possible in a single byte of information. A count is initiated at that particular position, to indicate that the particular byte has been found once within the thousand byte string. The next byte within the thousand byte string is similarly used to address a location in the 256-byte table (i.e., by adding the numerical value of the scanned byte to the base address (beginning address) of the table) and a count is added at that particular location. This is continued throughout the one thousand bytes in the string. Where bytes within the thousand byte string are identical, the count at the particular location in the 256-byte table will indicate that the particular byte of information appears more than once in the thousand byte string. The counters in the 256-byte table are not permitted to exceed the value 255 so that an absolute frequency count of the number of occurrences of a particular byte is not achieved if the byte occurs more than 255 times in the string. However, in going through any thousand byte string, there will be many of the positions or locations in the 256-byte table which will not be utilized as there is no byte in the thousand byte string corresponding to that location. Those locations which are "0" in the 256-byte table are determined by scanning the table. The corresponding numerical values (<255) of the 256-byte table positions (i.e., the number of bytes past the beginning of the table) are then transmitted by means of program instructions to the 256-byte array named LEXICON and stored in consecutive byte locations of LEXICON to act, at a later stage, as possible Type 1 code numbers (for Type 1 compression) for groups of bytes to be compressed. The count in each of the individual positions in the 256-byte table where there has been one or more counts is also scanned to determine whether any particular location shows more than 34 counts. This indicates that the particular byte is a candidate for Type 2 compression (which will be described below). Any location which shows more than 34 counts is also stored in LEXICON to act later as Type 2 codes. LEXICON has only 256 positions of storage. However, the positions in the 256 byte table mentioned above which show more than 34 counts are stored starting at the position 256 of LEXICON and working backwards. For example, if position 193 in the 256-byte table were to show more than 34 counts it would be placed in position 256 in LEXICON and if position 232 in the 256-byte table were also found to have more than 34 counts it would be stored in the 255th position in LEXICON. It should be noted that it is impossible for the Type 1 codes stored in LEXICON to overlap the Type 2 codes stored in LEXICON as these positions are derived from the 256-byte table mentioned above.

The next group of instructions in SANPAKC is shown at stage 14 of the flowchart. At this point the question is asked "are there any codes available?". At stage 14 LEXICON is checked at the front end thereof to see whether any Type 1 codes have been stored as a result of the steps taken at stage 12. It should be noted that when the available Type 1 codes were stored in LEXICON, a count was made of the number of available codes thus stored during the steps defined with respect to stage 12. When the step set forth in stage 14 occurs a check is made to determine whether the last mentioned counter has counted any available Type 1 codes being stored in LEXICON. It is possible, if the thousand byte string contains 256-bytes of information different from each other that

there may be a count in each one of the 256 counters in the 256-byte table mentioned previously. Accordingly, there are no available Type 1 codes stored in LEXICON. If the answer at stage 14 is that there are no available codes, a different type of compression, namely Type 2 compression, is used. However, Type 2 compression will be described more fully below.

If the answer at stage 14 is "yes" then the program continues to stage 16 where the determination is made as to whether the compression technique should be completed in the "Fast-Mode" or the "Slow-Mode". The Fast-Mode will be discussed separately as most of the operations in the Fast-Mode are included in the Slow-Mode and, in fact, the Fast-Mode may be considered a special case of the Slow-Mode. A full description of the Fast-Mode will be set forth below after consideration of the operation in the Slow-Mode. Thus, if the answer at stage 16 is "no", the compression process continues in the Slow-Mode and the program continues to stage 18 where counters R and RM are set. Counter R is a counter set to contain the value of the largest number of contiguous bytes in the pattern for the occurrence of which the string will be searched later. RM is equal to R-1. For purposes of the system, in actual use, R has been set at a maximum of 12 bytes and, of course, then RM would equal 11. After the counters R and RM are set, the program then proceeds to stage 20 where a scan of the input string is initiated to locate the occurrences of redundant patterns. A pattern is a contiguous grouping of bytes of variable length. Initially, we will consider groups of contiguous bytes of length 12 bytes. At stage 20, as shown in the diagram, there is a set of instructions to be followed entitled CCD: a number JII which designates the length of the string of information at any given point in the compression process (please note that in the example given above the string length was initially 1,000 bytes); and a number CS3 which is a number designating the number of bytes after the starting point at which redundancy is to be checked. The instruction CCD is described below.

A first contiguous group of bytes having a length determined by the counter R (in the first instance 12 bytes) is recorded and compared along the length of the string moving one byte at a time to find how many times and where the like of the first contiguous group of R bytes can be found in the string. If there is no other contiguous group of R bytes exactly like the first group being sampled, that result is transmitted to stage 22 of the flowchart where the question is asked as to whether a saving can be achieved by removing the redundant pattern of bytes. The answer to this question is determined by examining whether the formula $R+2N(R-1)$ is satisfied or not. N is the number of identical groups of bytes found during a single scan of the string. Since only one group had been found during the scan, $R+2=14$ and since $N(R-1)=11$; the answer to the question as to whether there was a saving is obviously "no". Since the answer is "no", the program continues by transmitting this response to stage 24 where a determination is made as to whether the operation is in the Fast Mode. As was discussed previously, this operation is being accomplished in the Slow Mode and the answer at stage 24 is also "no". If the answer were "yes", another operation would take place. This operation will be discussed in conjunction with the description of the Fast-Mode. Since the answer is "no", the program continues by transmitting this information to stage 26 wherein steps are taken to determine whether additional comparisons can be made. This determination is made by comparing whether $CS3+1+2R$ is less than JII. This equation determines whether the sampling has reached the end of the string since many more comparisons would be useless as there are not enough bytes left in the string to be able to effect a savings. Since, in this case, CS3 is "0" and the group $CS3+1+2R$ is equal to 25, and this is certainly less than 1,000 (JII) the answer at stage 26 is "yes". (What happens when the answer is "no" at stage 26 will be discussed below.) First, since the answer at stage 26 is "yes", program control is returned to stage 20 where CS3 now has the value of "1" by the addition of one byte and CCD continues starting with the

next byte of the groups of 12 bytes to be compared. Thus, starting at one byte past the first point of the string, the succeeding group of 12 bytes is checked for redundancy going byte by byte along the string to determine whether there are any similar groupings of 12 byte patterns along the length of the string. Assuming in this instance that three such groupings are found along the length of the string, then this information is passed to stage 22 and entered into the formula $R+2 < N(R-1)$ or "is 14 less than 33?". The answer obviously is "yes" and, accordingly, this information is passed on to stage 28. At stage 28 the counters associated with the compressor system are updated. That is, a counter, which for purposes of notation is designated as CS8, counts the number of compressions which have taken place on this particular string of information. That is, since this is the first compression which is to take place on this particular string of information the counter will be set equal to 1. An additional counter CS4 is actuated to count the number of compressions which have been taken with R bytes, that is, with 12 bytes, and, accordingly, since this is the first compression with 12 bytes this counter is also set equal to 1. Additionally, the counter CS6 which is associated with LEXICON to determine the spot where one will get a Type 1 code from the LEXICON code array is set. In this case this counter counts "1" and selects the first available code in the LEXICON code array which was set in the manner described with respect to stage 12. At this point, having selected a code from LEXICON, CS6 is now set at "2" so that it is ready to receive at a future time a request to select a new code from the second spot in the LEXICON. The next step is to move to stage 30. For consideration of what happens at stage 30, the effect of the program instructions in CCD at stage 20 must be understood. Each time that a redundant group was found during this stage, the addresses of the redundant groups were stored in core memory beginning at location TL, (which is an area of core memory). The address of the first redundant group, namely the first and original pattern, was stored at location TL in core memory and the address of each additional redundant group was positioned in increments of four bytes with the array TL. Each address stored in TL is located at a specific displacement past the beginning location TL (i.e.a multiple of four bytes past the beginning). The displacement where the address of the last redundant group can be found is set in a register IR4. For example, if there were three redundant groupings, the register IR4 would have a value equal to "8". It must be understood that the value "8" is in relative terms and the core address of the beginning of the TL array must be added to the displacement value (in IR4) in order to compute the absolute machine address desired. Thus, if the value in IR4 was "8", the absolute machine address is the initial address of TL in the core plus 8. Further, it should be noted that TL has physical storage limitations and can store addresses for a maximum of 200 repetitions of a pattern during any given pass. Of course, if the length of the string is maintained in reasonable bounds, this limitation should not be reached in the ordinary course of compression, but provision is made in the program for stopping the compression should there be an attempt to store more than 200 addresses of redundancies in the TL array.

At stage 30, the address of the match or redundancy from TL(IR4) (the last match found in the string) is loaded into a register IR2. Note that in this contact TL(IR4) means the initial core address of the TL array plus the displacement value past the beginning of TL which is recorded in register IR4. That is, the address of TL plus the contents of register IR4 gives the machine location where the desired information is to be found. Then, after IR2 is loaded with the address from TL(IR4), the program moves to stage 32 wherein a series of instructions named SAM are executed. This set of instructions first substitutes, in the string of information, a one byte code in place of the redundant pattern whose address is recorded in register IR2. The replacement code is the code from the LEXICON array pulled out at stage 28. This one byte code is then placed at the address contained in register IR2 in place of the

first byte of the redundancy. Then, a determination is made as to the number of bytes in the redundant pattern, namely R, the address of the redundant group, namely the address in IR2, and the length of the string before compression, namely JII. Then the remainder of the string following the last byte of the redundant pattern is moved to close the space between the newly added code information and the remainder of the string to compress the string by an amount equal to $R-1$ bytes. At this time, JII is changed to reflect the compression of the string by an amount $R=1$ bytes. It would be understood that the compression is $R-1$ as it was necessary to add one byte of information to account for the space required to store the replacement code. If the code had not been added, JII would have been reduced by R. It should be noted that at times, when Type 2 compression is being utilized (to be discussed below) no code information is placed in the space vacated by the matched grouping and, in such cases, JII would, in fact, be reduced by R. Since IR4 as set forth above is no longer pointing to the address of the last group of matched information in TL, (having already made the required substitution into the string), IR4 must be reduced by 4 as is accomplished at stage 34 so that, with its new value IR4 points to the address of the next to last group of matched bytes found in the scan.

This new IR4 is then checked at stage 36 to determine whether it is equal to or greater than "0". If it is equal to or greater than "0", then the above procedure is repeated starting at stage 30. The string is continually compressed through stages 30, 32, 34, and 36 until finally stage 36 determines that IR4 is less than "0". This situation occurs when all the redundant patterns located during a single scan of the string have been substituted for. When this occurs, the program continues from stage 36 to stage 38.

At stage 38, a determination is made as to whether the compression had been a Type 2 compression mentioned previously. Since this is not a Type 2 compression, the program continues on to stage 40. There, information relating to what has occurred in stages 28 through 36 is placed at the front of the string of information. First, the code taken from the LEXICON array, at stage 28 of the flowchart, is stored at the head of the string followed by the pattern which was replaced so that the code defines the particular 12 byte pattern. Thus, in decompression, when one scans the string and finds the particular code, the information at the head of the string will define the meaning of the code information. Following the addition of the code and pattern information to the head of the string, it is obvious that JII has now been increased by an amount equal to R plus 1. Accordingly, JII is increased by R plus 1. It should be noted that the pattern which has been replaced is stored in the machine at location CORD1.

Before continuing, it is important now to discuss an element of the invention which has not yet been discussed and which is germane to the Fast-Mode procedure. There is in storage a table known as the PCORDS Table in which are maintained a maximum of 200 patterns which are considered to be the most repetitive patterns in strings of information processed by SANPAKC. In certain instances, where one knows the basic contents of the strings of information being fed to SANPAKC, one can input a table of PCORDS (permanent cords or matched groups), which contains the repetitious patterns. Where one is dealing with unknown alphanumeric information or information whose content is now known, for example, information that is purely numeric and should have been transmitted through SNUPAK prior to entering SANPAKC), one must, in order to operate in the Fast-Mode, set up a PCORDS Table which will be continuously changing to optimize the Fast-Mode by selecting the PCORDS with the best savings ratio. By savings ratio, it is meant the original JII minus the new JII after compression divided by the old JII, or the number of bytes saved divided by the old string length. Obviously, it is desired to utilize those PCORDS which provide the best savings ratio and, if the best 200 PCORDS are utilized, it may not be necessary to go through the entire Slow-Mode of operation as was previously described. It is expected that by

utilizing the best 200 PCORDS one would be able to reach the optimum compression while saving an enormous amount of search time. Thus, it is important to obtain a PCORDS Table which represents the PCORDS having the best savings ratio. Obviously, the PCORDS Table will only record those patterns which have, in fact, affected a saving. A pattern which does not get past stage 22 (in the flowchart) will not be recorded in the PCORDS Table. Although the PCORDS Table can hold up to 200 different PCORDS, it may be that the best five or ten PCORDS will give such a substantial compression that it would be unnecessary to utilize any further PCORDS. Machine time can thus be substantially reduced since only those five or ten PCORDS would be searched for in the input string.

The PCORDS Table always has at least one PCORD therein and, it is expected in normal operation that the PCORDS Table will be initialized with six PCORDS of the following types: two PCORDS of 12 byte lengths (one containing all zeros and the other containing all blanks); two PCORDS of eight byte lengths (as described above) and two PCORDS of six byte lengths (as described above). By experience, it is known that in most strings of information there are groups of blanks and zeros which occur with regularity and, therefore, it is highly likely that these particular PCORDS will affect substantial savings in any string of information which might be fed to SANPAKC. The PCORDS Table includes 201 20-byte segments of storage space. The first 20-byte segment is control information which gives the status of the entire table. The next 20-byte segment is the first PCORD recorded in the PCORDS Table. This second 20-byte segment, like all succeeding 20-byte segments which are stored in the PCORDS Table, has its first byte signifying the number of bytes in the pattern to be stored. The succeeding bytes after the first byte are the stored pattern or PCORD followed by binary zeroes up to the 16th byte. From the 17th byte through the 20th byte is recorded the savings ratio associated with that particular PCORD. Thus, it can be seen that by scanning the first byte in each 20-byte segment one can determine the length of the PCORD in the 20-byte segment. By scanning the 17th through 20th byte in each 20-byte segment one can determine the savings ratio relating to the particular PCORD. The first 20-byte segment is the control information which gives the status of each PCORD in the PCORDS Table. In this first 20-byte segment, the first 12 bytes are used to indicate the number of patterns of each length (length one up to length 12) stored in the PCORDS Table. The first byte contains the number of length 1 patterns, the second byte contains the number of length 2 patterns, and so on up to the 12th byte. It should be noted that a byte can hold a value up to 256 and thus the number of entries in the PCORD Table for each length can be fully recorded in the first 12 bytes of the control segment, even if all the patterns are the same length. The next four bytes, namely bytes 13 through 16 of the control segment contain the address of the PCORD having the lowest savings ratio. Where the Table has not been filled, this address would be the address of the last 20-byte segment in the PCORD Table, as yet unfilled. However, this information is extremely important when the PCORDS Table is filled as it is desirable to replace the PCORD having the lowest savings ratio with a pattern having a better savings ratio. Since this information is recorded in the control segment, it is possible to replace the lowest savings ratio PCORD with the pattern found to have a higher savings ratio. Of course, in order to compare the lowest PCORDS savings ratio, it is necessary to know what that savings ratio is. This information is recorded in bytes 17 through 20 of the control segment.

In the PCORDS Table, the PCORDS with the longest length are placed at the top and the smallest length PCORDS at the bottom in sequential order. When a pattern is to be substituted in place of the lowest savings ratio PCORD already in the Table, provision is made for shifting the PCORDS so that this sequential arrangement is maintained at all times. The reason for this arrangement is that it is extremely desirable to com-

press starting with the longest length PCORDS and working downward to the shortest PCORDS since the highest savings are achieved with longer length PCORDS. Although, it has been found desirable to always search starting with the PCORD of the longest length working towards the PCORDS of a shorter length, it is, of course, possible to reverse the procedure without affecting the operation of the compression techniques, although a different amount of compression will, in all probability occur. It is expected that by operating from the PCORDS of the greater length and working towards those of a shorter length that optimal compression can be achieved.

It should be noted that providing the savings ration adjacent to each PCORD it is possible to select, for example, the 20 PCORDS having the best savings ratio and then scanning these selected PCORDS starting with the PCORDS having the greatest length. It is desirable to scan PCORDS of a common length so that it is not necessary to place at the beginning of the string, after compression with a particular PCORD, the length of the PCORD but that such information can be added after all PCORDS of the same length have been utilized in compressing the string. It will be understood that although SANPAKC has been set up to be its own lexicographer, (i.e., develop its own code depending upon the particular string of information supplied thereto) if there is a known input such as strictly alphanumeric information requiring only 50 different bytes, the remaining 206 bytes representable in an eight bit code are known to be available for use as coded information and, therefore, the PCORDS can be permanently assigned a code number without the necessity of going through stages 12 and 14 of the SANPAKC program.

After completing the program step at stage 40, the program transmits the new JII value and the redundant pattern stored in the location CORD1 to stage 42 wherein the savings ratio for this particular pattern is determined. It will be understood that the absolute savings ratio for this particular pattern in CORD1 is determined by the formula $N(R-1)/(R-2)$ divided by JII. For example, if the original JII was 1000 and the number of redundant matched segments was three and the length of CORD1 was 12, the savings ratio would be 1.9 percent. However, if the pattern in CORD1 was one of the PCORDS already stored in the PCORDS Table, it is necessary to compute the savings ratio for the pattern in the current string and to average this new savings ratio with the old one stored in the PCORDS Table. For example, if the old savings ratio was 3 percent and the savings ratio computed for this string is 1 percent, the savings ratio is determined by adding the old savings ratio to the new savings ratio and dividing by two. It should be noted that less emphasis is being placed on past performance of PCORDS than is placed on the performance on current or rather current strings. In this way the PCORDS Table can reflect very quickly changes in the type of input information so as to provide a better indication of the true savings ratio of the PCORDS being utilized on the particular information being supplied. The pattern in CORD1 with its savings ratio or new savings ratio is now added to the PCORDS Table. When the PCORDS Table is full, the pattern is not added to the Table if its savings ratio is less than the smallest savings ratio already stored for a PCORD in the Table. All this is determined by the particular state of the PCORDS Table and the computations carried out at program stage 42. Of course, if the pattern in CORD1 was not already present in the PCORDS Table and had effected a savings ratio greater than that of the lowest PCORDS savings ratio recorded in the control segment of the PCORDS Table, then it will be added to the PCORDS Table in the correct position.

After stage 42, the program execution continues to stage 44 where a determination is made as to whether there are more than 16 repeats of patterns with this R length. Since this is the first pattern of an R length of 12, the answer must be "no". However, assuming that in fact there have been 16 patterns of a length 12 when the program entered stage 44, then the program execution would be immediately transmitted to stage 46 (shown on Figure 1B) wherein there would have been placed

at the front of the string a one byte code indicating in the first four bits of the byte the number 16 and in the next four bits the length of the pattern, 12. The length of the string would accordingly be increased by one byte. At this point JII would have been adjusted to indicate this additional byte of information at the front of the string.

After completing this operation at stage 46, the program continues executing at stage 48 (shown on Figure 1C) wherein the question is asked whether this is a Type 2 compression. If the answer is "yes", the program next moves to stage 50 where a further question is asked as to whether this is the fast-Mode. If the answer is "no" then the program then moves to stage 90 (shown on Figure 1B) in the Slow-Mode for Type 2 operations which will be discussed in more detail below. If the answer at stage 50 is "yes" then the program next moves to stage 54 for the Fast-Mode series of steps, which will also be discussed in more detail below.

If the answer at stage 48 was "no", as is in this case, then the program moves to stage 56 where the question is asked "are there any codes available?". If the answer is "no", then the program continues to stage 58 wherein the question is asked "is this Fast-Mode?". If the answer is "no", then the program moves to stage 60 which is the start of the Type 2 compression in the Slow-Mode. This will be discussed with respect to Type 2 compression below. If the answer at stage 58 is "yes", then the program continues to stage 62 wherein the question is asked "are there any Type 2 PRCORDS available?". If the answer is "no", then the program moves to stage 64 which is the start of the exit procedures of SANPAKC which will be discussed at the end of the description of SANPAKC. If the answer at stage 62 is "yes", then the program moves to stage 66, where the counters for the PCORDS Table are updated so that the first Type 2 PCORD is the next PCORD to be picked up for scanning, eliminating all of the remaining Type 1 PCORDS. Then, the program moves back to stage 110 (in Figure 1B) to start the Fast-Mode scanning of the Type 2 PCORDS.

If there are any codes available as determined at stage 56, then the program moves to stage 68 wherein a counter CS4 is reset to zero. This is done so that it can count repeats of patterns of a particular R length during a further cycle of scans of the string by SANPAKC. A further count accumulating in CS4 will result in the addition of another composite byte being added at the head of the string at a later stage. These composite bytes which are added at various stages during the compression cycle, mainly whenever 16 different patterns of a particular length have affected a savings or whenever the length of the patterns, which are being scanned for in the compressor, is to be changed. The composite bytes are the most important pieces of information which are used during the decompression cycle to unscramble the compressed string. The importance of these composite bytes will become clear when discussing the alphanumeric decompressor, SANPAKD. After resetting of the counter CS4 to zero, the program moves to stage 70 wherein a determination is made as to whether the composite byte created at stage 46 (in Figure 1B) is in the list of available codes in LEXICON. If the answer to this question is "yes", then this available code in LEXICON has become non-available and must be deleted from LEXICON. Thus, if the answer is "yes", the program moves to stage 72 where this code is deleted from the LEXICON Table. Further, since the code has been eliminated, the counter of the number of available codes, CS5, must be changed to indicate one less available code, and the codes in LEXICON must be shifted one byte to the left to fill in the space created by the absence of this now non-available code. Once this has been accomplished, the program moves to stage 74 where a further determination must be made as to whether any codes are now available. This question must be asked because the removal at stage 72 of the code may have caused the LEXICON Table to be emptied of all available codes. If the answer at stage 74 is "no", then the program returns to stage 58 as was discussed previously.

As was discussed previously, if the answer at stage 74 is "yes", the program continues at stage 76. Additionally, if the answer at stage 70 had been "no", then the program would also have continued at stage 76. At stage 76 the question is asked "is this the Fast-Mode?" If the answer to this question is "yes", then the program continues at stage 110 (in Figure 1B) mentioned previously. If the answer at stage 76 is "no", then the program continues to stage 78 wherein the question is asked "is R to be decremented?". It will be understood that, as in this case, the reason why stage 46 (in Figure 1B) had been operated is that it had received its directions from stage 44 (in Figure 1A) answering "yes", and the answer at stage 78 would be "no". The "no" answer at stage 78 would be the proper one because at stage 44 control was sent to the routine which will add a composite byte to the head of the string because there was 16 repeats of a cord with a particular R length, in this case 12. This means that there may be more patterns of this length which may be found in the string, therefore decrementing of R at this point would not be desirable. If a "no" answer occurs at stage 78, then the next step in the program is to return to stage 20 for a new cycle in the Slow-Mode. If the answer at stage 78 is that R is to be decremented then the program continues at stage 80 and will operate in a manner to be discussed below.

If the answer at stage 44 (in Figure 1A) was "no", then the program would continue at stage 82 wherein the question to be asked is "are there any codes left in LEXICON which are available for substitution?". If the answer is "no", then the program moves to stage 84 (in Figure 1B) where the question is asked "are there any Type 2 codes available?". The manner of this operation will be discussed with respect to the entire Type 2 mode of operation.

If the answer at stage 82 (in Figure 1A) is "yes", then the program moves to stage 86 where the question is asked "is this the Fast-Mode?". If the answer to this question is "yes", then the program moves to stage 89 (in Figure 1B) in the Fast-Mode operations. This will be more fully discussed with respect to a direct discussion of the operation of the Fast-Mode. If the answer at stage 86 is "no", then the program moves to stage 88 wherein the question is asked "is this a type 2 compression?". If the answer to this question is "yes", then the program would move to stage 90 (in Figure 1B) in the Type 2 mode of operation. However, that mode of operation will be discussed in more detail below. If the answer at stage 81 is "no", then the program moves to stage 26 at which point the question is asked "can more comparisons be made with patterns of this R length?"

The operation at stage 26 has been discussed previously in detail. Obviously, if the answer is "yes", the cycle starts again at stage 20. However, if the answer is "no", meaning that all of the possible comparisons have been made in the string of information for this R length, then the program moves to stage at which point the question is asked "was there a Type 1 saving?". If the answer is that there was a Type 1 saving, then the program continues to stage 46 and will proceed through the succeeding stages from stage 46 in the manner previously discussed. If the answer at stage 92 is that there was no Type 1 saving, then the program continues to stage 80 where R (the length of a pattern to be scanned) is decremented by 1 and, therefore, R will be set equal to R-1 and RM will accordingly be set equal to RM-1. After this decrementing operation the program continues to stage 94 (in Figure 1B) at which stage the question is asked "is RM equal to '0'?". **If RM is not equal to zero and the answer is therefore "no", then the program is recycled in the Slow-Mode at stage 20 with the new value of R being equal to one less than its previously cycled value of R. If the answer at stage 94 is "yes", then we must be prepared to enter the Type 2 Slow-Mode of operation and the program continues to stage 84 where the question is asked "are there any Type 2 codes available?".** As was discussed previously, Type 2 codes are available when, in the LEXICON, there are more than 34 repetitions of single bytes in the string of information supplied to SANPAKC (see description

of stages 12 and 14). If the answer at stage 84 is "no", then the program continues to the exit routines of SANPAKC which starts at stage 64 as discussed previously. A complete discussion of the operations following stage 64 will be discussed below.

If the answer at stage 84 is that there are Type 2 codes available, then there is initiated the Type 2 Slow-Mode operation at stage 60. A discussion of what Type 2 compression involves is now needed.

Where a single byte has reoccurred more than 34 times in the input string fed to SANPAKC, there is good chance that, after compression of the string, that the single byte will in fact still appear more than 34 times in the first 256 bytes of the compressed string. If this occurs, Type 2 compression will be operative and an attempt will be made to compress the string further. Type 2 compression operates as follows:

A scan is made of the first 256 bytes, or a lesser grouping thereof, to determine how many occurrences there are of the particular byte pattern and the locations of these one byte patterns. If the scan shows less than 34 occurrences of the byte pattern in the first 256 bytes of the string then Type 2 compression will not effect a saving and no Type 2 compression will follow. If, however, there are more than 34 occurrences of the particular byte pattern, then a bitmap is formed. The first byte of the bitmap is the byte of the redundant pattern. The second byte of the bitmap is a number designating the number of bytes in the entire bitmap. After the second byte, a map is formed, which, for each byte of the string, a bit is used designate the presence or absence of the redundant byte starting from the first byte of the string and continuing through two hundred and fifty six bytes in the string. If, in fact, the bitmap extends for the full 256 bytes in the string, then since each byte has been substituted for a single bit, there are 32 bytes in the bitmap. However, starting from the end of the bitmap, if the last byte in the bitmap does not contain more than one bit showing the occurrence of a redundant byte, then it is wasteful to have the bitmap the full 32 bytes in length and the last byte of the bitmap is removed and the bitmap reduced in size accordingly. A check of the last byte or bytes of the bitmap is made to determine whether the length of the bitmap is optimal and, the bitmap is shortened until it has proved to be optimal in length. The bitmap is then, in fact, a map which shows where redundant bytes occur in the string of information. Once this bitmap is made, it is then only necessary to remove the redundant bytes from the string, substantially compressing the string, and then placing the bitmap at the head of the string to act as a pattern to indicate, (a) the redundant pattern; (b) the length of the bitmap; and (c) the locations of the redundant pattern in the succeeding string which is, of course, the bitmap.

It is now useful to discuss the flow of information through SANPAKC for the purposes of Type 2 compression. Starting at stage 60 (in Figure 1B) counters and registers are set up for Type 2 compression. After this set up step at stage 60, control then flows to stage 52 wherein the machine is next asked to obtain the next Type 2 code in LEXICON to search for in the string. Type 2 codes are the codes which previously were stored at the end of LEXICON. Alternatively, the PCORDS Table can be checked to determine whether there are any Type 2 PCORDS to be searched for. This alternative will be discussed with respect to the Fast-Mode operation. In most instances the length of the string to be searched for a single byte pattern will be 256 bytes in length, even though, the actual length of the string may be greater than that. If the string itself is less than 256 bytes in length then, the length of the string to be searched will equal the actual length of the string. This value of the number of bytes of the string to search, is stored in the machine location CS3. After stage 52 operations have been completed, the control of the program continues at stage 96 where the instructions are executed to scan the first CS3 bytes of the string for the code which has been picked up from the LEXICON Table as noted at stage 52. The program then continues to stage 98 wherein the question is asked "was there

a savings utilizing this particular code over the first CS3 bytes of the string?". If there are less than 34 occurrences of the one byte pattern (the selected code) then there is no saving and the answer is "no" at stage 98 and the program continues at stage 90 in the manner discussed previously with respect to stage 90. If the answer at stage 90 is "no", the program continues to stage 100 where again the question is asked "was there a Type 2 saving?". Since the answer must again be "no" at stage 100, the program then moves to stage 64 where it is prepared to end the operation of SANPAKC.

If the answer at stage 90 had been "yes", that there were more Type 2 codes available, then the program returns to stage 52 and the operation continues with a new Type 2 code retrieved from LEXICON or from the PCORDS Table. The program will then continue through stages 96 and 98 as was discussed previously. At stage 98 if the answer has been "yes", then the program would have continued to stage 102 wherein the computer proceeds to build the bitmap (BBM) by scanning the first CS3 bytes of the string and determine where in those CS3 bytes the code appears and recording that information in the bitmap, stored temporarily in location CORD1. Thus, at stage 102 the information of concern is the length of the string being compressed, CS3, the code of the redundant information which will be pulled out of the string to compress the string, and the bitmap or CORD1 which provides a road map of the places in the string where the code is present so that, after the redundant information is removed from the string, CORD1 provides a record of where that information was removed from the string, so that it can be later replaced during decompression. After stage 102, operations have been completed, the program continues to stage 28 (in Figure 1A) wherein the counters are updated. There is no need to select a code from the LEXICON array as this is Type 2 compression and such codes are unnecessary. The process of removing the Type 2 codes from the string is then continued through stages 28, 30, 32, 34 and 36 in the same manner as discussed with respect to Type 1 compression except for the special case, which is not found in Type 1 compression, wherein all of the located redundant bytes in the first CS3 bytes of the string may not be removed due to the fact that in optimizing the length of the bitmap as was discussed earlier, the occurrences of some of the patterns may not be removed even though they are in the first CS3 bytes of the string. Thus, it may be the case in Type 2 compression that the removal of a single byte pattern from the string may effect a savings, but all occurrences of the redundant byte may not be removed from the string which is contrary to the case of Type 1 compressions where all occurrences of the redundant pattern are removed from the string.

After completing the instructions at stage 152 and having recorded the exact position of every occurrence of the code in the string, the program continues execution at stage 154. At stage 154, register IR2 is decremented by 4 and this new value is loaded into register IR4. IR4 points to a location in the TL array where the address of the first spot in string, where the original byte pattern which was substituted for during the execution of SANPAKC is to be replaced, is stored. After completing stage 36, the program continues to stage 38 wherein the question is again asked "is this Type 2 compression?". The answer is "yes" and the program continues on to stage 104 (in Figure 1B) wherein the counters are set up to assemble the LEXICON at the head of the string. The counters are set by setting up R and RM. The meaning of R at this stage is indicative of the number of bytes in the bitmap and RM is again equal to R minus 1. It should be noted that, in Type 2 compression, this action can be performed by simply adding the total number of bytes in the bitmap to the current value contained in R and Rm for purposes which will be discussed hereinafter. After, this has been done, the program then continues to stage 40 (in Figure 1A) wherein the LEX instructions are completed which move the string to the right an amount equal to R+1 bytes, leaving a space at the front of the string for inserting the control information for this Type 2

compression. JII is changed by adding to the original JII and amount equal to R+1. It should be understood that this is necessary as R+1 bytes of control information will be placed at the head of the string. The control information relating to the Type 2 compression (a) one byte for the code of the byte being compressed (removed from the string); (b) one byte indicating the length of the bitmap; and (c) the actual bitmap. Thus JII will have been increased by R+1. The program then continues through stages 40, 42, 44, 86, and 88 in the same manner as was discussed previously with respect to Type 1 compression. At stage 88, when the answer to the question of "is this Type 2?" is answered "yes", the program continues to stage 90 (in Figure 1B) wherein the question is asked "are there any more Type 2 codes?". If the answer at stage 90 is "yes", and the program returns to stage 52 for another cycle of Type 2 compression. If the answer at stage 90 is "no", the program continues to stage 100 wherein the question is asked "was there a Type 2 saving?". If the answer at this stage 100 is "no", then the program continues on to stage 64 which will be discussed below. If the answer at stage 100 is "yes" that there was a Type 2 saving, the program control is passed to stage 46, where additional information is placed at the head of the string, namely a composite byte which in its first four bits gives the number of times Type 2 compression has been effected; and in its next four bits is the number of bytes in the pattern being compressed. In this case the length value would be "1". This will key the machine for recognizing the occurrence Type 2 compression during the decompression cycle. The program continues from stage 46 to stage 48 (in Figure 16) and to stage 50 in a manner discussed previously. If the answer at stage 50 was that this was not a Fast-Mode type compression, then the answer is "no". Then, the program continues again at stage 90.

If the answer at stage 50 was "yes", then the program continues at stage 89 (in FIG. 1B) wherein the question is asked "are there any more PCORDS for this length?". If the answer is "yes", then the program continues again at stage 54. If the answer is "no", the program continues to stage 108, where the question is asked "whether there was a savings?". If the answer at stage 108 is "no", then the program continues to stage 110 where the question is asked "are there any more PCORDS?". If the answer is "yes", the program returns to stage 54 where the next PCORD is retrieved for scanning the string. If the answer is "no", at stage 110, then the system continues at stage 64.

If SANPAKC is operating in the Fast-Mode, then the response at stage 16 (in FIG. 1A) would transfer program control directly to stage 112 (in FIG. 1B) wherein the counters and registers for the Fast-Mode will be set up. At this stage, and after setting up the counters and registers for the Fast-Mode, the program continues to stage 54 to get the next PCORD to search for in the string (from the PCORDS Table). If the program is in Type 2 compression, it would be searching for a Type 2 PCORD or, if in a Type 1 compression, it would of course, be checking the next Type 1 PCORD. This determination is made at the succeeding stage 114 wherein the question is "whether the PCORD to be scanned for is a Type 2 PCORD?" If the answer is "yes", then the program starts again at stage 96 and continues in the manner discussed previously with respect to Type 2 compression. If the answer is "no" at location 114, then the program continues to stage 20 (in FIG. 1A) where the string is searched for the Type 1 PCORD.

All of the Type 1 and Type 2 Fast and Slow-Mode branches discussed previously have eventually terminated at stage 64. At stage 64, steps are taken to end the compression operation on the string of information. Only "housekeeping" functions are completed from stage 64 to the end of SANPAKC. That is, at stage 64 the PCORDS Table is searched to find the PCORD with the lowest savings ratio. It should be noted that if the PCORDS Table is not filled, the lowest savings ratio is zero and the address of the PCORD with the lowest savings ratio is the last location in the PCORDS Table. As was discussed

previously, the PCORDS Table in its first 20 byte segment maintains the information as to the lowest savings ratio of a PCORD which is up for review at location 64. It should be noted that, specifically in the Slow-Mode, every PCORD in the PCORDS Table is updated as to its savings ratio. In the Fast-Mode, this would have been accomplished at stage 42 as discussed previously. However, if a particular PCORD had not been looked for during the Slow-Mode, this means that it was not in the string of information reviewed and, accordingly, the savings ratio associated with the particular PCORD is halved. It should be noted that the savings ratio is not averaged as might be expected, but is halved meaning that the last two strings of information have the greatest effect upon which PCORD remains in the Table with the highest savings ratio. This allows the system to rapidly change over from one type of information input to another. For example, if one were compressing information in English and there immediately followed information in German, where there might be different grouping of letters, the PCORDS Table would be very responsive to this change and after only a few strings of information had been fed through the compressor, the entries in the PCORDS Table would reflect this change to the new language being supplied to SANPAKC.

After operations are completed at stage 64, the program continues to stage 116. At stage 116, four additional bytes of information are placed at the head of the string. First, the length of the compressed string, JII, is recorded in the first three bytes of the four byte addition. JII, of course have been updated to include the last value of JII plus the four bytes of information to be added at this stage. As was stated, three bytes are used to record this value of the new JII with the fourth byte being utilized to provide a count of the number of different compressions which took place in the string to follow and which in fact has been made on the string prior to reaching stage 116. If the input information has not been compressed, then, of course, the fourth byte of the above mentioned four byte segment at the head of the string will be zero. After completion of the steps at stage 116, the remaining steps outlined in FIG. 1B are "housekeeping" machine functions which are completed merely to provide information as to the economics of the compression techniques completed in SANPAKC and to determine where the control of the program is to be continued.

For example, at stage 120, after completion of the instructions at stage 116 there is provided a set of instructions for increasing the amount storage within the machine which is addressable by the program. This is necessary because of a limitation on the number of instructions which can be addressed in one section of machine storage. At stage 122 the determination of the value of the variable MODE, an input command, indicates whether the string has come through SANPAKC or, whether at the input stages 11, the string and been directed not to be compressed such as would occur in a retrieval mode of operation. Accordingly, those strings of information which are not to be compressed are transferred directly from stage 11 to stage 120 and then stage 122 without ever passing through any portion of SANPAKC. At stage 122, if the determination is made that the information had not been compressed, it goes directly to the termination point of SANPAKC. If there had been a compression, and, therefore, the answer at stage 122 is that MODE is not equal to zero, then a stage 124 there is computed a savings ratio of the amount of savings achieved by the compression of the input string by SANPAKC. Thus, the savings ratio is the number of bytes saved divided by the original number of bytes in the input string, and accordingly, the actual savings achieved by SANPAKC can be determined for each string of information supplied.

SANPAKD

SANPAKD is the alphanumeric decompressor which is used for decompressing strings of information which have been compressed by SANPAKC. It is the purpose of SANPAKD to take such compressed information and return it to the form of the original input information.

DETAILED DESCRIPTION

Briefly, in the case of a string that has undergone both Type 1 and Type 2 compressions in SANPAKC, the first three bytes of the compressed string indicate its length in bytes. The fourth byte specifies the number of compressions carried out on the string in SANPAKC. These four bytes are removed and set to appropriate registers to be used for control purposes through the decompression process. The next two bytes in the compressed string relate to a Type 2 compression: one gives the Type 2 byte which was deleted from the string and must now be inserted in the proper location thereon, the other byte gives the length of the bit-map which follows next and will be used for finding the right locations in the string to carry out the insertions. The insertion process is carried out and then any other deleted Type 2 bytes are reinserted in the compressed string in the same manner. Next, decompression information relating to Type 1 compression is examined. As noted earlier, for each Type 1 compression, the string has at its head a Type 1 code byte, a byte designating in four bits the length of the replaced pattern and designating in the other four bits the number of patterns replaced. These two bytes are set to appropriate registers for control purposes, and the R bytes of the replaced pattern which follow at the head of the string are inserted in place of every occurrence in the string of the Type 1 code just mentioned. The process is repeated until all deleted patterns are replaced in the string.

Various housekeeping, control and error checking functions are also carried out. A detailed description of each step of the process, with particular references to the drawings, is given below.

The information flowchart for SANPAKD is shown in FIG. 2. In FIG. 2 at stage 130, the instruction SANPAKD is given which will initiate all the steps which follow as set forth in FIG. 2. The first step in the decompression of the string such as the output of SANPAKC discussed above, is to complete the instructions at stage 132. The instruction OPCORDS at stage 132 is to optimize the PCORDS Table if the input string has not already been compressed. But if, the input string is intended to be compressed, at stage 132 the PCORDS Table in SANPAKC will be optimized. This optimization is carried out by removing all patterns in the PCORDS Table except for a specified number of PCORDS with the highest savings ratio values. The actual number of PCORDS to be retained in the PCORDS Table is an option of the user. For example, if it is only desired to scan the best five PCORDS, then, in fact, only the top five PCORDS in terms of savings ratio will be utilized during the SANPAKC compression with all other PCORDS being deleted from the table. Once this optimization instruction is completed prior to entering SANPAKC, the program control then continue with all of the instructions in SANPAKC as was discussed previously. If the string at stage 132 is intended for decompression, then program execution is continued at stage 134 where all of the registers and counters for SANPAKD are set to receive a new string of information. After setting the registers and counters, the program continues at stage 136 wherein instructions are given to remove the first four bytes from the head of the input string. These four bytes, were, as discussed previously with respect to SANPAKC, comprised of three bytes which designated the length of the string and one byte which initiated the number of compressions which had been completed on the string while passing through the SANPAKC. The first three bytes of these four bytes are removed from the input string and are then stored in counter JII (length of the string). The fourth byte is stored in counter CS8. This last counter CS8 indicates the number of compression which had been completed on the

input string. After completing the instructions at stage 136, the program then moves to stage 138 wherein a determination is made as to whether counter CS8 has a value greater than zero. If CS8 is equal to zero, then the input string has not been compressed and there is therefore no need for sending the string through any further stages of SANPAKD. Accordingly, the program control is then immediately passed to stage 140 which will be discussed at the end of the operation of SANPAKD.

If CS8 is greater than zero, the string was compressed and therefore requires decompression. Thus, control passes to stage 142, where registers BRYY and BRY, which are registers in the computer, are loaded with information as to where the string of input information can be found in the computer. Once this is determined, the next step is taken at stage 144 where the first byte of the input string is examined. The first byte of the string thus processed will have, as was discussed with respect to SANPAKC, a composite byte comprising first (A) four bits which indicate the number of repetitions of a particular pattern length of follow; and (B) the next four bits indicate the length of the first group of repetitive patterns which have to be decompressed. For example, if two patterns of length 12 are at the head of the string, then (A) would be 2 and (B) would be 12. However, it should be noted that it is most likely the length of the pattern at the head of the string would be small as in SANPAKC compression occurs first with the longest patterns and works downward to the shortest patterns, with the last patterns to be the compressed being the Type 2 patterns, one byte in length. If there was any compression of the string, this composite byte would be at the head of the string. The first half of the composite byte, indicating the number of repetitions, is stored in counter CS4 and the length the patten is stored in location RM, remembering that R equals RM+1.

After the preliminary steps at stage 144, the instructions proceed to stage 146 wherein a determination is made as to whether the string should be decompressed for Type 2 or Type 1 information. Thus, if RM equals "0", then the information is in order for Type 2 decompression and the instructions would proceed to stage 148. If RM is greater than "0", then Type 1 decompression is in order and the instructions would proceed to stage 150. At stage 150, the input string must have, as its first byte the code which has been substituted for a particular pattern and, the succeeding R bytes comprising of the redundant pattern to which substitution has been effected. It will be understood that at stage 150 JII is reduced by R plus 1 bytes and the substitution code and the redundant pattern are stored in locations CODE and CORD1 respectively. The string is then moved, to the left, R+1 bytes to close up the space created by the removal of the above information from the head of the string.

After completing the steps at stage 150, the program continues at stage 152 where the instruction FIND is initiated. These instructions scan the string for the single byte in CODE stored during the operation at stage 150. This byte is the substitution code which replaced the occurrences of the redundant pattern during the execution of SANPAKC. The addresses of the locations in the string where this code is found are stored in the array TL. Register IR2 contains the number of bytes past the beginning of the TL array where the address of the last found occurrence of CODE is stored.

After completing the instructions at stage 152 and having recorded the exact position of every occurrence of the code in the string, the program continues execution at stage 154. At stage 154, register IR2 is decremented by 4 and its new value is loaded into register IR4. IR4 points to a location in the TL array where the address of the first spot in the string, where the original byte pattern which was substituted for during the execution of SANPAKC is to be replaced, is stored. After completing this step, the instructions continue at stage 156, where the number recorded in register IR4 is loaded into register IR5. Register IR5 indicates the last string address at which a code was found during the scan defined in the opera-

tions at stage 152. IR5 is now set equal to the value in IR5 minus the address of beginning point of the string of information. Thus, at this point IR5 is equal to the number of bytes from the beginning of the string where the last code found is actually located.

The next step is to determine at stage 158 whether the code we are dealing with is a Type 1 or Type 2 code. If it is a Type 1 code then the program control goes directly to stage 160. If it is a type 2 code, the program control goes to stage 162. This determination is made by merely checking, as at stage 146, as to whether RM is equal to zero or greater than zero. Considering the case with Type 1 compression, the program continues to stage 160 wherein the string is operated on by moving the remainder of the string one byte past the location pointed to by IR5 (the location of the CODE in the string) to the right an amount equal to RM (R minus 1). This leaves an opening of R bytes in the string subsequent to the location pointed to by IR5 one byte of which is the substitution code placed in the string by SANPAKC. Then, the pattern in CORD1 is inserted into the string in the R-byte space between the location pointed to by IR5 and the remainder of the string. The insertion of the pattern in CORD1 into the string erases the code which had replaced the pattern during the compression cycle and the new string will be returned toward its decompressed form. In the process, the actual length of the string has been increased by RM bytes and this amount is added to JII. After this stage is completed, IR4 is decremented by 4 so that it now points to the next lower address in the TL array where the next address in the string to be operated on is stored. This is accomplished at program stage 162. That is, the new IR4 is equal to IR4 minus 4 bytes which is the position of the next CODE address stored in TL. The program then continues to stage 164 where a determination is made whether the new IR4 is equal to or greater than zero. If the value is equal to or greater than zero, then the loop is executed again by returning to the instructions at stage 156 to a gain insert the pattern in CORD1 at particular CODE locations. This looping will continue until IR4 is less than zero. This means that all the codes for this particular pattern have been replaced by the original string pattern and the program will continue to the instructions at stage 166. At stage 166 are the instructions relating to checking the operation of the string decompression and insuring correctness. Thus, at stage 166 counter CS8 is decremented by 1 indicating that the first decompression step has been completed and that there are now left a new CS8 minus 1 decompression steps to be completed before total decompression of the string is achieved. Further, counter CS4 is also decremented by 1 meaning that for this particular length of pattern there are CS4 minus 1 decompression steps to be completed before a new composite byte is located or total decompression of the string has been achieved.

The next step is to determine, at stage 168, if counter CS4 is equal to or less than zero. If counter CS4 is greater than zero, then the program returns to stage 146 and the loop will continue until CS4 counts down to zero indicating that all patterns of this R length have been decompressed. When this occurs, the program continues to stage 170 where a determination is made as to whether counter CS8 is equal to zero. If counter CS8 is not equal to zero, then the program returns to stage 144 to remove a new composite byte which, at this stage, should be the first byte of the string and to continue through the decompression stages. If, in fact, all of the decompression steps have been completed, the counter CS8 will be equal to zero and the program will continue to stage 140. At stage 140, the now decompressed information is then set up for use in the numeric decompression portion of SNUPAK (if numeric decompression is required), the operation of which will be discussed below. This treatment involves breaking down the string of information into substrings in accordance with whether the information is textual, floating point, or integer information. All of this will be more fully discussed with respect to SNUPAK. After the steps at stage 140, the string would leave SANPAKD fully decompressed and ready for use wherever needed.

If the determination is made at stage 146 that RM is equal to zero and, thus, we are dealing with Type 2 compressed information, the next steps are taken at stage 148. At stage 148, the Type 2 control information is decompressed by removing three items of information from the head of the string. The first item of information is the 1 byte code which is to be replaced at selected locations of the string as is designated upon decompressing the bitmap which is to follow. The second item of information which is removed from the string is the byte following the one byte code in the string of information. This byte of information designates the number of bytes in the bitmap which follow this byte in the string. It should be noted that by the removal of this information from the head of the string the length of the string, JII, must be reduced by an amount equal to the length of the bitmap plus 2 bytes. The bitmap is also decompressed at this stage and the addresses of each location where a substitution must be made are stored seriatim starting at location TL in four byte increments. The number of bytes past the beginning of TL, where the last address is stored, is stored in counter CS10. The program then moves to stage 149 where the contents of counter CS10 is loaded into register IR4. The program then continues on to stage 156. Register IR4 is transformed at location 156 in the same manner that was discussed previously. Control is then transmitted to stage 158 wherein, again, determination is made whether RM is equal to zero. Since RM is equal to zero, the program continues to stage 162. At stage 162, there is the utilization of a counter CS11 which is the original number of one byte patterns from the string during the compression cycle as determined by the program when disassembling the bitmap at stage 148. This number minus 1 is loaded into register IR6 and the new number in IR6 is then stored back in counter CS11. A new value stored in register IR5 must be determined. IR5 contains the number of bytes from the head of the string to the location where the code to be inserted into the string and from this number must be subtracted the value recorded in register IR6 in order to determine the actual position in the compressed string where the coded information will be placed. This is required as the address stored originally in the bitmap has been changed by reason of the other compressions which had occurred during the Type 2 compression but have not been restored into the string yet. After completing the steps at stage 162, a correct value in IR5 has been computed which can be utilized at stage 160 to insert the Type 2 code into its correct position in the string. Once this has occurred, then the loop will continue through stage 162 in the same manner as was discussed with respect to Type 1 compression until the bitmap has been completely utilized to insert the Type 2 codes in their correct position in the string. The result of the SANPAKD operation is to produce, at the end, an absolute reproduction of the original input string into SANPAKC. It should be noted that the original string length JII originally recorded can be checked against the now new length JII at stage 140 and determine whether the length of the decompressed string corresponds to the length of the original input string. Further, there is a check as to whether the number of compressions equal the number of decompressions which were effected by SANPAKD. These cross checks insure that there is no error in compressing and decompressing the input strings. This completes the operation of the alphanumeric compressor and decompressor in COPAK.

NUPAKC

INTRODUCTION

NUPAKC is the numeric compressor. That is, NUPAKC is designed specifically for compressing numeric information. The machine is normally instructed that certain strings of information are basically in numeric form, and, such information will be transmitted to NUPAKC for compression. In FIG. 3, there is shown a flow diagram of the steps that take place in NUPAKC to compress the numeric information. In FIG. 3, there is a start up procedure which instructs the machine to

proceed to stage 182, where the input strings of numeric information are converted into integer number organized in four-byte words in a manner which will be more fully described in FIG. 4. This conversion into integers is the first compression step in that it removes the floating point exponent and allows the numerical information to be treated as an integer so as to conserve storage facilities and effect more efficient utilization in the remaining compression steps.

After conversion into integers, the program continues to a differencing stage. At this stage, successive integer words in a substring are differenced seriatim so as to substantially reduce the magnitudes of all the integers following the first integer in an optimal manner. The procedures at stage 184 are described in FIG. 6 and will only be accomplished if, in fact, such a differencing procedure will effect a saving and the number of differencing cycles will be limited to that number which reaches the optimal condensing of the input substring.

After completing the procedural steps at stage 184, the program continues to stage 186 described in FIG. 5 in which identical sequences are removed and condensed information replaces the sequential information and a map of the position of such information is placed at the head of the substring so as to indicate, for decompression purposes, where said condensed information can be found in the substring. After completion of the steps at program stage 186, the program continues to stage 188 where all of the substring integers are packed into eight byte double words in a optimal fashion, i.e., the maximum number of integers are placed in each double word so as to again condense the information. It has been found with NUPAKC, that it is possible, especially in dealing with highly repetitive information such as is found in graphical data, etc., that compression up to 99.99 percent is possible. However, more normally, compression of numerical data is in the range of 80 to 95 percent.

As indicated by the Table of FIG. 9, the string of these double words may then be directed to SANPAKC for further compression.

It should be noted that the longer the substring, the more likely are repetitive sequences to occur and more efficiently are the integers packed into double words. It has been found that when one substring, which could be compressed to save 88 percent, is included in a substring ten times as long it would give a savings of 95 percent. Thus, long substrings should in fact give rise to higher savings.

DETAILED DESCRIPTION

FIG. 4 is a structural flow diagram of the operation at stage 182 discussed with respect to FIG. 3. It should be understood that all substrings, which might be utilized in the COPAK system, have certain identifying words associated with them. (the substring command). One of the first words has been called SOS. If SOS is a number less than zero then the substring is intended to be compressed by SANPAKC only. If SOS is equal to zero then the substring is to be compressed by NUPAKC. It will further be understood that when substring information is read into the computer, this type of identifying material is controlled by the user through the input commands because he knows what the information type is (either numeric or alphanumeric) and, therefore, capable of numeric compression or alphanumeric compression. However, it should be noted that NUPAKC is not purely limited to numeric information and, in fact, alphanumeric information could be compressed by NUPAKC which could, assuming that all the information being entered into the machine is in fact in numeric form. However, for practical purposes, NUPAKC is intended strictly for numeric information. In addition to the SOS substring command, there is a second command called LSX. LSX is a substring command which determines the type of numeric compression which will be used. NUPAKC may use no truncation, truncation by the bin procedure using the value of LSX, or truncation by the logical right shift method. These operations are discussed below.

In the procedure at stage 182 (in FIG. 4) the first step is a determination at stage 190 whether LSX is equal to, less than, or greater than zero. At stage 190, register IR1 contains a value of the number of bytes past location LSX in core memory of the machine where the desired LSX value for the particular substring is stored. If LSX is equal to zero, then the program would continue with no truncation at stage 192. If LSX is less than zero, then the program continues at stage 194 to begin execution of the logical right shift method. If LSX is greater than zero, then the program continues at stage 196 wherein truncation by the bin procedure using the value of LSX is started. LSX is an indication of the degree of reliability which the user desires the information to be passed through NUPAKC. Thus, if one knows that the input data is correct to within 1 percent, then LSX would equal, i.e., 0.01. If the user states that LSX is less than zero (usually set to -1) this means that the logical right shift method will be used. In the logical right shift method there will only be a small variation in the seventh significant figure in the input data upon decompression. Thus, normally, one who wishes to use the logical right shift method would be interested in extremely accurate data with little or no loss of significant information during compression.

Register IR1 will be used throughout this description and it will have the following meaning. IR1 is associated with the address of information in various arrays which are used by NUPAKC for each particular substring. The first location in each array (such as SOS, LSX, BWX, YM, etc.) contains the compression commands for the first substring. The second location in each array is the information associated with the second substring, and so on for each substring in the string to be compressed. IR1 contains a count of the number of bytes past the beginning location of the array where the substring information is stored. Thus, for substring 1, IR1 will have a value of zero, indicating that the information is stored at the beginning of each array. For substring 2, IR1 contains the value of 4, meaning that the information is four-bytes past the beginning of the array. For notation purposes, the symbols such as SOS(IR1), LSX(IR1), etc., will indicate the above mentioned meaning. That is, SOS(IR1) means to use the beginning address of the SOS array plus the number of bytes past the beginning of the array (the value of IR1) to address the proper location of the substring information.

When LSX equals zero, as was previously stated the program continues to stage 192 wherein the floating point number 0.0 is stored in location BWX(IR1). The number 0.0 for LSX indicates that the string is not to be truncated. This information will be added to the head of the string (composed of all the substrings) after completion of the passage of all the substrings through NUPAKC. Thus, after completion of the compression, this information will be placed at the head of the string to indicate that, no truncation was completed on this particular substring so the proper decompression procedure can be affected.

After storing 0.0 in BWX, the program continues to stage 198 wherein instructions are provided to have the substring searched to find the minimum and maximum values in the substring. The minimum value is contained in register IR4 and the maximum value of the substring is contained in register IR5. After this is completed, the program continues to stage 200 wherein the median value of the substring is determined. This is YM(IR1) (the median value for the substring) is computed as the sum of the minimum and maximum values stored in IR4 and IR5 divided by 2. Then IR5 is then reset to equal the absolute value of the median value for the substring (i.e. IR5 will always be a positive value at this point).

After completing this stage, the program continues to stage 202 wherein a determination is made as to whether IR5 is greater than 67,108,864. This would occur where the input information was not in fact, numeric information or there had been some mistake in entering the input string. The number 67,108,864 is equal to 2 to the 27th power. If this were to occur, there obviously was an error in the kind of input informa-

tion entered and, in fact, the input information should not have been supplied to NUPAKC. Since the GR5 is indicative of the medium value and would indicate that there are some numbers above and some numbers below that value, any numbers that exceed the 27 power are too large for the numeric compressor to operate on and, accordingly, they should be bypassed through the numeric compressor. Thus, if IR5 minus 227 is greater than zero then the program continues to stage 204 wherein the number of times the differencing procedure has been executed (in this case 0) is stored in SOS(IR1). Then, SOS is loaded negatively so as to make SOS less than zero. As was discussed previously, when SOS is less than zero, the program is to use only SANPAKC for compression. After completing this storage, the program continues to stage 206 wherein a printout is made to tell the user that the string could not be compressed by NUPAKC.

At stage 208, immediately succeeding stage 206, the operations of NUPAKC on the input string have been completed and program control is transferred to the end of NUPAKC at location 210 as shown in FIG. 3.

If the answer at stage 202 is a negative number, then program continues at stage 212. At stage 212, the number of bytes in the substring is loaded into register IR3 from storage location CS2 and the address of the first byte of the substring is loaded into register BRY Y from counter CS15. After this step, the program continues at stage 214 to subtract the median value YM(IR1), determined at stage 200, from each word in the substring taking the substring word from storage to complete said subtraction step. Each word in the substring is stored in four byte intervals in a storage location addressed by register BRY Y and, at stage 214, the first word in the substring has YM subtracted therefrom. Register IR3 loaded with the number of bytes in the substring, has four bytes subtracted therefrom. Thus, IR3 equals IR3 minus 4. Register BRY Y is incremented by 4 to address the next word in the substring. Next, the program continues to stage 216 where determination is made as to whether IR3 minus 4 or the new IR3 is still greater than zero. If it is greater than zero, then the program returns to stage 214 and processes the next word in the substring recorded at BRY Y plus 4 and, further decrements IR3 by another four bytes. This value of the new IR3 is then checked at stage 216 until such time as the new IR3 is equal to zero. When this happens, the program continues to the terminal stage 208.

If LSX had been greater than zero, the program would have continued at stage 196 to execute the bin procedure truncation. At this stage, a value LSX(IR1) divided by two is computed. The LSX(IR1)/2.0 value is stored at location BWX(IR1).

After completing the storage step, the program continues at stage 218 where the substring is scanned to find a minimum and a maximum value of the floating point numbers in the substring. This value is contained in registers IR4 and IR5 respectively. At stage 220, the value of the minimum number of the substring, in IR4, is subtracted from IR5 and divided by the number stored in BWX(IR1), namely, (LSX/2) to obtain a value which is stored at location DUM1. Thus, it will be seen that as LSX approaches zero, DUM1 becomes larger and larger. If DUM1 becomes greater than 227, then the program returns to stage 194 to execute the logical right shift method. If, at stage 222, it is determined that DUM1 is less than 227, then the address of the first byte in the string and the number of bytes in the substring, recorded in counters CS15 and CS2 respectively, are loaded into registers BRY Y and IR3 so as to be ready for use. After loading the registers with the values from CS2 and CS15, the program continues to stage 226.

The value of the first word in the substring is replaced in its storage location by a value computed by subtracting from the original value at that location, the minimum number in the substring and dividing by the value (LSX/2) stored in BWX. After completion of this step, this word is then further operated on at stage 228 by truncation. The truncation removes all digits to the right of the decimal place in the word

and leaving only the digits to the left of the decimal place. This is known as truncation without rounding as there is no significance placed on the size of the number to the right of the decimal place and it does not affect the integer which remains.

The number or integer thus formed is now stored in the same location from which it was taken and, next, the program continues to stage 230 wherein register BRY Y is incremented to address the next number in the substring and IR3 is decreased by four bytes, the amount one has moved to find the next word in memory. IR3 is equal to four times the number of words remaining to be processed in the substring and by decreasing IR3 by four bytes for each word processed in memory in the substring, it will be understood that when the entire string has been completed, IR3 will be equal to zero.

At the next stage, a determination is made as to whether in fact IR3 has reached zero. At this stage 232 if IR3 is still greater than zero the program continues back to stage 226 and a new truncation is performed by utilizing the value of the second word in the substring and subtracting therefrom the minimum value and dividing by (LSX/2). This new number is then truncated at stage 228 by dropping all digits to the right of the decimal point and storing the new integer in the place for the second word in the substring and continuing to stage 230 to look for the third word in the string while decreasing the new IR3 by another four bytes. If, at stage 232, IR3 is still greater than zero, the loop continues again until IR3 reaches zero. If IR3 is less than or equal to zero, the program continues to the terminal stage 208.

If LSX had been less than zero, at stage 190, the program would continue to stage 194, where the floating point number "-1.0" would be stored at location BWX(IR1). After completion of this storage stage, the program continues at stage 234. At stage 234, the program takes the first number from the substring and logically adds the last four bits of this number to itself which is affective to round the number before the next step is taken of shifting the resultant sum logically to the right five bits, removing the last five bits from the number. This is a truncation with rounding and although one has lost the last five bits, the bits removed are the least significant bits in the floating point number.

The program then continues at stage 236 wherein the truncated, rounded number is returned to its storage location and BRY Y the address in storage is incremented by four bytes to address the next word in the string and IR3 is decreased by four bytes. If IR3 is still greater than zero, at the next stage of the program 238, the loop continues by returning the program back to stage 234 for truncation with rounding of the next word in the string. This continues until TR3 is equal to or less than zero. When this occurs, the program moves on to stage 240.

It should be noted that the resultant words in the substring are now all integers and not floating point numbers as, by shifting to the right five bits, each number in the substring is less than 227 and the number including its exponent at the front thereof can be considered, for all practical purposes as an integer. After this is completed, when the program continues at stage 240 wherein the last word in the substring is removed from the substring and later stored in location CHECK(IR1) for purposes of later utilizing said word as a check on the accuracy of the compression and decompression of the substring. The number so retrieved at stage 240 is first shifted logically left five bits before storing it in CHECK (IR1) at stage 242. This is so the number will be in the exact form in which it should be found after decompression of the string at the end of NUPAKD. These operations occur at stage 242. After completion of the program steps at stage 242, there is stored in memory the address of the beginning of the first word in the string, at location CS15, and the number of bytes in the string, at location CS2. This latter number is loaded into register IR3, which is the number of bytes past the beginning of the substring where the last word in the substring is stored. The address of the beginning of the substring is loaded into register BRY Y. After completion of this step, the program con-

tinues to stage 198 and the information is treated as though there had been no truncation, in the manner discussed previously with respect to integer numbers, including finding a median and subtracting the median from all numbers in the substring. After completion of stages 198, 200, 202, 212, 214 and 216 the converted substring reaches the terminal stage 208 for Step *a* of NUPAKC.

Step *b* of NUPAKC consists of identifying sequences and counting of significant bits so as to achieve condensation of information.

Step *b* of NUPAKC is best shown in FIG. 5. For purposes of definition the following are true:

IR4 is equal to the number of consecutive equal integers;

IR1 is the number of times a particular consecutive number is repeated in the substring;

IR3 is the number of bytes in the particular substring which is being compressed;

BRYY is the address in storage where the particular substring is stored.

In view of the detailed description heretofore given, it is now possible to describe the steps of the program in accordance with groups of steps and the functions which are achieved by the program steps without necessity of describing each individual step within a particular sequence of steps. The actual program listing at the end of the written description will also aid in the understanding.

In program Step *b*, in FIG. 5, it is first desirable to examine the string and search the string to find consecutive numbers which are repeated along the length of the string. After completing the scan, as accomplished at the stages identified collectively by the numeral 244, (i.e., to find repeated numbers, identifying the number of repeats in a particular sequence, and providing the address of each of these repeated groups in the string), all of this information is then stored in the computer.

After having completed the scan to determine the number of repeats of particular numbers, the address of the repeats and the particular number being repeated, the program continues to stage 246 where the determination is made as to whether the number of bytes which can be saved, based on the results of the scan during stage 244, is greater than the number of bytes needed for control information for replacing the sequences of consecutive numbers. If the answer to this question is "yes," i.e., that the number of bytes saved is greater than zero, then the program continues to stage 248 wherein the consecutive identical numbers in the string have substituted therefore two four byte words (a double word) in which the first four bytes contain a number corresponding to the number of repeats of the particular information and the second four byte word indicates the number which is being replaced. The address in the substring where this sequence begins is stored in location DUM1 (IR4). After completing the operation at stage 248, the program continues to the steps shown at stage 250 wherein the substring is closed up to erase the sequence information now represented in the substring by the double word described with respect to the steps at stage 248, providing a new substring where the consecutive identical numbers have been replaced by a double word indicating the number of times a particular number is repeated at a given address in the substring. This operation as described with respect to stages 244, 246, 248 and 250 is continued to find further consecutive repeated numbers (if they exist) in the substring. When more than ten such sequences are found, the program stops with respect to this particular means of compressing the string. The number ten is merely an arbitrary number selected to indicate that a multitude of consecutive number are found in the string. It is most probable that the overall saving is not going to be substantially increased by removing additional repeated numbers found in the substring and, therefore, there is no need to waste additional machine time searching the substring. The number of sequences found in the substring is recorded in register IR6. The actual length of the string (in bytes) is recorded in register IR7. When the

string has been completely scanned, and there is no more saving to be affected by executing the stages 244, 246, 248 and 250, the program continues to stage 252 wherein the stored address (in array DUM1) of the replaced consecutive groupings are placed at the head of the condensed substring.

If at stage 246, it was determined that for a particular repeated sequence found during the program steps at stage 244, that no saving would be affected by substitution for the repeated numbers, then the program continues to stage 254 wherein a determination is made as to whether the program should continue back to stage 244 to look for additional repeated numbers or whether to end this searching procedure and continue on to stage 252. In effect, stage 254 is substantially similar to the operation at stage 250 discussed previously.

After condensing the substring and placing the address information at the head thereof, the program continues to a stage 256 wherein the string is now prepared for the packing step which follows as described in FIG. 6. It should be remembered that no number in the string is greater than 227. Thus, in any four byte word, there must, necessarily, be five bits which are not used. Thus, the five leftmost bits in each four byte word are, of necessity, not used by any number in the substring. For purposes of packing, it will be necessary to determine, for each four byte word, the number of bits required to represent the number. In this regard, the string is prepared by moving the number in each four byte word to the left five bits leaving the rightmost five bits in each four byte word empty. Then, each number is scanned to determine the number of bits required to represent the number and this information is placed in the rightmost five bits. It will be understood that since the number of bits necessary to represent the number cannot be more than 27 (since the number cannot exceed 227) the number of the significant bits will be less than 27 or a number which can be designated within five bits of digital information. Accordingly, the substring, as it reaches stage 256, will be in a form wherein the number in the four byte word is recorded in the first 27 bits and the next five bits will provide information relating to the number of significant bits in those 27 bits.

FIG. 6 is a complete showing of the flow diagram for the program set forth schematically in FIG. 3. FIG. 6 indicates that there are, in fact, four steps required in NUPAKC. The steps are as follows:

- a. Truncation;
- b. Differencing;
- c. Sequencing;
- d. Packing.

In view of the detailed description given in respect to SANPAKC, it will be obvious to one skilled in the art after a discussion of the function of NUPAKC as to the manner in which NUPAKC operates from this functional description. Accordingly, although the flow diagram is a complete step by step analysis of the operation of NUPAKC, the description of the programming steps will be only in functional form without reference to specific counters, storage, and processing elements which will be accomplished in the computer by reason of the programming steps.

The step of truncation is effectively the step described in FIG. 4 with respect to stage 182. After completion of truncation which has been generally designated by the numeral 258, the substring proceeds to be operated on by the program at stage 184 where the differencing operation is completed. The differencing technique is basically an attempt to reduce the number of significant bits in the numbers being compressed so as to better enable the packing operation to be more efficient. Thus, the lower the number of significant figures in a given number, the better packing and more efficient packing is possible. Differencing is a means of achieving lower numbers without losing any information in the string.

The differencing operation is affected as follows:

- a. First, the substring is added in an absolute manner to determine the absolute sum of all of the numbers in the substring, regardless of the sign of any individual number.

- b. Each number in the substring is thus subtracted therefrom its next preceding number, seriatum, in a manner whereby, for example, if the first five numbers of the substring are the numbers (*a*, *b*, *c*, *d*, and *e*), after differencing, the new substring will have the numbers (*a*, (*b*-*a*), (*c*-*a*), (*d*-*c*), and (*e*-*d*).
- c. Then the new differenced substring is added in the same manner as in step *a*, taking the absolute value of the resultant numbers in the differenced substring to produce a second sum. If the second sum is greater than the first sum, (that is the absolute sum of the numbers achieved through the differencing operation is greater than the actual sum of the original numbers) then no improvement can be achieved by differencing and, accordingly, the original substring will be passed out of the differencing stage of the program without any differencing operation being completed thereon. If the actual sum of the differenced substring in accordance with step *b* is less than the absolute sum determined in step *a* then there has been some betterment by the differencing technique and a determination will then be made as to whether the substring can be even further improved by a second differencing step.
- d. A second differencing step similar to step *b* is then effected on the resultant substring of step (*b*) to achieve a new string which will be (*a*, [(*b*-*a*)-*a*][(c-b)-(b-a)], [(d-c)-(c-b)], [(e-d)-(d-c)]. The sum of the numbers in this new substring is then determined and if this absolute sum is greater than the absolute sum of the substring in step *b*, then the substring in step *b* is continued into the next step in the program. If the sum in step *d* is less than the sum in step *b* then the differencing technique is continued until in the last differencing step, the absolute sum of the new substring is greater than the previous differencing step. The step which produces a substring having the lowest absolute sum of the numbers therein is the substring which will be processed through the remainder of the program in NUPAKC. A record is kept of the number of differencing steps achieved in the program stage 184 and this number is recorded in the variable that maintains the status of the substring (SOS), which will later be placed at the head of the substring as information regarding the manner in which the substring can be decompressed.

After completion of the differencing technique at stage 184, the program continues the sequencing operation described in FIG. 5 with respect to step *b* at stage 186. After completing the sequencing operation, one has a string of information in the form of numbers, each in four byte words, with the last five bits of each four byte word giving the number of significant bits in the preceding 27 bits. Additionally, there is an address at the head of the string giving a the number of sequencing operations which have been performed on the string and *b* the addresses of the information which has been sequenced along the string. The purpose of the packing stage 188 is to take the string of information and compress it into its optimal form by the use of a packing technique which will be described as follows:

- a. The information is basically placed into sequential double words.
- b. In each double word, the first eight bits set forth the numbers which will follow in the next fifty-six bits of information. For example, if a string of information includes numbers whose largest number requires only five bits of significant information, then it is possible to place eleven numbers in the 56 bits following the eight bits control information at the head of the double word. That is, after the control information indicates that there are eleven numbers to follow, each one of the numbers in the succeeding string will be placed in five bit groupings within the double word, leaving, at the end, one bit of useless information at the end of the double word. It will thus be understood that considerable compression would have been achieved as 11 numbers would normally have taken

up 44 bytes, whereas, by this technique, it has been possible to compress this into eight bytes. From this limited point of view there had been a saving of 36 bytes.

- c. The first double word in the substring is different from all of the other double words in that it has, in its first eight bits, the number of words which are packed into the last 48 bits in the first double word. The second eight bits in the first double word provides the number of sequences which were found in the substring at stage 186. This leaves only 48 bits in the first double word. It will be understood that if there have been sequencing operations on the substring, after the information relating to the number of sequencing operations, there is at the head of the substring, the addresses where each one of these sequencing operations took place. It is thus possible to determine where the addresses for the sequencing operation begin (after the second eight bits in the first double word) and where they end (after the number of sequence address set forth in the second eight bits in the first double word). After all the addresses have been completed, then the remainder of the substring begins.

It should be understood that within each double word only that group of numbers which can be fit into the 56 bits following the control byte will be included within the 56 bits. For example, if the largest number of a group of successive numbers requires seven bits of significant information, then there will be eight numbers within the 56 bits, each in seven bit segments and the control number will be eight. In this manner, maximum packing will be achieved for a particular substring which is being operated on by NUPAKC. All of the information relating to the substrings which are being operated on by NUPAKC have completed their passage through NUPAKC. When this is completed, the information relating to each substring is placed at the head of the string sequentially and additional information is placed at the head of the string relating to the number of bytes in the now condensed string, and the number of bytes in the string prior to entering COPAK which operates as a check for the operation after decompression of this string.

NUPAKD

The input information to the decompressor NUPAKD, best shown in FIG. 8, is of the type wherein the head of the string has certain control information which has been placed in front of the compressed string immediately subsequent to the completion of the numeric compression in NUPAKC. The input control information has at the head thereof four bytes which are designated as JIR, the number of bytes in the original segment. After JIR, the next four bytes are designated PARM. The first three bytes of PARM are the number of bytes in the compressed segment, with the last byte indicating the number of substrings in the string. It should be noted that no string contains more than twenty substring and, therefore, this information can be placed in one byte.

After PARM, comes the first status-of-substring information (SOS). The SOS four bytes of information contains, in the first three bytes, the number of bytes in the compressed substring. The next four bits contain the format code, which indicates the original input format type of X, A, I, E or F for the information. The format code and its meaning with respect to the type of compression in the string is shown in FIG. 9. The last four bits in the SOS four byte word contain the number of differencing procedures which were accomplished on the substring when passing through NUPAKC stage 184. After the SOS four byte word, there comes a four byte word indicated by the term CHECK. CHECK is the last four bytes in the substring which should be reproduced upon decompression. Thus, after decompression, it will be possible to compare CHECK with the last four bytes of the decompressed substring to determine whether there has been an accurate decompression of the substring.

After the CHECK four byte word, the next substring has its four byte words of SOS and CHECK as indicated. If the second substring had passed through the NUPAKC compressor utilizing truncation from floating point to integer form, it would be necessary to add two additional four byte words relating to said truncation. These two four bytes words of information are BWX, the explanation of which has been discussed with respect to FIG. 4 and stage 196 and YM which is discussed with respect to FIG. 4 and stage 220. BWX and YM four byte words are only added if SOS indicates that there is an E or F type format code indicating that truncation by the bin procedure or truncation by the logical right shift method were affected on the input information. Where the format type in SOS is neither E nor F, then there will be no words for BWX or YM. It will be understood that as many SOS and CHECK four byte words are added to the head of the processed string as there are substrings in the string.

All of the above mentioned information placed at the head of the string is removed from the substring and stored for use during the NUPAKD procedure. The first double word which enters NUPAKD contains in its first eight bits the number of words compressed within the last 48 bits of the first double word. The second eight bits of the first double word contains the number of sequencing operations utilized in compressing the substring. In NUPAKD the input first double word is picked up at program stage 262 of FIG. 7A wherein the first double word is loaded into registers IR2 and IR3. The IR2 contains the first four bytes and IR3 contains the second four bytes of the first double word. At program stage 264 immediately following, the position in storage of the substring is incremented by eight bytes to indicate that the first double word is now being decompressed. Additionally, IR7, the register which contains the number of bytes in the condensed substring, is decreased by eight bytes. After this step, the program moves to stage 266 wherein the first eight bits in the double word are extracted from IR2 to provide the number of words in the substring. After extracting the first eight bits, the program continues at stage 268 to compute the number of bits in each word in the remaining portion of the double word. Since the first double word has 48 bits of information, if the number of words in the substring were nine, at location 268, a determination will be made that there are five bits in each segment of the double word which are to be expanded into full words. The program continues next to stage 270 wherein the information in the next double word is shifted to the left eight bits so that the next 56 bits in the double word can be considered. If this is the first double word in the input substring, then, at stage 272, the next eight bits (NOS) are taken from IR2 and that number is stored in RSX. If this is the first double word, then the remaining 48 bits are shifted another eight bits to the left to bring the last 48 bits to the head of the double word for operation thereon. If the number NOS is zero, then the program would skip to program stage 274 and operate in the manner which will be discussed below. However, if NOS is greater than zero indicating that there are some condensed sequences, then the LSX array is used to store the locations in the substring, of the sequences. In each four byte segment is placed the number from IR2. First, however, the number from IR2 is placed into register IR5 at stage 276 at the right hand end of the register so that, when placed into LSX in four byte segments, the number will appear at the correct position, mainly, the right hand end of the four byte segments. All of the above shift and storage into LSX of the information IR2 and IR3 is accomplished at program stage 278. If, by reason of a review of NOS, it appears that all of the sequence address have not been included in the first double word, provision is made through the use of the program stages 282 and 284 to indicate that the second double word must be similarly decompressed. It should be understood that with the second double word, as it enters stage 262, and continues in a manner discussed with respect to the first double word, that only the first eight bits would be looked at for current information, namely the number (NOS) and that at stage 268 the number

NOS would be divided into 56 the number of remaining bits in the second double word (and each succeeding double word). Thus, after all of the sequencing addresses have been stored in LSX, the program continues at stage 274 to extract, bit group by bit group, each word compressed in the remaining 56 bits in each double word. However, in order to save time in forming the string, it is necessary to first store each 56 words (four byte group) in core storage position DUN1. As each 56 words are stored in DUN1, they are transferred, as a group, to a second core storage location YY where they form the partially decompressed string. This is all accomplished in a series of steps herein noted as program step 286 (shown in FIG. 7B). These transfers eliminate the need for continuously shifting all of the partially decompressed substrings to the right as each additional word is added to the substring. By grouping 56 words in DUN1, it is possible to shift this entire amount in one operation to storage location YY in the correct position at the right hand end of the partially decompressed substring. This operation continues until all of the double words have been expanded back into their original form and the complete partially decompressed substring is presented in which all packing has been removed. When this has been completed, the program is at stage 288. At stage 288, the operation for decompressing the condensed sequences is initiated. This programmatic operation is done generally within the steps indicated as stage 290 (in FIG. 7C). If there was no sequencing operation, then, of course, the entire stage of 290 is bypassed and the program would continue at the stage succeeding stage 290 which will be discussed below.

If sequencing has been accomplished, then the register in the LSX is increased by four bytes and at stage 292 and the first address in LSX is computed at stage 294 to determine where in the substring the sequence begins. At that location in the substring, at the first word containing the number of times the sequence was repeated and the second word contains the actual number which was repeated. This is determined at stage 296. Thereafter, the program computes the size of the hole which has to be made in the substring in order to insert the repeated sequences.

This "hole" creation is accomplished at program stage 298. This program also creates this hole in the substring so as to allow the data to be inserted into the substring at the proper location. In the next sequence of steps in the program, the numbers to be inserted are regenerated and inserted into the substring at the proper address, and the new length for the string is computed for storage in counter JII. Finally, at sub-stage 302, the number NOS is reduced by one and, if it is greater than zero this indicates that there are additional sequencing addresses in LSX and the operation continues again at stage 288. If NOS is now zero, that means that all of the sequencing operations have been completed and all the sequencing numbers have been decompressed and the operation is ready to continue at stage 304. At stage 304, the differencing operation is reversed for the purposes of further expanding the substring. At stage 304, the differencing operation is reversed by continuing to stage 306 wherein the first word is added to the second word. The newly formed second word is added to the third word, the newly formed third word is added to the fourth word, etc. down the substring until the end of the substring thus reversing the differencing operation in NUPAKC and decompressing one complete differencing operation. If more than one differencing operation is required, at stage 308, a determination will be made that an additional differencing operation was accomplished on the compressed information and, accordingly, the steps at programs stage 306 will be repeated until all of the differencing steps have been reversed, returning the compressed information to its original form prior to differencing. When this has been completed, NDR will be equal to zero and the program will continue at stage 301. At stage 310, the truncation process is reversed. If there was no truncation, then nothing happens at stage 310. If the right shift truncation was used during compression, the words will be shifted to the left five bits thus reversing the right

truncation. If the bin method was used, then this too is reversed by multiplying BWX times each of the numbers in the substring and adding to each of the numbers in the substring the minimum YM which has been placed in storage. BWX is, of course, also placed in storage from information which was at the head of the substring prior to its application to NUPAKD. After completion of stage 310, the program continues, finally, to stage 312 wherein SOS is reconstructed in accordance with the new string, specifically adding the new JII and, further checking the new string against CHECK(1) and CHECK(2) and any other information which has been stored indicating the original information such as the original length of the string prior to compression and decompression. The information has thus been compressed in NUPAKC and decompressed in NUPAKD and is ready for whatever uses are desired by the user.

CONTROL PROGRAMS

The CONTROL routine can be viewed as a supervisory program that serves as a buffer between the O/S system of the IBM 360 and the SOLID System. It is coded in the "higher language" (ALLOCATE) that has evolved from the open-ended two-part design. The CONTROL routine performs the following functions:

- a. During assembly, CONTROL positions those components of the SOLID system that are compiled in the main system with the SUBMP BMP service macro.
- b. During execution, CONTROL calls the components when they are needed.
- c. Special termination procedures, which are designed to protect the AUXILIARY FILE, are executed in the CONTROL routine before the O/S system terminates the job in the normal way.

By changing the CONTROL routine and the SUBMP service macro, a user can easily alter the SOLID System to perform a specific task like data compression. The SOLID System can be tailored to fit a particular 360 configuration by altering the planned overlays. A step-by-step description of the implementation procedure follows:

Step 1

Code the control routine in ALLOCATE. If some components are not going to be used the SUBMP service-macro must be altered to include the dummy entry points for the omitted components. Also, the name of the component must be deleted from the overlay structure. For example, if the component SSEARCH is not being used SUMMP will contain the two statements:

```
DUMADD      PMARRAYR SEARCH
```

and SEARCH must be deleted from the planned overlay.

Step 2

Determine the amount of core storage that is available.

Step 3

Figure the amount of storage that is needed for the components and the CONTROL routine.

Step 4

Select values for the fourteen variable parameters, then compile the components and store them in the load module library, SOLID.LOAD. If the retrieval package is being used, the size of the memory block (defined by &NTRKS, &TRKL and <HAYY) should be as large as possible. A minimum of 22,000 bytes is needed for the CONTROL routine plus the largest component. Because frequent accesses to the load module library (SOLID.LOAD) are costly, it is suggested that the planned overlay structure should also be considered at this time.

Step 5

Construct the planned overlay structure so that the storage allocated for the programs will be fully and efficiently utilized. Separately compute the 31 components and store them in the load module library (SOLID, LOAD). Assemble the CONTROL routine.

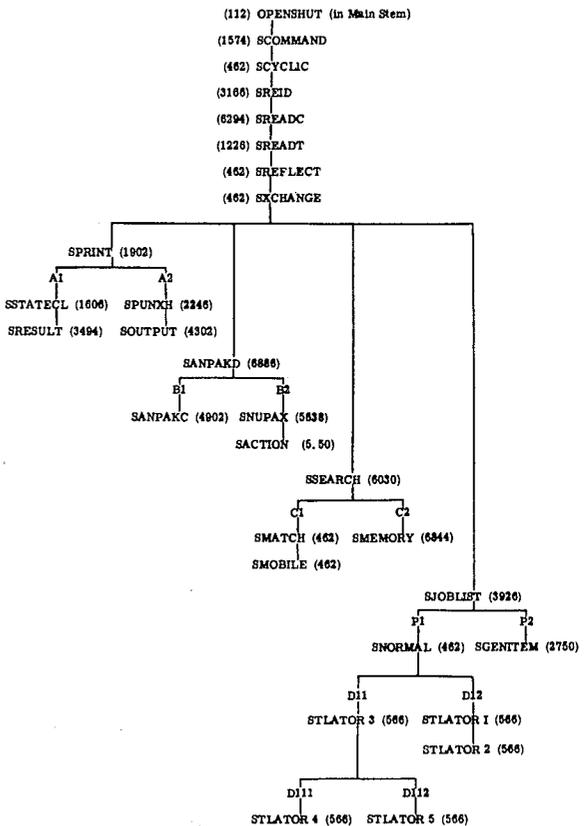
The overlay structures are discussed next, then the CONTROL routines for data-compression, data-transmission and the SOLID System are given.

A. Overlay Structure

Here the planned overlays for the IBM 360/40 (128K) and IBM 360/67 (768K) are given as an example. For details of the overlay technique the reader is directed to the IBM 360 Link Editor Manual.

i. IBM 360/40

The storage (in bytes) needed for each component is given in parentheses. Double buffers were assigned for the four tape DCB's. The single buffer for the disk DCB occupied 3600 bytes. A memory block was defined as ten (=INTRKS) logical records (length=&TRKL=3600). The two principal data arrays, &ARRAY and YY, had lengths of 1500 and 38000 bytes respectively. The storage figures given in parentheses below are approximate. This overlay arrangement requires about 27000 bytes of case-storage, exclusive of the storage needed for data-assay.



ii. IBM 360/67

On the 768K IBM 360/67 the 30 components after OPENSHT were on a single branch of the overlay. Two buffers were used for each tape DCB. About 84,000 bytes of core are needed for this arrangement. All 31 components can also be compiled with the CONTROL routine and positioned by the "SUBMP" type macro instruction.

B. CONTROL for the SOLID System

The program given for the CONTROL System (SOLIDO) requires both the SOLID.MACLIB and SOLID.LOAD libraries. Normally the CONTROL routine would be compiled separately and stored in the load module library, SOLID.LOAD.

Before the CONTROL routine is compiled the variable parameters associated with the RESERVE and SUBMP macro

instructions must be selected. Normally this would be done before the components are separately compiled and stored in the load module library, SOLIDLOAD. All variable parameters have been defined.

In the particular example described in this specification, the CONTROL routine accomplishes the following:

Storage:

An information path is traced in the AUXILIARY FILE with a JOBLIST item stored on a tape with the DCB named TAPEJB and a record number is assigned to the RFILE. The bulk referenced information (stored on the tape with DCB TAPEIND) is compressed and written on a tape with the DCB named TAPEOTC.

Retrieval:

The JOBLIST item stored on the tape with the DCB named TAPEJB is used to trace an information path in the AUXILIARY FILE to the bulk storage address in RFILE. This address, which is the number of a logical record on a tape read with the DCB named TAPEINC, is used to retrieve the compressed referenced information. This referenced information is decompressed and then appears on the device designated by OUTPXT. An unsuccessful search is terminated with an appropriate message.

The translated JOBLIST items can be arranged sequentially on the tape with the DCB named TAPEJB. The bulk referenced information, which is to be stored in compressed form, is located on the tape with DCB named TAPEIND. With OUTPXT=0 the compressed referenced information can be stored on the tape with DCB named TAPEOTC.

When this CONTROL routine is used on a production basis, the following steps are taken:

The compressed referenced information is stored on a bulk storage device like data-cells, disks or tapes. This might involve setting the tape-read (REIDT) and tape-write (WRITE) macro instructions and the first BULK address in the initialization macro MJARRAY, which is executed in SSTATECL.

The variable parameter <HAYY is set to equal to the size of a memory block plus 2000 (for the M and J arrays) plus $2\frac{1}{2}$ times the maximum string length ($LIT = SLENGTH + LLENGTH$).

For example, if strings are to be less than 2,000 bytes and a memory block is to have 100 logical records then: <HAYY=100 (memory block is $100 \times 7294 = 729,400$ bytes); LIT = 5000 (i.e., greater than twice times the maximum string length); <HAYY = $729,400 + 5,000 + 2,000 = 740,000$. The 2,000 is the number of bytes needed to store the first arrays associated with the prime index M and screen J.

The flow diagram for the CONTROL package SOLIDE is shown in FIG. 25. SOLIDE is the extended form of the CONTROL package where no overlays are required. If overlays were required, the stage 1020 shown in FIG. 25 calling for the macro-instruction SUBME would have substituted therefore the macro-instruction SUBMO with the numbers 100,000; 500; 500; 500; and 1500. However, for the purposes of simplicity of the description, the CONTROL package in its extended form will be fully discussed with only the program at the end of the specification being utilized to show the operation with overlays.

In the control package SOLIDE, first, the RESERVE macro instruction is executed at stage 1022. The RESERVE macro-instruction defines the storage areas, the registers, tapes, and initializes the SOLID System. The principal array YY, the override arrays, and the two arrays (JBLIST and JB1) are defined in the macro instruction SUBME at stage 1020. The system parameter &ADDL which is the composite address length is set, for example, at six bytes; the principal array length parameter <HAYY is set at 100,000 bytes, the number of PCORDS in the PCORD TABLE that are to be used in the fast mode of SANPAKC are set to 5 (&TPCORD); and the length of the two JOB-LIST arrays JBLIST and JBWORK is set equal to 1500 bytes (&LJBLIST). After these system parameters are set, control goes to the next stage, 1024, wherein the macro instruction DEVICE is executed.

The DEVICE macro sets up seven device commands which tell the system where to find information that is to be compressed and what to do with information after it has been compressed. These seven device commands are as follows: INPXT (tells what type of device information is coming in on); OUTPXT (tells the system what device the information should be put on after compression or decompression); RSKIPS (a tape command which tells how far to skip out on the tape before beginning); SLENGTH (the minimum number of bytes per segment of information, used as a compressor command); LLENGTH (the number of bytes in the LABEL that is not to be compressed at the beginning of each segment); RNOS (the number of strings or segments that have to be processed by the compressor before a new set of device commands is read); and TPCORD (the number of PCORDS in the PCORD TABLE that are to be used by SANPAKC in the fast mode). It should be noted that if TPCORD as set in the DEVICE command at stage 1024 is not entered, the value of TPCORD falls back to the value &TPCORD set at stage 1022.

After completion of the macro device at stage 1024, control goes to stage 1026 where the macro-instruction STRING is executed. The STRING macro reads the five string commands that define the status of the strings or segments of information. These five commands are MODE (tells whether the string or segment is to be compressed or decompressed or to be used to update the system); POSTOP (tells the system what to do after it has completed the current job, i.e., to get out of the system; to read a new set of device commands; or to read a new set of string commands); LEXCON (indicates whether the PACORD TABLE must be read or not read); LEXMODE (indicates whether the system is operating in the fast mode or the slow mode or simply extending the PCORDS TABLE); LEXPACH (indicates whether the PACORDS TABLE should be punched or not punched after the current string or segment has been compressed or decompressed).

After completion of stage 1026, this program continues to stage 1020 where the macro instruction GETJLIST is accomplished. The GETLIST macro performs all of the instructions relating to the fetching and translation of the descriptor sets to the JOBLIST form. There are nine instructions associated with the GETJLIST macro. These instructions are as follows: JLINPXT (designates the imprint input/output device); JLR-SKIP (a tape command which indicates the number of records that are to be skipped before the first record is read); JL-TRAN (designates the translator that is to be used); JLNORM (designates whether normalization is to be executed or not); KLENGTH (designates the number of bytes per kernel in the JOBLIST item); NJOBS (designates the number of bulk items to be stored for each information path); NTASKS (designates the number of items in the JOBLIST); and the last four items NVALUE, JVALUE, NUMDIAG and GENERATE are special instructions associated with the Monte Carlo generator for generating random JOBLIST items. They are read only when JLINPXT equals 16 which indicate that the Monte Carlo generator is to be used. When random generation of JOB-LIST items is being effected the NVALUE is the value of M, JVALUE is the maximum value of any J, NUMDIAG is the maximum number of diagonals or screens to be generated; and GENERATE is the location of the random number that is to be used by the Monte Carlo generator to generate the JOBLIST item. Monte Carlo or Random generators are used to debug the system or to determine the limits of the system and to determine the economics of its operation.

Control then goes to stage 1030 wherein the macro instruction CALL2 is executed. The CALL2 macro first executes the SSEARCH component, as discussed previously, then the component SRESULT is executed. SRESULT prints intermediate results of the search. In a continuing production system, it may be undesirable to even utilize the SRESULT component and, accordingly, the CALL2 macro instruction may have substituted therefore a macro instruction CALL1 which would call only the SEARCH procedure. After the Call1 or CALL2 instruction control goes to the location ANSWER. Thereafter,

at stage 1032, the instruction DISPENSE would be executed. In DISPENSE the determination is made as to whether control should pass through the compressor, back to stage 1030, back to stage 1028, back to stage 1026, or back to stage 1024. The other option is, of course, to leave the machine because the day's operations have been completed.

At some point, after completion of stage 1032, the program would continue to stage 1034 wherein another CALL1 instruction would be effected to pass control to the SREADC component which reads the substring command, and the bulk information, if it is on cards. Then control goes to stage 1036 which is a decision box. At stage 1036, determination is made as to whether the INPXT command is zero or not zero. If the INPXT command is zero, then control goes to stage 1038 wherein a CALL1 instruction is used to pass control to SREADT component which reads the sub-strings of information from magnetic tape. When INPXT is zero, this means that the bulk information is on tape. If INPXT is not zero, then the control passes directly to stage 1040 wherein RSKIPS is set to zero.

RSKIPS is normally used for the tape read-out and, since INPXT is not zero, this means that the bulk information is not on tape and, therefore, there is no need to have any value of RSKIPS. If the information had been on tape, and had been read out at stage 1038, RSKIPS would have been reset to zero so that, at a later stage, it would be reset to a new value in the macro DISPENSE at 1032.

After completion of stage 1040, control passes stage 1042 which is the COPAK macro discussed previously. After completion of the COPAK macro at stage 1042, control passes to stage 1044 wherein the macro instruction CALL1 is used to call the SOUTPUT component. SOUTPUT macro disposes of the information after compression or decompression in COPAK in accordance with the OUTPXT command set at stage 1024. After completion of the SOUTPUT command, control can pass either back to stages 1032, 1030, 1028, 1026 or 1024 or, alternatively, can pass out of the system. After completion of stage 1044, the last stage of the program SUBME at stage 1020 positions all the components correctly at compilation time. The fourteen system parameters discussed previously are defined at stage 1020.

CONTROL PROGRAM COPAKCD

If the COPAK compressor program were to stand alone without relation to the SOLID System, then a separate control program would be required for COPAK. This has been defined as COPAKCO. The control program flow diagram for COPAKCO is shown in FIG. 26.

The control program for COPAKCO is substantially similar to the control program for SOLIDE except that unnecessary macro-instructions relating strictly to the SOLID System have been eliminated and new or changed macro instructions have been substituted therefore. Macro instructions which are similar to the macro instructions shown in FIG. 25 have been shown in FIG. 26 with prime numerals. Basically the control program in FIG. 26 is substantially similar and operates in substantially the same manner as the control program of FIG. 25.

In FIG. 26, the first stage of the flow diagram for COPAKCO is stage 1046 wherein the macro instructions RESERCO is executed. In this instruction, only three system parameters are set, namely <HAYY, (the length of the principal data array); &TPCORD (the number of PCORDS in the PCORD TABLE used in the fast mode); and &LJBLIST (the length of the job list array). For purposes of example, in FIG. 26 &Lthayy has been set at 20,000 bytes, &TPCORD has been set at 5 and &LJBLIST has been set at 1,500 bytes. After setting these system parameters, the program continues through stages 1024' and 1026' to stage 1048 wherein the macro instruction DISPOSE is executed. DISPOSE performs the same functions as were performed by DISPENSE at stage 1032 except those procedures relating to the search operation have been omitted. After completion of stage 1048, the pro-

gram continues to stages 1034', 1036', 1038', 1040', 1042' and 1044' in the same manner as was discussed with respect to FIG. 25. Then, at stage 1050, the macro command SUBCE is performed which positions all Those components needed for compression or decompression at compilation time. SUBCE is used if OOPAKCO is to be used in the extended form. If the CONTROL routine is to be executed in the overlay form, then instruction SUBCO should be used in place of SUBCE.

CONTROL PROGRAM COPAKAN

If the alphanumeric compressor and decompressor is to be used as a stand alone program, then a separate CONTROL program should be used. This CONTROL program is shown in FIG. 27 and it is named COPAKAN. Similar programatic steps shown in FIG. 25 and 26 have been shown by either ' or '' numerals in FIG. 27 to indicate that there is no difference between these programatic steps and the steps used in FIG. 27.

In FIG. 27, the program continues as in FIG. 26 through stages 1046', 1024'', 1026'', 1048', 1034'', 1036'', 1038'', 1040'', to stage 1052. At stage 1052, the macro-instruction COPAJ is effected. COPAJ is a special macro-instruction which, in effect, is COPAK without the numeric compressor, decompressor comonent SNUPAK. After completion of stage 1052, the program continues to stage 1044''. Stage 1054 includes the instruction SUBCJ which positions all of the components necessary for COPAKAN. It should be noted that the instruction SUBCJ is for use in the extended form. If operation is in the overlay form, then there is substituted for the instruction SUBCJ, the macro instruction SUBCJO. Please note that for both the COPAKCO and COPAKAN and, additionally, for the COPAKNU instructions to be discussed hereinafter, there is only needed three system parameters, namely, <HAYY, &TPCORD, and &LJBLIST.

CONTROL PROGRAM COPAKNU

If the numeric compressor and decompressor SNUPAK is operated as a stand alone program without the alphanumeric compressor SANPAK then a special control program for the macro SNUPAK must be used. This is shown in FIG. 28. This is defined as COPAKNU. Similar programatic steps shown in FIGS. 25, 26, and 27 have been indicated with prime numerals to indicate similar instructions. In COPAKNU shown in FIG. 28, again the program starts at stage 1046'' continues through stage 1024''' to stage 1056. At stage 1056, the macro instruction STRING is effected, but only the first three string commands mode, POSTOP and LEXICON are read as the remaining instructions discussed with respect to FIGS. 27 and 26 relate to alphanumeric compression and, therefore, are not necessary.

After completion of stage 1056, the program continues through stages 1048'', 1034''', 1036''', 1038''', 1040''', to stage 1058 wherein the macro instruction COPAB is performed. COPAB is a variation of COPAK without alphanumeric compression or decompression. After completion of stage 1058, the program continues to stage 1044'''. Stage 1060 provides the macro-instruction SUBCB which provides all of the components of COPAKNU at the time of compilation. Again, SUBCB is the macro instruction in the extended form, if the system is operating in the overlay form, then a special instruction SUBCBO must be substituted for the instruction SUBCB.

It will be appreciated that all of the functions shown in block diagram in the drawings are implemented by digital program. The digital program listing in accordance with this invention will now be given sufficient details to enable those skilled in the art to carry it out. This routine is written in IBM BALL language and the program can be carried out by a number of suitable digital processing systems. As one example of a digital system on which this program has been performed, reference is made to the IBM Computer 360/40. The program is as follows:

ASADD

```

*****ASADD *****
      MACRO
&J      ASADD &ADDRESS,&RD,&RDNO,&RTRK,&RCYLN,&RFMADD
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---ASADD ASSEMBLES &ADDRESS FROM ITS COMPONENTS IN REGISTERS.
*---&ADDRESS IS THE LOCATION OF THE COMPOSITE ADDRESS.
*---&RD IS THE DEVICE.
*---&RDNO IS THE NUMBER OF THE DEVICE &RD.
*---&RTRK IS IS THE TRACK ON THE CYLINDER.
*---&RCYLN IS THE CYLINDER NUMBER.
*---&RFMADD IS THE FAST MEMORY ADDRESS.
*-----
&J      ST      &RD.,SERVICE
        ST      &RFMADD.,SERVICE+4
        ST      6,SERVICE+8
        SLL     &RD.,4
        ALR     &RD.,&RDNO
        SLL     &RD.,16
        MVC     DUN7(1),=C'&RTRK'
        CLC     DUN7(1),=C'&RCYLN'
        BE      AA&SYSNDX
        SLL     &RTRK.,10
AA&SYSNDX ALR     &RD.,&RTRK
        ALR     &RD.,&RCYLN
        SLL     &RD.,8
        STH     &RFMADD.,DUN7+4
        SRL     &RFMADD.,16
        ALR     &RD.,&RFMADD
        ST      &RD.,DUN7
        LA      6,&ADDRESS
        MVC     0(6,6),DUN7
        L       &RD.,SERVICE
        L       &RFMADD.,SERVICE+4
        L       6,SERVICE+8
        MEND
*****ASADD *****

```

*TAPE.

BBM (CONT.)

```

IA&SYSNDX  CLI  0(4),X'01'      *DOES BYTE HAVE 1 OR LESS '1'-BITS?
            BH   KA&SYSNDX      *NO - CONTINUE TO NEXT STAGE.
            BL   JA&SYSNDX      *NO '1'-BITS AT ALL.
            L    3,CS2          *1 '1'-BIT - DECREMENT ADDRESS POINTER.
            S    3,=F'4'
            ST   3,CS2
JA&SYSNDX  BCTR  4,0          *DECREMENT ADDRESS IN SIGNIFICANT BIT ARRAY.
            BCTR 5,0          *DECREMENT NO. BYTES IN BITMAP.
            BZ   MA&SYSNDX      *NO BITMAP LEFT - GET OUT.
            B    IA&SYSNDX      *CHECK NEXT BYTE IN BITMAP.
KA&SYSNDX  SR    1,1          *SEE IF BITMAP CAN BE REDUCED FURTHER.
            IC   1,0(4)        *R1= NO. OF SIG. BITS IN LAST BITMAP BYTE.
            LR   0,1
            BCTR 0,0          *R0= NO. OF PREV. BYTES IN BM TO CHECK.
            LR   2,4          *R2= ADDRESS IN SIG. BIT ARRAY TO CHECK.
LA&SYSNDX  BCTR  2,0
            CLI  0(2),X'00'      *IS NO. OF SIG. BITS = 0?
            BNE  MA&SYSNDX      *NO -- CANNOT REDUCE BITMAP FURTHER.
            CR   2,6            *RUN OUT OF BITMAP YET?
            BE   MA&SYSNDX      *YES - GET OUT.
            BCT  0,LA&SYSNDX    *CHECK NEXT LOWER BITMAP BYTE FOR ZERO.
            SR   5,1
            SR   4,1
            SLA  1,2
            L    3,CS2          *DECREMENT ADDRESS POINTER ARRAY.
            SR   3,1
            ST   3,CS2
            LTR  5,5            *RUN OUT OF BITMAP YET?
            BNE  IA&SYSNDX      *NO - CONTINUE THE SCAN.
MA&SYSNDX  STC   5,CS12+3      *CS12=NO. BYTES IN BITMAP PLUS 1.
            BCTR 5,0
            STC  5,&CORD1      *GR3=NO. BYTES IN BITMAP MINUS 1.
            LM   0,15,SERVICE
            MEND
*****BBM*****

```

BBMD

```

*****BBMD*****
      MACRO
&J      BBMD  &JII,&CODE
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---BBMD DISASSEMBLES BITMAPS.
*---&JII IS THE LENGTH OF THE ENTIRE STRING.
*---&CODE IS THE CODE WHICH IS TO BE INSERTED INTO THE STRING.
*-----
&J      STM    0,15,SERVICE
      LM     0,6,ZERO
      LR     BRY,BRY
      MVC    &CODE.(1),0(BRY)      *CODE PULLED OFF HEAD OF STRING.
      MVC    AA&SYSNDX.+1(1),1(BRY)
      MVC    CS2+3(1),1(BRY)
AA&SYSNDX MVC    LSX(0),2(BRY)      *BITMAP MOVED INTO LSX.
      A     BRY,CS2
      L     0,&JII
      S     0,=F'3'
      S     0,CS2
      ST    0,&JII      *NEW JII CALCULATED.
BA&SYSNDX MVI    CA&SYSNDX.+1,X'FF'
      C     0,=F'256'
      BNL   CA&SYSNDX
      BCTR  0,0
CA&SYSNDX STC    0,CA&SYSNDX.+1
      MVC    0(0,BRY),3(BRY)
      LA    BRY,256(0,BRY)
      LA    BRY,256(0,BRY)
      S     0,=F'256'
      BH    BA&SYSNDX
      LM    BRY,BRY,SERVICE+56
      L     2,CS2
      LA    2,1(0,2)
DA&SYSNDX SR     5,5
      LM    0,1,ZERO
      LA    6,8
      IC    1,LSX(5)
EA&SYSNDX SLL   1,24
      SLDL  0,1
      LTR   0,0
      BE    FA&SYSNDX
      ST    BRY,TL(3)
      LA    3,4(0,3)
      LA    4,1(0,4)
FA&SYSNDX LA    BRY,1(0,BRY)
      SR    0,0
      BCT  6,EA&SYSNDX
      LA    5,1(0,5)
      BCT  2,DA&SYSNDX
      S     3,=F'4'
      ST    3,CS10
      ST    4,CS11
      LM    0,15,SERVICE
      MEND
*****BBMD*****

```

BEGINS

```

*****BEGINS *****
      MACRO
&J      BEGINS
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---BEGINS IS THE INITIALIZING MACRO FOR THE AUXILIARY FILE.
*---BEGINS IS DEPENDENT ON THE VIRTUAL MEMORY CONFIGURATION.
*-----
&J      MVC   CHECK(4),=F'20'   *THE NUMBER OF TRACKS PER CYLINDER.
      MVC   SOS(4),=F'199'     *NUMBER OF CYLINDERS PER DISK.
      MVC   BWX(4),=F'1'      *DEVICE TYPE.
      MVC   LSX(4),=F'1'      *DEVICE NUMBER.
      MEND
*****BEGINS *****

```

BITTHROW

```

*****BITTHROW*****
MACRO
&J BITTHROW &NUMBITS,&BITMAP
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---BITTHROW THROWS A BIT MAP.
*---&NUMBITS IS THE NUMBER OF BITS TO BE THROWN.
*---&BITMAP IS THE ADDRESS OF THE BEGINNING OF THE BITMAP.
*-----
&J STM 0,15,DA&SYSNDX
LA 14,2
L 7,&NUMBITS *R7 = NUMBER OF ITEMS IN DIAGONAL
LR 6,7
SRL 6,3
SLL 6,3 *IS # ITEMS DIVISIBLE BY EIGHT?
CR 6,7
BE AA&SYSNDX
A 14,=F'1' *ONE EXTRA BYTE TO HOLD BITMAP
AA&SYSNDX SRL 6,3 *DIVIDE NUMBER OF BITS BY 8
AR 6,14 *ADD 2 FOR BITMAP+LENGTH INFO.
LA 15,&BITMAP
ST 6,EA&SYSNDX
MVC 0(2,15),EA&SYSNDX+2 *PUT LENGTH OF BITMAP IN PLACE
LA 15,2(15)
SR 6,6 *ZERO OUT FOR DIVIDE INSTRUCTION
D 6,=F'32' *R6 HOLDS REM., R7 HOLDS QUOT.
C 7,=F'0' *IS R7 ZERO--YES, THROW A BITMAP
BE CA&SYSNDX
BA&SYSNDX DICE =F'32',EA&SYSNDX,GENERATE
CLC EA&SYSNDX,=F'0' *IF BM IS ALL ZEROS--THROW AWAY
BE BA&SYSNDX
MVC 0(4,15),EA&SYSNDX *MOVE BITMAP INTO ARRAY
LA 15,4(15) *INCREMENT POINTER TO NEXT WORD
BCT 7,BA&SYSNDX *THROW AGAIN IF NECESSARY
C 6,=F'0' *IF R6=0 WE ARE DONE
BE FA&SYSNDX
CA&SYSNDX ST 6,EA&SYSNDX *SET UP ARGUMENT FOR DICE
DICE EA&SYSNDX,EA&SYSNDX,GENERATE
CLC EA&SYSNDX,=F'0' *IF BITMAP IS ALL ZEROS THROW
BE CA&SYSNDX
L 4,EA&SYSNDX *LOAD R4 WITH RANDOM NUMBER
L 3,=F'32'
SR 3,6 *DETERMINE # LEFT SHIFTS NEEDED
SLL 4,0(3) *LEFT ADJUST RANDOM NUMBER
ST 4,EA&SYSNDX
MVC 0(4,15),EA&SYSNDX *MOVE BITMAP INTO JOBLIST ITEM
B FA&SYSNDX *LEAVE ROUTINE
DA&SYSNDX DC 16F'0' *SAVE AREA FOR REGISTERS
EA&SYSNDX DS F *TEMPORARY STORAGE FOR RANDOM NO
FA&SYSNDX LM 0,15,DA&SYSNDX *RESTORE REGISTERS
MEND
*****BITTHROW*****

```

BULK

```

*****BULK *****
MACRO
&J BULK
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---BULK UPDATES THE BULK ADDRESS IN AUXFILE.
*-----
&J STM 0,3,SERVICE+8
APART BULK,0,1,2,2,3
C 0,=F'1'
BNL AA&SYSNDX *DEVICE TYPES NOT ASSIGNED.
A 2,=F'1' *TAPE IS BEING USED.
ASADD BULK,0,1,2,2,3
B BA&SYSNDX *EXIT.
AA&SYSNDX MVC BLANK+6(47),=C'MAIN FILE DEVICES>0 HAVE NOT YET BEEN AX
SSIGNED.'
PUT PRINT,BLANK
MVC BLANK+5(50),BLANK+4
B CL1 *EXIT
BA&SYSNDX LM 0,3,SERVICE+8
MEND
*****BULK *****

```

CALL1

```

*****CALL1 *****
MACRO
&J CALL1 &NAME,&ALINST
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---CALL1 IS FOR CALLING SINGLE COMPONENTS.
*---&NAME IS THE NAME OF THE COMPONENT.
*---&ALINST IS THE RETURN ADDRESS.
*-----
&J TESTL &NAME
B &ALINST *NEXT INSTRUCTION IS AT &ALINST.
MEND
*****CALL1 *****

```

CALL2

```

*****CALL2 *****
MACRO
&J CALL2 &NAME1,&NAME2,&ALINST
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---CALL2 CALLS TWO COMPONENTS CONSECUTIVELY.
*---&NAME1 IS THE NAME OF THE FIRST COMPONENT.
*---&NAME2 IS THE NAME OF THE SECOND COMPONENT.
*---&ALINST IS THE RETURN ADDRESS.
*-----
&J TESTL &NAME1
TESTL &NAME2
B &ALINST *NEXT INSTRUCTION IS AT &ALINST.
MEND
*****CALL2 *****

```

CALL3

```

*****CALL3 *****
      MACRO
&J      CALL3 &NAME1,&NAME2,&NAME3,&ALINST
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---CALL3 CALLS THREE COMPONENTS CONSECUTIVELY.
*---&NAME1 IS THE NAME OF THE FIRST COMPONENT.
*---&NAME2 IS THE NAME OF THE SECOND COMPONENT.
*---&NAME3 IS THE NAME OF THE THIRD COMPONENT.
*---&ALINST IS THE RETURN ADDRESS.
*-----
&J      TESTL &NAME1
          TESTL &NAME2
          TESTL &NAME3
          B      &ALINST          *NEXT INSTRUCTION IS AT &ALINST.
          MEND
*****CALL3 *****

```

CALL4

```

*****CALL4 *****
      MACRO
&J      CALL4 &NAME1,&NAME2,&NAME3,&NAME4,&ALINST
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---CALL4 CALLS FOUR COMPONENTS CONSECUTIVELY.
*---&NAME1 IS THE NAME OF THE FIRST COMPONENT.
*---&NAME2 IS THE NAME OF THE SECOND COMPONENT.
*---&NAME3 IS THE NAME OF THE THIRD COMPONENT.
*---&NAME4 IS THE NAME OF THE FOURTH COMPONENT.
*---&ALINST IS THE RETURN ADDRESS
*-----
&J      TESTL &NAME1
          TESTL &NAME2
          TESTL &NAME3
          TESTL &NAME4
          B      &ALINST          *NEXT INSTRUCTION IS AT &ALINST.
          MEND
*****CALL4 *****

```

CALL5

```

*****CALL5 *****
      MACRO
&J      CALL5 &NAME1,&NAME2,&NAME3,&NAME4,&NAME5,&ALINST
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---CALL5 CALLS FIVE COMPONENTS CONSECUTIVELY.
*---&NAME1 IS THE NAME OF THE FIRST COMPONENT.
*---&NAME2 IS THE NAME OF THE SECOND COMPONENT.
*---&NAME3 IS THE NAME OF THE THIRD COMPONENT.
*---&NAME4 IS THE NAME OF THE FOURTH COMPONENT.
*---&NAME5 IS THE NAME OF THE FIFTH COMPONENT.
*---&ALINST IS THE RETURN ADDRESS.
*-----
&J      TESTL &NAME1
          TESTL &NAME2
          TESTL &NAME3
          TESTL &NAME4
          TESTL &NAME5
          B      &ALINST          *NEXT INSTRUCTION IS AT &ALINST.
          MEND
*****CALL5 *****

```

CCD

```

*****CCD *****
MACRO
&J      CCD      &JII,&NBB
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---CCD COMPARES TWO CORDS OF A STRING (LENGTH IS R BYTES).
*---&JII IS THE NUMBER OF BYTES IN THE ENTIRE STRING.
*---&NBB CONTAINS THE LOCATION IN ARRAY YY OF THE COMPARAND CORD.
*-----
&J      STM      BRYY,BRY,TBRYY          *SAVE BRYY AND BRY
        MVC      AA&SYSNDX.+1(1),RM+3
        MVC      CA&SYSNDX.+1(1),RM+3
        SR       1,1
        LR       2,1
        L        3,R
        L        0,&JII
        CLI      SWITCH+3,X'00'          *IS SYSTEM IN FAST OR SLOW MODE?
        BE       BA&SYSNDX
        A        BRYY,&NBB
        S        0,&NBB          *GRO=NO BYTES IN STRING TO BE COMPARED
AA&SYSNDX MVC      CORD1(0),0(BRYY)      *COMPARE 'TO' AREA SET UP.
BA&SYSNDX SR       BRY,BRY              *BRY=POSITION IN TL ARRAY.
CA&SYSNDX CLC     0(0,BRYY),CORD1      *COMPARE CORDS FOR MATCH.
        BNE      DA&SYSNDX              *NO MATCH
        ST       BRYY,TL(BRY)
        LA      1,1(0,1)
        LA      BRY,4(0,BRY)
        C       BRY,=F'800'          *ARE THERE MORE THAN 200 MATCHES
        BE       EA&SYSNDX              *NO MORE COMPARES CAN BE MADE
        A        BRYY,R
        A        3,R
        CR      0,3          *CAN ANOTHER COMPARISON BE MADE
        BNH     EA&SYSNDX
        B       CA&SYSNDX              *YES
DA&SYSNDX LA      BRYY,1(0,BRYY)
        LA      3,1(0,3)
        CR      0,3          *CAN ANOTHER COMPARISON BE MADE
        BH     CA&SYSNDX              *YES
EA&SYSNDX L        3,RM          *FIND OUT IF SAVINGS WORTH-WHILE
        MR      2,1              *N*(R-1) IN GR3
        L       2,R
        LA      2,2(0,2)          *(R+2) IN GR2
        CR      3,2
        BH     FA&SYSNDX              *SAVINGS
        SR      1,1              *NO SAVINGS
FA&SYSNDX LM      BRYY,BRY,TBRYY      *RESTORE BRYY AND BRY
        MEND
*****CCD *****

```

COMPARE

```

*****COMPARE *****
      MACRO
&J      COMPARE &ADD1,&ADD2,&ADDL
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---COMPARE COMPARES THE SLOW MEMORY PARTS OF COMPOSITE ADDRESSES.
*---&ADD1 IS THE FIRST COMPOSITE ADDRESS.
*---&ADD2 IS THE SECOND COMPOSITE ADDRESS.
*---&ADDL IS THE NUMBER OF BYTES IN THE COMPOSITE ADDRESS.
*-----
&J      ST      0, SERVICE
        L      0,=F'&ADDL.'
        S      0,=F'4'          *FAST=3 BYTES.
        STC    0, AA&SYSNDX.+1
AA&SYSNDX CLC    &ADD1.( &ADDL. ), &ADD2
        L      0, SERVICE
      MEND
*****COMPARE *****

```

CONSTANT

```

*****CONSTANT*****
      MACRO
      CONSTANT
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---CONSTANT DEFINES THE PRINCIPAL LITERALS FOR THE SOLID SYSTEM.
*-----
ZERO      DC      10F'0'
CONM      DC      8XL1'FF'
BLANK     DC      136C' '
DZERO     DC      32CL1'0'
ASTERIK   DC      CL1'0',132CL1' #'
DASH      DC      CL1'0',132CL1'-'
HEXL      DC      C'0123456789ABCDEF' *FOR PRINT STATEMENTS ONLY.
ARRAY     DC      C' ARRAYS.'
ERRORF    DC      C'0FORMAT ERROR.'
ERRORP    DC      C'0ADDRESSING ERROR.'
      MEND
*****CONSTANT*****

```


COPAB

```

*****COPAB *****
      MACRO
&J      COPAB &RADD
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX COPAB  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---COPAB IS THE BOB SINCAVAGE VERSION OF COPAK(NUMERIC COMPRESSOR).
*---&RADD IS THE RETURN ADDRESS.
*-----
&J      CALL1 NUPAK,&RADD
      MEND
*****COPAB *****

```

COPAJ

```

*****COPAJ *****
      MACRO
&J      COPAJ &RADD
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX COPAJ  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---COPAJ IS THE JIM PERRY VERSION OF COPAK(ALPHANUMERIC COMPRESSOR).
*---&RADD IS THE RETURN ADDRESS.
*-----
&J      CALL2 ANPAKD,ANPAKC,&RADD
      MEND
*****COPAJ *****

```

COPAK

```

*****COPAK *****
      MACRO
&J      COPAK &RADD
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX COPAK  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---COPAK CALLS THE COMPRESSORS FOR THE COMPOSITE COMPRESSOR.
*---&RADD IS THE RETURN ADDRESS.
*-----
&J      CALL3 ANPAKD,NUPAK,ANPAKC,&RADD
      MEND
*****COPAK *****

```

COPAKEND

*****COPAKEND*****

MACRO

&J COPAKEND

*XXX.

*---COPAKEND IS EXECUTED AT THE END OF THE SOUTPUT COMPONENT.

*

```

&J      L      0,LJ
        C      0,ZERO
        BNL    AA&SYSNDX
        A      0,=F'1'
        ST     0,LJ
        BL     AA&SYSNDX
        MVC    SWITCH(4),ZERO
        MVC    LEXPCH(4),=F'1'
AA&SYSNDX L      1,NJOBS
        S      1,=F'1'
        ST     1,NJOBS
        BNH    BA&SYSNDX
*SET RSKIPS (TAPE) TO NEW VALUE (FOR BULK INFORMATION)
        L      15,ACARDRED
        BR     15
BA&SYSNDX MVC    NJOBS(4),RNJOBS
        CLI    NTASKS+3,X'00'
        BH     CA&SYSNDX
        L      15,AANSWER
        BR     15
CA&SYSNDX L      15,ALOCKFLE
        BR     15
MEND

```

*SWITCH TO FAST-MODE
*DON'T PUNCH PCORDS TABLE

*NEXT JOB FOR COPAK

*EXECUTE POSTOP IN DISPENSE

*SEARCH FOR NEW ITEM

*****COPAKEND*****

CREATE

```

*****CREATE *****
MACRO
&J CREATE &ADDL,&LSLOW,&LFAST,&NTRKS,&TRKL,&MATRIXL,&MATRIXS
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---CREATE CREATES SUB ARRAYS WHEN THEY ARE NEEDED FOR SSEARCH.
*---&ADDL IS THE NUMBER OF BYTES IN THE COMPOSITE ADDRESSES.
*---&LSLOW IS THE LENGTH OF THE SLOW PART OF COMPOSITE ADDRESSES.
*---&LFAST IS THE LENGTH OF THE FAST PART OF COMPOSITE ADDRESSES.
*---&NTRKS IS THE NUMBER OF TRACKS PER MEMORY BLOCK.
*---&TRKL IS THE NUMBER OF BYTES PER TRACK.
*---&MATRIXL IS THE NUMBER OF ITEMS IN THE PRINCIPAL SUB-ARRAYS.
*---&MATRIXS IS THE NUMBER OF ITEMS IN THE SECONDARY SUB-ARRAYS.
*-----
&J LR IR5,IR1
AR IR5,IR6 *IR6=FAST MEMORY ADDRESS OF NEW SUB-BK
L 1,=F'&TRKL'
L 0,ZERO
M 0,=F'&NTRKS'
LR BRY,1 *BRY=LENGTH OF MEM.BK.
LH 0,DUN1
L 1,=F'&ADDL'
CH 0,ZERO
BNL AA&SYSNDX
L 1,=F'&LSLOW'
SR 0,0
AA&SYSNDX AR 1,0
ST 1,DUN7
CLI MSIGNAL+2,X'02'
BL BA&SYSNDX *M AND J ARRAYS NEEDED
BE CA&SYSNDX *PRINCIPAL DIAGONAL.
L 14,=F'&MATRIXS' *ALL OTHER DIAGONALS.
B DA&SYSNDX
BA&SYSNDX L 14,=F'20'
B DA&SYSNDX
CA&SYSNDX L 14,=F'&MATRIXL'
DA&SYSNDX SR 0,0
MR 0,14
A 1,=F'4'
LR 0,1 *REGO CONTAINS CONTINUANCE ADDR.
A 1,=F'&ADDL' *R1=LENGTH OF SUB-ARRAY.
CR 1,BRY
BNH FA&SYSNDX
BCT 14,DA&SYSNDX
EA&SYSNDX L 15,SAVEYY
MVC 0(&ADDL,15),EMPTY
L 15,BWX+76 *=LAST INSERTED ADDR.
MVC 0(&ADDL,15),ZERO
APART EMPTY+&ADDL,0,1,2,3,4
C 14,ZERO

```

CREATE (CONT.)

```

BNE HA&SYSNDX
MVC BLANK+5(93),=C'ONE OF THE SCREENS PLUS ADDRESS EXCEEDSX
THE FAST MEMORY ALLOCATED.THE SYSTEM HAS BEEN PURGED.'
MVI BLANK,C'0'
PUT PRINT,BLANK
B CLI
FA&SYSNDX ST 0,DUN7 *RELATIVE CONTINUANCE ADDRESS STORED.
MVC 0(4,IR5),DUN7 *LEAD FOUR BYTES OF SUB-BLOCK.
A IR5,=F'4'
STM 0,4,C1
APART EMPTY,0,1,2,3,4
L 14,DUN7
A 14,=F'&ADDL'
AR 14,4
A 15,SAVEYY
S 15,AYY
CR 14,15
BH GA&SYSNDX
LR 4,14
ASADD EMPTY,0,1,2,3,4
LM 0,4,C1
GA&SYSNDX B MA&SYSNDX *REG4=EMPTY FAST MEMORY ADDRESS.
LINKHOLE DUN7,EMPTY
COMPARE EMPTY,EMPTY+&ADDL,&ADDL
BE EA&SYSNDX
L 15,SAVEYY
MVC 0(&ADDL,15),EMPTY *BLOCK-EMPTY STORED.
NI MSIGNAL,X'7F' *DONT CREATE A SUPER-BLOCK.
MVC ADDRESS(&ADDL),EMPTY+&ADDL
L 15,BWX+76
MVC 0(&ADDL,15),EMPTY+&ADDL
LM 0,4,C1
B NEWBLOCK
HA&SYSNDX L 4,SAVEYY
S 4,AYY
A 2,=F'&NTRKS.'
NI MSIGNAL,X'7F'
OI MSIGNAL,X'80'
IA&SYSNDX C 2,CHECK *CHECK=TRACKS/CYLINDER.
BNH JA&SYSNDX
S 2,CHECK *CHECK=TRACKS/CYLINDER.
A 3,=F'1'
B IA&SYSNDX
JA&SYSNDX C 3,SOS
BNH LA&SYSNDX
L 2,=F'1' *FIRST TRACK FOR THE NEXT DISK.
L 3,ZERO *FIRST CYLINDER FOR THE NEW DISK.
A 1,=F'1' *NEXT DEVICE NUMBER.
C 1,LSX *LSX=DEVICES OF ONE TYPE.

```

CREATE (CONT.)

```

BL      KA&SYSNDX                *NEXT NUMBERED DEVICE.
A       0,=F'1'
SR      1,1
KA&SYSNDX C      0,BWX                *BWX=TYPES OF DEVICES.
BL      LA&SYSNDX
MVC     WORKA(133),BLANK
MVC     WORKA(78),=C'   GET A LARGER CONFIGURATION OR INCREASEX
        THE NUMBER OF DISKS ON THIS SYSTEM. '
PUT     PRINT,WORKA
L       15,SOS                    *SOS=CYLINDERS/DEVICE
L       14,ZERO
M       14,CHECK                  *CHECK=TRACKS/CYLINDER.
M       14,BWX                    *BWX=NUMBER OF DEVICES OF EACH TYPE.
M       14,LSX                    *LSX=NO OF DIFFERENT TYPES OF DEVICES.
M       14,=F'&STRKL.'           *TIMES THE NUMBER OF BYTES PER TRACK.
ST      15,DUN1                   *DUN1=NO OF BYTES IN FILE SYSTEM.
DECPC   FILESIZE,DUN1,BYTES
MVI     MSIGNAL+1,X'FF'          *TERMINATION SIGNAL.
B       CL1                       *EXIT.
LA&SYSNDX ASADD  EMPTY+&ADDL,0,1,2,3,4
A       4,=F'&ADDL'
ASADD   ADDRESS,0,1,2,3,4
MVC     EMPTY(&ADDL),ADDRESS
L       15,BWX+76
MVC     0(&ADDL,15),EMPTY
LM      0,4,C1
B       NEWBLOCK
MA&SYSNDX LR      0,1
S       0,=F'4'
        COMPARE EMPTY,EMPTY+&ADDL,&ADDL
        BNE      NA&SYSNDX
NA&SYSNDX MVC     EMPTY+&ADDL,(&ADDL),EMPTY
ST      0,DUM1
        LARGEXC DUM1+2,0(IR5),0(IR5)
L       15,SAVEYY
MVC     0(&ADDL,15),EMPTY
MEND
*****CREATE*****

```

CSCREEN

```

*****CSCREEN*****
      MACRO
&J      CSCREEN &HLGTH,&ADD1,&ADD2
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---CSCREEN COMPARES SCREEN &ADD1 TO SCREEN &ADD2 AND ZERO.
*---&HLGTH IS THE LENGTH OF SCREEN BEGINNING IN &ADD1(HALF-WORD).
*---&ADD1 IS THE LOCATION OF THE FIRST EXECUTIVE POINTER.
*---&ADD2 IS THE ADDRESS OF THE SECOND EXECUTIVE POINTER.
*-----
&J      STM      1,4, SERVICE                      *SAVED REGISTERS.
      MVI      JI,X'00'
      L        3,=F'256'
      LH       4,&HLGTH
      XC       DUMX(256),DUMX
      MVI      BA&SYSNDX.+1,X'FF'
      MVI      DA&SYSNDX.+1,X'FF'
      LA       1,&ADD1
      LA       2,&ADD2
AA&SYSNDX CR      3,4
      BNH      BA&SYSNDX
      LR       3,4
      S        4,=F'1'
      STC      4,BA&SYSNDX+1
      STC      4,DA&SYSNDX+1
BA&SYSNDX CLC     0(256,1),0(2)
      BNE      CA&SYSNDX                      *IT IS NOT ZERO(> OR <)
GA&SYSNDX AR      1,3
      AR       2,3
      SR       4,3
      BH       AA&SYSNDX
      B        FA&SYSNDX                      *EQUAL. (JI=X'00')
CA&SYSNDX BNH     DA&SYSNDX
      OI       JI,X'04'                      *HIGH. (JI=X'04')
      B        FA&SYSNDX
DA&SYSNDX CLC     0(256,1),DUMX
      BE       EA&SYSNDX
      OI       JI,X'02'                      *LOW. (JI=X'02')
      B        FA&SYSNDX
EA&SYSNDX TM      JI,X'01'                      *ZERO. (JI=X'01')
      BO       GA&SYSNDX
      OI       JI,X'01'
      B        GA&SYSNDX
FA&SYSNDX LM      1,4, SERVICE
      MEND
*****CSCREEN*****

```

CSSCRN

```

*****CSSCRN *****
      MACRO
&J      CSSCRN &HLGTH,&ADD1,&ADD2
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---CSSCRN IS THE EXTENSION OF THE IBM CLC INSTRUCTION.
*---&HLGTH(HALF-WORD) CONTAINS THE LENGTH OF ITEMS TO BE COMPARED.
*---&ADD1 IS THE ADDRESS OF THE FIRST ITEM.
*---&ADD2 IS THE ADDRESS OF THE SECOND ITEM.
*-----
&J      STM      0,2,SERVICE
        LH       0,&HLGTH
        LA       1,&ADD1
        LA       2,&ADD2
        MVI     BA&SYSNDX+1,X'FF'
AA&SYSNDX C       0,=F'256'
        BNL     BA&SYSNDX
        S       0,=F'1'
        STC     0,BA&SYSNDX+1
BA&SYSNDX CLC     0(256,1),0(2)
        BNE     CA&SYSNDX
        A       1,=F'256'
        A       2,=F'256'
        S       0,=F'256'
        BH     AA&SYSNDX
CA&SYSNDX CLC     ZERO(4),ZERO+4
        LM      0,2,SERVICE
        MEND
*****CSSCRN *****

```

DCB MEM

```

*****DCB MEM *****
      MACRO
&J      DCB MEM
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---DCB MEM OPENS AND CLOSES THE DCB'S FOR THE GLOBAL MEMORY.
*-----
GLOBAL1  DCB     DSCRG=DA,MACRF=(RK,WAK),DDNAME=COPAK7,          X
        OPTCD=E,RECFM=F,KEYLEN=3,LIMCT=4000,                  X
        BLKSIZE=7290,BUFNO=2,BFALN=F,BUFL=7294
WORKM    DS      40F
CLOSE    CLOSE   GLOBAL1
        B       PMARRAY                      *TO CLOSE OTHER DEVICES.
&J      OPEN    (GLOBAL1,(OUTPUT))
        MEND
*****DCB MEM *****

```


DEVICE

```

*****DEVICE *****
      MACRO
&J      DEVICE &INPXT,&OUTPXT,&RSKIPS,&SLENGTH,&LLENGTH,&RNOS,&CORDS
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---DEVICE READS THE DEVICE COMMANDS FOR THE SOLID SYSTEM.
*---&INPXT DESIGNATES THE INPUT DEVICE FOR THE BULK INFORMATION.
*---&OUTPXT DESIGNATES THE OUTPUT DEVICE FOR THE BULK INFORMATION.
*---&RSKIPS IS THE NUMBER OF STRINGS TO BE SKIPPED ON THE INPUT TAPE.
*---&SLENGTH IS THE MINIMUM LENGTH OF EACH STRING READ FROM TAPE.
*---&LLENGTH IS THE NUMBER OF BYTES IN THE LABEL OF EACH STRING.
*---&RNOS IS THE NUMBER OF STRINGS TO BE PROCESSED FROM TAPE.
*---&CORDS IS THE NUMBER OF PERMANENT CORDS USED BY SANPAKC--FAST MODE.
*-----
&J      CALL1 COMANDD,STRING
      MEND
*****DEVICE *****

```

DEVICES

```

*****DEVICES *****
      MACRO
&J      DEVICES
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---DEVICES OPENS AND CLOSSES ALL PERIPHERAL DEVICES FOR SOLID.
*-----
HERE      LR      3,13
          LA      13,SAVEAREA
          ST      13,8(3)
          ST      3,4(13)
          LM      BR3,BR4,BASE
          CALL1 OPEN,SPRINGER
RTRANMIT  CALL1 ACTION,BA&SYSNDX          *DECOMPRESSION FAILURE.
ECOMP     MVC     BLANK+5(20),=C'COPAK5 (COMPRESSED '
AA&SYSNDX MVC     BLANK+26(60),=C'TAPE) CLOSED BECAUSE A READ ERROR OCCUR
          RRED.TAPE STATUS WAS:-'
          B      MEXIT
EDECPC    MVC     BLANK+5(20),=C'COPAK4 (UNCOMPRESSED'
          B      AA&SYSNDX
EOUTP     MVC     BLANK+5(76),=C'COPAK6 (OUTPUT TAPE) CLOSED BECAUSE A WX
          RITE ERROR OCCURRED.TAPE STATUS WAS:-'
          B      MEXIT
EJLST     MVC     BLANK+5(20),=C'COPAK8 (JOBLIST ITEM'
          B      AA&SYSNDX
MEXIT     PUT     PRINT,BLANK
          MVC     BLANK+5(90),BLANK+4
          ST      7,JII
BA&SYSNDX DECPC   STRING,NOS,FAILED
          DECPC  READ,JII,BYTES
CL1       TIME    BIN
          S      0,TTIME
          ST      0,TTIME
          DECPC  TOTAL,NOS,STRINGS
          DECPC  TOTAL,TTIME,SECS
          LM      7,8,MEMADD          *SMEMORY ADDRESSES LOADED.
          L      15,ASAVEFM          *TO SAVE RESIDENT MEMORY BLOCK.
          BALR   14,15
PMARRAYR  CALL1  SHUT,CA&SYSNDX
CA&SYSNDX L      13,4(13)
          RETURN (14,12)
          MEND
*****DEVICES *****

```

DICE

```

*****DICE *****
MACRO
&J DICE &NOBITS,&RANDNO,&ODDNO
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXDICE XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---DICE IS A VARIABLE SIDED DICE.
*---&NOBITS IS THE NUMBER OF RANDOM BITS IN THE GENERATED NUMBER.
*---&RANDNO IS THE LOCATION WHERE THE RANDOM NUMBER WILL BE STORED.
*---&ODDNO IS THE MULTIPLICAND THAT IS TO BE USED(=CLOCK TIME IF 0).
*-----
&J STM 0,15,SERVICE
L 1,=F'1'
XR 3,3
L 4,&NOBITS
C 4,=F'1'
BL DA&SYSNDX
L 0,&ODDNO
C 0,=F'0'
BNE BA&SYSNDX
TIME BIN
L 1,=F'1'
LR 2,0
SLL 0,1
ALR 0,1
N 2,=F'2047'
A 2,=F'5'
AA&SYSNDX LR 3,0
ALR 0,0
ALR 0,3
BCT 2,AA&SYSNDX
L 3,=F'0'
BA&SYSNDX SLL 3,1
CA&SYSNDX LR 2,0
ALR 0,2
BO EA&SYSNDX
ALR 0,2
BO FA&SYSNDX
ALR 3,1
DA&SYSNDX B FA&SYSNDX
MVC BLANK+5(23),=C'NO DICE WITH ZERO BITS.'
PUT PRINT,BLANK
MVC BLANK+5(24),BLANK+4
B CL1 *EXIT
EA&SYSNDX ALR 0,2
BO CA&SYSNDX
FA&SYSNDX BCT 4,BA&SYSNDX
ST 0,&ODDNO
ST 3,&RANDNO
LM 0,15,SERVICE
MEND

```

```

*****DICE *****

```

DISPENSE

```

*****DISPENSE*****
MACRO
&J    DISPENSE &RADD
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---DISPENSE PROCESSES RESULT OF EACH SEARCH.
*---&RADD IS THE RETURN ADDRESS.
*-----
&J    L        1,NTASKS
      S        1,=F'1'
      ST       1,NTASKS
      BNM      BA&SYSNDX          *SOME TASKS REMAIN
***ALL TASKS HAVE BEEN COMPLETED.
AA&SYSNDX  CLI   LJ+3,X'01'
          BE    NEWJOB
          BH    CL1
          B     STRING
BA&SYSNDX  CLI   NJOBS+3,X'00'
          BNE   CA&SYSNDX
          MVC   NJOBS(4),RNJOBS
          CLI   NTASKS+3,X'00'
          BE    AA&SYSNDX
          B     LOOKFILE          *SEARCH FOR NEXT ITEM
CA&SYSNDX  B     &RADD
          MEND
*****DISPENSE*****

```

DISPOSE

```

*****DISPOSE *****
MACRO
&J    DISPOSE &RADD
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---DISPOSE IS THE SPECIAL VERSION OF 'DISPENSE' FOR THE COMPRESSORS.
*---&RADD IS THE RETURN ADDRESS.
*-----
&J    L        1,NTASKS
      S        1,=F'1'
      ST       1,NTASKS
      BNM      BA&SYSNDX          *SOME TASKS REMAIN
***ALL TASKS HAVE BEEN COMPLETED.
AA&SYSNDX  MVC   NTASKS(4),=F'1'
          MVC   NJOBS(4),=F'1'
          CLI   LJ+3,X'01'
          BE    NEWJOB
          BH    CL1
          B     STRING
BA&SYSNDX  CLI   NJOBS+3,X'00'
          BNE   CA&SYSNDX
          MVC   NJOBS(4),=F'1'
          CLI   NTASKS+3,X'00'
          BE    AA&SYSNDX
          B     LOOKFILE          *SEARCH FOR NEXT ITEM
CA&SYSNDX  B     &RADD
          MEND
*****DISPOSE *****

```


ENDSS

```

*****ENDSS *****
MACRO
&J      ENDSS &RADD
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---ENDSS TESTS WHETHER ALL SUBSTRINGS HAVE BEEN PROCESSED BY SNUPAK.
*---&RADD IS THE RETURN ADDRESS.
*-----
&J      LM      0,1,CS1
        CLI     MODE+3,X'00'
        BNE    AA&SYSNDX
        L      7,SOS(1)
        ST     7,NDR
AA&SYSNDX CONSTRUCT SOS(1),JI,NDR
        L      7,JI
        SLL   7,1
        SRL   7,1
        ST     7,JI
        CLI     MODE+3,X'00'
        BNE    EA&SYSNDX
        L      BRYY,CS15
        A      BRYY,JI
        S      BRYY,=F'4'
        L      7,CHECK(1)
        MVC    DUMX(4),0(BRYY)
        C      7,DUMX
        BE     EA&SYSNDX
        MVC    BLANK+4(72),=C' CHECK FAILED ON SUBSTRING=      .CHEX
                CK(1)=          ;0(BRYY)=          .
        LA     7,BLANK+66
        L      2,=F'8'
        L      5,DUMX
BA&SYSNDX L      4,ZERO
        SLDL   4,4
        AL     4,=F'240'
        STC    4,0(7)
        A      7,=F'1'
        BCT    2,BA&SYSNDX
        EXTRAKT JII,NV,PARM
        L      0,CS1
        S      0,NV
        LPR    0,0
        A      0,=F'1'
        ST     0,NV
        LA     7,BLANK+31
        L      2,=F'8'
        L      5,NV
CA&SYSNDX L      4,ZERO
        SLDL   4,4
        AL     4,=F'240'

```

ENDSS (CONT.)

```

STC 4,0(7)
A 7,=F'1'
BCT 2,CA&SYSNDX
LA 7,BLANK+49
LA BRYY,CHECK(1)
L 2,=F'8'
L 5,0(BRYY)
DA&SYSNDX L 4,ZERO
SLDL 4,4
AL 4,=F'240'
STC 4,0(7)
A 7,=F'1'
BCT 2,CA&SYSNDX
PUT PRINT,BLANK
MVC BLANK+4(72),BLANK+3
B RTRANMIT
EA&SYSNDX A 1,=F'4'
ST 1,RM
S 0,=F'1'
ST 0,NV
CLI NV+3,X'00'
BNE 6RADD
MEND

```

*FOR NEXT SUBSTRING.

*****ENDSS*****

ENTRANCE

*****ENTRANCE*****

MACRO

ENTRANCE

XX

*--ENTRANCE DEFINES THE ENTRY POINTS FOR THE EXTENDED CONTROL ROUTINES

*

ENTRY ACTION
 ENTRY ANPAKC
 ENTRY ANPAKCC
 ENTRY ANPAKD
 ENTRY ANPAKDC
 ENTRY COMANDC
 ENTRY COMANDS
 ENTRY CYCLIC
 ENTRY GENITEM
 ENTRY JOBLIST
 ENTRY MATCH
 ENTRY MEMORY
 ENTRY MEMORYR
 ENTRY MOBILE
 ENTRY NORMAL
 ENTRY NUPAK
 ENTRY NUPAKCN
 ENTRY NUPAKR
 ENTRY OUTPUT
 ENTRY OUTPUTC
 ENTRY PMARRAY
 ENTRY PRINT
 ENTRY PUNXH
 ENTRY READC
 ENTRY READCON
 ENTRY READT
 ENTRY REFLECT
 ENTRY REID
 ENTRY RESULT
 ENTRY SAVEFM
 ENTRY SEARCH
 ENTRY STATECL
 ENTRY TLATOR1
 ENTRY TLATOR2
 ENTRY TLATOR3
 ENTRY TLATOR4
 ENTRY TLATOR5
 ENTRY XCHANGE
 MEND

*****ENTRANCE*****

FACE

```

*****FACE *****
MACRO
&J FACE &NFACES,&RANDNO,&ODDNO
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---FACE IS THE VARIABLE SIDED DICE.
*---&NFACES IS THE MAXIMUM NUMBER OF FACES ON THE DICE.
*---&RANDNO IS THE LOCATION WHERE THE RANDOM NUMBER WILL BE STORED.
*---&ODDNO IS THE MULTIFLICAND TO BE USED(CLOCK TIME USED IF &ODDNO=0).
*-----
&J STM 0,15,SERVICE
AA&SYSNDX L 1,=F'1'
LM 3,4,ZERO
L 0,&NFACES
C 0,=F'1'
BNH FA&SYSNDX
SR 0,1
BA&SYSNDX A 4,=F'1'
SRL 0,1
CR 0,3
BH BA&SYSNDX
L 0,&ODDNO
CR 0,3
BNE DA&SYSNDX
TIME BIN
L 1,=F'1'
LR 2,0
SLL 0,1
ALR 0,1
N 2,=F'2047'
A 2,=F'5'
CA&SYSNDX LR 3,0
ALR 0,0
ALR 0,3
BCT 2,CA&SYSNDX
L 3,=F'0'
DA&SYSNDX SLL 3,1
EA&SYSNDX LR 2,0
ALR 0,2
BO GA&SYSNDX
ALR 0,2
BO HA&SYSNDX
ALR 3,1
B HA&SYSNDX
FA&SYSNDX MVC BLANK+5(31),=C'NO DICE WITH ONE OR ZERO FACES.'
PUT PRINT,BLANK
MVC BLANK+5(32),BLANK+4
B CL1 *EXIT.
GA&SYSNDX ALR 0,2
BO EA&SYSNDX
HA&SYSNDX BCT 4,DA&SYSNDX
ST 0,&ODDNO
C 3,&NFACES
BNL AA&SYSNDX
ST 3,&RANDNO
LM 0,15,SERVICE
MEND
*****FACE *****

```

FIND

```

*****FIND *****
MACRO
&J      FIND  &JII,&CODE
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---FIND FINDS THE CODE-WORD IN THE STRING.
*---&JII IS THE LENGTH OF THE ENTIRE STRING.
*---&CODE IS THE CODE THAT IS TO BE FOUND.
*-----
&J      STM    BRYY,BRY,TBRYY
        LM     1,3,ZERO
        I      0,&JII
AA&SYSNDX  CLC    0(1,BRYY),&CODE          *COMPARE CODE FOR MATCH.
        BE     BA&SYSNDX                    *MATCH
        LA     BRYY,1(0,BRYY)              *NO MATCH.
        BCT   0,AA&SYSNDX
        B      CA&SYSNDX
BA&SYSNDX  ST     BRYY,TL(2)              *STORE ADDRESS OF MATCH IN TL ARRAY.
        LA     2,4(0,2)                    *GR2=POSITION IN 'TL' ARRAY.
        LA     3,1(0,3)                    *GR3=NUMBER OF MATCHES FOUND.
        LA     BRYY,1(0,BRYY)
        C      3,=F'200'
        BE     CA&SYSNDX
        BCT   0,AA&SYSNDX
CA&SYSNDX  LM     BRYY,BRY,TBRYY
        MEND
*****FIND *****

```

GETJLIST

```

*****GETJLIST*****
MACRO
&J      GETJLIST &JLNPXT,&JLRSKIP,&JLTRAN,&JLNORM,&KLENGTH,&NJOB,&X
        NTASKS,&MVALUE,&JVALUE,&NUMDIAG,&GNERATE
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---GETJLIST FETCHES,TRANSLATES AND NORMALIZES THE JOBLIST.
*---&JLNPXT DESIGNATES THE INPUT DEVICE(TAPE(0);CARDS(1);-GEN(10)).
*---&JLRSKIP IS THE NUMBER OF RECORDS TO SKIP ON TAPE(&JLNPXT=0).
*---&JLTRAN DESIGNATES THE TRANSLATOR(NONE(0),1,2,3,4,OR 5).
*---&JLNORM IS THE NORMALIZATION INDICATOR(NORM(1);DONT NORM(0)).
*---&KLENGTH IS THE NUMBER OF BYTES IN EACH KERNEL OR ELEMENT OF THE IR
* &NJOB IS THE NUMBER OF BULK ITEMS TO BE STORED FOR EACH INFO. PATH.
***&NTASKS,&MVALUE,&JVALUE,&NUMDIAG,AND &GNERATE ARE ON THE NEXT CARD.
*---&NTASKS IS THE NUMBER OF JOBLIST ITEMS TO BE GENERATED FOR THE JOB.
*---&MVALUE IS THE VALUE OF 'M' FOR THE JOBLIST ITEM TO BE GENERATED.
*---&JVALUE IS THE MAXIMUM VALUE FOR EACH SMALL 'M' IN THE JOBLIST ITEM
*---&NUMDIAG IS THE TOTAL NUMBER OF ALL TYPES OF DIAGONALS IN THE ITEM.
*---&GNERATE IS THE ODD NUMBER TO BE USED BY RANDOM NUMBER GENERATORS.
*-----
&J      CALL1 JOBLIST,AA&SYSNDX
AA&SYSNDX  NOPR 1
        MEND
*****GETJLIST*****

```


HEXPC

```

*****HEXPC *****
MACRO
&J      HEXPC &NAME,&LOC,&LENGTH
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---HEXPC PRINTS LABELED STRINGS OF HEXADECIMAL INFORMATION.
*---&NAME IS THE LABEL (LIMIT OF 20 BYTES).
*---&LOC IS THE BEGINNING ADDRESS OF THE STRING.
*---&LENGTH CONTAINS THE LENGTH OF THE STRING(LIMIT=65 BYTES).
*-----
&J      STM      0,15,SERVICE
        L        0,&LENGTH          *LENGTH REGISTER.
        LPR      0,0                *SET POSITIVE.
        C        0,=F'65'
        BNH      AA&SYSNDX
        L        0,=F'65'
AA&SYSNDX MVC     DUMX(27),=C'      &NAME. =
        LA      14,DUMX+5
        C        0,ZERO
        BH      BA&SYSNDX
        L        0,=F'65'
BA&SYSNDX CLI     0(14),C'='
        BE      CA&SYSNDX
        A        14,=F'1'
        B        BA&SYSNDX
CA&SYSNDX A        14,=F'1'
        XC      0(133,14),0(14)
        A        14,=F'1'
        LA      1,HEXL
        LA      2,&LOC
DA&SYSNDX L        3,=F'4'
        MVC     DUM1(4),0(2)
        L        5,DUM1
        A        2,=F'4'
EA&SYSNDX L        6,=F'2'
FA&SYSNDX SR      4,4
        SLDL    4,4
        AR      4,1
        MVC     0(1,14),0(4)
        A        14,=F'1'
        BCT     6,FA&SYSNDX
        S        0,=F'1'
        BNH      GA&SYSNDX
        BCT     3,EA&SYSNDX
        B        DA&SYSNDX
GA&SYSNDX PUT     PRINT,DUMX
        LM      0,15,SERVICE
        MEND
*****HEXPC *****

```


INSERT

```

*****INSERT *****
MACRO
&J      INSERT &ADDL,&LFAST,&LTHAYY
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---INSERT INSERTS EXECUTIVE POINTERS IN SUB-ARRAYS.
*---&ADDL IS THE LENGTH OF THE COMPOSITE ADDRESSES.
*---&LFAST IS THE LENGTH OF THE FAST PART OF COMPOSITE ADDRESSES.
*---&LTHAYY IS THE LENGTH OF THE PRINCIPAL DATA ARRAY.
*-----*
&J      CLI      JI,X'01'
        BNE      CA&SYSNDX
        STM      0,6,C1
AA&SYSNDX L      IR1,LEXICON+8
        LH      0,DUN1
        A      0,=F'&ADDL'
        STH     0,MASK1
        LMOVE   MASK1,0(IR6),0(IR1)
        MVC     ADDRESS(&ADDL),LEXICON+16
        L      IR1,LEXICON+4
        LMOVE   DUN1,0(IR2),0(IR1)      *ORIGINAL SCREEN IN JOBLIST
        MVI     LEXICON,X'00'
        LM      0,6,C1
        OI      MSIGNAL,X'20'
        B      TBADDNE
*****  CONTINUANCE INSERTION MADE HERE      ***** CONTINUANCE*****
BA&SYSNDX LMOVE   DUN1,0(IR6),0(IR2)      *SCREEN STORED.
        AR      IR2,BRYY
        AR      IR6,BRYY
        B      NA&SYSNDX
CA&SYSNDX LR      0,BRY
        LR      1,IR6
DA&SYSNDX CLC     0(4,1),ZERO
        BE      HA&SYSNDX
        AR      1,0
        CR      1,IR3
        BL      DA&SYSNDX
*****  CONTINUANCE INSERTION MADE HERE      ***** CONTINUANCE*****
        SR      1,0
        STM     0,6,C1
        SR      IR3,0
        TM      LEXICON,X'01'
        BO      EA&SYSNDX
        L      IR1,AYY
        A      IR1,=F'&LTHAYY'
        LH      2,DUN1
        SR      IR1,2
        ST      IR1,LEXICON+4      *HC3 ADDRESS.
        SR      IR1,0      *HC2 ADDRESS.
        ST      IR1,LEXICON+8      *HC2 ADDRESS.

```

INSERT (CONT.)

```

SR      IR1,0
ST      IR1,LEXICON+12          *HC1 ADDRESS.
MVC     LEXICON+16(&ADDL),ADDRESS
L       IR1,LEXICON+4
EA&SYSNDX LMOVE DUN1,0(IR1),0(IR2)      *SCREEN SAVED.
TM      LEXICON,X'02°
BO      GA&SYSNDX                *FRESH PROPOGATION.
OI      LEXICON,X'02°
L       IR1,LEXICON+8          *HC2 ADDRESS
FA&SYSNDX STH 0,MASK1
LMOVE  MASK1,0(IR1),0(IR3)      *LAST EX. POINTER IS SAVED.
LM      0,6,C1
B       HA&SYSNDX
GA&SYSNDX OI  LEXICON,X'04°
L       IR1,LEXICON+12        *HC1 ADDRESS.
B       FA&SYSNDX
***** CONTINUANCE INSERTION MADE HERE ***** CONTINUANCE*****
HA&SYSNDX LR  BRY,1              *CREATE A HOLE IN SUB-BLOCK.
SR      BRY,IR6                *HOLE SIZE
LR      BRY,1
S       BRY,=F'1°              *LAST FILLED.
LR      1,BRY                    EMPTY.                *FIRST
AR      1,0
RMVC   0,BRY,1,0,BRY
LH     BRY,DUN1                  *BRY=SCREEN LENGTH.
LR     BRY,BRY
A      BRY,=F°&ADDL°            *BRY=ITEM LENGTH.
STM    BRY,BRY,C15
***** CONTINUANCE INSERTION MADE HERE ***** CONTINUANCE*****
STM    0,6,C1
CLI    LEXICON,X'00°
BE     MA&SYSNDX                *NORMAL TERMINATION.
TM     LEXICON,X'04°
BZ     JA&SYSNDX
L      IR1,LEXICON+8          *HC2 ADDRESS.
STH   0,MASK1
LMOVE  MASK1,0(IR6),0(IR1)      *INSERTION OF CARRY MADE.
L      IR3,LEXICON+12          *HC1 ADDRESS.
LMOVE  MASK1,0(IR1),0(IR3)
NI     LEXICON,X'03°
IA&SYSNDX LMOVE DUN1,0(IR2),0(IR1) *JOB-LIST INSERT.
LM     0,6,C1
OI     MSIGNAL,X'20°
LR     IR6,IR3
B      NA&SYSNDX
JA&SYSNDX TM  LEXICON,X'01°
BZ     LA&SYSNDX
B      AA&SYSNDX
KA&SYSNDX L  IR1,LEXICON+8
B      IA&SYSNDX
LA&SYSNDX STH 0,MASK1
LARGE  MASK1,0(IR6),0(IR6)
OI     LEXICON,X'01°
B      KA&SYSNDX
***** CONTINUANCE INSERTION MADE HERE ***** CONTINUANCE*****
MA&SYSNDX LH 0,DUN1
AR     IR6,0
MVC   0(&ADDL,IR6),ZERO
SR    IR6,0
B     BA&SYSNDX
NA&SYSNDX NOPR 1
MEND

```

*****INSERT*****

JAR

*****JAR *****

MACRO

&J JAR &JII,&NR
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXJAR XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

*---JAR INSERTS THE NUMBER OF BYTES(JII) AND R'S AT HEAD OF STRING.

*---&JII IS THE NUMBER OF BYTES IN THE ENTIRE STRING.

*---&NR IS THE NUMBER OF R'S THAT WERE USED TO MAKE SAVINGS.

&J	STM	BRYY,BRY,TBRY	*SAVE BRY AND BRY
	L	0,&JII	
	LR	2,0	
	A	0,=F'4'	
	SLA	0,8	
	D	0,&NR	
	L	BRYY,SBRY	
	AR	BRYY,2	
	BCTR	BRYY,0	
	LR	BRY,BRY	
	RMVC	4,2,BRY,0,BRY	
	L	BRYY,SBRY	
	ST	0,0(BRY)	
	LA	2,4(0,2)	
	ST	2,&JII	
	LM	BRYY,BRY,TBRY	*RESTORE BRY AND BRY
	MEND		

*****JAR *****

JBLISTI (CONT.)

```

L      3,DUMX+4
ST     3,CA&SYSNDX
XC     DUMX+4(4),DUMX+4
LTR    3,3
BH     AA&SYSNDX
BA&SYSNDX LA 1,1(1)
STH    1,DUMX
MVI    0(2),C'*'
MVC    0(2,6),DUMX
LA     15,1(15)
ST     15,DUN7
MVC    0(2,14),DUN7+2
B      FA&SYSNDX
CA&SYSNDX DC F'0'
DA&SYSNDX DS F
EA&SYSNDX DS 16F
FA&SYSNDX LM 0,15,EA&SYSNDX
MEND

```

```

*INITIALIZE CII COUNT
*ZERO OUT DUMX+4
*THROW CII CONNECTORS
*ADD ASTRISK LENGTH
*MOVE ASTRISK TO TAIL
*PUT ARRAY LENGTH INTO LEADING 2 BYTES
*INCREMENT LABEL LENGTH
*MOVE IN NEW LABEL LENGTH
*QUIT
*HOLDS # ITEMS TO GENERATE
*HOLDS THE PROTECTIVE ADDRESS

```

*****JBLISTI*****

JIMP1

*****JIMP1*****

```

MACRO
&NUPAKR JIMP1 &RADD
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---JIMP1 IS THE MACRO FOR THE JIM PERRY ALPHANUMERIC COMPRESSOR.
*---&RADD IS THE RETURN ADDRESS.
*

```

```

&NUPAKR SOSCODE AA&SYSNDX
AA&SYSNDX L 0,SOS(1)
ST 0,NDR
ENDSS &NUPAKR
STRINGA BA&SYSNDX
BA&SYSNDX NOPR 1
MEND

```

*****JIMP1*****

JLITEM

```

*****JLITEM *****
      MACRO
&J      JLITEM &JBLIST,&MVALUE,&JVALUE,&NUMDIAG
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---JLITEM IS THE PARENT MACRO FOR JIM PERRY'S JOBLIST GENERATOR.
*---&JBLIST IS THE NAME OF THE JOBLIST ARRAY (JBLIST).
*---&MVALUE IS THE VALUE OF M.
*---&JVALUE IS THE LOCATION OF THE MAXIMUM VALE OF J.
*---&NUMDIAG IS THE NUMBER OF DIAGONALS TO BE THROWN.
*-----
&J      STM      0,15,BA&SYSNDX      *SAVE ALL REGISTERS
      LA      2,&JBLIST
      L      0,AJBWORK      *LOAD 0 WITH THE ADDRESS OF JBWORK
      S      0,=F'256'
      CR      0,2      *IF &JBLIST IS LONG ENOUGH--GO AHEAD
      BNL     AA&SYSNDX
      MVC     WORKA(132),=C'1*****X
      *JOB LIST ARRAY MUST BE AT LEAST 256 BYTES LONG*****X
      *****'
      PUT     PRINT,WORKA      *PRINT A NASTY NOTE
      B      CL1      *QUIT
AA&SYSNDX LA      3,&MVALUE      *R3=M
      LA      4,&JVALUE      *R4= MAXIMUM VALUE OF J
      LA      5,&NUMDIAG      *R5 = NUMBER OF DIAGONALS
      TRANSFER 2,GENITEM,CA&SYSNDX *MONTE CARLO ITEM GENERATOR.
BA&SYSNDX DS      16F      *REGISTER SAVE AREA
CA&SYSNDX LM      0,15,BA&SYSNDX *RESTORE REGISTERS
      MEND
*****JLITEM *****

```

JUNK

```

*****JUNK *****
      MACRO
      JUNK &TPCORD,&LJBLIST
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---JUNK IS THE PARENT RESERVE MACRO FOR THE SOLID SYSTEM.
*---&TPCORD IS THE MAXIMUM NUMBER OF PERMANENT CORDS TO BE USED.
*---&LJBLIST IS THE LENGTH OF THE JBLIST AND JBWORK ARRAYS.
*-----
      DS      D      *FOR MISSING RESERVE ITEMS.
      DEVICES
      SAVEAREA
      CONSTANT
      STORAGE &TPCORD,&LJBLIST
      MODADX
      INOUT
SPRINGER SKIPP 1
      MEND
*****JUNK *****

```


KERTHROW

*****KERTHROW*****

MACRO

&J KERTHROW &JBLIST

*XX

*---KERTHROW GENERATES THE PRINCIPAL DIAGONAL FOR A SINGLE JOBLIST ITEM

*---&JBLIST IS THE ADDRESS OF THE BEGINNING OF CURRENT JOBLIST ITEM.

*

```

&J      STM      5,15,EA&SYSNDX
        L        5,NOV
        LR       6,5
        M        4,KLENGTH
        LA       5,2(5)
        ST      5,CA&SYSNDX
        LA       2,&JBLIST
        LA       2,4(2)
        A        2,MVALUE
        LR       0,2
        MVC     0(2,2),CA&SYSNDX.+2
        LA       2,2(2)
AA&SYSNDX L        4,KLENGTH
BA&SYSNDX DICE    =F'8',DA&SYSNDX,GENERATE
        CLI     DA&SYSNDX.+3,X'00'
        BE     BA&SYSNDX
        MVC     0(1,2),DA&SYSNDX.+3
        LA       2,1(2)
        BCT    4,BA&SYSNDX
        BCT    6,AA&SYSNDX
        L       3,NOV
        BCTR   3,0
        ST     3,DUMX+4
        L      4,NUMDIAG
        BCTR   4,C
        ST     4,DUMX
        LA     1,4
        A      1,MVALUE
        AR     1,5
        LR     4,5
        B      FA&SYSNDX
CA&SYSNDX DS      F
DA&SYSNDX DS      F
EA&SYSNDX DS     11F
FA&SYSNDX LM     5,15,EA&SYSNDX
        MEND

```

*****KERTHROW*****

LBLTHROW

*****LBLTHROW*****

MACRO

&J LBLTHROW &JAYBITS,&BITMAP

*XX

*---LBLTHROW THROWS A LABEL ITEM FOR THE JOBLIST GENERATOR.

*---&JAYBITS IS THE NUMBER OF LABEL ITEMS TO BE THROWN.

*---&BITMAP IS THE BEGINNING ADDRESS OF THE BITMAP.

*

```

&J      STM      0,15,HA&SYSNDX          *SAVE REGISTERS
        LA       2,&BITMAP
        XC      GA&SYSNDX.(4),GA&SYSNDX
        MVC     GA&SYSNDX.+2(2),0(2)
        L       3,GA&SYSNDX
        AR      3,2                      *R3 POINTS TO BEGINNING OF LABEL
        LA      2,2(2)                   *R2 POINTS TO BITMAP
        L       1,&JAYBITS               *R1 CONTAINS THE # OF LABEL ITEMS
        LR      5,1
        M       4,KLENGTH
        LA      5,2(5)                   *ADD 2 TO LENGTH OF LABEL
        ST      5,GA&SYSNDX              *R5 CONTAINS LENGTH OF LABELS+2
        MVC     0(2,3),GA&SYSNDX.+2     *MOVE LENGTH INTO HEAD OF LABEL
        LA      3,2(3)                   *SET UP ADDRESS FOR FIRST BYTE
        SR      4,4                       *ZERO MASK
AA&SYSNDX IC      4,=X'80'               *INITIALIZE MASK
BA&SYSNDX LA      5,1                   *SET UP FOR BCT INSTRUCTION
        L       6,KLENGTH                 *FOR ADDRESS INCREMENT
        EX      4,FA&SYSNDX              *TEST UNDER MASK
        BZ      DA&SYSNDX                 *IF 0 SKIP THE BYTE
        L       5,KLENGTH                 *SET UP FOR BCT INSTRUCTION
        LA      6,1                       *FOR ADDRESS INCREMENT
CA&SYSNDX DICE    =F'8',GA&SYSNDX,GENERATE *GENERATE RANDOM BYTE INFO.
        CLI    GA&SYSNDX.+3,X'00'
        BE     CA&SYSNDX                  *IF LABEL IS 0 , THEN THROW AGAIN
        MVC    0(1,3),GA&SYSNDX.+3       *STORE "THROW" INTO BYTE
DA&SYSNDX LA      3,0(6,3)               *INCREMENT LABEL ADDRESS
        BCT    5,CA&SYSNDX                *THROW 'KLENGTH' BYTES FOR SINGLE ITEM
        SRL    4,1                         *CHANGE MASK
        LTR    4,4                         *IS MASK ZERO?
        BE     EA&SYSNDX                   *BRANCH IF MASK IS ZERO
        BCT    1,BA&SYSNDX                 *THROW AGAIN IF NECESSARY
EA&SYSNDX BCTR    1,0                      *IF MASK=0 DECREMENT COUNT
        BNH    IA&SYSNDX                   *IF COUNT IS ZERO , QUIT
        LA     2,1(2)                       *INCREMENT BITMAP ADDRESS
        B      AA&SYSNDX                   *THROW AGAIN
FA&SYSNDX TM      0(2),X'00'              *INSTRUCTION USED BY "EX"
GA&SYSNDX DC      F'0'                    *HOLDS RANDOM BYTE RIGHT JUST.
HA&SYSNDX DC      16F'0'                  *REGISTER SAVE AREA
IA&SYSNDX LM      0,15,HA&SYSNDX         *RESTORE REGISTERS
        MEND

```

*****LBLTHROW*****

LEX

```

*****LEX*****
      MACRO
&J      LEX      &JII,&CODE,&CORD1
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---LEX ASSEMBLES A LEXICCN AND PUTS IT AT HEAD OF STRING.
*---&JII IS THE LENGTH OF THE ENTIRE STRING.
*---&CODE IS THE CODE-WORD TO BE SUBSTITUTED FOR THE CORD.
*---&CORD1 IS THE CORD THAT WILL BE REPLACED BY THE &CODE.
*-----
&J      STM      BRYY,BRY,TBRYY          *SAVE BRYY AND BRY
      L          BRYY,SBRYY
      L          2,&JII
      AR         BRYY,2
      BCTR       BRYY,0
      LR         BRY,BRYY
      A          BRY,R
      RMVC       1,2,BRY,C,BRYY
      MVC        AA&SYSNDX,+1(1),RM+3
      L          BRYY,SBRYY
      MVC        0(1,BRYY),&CODE
AA&SYSNDX MVC     1(0,BRYY),&CORD1
      A          2,R
      LA         2,1(0,2)
      ST         2,&JII
      LM         BRYY,BRY,TBRYY          *RESTORE BRYY AND BRY
      MEND
*****LEX*****

```

LEXD

```

*****
MACRO
&J      LEXD  &JII,&CODE,&CORD1
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---LEXD TAKES LEXICON FROM HEAD OF STRING AND DECOMPOSES IT.
*---&JII IS THE LENGTH OF THE ENTIRE STRING.
*---&CODE IS THE CODE-WORD USED.
*---&CORD1 IS THE CORD THAT IS TO REPLACE THE CODE-WORD.
*-----
&J      STM   BRYY,BRY,TBRYY          *SAVE BRYY AND BRY
        L     BRYY,SBRYY
        LR    BRY,BRYY
        MVC   &CODE-1(1),0(BRYY)     *CODE-WORD STORED IN &CODE.
        L     1,RM
AA&SYSNDX MVC   &CORD1.(0),1(BRYY)    *CORD STORED IN &CORD1.
        LA    1,2(0,1)
        L     0,&JII
        SR    0,1                    *NUMBER BYTES IN NEW STRING.
        A     BRYY,R
        ST    0,&JII                  *NEW JII CALCULATED.
BA&SYSNDX C     0,=F'256'
        BNL   CA&SYSNDX
        BCTR  0,0
CA&SYSNDX STC   0,CA&SYSNDX.+1
        MVC   0(C,BRY),1(BRYY)
        LA    BRYY,256(0,BRYY)
        LA    BRY,256(0,BRY)
        S     0,=F'256'
        BH    BA&SYSNDX
        LM    BRYY,BRY,TBRYY        *RESTORE BRYY AND BRY
        MEND
*****
LEXD *****

```

LINKHOLE

```

*****LINKHOLE*****
MACRO
&J      LINKHOLE &EPLNGTH,&FEMPTY
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---LINKHOLE WILL SEARCH THE LINKED HOLES OF MEM.BLK. FOR STORAGE.
*---&EPLNGTH CONTAINS THE LENGTH OF THE EXECUTIVE POINTER.
*---&FEMPTY IS THE BEGINNING OF THE FIRST CHAINED HOLE.
*-----
&J      NOPR  1
***CHECK WHETHER OR NOT THERE ARE ANY EMPTY HOLES IN THIS MEMORY BLOCK
***THAT WOULD BE AVAILABLE FOR THE SUB-ARRAY THAT IS TO BE CREATED.
***THE BEGINNING ADDRESS OF THE FIRST CHAINED HOLE WILL FOLLOW THE
***M-J ARRAYS.SET A FALSE FLOOR OF THREE BYTES TO THE MEMORY BLOCK
***STARTING ADDRESS. IR5,RO AND R1 WILL BE MODIFIED IN THIS MACRO.
***THE EP POINTER(IR5) SHOULD POINT TO THE FIRST EXECUTIVE POINTER.
        MEND
*****LINKHOLE*****

```


MAS

```

*****MAS*****
MACRO
&J      MAS  &JII,&NBB,&CORD1
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---MAS MOVES THE STRING SO THAT A CORD CAN BE INSERTED.
*---&JII IS THE LENGTH OF THE ENTIRE STRING.
*---&NBB IS THE NUMBER OF BYTES FROM THE BEGINNING OF STRING.
*---&CORD1 IS THE ADDRESS OF THE FIRST BYTE IN THE CORD.
*-----
&J      STM   BRYY,BRY,TBRYY          *SAVE BRY AND BRY
        LR    1,&NBB                  *GR1= NO. OF BYTES FROM BEGINNING
        L     BRYY,SBRYY
        L     2,&JII
        LR    0,2
        SR    2,1
        A     BRYY,&JII
        BCTR  BRYY,0
        LR    BRY,BRY
        CLI   RM+3,X'00'
        BE    AA&SYSNDX
        A     BRY,R
        B     BA&SYSNDX
AA&SYSNDX A     BRY,R
        LA    2,1(0,2)
BA&SYSNDX RMVC  0,2,BRY,0,BRY
        L     BRYY,SBRYY
        AR    BRYY,&NBB
CA&SYSNDX MVC   CA&SYSNDX,+1(1),RM+3
        MVC  0(0,BRYY),&CORD1
        A     0,R
        CLI   RM+3,X'00'
        BE    DA&SYSNDX
        BCTR  0,0
DA&SYSNDX ST    0,&JII                *JII=LENGTH OF EXPANDED STRING.
        LM    BRYY,BRY,TBRYY        *RESTORE BRY AND BRY
        MEND
*****MAS*****

```

MJARRAY

```

*****MJARRAY*****
MACRO
&J MJARRAY &ADDL
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*--MJARRAY INITIALIZES THE M-J ARRAYS AT THE START OF EACH NEW FILE.
*--&ADDL IS THE NUMBER OF BYTES IN THE COMPOSITE ADDRESSES.
*-----
&J SR 0,0
SR 2,2
L 3,=F'118' *SET BY ARRAY LENGTHS.
M 2,=F'&ADDL'
A 3,=F'324'
L 1,AYY
ST 3,DUM1 *R3 AND DUM1 CONTAIN LENGTH OF M-J ARRAYS.
D 0,=F'8'
C 0,ZERO
BE AA&SYSNDX
S 0,=F'8'
LPR 0,0
A 0,AYY
ST 0,AYY *AYY CONTAINS DOUBLE WORD ADDRESS.
AA&SYSNDX L 5,AYY *R5=EMPTY ADDRESS.
LARGXC DUM1+2,0(5),0(5) *ZERO M-J ARRAY STORAGE AREA.
SR 0,0 *DEVICE TYPE=0(DISK).
SR 1,1 *DEVICE NUMBER=0.
ASADD 0(5),0,1,1,1,3 *EMPTY ADDRESS LOADED.
L 0,ZERO *DEVICE TYPE=0(TAPE).
A 5,=F'&ADDL' *R5=BULK ADDRESS.
ASADD 0(5),0,1,1,1,1 *BULK ADDRESS LOADED.
A 5,=F'&ADDL'
L 1,=F'10' *NUMBER OF EXECUTIVE POINTERS IN M ARRAY.
M 0,=F'&ADDL'
A 1,=F'4'
ST 1,DUM1
MVC 0(4,5),DUM1 *M ARRAY LENGTH LOADED
LR 2,5
A 2,=F'4'
AR 5,1
L 1,=F'20' *NUMBER OF EXECUTIVE POINTERS IN J ARRAYS.
M 0,=F'&ADDL'
A 1,=F'24'
L 4,=F'5'
BA&SYSNDX ST 1,DUM1
A 5,=F'&ADDL'
MVC 0(4,5),DUM1 *THE FIVE J ARRAYS CREATED.
LR 7,5
S 7,AYY
ASADD 0(2),0,0,0,0,7
A 2,=F'&ADDL.'

```

MJARRAY (CONT.)

```
AR      5,1
A       1,=F'20'
BCT     4,BA&SYSNDX
A       5,=F'&ADDL.'
MVC     0(&ADDL.,5),CONM          *CURRENT ADDRESS LOADED.
LR      4,3
LR      6,3
L       2,AYY
L       7,AMEMORY
ST      7,MEMADD
A       7,=F'4096'
ST      7,MEMADD+4
MVI     MSIGNAL,X'00'
MVC     EMPTY(&ADDL),EMPTY+&ADDL
MEND
```

*****MJARRAY*****

MJTHROW

```

*****MJTHROW*****
MACRO
&J MJTHROW &JBLIST,&M,&JAY
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---MJTHROW IS THE MACRO THAT THROWS M AND J INTO THE JOBLIST ITEM.
*---&JBLIST IS THE NAME OF THE JOBLIST ARRAY (JBLIST).
*---&M IS THE VALUE OF M.
*---&JAY IS THE MAXIMUM VALUE OF J.
*-----
&J STM 0,15,DA&SYSNDX *SAVE REGISTERS
SR 5,5 *ZERO R5
L 6,&M *PUT "M" INTO R6
L 7,&JAY *PUT "J" INTO R7
LA 2,&JBLIST *PUT ADDRESS OF JOBLIST INTO R2
C 6,=F'0'
BNH AA&SYSNDX *IF M=0 WRITE NASTY NOTE
C 7,=F'1'
BNH BA&SYSNDX *IF J=1 WRITE NASTY NOTE
B BA&SYSNDX
AA&SYSNDX MVI WORKA,C' '
MVC WORKA+1(131),WORKA
MVC WORKA(20),=CL20'1 INVALID M AND/OR J'
PUT PRINT,WORKA
B CL1 *QUIT
BA&SYSNDX MVC DUMX(4),&M *MOVE "M" INTO TEMP STORAGE
MVC 2(2,2),DUMX+2 *PUT "M" INTO BYTES 3 & 4
LA 2,4(2) *POINT TO BEGINNING OF J
CA&SYSNDX MVC EA&SYSNDX.(4),&JAY *MOVE "J" INTO TEMP STORAGE
FACE EA&SYSNDX,EA&SYSNDX,GENERATE
***GET INTEGER FROM 0 TO J-1
L 7,EA&SYSNDX
LA 7,1(7)
ST 7,EA&SYSNDX *INCREMENT SO ZERO CANNOT OCCUR
MVC 0(1,2),EA&SYSNDX.+3 *PUT J(I) INTO JOBLIST
AR 5,7 *ADD CURRENT J(I)
LA 2,1(2) *CLICK UP POINTER
BCT 6,CA&SYSNDX *GENERATE M NUMBER OF J(I)'S
ST 5,NOV *MOVE THE SUM TOTAL OF THE J(I) INTO ANY DUMMY
B FA&SYSNDX
DA&SYSNDX DC 16F'0' *REGISTER SAVE AREA
EA&SYSNDX DC F'0' *TEMPORARY STORAGE
FA&SYSNDX LM 0,15,DA&SYSNDX *RESTORE REGISTERS
MEND
*****MJTHROW*****

```

MMATCH

```

*****MMATCH *****
MACRO
&J      MMATCH &NOVER1,&NOVER2,&NOVER3,&CURSCRN,&JBLIST,&JBWORK,&KLEX
        NGTH
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---MMATCH IS THE MISMATCH MACRO FOR THE SSEARCH COMPONENT.
*---&NOVER1 CONTAINS THE NUMBER OF TYPE I OVER-RIDE.
*---&NOVER2 CONTAINS THE NUMBER OF TYPE II OVER-RIDES.
*---&NOVER3 CONTAINS THE NUMBER OF TYPE III OVER-RIDES.
*---&CURSCRN IS THE ADDRESS(IN JBLIST) OF THE CURRENT SCREEN(DIAGONAL).
*---&JBLIST IS THE ADDRESS OF THE CURRENT JOBLIST ITEM.
*---JBWORK IS THE WORK ARRAY FOR USE IN SMATCH AND SMOBILE.
*---&KLENGTH CONTAINS THE NO. OF BYTES IN EACH KERNEL OF THE I.R.
*-----
&J      MVI      SRGATE,X'02'
***SRGATE(SEARCH GATE) SETTINGS:FAILED(0);NEW PLAY(1);SUCCESS(2).
        CLI      MODE+3,X'01'
        BNE      AA&SYSNDX          *NOT STORAGE.
***A NEW SUB-PATH IS BEING CREATED FOR NEW INFORMATION.
        TM      MSIGNAL,X'01'
        BD      CA&SYSNDX          *WRITE THEN READ.
        OI      MSIGNAL,X'01'      *BIT TURNED ON.
        B       CA&SYSNDX          *CONTINUE SEARCH.
AA&SYSNDX XC      JII(4),JII
        MVI      SRGATE,X'00'      *SRGATE SET.
        TM      MSIGNAL,X'20'
        BD      CA&SYSNDX          *CONTINUANCE.
        CLI      &NOVER1+3,X'00'
        BNE      BA&SYSNDX          *TYPE I.
        CLI      &NOVER2+3,X'00'
        BNE      BA&SYSNDX          *TYPE II.
        CLI      &NOVER3+3,X'00'
        BE       CA&SYSNDX          *NO OVER-RIDES.
***TYPE III OVER-RIDE WAS PRESENT.
BA&SYSNDX STRATEGY &CURSCRN,&JBLIST,&JBWORK,&KLENGTH
***STRATEGY CALLS THE SMATCH AND SMOBILE COMPONENTS(LEVEL=3).
CA&SYSNDX CLI      SRGATE,X'01'
        BL      FINISHED          *FINISH.
        BE      NEWPLAY          *STRATEGIC MOVE.
        MEND
*****MMATCH *****

```

MODADI

```

*****MODADI *****
MACRO
MODADI
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---MODADI DEFINES THE ADDRESSES FOR INTERNAL USE IN SOLID.
*-----
A ACTION      DC      V(ACTION) *S ACTION IS FOR ERROR CORRECTING PROCEDURES.
A ANPAK      DC      V(ANPAK)      *FIRST ADDRESS OF SANPAK.
A ANPAKCC    DC      V(ANPAKCC)     *SECOND ADDRESS OF SANPAK.
A ANPAKD     DC      V(ANPAKD)     *FIRST ADDRESS OF SANPAKD AND SANPAKJ.
A ANPAKDC    DC      V(ANPAKDC)    *SECOND ADDRESS OF SANPAKJ.
A ANSWER     DC      V(ANSWER)     *GO TO THE DISPENSE MACRO.
A CARDREAD   DC      V(CARDREAD)   *READ A CARD.
A COMANDD    DC      V(COMANDD)    *FIRST ADDRESS OF THE SCOMMAND COMPONENT.
A COMANDS    DC      V(COMANDS)    *SECOND ADDRESS OF THE SCOMMAND COMPONENT.
A CYCLIC     DC      V(CYCLIC)
A GENITEM    DC      V(GENITEM)     *ADDRESS OF ITEM GENERATOR.
A JOBLIST    DC      V(JOBLIST)    *ADDRESS OF THE SJOBLIST COMPONENT.
A LOOKFILE   DC      V(LOOKFILE)   *BEGIN A NEW SEARCH.
A MATCH      DC      V(MATCH)
A MEMORY     DC      V(MEMORY)     *FIRST ADDRESS OF THE SMEMORY COMPONENT.
A MEMORYR    DC      V(MEMORYR)
A MOBILE     DC      V(MOBILE)
A NEWJOB     DC      V(NEWJOB)     *START A NEW JOB.
A NORMAL     DC      V(NORMAL)     *FOR NORMALIZATION.
A NUPAK      DC      V(NUPAK)     *FIRST ADDRESS OF THE SNUPAK COMPONENT.
A NUPAKCN    DC      V(NUPAKCN)    *SECOND ADDRESS OF THE SNUPAK COMPONENT.
A NUPAKR     DC      V(NUPAKR)    *SECOND ADDRESS OF THE SNUPAB COMPONENT.
A OPEN       DC      V(OPEN)       *FOR OPENING DEVICES.
A OUTPUT     DC      V(OUTPUT)     *FIRST ADDRESS OF THE SOUTPUT COMPONENT.
A OUTPUTC    DC      V(OUTPUTC)    *SECOND ADDRESS OF THE SOUTPUT COMPONENT.
A PMARRAY    DC      V(PMARRAY)    *ENTRY IN SMEMORY TO SAVE F-M.
A PRINT      DC      V(PRINT)     *FIRST ADDRESS OF THE SPRINT COMPONENT.
A PUNXH      DC      V(PUNXH)     *FIRST ADDRESS OF THE SPUNXH COMPONENT.
A READC      DC      V(READC)     *FIRST ADDRESS OF THE SREADC COMPONENT.
A READCON    DC      V(READCON)    *SECOND ADDRESS OF THE SREADC COMPONENT.
A READT      DC      V(READT)     *FIRST ADDRESS OF THE SREADT COMPONENT.
A REFLECT    DC      V(REFLECT)
A REID       DC      V(REID)       *FIRST ADDRESS OF THE SREID COMPONENT.
A RESULT     DC      V(RESULT)     *ADDRESS OF THE SRESULT COMPONENT.
A SAVEFM     DC      V(SAVEFM)     *THE SAVE FAST MEMORY ADDRESS.
A SEARCH     DC      V(SEARCH)     *FIRST ADDRESS OF THE SSEARCH COMPONENT.
A SHUT       DC      V(SHUT)       *FOR CLOSING DEVICES.
A STATECL    DC      V(STATECL)    *THE INITIALIZING ROUTINE..
A TLATOR1    DC      V(TLATOR1)    *FIRST TRANSLATOR.
A TLATOR2    DC      V(TLATOR2)    *SECOND TRANSLATOR.
A TLATOR3    DC      V(TLATOR3)    *THIRD TRANSLATOR.
A TLATOR4    DC      V(TLATOR4)    *FOURTH TRANSLATOR.
A TLATOR5    DC      V(TLATOR5)    *FIFTH TRANSLATOR.

```


NUPAKC

*****NUPAKC *****

MACRO

&J NUPAKC &RADD

*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXNUPAKC XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

*---NUPAKC IS THE NUMERIC COMPRESSOR PART OF COPAK.

*---&RADD IS THE RETURN ADDRESS.

*

```

&J      L      0, CONM
        SRL     0, 5
        L      5, ZERO
        ST     0, WDC
        LM     0, 3, CS1
        L     BRYY, CS15
        LE     0, LSX(1)
        CE     0, =E'0.0'
        BNL   CA&SYSNDX
AA&SYSNDX LE     0, =E'-1.0'
        STE    0, BWX(1)
BA&SYSNDX L      4, 0(BRYY)
        IC     5, =X'FF'
        NR     5, 4
        ALR    4, 5
        SRL    4, 8
        ST     4, 0(BRYY)
        A     BRYY, =F'4'
        S     3, =F'4'
        BH    BA&SYSNDX
        S     BRYY, =F'4'
        L     4, 0(BRYY)
        SLL   4, 8
        ST     4, CHECK(1)
CA&SYSNDX B     DA&SYSNDX
        BH    LA&SYSNDX
        LE     0, =E'0.0'
        STE    0, BWX(1)
DA&SYSNDX L      4, =F'1'
        SLL   4, 30
        LNR    5, 4
        L     BRYY, CS15
        L     3, CS4
EA&SYSNDX C     4, 0(BRYY)
        BNH   FA&SYSNDX
        L     4, 0(BRYY)
FA&SYSNDX C     5, 0(BRYY)
        BNL   GA&SYSNDX
        L     5, 0(BRYY)
GA&SYSNDX A     BRYY, =F'4'
        S     3, =F'4'
        BH    EA&SYSNDX

```

*TO COMPUTE FIXED PT. MINIMUM.
*LSX > 0.0

*JI RELOADED.

NUPAKC (CONT.)

	SR	5,4	
	SRL	5,1	
	AR	4,5	
	LPR	5,5	
	C	5,WDCD	
	BNH	JA&SYSNDX	
	LM	0,3,CS1	
	L	BRYY,CS15	
	L	0,BWX(1)	
	C	0,=F'0'	
	BNL	IA&SYSNDX	
HA&SYSNDX	L	4,C(BRYY)	
	SLL	4,8	
	ST	4,0(BRYY)	
	A	BRYY,=F'4'	
	S	3,=F'4'	
	BH	HA&SYSNDX	
IA&SYSNDX	L	5,=F'1'	
	SLL	5,31	
	AL	5,SOS(1)	
	STC	5,NDR+3	
	SRA	5,8	
	ST	5,JI	
	MVC	WORKA(133),BLANK	
	MVC	WORKA(42),=C'	COMPRESSION VIA NUPAK UNSUCCESSFUL. X
	PUT	PRINT,WORKA	
	B	ERADD	
JA&SYSNDX	ST	4,MY(1)	
	L	BRYY,CS15	*BASE YY RELOADED.
	L	3,CS4	*JI RELOADED.
KA&SYSNDX	L	5,C(BRYY)	
	SR	5,4	
	ST	5,0(BRYY)	*FIXED PT. DIFF. STORED IN YY.
	A	BRYY,=F'4'	
	S	3,=F'4'	
	BH	KA&SYSNDX	
	B	QA&SYSNDX	*TO COMPUTE NDR.
LA&SYSNDX	DE	0,=E'2.0'	
	CE	0,=E'0.0'	
	BE	AA&SYSNDX	*SAFETY FACTOR FOR BWX(1)=0.0.
	STE	0,BWX(1)	
	LE	0,=E'-1.0E+64'	*0--FOR MAXIMUM YY.
	LPER	2,0	*2-- FOR MINIMUM YY.
MA&SYSNDX	CE	2,0(BRYY)	
	BNH	NA&SYSNDX	
	LE	2,C(BRYY)	
NA&SYSNDX	CE	0,0(BRYY)	
	BNL	OA&SYSNDX	

NUPAKC (CONT.)

```

OA&SYSNDX  LE    0,0(BRYY)
            A    BRYY,=F'4'
            S    3,=F'4'
            BH   MA&SYSNDX
            SER  0,2
            DE   0,=E'2.0'
            AER  2,0
            STE  2,MY(1)
            DE   0,BWX(1)
            CE   0,=E'67108864.0'
OVERFLOW   BH   AA&SYSNDX      *SAFETY FACTOR IF TRUN. IS TOO LARGE.
***BEGIN THE TRUNCATION PROCESS NEXT.
            L    BRYY,CS15
            L    3,CS4
            LR   7,1
PA&SYSNDX  LE    0,0(BRYY)
            SER  0,2
            DE   0,BWX(7)
            STE  0,0(BRYY)
            TRUNC 0(BRYY),0(BRYY)
            A    BRYY,=F'4'
            S    3,=F'4'
            BH   PA&SYSNDX
            S    BRYY,=F'4'
            L    0,0(BRYY)
            A    BRYY,CS3
            A    BRYY,=F'4'
            ST   0,0(BRYY)
            CONVE 0(BRYY),0(BRYY)
            LE   0,0(BRYY)
            ME   0,BWX(7)
            AE   0,MY(7)
            STE  0,CHECK(7)
            LE   0,BWX(7)
            DE   0,=E'2.0'
            AE   0,CHECK(7)
            STE  0,CHECK(7)
***NEXT BEGINS THE CALCULATION OF NDR.
QA&SYSNDX  L    BRYY,CS15
            LM   0,3,CS1
            LM   0,1,ZERO
            LM   4,7,ZERO
RA&SYSNDX  L    7,0(BRYY)
            CR   7,0
            BNE  SA&SYSNDX
            A    5,=F'1'
            B    TA&SYSNDX
SA&SYSNDX  LPR  0,0
            AR   0,5

```

*7-- CONTAINS RM.

*TERMS NOT EQUAL.

NUPAKC (CONT.)

	AR	4,0	
	BO	UA&SYSNDX	*OVERFLOW;SET IR1=3FFFFFFF.
	LR	0,7	
	L	5,ZERO	
TA&SYSNDX	A	BRYY,=F'4'	
	S	3,=F'4'	
	BH	RA&SYSNDX	
	LPR	0,0	
	AR	0,5	
	BO	UA&SYSNDX	*OVERFLOW.
	AR	4,0	
	LR	1,4	*FIRST ABSOLUTE SUM OF ARRAY YY.
	B	VA&SYSNDX	
UA&SYSNDX	L	1,CONM	*CONM=FFFFFFFFFFFFFFFF.
	SRL	1,1	
VA&SYSNDX	L	BRYY,CS15	*BEGIN SUMMING ABSOLUTE DIFFERENCES.
	L	3,CS4	
	LM	4,7,ZERO	
	L	0,C(BRYY)	
	L	6,C(BRYY)	
WA&SYSNDX	A	BRYY,=F'4'	
	S	3,=F'4'	
	BNH	YA&SYSNDX	*SUMMING FINISHED.
	L	7,0(BRYY)	
	SR	7,6	
	LPR	BRYY,7	
	C	BRYY,WCDC	
	BNL	BB&SYSNDX	
	L	6,0(BRYY)	
	CR	7,0	
	BNE	XA&SYSNDX	*NOT EQUAL DIFFERENCES.
	A	5,=F'1'	
	B	WA&SYSNDX	
XA&SYSNDX	LPR	0,0	
	AR	0,5	
	AR	4,0	
	BO	BB&SYSNDX	*OVERFLOW.
	LR	0,7	
	L	5,ZERO	
	B	WA&SYSNDX	
YA&SYSNDX	LPR	0,0	*SUMMING FINISHED.
	AR	0,5	
	AR	4,0	
	BO	BB&SYSNDX	*OVERFLOW.
	CR	1,4	
	BNH	BB&SYSNDX	*QUIT(SAME AS OVERFLOW.
	L	BRYY,CS15	
	L	3,CS4	
	L	6,0(BRYY)	

NUPAKC (CONT.)

```

ZA&SYSNDX  A    BRYY,=F'4'
            S    3,=F'4'
            BNH  AB&SYSNDX
            L    7,0(BRYY)
            SR   7,6
            L    6,0(BRYY)
            ST   7,0(BRYY)
            B    ZA&SYSNDX
AB&SYSNDX  LR    1,4
            L    4,NDR
            A    4,=F'1'
            ST   4,NDR
            C    4,=F'14'
            BL   VA&SYSNDX
BB&SYSNDX  L    4,NDR
            A    4,=F'1'
            ST   4,NDR
            L    BRYY,CS15
            LM   0,3,CS1
            AL   4,SOS(1)
            ST   4,SOS(1)
*****
***NEXT BEGINS CONDENSATION OF STRINGS.
            L    7,CS4
            LM   0,6,ZERO
            A    4,=F'4'
            LR   2,BRYY
            LR   3,2
CB&SYSNDX  L    0,0(BRYY)
DB&SYSNDX  C    0,0(2)
            BNE  EB&SYSNDX
            A    1,=F'1'
            A    2,=F'4'
            S    7,=F'4'
            BH   DB&SYSNDX
EB&SYSNDX  C    1,=F'3'
            BH   GB&SYSNDX
FB&SYSNDX  ST   0,0(3)
            A    3,=F'4'
            A    5,=F'4'
            A    BRYY,=F'4'
            L    0,0(BRYY)
            BCT  1,FB&SYSNDX
            L    1,ZERO
            C    7,ZERO
            BH   DB&SYSNDX
            B    LB&SYSNDX
GB&SYSNDX  STM  6,7,DUM1
            LPR  7,0

```

*FINISHED DIFFERENCING.

*TRANSFERED SUM.

*LSX FOR STORING CIR6 IS WITH IR4=0.

*TERMS NOT EQUAL.

*TO CONDENSE STRING WITH MORE THAN 3.

*CONTINUE
*FINISHED.

NUPAKC (CONT.)

	L	6,ZERO	
	SLL	7,1	
	C	7,ZERO	
	BNE	HB&SYSNDX	
	A	7,=F'2'	
HB&SYSNDX	MR	6,1	
	C	7,=F'30'	
	BNL	IB&SYSNDX	
	DR	6,1	
	SLL	7,0(1)	
	CR	7,5	
	BNL	IB&SYSNDX	
	LM	6,7,DUM1	
	B	FB&SYSNDX	
IB&SYSNDX	LM	6,7,DUM1	
	ST	1,0(3)	*NO OF REPEATS STORED IN NEW YY.
	ST	5,TL(4)	*LOCATIONS OF REPEAT TERMS STORED.
	A	3,=F'4'	
	A	4,=F'4'	
	ST	0,0(3)	*COMMON TERM STORED IN YY.
	A	3,=F'4'	
	A	5,=F'8'	*NEXT LOCATION IN NEW YY.
	A	6,=F'1'	*6---CONDENSED STRINGS.
	C	6,=F'9'	
	BL	KB&SYSNDX	*STRING LIMIT NOT REACHED.
JB&SYSNDX	C	7,ZERO	
	BNH	LB&SYSNDX	*FINISHED.
	S	7,=F'4'	
	L	0,0(2)	
	ST	0,0(3)	
	A	3,=F'4'	
	A	2,=F'4'	
	B	JB&SYSNDX	
KB&SYSNDX	LR	BRY,2	
	L	1,ZERO	
	C	7,ZERO	
	BH	CB&SYSNDX	*CONTINUE
LB&SYSNDX	ST	6,TL	
	LR	1,4	*1----CONTAINS NO OF BYTES USED IN LSX
	S	1,=F'4'	*1----SIZE HOLE NEEDED IN YY.
	LR	BRY,3	
	S	3,CS15	*IR3 NOW CONTAINS NO OF BYTES IN YY.
	C	6,ZERO	
	BNH	MB&SYSNDX	*THERE ARE NO CONDENSED STRINGS.
	S	BRY,=F'1'	*TO LOCATE LAST TERM OF CONDENSED YY.
	LR	BRY,3	
	AR	BRY,1	
	RMVC	0,3,BRY,0,BRY	
	L	BRY,CS15	*BEGIN TRANSFER OF REPEAT ADDRESSES.

NUPAKC (CONT.)

```

      ST      1,DUM2          *LENGTH OF REPEATS ADDRESSES STRING.
      LMOVE  DUM2+2,0(BRYY),TL+4
MB&SYSNDX  AR      3,4
      ST      3,JI          *JI=NUMBER OF BYTES(ALL) TO COMPRESS.
      L      BRYY,CS15
***NEXT BEGINS THE ACTUAL COMPRESSION.
***** *****
      S      3,=F'4'          *TO ACCOUNT FOR LSX NOT IN STRING.
NB&SYSNDX  LM      4,5,ZERO
      L      6,=F'1'          *TO ALLOW FOR SIGN.
      L      7,ZERO
      L      4,0(BRYY)
      C      4,ZERO
      BNL    QB&SYSNDX
      LPR    4,4
      L      7,=F'1'
      C      4,ZERO
CB&SYSNDX  BNE    QB&SYSNDX          *NOT ZERO.
PB&SYSNDX  LR      4,7
      SRDL   4,1
      A      6,=F'1'
      ALR    5,6
      ST      5,0(BRYY)
      A      BRYY,=F'4'
      S      3,=F'4'
      BH     NB&SYSNDX
      B      RB&SYSNDX          *FINISHED.
QB&SYSNDX  SRDL   4,1
      A      6,=F'1'
      C      4,ZERO
      BH     QB&SYSNDX
      S      6,=F'1'
      B      PB&SYSNDX
RB&SYSNDX  L      BRYY,CS15
***NEXT BEGINS THE ACTUAL PACKING.
***** *****
      MVI    CORD1,C'1'
      ST      8,WCD          *SAVED REGISTER 8.
      MVC    CS5(4),=F'48'
      LR     BRY,BRYY
      LR     6,BRY
      L      7,JI
      S      7,=F'4'          *TO ALLOW FOR LSX.
      MVC    MASK2(4),=F'31'
      L      1,CONM
      SLL   1,5
      ST      1,MASK1
      L      1,TL
      L      8,ZERO          *FOR EXTRACTING SEGMENTS.
SB&SYSNDX  *NO OF CONDENSED STRINGS LOADED.
          *RETURN FROM STORING DW IN YY.

```

NUPAKC (CONT.)

	MVC	CS6(8),ZERO	
TB&SYSNDX	L	3,0(BRY)	
	N	3,MASK2	*NEXT SEGMENT.
	A	8,=F'1'	*EXT.NO OF BITS IN SEGMENT.
	C	3,CS6	*INC. NO OF SEGMENTS.
	BNH	UB&SYSNDX	
	MVC	CS7(4),CS6	*SET CS7=CS6.
	ST	3,CS6	*NEW CS6(LARGEST SEGMENT).
UB&SYSNDX	L	3,CS6	
	L	2,ZERO	
	MR	2,8	
	C	3,CS5	
	BH	VB&SYSNDX	*NO GOOD;DECREASE 8 THEN STACK.
	A	BRY,=F'4'	
	S	7,=F'4'	
	BH	TB&SYSNDX	
	B	WB&SYSNDX	*ALL TERMS PROCESSED.
VB&SYSNDX	S	8,=F'1'	
	MVC	CS6(4),CS7	
WB&SYSNDX	L	3,CS5	
	L	2,ZERO	
	DR	2,8	*IR3 CONTAINS THE NUMBER OF BITS/SEGMENT.
	ST	8,CS8	*CS8=NO OF SEGMENTS.
	ST	2,WCD+4	
XB&SYSNDX	L	2,0(BRY)	
	L	4,MASK2	
	NR	4,2	
	ST	4,CS13	
	LR	4,3	
	C	3,=F'32'	
	BNH	YB&SYSNDX	
	L	4,=F'32'	
YB&SYSNDX	S	4,CS13	
	L	5,0(BRY)	
	SLL	2,1	
	SRL	2,1	
	SRA	2,0(4)	
	C	5,ZERO	
	BNL	ZB&SYSNDX	
	L	5,=F'1'	
	SLL	5,31	
	ALR	2,5	
ZB&SYSNDX	L	4,=F'32'	
	SR	4,3	
	C	3,=F'32'	
	BNH	AC&SYSNDX	
	L	4,ZERO	
AC&SYSNDX	SRL	2,0(4)	
	SLDL	0,0(3)	

NUPAKC (CONT.)

```

ALR 1,2
A   BRY,=F'4'
BCT 8,XB&SYSNDX
L   2,WCD+4
SLDL 0,0(2)
L   4,CS8
SLL 4,27
ALR 0,4
LR  BRY,BRYY
CLI CORD1,C'1'
BE  BC&SYSNDX
MVC 0(8,6),MASK4
A   6,=F'8'
BC&SYSNDX STM 0,1,MASK4
MVI CORD1,X'00'
LM  0,1,ZERO
MVC CS5(4),=F'56'
C   7,ZERO
BH  SB&SYSNDX
MVC 0(8,6),MASK4
A   6,=F'8'
L   BRY,CS15
LR  BRYY,6
SR  6,BRY
L   5,CS3
A   BRY,CS4
C   5,ZERO
BNH DC&SYSNDX
CC&SYSNDX MVC 0(C,BRYY),0(BRY)
A   BRY,=F'1'
A   BRYY,=F'1'
DC&SYSNDX BCT 5,CC&SYSNDX
L   BRYY,CS15
LM  0,1,CS1
ST  6,CS4
ST  6,JI
SLL 6,8
L   7,SOS(1)
STC 7,NDR+3
IC  6,NDR+3
ST  6,SOS(1)
L   8,WCD
B   &RADD
MEND

```

*FIRST EMPTY IN ARRAY YY.
*NUMBER OF BYTES USED IN ARRAY YY(6).

*FIRST FILLED.

*NEW JI STORED.
*NEW JI STORED.

*****NUPAKC*****

NUPAKD (CONT.)

DA&SYSNDX	L	0,MASK2	
	NR	0,2	
	SRL	0,24	*REGO=NO OF CONDESED STRINGS.
	ST	0,TL	*TL=NO CONDENSED STRINGS.
	SLDL	2,8	
	C	4,=F'32'	
	BNE	EA&SYSNDX	
	SLDL	2,16	
EA&SYSNDX	C	0,ZERO	
	BE	HA&SYSNDX	*NEXT SEGMENT IS IN YY.
FA&SYSNDX	A	1,=F'4'	
	L	5,MASK1	
	NR	5,2	
	SLDL	4,1	
	SRL	5,0(6)	
	SRDL	4,1	
	ST	5,TL(1)	
	L	5,NS	
	S	5,=F'1'	
	BNE	GA&SYSNDX	
	C	0,=F'1'	
	BE	GA&SYSNDX	
	S	0,=F'1'	
	ST	0,CS7	
	B	LA&SYSNDX	
GA&SYSNDX	ST	5,NS	
	SLDL	2,0(4)	
	BCT	0,FA&SYSNDX	
	ST	0,CS7	
HA&SYSNDX	L	0,NS	
	L	1,ZERO	
	C	0,ZERO	
	BE	LA&SYSNDX	
IA&SYSNDX	L	5,MASK1	
	NR	5,2	
	SLDL	4,1	
	SRL	5,0(6)	
	SRDL	4,1	
	ST	5,DUM1(1)	
	A	1,=F'4'	
	C	1,=F'256'	
	BNE	KA&SYSNDX	
JA&SYSNDX	LR	5,7	*CONTINUE
	A	5,CS3	
	ST	5,CS11	
	AR	BRY,1	
	RMOVE	CS11+2,0(BRY),0(BRY)	
	LR	BRY,1	
	SR	BRY,1	

NUPAKD (CONT.)

```

ST      1,CS11
LMOVE  CS11+2,0(BRY),DUM1
LR      BRY,BRY
L       1,ZERO
C       0,ZERO
BE      MA&SYSNDX
KA&SYSNDX  SLDL  2,0(4)
BCT     0,IA&SYSNDX
LA&SYSNDX  MVC   CS5(4),=F'56'
C       7,ZERO
PH      AA&SYSNDX
B       JA&SYSNDX
MA&SYSNDX  S     BRY,CS15
ST      BRY,CS4
ST      BRY,JI
***THIS CONCLUDES SEPARATION OF THE TERMS.
***** *****
L       5,CS4                      *5-- NEW JI.
LM      1,2,ZERO
L       0,TL                        *REGO--NO OF CONDENSED STRINGS
NA&SYSNDX  C     0,ZERO                      *FINISHED.
BE      PA&SYSNDX
A       1,=F'4'
L       3,TL(1)                      *LOCATION OF NUMBER OF REPEATS.
AR      3,2
ST      3,TL(1)
LR      BRY,3
A       BRY,CS15
L       3,0(BRY)                      *3--NO OF REPEATS.
L       6,CS4
S       6,TL(1)
S       6,=F'8'
A       6,CS3                          *6--NO OF TERMS TO MOVE.
A       BRY,=F'4'
L       4,0(BRY)                      *REPEATED TERM.
A       BRY,=F'4'
ST      6,CS5
L       6,ZERO
L       7,=F'4'
MR      6,3
S       7,=F'8'
AR      2,7
AR      5,7
LR      BRY,7                          *NEW JI ADJUSTED.
AR      BRY,BRY                          *FIRST EMPTY BYTE.
RMOVE  CS5+2,C(BRY),0(BRY)
L       BRY,CS15
A       BRY,TL(1)
OA&SYSNDX  ST   4,0(BRY)

```

NUPAKD (CONT.)

A BRY,=F'4'
 BCT 3,QA&SYSNDX
 ST 5,CS4
 ST 5,JI
 BCT 0,NA&SYSNDX

***NEXT REVERSE DEPTH OF REPRESENTATION.

PA&SYSNDX L 0,NDR *0 -- LOADED WITH NDR.

L 3,JI
 L 5,ZERO
 L BRY,CS15

QA&SYSNDX L 5,0(BRY)
 C 5,ZERO
 BNL RA&SYSNDX
 SLL 5,1
 SRL 5,1
 LNR 5,5

RA&SYSNDX ST 5,0(BRY)
 A BRY,=F'4'
 S 3,=F'4'
 BH QA&SYSNDX

SA&SYSNDX L 1,JI
 S 1,=F'4'
 L BRYY,CS15
 L BRY,CS15
 S 0,=F'1'

TA&SYSNDX BE UA&SYSNDX *FINISHED.
 A BRYY,=F'4'
 L 2,0(BRYY)
 A 2,0(BRY)
 ST 2,0(BRYY)
 LR BRY,BRYY
 S 1,=F'4'
 BE SA&SYSNDX
 B TA&SYSNDX

***CONVERSION TO THE ORIGINAL NUMBER BEGINS NEXT.

UA&SYSNDX L 4,CS2 *RM IS LOADED INTO IR4.
 L BRYY,CS15 *BASE YY LOADED.

L 7,JI
 L 0,BWX(4)
 C 0,ZERO
 BH XA&SYSNDX

VA&SYSNDX L 2,0(BRYY)
 A 2,MY(4)
 C 0,ZERO
 BE WA&SYSNDX

WA&SYSNDX SLL 2,8 *FIXED POINT.
 ST 2,0(BRYY) *SHIFT TRUNCATION REVERSED.

NUPAKD (CONT.)

```

A      BRYY,=F'4'
S      7,=F'4'
BH     VA&SYSNDX
B      YA&SYSNDX
***NORMAL TRUNCATION IS REVERED NEXT.
XA&SYSNDX CONVE 0(BRYY),0(BRYY) *CONVERT FROM FIXED TO FL. PT.
LE     0,0(BRYY)
AE     0,=E'0.5'
ME     0,BWX(4)
AE     0,MYX(4)
STE    0,0(BRYY)
A      BRYY,=F'4'
S      7,=F'4'
BH     XA&SYSNDX
YA&SYSNDX L      BRYY,CS15
LM     0,3,CS1
L      4,SOS(1)
STC    4,NDR
L      4,JI
IC     4,NDR
ST     4,SOS(1)
B      &RADD
MEND
*****NUPAKD *****

```

*FINISHED.

OPCORDS

*****OPCORDS*****

MACRO

&J OPCORDS &RADD

*XXX

*---OPCORDS REORGANIZES THE PCORDS TABLE (EXECUTING &TPCORD).

*---&RADD IS THE RETURN ADDRESS.

&J CLI MODE+3,X'00'
 BE AA&SYSNDX
 MVC BLANK+15(39),=C'ENTERING COPAK COMPRESSOR.BEGIN TIME: X
 ,

 B BA&SYSNDX
 AA&SYSNDX MVC BLANK+15(39),=C'ENTERING COPAK DECOMPRESSOR.BEGIN TIMEX
 :'

BA&SYSNDX PUT PRINT,BLANK
 MVC BLANK+15(39),BLANK+60
 TIME BIN
 ST 0,SEGTIME *BEGIN SEGMENT TIME.
 CLI OUTPXT+3,X'02'
 BL OPCORDSN
 CLI MODE+3,X'00'
 BE OPCORDSN
 L 5,SBRY Y
 L 6,JII
 DECP C COMPRESS,JII,BYTES
 MVI DA&SYSNDX.+1,X'84'

CA&SYSNDX MVC WOKKA(133),BLANK
 C 6,=F'132'
 BNL DA&SYSNDX
 S 6,=F'1'

DA&SYSNDX STC 6,DA&SYSNDX.+1
 MVC WOKKA+1(132),0(5)
 PUT PRINT,WOKKA
 A 5,=F'132'
 S 6,=F'132'
 BH CA&SYSNDX
 TIME BIN
 ST 0,SEGTIME

OPCORDSN L 0,NOSEG
 S 0,=F'1'
 ST 0,NOSEG
 CLI GATE,X'00'
 BNE NA&SYSNDX
 CLI INPXT+3,X'00'
 BNE EA&SYSNDX
 MVI GATE,X'01'

EA&SYSNDX CLI SWITCH+3,X'02'
 BNE FA&SYSNDX
 MVI SWITCH+3,X'01'

OPCORDS (CONT.)

```

FA&SYSNDX  CLI  MODE+3,X'00'
            BE  NA&SYSNDX
            CLI  TPCORD+3,X'00'
            BE  GA&SYSNDX
            CLI  LEXMODE+3,X'00'
            BE  NA&SYSNDX
            CLI  LEXCON+3,X'00'
            BNE  NA&SYSNDX
            CLI  LEXMODE+3,X'02'
            BE  NA&SYSNDX
GA&SYSNDX  L    1,=F'100'
            LA  2,PCORDS
HA&SYSNDX  MVC  0(40,2),ZERO
            A   2,=F'40'
            BCT 1,HA&SYSNDX
            MVC 0(24,2),ZERO
            MVC  TPCORD(4),ZERO
            CLI  LEXMODE+3,X'00'
            BNE  NA&SYSNDX
            L    1,=F'6'
            ST  1,TPCORD
            LE  0,=E'0.01'
            LA  3,PCORDS+4000
            ST  3,PCORDS+12
            L    3,=F'20'
IA&SYSNDX  A    3,=F'16'
            STE 0,PCORDS(3)
            A    3,=F'4'
            BCT 1,IA&SYSNDX
            MVI PCORDS,X'02'
            MVI PCORDS+1,X'02'
            MVI PCORDS+2,X'02'
            LA  2,PCORDS+20
            L    1,=F'3'
            L    3,=F'2'
JA&SYSNDX  STC  3,0(2)
            A    2,=F'20'
            STC 3,0(2)
            S    3,=F'1'
            A    2,=F'20'
            BCT 1,JA&SYSNDX
            LA  2,PCORDS+21
            L    1,=F'3'
            L    4,=F'2'
KA&SYSNDX  STC  4,MA&SYSNDX+1
            STC 4,LA&SYSNDX+1
LA&SYSNDX  MVC  0(99,2),ZERO
            A    2,=F'20'
MA&SYSNDX  MVC  0(99,2),=C'

```

*RETRIEVAL MODE GOES TO JOB-LIST.

OPCORDS (CONT.)

	A	2,=F'20'	
	S	4,=F'1'	
	BCT	1,KA&SYSNDX	
NA&SYSNDX	CLI	MODE+3,X'00'	
	BE	UA&SYSNDX	
	CLC	TPCORD+3(1),PCGATE	
	BNH	UA&SYSNDX	
	STM	0,15,TL+240	
	LM	0,7,ZERO	
	IC	2,PCGATE	
	L	1,TPCORD	
	SR	1,2	
	C	1,=F'0'	
	BNH	TA&SYSNDX	*DON'T HAVE TO REARRANGE
	L	5,TPCORD	
	M	4,=F'20'	
	LA	4,PCORDS+20	
	AR	5,4	*IR5=LAST PCORD ADDRESS.
OA&SYSNDX	LA	3,PCORDS+36	
	L	2,TPCORD	
	LE	0,0(3)	
	ST	3,TL	
PA&SYSNDX	LE	2,0(3)	
	CER	2,0	
	BNL	QA&SYSNDX	*SR TOO LARGE.
	LER	0,2	*FOUND NEW LOWEST SR.
	ST	3,TL	
QA&SYSNDX	A	3,=F'20'	
	BCT	2,PA&SYSNDX	*MORE PCORDS TO CHECK.
	L	2,TPCORD	*DELETE THE LOWEST PCORD.
	S	2,=F'1'	
	ST	2,TPCORD	*TPCORD NOW UPDATED.
	L	3,TL	
	S	3,=F'16'	
	IC	6,0(3)	
	LA	7,PCORDS	
	AR	7,6	
	IC	6,0(7)	
	S	6,=F'1'	
	STC	6,0(7)	*PCORD CONTROL AREA NOW UPDATED.
	LR	7,5	
	SR	7,3	
	LR	4,3	
	A	4,=F'20'	
RA&SYSNDX	MVI	SA&SYSNDX+1,X'FF'	
	C	7,=F'256'	
	BNL	SA&SYSNDX	
	S	7,=F'1'	
	STC	7,SA&SYSNDX+1	

OVERRIDE

*****OVERRIDE*****

MACRO
OVERRIDE

XX

*---OVERRIDE IS CALLED FROM SJOBLIST ONLY

*

CLI MODE+3,X'03'

***THE READING OF THE OVER-RIDE INFORMATION BEGINS NEXT.

BL	HA&SYSNDX	*MODE<3.
BH	BA&SYSNDX	*MODE=4.
REID1	I,NOVER3	*MODE=3.
L	3,NOVER2	*MODE=3.
C	3,ZERO	*MODE=3.
BNH	HA&SYSNDX	
L	4,AAOVER2R	
AA&SYSNDX	REID1 A,0(4)	*TYPE II.
A	4,JII	*TYPE II.
BCT	3,AA&SYSNDX	*TYPE II.
B	HA&SYSNDX	*TO TYPE III.
BA&SYSNDX	REID3 I,NOVER1,NOVER2,NOVER3	*MODE=4.
L	3,NOVER1	*MODE=4.
C	3,ZERO	*MODE=4.
BNH	DA&SYSNDX	*MODE=4.
L	4,AAOVER1	*MODE=4.
CA&SYSNDX	REID1 J,0(4)	*MODE=4.
A	4,JII	*MODE=4.
BCT	3,CA&SYSNDX	*MODE=4.
DA&SYSNDX	L 3,NOVER2	*MODE=4.
C	3,ZERO	*MODE=4.
BNH	FA&SYSNDX	*MODE=4.
L	4,AAOVER2	*MODE=4.
EA&SYSNDX	REID1 J,0(4)	*MODE=4.
A	4,JII	*MODE=4.
BCT	3,EA&SYSNDX	*MODE=4.
FA&SYSNDX	L 3,NOVER3	*MODE=4.
C	3,ZERO	*MODE=4.
BNH	JA&SYSNDX	
L	4,AAOVER3	*MODE=4.
GA&SYSNDX	REID1 J,0(4)	*MODE=4.
A	4,JII	*MODE=4.
BCT	3,GA&SYSNDX	*MODE=4.
B	JA&SYSNDX	
HA&SYSNDX	L 3,NOVER3	*MODE<4.
C	3,ZERO	*MODE<4.
BNH	JA&SYSNDX	*MODE<4.
L	4,AAOVER3R	*MODE<4.
IA&SYSNDX	REID1 A,0(4)	*MODE<4.
A	4,JII	*MODE<4.
BCT	3,IA&SYSNDX	*MODE<4.
JA&SYSNDX	NOPR 1	
	MEND	

*****OVERRIDE*****

PPCORDS

*****PPCORDS *****

MACRO

&J PPCORDS &RADD

*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXPPCORDS XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

*---PPCORDS PUNCHES AND PRINTS THE PCORDS TABLE IF LEXPCH=0.

*---&RADD IS THE RETURN ADDRESS.

*

```

-----
&J      L      2,LJ
        C      2,ZERO
        BNL    AA&SYSNDX
        C      2,=F'-1'
        BNE    &RADD
AA&SYSNDX  CLI   LEXPCH+3,X'00'
        BNE    &RADD
        CLI   TPCORD+3,X'00'
        BE     &RADD
        MVC   WORKA(80),ASTERIK+1
        PUT   PUNCH,WORKA
        MVC   WORKA(133),BLANK
        MVC   WORKA(42),=C'*      PERMANENT CORDS PUNCHED IN A FORMAT.X

        PUT   PUNCH,WORKA
        MVC   WORKA(42),=C'      PERMANENT CORDS ARE NEXT(B FORMAT)X

        PUT   PRINT,WORKA
        DECPC TPCORD,TPCORD,CORDS
        MVC   WORKA(80),BLANK
        MVI   WORKA+20,X'EG'
        L     7,TPCORD
        ST    7,WORKA
        PUT   PUNCH,WORKA
        A     7,=F'1'
        LA    6,PCORDS
BA&SYSNDX  MVC   WORKA(20),0(6)
        A     6,=F'20'
        PUT   PUNCH,WORKA
        BCT   7,BA&SYSNDX
        LA    7,PCORDS
        S     6,=F'4'
        PRINT B,0(7),0(6)
        MVC   WORKA(80),ASTERIK+1
        PUT   PUNCH,WORKA
        MEND

```

*****PPCORDS *****

PRINT

```

*****PRINT *****
      MACRO
&J      PRINT &FORMAT,&FROM,&TO
*****PRINT *****
*-----PRINT IS THE PRINT PSEUDO OPERATION.
*-----&FORMAT IS THE FORMAT.
*-----&FROM IS THE FROM ADDRESS.
*-----&TO IS THE TO ADDRESS.
*-----
&J      STM      0,15,IOSAVE
AA&SYSNDX LA      0,&FROM
        LA      3,&TO
        LR      2,0
        LA      5,AA&SYSNDX.-8
        MVC     FORMAT(8),=C'&FORMAT.
        MVC     WORKA(133),BLANK
        MVC     WORKA(74),BA&SYSNDX
        B       CA&SYSNDX
BA&SYSNDX DC      C'0 AFTER STATEMENT .FIRST=&FROM. ;LAST=&TX
        O.;FORMAT=&FORMAT.
CA&SYSNDX TESTL PRINT
        LM      0,15,IOSAVE
        MEND
*****PRINT *****

```

PUNSH

*****PUNSH *****

MACRO

&J PUNSH &FORMAT,&FROM,&TO
*XX
*---PUNSH PUNCHES COLUMN BINARY ON CARDS.
*---&FORMAT IS THE FORMAT.
*---&FROM IS THE FROM ADDRESS.
*---&TO IS THE TO ADDRESS.
*-----

&J STM 0,15,IOSAVE
MVC WORKA(133),BLANK
LA 0,&TO
LA 2,&FROM
LR 3,0
SR 3,2
A 3,=F'4'
BP BA&SYSNDX
MVC WORKA(45),AA&SYSNDX
PUT PRINT,WORKA
B GA&SYSNDX *FINISHED.
AA&SYSNDX DC C'0 ERROR IN X FORMAT ADDRESS(&FROM.>&TO.). X
BA&SYSNDX MVC WORKA(80),ASTERIK+1
PUT PUNCH,WORKA
MVC WORKA(80),CA&SYSNDX
PUT PRINT,WORKA
MVI WORKA,C'*'
PUT PUNCH,WORKA
B DA&SYSNDX
CA&SYSNDX DC C'0 PUNCHED OUTPUT IN &FORMAT. FORMAT FROM &FROM. TX
O &TO..
DA&SYSNDX MVC WORKA(80),BLANK
ST 3,WORKA
PUT PUNCH,WORKA
EA&SYSNDX C 3,=F'80'
MVC WORKA(80),BLANK
BNL FA&SYSNDX
S 3,=F'1'
STC 3,FA&SYSNDX.+1
FA&SYSNDX MVC WORKA(80),O(2)
A 2,=F'80'
PUT PUNCH,WORKA
S 3,=F'80'
BH EA&SYSNDX
MVI FA&SYSNDX.+1,X'4F'
MVC WORKA(80),ASTERIK+1
PUT PUNCH,WORKA
GA&SYSNDX LM 0,15,IOSAVE
MEND X

*****PUNSH *****

PUNXH

```

*****PUNXH *****
MACRO
&J PUNXH &FORMAT,&FROM,&TO
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---PUNXH IS THE CARD-PUNCHING COMPONENT CALL MACRO.
*---&FORMAT IS THE FORMAT.
*---&FROM IS THE FROM ADDRESS.
*---&TO IS THE TO ADDRESS.
*-----*
&J STM 0,15,IOSAVE
AA&SYSNDX LA 0,&FROM
LA 3,&TO
LR 2,0
LA 5,AA&SYSNDX.-8
MVC FORMAT(8),=C'&FORMAT.
MVC WORKA(133),BLANK
MVC WORKA(80),BA&SYSNDX
B CA&SYSNDX
BA&SYSNDX DC C'0 PUNCHED AFTER .FIRST=&FROM. ;LAST=&TX
O.;FORMAT=&FORMAT.
CA&SYSNDX TESTL PUNXH
LM 0,15,IOSAVE
MEND
*****PUNXH *****

```


REID4

```

*****REID4 *****
MACRO
&J      REID4 &FORMAT,&W1,&W2,&W3,&W4
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---REID4 READS FOUR VARIABLES OR ARRAYS FROM CARDS.
*---&FORMAT IS THE FORMAT.
*---&W1 IS THE FIRST VARIABLE OR ARRAY.
*---&W2 IS THE SECOND VARIABLE OR ARRAY.
*---&W3 IS THE THIRD VARIABLE OR ARRAY.
*---&W4 IS THE FOURTH VARIABLE OR ARRAY.
*-----
&J      STM    0,15,IOSAVE
        L      0,=F'4'
        ST     0,NOV
AA&SYSNDX B      CA&SYSNDX
BA&SYSNDX DC     C'OREAD IN AFTER STATEMENT          . INFORMATION FOR X
        &W1.,&W2.,&W3.,&W4.,
CA&SYSNDX MVC    FORMAT(10),=C'&FORMAT.
        MVC    WORKA(133),BLANK
        MVC    WORKA(85),BA&SYSNDX
        LA     0,&W1
        ST     0,ADDRESS
        LA     0,&W2
        ST     0,ADDRESS+4
        LA     0,&W3
        ST     0,ADDRESS+8
        LA     0,&W4
        ST     0,ADDRESS+12
        LA     5,AA&SYSNDX.-8
        TESTL  REID
        LM     0,15,IOSAVE
        MEND
*****REID4 *****

```


REID6

```

*****REID6 *****
MACRO
&J REID6 &FORMAT,&W1,&W2,&W3,&W4,&W5,&W6
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---REID6 READS SIX VARIABLES OR ARRAY.
*---&FORMAT IS THE FORMAT.
*---&W1 IS THE FIRST VARIABLE OR ARRAY.
*---&W2 IS THE SECOND VARIABLE OR ARRAY.
*---&W3 IS THE THIRD VARIABLE OR ARRAY.
*---&W4 IS THE FOURTH VARIABLE OR ARRAY.
*---&W5 IS THE FIFTH VARIABLE OR ARRAY.
*---&W6 IS THE SIXTH VARIABLE OR ARRAY.
*-----
&J STM 0,15,IOSAVE
L 0,=F'6'
ST 0,NOV
AA&SYSNDX B CA&SYSNDX
BA&SYSNDX DC C'READ IN AFTER STATEMENT . INFORMATION FOR X
CA&SYSNDX MVC FORMAT(10),=C'&FORMAT.
MVC WORKA(133),BLANK
MVC WORKA(100),BA&SYSNDX
LA 0,&W1
ST 0,ADDRESS
LA 0,&W2
ST 0,ADDRESS+4
LA 0,&W3
ST 0,ADDRESS+8
LA 0,&W4
ST 0,ADDRESS+12
LA 0,&W5
ST 0,ADDRESS+16
LA 0,&W6
ST 0,ADDRESS+20
LA 5,AA&SYSNDX.-8
TESTL REID
LM 0,15,IOSAVE
MEND
*****REID6 *****

```


RESERCO

```

*****RESERCO *****
      MACRO
      RESERCO &LTHAYY,&TPCORD,&LJBLIST
*****RESERCO *****
*-----RESERCO  INITIALIZES THE COPAK COMPRESSORS.
*-----&LTHAYY IS THE LENGTH OF THE PRINCIPAL DATA ARRAY (YY).
*-----&TPCORD IS THE MAXIMUM NUMBER OF PERMANENT CORDS TO BE USED.
*-----&LJBLIST IS THE LENGTH OF THE JOB-LIST WORK ARRAYS.
*-----
SOLID      ENTRY SOLID
           SAVE (14,15)
           CNOP 6,8
           BALR BR1,0
           USING HERE,BR1,BR3,BR4
           JUNKC &TPCORD,&LJBLIST
SPRINGER   SKIPP 1           *SKIPP PAGE AT BEGINNING OF START.
           L      BRY, AYY           *BASE OF YY LOADED.
           MVC   NJOBS(4),ZERO
           LR    BRYY,BRY
           ST    BRYY,SAVEYY
           A     BRY,=F'&LTHAYY.'   *&LTHAYY=LENGTH OF ARRAY YY.
           S     BRY,=F'340'
           STM   BRYY,BRY,SBRY
           L     1,=F'101'
           LA    2,PCORDS
AA&SYSNDX  MVC   0(40,2),ZERO
           A     2,=F'40'
           BCT  1,AA&SYSNDX
           XC   TPCORD(4),TPCORD
           MVC  NTASKS(4),=F'1'
           MVC  NJOBS(4),=F'1'
           MVC  NV(4),=F'1'
           MEND
*****RESERCO *****

```

RESERVE

```

*****RESERVE *****
      MACRO
      RESERVE &ADDL,&LTHAYY,&TPCORD,&LJBLIST
*****RESERVE *****
*-----RESERVE INITIALIZES THE SOLID SYSTEM CONTROL PROGRAM.
*-----&ADDL IS THE NUMBER OF BYTES IN EACH COMPOSITE ADDRESS.
*-----&LTHAYY IS THE LENGTH OF THE PRINCIPLE DATA ARRAY (YY).
*-----&TPCORD IS THE MAXIMUM NUMBER OF PERMANENT CORDS TO BE USED.
*-----&LJBLIST IS THE LENGTH OF THE JOB-LIST WORK ARRAYS.
*-----
SOLID      ENTRY SOLID
           SAVE (14,15)
           CNOP 6,8
           BALR BR1,0
           USING HERE,BR1,BR3,BR4
           JUNKR &TPCORD,&LJBLIST
SPRINGER   SKIPP 1           *SKIPP PAGE AT BEGINNING OF START.
           CALL1 STATECL,AA&SYSNDX
AA&SYSNDX  NOPR 1
           MEND
*****RESERVE *****

```


RMVC

```

*****RMVC *****
MACRO
&J RMVC &D1,&IR,&B1,&D2,&B2
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---RMVC IS THE IBM FORM OF THE RMOVE PSEUDO INSTRUCTION.
*---&D1 IS THE DISPLACEMENT FOR THE TO ADDRESS.
*---&IR IS THE REGISTER CONTAINING THE AMOUNT OF THE RIGHT MOVE.
*---&B1 IS THE BASE REGISTER FOR THE TO ADDRESS.
*---&D2 IS THE DISPLACEMENT FOR THE FROM ADDRESS.
*---&B2 IS THE BASE REGISTER FOR THE FROM ADDRESS.
*-----*
&J STM 0,15,SERVICE
ST &IR,C16
LA 0,&D2.(&B2.)
LA 1,&D1.(&B1.)
LR 2,0
L 0,C16
C 0,ZERO
BNH EA&SYSNDX
L 3,=F'255'
STC 3,CA&SYSNDX.+1
STC 3,DA&SYSNDX.+1
AA&SYSNDX C 0,=F'256'
BNL BA&SYSNDX
LR 3,0
S 3,=F'1'
STC 3,CA&SYSNDX.+1
STC 3,DA&SYSNDX.+1
BA&SYSNDX SR 1,3
SR 2,3
CA&SYSNDX MVC DUMX(256),0(2)
DA&SYSNDX MVC 0(256,1),DUMX
S 1,=F'1'
S 2,=F'1'
S 0,=F'256'
BH AA&SYSNDX
EA&SYSNDX LM 0,15,SERVICE
MEND
*****RMVC *****

```

*THE TO ADDRESS.
*THE FROM ADDRESS.

*EXIT.

SAM

```

*****SAM*****
MACRO
&J      SAM  &JII,&NBB,&CODE
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---SAM SUBSTITUTES CODE WORD FOR A CORD AND CLOSES THE STRING.
*---&JII IS THE LENGTH OF THE ENTIRE STRING.
*---&NBB IS NUMBER OF BYTES FROM THE HEAD OF THE STRING.
*---&CODE IS THE CODE WORD TO BE SUBSTITUTED FOR THE CORD.
*-----
&J      STM  BRYY,ERY,TBRYY          *SAVE BRYY AND BRY
        L    0,&JII
        LR   BRYY,0
        L    BRYY,SBRYY
        AR   BRYY,&NBB
        LR   2,BRYY
        LR   1,&NBB
        A    BRYY,R                  *'FROM' POSITION
        SR   0,1                      *BYTES TO MOVE
        CLI  RM+3,X'0C'
        BE   AA&SYSNDX                *TYPE 2 NO CODE INSERTED.
        MVC  0(1,2),&CODE            *CODE INSERTED INTO THE ARRAY.
        LA   2,1(0,2)
AA&SYSNDX LTR  0,0
        BNH  DA&SYSNDX
        MVI  CA&SYSNDX.+1,X'FF'
BA&SYSNDX C    0,=F'256'
        BNL  CA&SYSNDX
        BCTR 0,0
CA&SYSNDX STC  0,CA&SYSNDX.+1
        MVC  0(0,2),0(BRYY)
        LA   BRYY,256(0,BRYY)
        LA   2,256(0,2)
        S    0,=F'256'
        BH   BA&SYSNDX
DA&SYSNDX S    BRY,R
        CLI  RM+3,X'0C'
        BE   EA&SYSNDX
        LA   BRY,1(0,BRY)
EA&SYSNDX ST   BRY,&JII
        LM   BRYY,ERY,TBRYY          *RESTORE BRYY AND BRY
        MEND
*****SAM*****

```

SANPAKC

*****SANPAKC*****

MACRO

SANPAKC &NAME, &UR, &RR, &DUMMY

XX

*---SANPAKC IS THE ALPHANUMERIC PART OF THE COPAK COMPRESSOR.

*---&NAME EQUALS ANPAKC.

*---&UR IS THE USING REGISTER (8 OR 9).

*---&RR IS THE RETURN REGISTER.

*---&DUMMY EQUALS SOLID (EXTENDED FORM) OR DUMMY (FOR OVERLAYS).

	USING	&UR	
	DS	OF	
ANPAKC	STSR	&UR, &RR, &DUMMY	
	ST	&RR, SPSAVEBR	
	CLI	MODE+3, X'00'	
	BE	CD&SYSNDX	
	LM	0,7,ZERO	*BEGIN ANPAK COMPRESSOR.
	XC	CS1(64),CS1	*SET COUNTERS TO ZERO.
	XC	R(8),R	
	L	0,JII	
	LTR	0,0	*IS COMPRESSION DESIRED.
	BNH	BD&SYSNDX	*NO COMPRESSION WANTED
	ST	0,CS7	*RECORD OF ENTERING JII USED FOR SR.
	LM	BRY, BRY, SBRY	
	XC	0(256,BRY),0(BRY)	*256 BYTES OF BRY SET TO ZERO.
	XC	LEXICON(256),LEXICON	*LEXICON ARRAY SET TO ZERO.
	L	IR1,JII	
AA&SYSNDX	IC	IR3,0(0,BRY)	*SEARCH TO FIND AVAILABLE CODES.
	IC	IR2,0(IR3,BRY)	
	C	IR2,=F'35'	
	BE	BA&SYSNDX	
	LA	IR2,1(0,IR2)	*R2=NUMBER OF TIMES THIS CODE FOUND.
BA&SYSNDX	STC	IR2,0(IR3,BRY)	
	LA	BRY,1(BRY)	
	BCT	IR1,AA&SYSNDX	
	L	BRY,SBRY	
	LM	4,6,ZERO	
CA&SYSNDX	LA	2,LEXICON+255	*FINISHED SEARCH--SEE WHICH CODES USED.
	IC	IR2,0(0,BRY)	
	LTR	IR2,IR2	
	BE	EA&SYSNDX	*CODE AVAILABLE
	C	IR2,=F'34'	*SEE IF CODE AVAILABLE FOR TYPE 2
	BNH	DA&SYSNDX	*NO--CODE NOT PUT INTO LEXICON.
	STC	IR4,0(0,2)	*YES--PUT CODE AT END OF LEXICON AREA.
	BCTR	2,0	
	LA	1,1(1)	
DA&SYSNDX	LA	BRY,1(BRY)	
	LA	IR4,1(IR4)	
	C	IR4,=F'256'	*HAVE ALL CODES BEEN LOOKED AT?

SANPAK (CONT.)

	BNE	CA&SYSNDX	*NO--GO TO NEXT CODE.
	B	FA&SYSNDX	*YES--START COMPRESSION.
EA&SYSNDX	STC	IR4, LEXICON(3)	*USABLE CODE STORED IN LEXICON.
	LA	BRY, 1(BRY)	
	LA	3, 1(3)	
	LA	IR4, 1(IR4)	
	C	IR4, =F'256'	*HAVE ALL CODES BEEN LOOKED AT?
	BNE	CA&SYSNDX	*NO--GO TO NEXT CODE.
FA&SYSNDX	ST	3, CS5	*CS5= NO, OF AVAILABLE CODE WORDS.
	ST	1, CS1	*CS1=NO, OF CODES TO SEARCH IN TYPE 2.
	LM	IR1, IR4, ZERO	
	LM	BRY, BRY, SBRY	
	CLI	CS5+3, X'00'	*ARE THERE ANY AVAILABLE CODES?
	BNE	GA&SYSNDX	*YES--CONTINUE ON WITH THE COMPRESSION.
	CLI	CS1+3, X'00'	*ANY TYPE 2 CODES?
	BE	BD&SYSNDX	*NO - NO COMPRESSION WITH ANPAK.
	CLI	SWITCH+3, X'01'	*IS THIS SLOW-MODE COMPRESSION?
	BE	VB&SYSNDX	*YES - GO TO TYPE 2 SLOW-MODE COMPRESSION.
	CLI	PCORDS, X'00'	*ANY TYPE 2 FAST-MODE CODES?
	BE	BD&SYSNDX	*NO - NO COMPRESSION WITH ANPAK.
	SR	1, 1	
	L	3, TPCORD	*R3=NUMBER OF PCORDS IN TABLE.
	IC	1, PCORDS	*R1=NUMBER OF TYPE 2 PCORDS.
	ST	1, CS13	*COUNT OF PCORDS LEFT TO SEARCH.
	SR	3, 1	
	LA	1, PCORDS+20	
	M	2, =F'20'	
	AR	3, 1	
	ST	3, CS10	*R3=ADDRESS OF FIRST TYPE 2 PCORD.
	B	GC&SYSNDX	*BEGIN TYPE 2 FAST-MODE COMPRESSION.
GA&SYSNDX	CLI	SWITCH+3, X'00'	*USE FAST OR SLOW MODE?
	BE	FC&SYSNDX	*FASTMODE TO BE USED.
HA&SYSNDX	MVI	R+3, X'0C'	*CORD LENGTH SET = 12.
	MVI	RM+3, X'0B'	
IA&SYSNDX	CLI	CS8+3, X'FF'	
	BE	VC&SYSNDX	
	MVI	CS9+1, X'00'	*RESET COMPRESSION ACHIEVED INDICATOR.
	CCD	J11, CS3	*ENTRY POINT FOR COMPARE RECYCLE
	LTR	1, 1	*WAS THERE A SAVINGS.
	BE	KB&SYSNDX	*NO
	L	IR1, CS5	
	BCTR	IR1, 0	
	ST	IR1, CS5	*CS5 UPDATED (NUMBER OF LEXICONS)
	MVI	CS9+1, X'01'	*SET COMPRESSION ACHIEVED INDICATOR.
	L	IR1, CS6	*CS6=SPOT IN LEXICON TO GET NEW CODE.
	IC	IR2, LEXICON(IR1)	
	STC	IR2, CODE	*CODE TAKEN FROM LEXICON ARRAY.
	LA	IR1, 1(IR1)	
	ST	IR1, CS6	*POSITION IN 'LEXICON' UPDATED.

SANPAKC (CONT.)

```

L      IR1,CS4
***CS4=NUMBER OF COMPRESSIONS WITH THIS R LENGTH.
LA     IR1,1(IR1)
ST     IR1,CS4          *NO. OF R REPEATS UPDATED.
L      IR1,CS8          *CS8=TOTAL NUMBER OF COMPRESSIONS MADE.
LA     IR1,1(IR1)
ST     IR1,CS8          *NO. OF R'S USED UPDATED
LR     3,1              *PUT NO. OF MATCHES IN GR3.
BCTR   3,0
SLA    3,2              *MULTIPLY R3 BY 4.
LR     IR1,3
JA&SYSNDX L      IR2,TL(IR1)    *ENTRY POINT FOR SUBSTITUTION CYCLE.
L      BRYY,SBRY
SR     IR2,BRY
SAM    JII,IR2,CODE    *MAKE SUBSTITUTIONS FOR MATCHED CORDS.
S      IR1,=F'4'
BNM    JA&SYSNDX      *ANOTHER SUBSTITUTION TO BE MADE.
CLI    RM+3,X'00'
BE     ZB&SYSNDX
KA&SYSNDX LEX      JII,CODE,CORD1  *ASSEMBLE LEXICON AT HEAD OF STRING.
MVC    SR4(1),CODE    *SAVE CODE FOR THE TIME BEING.
B      JC&SYSNDX      *COMPUTE SAVINGS RATIO FOR THIS TRIAL.
LA&SYSNDX MVI     R+3,X'01'      *RESET CORD LENGTH VALUE.
MVI    RM+3,X'00'
MVC    CORD1(1),CODE  *MOVE TYPE 2 CODE INTO CORD1.
B      NA&SYSNDX
MA&SYSNDX CLI     CS9,X'01'      *IS CONTROL IN TYPE 2-SLOW MODE?
BE     LA&SYSNDX      *YES--RESET R AND RM.
NA&SYSNDX L      3,RM      *RETURN POINT FROM COMPUTING SAVINGS RATIO.
LA     2,PCORDS+20    *FIND OUT WHETHER TO UPDATE,ADD,OR SUB.
LA     0,PCORDS+4020 *R0=LAST ADDRESS IN PCORDS TABLE.
LE     4,ZERO
OA&SYSNDX CLC     0(1,2),RM+3
BNH    VA&SYSNDX
LA     2,20(2)        *HAVEN'T FOUND THE SPOT YET.
CR     2,0            *HAVE ALL CORDS BEEN CHECKED YET ?
BL     OA&SYSNDX      *NO - GO CHECK NEXT CORD.
PA&SYSNDX LA     2,PCORDS+4000 *SET UP POSITION FOR NEW CORD ADDITION.
QA&SYSNDX LE     0,PCORDS+16    *PCORD AREA FULL -- DELETE ONE CORD.
CER    2,0            *IS THIS SR LARGER THAN THE SMALLEST IN PCORDS?
BL     JB&SYSNDX      *NO CORD TO BE ADDED--SR TOO SMALL.
L      4,PCORDS+12    *LOAD ADDRESS WHERE LOWEST SR STORED.
SR     1,1
IC     1,0(0,4)      *GET CORD LENGTH OF CORD TO BE DELETED.
CR     4,2            *COMPARE ADDRESSES OF OLD AND NEW CORDS.
BE     AB&SYSNDX      *JUST INSERT NEW CORD IN OLD CORD SPOT.
BH     UA&SYSNDX
***MUST MOVE PCORDS RIGHT TO DELETE OLD CORD.
LR     5,2            *MUST MOVE PCORDS LEFT TO DELETE OLD CORD.

```

SANPAKC (CONT.)

	SR	5,4	*5=NUMBER OF BYTES TO MOVE.
	LA	7,PCORDS+4000	
	CR	7,2	
	BE	RA&SYSNDX	
	S	5,=F'20'	
	S	2,=F'20'	
RA&SYSNDX	LA	7,20(0,4)	
	MVI	TA&SYSNDX+1,X'FF'	
SA&SYSNDX	C	5,=F'256'	*ENTRY POINT FOR MOVE LEFT RECYCLE.
	BNL	TA&SYSNDX	
	LR	6,5	
	BCTR	6,0	
	STC	6,TA&SYSNDX+1	
TA&SYSNDX	MVC	0(0,4),0(7)	*MOVING CORDS LEFT TO DELETE OLD CORD.
	S	5,=F'256'	
	BNH	AB&SYSNDX	*MOVE FINISHED -- INSERT NEW PCORD.
	LA	4,256(4)	
	LA	7,256(7)	
	B	SA&SYSNDX	
UA&SYSNDX	LR	5,4	*MOVING CORDS RIGHT TO DELETE OLD CORD.
	SR	5,2	*5= NUMBER OF BYTES TO MOVE.
	BCTR	4,0	
	LA	3,20(4)	3=TO ADDR*4=FROM ADDRESS
	RMVC	0,5,3,0,4	
	B	AB&SYSNDX	*MOVE FINISHED--INSERT NEW CORD.
VA&SYSNDX	LA	5,PCORDS+4020	*CHECK TO FIND IF CORD IN PCORDS.
	LR	6,2	
	LA	7,1(2)	
	MVC	XA&SYSNDX.+9(1),RM+3	
	L	4,RM	
WA&SYSNDX	LR	2,6	
	IC	3,0(0,6)	
	CR	3,4	*ARE THE CORD LENGTHS STILL THE SAME.
	BNE	YA&SYSNDX	*CORD NOT FOUND IN PCORDS.
XA&SYSNDX	CE	4,16(0,6)	*IS SAVINGS RATIO = ZERO ?
	BE	AB&SYSNDX	*YES - INSERT NEW PCORD.
	CLC	1(0,6),CORD1	*IS THIS THE SAME CORD ?
	BE	ZA&SYSNDX	*SAME CORD FOUND.
	LA	6,20(6)	*NO GO TRY THE NEXT CORD.
	LA	7,20(7)	
	CR	6,5	*HAVE ALL THE CORDS BEEN CHECKED?
	BE	PA&SYSNDX	*YES--GO TO PCORDS FULL ROUTINE.
	B	WA&SYSNDX	*NO--GO CHECK NEXT CORD.
YA&SYSNDX	CLI	TPCORD+3,X'C8'	
	BE	QA&SYSNDX	
	LA	4,PCORDS+4000	
	B	UA&SYSNDX	
ZA&SYSNDX	LE	0,16(0,6)	*LOAD OLD SR.
	AER	0,2	*ADD NEW SR.

SANPAK (CONT.)

```

HER      2,0                                *AVERAGE THE SR.
LNER     2,2                                *LOAD NEGATIVE TO SHOW CORD LOOKED AT.
STE      2,16(0,6)                          *STORE NEW SR IN THE OLD SPOT.
B        IB&SYSNDX                          *GO BACK TO CONTROL PROGRAM
AB&SYSNDX LNER     2,2                        *INSERT NEW PCORD INTO PCORD ARRAY.
XC       0(20,2),0(2)
MVC      0(1,2),RM+3
MVC      BB&SYSNDX+1(1),RM+3
BB&SYSNDX MVC     1(0,2),CORD1              *CORD STORED INTO PCORDS.
STE      2,16(0,2)                          *SR FOR THE NEW PCORD STORED INTO PCORDS.
L        2,TPCORD
C        2,=F'200'                          *IS PCORDS ARRAY FULL?
BNE      CB&SYSNDX                          *NO--UPDATE TPCORD AND NEW CORD LENGTH.
LA       3,PCORDS
AR       3,1                                *GR1=LENGTH OF DELETED CORD.
IC       1,0(3)
BCTR     1,0                                *UPDATE NO. CORDS OF DELETED CORD'S LENGTH.
STC      1,0(3)
B        DB&SYSNDX
CB&SYSNDX LA      2,1(2)                    *UPDATE NO. CORDS IN PCORDS.
ST       2,TPCORD
DB&SYSNDX LA      3,PCORDS                  *UPDATE NO. OF CORDS FOR NEW CORD LENGTH.
L        4,RM
AR       3,4
IC       4,0(3)
LA       4,1(4)
STC      4,0(3)
B        IB&SYSNDX                          *GO BACK TO CONTROL PROGRAM.
EB&SYSNDX LA      1,PCORDS+36
LA       2,200
FB&SYSNDX LE      0,0(1)                    *LOAD SR FOR EACH CORD IN PCORDS.
CE       0,ZERO                             *WAS CORD LOOKED AT IN SLOW-MODE ?
BL       GB&SYSNDX                          *YES--LOAD NUMBER POSITIVE AND RE-STORE
HER      2,0
B        HB&SYSNDX                          *STORE UPDATED SR.
GB&SYSNDX LPER    2,0
HB&SYSNDX STE     2,0(1)                    *CORRECTED SR RE-STORED FOR THIS CORD.
LA       1,20(1)
BCT      2,FB&SYSNDX                        *ARE THERE ANY MORE CORDS TO SEARCH?
B        VC&SYSNDX                          *NO--GO TO EXIT FROM ANPAK.
IB&SYSNDX CLI     TPCORD+3,X'C8'           *IS PCORD TABLE FULL ?
BNE      JB&SYSNDX                          *NO - CONTINUE ON.
MVI     DUN5,X'01'                          *YES - FIND PCORD WITH LOWEST SR.
B        WC&SYSNDX                          *GO FIND PCORD WITH THE LOWEST SR.
JB&SYSNDX CLI     CS9,X'01'                 *IS THIS TYPE 2 ?
BE       BC&SYSNDX                          *YES.
CLI     CS4+3,X'10'                          *WAS THIS THE 16TH REPEAT OF AN R?
BE       MB&SYSNDX                          *YES (GO TO 'RRL' MACRO)
CLI     CS5+3,X'00'                          *ARE THERE ANY CODES LEFT FOR TYPE 1?

```

SANPAKC (CONT.)

	BE	MB&SYSNDX	*NO MORE CODES LEFT.
	B	LB&SYSNDX	
KB&SYSNDX	CLI	SWITCH+3,X'00'	*IS CONTROL IN FAST MODE?
	BE	JC&SYSNDX	*YES
	L	IR1,CS3	*PREPARE FOR COMPARE RE-CYCLE.
	LA	IR1,1(IR1)	*(NO SAVINGS ON PREVIOUS TRIAL)
	ST	IR1,CS3	*CS3=NO. BYTES FROM BEGINNING OF STRING.
LB&SYSNDX	L	IR4,R	*CAN MORE COMPARES BE MADE ?
	SLL	IR4,2	*MULTIPLY R BY 4.
	A	IR4,CS3	*PLUS BYTES PAST STRING HEAD.
	S	IR4,JII	*MINUS LENGTH OF STRING.
	BM	IA&SYSNDX	*YES - CONTINUE WITH SCAN.
	CLI	CS4+3,X'00'	
***NO--WAS	THESE	A SAVINGS THIS CORD LENGTH.	
	BE	UB&SYSNDX	*NO SAVINGS THIS CORD LENGTH
MB&SYSNDX	RRL	JII,CS4	*NO - PUT ON REPEATS - R BYTE.
	LM	2,4,ZERO	
	ST	4,CS4	*RESET 'REPEATS' COUNTER TO ZERO.
	L	BRYY,SBRYY	*RESTORE STRING HEAD ADDRESS.
	CLI	RM+3,X'00'	*IS THIS TYPE 2 ?
	BNE	NB&SYSNDX	*NO - THIS IS TYPE 1.
	CLI	SWITCH+3,X'00'	*IS THIS FAST-MODE ?
	BE	UC&SYSNDX	*YES - TYPE 2 FAST-MODE.
	L	4,CS5	*NO - TYPE 2 SLOW-MODE.
	B	DC&SYSNDX	*GO BACK TO TYPE 2 SLOW-MODE.
NB&SYSNDX	CLI	CS5+3,X'00'	*ANY TYPE 1 CODES LEFT ?
	BNE	PB&SYSNDX	*YES - CONTINUE WITH TYPE 1 SCAN.
OB&SYSNDX	CLI	SWITCH+3,X'00'	*IS THIS FAST-MODE ?
	BE	KC&SYSNDX	*YES - START TYPE 2 FAST-MODE.
	CLI	CS1+3,X'00'	*CHECK IF TYPE 2 SLOW-MODE POSSIBLE.
	BNE	VB&SYSNDX	*START TYPE 2 SLOW-MODE.
	B	VC&SYSNDX	*NO - END ANPAK EXECUTION.
PB&SYSNDX	IC	4,0(O,BRYY)	*R4= COMPOSITE BYTE.
QB&SYSNDX	IC	3,LEXICON(2)	*R3=BYTE FROM LEXICON ARRAY.
	CR	3,4	*FIND WHERE RRL COMPOSITE BYTE IS LOCATED
	BNL	RB&SYSNDX	
	LA	2,1(2)	
	B	QB&SYSNDX	
RB&SYSNDX	BNE	TB&SYSNDX	*BYTE IS NOT ONE OF THE AVAILABLE CODES.
	IC	3,SR4	
	CR	3,4	*WILL BYTE CAUSE TROUBLE IN DECOMPRESSION
	BH	TB&SYSNDX	*NO - DON'T HAVE TO DELETE BYTE.
	L	3,CS5	*BYTE WILL CAUSE TROUBLE.
	A	3,CS6	*MUST DELETE IT FROM LEXICON ARRAY.
	SR	3,2	
	LA	1,LEXICON(2)	
	LA	2,1(0,1)	
	BCTR	3,0	
	STC	3,SB&SYSNDX+1	

SANPAK (CONT.)

SB&SYSNDX	MVC	0(0,1),0(2)	
	L	IR1,CS5	
	BCTR	IR1,0	
	ST	IR1,CS5	*UPDATE NO. AVAILABLE CODES.
TB&SYSNDX	CLI	CS5+3,X'00'	*ANY TYPE 1 CODES LEFT ?
	BE	OB&SYSNDX	*NO TYPE 1 CODES LEFT.
	CLI	SWITCH+3,X'00'	*IS THIS FAST-MODE ?
	BNE	LB&SYSNDX	
	L	1,CS11	
	B	NC&SYSNDX	
UB&SYSNDX	CLI	SWITCH+3,X'00'	*ENTRY POINT TO PREPARE FOR RECYCLE.
	BE	NC&SYSNDX	
	LTR	IR4,IR4	*IS R TO BE DECREMENTED?
	BM	IA&SYSNDX	*NO - GO BACK TO CCD MACRO.
	L	IR2,R	*YES
	BCTR	IR2,0	
	ST	IR2,R	*CORD LENGTH REDUCED BY ONE.
	BCTR	IR2,0	
	ST	IR2,RM	
	LTR	IR2,IR2	
	BE	VB&SYSNDX	*GO TO TYPE 2 COMPRESSION.
	SR	IR2,IR2	*RESET COUNTER FOR RECYCLE.
	ST	IR2,CS3	
	LM	BRYY,BRY,SBRYY	
	B	IA&SYSNDX	*GO BACK TO CCD MACRO.
VB&SYSNDX	LM	1,6,ZERO	*START TYPE 2 COMPRESSION
	STM	1,5,CS2	*COUNTERS RESET.
	ST	1,RM	*MAKE SURE RM SET CORRECTLY
	LA	1,1(1)	
	ST	1,R	*MAKE SURE R SET CORRECTLY.
	LA	1,256	*NUMBER OF BYTES TO SEARCH IN TYPE 2.
	ST	1,CS3	*CS3=NUMBER OF BYTES TO TEST IN TYPE 2
	BCTR	1,0	
	ST	1,CS6	*CS6=POSITION IN 'LEXICON' ARRAY.
	MVI	CS9,X'01'	
WB&SYSNDX	LM	BRYY,BRY,SBRYY	
	L	IR1,JII	
	C	IR1,=F'256'	*IS STRING LONGER THAN 256 BYTES?
	BH	XB&SYSNDX	*NO--COMPARE ENTIRE STRING.
	ST	IR1,CS3	*JII=NUMBER OF BYTES TO COMPARE.
XB&SYSNDX	L	IR1,CS6	*GET POSITION IN 'LEXICON' ARRAY.
	IC	IR2,LEXICON(IR1)	*TAKE CODE FROM ARRAY.
	STC	IR2,CODE	*STORE CODE IN 'CODE' FOR SEARCH.
YB&SYSNDX	CLI	CS8+3,X'FF'	
	BE	VC&SYSNDX	
	MVI	CS9+1,X'00'	*RESET COMPRESSION ACHIEVED INDICATOR.
	FIND	CS3,CODE	*FIND ALL REPEATS OF CODE IN 256 BYTES.
	C	3,=F'34'	*IS THERE A SAVINGS?
	BNH	CC&SYSNDX	*NO.

SANPAKC (CONT.)

	B	HA&SYSNDX	*NO PCORDS GO TO SLOW MODE CONTROL.
GC&SYSNDX	L	1,CS10	
	SR	2,2	
	IC	2,0(1)	*PULL CORD LENGTH OFF PCORDS.
	ST	2,RM	*CORD LENGTH MINUS 1 SET FOR THIS TRIAL.
	STC	2,IC&SYSNDX+1	
	LA	2,1(2)	
	ST	2,R	*CORD LENGTH SET FOR THIS TRIAL.
	CLI	CS11+3,X'00'	
	BH	HC&SYSNDX	
	LA	2,PCORDS	*R2=ADDRESS OF PCORDS CONTROL BLOCK.
	A	2,RM	
	MVC	CS11+3(1),0(2)	*NUMBER OF CORDS FOR THIS LENGTH.
HC&SYSNDX	CLI	RM+3,X'00'	*IS CONTROL TO GO TO TYPE 2 COMPRESSION
	BE	QC&SYSNDX	*YES--GO TO TYPE 2 COMPRESSION.
IC&SYSNDX	MVC	CORD1(0),1(1)	
	B	IA&SYSNDX	*SEARCH STRING FOR MATCHES TO THIS CORD.
JC&SYSNDX	L	1,CS7	*STRING LENGTH AFTER LAST TRIAL.
	S	1,JII	*GR1=NUMBER OF BYTES SAVED ON THIS TRIAL.
	L	BRY,SBRY	
	ST	1,0(BRY)	
	CONVE	0(BRY),0(BRY)	*CONVERT SAVINGS TO FLOATING POINT.
	LPER	2,0	
	L	1,CS7	
	ST	1,0(BRY)	
	CONVE	0(BRY),0(BRY)	*CONVERT LENGTH TO FLOATING POINT.
	MVC	CS7(4),JII	*NEW LENGTH STORED IN CS7.
	LPER	0,0	
	DER	2,0	*COMPUTE SAVINGS RATIO FOR THIS TRIAL.
	CLI	SWITCH+3,X'01'	*IS CONTROL IN SLOW MODE?
	BE	MA&SYSNDX	*YES--GO BACK TO SLOW MODE.
	L	1,CS10	*LOAD ADDRESS WHERE PRESENT PCORD FOUND.
	LE	0,16(0,1)	*LOAD SR FOR THIS CORD.
	AER	0,2	*ADD OLD SR TO NEW SR.
	HER	0,0	*AVERAGE THE SR'S
	LNER	0,0	*LOAD NEGATIVE THE RESULT
	STE	0,16(0,1)	*NEW SAVINGS RATIO STORED.
	L	1,CS10	
	LA	1,20(1)	
	ST	1,CS10	*POSITION IN PCORDS UPDATED.
	L	1,CS13	
	BCTR	1,0	
	ST	1,CS13	*NUMBER OF PCORDS TO SEARCH UPDATED.
	L	1,CS11	
	BCTR	1,0	
	ST	1,CS11	*NUMBER OF CORDS FOR THIS LENGTH UPDATED.
	CLI	CS9,X'01'	
	BE	PC&SYSNDX	
	CLI	CS5+3,X'00'	*ARE THERE ANY MORE CODES LEFT?

SANPAKC (CONT.)

```

      BNE      MC&SYSNDX
KC&SYSNDX  CLI      PCORDS,X'00'      *ARE THERE ANY TYPE 2 PCORDS ?
      BNE      LC&SYSNDX      *YES--GO TO TYPE 2 PCORDS.
      MVC      CS13(4),ZERO      *WANT TO END COMPRESSION STAGE.
      B        OC&SYSNDX
LC&SYSNDX  SR        1,1      *COMPUTE WHERE TYPE 2 PCORDS BEGIN IN TABLE.
      IC        1,PCORDS      *R1= NUMBER OF TYPE 2 PCORDS IN TABLE.
      STC      1,CS11+3      *CS11=NUMBER OF PCORDS WITH THIS LENGTH.
      STC      1,CS13+3      *CS13= NUMBER PCORDS LEFT TO SEARCH.
      L        3,TPCORD      *R3= NUMBER OF PCORDS IN TABLE.
      SR        3,1
      M        2,=F'20'
      LA        1,PCORDS+20
      AR        1,3      *R1= ADDRESS OF FIRST TYPE 2 PCORD.
      ST        1,CS10
      B        OC&SYSNDX
MC&SYSNDX  CLI      CS4+3,X'10'      *WAS THIS THE 16TH REPEAT OF AN R LENGTH.
      BE      MB&SYSNDX      *YES - PUT ON RRL BYTE.
NC&SYSNDX  LTR      1,1      *ARE THERE ANY MORE CODES FOR THIS LENGTH.
      L        1,CS10
      BNE      IC&SYSNDX
OC&SYSNDX  CLI      CS4+3,X'00'
***WAS THERE A SAVINGS WITH THIS CORD LENGTH?
      BNE      MB&SYSNDX      *YES--GO TO 'RRL' MACRO.
      CLI      CS13+3,X'00'      *ARE THERE ANY MORE CORDS TO SEARCH ?
      BH      GC&SYSNDX      *YES--GO BACK AND START WITH NEW CORD.
      BE      VC&SYSNDX      *COMPRESSION STAGE IS FINISHED.
      MVC      BLANK+5(50),=C'TPCORD IS NEGATIVE--ERROR IN ENTERED PCX
      ORDS.....'
      PUT      PRINT,BLANK
      MVC      BLANK+5(50),BLANK+60
      LA      &RR,CLI
      ST      &RR,SPSAVEBR
      B        CD&SYSNDX      *BRANCH OFF MACHINE WITH ERROR MESSAGE.
PC&SYSNDX  CLI      CS9+1,X'00'
      BE      UC&SYSNDX
      MVI      RM+3,X'00'
      MVI      R+3,X'01'
      L        1,CS4
      LA      1,1(1)
      ST      1,CS4      *UPDATE REPEATS FOR THIS R LENGTH.
      L        1,CS8
      LA      1,1(1)
      ST      1,CS8      *UPDATE NUMBER OF R'S USED.
      B        TC&SYSNDX
QC&SYSNDX  MVI      CS9,X'01'      *TYPE 2 COMPRESSION INDICATOR TURNED ON.
      MVC      CS3(4),=F'256'      *LENGTH OF STRING TO SEARCH.
      L        1,JII      *R1= STRING LENGTH.
      C        1,=F'256'      *IS STRING LONGER THAN 256 BYTES ?

```

SANPAKC (CONT.)

```

BH      RC&SYSNDX      *YES--CONTINUE ON.
ST      1,CS3          *RESET LENGTH OF STRING TO SEARCH.
RC&SYSNDX L      1,CS10
MVC     CODE(1),1(1)  *TYPE 2 CODE FOR SEARCH LOADED.
B       YB&SYSNDX    *GO FIND REPEATS OF TYPE 2 CODE
SC&SYSNDX L      1,CS5
LA      1,1(1)
ST      1,CS5
B       JC&SYSNDX    *COMPUTE NEW SAVINGS RATIO.
TC&SYSNDX CLI     CS4+3,X'10'
BE      MB&SYSNDX    *YES - PUT ON RRL BYTE.
UC&SYSNDX L      1,CS11
LTR     1,1
BNE     RC&SYSNDX    *YES--GO BACK AND TRY NEXT CORD.
CLI     CS4+3,X'00'
BNE     MB&SYSNDX    *YES--GO TO 'RRL' MACRO.
VC&SYSNDX LM      BRYY,BRY,SBRY  *ENTRY POINT TO EXIT FROM ANPAK.
LE      0,PCORDS+16
***BEGIN ROUTINE TO FIND PCORD WITH LOWEST SR
LPER   0,0
MVI    DUN5,X'00'
L      2,PCORDS+12
B      XC&SYSNDX
WC&SYSNDX LE     0,PCORDS+36
LPER   0,0
LA     2,PCORDS+20
XC&SYSNDX LA     1,PCORDS+4016
ST     2,DUM1
LA     2,PCORDS+36
LA     3,PCORDS+20
YC&SYSNDX LE     2,0(2)  *IS THIS SR LOWER THAN PRESENT LOWEST SR?
LPER   2,2
CLI    DUN5,X'00'
BNE    ZC&SYSNDX
STE    2,0(2)
ZC&SYSNDX CER    0,2
BL     AD&SYSNDX    *NO -- CHECK NEXT SR FOR LOWEST
LER    0,2          *NEW LOWEST IS FOUND.
ST     3,DUM1      *ADDRESS OF LOWEST SR STORED.
AD&SYSNDX LA     2,20(2)
LA     3,20(3)
CR     2,1
BNH    YC&SYSNDX
STE    0,PCORDS+16
MVC    PCORDS+12(4),DUM1
CLI    DUN5,X'01'
BE     JB&SYSNDX
BD&SYSNDX L      1,J11
LPR    1,1

```

SANPAKC (CONT.)

```

ST      1,JII
JAR     JII,CS8      *PUT ON LENGTH & NUMBER OF COMPRESSIONS.
CD&SYSNDX TESTL ANPAKCC      *CONTINUE USING OF ANPAK.
LTORG
USING  ANPAKCC,&UR
DS      OF
ANPAKCC STSR  &UR,&RR,&DUMMY
STM     0,15,TL
LA      2,PCORDS
LM      0,1,ZERO
LA      3,12
LOOP20  IC     0,0(2)
AR      1,0
LA      2,1(0,2)
BCT     3,LOOP20
C       1,TPCORD
BE      LOOP21
MVC     WORKA+1(132),BLANK
MVI     WORKA,C'1'
MVC     WORKA+10(42),=C'*** PCORD CONTROL SEGMENT IS FOULED UPX
        ***'
PUT     PRINT,WORKA
MVC     WORKA+10(42),BLANK
PUT     PRINT,WORKA
LOOP21  LM     0,15,TL
DD&SYSNDX SAVINGS DD&SYSNDX
TIME    BIN
S       0,SEGTIME
ST      0,SEGTIME
DECPC   SEGMENT,SEGTIME,SECS
L       &RR,SPSAVEBR
BR      &RR
LTORG
MEND

```

*****SANPAKC*****

SANPAKD (CONT.)

CA&SYSNDX	B	DA&SYSNDX	
	L	0, LLENGTH	
	BCTR	0, 0	
	ST	0, LLENGTH	
DA&SYSNDX	OPCORDS	KA&SYSNDX	
	LM	BRYY, BRY, SBRY	
	LM	1, 6, ZERO	*BEGIN ANPAK DECOMPRESSOR.
	XC	CS1(64), CS1	*ZERO OUT COUNTERS.
	STM	1, 2, R	*ZERO R AND RM FOR DECOMPRESSOR.
	XC	CORD1(24), CORD1	
	LAND	JII, CS8	*REMOVE LENGTH AND R'S FROM HEAD OF STRING.
	L	0, JII	
	LTR	0, 0	
	BNH	JA&SYSNDX	
	C	0, PARM	
	BH	JA&SYSNDX	
	CLI	CS8+3, X'00'	*WAS THERE ANY COMPRESSION?
	BE	JA&SYSNDX	*NO--RETURN CONTROL TO DRIVER.
	LM	BRYY, BRY, SBRY	
EA&SYSNDX	RAND	JII, CS4	*CS4= NUMBER OF REPEATS THIS R LENGTH.
	A	BRY, JII	
FA&SYSNDX	CLI	RM+3, X'00'	*WAS THIS TYPE 2 COMPRESSION?
	BE	IA&SYSNDX	*YES--USE TYPE 2 DECOMPRESSOR.
	LEXD	JII, CODE, CORD1	*REMOVE LEXICON FROM STRING.
	FIND	JII, CODE	*FIND WHERE CODES WERE INSERTED.
	S	2, =F'4'	
	LR	4, 2	*GR4= POSITION IN 'TL' ARRAY.
GA&SYSNDX	L	5, TL(4)	*GR5= ADDRESS OF A MATCHED CODE-WORD.
	SR	5, BRYY	
	CLI	RM+3, X'00'	*IS THIS TYPE 2 COMPRESSION.
	BNE	HA&SYSNDX	*NO GO DIRECTLY TO MAS MACRO.
	L	6, CS11	
	BCTR	6, 0	
	ST	6, CS11	
	SR	5, 6	
HA&SYSNDX	MAS	JII, 5, CORD1	*MAKE SUBSTITUTION FOR CODE-WORD.
	S	4, =F'4'	
	BNM	GA&SYSNDX	*MORE SUBSTITUTIONS TO MAKE.
	L	4, CS8	*CS8=NUMBER OF R'S USED.
	BCTR	4, 0	
	ST	4, CS8	*NUMBER OF R'S USED UPDATED.
	L	5, CS4	
	BCTR	5, 0	
	ST	5, CS4	*NUMBER OF REPEATS THIS LENGTH UPDATED.
	LTR	5, 5	*ARE THERE MORE CODES FOR THIS CORD-LENGTH.
	BNE	FA&SYSNDX	*MORE CODES LEFT FOR THIS CORD-LENGTH.
	LTR	4, 4	*ARE THERE ANY MORE CODES FOR OTHER LENGTH
	BNE	EA&SYSNDX	*YES--GET NEXT R AND REPEATS.
	B	JA&SYSNDX	*NO--RETURN CONTROL TO DRIVER.

SANPAKD (CONT.)

```

IA&SYSNDX  BBMD  JII,CORDI          *TYPE 2 DECOMPRESSION.
L           4,CS10          *BRANCH TO MAKE SUBSTITUTIONS.
B           GA&SYSNDX      *GO TO SUBSTITUTIONS FOR CODE-WORD.
JA&SYSNDX  NOPR  1
KA&SYSNDX  TESTL ANPAKDC
           LTORG
           USING ANPAKDC,&UR
           DS      OF
ANPAKDC    STSR  &UR,&ERR,&DUMMY
           STRINGD LA&SYSNDX
LA&SYSNDX  L     ERR,SPSAVEBR
           BR     ERR
           LTORG
           MEND

```

*EXIT

*****SANPAKD*****

SANPAKJ (CONT.)

	ST	2, PARM	
	B	DA&SYSNDX	
CA&SYSNDX	L	0, LLENGTH	
	S	0, =F'1'	
	ST	0, LLENGTH	
DA&SYSNDX	OPCORDS	KA&SYSNDX	
	LM	BRY, BRY, SBRY	
	LM	0, 7, ZERO	*BEGIN ANPAK DECOMPRESSOR.
	STM	0, 7, CS1	*LOAD COUNTERS TO ZERO
	STM	0, 7, CS9	*LOAD COUNTERS TO ZERO.
	STM	1, 2, CS15	*LOAD COUNTERS TO ZERO.
	STM	1, 2, R	*ZERO R AND RM FOR DECOMPRESSOR.
	XC	CORD1(24), CORD1	
	LAND	JII, CS8	
***REMOVE	LENGTH	AND NO. R'S FROM HEAD OF STRING.	
	L	0, JII	
	LTR	0, 0	
	BNH	JA&SYSNDX	
	C	0, PARM	
	BH	JA&SYSNDX	
	CLI	CS8+3, X'00'	*WAS THERE ANY COMPRESSION?
	BE	JA&SYSNDX	*NO--RETURN CONTROL TO DRIVER.
	LM	BRY, BRY, SBRY	
EA&SYSNDX	RAND	JII, CS4	*CS4= NUMBER OF REPEATS THIS R LENGTH.
	A	BRY, JII	
FA&SYSNDX	CLI	RM+3, X'00'	*WAS THIS TYPE 2 COMPRESSION?
	BE	IA&SYSNDX	*YES--USE TYPE 2 DECOMPRESSOR.
	LEXD	JII, CODE, CORD1	*NO--REMOVE LEXICON FROM STRING.
	FIND	JII, CODE	*FIND WHERE CODES WERE INSERTED.
	S	2, =F'4'	
	LR	4, 2	*GR4= POSITION IN 'TL' ARRAY.
GA&SYSNDX	L	5, TL(4)	*GR5= ADDRESS OF A MATCHED CODE-WORD.
	SR	5, BRY	
	CLI	RM+3, X'00'	*IS THIS TYPE 2 COMPRESSION.
	BNE	HA&SYSNDX	*NO GO DIRECTLY TO MAS MACRO.
	L	6, CS11	
	BCTR	6, 0	
	ST	6, CS11	
	SR	5, 6	
HA&SYSNDX	MAS	JII, 5, CORD1	*MAKE SUBSTITUTION FOR CODE-WORD.
	S	4, =F'4'	
	BNM	GA&SYSNDX	*MORE SUBSTITUTIONS TO MAKE.
	L	4, CS8	*CS8=NUMBER OF R'S USED.
	BCTR	4, 0	
	ST	4, CS8	*NUMBER OF R'S USED UPDATED.
	L	5, CS4	
	BCTR	5, 0	
	ST	5, CS4	*NUMBER OF REPEATS THIS LENGTH UPDATED.
	LTR	5, 5	*ARE THERE MORE CODES FOR THIS CORD-LENGTH.

SANPAKJ (CONT.)

```

BNE FA&SYSNDX *MORE CODES LEFT FOR THIS CORD-LENGTH.
LTR 4,4 *ARE THERE ANY MORE CODES FOR OTHER LENGTH
BNE EA&SYSNDX *YES--GET NEXT R AND REPEATS.
B JA&SYSNDX *NO--RETURN CONTROL TO DRIVER.
IA&SYSNDX BBMD JII,CORD1 *TYPE 2 DECOMPRESSION.
L 4,CS10 *BRANCH TO MAKE SUBSTITUTIONS.
B GA&SYSNDX *GO TO SUBSTITUTIONS FOR CODE-WORD.
JA&SYSNDX NOPR 1
KA&SYSNDX TESTL ANPAKDC
LTORG
USING ANPAKDC,&UR
DS OF
ANPAKDC STSR &UR,&ERR,&DUMMY
STRINGD LA&SYSNDX
LA&SYSNDX JIMPI MA&SYSNDX
MA&SYSNDX L &ERR,SPSAVEBR
BR &ERR *EXIT
LTORG
MEND

```

*****SANPAKJ*****

SAVEAREA

*****SAVEAREA*****

MACRO
SAVEAREA

XX

*---SAVEAREA IS THE REGISTER DEFINITION AND SAVE AREAS FOR SOLID.

BASE DC A(HERE+4096,HERE+8192) *USING REGISTERS 10,11,AND 12.

***REGISTER DEFINITION AND SAVE AREA ARE NEXT.

SAVEAREA	DS	18F	*SAVE AREA FOR THE IBM SUPERVISOR.
SR4	DS	F	*FOR SAVING IR4.
LBRY	DS	F	*ORIGEN OF CURRENT STRING.
HBRY	DS	F	*HIGH ADDRESS FOR TAPE WORK.
SBRY	DS	F	*BEGINNING ADDRESS FOR CURRENT SEGMENT.
SBRY	DS	F	*ADDRESS FOR LEXICON AREA.
RLBRY	DS	F	*ORIGEN OF STRING(RESERVE)
RHBRY	DS	F	*RESERVE HIGH ADDRESS FOR TAPE WORK
RSBRY	DS	F	*RESERVE START ADDRESS OF CURRENT SEGMENT.

***REGISTER USED IN I/O ARE STORED HERE.

C1	DS	F	*FOR I/O ROUTINES ONLY.
C2	DS	F	*FOR I/O ROUTINES ONLY.
C3	DS	F	*FOR I/O ROUTINES ONLY.
C4	DS	F	*FOR I/O ROUTINES ONLY.
C5	DS	F	*FOR I/O ROUTINES ONLY.
C6	DS	F	*FOR I/O ROUTINES ONLY.
C7	DS	F	*FOR I/O ROUTINES ONLY.
C8	DS	F	*FOR I/O ROUTINES ONLY.
C9	DS	F	*FOR I/O ROUTINES ONLY.
C10	DS	F	*FOR I/O ROUTINES ONLY.
C11	DS	F	*FOR I/O ROUTINES ONLY.
C12	DS	F	*FOR I/O ROUTINES ONLY.
C13	DS	F	*FOR I/O ROUTINES ONLY.
C14	DS	F	*FOR I/O ROUTINES ONLY.
C15	DS	F	*FOR I/O ROUTINES ONLY.
C16	DS	F	*FOR I/O ROUTINES ONLY.

***GENERAL PURPOSE REGISTERS ARE STORED HERE.

CS1	DS	F	*GENERAL PURPOSE COUNTERS.
CS2	DS	F	*GENERAL PURPOSE COUNTERS.
CS3	DS	F	*GENERAL PURPOSE COUNTERS.
CS4	DS	F	*GENERAL PURPOSE COUNTERS.
CS5	DS	F	*GENERAL PURPOSE COUNTERS.
CS6	DS	F	*GENERAL PURPOSE COUNTERS.
CS7	DS	F	*GENERAL PURPOSE COUNTERS.
CS8	DS	F	*GENERAL PURPOSE COUNTERS.
CS9	DS	F	*GENERAL PURPOSE COUNTERS.
CS10	DS	F	*GENERAL PURPOSE COUNTERS.
CS11	DS	F	*GENERAL PURPOSE COUNTERS.

SAVEAREA (CONT.)

CS12	DS	F	*GENERAL PURPOSE COUNTERS.
CS13	DS	F	*GENERAL PURPOSE COUNTERS.
CS14	DS	F	*GENERAL PURPOSE COUNTERS.
CS15	DS	F	*GENERAL PURPOSE COUNTERS.
CS16	DS	F	*GENERAL PURPOSE COUNTERS.

***FOR SAVING REGISTERS IN TRANSFER TO COMPONENT.

TBRY	DS	2F	FOR SAVING ARRAY REGISTERS.
SRF	DS	16F	*FOR SAVING REGISTERS URO TO IR15 IN I/O.
SERVICE	DS	16F	*ONLY FOR THE SERVICE MACROS.
IOSAVE	DS	16F	*SPECIAL AREA FOR SAVING REGISTERS IN I/O
COSAVEME	DS	16F	*SPECIAL AREA FOR SAVING REGISTERS IN SMEMORY
COSAVE1	DS	16F	*SAVE-AREA FOR THE TRANSFER INSTRUCTION.
COSAVE2	DS	16F	*SAVE-AREA FOR THE TRANSFER INSTRUCTION.
COSAVE3	DS	16F	*SAVE-AREA FOR THE TRANSFER INSTRUCTION.
COSAVE4	DS	16F	*SAVE-AREA FOR THE TRANSFER INSTRUCTION.
COSAVE5	DS	16F	*SAVE-AREA FOR THE TRANSFER INSTRUCTION.

***TO SAVE BRANCH REGISTERS.

SPSAVEBR	DS	1F	*SPECIAL FOR SAVING BRANCH REGISTER (SNUPAK).
MEMADD	DS	2F	*FOR SAVING THE USING REGISTERS OF SMEMORY.

***TO SAVE REGISTERS IN TESTL AND STSR INSTRUCTIONS.

WCD	DS	D	*FOR SAVING BRY AND BRY.
WCDC	DS	4F	*TO SAVE IRO,IR1,BRY AND BRY.

***EQUIVALENT STATEMENTS ARE NEXT.

IR1	EQU	4	*INDEX REGISTER.
IR2	EQU	5	*INDEX REGISTER.
IR3	EQU	6	*INDEX REGISTER.
IR4	EQU	7	*INDEX REGISTER.
IR5	EQU	2	
IR6	EQU	3	
BR1	EQU	10	*SYSTEM BASE REGISTER.
BR3	EQU	11	*SYSTEM BASE REGISTER.
BR4	EQU	12	*SYSTEM BASE REGISTER.
BRY	EQU	15	*BASE FOR ARRAY Y.
BRY	EQU	14	*BASE FOR ARRAY YY.

MEND

*****SAVEAREA*****

SAVINGS

```

*****SAVINGS *****
      MACRO
&J      SAVINGS &RADD
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---SAVINGS COMPUTES PERCENTILE SAVINGS AND THRUPUT RATE.
*---&RADD IS THE RETURN ADDRESS.
*-----
&J      CLI      MODE+3,X'00'
        BNE      AA&SYSNDX
        MVC      PSAVINGS(4),JII
AA&SYSNDX B      BA&SYSNDX
        L        5,PSAVINGS
        S        5,JII
        S        5,LLENGTH
        L        4,ZERO
        M        4,=F'100'
        D        4,PSAVINGS
        ST       5,DUM1
        DECPC    PERCENT,DUM1,SAVINGS
BA&SYSNDX SR      4,4
        TIME     BIN
        S        0,SEGTIME
        L        5,PSAVINGS
        M        4,=F'100'
        C        0,=F'1'
        BL      CA&SYSNDX
CA&SYSNDX DR      4,0
        ST       5,DUM1
        DECPC    THRUPUT,DUM1,RATE
        MEND
*****SAVINGS *****

```

*BYTES SAVED IN FIXED POINT.

SCOMMAND (CONT.)

```

=====
***IN THIS SECTION THE MESSAGES ARE PRINTED.
      USING COMANDS, EUR
      DS      OF
COMANDS  STSR  EUR, ERR, EDUMMY
          ST   ERR, SRRMANDS
          L   3, NOS
          C   3, RNOS
          BNE CA&SYSNDX
          L   15, ANEWJOB
          BR   15
CA&SYSNDX L   2, ZERO
          D   2, =F'10'
          C   2, ZERO
          BNE DA&SYSNDX
          SKIPP 1
          TALE ASTERIK, STRING, 4
DA&SYSNDX MVC  NOSEG(4), RNOSEG
=====
***IN THIS SECTION THE NUMBER OF STRINGS ARE COUNTED.
          L   0, NOS
          A   0, =F'1'
          ST   0, NOS
                                     *NUMBER OF STRINGS.
=====
***IN THIS SECTION THE STRING REGISTERS ARE COMPUTED AND STORED.
          L   BRY, SBRY
          S   BRY, =F'8'
          L   BRY, LLENGTH
          C   BRY, ZERO
          BL  EA&SYSNDX
          C   BRY, =F'255'
          BL  FA&SYSNDX
EA&SYSNDX L   BRY, =F'255'
          ST  BRY, LLENGTH
FA&SYSNDX C   BRY, SLENGTH
          BNH GA&SYSNDX
          L   BRY, SLENGTH
          ST  BRY, LLENGTH
GA&SYSNDX A   BRY, SLENGTH
          A   BRY, =F'4'
          C   BRY, =F'300'
          BNL HA&SYSNDX
          L   BRY, =F'300'
HA&SYSNDX SRL  BRY, 2
          SLL  BRY, 2
          SR   BRY, BRY
          ST   BRY, RHBRY
          SR   BRY, BRY
          ST   BRY, RLBRY

```

SCOMMAND (CONT.)

```

A      BRYY,LLENGTH
ST     BRYY,RSBRYY
CLI    GATE,X'00'
BNE    IA&SYSNDX

```

```

=====
***IN THIS SECTION THE STRING COMMANDS ARE READ.
MVC    MODE(4),=F'5'           *DEFAULT SET.
MVC    LJ(4),=F'-2'           *DEFAULT SET.
MVC    LEXCON(4),=F'1'        *DEFAULT SET.
MVC    LEXMODE(4),=F'2'       *DEFAULT SET.
XC     LEXPCH(4),LEXPCH       *DEFAULT SET.
REIDS  I,MODE,LJ,LEXCON,LEXMODE,LEXPCH
CLI    MODE+3,X'05'
BNI    CLI                     *EXIT.
IA&SYSNDX TIME BIN
ST     0,STIME                 *STRING TIME.
MVC    LBRYY(12),RLBRYY        *YY REGISTERS RELOADED.
L      &RR.,SRRMANDS
BR     &RR                     *EXIT.
SRRMANDS DS F
LTORG
MEND
*****SCOMMAND*****

```

SCREEN

*****SCREEN *****

MACRO

&J SCREEN &ADDL,&LFAST,&ADD1,<HAYY
 *XX
 *---SCREEN IS USED TO LOCATE POSITIONS OF EXEC. POINTERS IN SUB-ARRAYS.
 *---&ADDL IS THE NUMBER OF BYTES IN THE COMPOSITE ADDRESSES.
 *---&LFAST IS THE LENGTH OF THE FAST PART OF COMPOSITE ADDRESSES.
 *---&ADD1 CONTAINS THE ADDRESS OF FIRST EXEC.POINTER IN SUB-ARRAY.
 *---<HAYY IS THE NUMBER OF BYTES IN EACH MEMORY BLOCK.
 *-----

```

&J      MVI    JI,X'00'
        STM    0,15,DUMX+8
        SR     0,0
        LH     14,DUN1
        A      14,=F'&ADDL'
        LA     2,&ADD1
        S      2,=F'4'
        MVC    DUMX(4),0(2)
        L      1,DUMX
        S      1,=F'4'
        DR     0,14
        C      0,ZERO
        BE     AA&SYSNDX
        LM     0,15,DUMX+8
        B      HA&SYSNDX
AA&SYSNDX LM     0,15,DUMX+8
***INSERT TEST FOR OVER-RIDES. IF THEY ARE PRESENT CONTROL GOES TO
***MMATCH(AT MMATCHN) FOR DIRECTION(IN STRATEGY) TO COMPONENT SMATCH.
        LH     BRYY,DUN1
        CH     BRYY,ZERO
        BH     BA&SYSNDX
        LA     1,&ADD1
        CLC    0(&ADDL,1),ZERO
        BNE    DA&SYSNDX
        OI     JI,X'01'
        B      JA&SYSNDX
BA&SYSNDX SUPERSCH &ADDL,&LFAST
CA&SYSNDX CSCREEN DUN1,0(IR6),0(IR2)
        CLI    JI,X'01'
        BNL    EA&SYSNDX
DA&SYSNDX AR     IR2,BRYY
        AR     IR6,BRYY
        B      LA&SYSNDX
EA&SYSNDX BE     JA&SYSNDX
FA&SYSNDX CLI    JI,X'02'
        BE     GA&SYSNDX
        CLI    MODE+3,X'01'
        BNE    MMATCHN
        B      KA&SYSNDX

```

*SCREEN LENGTH.

*REPLACE &ADDL BY &LFAST

*R1=LENGTH OF POINTERS.

*SCREEN LENGTHS MATCH.

*FETCH CONTINUANCE.

*SCREEN EXISTS.

*EQUAL.

*ZERO.

*JI>X'00'.

*JI=X'00' (EQUAL).

*EXIT.

*JI=X'01' (ZERO).

*JI=X'04' (HIGH).

SCREEN (CONT.)

```

GA&SYSNDX  AR   IR6,BRY                      *JI=X'02' (LOW).
            CR   IR6,IR3
            BL   CA&SYSNDX
HA&SYSNDX  OI   MSIGNAL,X'02'
            LR   IR6,IR3
            B    LA&SYSNDX                      *EXIT.
IA&SYSNDX  LMOVE DUN1,0(IR6),0(IR2)
            B    DA&SYSNDX
JA&SYSNDX  CLI   MODE+3,X'01'
            BNE  MMATCHN                      *SEARCH FAILED.
*****    CONT INUANCE INSERTION MADE HERE  ***** CONTINUANCE*****
            CLI   LEXICON,X'00'
            BE   IA&SYSNDX
KA&SYSNDX  INSERT &ADDL,&LFAST,&LTHAYY
LA&SYSNDX  NOPR  1
            MEND
*****SCREEN *****
    
```

SCYCLIC

```

*****SCYCLIC *****
MACRO
SCYCLIC &NAME,&UR,&RR,&DUMMY                      LEVEL=3
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXSCYCLIC XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXLEVEL 3
*---SCYCLIC IS THE LEFT- AND RIGHT- CYCLIC SHIFT COMPONENT.
*---&NAME EQUALS CYCLIC
*---&UR IS THE USING REGISTER(NOT 0,1,OR 10-15).
*---&RR IS THE BRANCH REGISTER(NOT 0 OR &UR).
*---&DUMMY IS SOLID(EXTENDED FORM) OR DUMMY(OVERLAY FORM).
*-----
USING CYCLIC,&UR
DS    OF
CYCLIC STSR  &UR,&RR,&DUMMY
      ST    &RR,SRRYCLIC
***ENTRY INFORMATION IS DEFINED IN SNORMAL
***EXIT INFORMATION IS DEFINED IN SNORMAL
      L    &RR,SRRYCLIC
      BR   &RR
SRRYCLIC DS    F
      LTORG
      MEND
*****SCYCLIC *****
    
```


SINCOP

```

*****SINCOP *****
MACRO
&J      SINCOP &RADD
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---SINCOP IS THE SPECIAL MACRO FOR COPAKNU (BOB SINCAVAGE).
*---&RADD IS THE RETURN ADDRESS.
*-----
&J      CLI      MODE+3,X'00'
        BNE      AA&SYSNDX
        SR      0,0
        L        1,RLBRY
        IC      0,0(1)
        A        0,=F'1'
        ST      0,LLENGTH
AA&SYSNDX L        BRY,RLBRY
        LR      BRY,BRY
        A        BRY,LLENGTH
        A        BRY,=F'256'
        L        1,JII
        S        1,LLENGTH
        CLI      MODE+3,X'00'
        BNE      BA&SYSNDX
        A        1,=F'1'
BA&SYSNDX ST      1,JII
        RMOVE   JII+2,0(BRY),0(BRY)
        ST      BRY,SBRY
        ST      BRY,RSBRY
        CLI      MODE+3,X'00'
        BE      CA&SYSNDX
        L        BRY,RLBRY
        LR      BRY,BRY
        A        BRY,=F'1'
        RMOVE   LLENGTH+2,0(BRY),0(BRY)
        MVC     0(1,BRY),LLENGTH+3
        L        0,LLENGTH
        SLL     0,8
        L        2,SOS
        SLR     2,0
        ST      2,SOS
        L        2,PARM
        SLR     2,0
        ST      2,PARM
        B        DA&SYSNDX
CA&SYSNDX L        0,LLENGTH
        S        0,=F'1'
        ST      0,LLENGTH
DA&SYSNDX CLI      MODE+3,X'00'
        BE      EA&SYSNDX
        MVC     BLANK+15(39),=C'ENTERING COPAK COMPRESSOR.BEGIN TIME: X

```

SINCOP (CONT.)

```

EA&SYSNDX  B   FA&SYSNDX
            MVC  BLANK+15(39),=C'ENTERING COPAK DECOMPRESSOR.BEGIN TIMEX
            :
FA&SYSNDX  PUT  PRINT, BLANK
            MVC  BLANK+15(39), BLANK+60
            TIME BIN
            ST   0, SEGTIME
            CLI  OUTPXT+3, X'02'
            BL   IA&SYSNDX
            CLI  MODE+3, X'00'
            BE   IA&SYSNDX
            L    5, LBRY
            L    6, JII
            DECPC COMPRESS, JII, BYTES
            MVI  HA&SYSNDX.+1, X'84'
GA&SYSNDX  MVC  WORKA(133), BLANK
            C    6, =F'132'
            BNL  HA&SYSNDX
            S    6, =F'1'
            STC  6, HA&SYSNDX.+1
HA&SYSNDX  MVC  WORKA+1(132), 0(5)
            PUT  PRINT, WORKA
            A    5, =F'132'
            S    6, =F'132'
            BH   GA&SYSNDX
            TIME BIN
            ST   0, SEGTIME
IA&SYSNDX  L    0, NOSEG
            S    0, =F'1'
            ST   0, NOSEG
            CLI  GATE, X'00'
            BNE  JA&SYSNDX
            CLI  INPXT+3, X'00'
            BNE  JA&SYSNDX
            MVI  GATE, X'01'
JA&SYSNDX  L    0, RNOSEG
            S    0, NOSEG
            ST   0, NDR
            DECPC CURRENT, NOS, STRING
            DECPC CURRENT, NOR, SEGMENT
            CLI  MODE+3, X'00'
            BNE  KA&SYSNDX
            L    BRY, SBRY
            LAND JII, CS8
KA&SYSNDX  STRINGD STRINGDR
STRINGDR  NOPR 1
            MEND
*****SINCOP*****

```

*BEGIN SEGMENT TIME.

SJOBLIST (CONT.)

* JVALUE IS THE MAXIMUM VALUE OF THE SMALL 'M' IN JLITEM GENERATOR.
 * NUMDIAG IS THE TOTAL NUMBER OF DIAGONALS TO BE GENERATED.
 * GENERATE IS THE ODD NUMBER USED BY THE RANDOM NUMBER GENERATORS.

BA&SYSNDX L 1,AJBLIST *ADDRESS OF JBLIST.
 L 2,AJBWORK
 CLI JLGATE,X'10' *JLGATE=X'10' MEANS GENERATE ITEM.
 BL DA&SYSNDX
 L 3,NTASKS
 C 3,ZERO
 BNH HA&SYSNDX
 SR 6,6

CA&SYSNDX JLITEM 0(1),MVALUE,JVALUE,NUMDIAG
 * JLITEM GENERATES A JOBLIST ITEM IN ADDRESS 0(1).
 * 0(1) IS THE ADDRESS OF THE JOB-LIST ARRAY(JBLIST).
 * MVALUE IS THE VALUE OF 'M' IN THE I.R.
 * JVALUE IS THE MAXIMUM VALUE OF THE SMALL "M'S" IN THE I.R.
 * NUMDIAG IS THE NUMBER OF DIAGONALS(OR SCREENS) TO BE GENERATED.

XC NDR(4),NDR *LOAD NDR WITH LENGTH
 MVC NDR+2(2),0(1) *UPDATE TOTAL LENGTH REGISTER
 A 6,NDR *UPDATE POINTER TO NEXT JOBLIST
 A 1,NDR *GENERATE 'NTASKS' TIMES
 BCT 3,CA&SYSNDX *STORE TOTAL LENGTH IN JLL.
 ST 6,JLL *NO TRANSLATION.
 B HA&SYSNDX

DA&SYSNDX CLI JLGATE,X'01'
 BNL FA&SYSNDX
 L 5,JLRSKIP *READ TAPE ITEM.
 A 5,=F'1'

EA&SYSNDX REIDJB 0(2),JLL
 BCT 5,EA&SYSNDX
 XC JLRSKIP(4),JLRSKIP

***TAPE SKIPPED OUT AND THE JOBLIST ITEM WAS READ.

B HA&SYSNDX *TRANSLATE.
 FA&SYSNDX BH GA&SYSNDX *FUTURE DEVICES.

***THE JOB-LIST ITEM IS TO BE READ FROM CARDS.

REID1 *A,0(2)
 MVC JLL(4),JII
 B HA&SYSNDX *TRANSLATE.

GA&SYSNDX MVC BLANK+5(69),=C'THIS VALUE OF THE JLISTDEV RESERVED FORX
 AS YET UNDEFINED INPUT MODES.'
 PUT PRINT,BLANK *MESSAGE.
 MVC BLANK+4(71),BLANK+3
 B CLI *QUIT SYSTEM.

HA&SYSNDX TRANLATE 0(2),NTASKS,JLL,KLENGTH
 * 0(2) CONTAINS THE DEXCRIPTOR SET THAT IS TO BE TRANSLATED.
 * NJOBS IS THE NO. OF BULK INFO. ITEMS STORED UNDER ONE PATH.
 * NTASKS ARE THE TASKS (JOBLIST ITEMS) ASSIGNED IN THE TRANSLATOR.
 * JLL IS THE LENGTH OF THE DISCRIPTOR SET.
 * KLENGTH IS THE NUMBER OF BYTES IN EACH KERNEL OR ELEMENT OF THE I.R.

SJOBLIST (CONT.)

```

* EXIT DATA:- JOBLIST IN JBLIST;LENGTH IN JLL,NJOBS, AND NTASKS.
  OVERRIDE
IA&SYSNDX  CLI  NFORM,X'00'
            BE   JA&SYSNDX
            NORMFORM 0(1),NTASKS,JLL,KLENGTH
* (0(1)) IS THE LOCATION OF JOBLIST.
* NTASKS ARE THE TASKS (JOBLIST ITEMS) ASSIGNED IN THE TRANSLATOR.
* JLL IS THE LENGTH OF THE JOBLIST.
* KLENGTH IS THE NUMBER OF BYTES IN EACH KERNEL OR ELEMENT OF THE I.R.
* EXIT DATA:- NORMALIZED JOBLIST IN JBLIST;JLL,NJOBS AND NTASKS.
* THE ARRAY JBWORK CAN BE USED FOR A WORK AREA IN THE NORMALIZATION.
JA&SYSNDX  L   ERR,SRRBLIST
            BR   ERR
            LTORG
SRRBLIST   DS   F
            MEND

```

```

*NO NORMALIZATION.
*NORMALIZE.

```

```

*RETURN

```

```

*****SJOBLIST*****

```

SKIPP

```

*****SKIPP *****
      MACRO
&J      SKIPP &LC
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---SKIPP FOR SKIPS PAGES ON THE PRINTER.
*---&LC IS THE PAGE SKIPPING CODE (1 IS ONE PAGE).
*-----
&J      STM   0,15,SERVICE
        HVI   BLANK,C'&LC.'
        PUT   PRINT,BLANK
        MVI   BLANK,C' '
        LM    0,15,SERVICE
        MEND
*****SKIPP *****

```


SMEMORY (CONT.)

```

LR      5,3
SLL     3,8
O       3,=F'1'
BA&SYSNDX SLL     3,8
ST      3,WTKEY          *NOW HAVE KEY FOR THIS BLOCK.
MVC     WTREF(3),WTKEY *RELATIVE ADDRESS MOVED FOR WRITE USE.
ST      BRYY,MYBRYY
TM      MSIGNAL,X'02'
BZ      DA&SYSNDX          *WANT TO UPDATE BLOCK ON DISK.
ST      BRYY,NEWBLK+12    *WANT TO ADD A NEW BLOCK TO DISK.
WRITE   NEWBLK,DA,GLOBAL1,WORKM,'S',WTKEY,WTREF
A       5,=F'1'
LR      3,5
SLL     3,8
O       3,=F'1'
WAIT    1,ECB=NEWBLK      *WAIT FOR END OF WRITE TRANSFER.
CLI     NEWBLK+1,X'00'
BNE     JA&SYSNDX          *WRITE OPERATION NOT SUCCESSFUL.
CA&SYSNDX L       BRYY,MYBRYY
A       BRYY,=F'&TRKL.'    *&TRKL EQUALS BYTES/TRACK.
BCT     4,BA&SYSNDX        *MORE WRITES TO DO.
NI      MSIGNAL,X'FD'
B       EA&SYSNDX
DA&SYSNDX ST     BRYY,OLDBLK+12
WRITE   OLDBLK,DK,GLOBAL1,WORKM,'S',WTKEY,WTREF
A       5,=F'1'
LR      3,5
SLL     3,8
O       3,=F'1'
WAIT    1,ECB=OLDBLK
CLI     OLDBLK+1,X'00'
BNE     JA&SYSNDX          *WRITE OLD BLOCK NOT SUCCESSFUL
EA&SYSNDX B       CA&SYSNDX        *MORE WRITES TO DO.
TM      MSIGNAL,X'80'
BZ      FA&SYSNDX
NI      MSIGNAL,X'7D'
OI      MSIGNAL,X'02'
APART   EMPTY+&ADDL,0,1,2,3,4
A       4,=F'&ADDL'
ST      4,COSAVEME+16
ASADD   EMPTY+&ADDL.,0,1,2,3,4
MVC     ADDRESS(&ADDL),EMPTY+&ADDL.
MVC     EMPTY(&ADDL.),EMPTY+&ADDL.
L       15,SAVEYY
MVC     0(&ADDL,15),EMPTY
B       QA&SYSNDX
FA&SYSNDX APART   ADDRESS,0,1,2,3,4
CLC     CURRENT(&ADDL.),CONM
BNE     GA&SYSNDX

```

SMEMORY (CONT.)

	STM	0,3,CURDEV	
GA&SYSNDX	STM	0,3,NXTDEV	
	CLC	NXTDEV(4),BWX	
	BNL	PA&SYSNDX	*DEVICE TYPE DIFFERENT
	CLC	NXTNUM(4),LSX	
	BNL	IA&SYSNDX	*DEVICE NUMBER DIFFERENT.
	SR	2,2	*WANT TO READ FROM SAME DEVICE.
	L	3,NXTCYL	
	M	2,CHECK	
	A	3,NXTTRK	
	S	3,=F'1'	
	L	4,=F'&NTRKS.'	
	L	BRYY,SAVEYY	
	LR	5,3	
	SLL	3,8	
	D	3,=F'1'	
HA&SYSNDX	SLL	3,8	
	ST	3,RDKEY	*HAVE KEY TO FIND BLOCK.
	MVC	RDREF(3),RDKEY	*RELATIVE ADDRESS FOR READ USE.
	ST	BRYY,MYBRYY	
	ST	BRYY,RDBLK+12	
	READ	RDBLK,DK,GLOBAL1,WORKM,'S',WTKEY,WTREF	
	A	5,=F'1'	
	LR	3,5	
	SLL	3,8	
	D	3,=F'1'	
	WAIT	1,ECB=RDBLK	*WAIT FOR END OF READ TRANSFER.
	CLI	RDBLK+1,X'00'	
	BNE	MA&SYSNDX	*READ OPERATION NOT SUCCESSFUL
	L	BRYY,MYBRYY	
	A	BRYY,=F'&TRKL.'	
	BCT	4,HA&SYSNDX	*MORE READS TO DO.
	L	15,SAVEYY	
	MVC	EMPTY(&ADDL),0(15)	
	TM	MSIGNAL,X'04'	
	BZ	QA&SYSNDX	
	MVC	0(&ADDL,15),ADDRESS	
	MVC	EMPTY(&ADDL),ADDRESS	
	B	QA&SYSNDX	*FINISHED READING.
IA&SYSNDX	MVC	WORKA(133),BLANK	
	MVC	WORKA+1(98),=C' DEVICE NUMBER >0 WAS SPECIFIED. SYSX TEM DOES NOT HAVE THIS DEVICE AVAILABLE. BLOCK NOT WRITTX EN.'	
	PUT	PRINT,WORKA	
	MVI	MSIGNAL+1,X'FF'	
	B	CL1	
JA&SYSNDX	MVC	WORKA(133),BLANK	
	STM	0,15,SRF	
	MVC	WORKA+3(107),ERRMES	

SMEMORY (CONT.)

```

MVC  WORKA+60(5),=C'WRITE'
L     5,CHECK
SR    5,4
A     5,=F'1'
BE    KA&SYSNDX
O     5,=F'240'
B     LA&SYSNDX
KA&SYSNDX L     5,=F'61936'
LA&SYSNDX LR    4,5
      ST     4,DUM1
MVC  WORKA+70(4),DUM1
MVC  WORKA+101(7),=C'WRITTEN'
PUT  PRINT,WORKA
MVI  MSIGNAL+1,X'FF'
B     CL1
MA&SYSNDX MVC  WORKA(133),BLANK
MVC  WORKA+3(107),ERRMES
MVC  WORKA+60(5),=C'READ '
L     5,CHECK
SR    5,4
A     5,=F'1'
C     5,CHECK
BE    NA&SYSNDX
O     5,=F'240'
B     OA&SYSNDX
NA&SYSNDX L     5,=F'61936'
OA&SYSNDX LR    4,5
      ST     4,DUM1
MVC  WORKA+70(4),DUM1
MVC  WORKA+101(7),=C' READ '
PUT  PRINT,WORKA
MVI  MSIGNAL+1,X'FF'
B     CL1
ERRMES DC    C' A PERMANENT ERROR CONDUCED WHILE ATTEMPTIX
      NG TO THE TH TRACK ON DISK. DISK NOT .'
PA&SYSNDX MVC  WORKA(133),BLANK
MVC  WORKA+1(92),=C' DEVICE TYPE >0 WAS SPECIFIED. SYSTEMX
      DOES NOT HAVE THIS DEVICE AVAILABLE. BLOCK NOT READ.'
PUT  PRINT,WORKA
MVI  MSIGNAL+1,X'FF'
B     CL1
DS    OF
CURDEV DS    F
CURNUM DS    F
CURTRK DS    F
CURCYL DS    F
NXTDEV DS    F
NXTNUM DS    F
NXTTRK DS    F

```

SMEMORY (CONT.)

```

NXTCYL      DS      F
MYBRYY      DS      F
RDKEY       DS      F
WTKEY       DS      F
RDREF       DS      2F
WTREF       DS      2F
QA&SYSNDX   NI      MSIGNAL,X'7E'
MVC         CORD1(&ADDL),EMPTY+&ADDL
TIME        BIN
S           0,FORMAT
A           0,ACESTIME
ST          0,ACESTIME
L           ERR,SRMEMORY      *LOAD BRANCH ADDRESS IN TBADD.
BR          ERR                *BRANCH TO TBADD.
SAVEFM      CLI     MSIGNAL+1,X'FF'
BE          PMARRAY
L           BRYY,SAVEYY
TM         MSIGNAL,X'01'
BZ         PMARRAY
APART      CURRENT,0,1,2,3,4
STM        0,3,CURDEV        *BEGIN TO WRITE ON DISK.
C          0,BWX
BNL        RA&SYSNDX
C          1,LSX
BL         SA&SYSNDX
RA&SYSNDX  MVC     WORKA(102),BLANK
MVC        WORKA+5(102),=C'PROGRAMMING ERROR OF CURRENT IN SSEARCX
           H COMPONENT. START BREEDING THE FILE SYSTEM FROM SCRATCX
           H AGAIN.'
PUT        PRINT,WORKA
B          CLOSE                *EXIT
SA&SYSNDX  SR      2,2
L          4,=F'&NTRKS.'
L          3,CURCYL
M          2,CHECK
A          3,CURTRK            *COMPUTED RELATIVE ADDRESS ON DISK.
S          3,=F'1'
LR         5,3
SLL        3,8
O          3,=F'1'
TA&SYSNDX  SLL     3,8
ST         3,WTKEY            *NOW HAVE KEY FOR THIS BLOCK.
MVC        WTREF(3),WTKEY:*RELATIVE ADDRESS MOVED FOR WRITE USE.
ST         BRYY,MYBRYY
TM         MSIGNAL,X'02'
BZ         VA&SYSNDX          *WANT TO UPDATE BLOCK ON DISK.
ST         BRYY,FNWBLK+12    *WANT TO ADD A NEW BLOCK TO DISK.
WRITE     FNWBLK,DA,GLOBAL1,WORKM,'S',WTKEY,WTREF
A          5,=F'1'

```

SMEMORY (CONT.)

```

LR      3,5
SLL     3,8
O       3,=F'1'
WAIT    1,ECB=FNWBLK          *WAIT FOR END OF WRITE TRANSFER.
CLI     FNWBLK+1,X'00'
BNE     FB&SYSNDX            *WRITE OPERATION NOT SUCCESSFUL.
UA&SYSNDX L    BRYY,MYBRYY
A       BRYY,=F'&TRKL.'
BCT     4,TA&SYSNDX          *MORE WRITES TO DO.
B       IB&SYSNDX            *FINISHED WRITING ON DISK.
VA&SYSNDX ST    BRYY,DCDBLK+12
WRITE   DCDBLK,DK,GLOBAL1,WORKM,'S',WTKEY,WTREF
A       5,=F'1'
LR      3,5
SLL     3,8
O       3,=F'1'
WAIT    1,ECB=DODBLK
CLI     DODBLK+1,X'00'
BNE     FB&SYSNDX            *WRITE OLD BLOCK NOT SUCCESSFUL
B       UA&SYSNDX            *MORE WRITES TO DO.
PMARRAY XC    WORKA(133),WORKA
MVC     WORKA(54),=C'        THE STATUS OF THE AUXILLIARY AND MAIX
        N FILES ARE:--'
PUT     PRINT,WORKA
XC      WORKA(54),WORKA
MVC     WORKA(21),=C'        AUXILLIARY FILE:'
PUT     PRINT,WORKA
COMPARE EMPTY,EMPTY+&ADDL,&ADDL
BE      WA&SYSNDX
MVC     EMPTY(&ADDL),EMPTY+&ADDL
WA&SYSNDX APART  EMPTY,0,1,2,3,4
STM     0,4,C1
HEXPC   EMPTY,EMPTY,=F'&ADDL'
DECPC   DEVICE,C1,TYPE
DECPC   DEVICE,C2,NUMBER
DECPC   NUMBER,C3,TRACK
DECPC   NUMBER,C4,CYLINDER
DECPC   RFMADDRESS,C5,BYTES
XC      WORKA(133),WORKA
MVC     WORKA(15),=C'        MAIN FILE:'
PUT     PRINT,WORKA
HEXPC   BULK,BULK,=F'&ADDL'
APART   BULK,0,1,2,3,4
STM     0,4,C1
DECPC   DEVICE,C1,TYPE
DECPC   DEVICE,C2,NUMBER
DECPC   NUMBER,C3,TRACK
DECPC   NUMBER,C4,CYLINDER
DECPC   RFMADDRESS,C5,BYTES

```

SMEMORY (CONT.)

```

CLI    MSIGNAL+1,X'FF'
BNE    AB&SYSNDX
MVC    EMPTY(&ADDL),CORD1
MVC    WORKA(133),BLANK
MVC    WORKA+10(115),=C' THE LAST           JOB WAS NOT EXECUX
      TED.START IT AGAIN.THE FILE SYSTEM WAS NOT DAMAGED.THE UX
      NEXECUTED JOB-LIST IS:'
X&SYSNDX CLI    MODE+3,X'00'
      BNE    XA&SYSNDX                       *STORAGE OR UPDATING.
      MVC    WORKA+20(9),=C'RETRIEVAL'
      B     ZA&SYSNDX
YA&SYSNDX CLI    MODE+3,X'01'
      BNE    YA&SYSNDX                       *UPDATING.
      MVC    WORKA+20(7),=C'STORAGE'
      B     ZA&SYSNDX
ZA&SYSNDX MVC    WORKA+20(8),=C'UPDATING'
      PUT    PRINT,WORKA
      L     IR6,A&JBLIST
      LR    IR5,IR6
      A     IR5,JLL
      A     IR5,=F'22'
      PRINT B,C(IR6),O(IR5)
AB&SYSNDX L     IR1,AYY
      CLC   O(&ADDL,IR1),ZERO
      BNE   CLOSE                             *M-ARRAY WAS NOT CHANGED.
      MVC   WORKA(133),BLANK
      MVC   WORKA+5(120),=C' BEFORE BEGINNING THE NEXT JOB WITH THX
      IS RETRIEVAL SYSTEM INSERT THE NEWLY PUNCHED M-ARRAY AT X
      THE HEAD OF THE DATA DECK.'
BB&SYSNDX L     IR6,=F'10'
      PUT   PRINT,WORKA
      BCT   IR6,BB&SYSNDX
      MVC   O(&ADDL,IR1),EMPTY
      MVC   ASTERIK+27(21),=C'BEGIN READING M-ARRAY'
      MVI   ASTERIK,C'*'
      PUT   PUNCH,ASTERIK
      MVI   ASTERIK,C'0'
      PUT   PRINT,ASTERIK
      MVC   ASTERIK+27(21),ASTERIK+60
      TIME  BIN
      ST    O,WORKA
      ST    O,DATEC
      MVC   WORKA+4(80),BLANK+1
      PUT   PUNCH,WORKA
      DECPC CURRENTDATE,DATEC,
      LR    IR2,IR1
      A     IR2,=F'&ADDL'
      MVC   O(&ADDL.,IR2),BULK
      A     IR2,=F'&ADDL'

```

SMEMORY (CONT.)

```

CB&SYSNDX  L      1,=F'6'
            MVC    DUMX(4),O(IR2)
            A      IR2,DUMX
            A      IR2,=F'&ADDL'
            BCT    1,CB&SYSNDX
            A      IR2,=F'&ADDL'
            S      IR2,=F'4'
            PRINT  B,C(IR1),O(IR2)
            MVC    DUM1+27(4),DATEC
            MVC    DUM1(27),=C'* END OF THE M ARRAY DATED
            LR     IR5,IR1
            LR     IR6,IR2
            A      IR6,=F'4'
            S      IR6,IR1
DB&SYSNDX  MVC    WORKA(80),BLANK
            C      IR6,=F'80'
            BNL    EB&SYSNDX
            S      IR6,=F'1'
            STC    IR6,EB&SYSNDX+1
EB&SYSNDX  MVC    WORKA(80),O(IR5)
            PUT    PUNCH,WORKA
            A      IR5,=F'80'
            S      IR6,=F'80'
            BH     DB&SYSNDX
            MVI    EB&SYSNDX+1,X'4F'
            A      5,=F'4'
            MVC    O(42,5),=C' *****X
            *
            PUT    PUNCH,DUM1
            MVI    DUM1,C'0'
            B      CLOSE
FB&SYSNDX  MVC    WORKA(133),BLANK
            MVC    WORKA+3(107),ERRMES
            MVC    WORKA+60(6),=C' WRITE'
            L      5,=F'10'
            SR     5,4
            A      5,=F'1'
            C      5,=F'10'
            BE     GB&SYSNDX
            O      5,=F'240'
            B      HB&SYSNDX
GB&SYSNDX  L      5,=F'61936'
HB&SYSNDX  LR     4,5
            ST     4,DUM1
            MVC    WORKA+70(4),DUM1
            MVC    WORKA+101(8),=C' WRITTEN'
            PUT    PRINT,WORKA
            MVI    MSIGNAL+1,X'FF'
            B      PMARRAY
IB&SYSNDX  MVC    CURRENT(&ADDL),ZERO
            NI     MSIGNAL,X'40'
            B      PMARRAY
JB&SYSNDX  DCBMEM
            B      AA&SYSNDX
            LTORG
            MEND

```

*M-ARRAY WAS CHANGED.

*NOW TERMINATE THE JOB.

*NOW TERMINATE THE JOB.

*****SMEMORY *****

SOUTPUT

*****SOUTPUT*****

MACRO

SOUTPUT &NAME, &UR, &RR, &DUMMY

LEVEL=1

*****SOUTPUT*****LEVEL=1

*---SOUTPUT HANDLES ALL OUTPUT FOR THE SOLID SYSTEM.

*---&NAME EQUALS OUTPUT.

*---&UR IS THE USING REGISTER B.

*---&RR IS THE RETURN REGISTER.

*---&DUMMY EQUALS SOLID (DXTENDED FORM) OR DUMMY (OVERLAYS).

```

-----
          USING OUTPUT, &UR                      *PRINTED OUTPUT FROM SOLID.
OUTPUT   DS      OF
          STSR   &UR, &RR, &DUMMY
          L      4, SBRY
          CLI    MODE+3, X'00'
          BNE   AA&SYSNDX
          L      1, PARM
          B     BA&SYSNDX
AA&SYSNDX L      1, 0(4)
BA&SYSNDX SRL   1, 8
          ST     1, JII
          L      BRY, RLBRY
          LR     BRY, BRY
          A      BRY, =F'256'
          A      BRY, LLENGTH
          A      BRY, =F'1'
          LMOVE JII+2, 0(BRY), 0(BRY)
          L      0, LLENGTH
          AR     1, 0
          CLI    MODE+3, X'00'
          BNE   CA&SYSNDX
          STH   1, DUM1
          L      BRY, RLBRY
          LR     BRY, BRY
          A      BRY, =F'1'
          LMOVE DUM1, 0(BRY), 0(BRY)
CA&SYSNDX B      DA&SYSNDX
          A      0, =F'1'
          A      1, =F'1'
DA&SYSNDX ST     1, JII
          SLL   1, 8
          IC    1, PARM+3
          ST     1, PARM
          SLL   0, 8
          AL    0, SOS
          ST     0, SOS
          PPCORDS EA&SYSNDX
EA&SYSNDX TESTL OUTPUTC
          LTRG

```

SOUTPUT (CONT.)

```

DS      OF
USING  OUTPUTC,&UR
OUTPUTC STSR  &UR,&ERR,&DUMMY
        L    4, LBRY
        L    5, JII
        AR   5, 4
        CLI  NOSEG+3, X'00'
        BNE  FA&SYSNDX
        TIME BIN
        S    0, STIME
        ST   0, STIME
        DECPC STRING, STIME, SECS
        L    0, OUTPXT
        C    0, =F'2'
        BNH  FA&SYSNDX
        S    0, =F'1'
        ST   0, OUTPXT
        B    RA&SYSNDX
FA&SYSNDX DECPC TRANSFER, JII, BYTES
        MVC  WORKA(54), =C'1
                                DECOMPRESSED INFORMATION TRANSFEREX
                                D TO PRINTER.'
        CLI  MODE+3, X'00'
        BE   GA&SYSNDX
        MVI  WORKA+7, C' '
        MVI  WORKA+8, C' '
GA&SYSNDX CLI  OUTPXT+3, X'02'
        BE   IA&SYSNDX
        MVI  WORKA, C' '
        LA   2, WORKA
        A    2, =F'46'
        MVC  0(8, 2), =C'
        CLI  OUTPXT+3, X'00'
        BE   HA&SYSNDX
        MVC  0(6, 2), =C'CARDS.'
                                *TAPE
        B    IA&SYSNDX
HA&SYSNDX MVC  0(5, 2), =C'TAPE.'
IA&SYSNDX PUT  PRINT, WORKA
        L    0, OUTPXT
        CLI  OUTPXT+3, X'02'
        BNE  JA&SYSNDX
                                *TAPE OR CARDS.
        PUT  PRINT, BLANK
        PUT  PRINT, BLANK
        MVC  OUTPXT(4), ROUTPXT
        CLI  MODE+3, X'00'
        BE   LA&SYSNDX
        PRINT B, 0(4), 0(5)
        SKIPP 1
                                *AFTER PRINTING COMP. INFO.
        B    RA&SYSNDX
JA&SYSNDX C    0, =F'1'

```

SOUTPUT (CONT.)

```

BE      KA&SYSNDX
WRITE  O(4),JII
KA&SYSNDX B      RA&SYSNDX
MVI    WORKA,C'*'
PUT    PUNCH,WORKA
PUNXH  X,O(4),O(5)
LA&SYSNDX B      RA&SYSNDX
DECPC  STATUS,MODE,MODE
DECPC  NEXTJOB,LJ,LJ
L      4,PARM
SRL    4,8
ST     4,JII
LM     2,7,ZERO
IC     2,PARM+3
ST     2,NV
DECPC  NUMBER,NV,SUBSTRINGS
PUT    PRINT,BLANK
MA&SYSNDX L      BRYY,LBRYY
L      4,SOS(3)
STC    4,NDR
NI     NDR,X'FO'
PRINT  B,SOS(3),SOS(3)
IC     5,NDR
SRL    5,4
SRL    4,8
LR     BRYY,BRYY
AR     BRYY,4
S      BRYY,=F'4'
C      5,=F'4'
BNE    NA&SYSNDX
PRINT  B,O(BRYY),O(BRYY)
NA&SYSNDX B      QA&SYSNDX
C      5,=F'1'
BNE    OA&SYSNDX
PRINT  A,O(BRYY),O(BRYY)
OA&SYSNDX B      QA&SYSNDX
C      5,=F'2'
BH     PA&SYSNDX
PRINT  I,O(BRYY),O(BRYY)
PA&SYSNDX B      QA&SYSNDX
QA&SYSNDX PRINT  E,O(BRYY),O(BRYY)
A      BRYY,=F'4'
A      3,=F'4'
SKIPP  1
RA&SYSNDX BCT    2,MA&SYSNDX
COPAKEND
LTORG
MEND

```

*3--- STRING COUNTER.

*5-- FORMAT CODE.

*4-- CONTAINS JI.

*****SOUTPUT*****

SPRINT

```

*****SPRINT *****
MACRO
  SPRINT &NAME,&UR,&RR,&DUMMY                                GLOBAL
*****SPRINT *****SPRINT *****GLOBAL
*---SPRINT IS THE PRINT COMPONENT OF THE SOLID SYSTEM.
*---&NAME EQUALS TO PRINT.
*---&UR IS THE USING REGISTER 9.
*---&RR IS THE RETURN REGISTER 1.
*---&DUMMY EQUALS TO SOLID (EXTENDED FORM) OR DUMMY (OVERLAYS).
*-----
      USING PRINT,&UR
PRINT  DS      OD
      STSR   &UR,&RR,&DUMMY
      STM    BRY,Y,BRY,WCD
      ST     &RR.,SRRPRINT
      PUT    PRINT,DASH
      L      1,=F'8'
      L      6,=F'21'
AA&SYSNDX L      4,ZERO
      SLDL   4,4
      IC     7,HEXL(4)
      STG    7,WORKA(6)
      A      6,=F'1'
      BCT    1,AA&SYSNDX
      PUT    PRINT,WORKA
      MVC    WORKA(133),BLANK
      CR     2,3
      BNH    CA&SYSNDX
      MVC    WORKA(18),ERRORP
BA&SYSNDX PUT    PRINT,WORKA
      PUT    PRINT,DASH
      LM     BRY,Y,BRY,WCD
      L      &RR.,SRRPRINT
      BR     &RR
SRRPRINT DS      F
CA&SYSNDX C      2,=F'15'
      BH     KA&SYSNDX
      C      2,=F'14'
      BNE    EA&SYSNDX
      LA     2,WCD
      C      3,=F'15'
      LA     3,WCD+4
      BE     KA&SYSNDX
DA&SYSNDX LR     3,2
      B      KA&SYSNDX
EA&SYSNDX C      2,=F'15'
      BL     FA&SYSNDX
      LA     2,WCD+4
      B      DA&SYSNDX
*RESTORED REGISTERS FOR YY AND Y.
*EXIT FROM SUBROUTINE.

```

SPRINT (CONT.)

FA&SYSNDX	SR	3,2	
	LA	0,SRF	
	C	2,ZERO	
	BE	HA&SYSNDX	
GA&SYSNDX	A	0,=F'4'	
	BCT	2,GA&SYSNDX	
HA&SYSNDX	LR	2,0	
	C	3,ZERO	
	BE	JA&SYSNDX	
IA&SYSNDX	A	0,=F'4'	
	BCT	3,IA&SYSNDX	
JA&SYSNDX	LR	3,0	
KA&SYSNDX	LM	0,1,ZERO	
	L	4,ZERO	
	IC	0,BLANK	
LA&SYSNDX	IC	1,FORMAT(4)	
	CR	0,1	
	BE	MA&SYSNDX	
	A	4,=F'1'	
	B	LA&SYSNDX	
MA&SYSNDX	LR	1,3	
	SR	1,2	
	A	1,=F'4'	
	L	0,ZERO	
	D	0,=F'4'	
	CR	1,4	
	BNH	OA&SYSNDX	*FORMAT ARE OK.
	C	4,=F'1'	
	BE	NA&SYSNDX	*REPEAT FORMAT.
	MVC	WORKA(133),BLANK	
	MVC	WORKA(14),ERRORF	
	B	BA&SYSNDX	
NA&SYSNDX	IC	0,FORMAT	
	STC	0,FORMAT-1	
	MVC	FORMAT(32),FORMAT-1	
OA&SYSNDX	LR	7,2	
	PUT	PRINT,BLANK	
	ST	3,ADDRESS	
	MVC	DUMF(32),FORMAT	
PA&SYSNDX	L	2,=F'4'	*LOOP ON LINES BEGINS HERE.
	MVC	WORKA(133),BLANK	
	MVC	FORMAT(32),DUMF	
QA&SYSNDX	L	0,=F'16'	
RA&SYSNDX	CLI	FORMAT,C'A'	
	BE	SA&SYSNDX	
	L	5,0(7,0)	
	B	VA&SYSNDX	
SA&SYSNDX	MVC	DUM1(4),0(7)	
	L	5,DUM1	

SPRINT (CONT.)

	ST	5,WORKA(2)	
	A	7,=F'4'	
	C	7,ADDRESS	
	BH	BA&SYSNDX	*JOB FINISHED.
	CLI	FORMAT+1,C'A'	
	BE	TA&SYSNDX	
	L	0,=F'4'	
TA&SYSNDX	A	2,=F'12'	
	A	2,=F'4'	
	C	2,=F'132'	
	BE	UA&SYSNDX	*LINE FINISHED.
	S	0,=F'4'	
	BH	RA&SYSNDX	*FIELD NOT FINISHED.
	MVC	FORMAT(32),FORMAT+1	
	B	QA&SYSNDX	*FIELD FINISHED.
UA&SYSNDX	PUT	PRINT,WORKA	*PRINT FINISHED LINE.
	B	PA&SYSNDX	
VA&SYSNDX	C	0,=F'16'	
	BE	WA&SYSNDX	
	AR	2,0	
WA&SYSNDX	CLI	FORMAT,C'B'	
	BNE	ZA&SYSNDX	*I,E, OR F FORMATS.
	A	2,=F'2'	*TWO LEFT BLANKS IN FIELD.
	L	0,=F'8'	
XA&SYSNDX	L	4,=F'0'	
	SLDL	4,4	
	IC	3,HEXL(4)	
	STC	3,WORKA(2)	
	A	2,=F'1'	
	BCT	0,XA&SYSNDX	
	A	2,=F'6'	*LINED TO NEXT FIELD.
YA&SYSNDX	A	7,=F'4'	
	C	7,ADDRESS	
	BH	BA&SYSNDX	*JOB FINISHED.
	MVC	FORMAT(32),FORMAT+1	
	C	2,=F'132'	
	BE	UA&SYSNDX	*LINE FINISHED.
	B	QA&SYSNDX	*TO NEXT FIELD.
ZA&SYSNDX	CLI	FORMAT,C'I'	
	BNE	IB&SYSNDX	*E OR F.
	CVD	5,DUM5	
	LM	4,5,DUM5	
	SLDL	4,16	*LEFT ADJUSTED PACKED DECIMAL.
	STM	4,5,DUM5	
	UNPK	C1(12),DUM5(6)	
	NI	C1+11,X'OF'	
	OI	C1+11,X'FO'	
	MVC	C1(11),C1+1	
	L	4,ZERO	

SPRINT (CONT.)

AB&SYSNDX	L	3,=F'9'	
	CLI	C1,C'0'	
	BNE	CB&SYSNDX	*NOT ZERO
	A	4,=F'1'	
BB&SYSNDX	MVI	C1,C'0'	
	LH	6,AB&SYSNDX+2	
	AL	6,=F'1'	
	STH	6,AB&SYSNDX+2	
	LH	6,BB&SYSNDX+2	
	AL	6,=F'1'	
	STH	6,BB&SYSNDX+2	
	BCT	3,AB&SYSNDX	
CB&SYSNDX	LH	6,AB&SYSNDX+2	
	SLR	6,4	
	STH	6,AB&SYSNDX+2	
	LH	6,BB&SYSNDX+2	
	SLR	6,4	
	STH	6,BB&SYSNDX+2	
	C	4,ZERO	
	BE	DB&SYSNDX	
	S	4,=F'1'	
DB&SYSNDX	L	6,0(7,0)	
	C	6,ZERO	
	BNL	EB&SYSNDX	
	IC	6,=C'-'	
	STC	6,C1(4)	
EB&SYSNDX	IC	4,=C'0'	
	L	3,=F'5'	
	L	6,=F'11'	
FB&SYSNDX	STC	4,C1(6)	
	A	6,=F'1'	
	BCT	3,FB&SYSNDX	
GB&SYSNDX	A	2,=F'2'	*TWO BLANKS IN LEFT FIELD.
	LH	3,HB&SYSNDX+2	
	ALR	3,2	
	STH	3,HB&SYSNDX+2	
	L	6,=F'14'	
	STC	6,HB&SYSNDX+1	
HB&SYSNDX	MVC	WORKA(99),C1	
	SLR	3,2	
	STH	3,HB&SYSNDX+2	
	AR	2,6	
	B	YA&SYSNDX	
IB&SYSNDX	C	5,ZERO	
	MVC	DUM5(32),DZERO	
	BE	MB&SYSNDX	*FL.PT. IS ZERO.
	L	4,ZERO	
	SLL	5,1	
	SLDL	4,7	

SPRINT (CONT.)

	C	4,=F'58'	
	BL	NB&SYSNDX	*LIST AS HEXADECIMAL.
	C	4,=F'72'	
	BH	NB&SYSNDX	*LIST AS HEXADECIMAL.
	S	4,=F'64'	
	LR	3,4	
	L	4,ZERO	
	CR	3,4	
	BE	PB&SYSNDX	*HEX.DEC.FRACTION ONLY IN GR5.
	BL	KB&SYSNDX	*NEGATIVE EXPONENT.
JB&SYSNDX	SLDL	4,4	
	BCT	3,JB&SYSNDX	
	B	OB&SYSNDX	*CONVERT THE NO IN 4-5.
KB&SYSNDX	LPR	3,3	
LB&SYSNDX	SRDL	4,4	
	BCT	3,LB&SYSNDX	
	B	PB&SYSNDX	*FRACTION IN GR5 ONLY.
MB&SYSNDX	MVC	C1(15),=C'+0.0000000	
	B	GB&SYSNDX	*0.0 TAKEN CARE OF.
NB&SYSNDX	MVI	FORMAT,C'B'	
	L	5,0(7,0)	
	B	WA&SYSNDX	*HEXADECIMAL LISTING TAKEN CARE OF.
OB&SYSNDX	CVD	4,DUM5	
	UNPK	C1(16),DUM5(8)	
	NI	C1+15,X'OF'	
	OI	C1+15,X'FO'	
	MVC	DUM5(16),C1	
PB&SYSNDX	L	1,=F'16'	
	LH	4,QB&SYSNDX+2	
	ALR	4,1	
	STH	4,QB&SYSNDX+2	
QB&SYSNDX	MVC	DUM5(6),ZERO	
	MVI	DUM5+21,X'OF'	
	SLR	4,1	
	STH	4,QB&SYSNDX+2	
	L	6,ZERO	
	L	3,=F'8'	
	LH	4,TB&SYSNDX+2	
	ALR	4,1	
	STH	4,TB&SYSNDX+2	
RB&SYSNDX	L	4,ZERO	
	SLDL	4,4	
	C	4,ZERO	
	BE	UB&SYSNDX	*ZERO
	CVD	4,DUMX	
	MVC	C3(8),DUMX	
	MVC	C2(4),ZERO	
	LH	4,SB&SYSNDX+4	
	ALR	4,6	

SPRINT (CONT.)

SB&SYSNDX	STH	4, SB&SYSNDX+4
	MP	C2(12), CONSTANT(6)
	MVC	C1(6), C2+6
	SLR	4, 6
TB&SYSNDX UB&SYSNDX	STH	4, SB&SYSNDX+4
	AP	DUM5(6), C1(6)
	A	6, =F'6'
	BCT	3, RB&SYSNDX
	LH	4, TB&SYSNDX+2
	SLR	4, 1
	STH	4, TB&SYSNDX+2
	MVC	C1(6), DUM5+16
	LH	4, VB&SYSNDX+2
	ALR	4, 1
VB&SYSNDX	STH	4, VB&SYSNDX+2
	UNPK	DUM5(12), C1(6)
	STH	4, WB&SYSNDX+2
	AL	4, =F'1'
WB&SYSNDX	STH	4, WB&SYSNDX+4
	SL	4, =F'1'
	MVC	DUM5(10), DUM5
	MVI	DUM5+26, C'0'
	SLR	4, 1
	STH	4, VB&SYSNDX+2
	NI	DUM5+27, X'OF'
	OI	DUM5+27, X'FO'
	L	4, ZERO
	L	6, =F'28'
XB&SYSNDX	MVC	C1(28), DUM5
	CLI	C1, C'0'
	BNE	YB&SYSNDX
	A	4, =F'1'
	LH	5, XB&SYSNDX+2
	AL	5, =F'1'
	STH	5, XB&SYSNDX+2
	BCT	6, XB&SYSNDX
YB&SYSNDX	LH	5, XB&SYSNDX+2
	SLR	5, 4
	STH	5, XB&SYSNDX+2
	MVC	DUM5(15), ZERO
	MVI	DUM5, C'+'
	L	3, =F'7'
	L	5, 0(7, 0)
	C	5, ZERO
	BNL	ZB&SYSNDX
	MVI	DUM5, C'-'
ZB&SYSNDX	MVI	DUM5+1, C'0'
	MVI	DUM5+2, C'.'
	L	5, =F'3'

*TAG ON DUM5.

SPRINT (CONT.)

```

C      4,=F'16'
BH     BC&SYSNDX
MVI    DUM5+11,C'+
L      6,=F'16'
SR     6,4
AC&SYSNDX CVD  6,DUM1
MVC    DUM1(2),DUM1+6
UNPK   DUM2(4),DUM1(2)
NI     DUM2+3,X'OF'
OI     DUM2+3,X'FO'
MVI    DUM5+10,C'E'
L      6,BLANK
ST     6,DUM5+12
LH     6,DUM2+2
STH    6,DUM5+12
B      CC&SYSNDX
BC&SYSNDX LR  6,4
S      6,=F'16'
MVI    DUM5+11,C'-
B      AC&SYSNDX
CC&SYSNDX L   6,=F'28'
SR     6,4
DC&SYSNDX IC  1,C1(4)
STC    1,DUM5(5)
A      4,=F'1'
A      5,=F'1'
BCT    6,EC&SYSNDX
B      FC&SYSNDX
EC&SYSNDX BCT  3,DC&SYSNDX
FC&SYSNDX MVC  C1(15),DUM5
B      GB&SYSNDX
CONSTANT DC  X'062500'
DC     X'00000F'
DC     X'003906'
DC     X'25000F'
DC     X'000244'
DC     X'14062F'
DC     X'000015'
DC     X'25878F'
DC     X'000000'
DC     X'95367F'
DC     X'000000'
DC     X'05950F'
DC     X'000000'
DC     X'00372F'
DC     X'000000'
DC     X'00023F'
LTORG
MEND

```

*NEGATIVE EXPONENT.

*****SPRINT*****

SPUNXH (CONT.)

	C	2,=F'14'	
	BNE	FA&SYSNDX	
	LA	2,WCD	
	C	3,=F'15'	
	LA	3,WCD+4	
	BE	LA&SYSNDX	
EA&SYSNDX	LR	3,2	
	B	LA&SYSNDX	
FA&SYSNDX	C	2,=F'15'	
	BL	GA&SYSNDX	
	LA	2,WCD+4	
	B	EA&SYSNDX	
GA&SYSNDX	SR	3,2	
	LA	0,SRF	
	C	2,ZERO	
	BE	IA&SYSNDX	
HA&SYSNDX	A	0,=F'4'	
	BCT	2,HA&SYSNDX	
IA&SYSNDX	LR	2,0	
	C	3,ZERO	
	BE	KA&SYSNDX	
JA&SYSNDX	A	0,=F'4'	
	BCT	3,JA&SYSNDX	
KA&SYSNDX	LR	3,0	
LA&SYSNDX	LM	0,1,ZERO	
	L	4,ZERO	
	IC	0,BLANK	
MA&SYSNDX	IC	1,FORMAT(4)	
	CR	0,1	
	BE	NA&SYSNDX	
	A	4,=F'1'	
	B	MA&SYSNDX	
NA&SYSNDX	LR	1,3	
	SR	1,2	
	A	1,=F'4'	
	L	0,ZERO	
	D	0,=F'4'	
	CR	1,4	
	BNH	PA&SYSNDX	*FORMAT ARE OK.
	C	4,=F'1'	
	BE	OA&SYSNDX	*REPEAT FORMAT.
	MVC	WORKA(133),BLANK	
	MVC	WORKA(14),ERRORF	
	B	BA&SYSNDX	
OA&SYSNDX	IC	0,FORMAT	
	STC	0,FORMAT-1	
	MVC	FORMAT(32),FORMAT-1	
PA&SYSNDX	LR	7,2	
	ST	3,ADDRESS	

SPUNXH (CONT.)

	BE	YA&SYSNDX	
	A	2,=F'15'	
	S	7,=F'4'	
	B	FB&SYSNDX	
YA&SYSNDX	A	2,=F'4'	
	C	2,=F'80'	
	BE	BB&SYSNDX	*LINE FINISHED.
	S	0,=F'4'	
	BH	WA&SYSNDX	*FIELD NOT FINISHED.
	MVC	FORMAT(32),FORMAT+1	
	B	VA&SYSNDX	*FIELD FINISHED.
ZA&SYSNDX	A	2,=F'4'	
	C	2,=F'80'	
	BE	BB&SYSNDX	
	S	7,=F'4'	
	C	7, ADDRESS	
	BE	AB&SYSNDX	
	S	2,=F'4'	
AB&SYSNDX	IC	7,=X'E0'	
	A	2,=F'1'	
	STC	7,WORKA(2)	
	A	2,=F'1'	
	B	BA&SYSNDX	
BB&SYSNDX	PUT	PUNCH,WORKA	*PUNCH FINISHED LINE.
	B	QA&SYSNDX	
CB&SYSNDX	C	0,=F'16'	
	BE	DB&SYSNDX	
	AR	2,0	
DB&SYSNDX	CLI	FORMAT,C'B'	
	BNE	GB&SYSNDX	*I,E, OR F FORMATS.
	A	2,=F'6'	*SIX LEFT BLANKS IN THE FIELD.
	L	0,=F'8'	
EB&SYSNDX	L	4,=F'0'	
	SLDL	4,4	
	IC	3,HEXL(4)	
	STC	3,WORKA(2)	
	A	2,=F'1'	
	BCT	0,EB&SYSNDX	
	A	2,=F'1'	*LINED TO NEXT FIELD.
FB&SYSNDX	A	7,=F'4'	
	C	7, ADDRESS	
	BH	BA&SYSNDX	*JOB FINISHED.
	MVC	FORMAT(32),FORMAT+1	
	C	2,=F'75'	
	BNL	BB&SYSNDX	*LINE FINISHED.
	B	VA&SYSNDX	*TO NEXT FIELD.
GB&SYSNDX	CLI	FORMAT,C'I'	
	BNE	PB&SYSNDX	*E OR F.
	CVD	5,DUM5	

SPUNXH (CONT.)

	LM	4,5,DUM5	
	SLDL	4,16	*LEFT ADJUSTED PACKED DECIMAL.
	STM	4,5,DUM5	
	UNPK	C1(12),DUM5(6)	
	NI	C1+11,X'0F'	
	OI	C1+11,X'F0'	
	MVC	C1(11),C1+1	
	L	4,ZERO	
	L	3,=F'9'	
HB&SYSNDX	CLI	C1,C'0'	
	BNE	JB&SYSNDX	*NOT ZERO
	A	4,=F'1'	
IB&SYSNDX	MVI	C1,C' '	
	LH	6,HB&SYSNDX+2	
	AL	6,=F'1'	
	STH	6,HB&SYSNDX+2	
	LH	6,IB&SYSNDX+2	
	AL	6,=F'1'	
	STH	6,IB&SYSNDX+2	
JB&SYSNDX	BCT	3,HB&SYSNDX	
	LH	6,HB&SYSNDX+2	
	SLR	6,4	
	STH	6,HB&SYSNDX+2	
	LH	6,IB&SYSNDX+2	
	SLR	6,4	
	STH	6,IB&SYSNDX+2	
	C	4,ZERO	
	BE	KB&SYSNDX	
	S	4,=F'1'	
KB&SYSNDX	L	6,0(7,0)	
	C	6,ZERO	
	BNL	LB&SYSNDX	
	IC	6,=C'-'	
LB&SYSNDX	STC	6,C1(4)	
	IC	4,=C' '	
	L	3,=F'5'	
	L	6,=F'11'	
MB&SYSNDX	STC	4,C1(6)	
	A	6,=F'1'	
	BCT	3,MB&SYSNDX	
NB&SYSNDX	LH	3,OB&SYSNDX+2	
	***NO BLANKS IN LEFT FIELD.		
	ALR	3,2	
	STH	3,OB&SYSNDX+2	
	L	6,=F'13'	
	STC	6,OB&SYSNDX+1	
OB&SYSNDX	MVC	WORKA(99),C1	
	SLR	3,2	
	STH	3,OB&SYSNDX+2	

SPUNXH (CONT.)

	A	2,=F'15'	*LINED TO NEXT FIELD.
	B	FB&SYSNDX	
PB&SYSNDX	C	5,ZERO	
	MVC	DUM5(32),DZERO	
	BE	TB&SYSNDX	*FL.PT. IS ZERO.
	L	4,ZERO	
	SLL	5,1	
	SLDL	4,7	
	C	4,=F'58'	
	BL	UB&SYSNDX	*LIST AS HEXADECIMAL.
	C	4,=F'72'	
	BH	UB&SYSNDX	*LIST AS HEXADECIMAL.
	S	4,=F'64'	
	LR	3,4	
	L	4,ZERO	
	CR	3,4	
	BE	WB&SYSNDX	*HEX.DEC.FRACTION ONLY IN GR5.
	BL	RB&SYSNDX	*NEGATIVE EXPONENT.
QB&SYSNDX	SLDL	4,4	
	BCT	3,QB&SYSNDX	
	B	VB&SYSNDX	*CONVERT THE NO IN 4-5.
RB&SYSNDX	LPR	3,3	
SB&SYSNDX	SRDL	4,4	
	BCT	3,SB&SYSNDX	
	B	WB&SYSNDX	*FRACTION IN GR5 ONLY.
TB&SYSNDX	MVC	C1(15),=C'+0.000000	
	B	NB&SYSNDX	*0.0 TAKEN CARE OF.
UB&SYSNDX	MVI	FORMAT,C'B'	
	L	5,0(7,0)	
	B	DB&SYSNDX	*HEXADECIMAL LISTING TAKEN CARE OF.
VB&SYSNDX	CVD	4,DUM5	
	UNPK	C1(16),DUM5(8)	
	NI	C1+15,X'OF'	
	OI	C1+15,X'FO'	
WB&SYSNDX	MVC	DUM5(16),C1	
	L	1,=F'16'	
	LH	4,XB&SYSNDX+2	
	ALR	4,1	
XB&SYSNDX	STH	4,XB&SYSNDX+2	
	MVC	DUM5(6),ZERO	
	MVI	DUM5+21,X'OF'	
	SLR	4,1	
	STH	4,XB&SYSNDX+2	
	L	6,ZERO	
	L	3,=F'8'	
	LH	4,AC&SYSNDX+2	
	ALR	4,1	
YB&SYSNDX	STH	4,AC&SYSNDX+2	
	L	4,ZERO	

SPUNXH (CONT.)

	SLDL	4,4	
	C	4,ZERO	
	BE	BC&SYSNDX	*ZERO
	CVD	4,DUMX	
	MVC	C3(8),DUMX	
	MVC	C2(4),ZERO	
	LH	4,ZB&SYSNDX+4	
	ALR	4,6	
	STH	4,ZB&SYSNDX+4	
ZB&SYSNDX	MP	C2(12),CONSTANZ(6)	
	MVC	C1(6),C2+6	
	SLR	4,6	
	STH	4,ZB&SYSNDX+4	
AC&SYSNDX	AP	DUM5(6),C1(6)	
BC&SYSNDX	A	6,F'6'	
	BCT	3,YB&SYSNDX	
	LH	4,AC&SYSNDX+2	
	SLR	4,1	
	STH	4,AC&SYSNDX+2	
	MVC	C1(6),DUM5+16	
	LH	4,CC&SYSNDX+2	
	ALR	4,1	
	STH	4,CC&SYSNDX+2	
CC&SYSNDX	UNPK	DUM5(12),C1(6)	
	STH	4,DC&SYSNDX+2	
	AL	4,F'1'	
	STH	4,DC&SYSNDX+4	
DC&SYSNDX	SL	4,F'1'	
	MVC	DUM5(10),DUM5	
	MVI	DUM5+26,C'0'	
	SLR	4,1	
	STH	4,CC&SYSNDX+2	
	NI	DUM5+27,X'OF'	
	OI	DUM5+27,X'FO'	
	L	4,ZERO	
	L	6,F'28'	
	MVC	C1(28),DUM5	
EC&SYSNDX	CLI	C1,C'0'	
	BNE	FC&SYSNDX	
	A	4,F'1'	
	LH	5,EC&SYSNDX+2	
	AL	5,F'1'	
	STH	5,EC&SYSNDX+2	
	BCT	6,EC&SYSNDX	
FC&SYSNDX	LH	5,EC&SYSNDX+2	
	SLR	5,4	
	STH	5,EC&SYSNDX+2	
	MVC	DUM5(15),ZERO	
	MVI	DUM5,C'+'	

SPUNXH (CONT.)

	L	3,=F'7'
	L	5,0(7,0)
	C	5,ZERO
	BNL	GC&SYSNDX
	MVI	DUM5,C'-'
GC&SYSNDX	MVI	DUM5+1,C'0'
	MVI	DUM5+2,C'.'
	L	5,=F'3'
	C	4,=F'16'
	BH	IC&SYSNDX
	MVI	DUM5+11,C'+'
	L	6,=F'16'
	SR	6,4
HC&SYSNDX	CVD	6,DUM1
	MVC	DUM1(2),DUM1+6
	UNPK	DUM2(4),DUM1(2)
	NI	DUM2+3,X'0F'
	OI	DUM2+3,X'F0'
	MVI	DUM5+10,C'E'
	L	6,BLANK
	ST	6,DUM5+12
	LH	6,DUM2+2
	STH	6,DUM5+12
	B	JC&SYSNDX
IC&SYSNDX	LR	6,4
	S	6,=F'16'
	MVI	DUM5+11,C'--'
	B	HC&SYSNDX
JC&SYSNDX	L	6,=F'28'
	SR	6,4
KC&SYSNDX	IC	1,C1(4)
	STC	1,DUM5(5)
	A	4,=F'1'
	A	5,=F'1'
	BCT	6,LC&SYSNDX
	B	MC&SYSNDX
LC&SYSNDX	BCT	3,KC&SYSNDX
MC&SYSNDX	MVC	C1(15),DUM5
	B	NB&SYSNDX
CONSTANZ	DC	X'062500'
	DC	X'00000F'
	DC	X'003906'
	DC	X'25000F'
	DC	X'000244'
	DC	X'14062F'
	DC	X'000015'
	DC	X'25878F'
	DC	X'000000'
	DC	X'95367F'

*TAG ON DUM5.

*NEGATIVE EXPONENT.

SQUEEZE

*****SQUEEZE*****

MACRO

&J SQUEEZE &NUMBER,&ADDRESS
 *XX
 *---SQUEEZE IS THE MACRO FOR SQUEEZING OUT ZEROS FROM LABEL ITEMS
 *---&NUMBER IS THE NUMBER OF ITEMS IN THE ENTIRE LABEL
 *---&ADDRESS POINTS TO THE BEGINNING OF BITMAP (I.E. THE BITMAP HEAD)
 *-----

```

&J      STM      0,15,NA&SYSNDX
        LA       1,&ADDRESS          *LOAD ADDRESS OF BEGINNING OF BM
        XC      MA&SYSNDX.(4),MA&SYSNDX *ZERO OUT TEMPORARY STORAGE
        MVC     MA&SYSNDX.+2(2),0(1)  *MOVE IN BITMAP LENGTH + 2
        A       1,MA&SYSNDX          *R1 POINTS TO HEAD OF LABEL
        L       2,&NUMBER
        BCTR    2,0                  *R2 CONTAINS # LABEL ITEMS - 1
        L       5,&NUMBER
        M       4,KLENGTH
        BCTR    5,0                  *CALCULATE INITIAL MVC LENGTH
        LR      3,1
        LA      3,2(3)              *R3 POINTS TO CURRENT LABEL ITEM
        SR      4,4
        IC      4,0(1)
        SLL     4,8
        IC      4,1(1)              *LOAD R4 WITH LABEL LENGTH
        L       6,KLENGTH
        LA      6,2(6)
        CR      4,6
        BE      IA&SYSNDX           *IF ONLY ONE LABEL ITEM--BRANCH
AA&SYSNDX L       15,KLENGTH
        SR      6,6                  *COUNTS HEX ZEROS (UP TO 'KLENGTH')
        LR      7,6                  *R7 HOLDS DISPLACEMENT FROM R3
BA&SYSNDX STC     7,CA&SYSNDX.+3
CA&SYSNDX CLI     0(3),X'00'
        BNE     DA&SYSNDX
DA&SYSNDX LA      6,1(6)           *INCREMENT ZERO COUNT
        LA      7,1(7)           *INCREMENT DISPLACEMENT
        BCT     15,BA&SYSNDX
        C       6,KLENGTH
        BNE     HA&SYSNDX
EA&SYSNDX L       15,KLENGTH
FA&SYSNDX STC     5,GA&SYSNDX.+1
GA&SYSNDX MVC     0(0,3),1(3)
        BCTR    5,0                  *DECREMENT LENGTH OF MVC
        BCT     15,FA&SYSNDX
        S       4,KLENGTH
        S       3,KLENGTH           *KEEP POINTER CONSTANT
HA&SYSNDX A       3,KLENGTH       *INCREMENT POINTER TO LABEL ITEMS
        BCT     2,AA&SYSNDX
IA&SYSNDX L       15,KLENGTH
        DC      X'000000'
        DC      X'05950F'
        DC      X'000000'
        DC      X'00372F'
        DC      X'000000'
        DC      X'00023F'
        LTORG
        MEND

```

*****SPUNXH*****

SQUEEZE (CONT.)

	SR	6,6	*R6 COUNTS HEXADECIMAL ZEROS
JA&SYSNDX	CLI	0(3),X'00'	
	BNE	KA&SYSNDX	
	LA	6,1(6)	*INCREMENT HEXADECIMAL ZERO COUNT
KA&SYSNDX	LA	3,1(3)	*INCREMENT POINTER TO ITEMS
	BCT	15,JA&SYSNDX	
	C	6,KLENGTH	
	BNE	LA&SYSNDX	
	S	4,KLENGTH	*DECREMENT LABEL LENGTH
LA&SYSNDX	STC	4,1(1)	
	SRL	4,8	
	STC	4,0(1)	
	B	OA&SYSNDX	
MA&SYSNDX	DC	F'0'	
NA&SYSNDX	DS	16F	*FOR SAVING REGISTERS 0-15
OA&SYSNDX	LM	0,15,NA&SYSNDX	
	MEND		

*****SQUEEZE*****

SREADC

*****SREADC *****

MACRO

SREADC &NAME,&UR,&RR,&DUMMY

LEVEL=1

*****SREADC *****LEVEL=1

*---SREADC READS SUBSTRING COMMANDS AND DATA FROM CARDS.

*---&NAME EQUALS READC.

*---&UR IS THE USING REGISTER 8.

*---&RR IS THE RETURN REGISTER.

*---&DUMMY EQUALS SOLID (EXTENDED FORM) OR DUMMY (OVERLAYS).

```

      USING READC,&UR
      DS      OF
READC  STSR  &UR,&RR,&DUMMY
      ST      &RR,>SPSAVEBR
      CLI     GATE,X'00'
      BNE     OLDTREAD          *THROUGH BEFORE.
      MVC     NV(4),=F'21'      *DEFAULT SET.
      MVC     SWITCH(4),LEXMODE
      CLI     LEXCON+3,X'00'
      BE      IA&SYSNDX          *NO LEXICON IS TO BE READ.
      MVC     WORKA(133),BLANK
      MVC     WORKA(28),=C'0     PERMANENT CORDS READ: '
      L       1,=F'101'
      LA      2,PCORDS
AA&SYSNDX MVC  0(40,2),ZERO
      A       2,=F'40'
      BCT     1,AA&SYSNDX
BA&SYSNDX PUT  PRINT,WORKA
      GET     MASTER,WORKA
      CLI     WORKA,C'*'
      BE      BA&SYSNDX
      L       3,WORKA
      ST      3,TPCORD
      A       3,=F'1'
      LA      2,PCORDS
      C       3,=F'1'
      BNH     CA&SYSNDX          *TPCORD IS WRONG.
      C       3,=F'201'
CA&SYSNDX BL   DA&SYSNDX
      MVC     WORKA(90),=C'0::::::::::PCORDS ARE NOT RIGHT.X
      THE JOB IS TERMINATED::::::::::X
      :''
      PUT     PRINT,WORKA
      LA      &RR,CL1
      BR      &RR
DA&SYSNDX GET  MASTER,WORKA
      MVC     0(20,2),WORKA
      A       2,=F'20'
      BCT     3,DA&SYSNDX          *FINISHED LDING PCORDS.

```

SREADC (CONT.)

	STM	0,5,C1	*SAVE ALL REGISTERS.
	LA	2,PCORDS+20	
	LR	5,2	
	A	5,=F'20'	*REG. 5 SET TO NEXT PCORD.
	L	1,TPCORD	*REG. 1= NO. OF PCORDS
	S	1,=F'1'	
	LM	3,4,ZERO	
EA&SYSNDX	IC	3,0(2)	
FA&SYSNDX	IC	4,0(5)	
	CR	4,3	
	BH	CA&SYSNDX	*PCORDS FOULED UP--GET OFF MACHINE.
	BE	GA&SYSNDX	*CORD LENGTH IS EQUAL--STILL OKAY.
	LR	2,5	*CORD LENGTH REDUCED--STILL OKAY.
	A	5,=F'20'	
	BCT	1,EA&SYSNDX	
	B	HA&SYSNDX	*GET OUT OF LOOP.
GA&SYSNDX	A	5,=F'20'	
	BCT	1,FA&SYSNDX	*GO CHECK NEXT CORD.
HA&SYSNDX	LM	0,5,C1	*RESET ALL REGISTERS
IA&SYSNDX	L	BRYY,LBRY	
	CLI	INPXT+3,X'00'	
	BNE	JA&SYSNDX	
	CLI	MODE+3,X'00'	
	BE	TA&SYSNDX	
JA&SYSNDX	MVC	RM(4),ZERO	
	CLI	MODE+3,X'00'	
	BNE	KA&SYSNDX	*STORAGE OR UPDATING MODE.
	REID1	*X,0(BRY)	
	B	TA&SYSNDX	
KA&SYSNDX	REID2	1,NV,JI	
	L	2,NV	
	C	2,ZERO	
	BNH	CL1	*EXIT.
	C	2,=F'21'	
	BNL	CL1	*EXIT.
	ST	2,RNV	
	L	3,ZERO	
	CLI	INPXT+3,X'00'	
	BNE	MA&SYSNDX	*CARD ROUTINE.
LA&SYSNDX	REID2	IE,RSOS(3),RLSX(3)	
	L	6,RSOS(3)	
	C	6,=F'1'	
	BH	CL1	*EXIT.
	C	6,=F'-1'	
	BL	CL1	*EXIT
	A	3,=F'4'	
	BCT	2,LA&SYSNDX	
	B	READTAPE	
MA&SYSNDX	REID2	IE,SOS(3),LSX(3)	

SREADC (CONT.)

	L	6, SOS(3)	
	C	6,=F'1'	
	BH	CL1	*EXIT.
	C	6,=F'-1'	
	BL	CL1	*EXIT
	C	6,ZERO	
	BNL	NA&SYSNDX	
	L	6,=F'1'	
	SLL	6,31	
	L	4,LSX(3)	
	C	4,ZERO	
	BE	QA&SYSNDX	*IT IS A FORMAT.
	REID1	*X,0(BRYY)	*X=X'40'(B); SOS<0,LSX NOT ZERO.
	AL	6,=F'64'	*X=X'40'(B); SOS<0,LSX NOT ZERO.
	B	QA&SYSNDX	*X=X'40'(B); SOS<0,LSX NOT ZERO.
NA&SYSNDX	BNE	QA&SYSNDX	
	L	4,LSX(3)	
	C	4,ZERO	
	BE	PA&SYSNDX	*I FORMAT
	REID1	*E,0(BRYY)	*E & F=X'30'(E); SOS=0,LSX #0.
	AL	6,=F'48'	*E & F=X'30'(E); SOS=0,LSX #0.
	B	QA&SYSNDX	*E & F=X'30'(E); SOS=0,LSX #0.
QA&SYSNDX	REID1	*A,0(BRYY)	*A = X'10'(A); SOS<0,LSX=0.
	AL	6,=F'16'	*A = X'10'(A); SOS<0,LSX=0.
	B	QA&SYSNDX	*A = X'10'(A); SOS<0,LSX=0.
PA&SYSNDX	REID1	*I,0(BRYY)	*I = X'20'(I); SOS=0,LSX=0.
	AL	6,=F'32'	*I = X'20'(I); SOS=0,LSX=0.
QA&SYSNDX	L	7,JII	
	ST	6,SOS(3)	
	MVC	NAP(4),DUMF	
	L	6,ZERO	
	D	6,=F'4'	
	LR	5,BRYY	
	L	7,JII	
	AR	BRYY,7	
	C	6,ZERO	
	BE	SA&SYSNDX	*JII(IN IR7) DIVIDES BY FOUR.
	S	6,=F'4'	
	LPR	6,6	
RA&SYSNDX	MVI	0(BRYY),X'00'	
	A	BRYY,=F'1'	
	A	7,=F'1'	
	BCT	6,RA&SYSNDX	
SA&SYSNDX	ST	7,JII	
	SLA	7,8	
	AL	7,SOS(3)	
	ST	7,SOS(3)	*LEFT 3 BYTES OF SOS CONTAINS BYTES/STRING.
	L	6,RM	
	A	6,JII	

SREADC (CONT.)

```

ST      6, RM
S       BRYY, =F'4'
L       7, 0(BRYY)
ST      7, CHECK(3)
A       BRYY, =F'4'
A       3, =F'4'
BCT     2, MA&SYSNDX
L       5, RM
ST      5, JII
ST      5, PSAVINGS      *FOR CALCULATING PERCENT SAVINGS
SLL     5, 8
IC      5, NV+3
ST      5, PARM
TA&SYSNDX SKIPP 1
READTAPE TALE ASTERIK, SEGMENT, 2
OLDTREAD TESTL READCON
LTORG
USING  READCON, 9
DS     OD
READCON STSR  9, 2, 8DUMMY
      CLI  GATE, X'00'
      BNE  BB&SYSNDX
      DECPC INPUT, INPXT, CODE
      CLI  MODE+3, X'00'
      BNE  UA&SYSNDX
      CLI  INPXT+3, X'00'
      BE   VA&SYSNDX
      L    BRYY, LBRYY
      L    0, ZERO
      IC   0, 0(BRYY)
      ST   0, LLENGTH
UA&SYSNDX DECPC LABEL, LLENGTH, BYTES
      CLI  INPXT+3, X'00'
      BNE  WA&SYSNDX
VA&SYSNDX DECPC SEGMENT, SLENGTH, BYTES
      DECPC SKIPS, RSKIPS, SEGMENTS
WA&SYSNDX DECPC STRING, RNOSEG, SEGMENTS
      DECPC PROCESS, RNOS, STRINGS
      DECPC OUTPUT, OUTPXT, CODE
      DECPC POSTOP, LJ, LJ
      DECPC RETRIEVALCODE, MODE, MODE
      CLI  MODE+3, X'00'
      BE   XA&SYSNDX
      DECPC READ, NV, SUBSTRINGS
      CLI  INPXT+3, X'00'
      BE   XA&SYSNDX
XA&SYSNDX DECPC SEGMENT, JII, BYTES
      DECPC LEXCON, LEXCON, CONTROL
      CLI  LEXCON+3, X'00'

```

*THROUGH BEFORE.

SREADC (CONT.)

```

BE      YA&SYSNDX
DECPC  READ,TPCORD,PCORDS
YA&SYSNDX DECPC  LEXMODE,LEXMODE,CONTROL
CLI    LEXMODE+3,X'00'
BNE    ZA&SYSNDX
CLI    LEXCON+3,X'00'
BNE    ZA&SYSNDX
ZA&SYSNDX DECPC  LEXPCH,LEXPCH,CONTROL
L      ERR,SBRY
S      ERR,LBRY
C      ERR,JII
BNL    BB&SYSNDX
MVC    BLANK+6(99),=C'ARRAYS READ EXCEED THE MAX. ALLOWED LENX
      GTH(=2.0 X SLENGTH). INCREASE SLENGTH TO ( LTHAYY-340)/2X
      .0.
L      5,=F'10'
AB&SYSNDX PUT    PRINT,BLANK
BCT    5,AB&SYSNDX
B      CLI
BB&SYSNDX MVC    NV(4),RNV      *SUBSTRING COUNTER RELOADED.
L      ERR,SPSAVEBR
BR     ERR      *BRANCH BACK TO ALLOCATE.
LTORG
MEND

```

*****SREADC*****

SREADT (CONT.)

	A	7,JII	
	CLI	MODE+3,X'00'	
	BNE	GA&SYSNDX	
	CLC	SLENGTH(4),ZERO	
	BNE	GA&SYSNDX	
	LM	BRYY,BRY,WCD	
	SR	1,1	
	IC	1,0(BRY)	
	LR	0,1	
	A	1,=F'1'	
	AR	1,BRY	
	MVC	SLENGTH+1(3),0(1)	
	A	0,SLENGTH	
	A	0,=F'1'	
GA&SYSNDX	ST	0,SLENGTH	
	LM	BRYY,BRY,WCD	
	CLI	MODE+3,X'00'	
	BE	LA&SYSNDX	
	A	5,=F'1'	
	C	5,RNV	*REMOVE FOR SUBSTRING WORK.
	BH	LA&SYSNDX	*MAKE NV=1,SOS=A FORMAT AT END.
	L	1,LBRY	
	A	1,JII	
	S	1,=F'4'	
	MVC	DUM1(4),0(1)	
	L	1,DUM1	
	ST	1,CHECK(6)	
	L	0,RLSX(6)	
	L	1,RSOS(6)	
	C	1,ZERO	
	BNL	HA&SYSNDX	
	L	1,=F'1'	
	SLL	1,31	
	C	0,ZERO	
	BE	IA&SYSNDX	
	AL	1,=F'64'	*X=X'40' SOS LESS THAN ZERO LSX NOT ZERO
	B	KA&SYSNDX	
HA&SYSNDX	BNE	IA&SYSNDX	
	C	0,ZERO	
	BE	JA&SYSNDX	
	AL	1,=F'48'	*E=X'30'
	B	KA&SYSNDX	
IA&SYSNDX	AL	1,=F'16'	*A=X'10'
	B	KA&SYSNDX	
JA&SYSNDX	AL	1,=F'32'	*I=X'20'
KA&SYSNDX	L	3,JII	
	SLL	3,8	
	ALR	1,3	
	ST	1,SOS(6)	

SREADT (CONT.)

```

ST      0,LSX(6)
A      6,=F'4'
LA&SYSNDX C      7,SLENGTH
BNL    MA&SYSNDX
A      BRYY,JII
STM    BRYY,BRY,WCD
REIDT  0(BRY),JII
LM     BRYY,BRY,WCD
A      7,JII
LMOVE  JII+2,0(BRYY),0(BRY)
MA&SYSNDX B      GA&SYSNDX
ST      7,JII
CLI    MODE+3,X'00'
BE     OA&SYSNDX
ST     7,PSAVINGS
SLL    7,8
C      5,RNV
BNH    NA&SYSNDX
L      5,LBRY
A      5,PSAVINGS
S      5,=F'4'
MVC    CHECK(4),0(5)
L      0,=F'1'
ST     0,NV
NI     SOS,X'80'
IC     0,SOS
SLL    0,24
IC     0,SOS+3
ALR    0,7
ST     0,SOS
NA&SYSNDX AL    7,NV
ST     7,PARM
OA&SYSNDX MVC    SLENGTH(4),PA&SYSNDX
L      &RR.,SRRREADT
BR     &RR
SRRREADT DS    F
PA&SYSNDX DS    F
MEND

```

*OUT.

*REMOVE FOR SUBSTRING WORK.

*LENGTH RELOADED.

*TO SAVE THE SLENGTH FOR DECOMPRESSION.

*****SREADT*****

SREID (CONT.)

	CR	1,2	
	BE	JA&SYSNDX	*OUT
	L	0, ADDRESS(3)	
	B	IA&SYSNDX	
HA&SYSNDX	A	0,=F'4'	
	A	3,=F'4'	
	ST	0, ADDRESS(3)	
IA&SYSNDX	BCT	2, HA&SYSNDX	
JA&SYSNDX	L	3,=F'8'	
	L	6,=F'25'	
KA&SYSNDX	L	4, ZERO	
	SLDL	4,4	
	IC	0, HEXL(4)	
	STC	0, WORKA(6)	
	A	6,=F'1'	
	BCT	3, KA&SYSNDX	
	LM	2,3, ZERO	
	IC	2,=C'0'	
LA&SYSNDX	IC	3, WORKA(6)	
	CR	2,3	
	BE	NA&SYSNDX	
MA&SYSNDX	A	6,=F'1'	
	B	LA&SYSNDX	
NA&SYSNDX	BCT	1, MA&SYSNDX	
	CLI	AC,C'*	
	BE	OA&SYSNDX	
	IC	0,=C'0'	
	STC	0, WORKA(6)	
	A	6,=F'1'	
	B	QA&SYSNDX	
OA&SYSNDX	L	1,=F'8'	
	L	2, ZERO	
PA&SYSNDX	IC	0, ARRAY(2)	
	STC	0, WORKA(6)	
	A	6,=F'1'	
	A	2,=F'1'	
	BCT	1, PA&SYSNDX	
QA&SYSNDX	LH	7, RA&SYSNDX+2	
	ALR	7,6	
	STH	7, RA&SYSNDX+2	
RA&SYSNDX	MVC	WORKA(7),=C'FORMAT='	
	AL	7,=F'7'	
	A	6,=F'7'	
	STH	7, SA&SYSNDX+2	
SA&SYSNDX	MVC	WORKA(8), FORMAT	
	SLR	7,6	
	STH	7, RA&SYSNDX+2	
	PUT	PRINT, WORKA	
	MVC	DUMF(20), FORMAT	

SREID (CONT.)

	MVC	JII(4),ZERO	*TO COUNT BYTES OCCUPIED BY DATA.
	MVC	BWX(133),BLANK	
	MVC	WORKA(133),BLANK	
TA&SYSNDX	MVC	FORMAT(20),DUMF	*BEGIN A CARD.
	GET	MASTER,WORKA	
	CLI	WORKA,C' '*'	
	BNE	UA&SYSNDX	*BEGIN CALCULATIONS.
	MVI	WORKA,C' '°	
	PUT	PRINT,WORKA	*PRINTING THE COMMENT.
	B	TA&SYSNDX	*READ NEXT CARD FOR THIS FORMAT.
UA&SYSNDX	L	7,JII	
	C	7,ZERO	
	BNE	XA&SYSNDX	
VA&SYSNDX	MVC	BWX(34),=C'°	THE FIRST 16 BYTES READ WERE: '
	LH	7,VA&SYSNDX+2	
	AL	7,=F'34°	
	STH	7,WA&SYSNDX+2	
WA&SYSNDX	MVC	BWX(16),WORKA	
	MVI	BWX+50,C'°°	
XA&SYSNDX	CLI	FORMAT,C'X'°	
	BNE	CB&SYSNDX	*A,J,B,I,E AND F FORMAT.
	L	1,WORKA	*IRI=NO OF BYTES IN THE STRING.
	ST	1,JII	*JII=NO OF BYTES IN THE STRING.
	LR	7,1	
	L	2,ADDRESS	
	L	3,=F'80°	
	L	4,ZERO	
YA&SYSNDX	GET	MASTER,WORKA	
	ST	2,ADDRESS	
	B	IB&SYSNDX	*TO TEST ADDRESS
ZA&SYSNDX	C	7,=F'80°	
	BNL	AB&SYSNDX	
	S	7,=F'1°	
	STC	7,AB&SYSNDX+1	
AB&SYSNDX	MVC	0(80,2),WORKA	
	SR	7,3	
	BNH	BB&SYSNDX	*FINISHED READING X FORMAT.
	AR	2,3	
	B	YA&SYSNDX	
BB&SYSNDX	S	3,=F'1°	
	STC	3,AB&SYSNDX+1	
	L	0,=F'1°	
	ST	0,AC	
	B	RB&SYSNDX	
CB&SYSNDX	L	0,NF	
	LM	2,4,ZERO	
DB&SYSNDX	L	1,ZERO	*BEGIN FIELD.
	LR	3,2	
EB&SYSNDX	CLI	WORKA,X'E0°	

SREID (CONT.)

	BE	GB&SYSNDX	*HIT AN 0-8-2 PUNCH.
FB&SYSNDX	CLI	WORKA,C°I°	
	BE	GB&SYSNDX	
	LH	6,EB&SYSNDX+2	
	AL	6,F°1°	
	STH	6,EB&SYSNDX+2	
	STH	6,FB&SYSNDX.+2	
	A	2,F°1°	
	A	1,F°1°	
	C	2,F°80°	
	BE	GB&SYSNDX	*FINISHED CARD AND FIELD.
	B	EB&SYSNDX	
GB&SYSNDX	CLI	FORMAT,C°A°	
	BNE	XB&SYSNDX	*B,I,E,OR F.
HB&SYSNDX	S	1,F°1°	
	STC	1,MB&SYSNDX+1	
	A	1,F°1°	
	LH	6,MB&SYSNDX+4	
	ALR	6,3	
	STH	6,MB&SYSNDX+4	
IB&SYSNDX	LA	5,HERE	
	C	5,ADDRESS(4)	
	BNH	KB&SYSNDX	
	MVC	WORKA(133),BLANK	
	MVC	WORKA(28),=C°O LOW CORE ADDRESSING ERROR.'	
JB&SYSNDX	PUT	PRINT,WORKA	
	L	0,F°1°	
	ST	0,AC	
	B	CL1	
KB&SYSNDX	L	5,SBRY	
	A	5,F°340°	
	C	5,ADDRESS(4)	
	BH	LB&SYSNDX	
	MVC	WORKA(133),BLANK	
	MVC	WORKA(29),=C°O HIGH CORE ADDRESSING ERROR.'	
	B	JB&SYSNDX	
LB&SYSNDX	L	5,ADDRESS(4)	
	CLI	FORMAT,C°A°	
	BE	MB&SYSNDX	
	CLI	FORMAT,C°X°	
	BE	ZA&SYSNDX	
	CLI	FORMAT,C°J°	
	BNE	PC&SYSNDX	
MB&SYSNDX	MVC	0(99,5),WORKA	
	AR	5,1	
	ST	5,ADDRESS(4)	
	A	4,F°4°	
	SLR	6,3	
	STH	6,MB&SYSNDX+4	

SREID (CONT.)

			*BYTES COUNT
NB&SYSNDX	A	1,JII	
	ST	1,JII	
	C	2,=F'80'	
	BE	WB&SYSNDX	*JOB FINISHED RESET REID18 AND TEST AC
	A	2,=F'1'	
	C	2,=F'80'	
	BE	WB&SYSNDX	
	LH	6,OB&SYSNDX+2	
	ALR	6,2	
	STH	6,OB&SYSNDX+2	
	STH	6,EB&SYSNDX+2	
	STH	6,FB&SYSNDX.+2	
	STH	6,PB&SYSNDX.+2	
OB&SYSNDX	CLI	WORKA,X'E0'	
	BE	VB&SYSNDX	*RESET REID 18 AND REID 21
PB&SYSNDX	CLI	WORKA,C'1'	
	BE	VB&SYSNDX	*RESET REID 18 AND REID 21
QB&SYSNDX	SLR	6,2	
	STH	6,OB&SYSNDX+2	
	STH	6,PB&SYSNDX.+2	
RB&SYSNDX	MVC	FORMAT(8),FORMAT+1	
	BCT	0,DB&SYSNDX	
	MVC	EB&SYSNDX.(4),OB&SYSNDX	
	MVC	FB&SYSNDX.(4),PB&SYSNDX	
	CLI	AC,C'*'	
	BE	TA&SYSNDX	
	MVC	DUM4(16),=C' BYTES STORED.	
	L	4,JII	
	CVD	4,DUM1	
	UNPK	DUM2(16),DUM1(8)	
	NI	DUM2+15,X'0F'	
	OI	DUM2+15,X'F0'	
	L	0,=F'15'	
SB&SYSNDX	CLI	DUM2,C'0'	
	BNE	TB&SYSNDX	
	MVC	DUM2(31),DUM2+1	
	BCT	0,SB&SYSNDX	
TB&SYSNDX	LH	7,UB&SYSNDX+2	
	AL	7,=F'60'	
	STH	7,UB&SYSNDX+2	
UB&SYSNDX	MVC	BWX(30),DUM2	
	SL	7,=F'60'	
	STH	7,UB&SYSNDX+2	
	PUT	PRINT,BWX	
	PUT	PRINT,ASTERIK	
	LM	BRYY,BRY,WCD	
	L	ERR.,SRRSREID	
	BR	ERR	
SRRSREID	DS	F	

SREID (CONT.)

	BH	EC&SYSNDX
	ST	1,CS2
	LM	0,7,CS1
	L	2,=F ⁰ 8 ⁰
	B	NB&SYSNDX
EC&SYSNDX	MVC	DUM1(80),DUM5
	MVC	DUM5(8),ZERO
	CLI	FORMAT,C ⁰ B ⁰
	BE	KC&SYSNDX
	CLI	FORMAT,C ⁰ J ⁰
	BNE	QC&SYSNDX
	LR	0,2
	L	1,ZERO
FC&SYSNDX	PACK	DUN1(9),DUM1(16)
	MVG	DUN3(9),DUN1(9)
GC&SYSNDX	MVC	DUM5(8),DUN3
	C	0,=F ⁰ 16 ⁰
	BNH	HC&SYSNDX
	S	0,=F ⁰ 16 ⁰
	LH	6,GC&SYSNDX+2
	AL	6,=F ⁰ 8 ⁰
	STH	6,GC&SYSNDX+2
	A	1,=F ⁰ 8 ⁰
	MVC	DUM1(64),DUM1+16
	B	FC&SYSNDX
HC&SYSNDX	LH	6,GC&SYSNDX+2
	SLR	6,1
	STH	6,GC&SYSNDX+2
	LM	0,1,ZERO
	LR	0,2
	SRDL	0,1
	C	1,ZERO
	BE	IC&SYSNDX
	A	0,=F ⁰ 1 ⁰
IC&SYSNDX	ST	0,CS2
	LM	0,7,CS1
	LH	6,JC&SYSNDX+2
	ALR	6,3
	STH	6,JC&SYSNDX+2
	S	1,=F ⁰ 1 ⁰
	STC	1,JC&SYSNDX+1
	A	1,=F ⁰ 1 ⁰
JC&SYSNDX	MVC	WORKA(99),DUM5
	SLR	6,3
	STH	6,JC&SYSNDX+2
	B	HB&SYSNDX
KC&SYSNDX	C	2,=F ⁰ 8 ⁰
	BNH	LC&SYSNDX
	L	2,=F ⁰ 8 ⁰

*B FORMAT

*I,E OR F FORMATS.

SREID (CONT.)

LC&SYSNDX	S	2,=F'1'
	STC	2,MC&SYSNDX+1
	A	2,=F'1'
	L	3,=F'8'
	SR	3,2
	LH	6,MC&SYSNDX+2
	ALR	6,3
	STH	6,MC&SYSNDX+2
MC&SYSNDX	MVC	DUM5(99),DUM1
	SLR	6,3
	STH	6,MC&SYSNDX+2
	PACK	DUM1(5),DUM5(8)
	LM	4,5,DUM1
	SLDL	4,4
	CLI	FORMAT,C'I'
	BNE	NC&SYSNDX
	SRDL	4,28
	STM	4,5,DUM1
	CVB	4,DUM1
NC&SYSNDX	ST	4,DUM1
OC&SYSNDX	LM	0,7,CS1
	B	IB&SYSNDX
PC&SYSNDX	MVC	0(4,5),DUM1
	A	5,=F'4'
	ST	5,ADDRESS(4)
	L	1,=F'4'
	CLI	NF+3,X'14'
	BE	NB&SYSNDX
	A	4,=F'4'
	B	NB&SYSNDX
QC&SYSNDX	CLI	FORMAT,C'I'
	BNE	TC&SYSNDX
	CLI	DUM1,C'+'
	BNE	SC&SYSNDX
RC&SYSNDX	MVC	DUM1(16),DUM1+1
	S	2,=F'1'
	B	KC&SYSNDX
SC&SYSNDX	CLI	DUM1,C'-'
	BNE	KC&SYSNDX
	S	2,=F'1'
	IC	6,DUM1(2)
	N	6,=F'15'
	O	6,=F'176'
	STC	6,DUM1(2)
	A	2,=F'1'
	B	RC&SYSNDX
TC&SYSNDX	MVC	DUM5(16),ZERO
	MVC	DUM3(4),ZERO
	LR	0,2

*B FORMAT

*LOADED ALL REGISTERS.

*E OR F FORMATS.

*NEGATIVE OR NO SIGN.

SREID (CONT.)

```

C      2,=F'16'
BNH   UC&SYSNDX
L      0,=F'16'
UC&SYSNDX LM  1,5,ZERO
      CLI  DUM1,C'+ '
      BNE  WC&SYSNDX
VC&SYSNDX A   4,=F'1'
      S   0,=F'1'
      B   XC&SYSNDX
WC&SYSNDX CLI  DUM1,C'- '
      BE   VC&SYSNDX
XC&SYSNDX LH  7,YC&SYSNDX+2
      LH  6,BD&SYSNDX+2
      ALR 7,4
      STH 7,BD&SYSNDX+4
      STH 7,AD&SYSNDX+2
      STH 7,YC&SYSNDX+2
YC&SYSNDX CLI  DUM1,C'. '
      BNE  AD&SYSNDX
      L   3,=F'1'
ZC&SYSNDX AL  7,=F'1'
      STH 7,AD&SYSNDX+2
      STH 7,YC&SYSNDX+2
      STH 7,BD&SYSNDX+4
      A   4,=F'1'
      BCT 0,YC&SYSNDX
      L   0,=F'1'
      B   CD&SYSNDX
AD&SYSNDX CLI  DUM1,C'E '
      BE   CD&SYSNDX
BD&SYSNDX MVC  DUM5(1),DUM1
      AL  6,=F'1'
      STH 6,BD&SYSNDX+2
      A   2,=F'1'
      C   3,ZERO
      BH  ZC&SYSNDX
      A   1,=F'1'
      B   ZC&SYSNDX
CD&SYSNDX SLR  7,4
      STH 7,YC&SYSNDX+2
      STH 7,AD&SYSNDX+2
      STH 7,BD&SYSNDX+4
      SLR 6,2
      STH 6,BD&SYSNDX+2
*****REPLACE ALL BEYOND THIS POINT WITH THE NEW E AND F ROUTINE.**
      MVC CS15(4),=F'15'
      CLI  DUM1,C'+ '
      BE   DD&SYSNDX
      CLI  DUM1,C'- '

```

*ALIGNED FOR SIGN.

*E FORMAT.

SRETD (CONT.)

	BNE	DD&SYSNDX	
	MVC	CS15(4),=F'11'	
DD&SYSNDX	BCT	0,ED&SYSNDX	
	B	JD&SYSNDX	
ED&SYSNDX	A	4,=F'1'	
	ALR	7,4	
	STH	7,FD&SYSNDX+4	
FD&SYSNDX	MVC	DUM3(3),DUM1	
	SLR	7,4	
	STH	7,FD&SYSNDX+4	
	MVI	DUM4,C'0'	
	CLI	DUM3,C'+'	
	BE	GD&SYSNDX	
	CLI	DUM3,C'-'	
	BNE	HD&SYSNDX	*NO SIGN
	MVI	DUM4,X'B0'	
GD&SYSNDX	MVC	DUM3(2),DUM3+1	
	BCT	0,HD&SYSNDX	
	MVC	DUM3(4),ZERO	
	B	JD&SYSNDX	*NO EXPONENT.
HD&SYSNDX	C	0,=F'1'	
	BH	ID&SYSNDX	
	IC	0,DUM3	
	STC	0,DUM3+1	
	MVI	DUM3,X'F0'	
ID&SYSNDX	NI	DUM3+1,X'0F'	
	IC	0,DUM3+1	
	IC	5,DUM4	
	ALR	0,5	
	STC	0,DUM3+1	
	PACK	DUM4(4),DUM3(2)	
	L	5,DUM4	
	MVC	DUM4(4),ZERO	
	ST	5,DUM4+4	
	CVB	5,DUM4	
	ST	5,DUM3	*EXPONENT IS IN DUM3
JD&SYSNDX	CLI	DUM5,C'0'	
	BNE	KD&SYSNDX	
	MVC	DUM5(15),DUM5+1	
	S	1,=F'1'	
	S	2,=F'1'	
	IC	7,=X'00'	
	STC	7,DUM5(2)	
	B	JD&SYSNDX	
KD&SYSNDX	L	4,DUM3	
	AR	4,1	
	ST	4,DUM1	*EXPONENT IS STORED IN DUM1.
	PACK	DUM1(9),DUM5(16)	
	MVO	DUM5(9),DUM1(9)	

SREID (CONT.)

	L	7, DUM5
	MVC	DUM3(24), ZERO
	C	4, ZERO
	BNE	LD&SYSNDX
	B	MD&SYSNDX
LD&SYSNDX	LM	5, 6, ZERO
	BH	OD&SYSNDX
	LPR	4, 4
	SRDL	4, 1
	C	5, ZERO
	BZ	MD&SYSNDX
MD&SYSNDX	SRL	7, 4
	LH	5, ND&SYSNDX+2
	A	4, =F'2'
	MVC	DUM4(2), ZERO
	ALR	5, 4
	STH	5, ND&SYSNDX+2
	ST	7, DUM5
ND&SYSNDX	MVC	DUM4(4), DUM5
	SLR	5, 4
	STH	5, ND&SYSNDX+2
OD&SYSNDX	B	QD&SYSNDX
	C	4, =F'7'
	BNH	PD&SYSNDX
	L	6, =F'1600000'
	B	RD&SYSNDX
PD&SYSNDX	SLDL	6, 4
	BCT	4, PD&SYSNDX
	SLL	6, 4
	ST	6, DUM3+4
	L	4, ZERO
	B	MD&SYSNDX
QD&SYSNDX	L	6, =F'15'
	AL	6, DUM3+4
	ST	6, DUM3+4
	CVB	6, DUM3
RD&SYSNDX	MVC	DUM3(8), ZERO
	ST	6, DUM3
	MVI	DUM4+15, X'OF'
	MVC	DUN1(2), =X'016F'
	MVC	DUN7(8), ZERO
	L	1, =F'27'
	L	2, ZERO
	MVC	DUM5(16), ZERO
	MVC	DUN2(16), ZERO
	IC	4, DUM4+15
	N	4, =F'15'
	STC	4, DUN2+15
	L	5, ZERO

*TO DECEDER SUBROUTINE.

*NO IS TOO LARGE

SREID (CONT.)

SD&SYSNDX	L	0,=F'2'
TD&SYSNDX	MP	DUM4(16),DUN1(2)
	IC	5,DUM4+1
	SLL	5,4
	AL	5,=F'15'
	ST	5,DUN7+4
	CVB	5,DUN7
	MVC	DUM4(2),ZERO
	CLC	DUM4(16),DUN2
	BE	XD&SYSNDX
UD&SYSNDX	BCT	0,WD&SYSNDX
VD&SYSNDX	SLL	5,28
	SLDL	4,4
	STC	4,DUM5(2)
	A	2,=F'1'
	BCT	1,SD&SYSNDX
	B	YD&SYSNDX
WD&SYSNDX	SLDL	4,32
	BCT	1,TD&SYSNDX
	L	1,=F'1'
	B	VD&SYSNDX
XD&SYSNDX	L	1,=F'1'
	B	UD&SYSNDX
YD&SYSNDX	LH	7,ZD&SYSNDX+2
	AL	7,=F'4'
	STH	7,ZD&SYSNDX+2
ZD&SYSNDX	MVC	DUM3(14),DUM5
	SL	7,=F'4'
	STH	7,ZD&SYSNDX+2
	NI	DUM3,X'7F'
	L	0,=F'72'
	A	2,=F'4'
	L	3,=F'2'
AE&SYSNDX	CLI	DUM3,X'00'
	BNE	BE&SYSNDX
	S	0,=F'2'
	S	2,=F'1'
	MVC	DUM3(17),DUM3+1
	BCT	3,AE&SYSNDX
BE&SYSNDX	LM	3,4,ZERO
	STM	3,4,DUM4+8
CE&SYSNDX	CLI	DUM3,X'00'
	BNE	DE&SYSNDX
	S	0,=F'2'
	MVC	DUM3(17),DUM3+1
	BCT	2,CE&SYSNDX
	L	0,ZERO
	B	EE&SYSNDX
DE&SYSNDX	IC	3,DUM3

*CLEAN-UP FINISH.

*FINISHED.

*NO IS 0.0

SREID (CONT.)

```

N      3,=F'240'
C      3,ZERO
BNE   EE&SYSNDX
S      0,=F'1'
MVO   DUN2(16),DUM3(16)
MVC   DUM3(4),DUN2
FE&SYSNDX I      4,DUM3
L      1,ZERO
SRL   4,8
SRDL  0,7
CLI   CS15+3,X'0B'
BNE   FE&SYSNDX
A      0,=F'1'
FE&SYSNDX SLDL  0,31
ALR   4,0
ST    4,DUM1
B     OC&SYSNDX
L TORG
MEND

```

*****SREID*****

SRESULT

*****SRESULT *****

MACRO

SRESULT &NAME, &UR, &RR, &LSLOW, &JBLIST, &DUMMY LEVEL=2

*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXSRESULT XXXXXXXXXXXXXXXXXXXXXXXXXXXXXLEVEL=2

*---SRESULT GIVES THE RESULTS FOR EACH SEARCH, STORAGE AND UPDATE JOB.

*---&NAME EQUALS RESULT.

*---&UR IS THE USING REGISTER (8).

*---&RR IS THE RETURN REGISTER.

*---&LSLOW IS THE LENGTH OF THE SLOW MEMORY ADDRESS.

*---&JBLIST IS THE ADDRESS OF THE JOBLIST ARRAY.

*---&DUMMY EQUALS SOLID (EXTENDED FORM) OR DUMMY (OVERLAYS).

```

-----
      USING &NAME, &UR
RESULT  DS      OD
        STSR   &UR, &RR, &DUMMY
        ST     &RR., SRRESULT
        CLC    RNTASKS(4), NTASKS
        BNE    LA&SYSNDX                *REQUEST NOT FINISHED.
        XC     WORKA(133), WORKA
        MVC    WORKA(40), =C'1          THE RESULTS OF A NEW '
        LA     1, WORKA+40
        CLI    MODE+3, X'04'
        BNH    BA&SYSNDX                *MODE=0,1,2,3 OR 4.
        MVC    WORKA(35), =C'          MODE>4 HAS NOT BEEN ASSIGNED.'
        XC     WORKA+35(10), WORKA+35
AA&SYSNDX PUT    PRINT, WORKA
        B      CL1                      *MODE>4 MESSAGE.
BA&SYSNDX BL     CA&SYSNDX              *MODE=0,1,2 OR 3.
        MVC    0(31,1), =C'MAIN FILE UPDATING JOB(MODE=4).'
        B      GA&SYSNDX              *MODE=4 MESSAGE.
CA&SYSNDX CLI    MODE+3, X'02'
        BNH    DA&SYSNDX              *MODE=0,1 OR 2.
        MVC    0(37,1), =C'AUXILLIARY FILE UPDATING JOB(MODE=3).'
        B      GA&SYSNDX              *MODE=3 MESSAGE.
DA&SYSNDX BL     EA&SYSNDX            *MODE=0 OR 1.
        MVC    0(37,1), =C'AUXILLIARY FILE UPDATING JOB(MODE=2).'
        B      GA&SYSNDX              *MODE=2 MESSAGE.
EA&SYSNDX CLI    MODE+3, X'01'
        BL     FA&SYSNDX              *MODE=0.
        MVC    0(49,1), =C'STORAGE/UPDATING JOB OF THE SOLID SYSTEM(MOX
        DE=1).'
FA&SYSNDX B      GA&SYSNDX            *MODE=1 MESSAGE.
        MVC    0(22,1), =C'RETRIEVAL JOB(MODE=0).' *MODE=0 MESSAGE.
GA&SYSNDX PUT    PRINT, WORKA
        XC     WORKA(133), WORKA
        CLI    OUTPXT+3, X'03'
        BL     HA&SYSNDX              *OUTPXT=0,1 OR 2.
        MVC    WORKA(56), =C'0        OUTPUT DEVICE(OUTPXT>2) HAS NOT YETX
        BEEN ASSIGNED.'              *OUTPXT>2 MESSAGE.

```

SRESULT (CONT.)

```

HA&SYSNDX  B      AA&SYSNDX
MVC      WORKA(29),=C'0      ANSWERS WILL APPEAR ON '
CLI      OUTPXT+3,X'01'
BNH      IA&SYSNDX          *OUTPXT=0 OR 1.
MVC      WORKA+29(22),=C'THE PRINTER(OUTPXT=2). '
B        KA&SYSNDX          *OUTPXT=2 MESSAGE.
IA&SYSNDX BL      JA&SYSNDX          *OUTPXT=0 MESSAGE.
MVC      WORKA+29(16),=C'CARDS(OUTPXT=1). '
B        KA&SYSNDX          *OUTPXT=1 MESSAGE.
JA&SYSNDX MVC      WORKA+29(24),=C'MAGNETIC TAPE(OUTPXT=0). '
KA&SYSNDX PUT      PRINT,WORKA
LA&SYSNDX L        O,RTASKS
S        O,NTASKS
ST       O,DUM1
XC       WORKA(133),WORKA
MVI      WORKA,C'- '
PUT      PRINT,WORKA
DECPC    NUMBER,DUM1,TASK
DECPC    DID,NJOBS,JOBS
CLI      NJOBS+3,X'00'
BNH      NA&SYSNDX          *UNSUCCESSFUL.
L        1,NJOBS
L        2,SBRY
MA&SYSNDX HEXPC    MAINFILE,0(2),=F'&LSLOW'
A        2,=F'&LSLOW.'
NA&SYSNDX BCT      1,MA&SYSNDX
L        1,A&JBLIST
L        O,RTASKS
XC       DUM1(4),DUM1
S        O,NTASKS
BNH      PA&SYSNDX
OA&SYSNDX MVC      DUM1+2(2),0(1)
A        1,DUM1
BCT      O,CA&SYSNDX
PA&SYSNDX MVC      DUM1+2(2),0(1)
HEXPC    JOBLISTITEM,0(1),DUM1
TIME     BIN
S        O,SRTIME
S        O,ACESTIME
ST       O,SRTIME
DECPC    SEARCHTIME,SRTIME,SECS
DECPC    ACCESSTIME,ACESTIME,SECS
L        ERR.,SRRESULT
BR       ERR
SRRESULT DS      F
LTORG
MEND

```

*****RESULT*****

SSEARCH (CONT.)

```

SR      0,0
M      0,=F'&ADDL'
A      1,=F'4'
LR     IR3,1
L      IR1,AYY
A      IR2,=F'2'
OI     MSIGNAL,X'10'
L      IR6,=F'&ADDL'
L      1,SAVEYY
MVC    CURRENT(4),0(1)
TM     MSIGNAL,X'40'
BO     CA&SYSNDX
MVC    EMPTY+&ADDL.(&ADDL.),0(IR1)
OI     MSIGNAL,X'40'
A      IR1,=F'&ADDL'
MVC    BULK(&ADDL),0(IR1)
TM     MSIGNAL,X'80'
BZ     BA&SYSNDX
MVC    0(&ADDL.,1),EMPTY+&ADDL
BA&SYSNDX MVC EMPTY(&ADDL.),EMPTY+&ADDL
CA&SYSNDX COMPARE EMPTY,EMPTY+&ADDL,&ADDL
BNE    DA&SYSNDX
DA&SYSNDX MVC EMPTY+&ADDL.(&ADDL.),EMPTY
TM     CORD1(&ADDL.),EMPTY+&ADDL
BZ     EA&SYSNDX
STM    0,4,C1
APART  EMPTY,0,1,2,3,4
A      4,=F'&ADDL'
ASADD  EMPTY,0,1,2,3,4
EA&SYSNDX LM 0,4,C1
NI     MSIGNAL,X'DB'
AR     IR6,IR1
MVC    DUM1(4),0(IR6)
L      IR5,DUM1
TM     MSIGNAL,X'08'
BZ     GA&SYSNDX
LR     IR3,IR6
AR     IR3,IR5
A      IR6,=F'4'
LH     0,DUN1+2
CH     0,ZERO
BNL    FA&SYSNDX
AUXFILE &LSLOW,&ADDL
TM     MSIGNAL,X'20'
BO     IA&SYSNDX
B      FINISHED
FA&SYSNDX SCREEN &ADDL,&LFAST,0(IR6),&LTHAYY
***SCREEN SETS THE CONTINUANCE SIGNAL(X'20').

```

*INDEX FOR M-ARRAY.
*ADDRESS OF PART A.
*LINED ON J SCREEN.
*A SCREEN IS NEXT.
*IR6=ADDRESS LENGTH.
*BEGINNING OF MB.

*THROUGH BEFORE.
*SET THROUGH BEFORE.
*BULK ADDRESS LOADED.
*THE FILE EXISTS.
*COMPARE SLOW MEM ADD.
*SAVED TO PROTECT MB.

*IR6=ABS ADDRESS OF SUB-BLOCK.

*INDEX USED.
*IR3
*IR3
*LINED ON FIRST SCREEN.
*DUN1+2=NO OF BYTES IN JOB-LIST.
*STILL TRACING TO DO.
*SLOW AND FAST MEMORY ADDRESS LENGTHS.
*CONTINUANCE OF AUXILLARY FILE.
*GOES OUT OF THE SEARCH MACRO-SUBRT.

SSEARCH (CONT.)

	B	IA&SYSNDX	
GA&SYSNDX	CR	IR3, IR5	
	BL	HA&SYSNDX	
	AR	IR6, IR5	*IR6=ABS CONTINUANCE ADDRESS.
	OI	MSIGNAL, X'20'	
	SR	IR3, IR5	
	A	IR3, =F'4'	
	B	IA&SYSNDX	
HA&SYSNDX	AR	IR6, IR3	*IR6=ABS INDEX FOR SUB-BLOCK.
IA&SYSNDX	CLC	O(&ADDL, IR6), ZERO	
	BNE	JA&SYSNDX	
MMATCHN	MMATCH	NOVER1, NOVER2, NOVER3, O(IR2), JBARRAY, JBWORK, KLENGTH	
	MVC	O(&ADDL, IR6), EMPTY	
	ST	IR6, BWX+76	
	NI	MSIGNAL, X'FB'	
	OI	MSIGNAL, X'04'	
	MVC	O(&ADDL, IR1), ZERO	
JA&SYSNDX	MVC	ADDRESS(&ADDL), O(IR6)	
	TM	MSIGNAL, X'20'	
	BO	TBADDNE	*CONTINUANCE USE DUN1.
	MVC	DUN1(2), MASK2	*FOR THE NEXT SCREENS SUB-BLOCK.
	IC	O, MSIGNAL+2	
	AL	O, =F'1'	
	STC	O, MSIGNAL+2	
TBADDNE	TBADD	&ADDL, &LSLOW, &LFAST, &NTRKS, &TRKL, &MATRIXL, &MATRIXS	
	TM	MSIGNAL, X'20'	
	BO	EA&SYSNDX	*CONTINUANCE
	TM	MSIGNAL, X'18'	
	BO	KA&SYSNDX	*SCREEN NEXT.
	OI	MSIGNAL, X'08'	*INDEX BIT IS ON.
KA&SYSNDX	LH	BRYY, DUN1+2	*DUN1+2=NUMBER OF BYTES IN JBLIST
	CH	BRYY, ZERO	
	BH	LA&SYSNDX	*STILL DIAGONALS TO TRACE.
	MVC	DUN1+2(2), =H'-1'	
	MVC	DUN1(2), DUN1+2	
	B	EA&SYSNDX	
LA&SYSNDX	MVC	MASK2(2), O(IR2)	
	LH	O, MASK3	
	AR	IR2, O	
	CH	BRYY, MASK3	
	BH	MA&SYSNDX	
	MVC	MASK2(2), DUN1	
	B	NA&SYSNDX	
MA&SYSNDX	MVC	MASK2(2), O(IR2)	
	LH	O, MASK2	
	SH	O, =H'2'	
	STH	O, MASK2	
NA&SYSNDX	LH	O, MASK3	
	SR	IR2, O	

SSEARCH (CONT.)

```

LH    BRY,MASK3
A     IR2,=F'2'
SR    BRY,BRY
STH   BRY,DUN1+2
S     BRY,=F'2'
STH   BRY,DUN1
OA&SYSNDX LH 0,DUN1
AR    IR2,0
B     EA&SYSNDX
FINISHED MVI SEEKANS,X'00'
MVI   SRGATE,X'00'
CLC   JII(4),ZERO
BE    PA&SYSNDX
MVI   SEEKANS,X'01'
***SEEKANS=X'01' MEANS THE SEARCH WAS SUCCESSFUL.
***SEEKANS=X'00' MEANS THE SEARCH WAS UNSUCCESSFUL.
CLI   MODE+3,X'01'
BNE   PA&SYSNDX
MVC   NJOBS(4),RNJOBS
PA&SYSNDX L 1,AJBWORK
L     2,SBRY
L     5,NJOBS
SR    4,4
M     4,=F'&LSLOW.'
ST    5,DUM1
LMOVE DUM1+2,0(1),0(2)
***THE NJOBS BULK-FILE ADDRESSES HAVE BEEN TRANSFERED TO ARRAY JBWORK.
L     ERR.,SRREARCH
BR    ERR
SRREARCH DS F
LTORG
MEND
*****SSEARCH*****

```

*IR2 LINED FOR SCREEN.

*DUN1+2 HAS BEEN DECREASED.

*EXIT FROM SEARCH.

SSTATECL

```

*****SSTATECL*****
      MACRO
        SSTATECL &NAME,&UR,&ERR,&ADDL,&LTHAYY,&DUMMY
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---SSTATECL INITIALIZES THE ENTIRE SOLID SYSTEM.
*---&NAME EQUALS STATECL.
*---&UR IS THE USING REGISTER (8 OR 9).
*---&ERR IS THE RETURN REGISTER.
*---&ADDL IS THE NUMBER OF BYTES IN EACH COMPOSITE ADDRESS.
*---&LTHAYY IS THE NUMBER OF BYTES IN THE PRINCIPAL DATA ARRAY YY.
*---&DUMMY EQUALS TO SOLID (EXTENDED FORM) OR DUMMY (OVERLAYS).
*-----
      USING STATECL,&UR
      DS      OF
STATECL    STSR  &UR,&ERR,&DUMMY
           ST   &ERR,&SRRATECL
AA&SYSNDX MJARRAY &ADDL          *GENERATES THE M-J ARRAY DATA-DECK.
BA&SYSNDX GET   MASTER,WORKA
           CLC  WORKA+1(7),=C'NEWFILE'
           BE   EA&SYSNDX          *PRINT M-J ARRAYS.
           CLC  WORKA(4),=C'****'
           BE   CA&SYSNDX
           LA   5,DUM1
           MVC  DUM1(27),=C'* END OF THE M ARRAY DATED
           A    5,=F'27'
           MVC  0(10,5),WORKA
           MVC  DATE(10),WORKA    *DATE STORED.
           MVC  DATEC(10),WORKA  *DATE STORED.
           B    DA&SYSNDX
CA&SYSNDX MVI  WORKA,C' '
           PUT  PRINT,WORKA
           B    BA&SYSNDX
DA&SYSNDX GET  MASTER,WORKA
           MVC  0(80,2),WORKA
           A    2,=F'80'
           S    3,=F'80'
           BH  DA&SYSNDX
           GET  MASTER,WORKA
           CLC  WORKA(37),DUM1
           BE   EA&SYSNDX
           MVC  BLANK(42),=C' THERE IS SOMETHING WRONG WITH THE M ARRAX
           Y'
           PUT  PRINT,BLANK
           B    CL1
EA&SYSNDX MVI  WORKA,C' '
           PUT  PRINT,WORKA
           L    2,AYY
           AR   4,2
           PRINT B,C(2),0(4)          *PRINT M ARRAY

```


STORAGE

```

*****STORAGE *****
      MACRO
      STORAGE &TPCORD,&LJBLIST
*****XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---STORAGE IS THE COMMON WORK AREA FOR THE SOLID SYSTEM.
*---&TPCORD IS THE NUMBER OF PERMANENT CORDS USED IN ANPAKC.
*---&LJBLIST IS THE LENGTH OF THE JOBLIST WORK ARRAYS.
*-----
***THE DEVICE COMMANDS ARE NEXT.
INPXT    DC    F'0'      *INPUT DEVICE(TAPE(0);CARDS(1);EXIT(>1)).
OUTPXT   DC    F'0'      *OUTPUT DEVICE(TAPE(0);CARDS(1);PRINTER(2))
***IF OUTPXT>2,THE (OUTPXT-2) LISTINGS ARE SKIPPED.
RSKIPS   DS    F        *NO OF SEGMENTS ON TAPE THAT ARE SKIPPED.
SLENGTH  DS    F        *MAXIMUM LENGTH OF SEGMENT ON TAPE.
LLENGTH  DS    F        *NO OF BYTES IN LABEL FOR EACH STRING.
RNOSSEG  DC    F'1'     *NO OF SEGMENT PER STRING.
RNOS     DS    F        *NO OF STRINGS TO PROCESS FROM TAPE.
ROUTPXT  DC    F'0'     *PRINTER INDICATOR(SET EQUAL TO OUTPXT).
NOSSEG   DS    F        *SEGMENT COUNTER.
NOS      DS    F        *STRING COUNTER.
*=====
***THE STRING COMMANDS ARE NEXT.
MODE     DS    F        *RET(0);STORE(1);UPDATE MF(2);UPDATE AF(3).
LJ       DS    F        *POST-OP. CONTROL(>0=NO OF SLOW MODES(TAPE)
LEXCON   DS    1F      *NO PCORDS READ(0); PCORDS READ(1);GEN(2).
LEXMODE  DS    1F      *FAST MODE(0);SLOW MODE(NOT ZERO).
LEXPCH   DS    F        *PUNCH PCORDS WHEN JOB IS DONE(ZERO).
SWITCH   DS    F        *SET EQUAL TO LEXMODE BY THE MACHINE.
*=====
***THE SUBSTRING COMMANDS ARE NEXT.
NV       DS    F        *NO OF SUBSTRINGS PER SEGMENT.
RNV      DC    F'1'     *TO RESERVE NV FOR TAPE WORK.
JI       DS    F        *NO OF BYTES PER SUBSTRING(DUMMY).
SOS      DS    20F     *STATE OF SUBSTRING INDICATORS.
RSOS     DS    20F     *RESERVE STATE OF SUBSTRING COMMAND(TAPE)
LSX      DS    20F     *LEAST LIMIT OF SIGNIFICANCE INDICATORS.
RLSX     DS    20F     *RESERVE LEAST LIMIT OF SIGNIFICANCE(TAPE).
*=====
***INFORMATION COMPUTED IN THE NUMERIC COMPRESSOR.
BWX      DS    20F     *BIN WIDTH INDICATOR.
MYX      DS    20F     *MINIMUM OF NUMERIC SUBSTRINGS.
CHECK    DS    20F     *FOR ABSOLUTE CHECKS OF SUBSTRINGS ON DECOM
NDR      DS    F
NS       DS    F
*=====
***INFORMATION COMPUTED IN THE COMPRESSORS.
PARM     DS    F        *STATE OF SEGMENT INDICATOR.
JII      DS    F        *NO OF BYTES IN SEGMENT.
RJII     DS    F        *THE TOTAL NUMBER OF BYTES IN A STRING.

```

STORAGE (CONT.)

```

PSAVINGS  DS    1F                                *TO CHECK ANPAH DECOMPRESSION.
=====
***THE FOLLOWING ARE USED IN THE ANPAKC AND ANPAKD COMPONENTS.
R          DS    F
RM         DS    F                                *FOR ANPAK
TPCORD    DS    1F                                *NO OF PERMANENT CORDS IN PCORD.
PCORDS    DS    503D                              *AREA FOR STORING PCORDS.
CODE      DS    F                                *FOR ANPAK(CODE WORD)
NAP       DS    F                                *DUMMY
TL        DS    100D
LEXICON   DS    32D                              *FOR CONSTRUCTING LEXICON IN ANPAK.
CORD1     DS    3D                                *FOR ANPAK ONLY
CORD2     DS    3D                                *FOR ANPAK ONLY
=====
***WORDS FOR STORING THE VARIOUS TIMES ARE GIVEN NEXT.
SRTIME    DS    F                                *TO TIME SEARCHES BEGINNING AT START OF TRANLATE.
ACESTIME  DS    F                                *TO TIME ACCESS OF DISK IN SEARCH.
TIME      DS    1F                                *TO TIME COMPONENTS.
SEGTIME   DS    F                                *TO TIME SEGMENTS.
STIME     DS    F                                *TO TIME STRINGS.
TTIME     DS    F                                *TO TIME EACH JOB(TOTAL TIME).
DATEC     DS    10C                              *TO CHECK DATE ON M ARRAY .
DATE      DS    10C                              *FOR DATE READ IN M ARRAY.
=====
***THE ARRAYS REQUIRED FOR JOB-LIST ARE RESERVED NEXT.
NJOB      DS    F                                *NO OF JOBS IN THE JOB-LIST.
JLRSKIP   DS    F                                *THE NUMBER OF TAPE RECORDS TO BE SKIPPED.
NTASKS    DS    F                                *THE NUMBER OF TASKS PER JOB.
KLENGTH   DC    F'0'                            *THE NUMBER OF BYTES IN EACH KERNEL OR ELEMENT.
MVALUE    DS    F                                *THE VALUE OF M FOR GENITEM.
JVALUE    DS    F                                *THE MAXIMUM VALUE OF J FOR GENITEM.
NUMDIAG   DS    F                                *THE NUMBER OF DIAGONALS FOR GENITEM.
GENERATE   DC    F'0'                            *STORE RANDOM ODD NUMBER FOR GENERATOR.
JLL       DS    F                                *THE NUMBER OF BYTES IN JBLIST(TAPE).
NOVER1    DC    F'0'                            *NUMBER OF TYPE I OVER-RIDES.
NOVER2    DC    F'0'                            *NUMBER OF TYPE II OVER-RIDES.
NOVER3    DC    F'0'                            *NUMBER OF TYPE III OVER-RIDES.
LJBLIST   DC    F'&LJBLIST.'                    *LENGTH OF THE JOBLIST ARRAY(JBLIST).
ODDNO     DC    F'0'                            *TO STORE RANDOM ODD NUMBER FOR GENERATOR.
ODDNO1    DC    F'0'                            *TO STORE RANDOM ODD NUMBER FOR GENERATOR.
ODDNO2    DC    F'0'                            *TO STORE RANDOM ODD NUMBER FOR GENERATOR.
ODDNO3    DC    F'0'                            *TO STORE RANDOM ODD NUMBER FOR GENERATOR.
ODDNO4    DC    F'0'                            *TO STORE RANDOM ODD NUMBER FOR GENERATOR.
ODDNO5    DC    F'0'                            *TO STORE RANDOM ODD NUMBER FOR GENERATOR.
ODDNO6    DC    F'0'                            *TO STORE RANDOM ODD NUMBER FOR GENERATOR.
ODDNO7    DC    F'0'                            *TO STORE RANDOM ODD NUMBER FOR GENERATOR.
ODDNO8    DC    F'0'                            *TO STORE RANDOM ODD NUMBER FOR GENERATOR.
ODDNO9    DC    F'0'                            *TO STORE RANDOM ODD NUMBER FOR GENERATOR.
=====

```

STORAGE (CONT.)

***SPECIAL ARRAYS FOR THE DEFINED PURPOSE.

MASK1	DS	D	*FOR GENERAL PURPOSE OR FOR MASKS.
MASK2	DS	D	
MASK3	DS	D	
MASK4	DS	D	
DUMX	DS	256C	*FOR TRANSFERING BYTES IN RMVC.
DUN1	DS	D	*FOR GENERAL USE ANYWHERE IN PROGRAM.
DUN2	DS	D	*FOR GENERAL USE ANYWHERE IN PROGRAM.
DUN3	DS	D	*FOR GENERAL USE ANYWHERE IN PROGRAM.
DUN4	DS	D	*FOR GENERAL USE ANYWHERE IN PROGRAM.
DUN5	DS	D	*FOR GENERAL USE ANYWHERE IN PROGRAM.
DUN6	DS	D	*FOR GENERAL USE ANYWHERE IN PROGRAM.
DUN7	DS	D	*FOR GENERAL USE ANYWHERE IN PROGRAM.

 ***THE COMMANDS AND WORK AREA FOR THE INPUT/OUTPUT ROUTINES ARE NEXT.

GATE	DC	X'00'	*CARD TAPE GATE.
SGATE	DC	X'00'	*SEGMENT READ GATE.
TGATE	DC	X'00'	*TAPE GATE
RINPXT	DC	X'00'	*GATE FOR CARD-TAPE INPUT COMBINATIONS.
JLMGATE	DC	X'00'	*JOBLIST MESSAGE GATE.
JLGATE	DC	X'99'	*GATE FOR SGENITEM.
NFORM	DC	X'00'	*NORMALIZATION GATE.
SEEKANS	DC	X'00'	*SEARCH GATE.
PCGATE	DC	X'&TPCORD.'	*FOR SELECTING OPTIMUM PCORDS.
SRGATE	DC	X'00'	*THE SEARCH RETRIEVAL GATE
	DS	C	*FOR FUTURE GATE
	DS	C	*FOR FUTURE GATE
	DS	C	*FOR FUTURE GATE
	DS	C	*FOR FUTURE GATE
	DS	C	*FOR FUTURE GATE
	DS	C	*FOR FUTURE GATE
	DS	C	*FOR FUTURE GATE
NF	DS	F	*NUMBER OF FORMATS(FOR I/O)
NOV	DS	F	*NUMBER OF VARIABLES(FOR I/O)
AC	DS	F	*ASTERIK CONVENTION(FOR I/O FORMATS)
SRNAME	DS	F	
FORMAT	DS	8F	*FOR STORING FORMATS FOR I/O
DUMF	DS	4D	*TO RESERVE FORMAT IN I/O ROUTINES.
DUM1	DS	D	*FOR I/O USE ONLY.
DUM2	DS	D	*FOR I/O USE ONLY.
DUM3	DS	D	*FOR I/O USE ONLY.
DUM4	DS	7D	*FOR I/O USE ONLY.
DUM5	DS	10D	*FOR I/O USE ONLY.
ADDRESS	DS	20F	*FOR ADDRESSES OF VARIABLES IN I/O ROUTINES
WORKA	DS	CL160	*THE ALL PURPOSE GET AND PUT ARRAY.

 ***INDICATORS AND CONSTANTS ASSOCIATED WITH SSEARCH.

MSIGNAL	DS	F	*TO SIGNAL CONDITION OF DEVICES IN SEARCH.
CURRENT	DS	F	*FOR CURRENT ADDRESS IN SEARCH ROUTINE.
	DS	F	*FOR CURRENT ADDRESS IN SEARCH ROUTINE.
BULK	DS	F	*FOR STORING THE BULK INFORMATION ADDRESS
	DS	F	*FOR CURRENT ADDRESS IN SEARCH ROUTINE.
EMPTY	DS	F	*LOC. OF FIRST EMPTY DEVICE AND FM STORAGE.
	DS	F	*LOC. OF FIRST EMPTY DEVICE AND FM STORAGE.
	DS	F	*LOC. OF FIRST EMPTY DEVICE AND FM STORAGE.
	DS	F	*LOC. OF FIRST EMPTY DEVICE AND FM STORAGE.
AJBARRAY	DS	F	*FOR STORING THE JOBLIST ITEM ADDRESS.
NSBBYTES	DS	F	*NUMBER OF BYTES IN SUB-BLOCK.
RNTASKS	DS	F	*NUMBER OF TASKS.
RNJOBS	DS	F	*NUMBER OF JOBS.
ACONTINA	DS	F	*REL. ADDRESS IN SUB-BLOCK OF CONTINUANCE

MEND

*****STORAGE *****

STRATEGY

```

*****STRATEGY*****
MACRO
&J      STRATEGY &CURSCRN,&JBLIST,&JBWORK,&KLENGTH
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---STRATEGY DETERMINES IF THE SMATCH AND SMOBILE COMPONENTS ARE USED.
*---&CURSCRN IS THE ADDRESS OF THE CURRENT SCREEN OR DIAGONAL.
*---&JBLIST IS THE ADDRESS OF THE JOBLIST ITEM CONSIDERED.
*---&JBWORK IS THE JOBLIST WORK ARRAY.
*---&KLENGTH IS THE NO. OF BYTES IN EACH KERNEL OF THE I.R.
*-----
&J      NOPR 1
***AT THIS POINT THERE IS THE FOLLOWING INFORMATION.
***  REGISTERS:-
***    IR2(R5)-CONTAINS THE ADDRESS OF THE SCREEN IN THE JOBLIST ITEM.
***    IR3(R6)-CONTAINS THE ABSOLUTE CONTINUANCE ADDRESS LOCATION.
***    IR6(R3)-CONTAINS THE FIRST EX. PT. LOCATION IN THE ARRAY.
***    BRYR(R14)-CONTAINS THE SCREEN LENGTH.
***    BRY(R15)-CONTAINS THE EXECUTIVE POINTER LENGTH.
***  FULL-WORDS:-
***    AJBARRAY-CONTAINS THE BEGINNING ADDRESS OF THE JOBLIST ITEM.
***    AJBLIST-CONTAINS THE BEGINNING ADDRESS OF THE JOBLIST ARRAY.
***    JLL-CONTAINS THE TOTAL LENGTH OF ALL JOBLIST ITEMS.
***    NTASKS-CONTAINS THE NUMBER OF ITEMS IN THE JOBLIST.
***    KLENGTH-CONTAINS THE LENGTH OF EACH KERNEL.
***    AJBWORK-CONTAINS THE BEGINNING ADDRESS OF THE WORK ARRAY.
***    LJBLIST-CONTAINS THE LENGTH OF THE JOBLIST AND WORK ARRAYS.
***  HALF-WORDS:-
***    DUN1-CONTAINS THE LENGTH OF THE CURRENT SCREEN.
***    MASK2-CONTAINS THE LENGTH OF THE NEXT SCREEN IN THE ITEM.
***    DUN1+2-CONTAINS THE REMAINING LENGTH OF THE JOBLIST ITEM<
***          (THIS IS FROM THE END OF THE CURRENT SCREEN).
***  BYTES:-
***    MSIGNAL+2-CONTAINS THE NUMBER OF THE CURRENT SCREEN.
***    SRGATE-IS SET TO X'00'. THE MEANINGS ARE:
***      SRGATE=X'00'-SEARCH UNSUCCESSFUL OR FINISHED.
***      SRGATE=X'01'-BEGIN A NEW SEARCH(SET AJBARRAY=NEXT ITEM).
***      SRGATE=X'02'-CONTINUE THIS SEARCH OR FETCH CONTINUANCE.
***    MSIGNAL X'20' BIT ON AND SRGATE=X'02' MEANS FETCH CONTINUANCE.
***    NOTE:--IF SRGATE=X'02' THE REGISTER IR6 MUST CONTAIN THE LOCAT-
***          ION OF AN ADDRESS(OF LENGTH BRY-BRYR).IF THE MSIGNAL
***          X'20' BIT IS OFF THEN IR2=IR2+BRYR(NEXT SCREEN).
***  OVER-RIDE DATA:-
***    NOVER1-CONTAINS THE NUMBER OF TYPE I IN THE COMPLETE JOBLIST.
***    NOVER2-CONTAINS THE NUMBER OF TYPE II IN THE COMPLETE JOBLIST.
***    NOVER3-CONTAINS THE NUMBER OF TYPE III IN THE COMPLETE JOBLIST.
***    AOVER1-CONTAINS THE LOCATIONS OF THE TYPE I.
***    AOVER2-CONTAINS THE LOCATIONS OF THE TYPE II.
***    AOVER3-CONTAINS THE LOCATIONS OF THE TYPE III.
***    AOVER3R-CONTAINS THE TYPE III GATES ENTERED IN PAIRS.

```

STRATEGY (CONT.)

```

AA&SYSNDX TRANSFER 3,MATCH,AA&SYSNDX
           CLI   SRGATE,X'01'
           BL    FINISHED
           BH    CA&SYSNDX
           TRANSFER 3,MOBILE,CA&SYSNDX
MA&SYSNDX DS    F
CA&SYSNDX CLI   SRGATE,X'01'
           BL    FINISHED
           BE    NEWPLAY
           ST    0,BA&SYSNDX
           LR    0,BRY
           SR    0,BRY
           SR    0,=F'1'
           STC   0,DA&SYSNDX.+1
DA&SYSNDX MVC   ADDRESS(99),0(IR6)
           L    0,BA&SYSNDX
           B    TBADDNE
           MEND

```

*SEARCH FAILED.
*CONTINUE SEARCH.

*SEARCH COMPLETED.
*BEGIN NEW SEARCH.

*CONTINUE SEARCH.

*****STRATEGY*****

STRING

```

*****STRING *****
MACRO
&J STRING &MODE,&PCSTOP,&LEXCON,&LEXMODE,&LEXPCH
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---STRING READS THE STRING COMMANDS FOR THE SOLID SYSTEM.
*---&MODE IS THE RETRIEVAL COMMAND FOR THE SOLID SYSTEM.
*---&POSTOP IS THE POST-OPERATION FOR THE SOLID SYSTEM.
*---&LEXCON IS FOR ENTERING THE PERMANENT CORD TABLE FOR SANPAKC.
*---&LEXMODE DESIGNATES THE COMPRESSION MODE OF SANPAKC.
*---&LEXPCH CONTROLS THE PUNCHING OF NEW PERMANENT CORD TABLES.
*-----
&J CALL1 COMANDS,AA&SYSNDX
AA&SYSNDX NOPR 1
MEND
*****STRING *****

```


STRINGA (CONT.)

FA&SYSNDX	LM	0,1,ZERO	
	S	5,LENGTH	
	IC	0,PARM+3	
	L	BRY,SBRY	*0--CONTAINS NV.
	LA	BRY,DUN1	*RELOADED BASE OF ARRAY YY.
	A	BRY,=F'8'	*DUN1 TO STORE SDS,BWX, AND LSX.
FA&SYSNDX	L	2,SDS(1)	*TO ACCOUNT FOR P SAVINGS AND PARM
	STC	2,NDR	
	SLL	2,1	
	SRL	2,1	
	ST	2,0(BRY)	
	A	BRY,=F'4'	
	L	2,CHECK(1)	
	ST	2,0(BRY)	
	A	BRY,=F'4'	
	A	5,=F'8'	
	NI	NDR,X'0F'	
	CLI	NDR,X'00'	
	BE	GA&SYSNDX	*NO COMPRESSION VIA NUPAK.
	L	2,BWX(1)	
	ST	2,0(BRY)	
	A	BRY,=F'4'	
	L	2,MY(1)	
	ST	2,0(BRY)	
	A	BRY,=F'4'	
	A	5,=F'8'	
GA&SYSNDX	A	1,=F'4'	
	BCT	0,FA&SYSNDX	
	L	6,PARM	
	SRL	6,8	*6--CONTAINS THE OLD JII.
	A	5,=F'8'	
	SLL	5,8	
	IC	5,PARM+3	
	ST	5,PARM	*PARM RECONSTRUCTED WITH NEW JII.
	ST	5,DUN1+4	
	MVC	DUN1(4),PSAVINGS	*PSAVINGS AND PARM IN DUN1.
	SRL	5,8	*NEW JII.
	LR	7,5	
	L	4,JII	
	C	4,ZERO	
	BH	HA&SYSNDX	
	LNR	7,7	
HA&SYSNDX	ST	7,JII	
	S	5,=F'1'	*NEW JII.
	S	6,=F'1'	*OLD JII.
	LR	BRY,BRY	
	AR	BRY,5	
	AR	BRY,6	
	SR	5,6	
	A	6,=F'1'	
	RMVC	0,6,BRY,0,BRY	
	L	BRY,SBRY	
	S	5,=F'1'	
	STC	5,IA&SYSNDX+1	
IA&SYSNDX	MVC	0(0,BRY),DUN1	
JA&SYSNDX	B	ERADD	
	MEND		

*****STRINGA*****

STRINGD (CONT.)

```

ST      0,SEGTIME
DECPC  SEGMENT,SEGTIME,SECS
DECPC  NOW,PSAVINGS,BYTES
EXTRAKT JII,NV,DUM1+4
DECPC  WAS,JII,BYTES
DECPC  NUMBER,NV,SUBSTRINGS
MVC    WORKA(50),=C'0      ERROR OCCURED IN ALPHANUMERIC DECOMPR
      RESSION.
PUT    PRINT,WORKA
LM     0,15,C1
B      RTRANMIT
CA&SYSNDX MVC  PSAVINGS(4),DUM1
LR     3,2
SLL   3,8
AL    3,NV
ST    3,PARM
L     1,SBRY
DA&SYSNDX C    2,=F'256'
BNL   EA&SYSNDX
STC   2,EA&SYSNDX+1
EA&SYSNDX MVC  0(256,1),0(BRY)
A     1,=F'256'
A     BRY,=F'256'
S     2,=F'256'
BH    DA&SYSNDX
MVI   EA&SYSNDX+1,X'FF'
FA&SYSNDX MVC  0(40,BRY),ZERO
LM    0,7,ZERO
ST    0,RM
EXTRAKT JII,NV,PARM
MVC   CS15(4),SBRY      *SAVE THE BASE REGISTER OF YY.
MVC   JI(4),ZERO
MEND

```

*****STRINGD*****

STSR

```

*****STSR*****
MACRO
&J      STSR  &UR,&ERR,&DUMMY
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---STSR IS USED IN CONJUNCTION WITH TESTL TO EXTEND USING .
*---&UR IS THE REGISTER TO BE USED FOR USING CONTROL.
*---&ERR IS THE REGISTER THAT IS TO CONTAIN THE RETURN ADDRESS.
*---&DUMMY EQUALS SOLID (EXTENDED FORM) OR DUMMY (OVERLAYS).
*-----
&J      LR    &UR,15      *USING REGISTER HAS BEEN LOADED.
      USING &DUMMY.+8,BR1,BR3,BR4
      LM    0,1,WCD+8
      LR    &ERR,14      *BRANCH REGISTER IS LOADED.
      LM    14,15,WCD
MEND
*****STSR*****

```


SUBCBO

*****SUBCBO *****

MACRO

SUBCBO <HAYY

*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXSUBCBO XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

*---SUBCBO IS THE OVERLAY FORM FOR THE STAND-ALONE NUMERIC COMPRESSOR.

*---<HAYY IS THE NUMBER OF BYTES IN THE PRINCIPAL DATA ARRAY (YY).

*-----

LTORG

ENTRY ANPAKC

ENTRY ANPAKCC

ENTRY ANPAKD

ENTRY ANPAKDC

ENTRY CYCLIC

ENTRY GENITEM

ENTRY JOBLIST

ENTRY MATCH

ENTRY MEMORY

ENTRY MEMORYR

ENTRY MOBILE

ENTRY NORMAL

ENTRY PMARRAY

ENTRY REFLECT

ENTRY RESULT

ENTRY SAVEFM

ENTRY SEARCH

ENTRY STATECL

ENTRY TLATOR1

ENTRY TLATOR2

ENTRY TLATOR3

ENTRY TLATOR4

ENTRY TLATOR5

ENTRY XCHANGE

DUMADD PMARRAYR, ANPAKC

DUMADD PMARRAYR, ANPAKCC

DUMADD PMARRAYR, ANPAKD

DUMADD PMARRAYR, ANPAKDC

DUMADD PMARRAYR, AOVER1X

DUMADD PMARRAYR, AOVER2X

DUMADD PMARRAYR, AOVER3X

DUMADD PMARRAYR, AOVER2RX

DUMADD PMARRAYR, AOVER3RX

DUMADD PMARRAYR, CYCLIC

DUMADD PMARRAYR, GENITEM

DUMADD PMARRAYR, JBLISTX

DUMADD PMARRAYR, JBWORKX

DUMADD PMARRAYR, JOBLIST

DUMADD PMARRAYR, LOOKFILE

DUMADD PMARRAYR, MATCH

DUMADD PMARRAYR, MEMORY

SUBCBO (CONT.)

DUMADD PMARRAYR, MEMORYR
 DUMADD PMARRAYR, MOBILE
 DUMADD PMARRAYR, NORMAL
 DUMADD PMARRAYR, PMARRAY
 DUMADD PMARRAYR, REFLECT
 DUMADD PMARRAYR, RESULT
 DUMADD PMARRAYR, SAVEFM
 DUMADD PMARRAYR, SEARCH
 DUMADD PMARRAYR, STATECL
 DUMADD PMARRAYR, TLATOR1
 DUMADD PMARRAYR, TLATOR2
 DUMADD PMARRAYR, TLATOR3
 DUMADD PMARRAYR, TLATOR4
 DUMADD PMARRAYR, TLATOR5
 DUMADD PMARRAYR, XCHANGE

DS OD
 YY DS <HAYY.C *THE MAIN STORAGE AREA FOR THE ARRAY YY.
 *****SUBROUTINES NEXT*****SUBROUTINES NEXT*****SUBROUTINES
 OPENSHT
 *****FINISH SOLID SYSTEM*****
 MEND
 *****SUBCBO *****

SUBCJO

*****SUBCJO *****

MACRO

SUBCJO <HAYY

*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXSUBCJO XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

*---SUBCJO IS THE IS FOR THE OVERLAY FORM OF COPAKAN.

*---<HAYY IS THE NUMBER OF BYTES IN THE PRINCIPAL DATA ARRAY(YY).

*-----

LTORG

ENTRY CYCLIC

ENTRY GENITEM

ENTRY JOBLIST

ENTRY MATCH

ENTRY MEMORY

ENTRY MEMORYR

ENTRY MOBILE

ENTRY NORMAL

ENTRY NUPAK

ENTRY NUPAKCN

ENTRY NUPAKR

ENTRY OVERFLOW

ENTRY PMARRAY

ENTRY REFLECT

ENTRY RESULT

ENTRY SAVEFM

ENTRY SEARCH

ENTRY STATECL

ENTRY TLATOR1

ENTRY TLATOR2

ENTRY TLATOR3

ENTRY TLATOR4

ENTRY TLATOR5

ENTRY XCHANGE

DUMADD PMARRAYR,AOVER1X

DUMADD PMARRAYR,AOVER2X

DUMADD PMARRAYR,AOVER3X

DUMADD PMARRAYR,AOVER2RX

DUMADD PMARRAYR,AOVER3RX

DUMADD PMARRAYR,CYCLIC

DUMADD PMARRAYR,GENITEM

DUMADD PMARRAYR,JBLISTX

DUMADD PMARRAYR,JBWORKX

DUMADD PMARRAYR,JOBLIST

DUMADD PMARRAYR,LOCKFILE

DUMADD PMARRAYR,MATCH

DUMADD PMARRAYR,MEMORY

DUMADD PMARRAYR,MEMORYR

DUMADD PMARRAYR,MOBILE

DUMADD PMARRAYR,NORMAL

DUMADD PMARRAYR,NUPAK

SUBCJO (CONT.)

DUMADD PMARRAY,NUPAKCN
 DUMADD PMARRAY,NUPAKR
 DUMADD PMARRAY,OVERFLOW
 DUMADD PMARRAY,PMARRAY
 DUMADD PMARRAY,REFLECT
 DUMADD PMARRAY,RESULT
 DUMADD PMARRAY,SAVEFM
 DUMADD PMARRAY,SEARCH
 DUMADD PMARRAY,STATECL
 DUMADD PMARRAY,TLATOR1
 DUMADD PMARRAY,TLATOR2
 DUMADD PMARRAY,TLATOR3
 DUMADD PMARRAY,TLATOR4
 DUMADD PMARRAY,TLATOR5
 DUMADD PMARRAY,XCHANGE

DS OD

YY DS <HAYY.C *THE MAIN STORAGE AREA FOR THE ARRAY YY.

*****SUBROUTINES NEXT*****SUBROUTINES NEXT*****SUBROUTINES
 OPENSHT

*****FINISH SOLID SYSTEM*****
 MEND

*****SUBCJO *****

SUBCO

```

*****SUBCO *****
MACRO
SUBCO &LTHAYY
*XXXXXXXXXXXXXXXXXXXXXXXXXSUBCO XXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---SUBCO IS THE POSITIONING MACRO FOR THE OVERLAY FORM OF SOLID.
*---&LTHAYY IS THE NUMBER OF BYTES IN THE PRINCIPAL DATA ARRAY(YY).
*-----

```

```

LTORG
ENTRY CYCLIC
ENTRY GENITEM
ENTRY JOBLIST
ENTRY MATCH
ENTRY MEMORY
ENTRY MEMORYR
ENTRY MOBILE
ENTRY NORMAL
ENTRY NUPAKR
ENTRY PMARRAY
ENTRY REFLECT
ENTRY RESULT
ENTRY SAVEFM
ENTRY SEARCH
ENTRY STATECL
ENTRY TLATOR1
ENTRY TLATOR2
ENTRY TLATOR3
ENTRY TLATOR4
ENTRY TLATOR5
ENTRY XCHANGE
DUMADD PMARRAYR,AOVER1X
DUMADD PMARRAYR,AOVER2X
DUMADD PMARRAYR,AOVER3X
DUMADD PMARRAYR,AOVER2RX
DUMADD PMARRAYR,AOVER3RX
DUMADD PMARRAYR,CYCLIC
DUMADD PMARRAYR,GENITEM
DUMADD PMARRAYR,JBLISTX
DUMADD PMARRAYR,JBWORKX
DUMADD PMARRAYR,JOBLIST
DUMADD PMARRAYR,LOOKFILE
DUMADD PMARRAYR,MATCH
DUMADD PMARRAYR,MEMORY
DUMADD PMARRAYR,MEMORYR
DUMADD PMARRAYR,MOBILE
DUMADD PMARRAYR,NORMAL
DUMADD PMARRAYR,NUPAKR
DUMADD PMARRAYR,PMARRAY
DUMADD PMARRAYR,REFLECT
DUMADD PMARRAYR,RESULT

```

SUBCO (CONT.)

```
DUMADD PMARRAYR,SAVEFM
DUMADD PMARRAYR,SEARCH
DUMADD PMARRAYR,STATECL
DUMADD PMARRAYR,TLATOR1
DUMADD PMARRAYR,TLATOR2
DUMADD PMARRAYR,TLATOR3
DUMADD PMARRAYR,TLATOR4
DUMADD PMARRAYR,TLATOR5
DUMADD PMARRAYR,XCHANGE
DS      OD
YY      DS      &LTHAYY.C      *THE MAIN STORAGE AREA FOR THE ARRAY YY.
*****SUBROUTINES NEXT*****SUBROUTINES NEXT*****SUBROUTINES
OPENSHUT
*****FINISH SOLID SYSTEM*****
MEND
*****SUBCO *****
```

SUBME

```

*****SUBME *****
MACRO
  SUBME &ADDL,&LSLOW,&LFAST,&NTRKS,&TRKL,&LTHAYY,&JBLIST,&LJBLX
        IST,&LOVER1,&LOVER2,&LOVER3,&MATRIXL,&MATRIXS
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---SUBME IS THE FINAL MACRO FOR THE EXTENDED FORM OF THE SOLID SYSTEM.
*---&ADDL IS THE NUMBER OF BYTES IN EACH COMPOSITE ADDRESS.
*---&LSLOW IS THE LENGTH OF THE SLOW MEMORY ADDRESS.
*---&LFAST IS THE LENGTH OF THE FAST MEMORY ADDRESS.
*---&NTRKS IS THE NUMBER OF TRACKS IN EACH MEMORY BLOCK.
*---&TRKL IS THE NUMBER OF BYTES IN EACH TRACK OR RECORD.
*---&LTHAYY IS THE NUMBER OF BYTES IN THE PRINCIPAL DATA ARRAY (YY).
*---&JBLIST IS THE ADDRESS OF THE JOBLIST ARRAY.
*---&LJBLIST IS THE LENGTH OF THE &JBLIST ARRAY.
*---&LOVER1 IS THE LENGTH OF TYPE I,II, AND III OVER-RIDE TABLES.
*---&LOVER2 IS THE LENGTH OF THE UPDATING ARRAY(AOVER2RX).
*---&LOVER3 IS THE LENGTH OF TYPE III OVER-RIDE INFO. TABLE(AOVER3RX).
*---&MATRIXL IS THE NUMBER OF ITEMS IN EACH PRINCIPAL SUB-ARRAY.
*---&MATRIXS IS THE NUMBER OF ITEMS IN EACH SECONDARY SUB-ARRAY.
*-----
LTORG
ENTRANCE
DUMADD PMARRAYR,NUPAKR
DS      OD
YY      DS      &LTHAYY.C      *THE MAIN STORAGE AREA FOR THE ARRAY YY.
*****SUBROUTINES NEXT*****SUBROUTINES NEXT*****SUBROUTINES
OPENSHT
SACTION ACTION,8,1,SOLID
SANPAKC ANPAKC,8,1,SOLID
SANPAKD ANPAKD,8,1,SOLID
SCOMMAND COMANDD,8,9,SOLID
SCYCLIC CYCLIC,8,1,SOLID
SGENITEM GENITEM,8,1,SOLID
SJOBLIST JOBLIST,8,1,SOLID
SMATCH MATCH,8,1,SOLID
SMEMORY MEMORY,7,8,1,&ADDL,&NTRKS,&TRKL,&JBLIST,SOLID
SMOBILE MOBILE,8,1,SOLID
SNORMAL NORMAL,8,1,SOLID
SNUPAK NUPAK,8,1,SOLID
SOUTPUT OUTPUT,8,1,SOLID
SPRINT PRINT,9,1,SOLID
SPUNXH PUNXH,9,1,SOLID
SREADC READC,8,1,SOLID
SREADT READT,8,1,SOLID
SREFLECT REFLECT,8,1,SOLID
SREID REID,8,1,SOLID
SRESULT RESULT,8,1,&LSLOW,&JBLIST,SOLID
SSEARCH SEARCH,7,8,1,&ADDL,&LSLOW,&LFAST,&NTRKS,&TRKL,&LTHAYY
        Y,&JBLIST,&LJBLIST,&MATRIXL,&MATRIXS,SOLID
SSTATECL STATECL,8,1,&ADDL,&LTHAYY,SOLID
STLATOR1 TLATOR1,8,1,SOLID
STLATOR2 TLATOR2,8,1,SOLID
STLATOR3 TLATOR3,8,1,SOLID
STLATOR4 TLATOR4,8,1,SOLID
STLATOR5 TLATOR5,8,1,SOLID
SXCHANGE XCHANGE,8,1,SOLID
WORKAREA &LOVER1,&LOVER2,&LOVER3,&LJBLIST
*****SUBROUTINES ENDS*****SUBROUTINES ENDS*****SUBROUTINES
*****FINISH SOLID SYSTEM*****
MEND
*****SUBME *****

```


SUPERSCH

*****SUPER SCH*****

MACRO

&J SUPERSCH &ADDL,&LFAST

*XX

*---SUPER SCH IS THE SUPER SEARCH ROUTINE USED IN THE SCREEN PROCEDURE.

*---&ADDL IS THE LENGTH OF THE COMPOSITE ADDRESSES.

*---&LFAST IS THE LENGTH OF THE FAST PART OF THE COMPOSITE ADDRESSES.

*-----

```

&J      STM      0,15,C1
        LH       0,DUN1          *R0=SCREEN LENGTH
        LR       2,0
        A        2,=F'&ADDL.°   *R2=EX.PT. LENGTH
        LR       6,0
        STM      0,3,DUMX
        L        4,C4          *R4=FIRST EP(ABS. ADD)
        S        4,=F'4'
        MVC      MASK1(4),0(4)  *MASK1=REL. CONT. ADD.
        L        1,MASK1
        S        1,=F'4'      *R1=TOTAL EP LENGTH
        SR       0,0
        DR       0,2
        C        0,ZERO
        BNE     NA&SYSNDX      *ERROR MESSAGE FOR SUPERSCH
        LR       9,1          *R9=NO OF EP'S
        LM      0,2,ZERO
        L        3,=F'10'
AA&SYSNDX AR      2,3
        LR       1,2
        MR       0,2
        CR       1,9
        BL      AA&SYSNDX
        BE      BA&SYSNDX      *FOUND SQUARE ROOT IN R2
        SR       2,3
        C        3,=F'2'
        SRL     3,1
        BH      AA&SYSNDX
        L        3,=F'1'
BA&SYSNDX BE      AA&SYSNDX
        ST       2,MASK1      *MASK1=R2=SQ. ROOT
        LM      0,3,DUMX
        L        1,C7          *R1=C7=ABS. CONT. ADD.
        SR       1,2          *R1=ABS. ADD. LAST EP.
        LARGXC  DUN1,DUMX,DUMX
        MVI     JI,X'60'      *JI SET
CA&SYSNDX NI      JI,X'60'   *JI X'80' BIT OFF
        TM      JI,X'40'
        BZ      DA&SYSNDX
        OI      JI,X'10'
DA&SYSNDX CSSCRN DUN1,0(1),DUMX *IS EP SCREEN ZERO?

```

SUPERSCH (CONT.)

	BE	EA&SYSNDX	
	CSSCRN	DUN1,0(1),0(IR2)	*ARE EP AND JBLIST SCREENS EQUAL?
	BE	OA&SYSNDX	
	BH	EA&SYSNDX	
EA&SYSNDX	OI	JI,X'80'	*JI X'80' BIT ON
	TM	JI,X'20'	
	BZ	FA&SYSNDX	
	NI	JI,X'D0'	*FIRST TIME(X'20' OFF)
	BCTR	9,0	
	B	GA&SYSNDX	
FA&SYSNDX	S	9,MASK1	
GA&SYSNDX	C	9,ZERO	
	BNH	JA&SYSNDX	*EXIT
	SR	14,14	
	LR	15,2	
	MR	14,9	*R15=R9*R2
	TM	JI,X'80'	
	BZ	HA&SYSNDX	
	NI	JI,X'80'	
	OI	JI,X'40'	
	AR	15,1	
	C	15,C7	
	BNL	LA&SYSNDX	
	B	IA&SYSNDX	
HA&SYSNDX	NI	JI,X'90'	
	SR	15,1	
	LPR	15,15	
	CR	15,IR6	
	BL	KA&SYSNDX	
IA&SYSNDX	LR	1,15	
	TM	JI,X'50'	
	BM	CA&SYSNDX	*NOT FOUND BLOCK YET
	BO	LA&SYSNDX	*BEGINNING OF BLOCK FOUND
JA&SYSNDX	S	1,MASK1	
	CR	1,IR6	
	BNL	LA&SYSNDX	
KA&SYSNDX	LR	1,IR6	
LA&SYSNDX	CSSCRN	DUN1,0(1),DUMX	*BEGIN SINGLE ITEM SEARCH
	BE	OA&SYSNDX	
	CSSCRN	DUN1,0(1),0(IR6)	
	BNL	OA&SYSNDX	
	AR	1,2	
	C	1,C7	
	BL	LA&SYSNDX	
	BH	NA&SYSNDX	
MA&SYSNDX	OI	MSIGNAL,X'20'	*CONTINUANCE BIT TURNED ON.
	B	OA&SYSNDX	
NA&SYSNDX	MVC	BLANK+10(48),=C'PROGRAMMING ERROR IN THE SUPER-SEARCH X PROCEDURE.'	

TALE

```

*****TALE *****
MACRO
&J TALE &DASH,&MESSAGE,&NOTIMES
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---TALE PRINTS MESSAGES IN THE DESIGNATED BACKGROUND.
*---&DASH IS THE BACKGROUND(ASTERIK,DASH,BLANK,DZERO).
*---&MESSAGE IS THE MESSAGE TO BE PRINTED.
*---&NOTIMES IN THE NUMBER OF TIMES THE MESSAGE IS TO BE PRINTED.
*-----
&J STM 0,15,SERVICE
IA 2,&DASH
A 2,=F'45'
MVC 0(20,2),=C'START OF NEW &MESSAGE.
MVC SRNAME(1),&DASH
MVI &DASH,C' '
L 3,=F'&NOTIMES.'
AA&SYSNDX PUT PRINT,&DASH
BCT 3,AA&SYSNDX
MVC 0(20,2),&DASH.+100
MVC &DASH.(1),SRNAME
LM 0,15,SERVICE
MEND
*****TALE *****

```

TBADD

```

*****TBADD *****
MACRO
&J TBADD &ADDL,&LSLOW,&LFAST,&NTRKS,&TRKL,&MATRIXL,&MATRIXS
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---TBADD IS THE TEST BLOCK ADDRESS MACRO.IT CALLS SMEMORY.
*---&ADDL IS THE NUMBER OF BYTES IN THE COMPOSITE ADDRESS.
*---&LSLOW IS THE LENGTH OF THE SLOW PART OF COMPOSITE ADDRESSES.
*---&LFAST IS THE LENGTH OF THE FAST PART OF COMPOSITE ADDRESSES.
*---&NTRKS IS THE NUMBER OF TRACKS(OR RECORDS) IN EACH MEMORY BLOCK.
*---&TRKL IS THE NUMBER OF BYTES IN EACH TRACK OR RECORD.
*---&MATRIXL IS THE NUMBER OF ITEMS IN THE PRINCIPAL SUB-ARRAYS.
*---&MATRIXS IS THE NUMBER OF ITEMS IN THE SECONDARY SUB-ARRAYS.
*-----
&J COMPARE CURRENT,ADDRESS,&ADDL
BE AA&SYSNDX
COMPARE ADDRESS,ZERO,&ADDL
BE AA&SYSNDX
NEWBLOCK GLOBAL &ADDL,&NTRKS,MEMORYR
MEMORYR MVC CURRENT(&ADDL),ADDRESS
AA&SYSNDX APART ADDRESS,0,0,0,0,IR6
TM MSIGNAL,X'04'
BZ BA&SYSNDX
CREATE &ADDL,&LSLOW,&LFAST,&NTRKS,&TRKL,&MATRIXL,&MATRIXS
BA&SYSNDX NI MSIGNAL,X'FB'
MEND
*****TBADD *****

```


TRANLATE

```

*****TRANLATE*****
      MACRO
&J      TRANLATE &ARRAY,&NTASKS,&JLL,&KLENGTH
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---TRANLATE CALLS THE TRANSLATOR COMPONENTS DESIGNATED BY GETJLIST.
*---&ARRAY IS THE NAME OF THE ARRAY WHERE THE JOBLIST DISCRIPOTRS ARE.
*---&NTASKS WILL BE DETERMINED IN THE SELECTED TRANSLATOR.
*---&JLL IS THE NUMBER OF BYTES OF ALL DISCRIPOTRS TO BE TRANSLATED.
*---&KLENGTH IS THE NUMBER OF BYTES IN EACH KERNEL.
***&NJOBS IS THE NUMBER OF BULK INFO. ITEMS STORED UNDER ONE PATH.
*-----
&J      TIME  BIN
      ST    0,SRTIME                      *BEGIN TIMING OF SEARCH.
      XC    GA&SYSNDX.(50),GA&SYSNDX
      XC    NOVER1(12),NOVER1             *NUMBER OF OVERRIDES LOADED
      MVC    GA&SYSNDX.(18),=C'TRANSLATOR  USED.'
      CLI    TGATE,X'01'
***TGATE IS THE TRANSLATOR GATE WHICH IS SET IN SJOBLIST(INPUT).
      BNL    AA&SYSNDX
      MVC    GA&SYSNDX.(23),=C'NO TRANSLATOR WAS USED.'
      B      HA&SYSNDX                    *NO TRANSLATOR WAS USED.
AA&SYSNDX BH    BA&SYSNDX
***TRANSLATOR NUMBER ONE WILL BE USED.
      MVI    GA&SYSNDX.+12,C'1'
      TRANSFER 3,TLATOR1,HA&SYSNDX
BA&SYSNDX CLI    TGATE,X'03'
      BNL    CA&SYSNDX
***TRANSLATOR NUMBER TWO WILL BE USED.
      MVI    GA&SYSNDX.+12,C'2'
      TRANSFER 3,TLATOR2,HA&SYSNDX
CA&SYSNDX BH    DA&SYSNDX
***TRANSLATOR NUMBER THREE WILL BE USED.
      MVI    GA&SYSNDX.+12,C'3'
      TRANSFER 3,TLATOR3,HA&SYSNDX
DA&SYSNDX CLI    TGATE,X'05'
      BNL    EA&SYSNDX
***TRANSLATOR NUMBER FOUR WILL BE USED.
      MVI    GA&SYSNDX.+12,C'4'
      TRANSFER 3,TLATOR4,HA&SYSNDX
EA&SYSNDX BH    FA&SYSNDX
***TRANSLATOR NUMBER FIVE WILL BE USED.
      MVI    GA&SYSNDX.+12,C'5'
      TRANSFER 3,TLATOR5,HA&SYSNDX
***JLGATE>05 EXITS NEXT WITH AN ERROR MESSAGE.
FA&SYSNDX MVC    GA&SYSNDX.+5(66),=C'TRANSLATORS > 5 ARE TO BE ASSIGNEDX
      IN THE TRANLATE MACRO-INSTRUCTION.'
      PUT    PRINT,GA&SYSNDX
      B      CLI                          *EXIT FROM SOLID.
GA&SYSNDX DC    132C' '                  *FOR BUILDING MESSAGE.
HA&SYSNDX CLI    JLMGATE,X'00'
      BH    IA&SYSNDX                    *DONT PRINT MESSAGE.
***FINISH CONSTRUCTING THE MESSAGE AND PRINT IT.
      PUT    PRINT,GA&SYSNDX
      MVI    JLMGATE,X'01'
IA&SYSNDX NOPR  1
      MVC    RNTASKS(4),NTASKS          *SAVED NUMBER OF TASKS.
      MEND
*****TRANLATE*****

```

TRANSFER

```

*****TRANSFER *****
MACRO
&J      TRANSFER &N,&NAME,&RETURN
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---TRANSFER IS FOR CALLING COMPONENTS FROM WITHIN OTHER COMPONENTS.
*---&N IS THE DEPTH OF NESTING OF THE MACRO WHERE THE CALL ORIGINATES.
*---&NAME IS THE NAME OF THE COMPONENT REQUESTED.
*---&RETURN IS THE RETURN ADDRESS .
*-----
&J      STM    0,15,COSAVE&N          *SAVE ALL REGISTERS.
        TESTL  &NAME
        LM     0,15,COSAVE&N          *LOAD ALL REGISTERS.
        B      &RETURN                *RETURN.
        MEND
*****TRANSFER *****

```

TRUNC

```

*****TRUNC *****
MACRO
&J      TRUNC &FROM,&TO
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---TRUNC IS THE MORE USEFUL FORM OF TRUN.
*---&FROM IS THE FROM ADDRESS
*---&TO IS THE TO ADDRESS
*-----
&J      STM    0,15,SERVICE
        L      1,&FROM
        L      2,=F'127'
        SLDL   0,8
        NR     2,0
        S      2,=F'64'
        SR     0,0
        CR     2,0
        BNH    BA&SYSNDX
        C      2,=F'8'
        BH     OVERFLOW                *NUMBER IS TOO LARGE
        SLL    2,2                    *MULT. SIG. DIGITS BY 4
        SLDL   0,0(2)                *SHIFT NO.
        L      1,&FROM
        C      1,ZERO                  *CHECK FOR NEG. NO.
        BNL    BA&SYSNDX
        LNR    0,0                    *MAKE NO. NEG.
BA&SYSNDX ST    0,&TO
        LM     0,15,SERVICE
        MEND
*****TRUNC *****

```

WORKAREA

```

*****WORKAREA*****
      MACRO
      WORKAREA &LOVER1,&LOVER2,&LOVER3,&LJBLIST
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---&LOVER1 IS THE LENGTH OF TYPE I,II, AND III OVER-RIDE TABLES.
*---&LOVER2 IS THE LENGTH OF THE UPDATING ARRAY(AOVER2RX).
*---&LOVER3 IS THE LENGTH OF TYPE III OVER-RIDE INFO. TABLE(AOVER3RX).
*---&LJBLIST IS THE LENGTH OF THE JOB-LIST WORK ARRAYS.
*-----
***ARRAYS USED FOR LOADING,MANIPULATING AND NORMALIZING JOBLIST.
      AOVER1X  DS    CF
                DS    &LOVER1.C          *ARRAY FOR TYPE I OVER-RIDES.
      AOVER2X  DS    OF
                DS    &LOVER1.C          *ARRAY FOR TYPE II OVER-RIDES.
      AOVER2RX DS    OF
                DS    &LOVER2.C          *ARRAY FOR UPDATING PATHS.
      AOVER3X  DS    OF
                DS    &LOVER1.C          *ARRAY FOR TYPE III OVER-RIDES.
      AOVER3RX DS    OF
                DS    &LOVER3.C          *RANGE OF TYPE III OVER-RIDES.
      JBLISTX  DS    &LJBLIST.C          *THE JOB-LIST ARRAY.
      JBWORKX  DS    OF
                DS    &LJBLIST.C          *THE JOB-LIST WORK ARRAY.
      MEND
*****WORKAREA*****

```

WRITE

```

*****WRITE *****
      MACRO
&J      WRITE &ADDRESS,&JII
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*---WRITE WRITE ONTO THE TAPE WITH DCB TAPEOTC(COPAK6).
*---&ADDRESS IS THE LOCATION OF THE FIRST BYTE IN CORE.
*---&JII IS THE NUMBER OF BYTES TO BE WRITTEN ONTO TAPE.
*-----
&J      STM      0,15,IOSAVE
        LA      0,&ADDRESS
        ST      0,DUM1
        SR      1,1
        L       3,&JII
AA&SYSNDX L      0,DUM1
        S       0,=F'4'
        LR      2,0
        MVC     DUM2(4),0(2)
        L       1,=F'2988'
        CR      3,1
        BNL     BA&SYSNDX
        LR      1,3
BA&SYSNDX A      1,=F'4'
        SLL     1,16
        ST      1,0(2)
        PUT     TAPEOTC,(0)
        MVC     0(4,2),DUM2
        L       1,=F'2988'
        A       1,DUM1
        ST      1,DUM1
        S       3,=F'2988'
        BH     AA&SYSNDX
        LM      0,15,IOSAVE
        MEND
*****WRITE *****

```


COPAKANO

```

COPAKANO START X'22020'          START OF THE COPAKANO PROGRAM.
      RESERCO 200000,5,1500
* &LTHAYY=200,000 IS THE LENGTH OF THE PRINCIPAL DATA-ARRAY.
* &TPCORD=5 IS THE NUMBER OF PERMANENT CORDS USED BY THE ALPHANUMERIC
* COMPRESSOR.
* &LJBLIST=1500 IS THE LENGTH OF ARRAYS JBLIST AND JBWORK
NEWJOB  DEVICE INPXT,OUTPXT,RSKIPS,SLENGTH,LLENGTH,RNOS,TPCORD
* INPXT-----TAPE(0),,CARDS(1),TERMINATE(>1)
* OUTPXT-----TAPE(0),CARDS(1),PRINTER(>2 FOR OUTPXT-2 SKIPS).
* RSKIPS-----NUMBER OF STRINGS TO BE SKIPPED AT BEGINNING OF JOB.
* SLENGTH-----THE MINIMUM BYTES IN EACH SEGMENT (A TAPE COMMAND).
* LLENGTH-----THE NUMBER OF BYTES(<256) IN THE LABEL.
* RNOS-----THE NUMBER OF STRINGS TO BE PROCESSED(TAPE COMMAND).
* TPCORD-----THE NUMBER OF PERMANENT CORDS TO BE USED
STRING  STRING MODE,POSTOP,LEXCON,LEXMODE,LEXPCH
* MODE-----DECOMPRESS(0),COMPRESS(1).
* POSTOP-----TERMINATE(<1),NEWJOB(=1),STRING(0 OR <0).
* -----FOR POSTOP NEGATIVE THE LEXMODE AND LEXPCH COMMANDS ARE
* -----CHANGED TO THE FAST MODE AND NO PERMANENT CORDS ARE
* -----PUNCHED AFTER MINUS POSTOP STRINGS HAVE BEEN PROCESSED.
* -----THIS COMMAND IS FOR TAPE ONLY.
* LEXCON-----READ PERMANENT CORDS COMMAND (READ=1 ; NO CARDS=0).
* LEXMODE-----FAST-MODE (0); SLOW-MODE (1); EXTEND PCORDS TABLE (2).
* LEXPCH-----PUNCH PCORDS TABLE (0); DON'T PUNCH TABLE (NOT 0).
ANSWER  DISPOSE CARDREAD
CARDREAD CALL1 READC,TAPEREAD
* SUB-STRING COMMANDS ARE READ IN THE SREADC COMPONENT.
* NV IS THE NUMBER OF SUBSTRINGS PER STRING.
* SOS IS THE STATE-OF-SUBSTRING COMMAND.
* LSX IS THE "LOWEST LIMIT OF LEAST SIGNIFICANCE".
* SOS AND LSX TOGETHER DEFINE THE FORMAT OF SUBSTRINGS AND SIGNIFY THE
* PATH TAKEN THROUGH THE COMPRESSORS.
* SOS<0 ;LSX#0 -----X FORMAT;DONT COMPRESS THE SUBSTRING.
* SOS<0 ;LSX=0 -----A FORMAT;DONT COMPRESS THE SUBSTRING.
* SOS=0 ;LSX=0 -----I FORMAT;COMPRESS WITH SNUPAK(INTERGER).
* SOS=0 ;LSX<0 -----E FORMAT;COMPRESS WITH SNUPAK(LEFT-SHIFT).
* SOS=0 ;LSX>0 -----E FORMAT;COMPRESS WITH SNUPAK(BIN METHOD).
* SOS>0 ;-----A FORMAT; DONT COMPRESS THE SUBSTRING.
TAPEREAD CLI  INPXT+3,X'CO'
      BNE  SREADTR          NOT ON TAPE.
      CALL1 READT,SREADTR
SREADTR MVC  RSKIPS(4),ZERO
      COPAJ OUTPUTN
OUTPUTN CALL1 OUTPUT,NEWJOB
      SUBCJO 200000
* &LTHAYY=200,000 IS THE LENGTH OF THE DATA ARRAY.
* 80000 BYTES IS ABOUT THE MAXIMUM STRING LENGTH.
      END  COPAKANO          END OF THE COPAKANO PROGRAM.

```


SOLIDE

```

SOLIDE  START X*22020*          *START OF THE SOLIDE CONTROL PROGRAM
RESERVE 6,100000,5,1500
* &ADDL=6 IS THE NUMBER OF BYTES IN THE COMPOSITE ADDRESSES
* &LTHAYY=100,000 IS THE LENGTH OF THE PRINCIPAL DATA-ARRAY(YY).
* &TPCORD=5 IS THE NUMBER OF PERMANENT CORDS TO BE USED BY SANPAKC
* &LJBLIST=1500 IS THE LENGTH OF ARRAYS JBLIST AND JBWORK
NEWJOB  DEVICE INPXT,OUTPXT,RSKIPS,SLENGTH,LLENGTH,RNOS,TPCORD
* INPXT-----TAPE(0),,CARDS(1),TERMINATE(>1)
* OUTPXT-----TAPE(0),CARDS(1),PRINTER(>2 FOR OUTPXT-2 SKIPS).
* RSKIPS-----NUMBER OF STRINGS TO BE SKIPPED AT BEGINNING OF JOB.
* SLENGTH-----THE MINIMUM BYTES IN EACH SEGMENT (A TAPE COMMAND).
* LLENGTH-----THE NUMBER OF BYTES(<256) IN THE LABEL.
* RNOS-----THE NUMBER OF STRINGS TO BE PROCESSED(TAPE COMMAND).
* TPCORD-----THE NUMBER OF PERMANENT CORDS TO BE USED
STRING  STRING MODE,POSTOP,LEXCON,LEXMODE,LEXPCH
* MODE-----RETRIEVE(0),STORE(1),UPDATE(2,3,4).
* POSTOP-----TERMINATE(<1),NEWJOB(=1),STRING(0 OR <0).
* -----FOR POSTOP NEGATIVE THE LEXMODE AND LEXPCH COMMANDS ARE
* -----CHANGED TO THE FAST MODE AND NO PERMANENT CORDS ARE
* -----PUNCHED AFTER MINUS POSTOP STRINGS HAVE BEEN PROCESSED.
* -----THIS COMMAND IS FOR TAPE ONLY.
* LEXCCN-----READ PERMANENT CORDS COMMAND (READ=1 ; NO CARDS=0).
* LEXMODE-----FAST-MODE (0); SLOW-MODE (1); EXTEND PCORDS TABLE (2).
* LEXPCH-----PUNCH PCORDS TABLE (0); DON'T PUNCH TABLE (NOT 0).
ITEMS   GETJLIST JLINPXT,JLRSKIP,JLTRAN,JLNORM,KLENGTH,NJOBS,NTASKS,X
        MVALUE,JVALUE,NUMDIAG,GENERATE
***ELEVEN INSTRUCTIONS ARE READ FROM TWO CARDS.
* JLINPXT DESIGNATES THE INPUT DEVICE (TAPE (0);CARDS(1);-GEN(16)).
* JLRSKIP IS THE NUMBER OF RECORDS TO SKIP ON TAPE(&JLINPXT=0).
* JLTRAN DESIGNATES THE TRANSLATOR(NONE(0),1,2,3,4, OR 5).
* JLNORM IS THE NORMALIZATION INDICATOR(NORM(1);DON'T(0)).
* KLENGTH IS THE NUMBER OF BYTES IN EACH KERNEL OR ELEMENT OF THE IR.
* NJOBS IS THE NUMBER OF BULK ITEMS TO BE STORED FOR EACH INFO. PATH.
***THE SECOND CARD HAS FIVE ITEMS FOR THE JOBLIST GENERATOR
* NTASKS IS THE NUMBER OF ITEMS IN THE JOBLIST.
* MVALUE IS THE VALUE OF 'M' FOR THE JOBLIST ITEM TO BE GENERATED.
* JVALUE IS THE MAXIMUM VALUE FOR EACH SMALL 'M' IN THE JOBLIST ITEM
* NUMDIAG IS THE TOTAL NUMBER OF ALL TYPES OF DIAGONALS IN THE ITEM.
* GENERATE IS THE ODD NUMBER TO BE USED BY RANDOM NUMBER GENERATORS.
***OVER-RIDE INFORMATION IS READ FROM CARDS NEXT.
LOOKFILE CALL2 SEARCH,RESULT,ANSWER
ANSWER  DISPENSE CARDREAD
CARDREAD CALL1 READC,TAPEREAD
* SUB-STRING COMMANDS ARE READ IN THE SREADC COMPONENT.
* NV IS THE NUMBER OF SUBSTRINGS PER STRING.
* SOS IS THE STATE-OF-SUBSTRING COMMAND.
* LSX IS THE "LOWEST LIMIT OF LEAST SIGNIFICANCE".
* SOS AND LSX TOGETHER DEFINE THE FORMAT OF SUBSTRINGS AND SIGNIFY THE
* PATH TAKEN THROUGH THE COMPRESSORS.

```

SOLIDE (CONT.)

```

* SOS<0 ;LSX#0 -----X FORMAT;DONT COMPRESS THE SUBSTRING.
* SOS<0 ;LSX=0 -----A FORMAT;DONT COMPRESS THE SUBSTRING.
* SOS=0 ;LSX=0 -----I FORMAT;COMPRESS WITH SNUPAK(INTERGER).
* SOS=0 ;LSX<0 -----E FORMAT;COMPRESS WITH SNUPAK(LEFT-SHIFT).
* SOS=0 ;LSX>0 -----E FORMAT;COMPRESS WITH SNUPAK(BIN METHOD).
* SOS>0 ;-----A FORMAT; DONT COMPRESS THE SUBSTRING.
TAPERREAD CLI INPXT+3,X'00'
          BNE COPAKST
          CALL1 READT,COPAKST
COPAKST MVC RSKIPS(4),ZERO
          COPAK OUTPUTN
OUTPUTN CALL1 OUTPUT,NEWJOB
          SUBME 6,3,3,10,7294,100000,JBLIST,1500,500,500,500,100,1
* &ADDL=6 IS THE NUMBER OF BYTES EACH MEMORY BLOCK
* &LSLOW=3 IS THE LENGTH OF THE SLOW PART OF COMPOSITE ADDRESSES.
* &LFAST=3 IS THE LENGTH OF THE FAST PART OF COMPOSITE ADDRESSES.
* &NTRKS=10 IS THE NUMBER OF TRACKS IN EACH MEMORY BLOCK.
* &TRKL=7294 IS THE NUMBER OF BYTES IN EACH TRACK OR RECORD.
***MEMORY-BLOCK SIZE IS &NTRKS TIMES &TRKL
* &LTHAYY=100,000 IS THE NUMBER OF BYTES IN THE PRINCIPAL DATA ARRAY.
* 4000 IS ABOUT THE MAXIMUM STRING LENGTH
* &JBLIST=JBLIST IS THE ADDRESS OF THE JOBLIST ARRAY.
* &LJBLIST=1500 IS THE LENGTH OF THE &JBLIST ARRAY.
* &LOVER1=500 IS THE LENGTH OF OVER-RIDE ARRAYS AOVER1, AOVER2, & AOVER3
* &LOVER2=500 IS THE LENGTH OF THE UPDATING ARRAY(AOVER2R).
* &LOVER3=500 IS THE LENGTH OF THE GATE ARRAY AOVER3R.
* &MATRIXL=100 IS THE NUMBER OF ITEMS IN EACH PRINCIPAL SUB-ARRAY.
* &MATRIXS=1 IS THE NUMBER OF ITEMS IN EACH SECONDARY SUB-ARRAY.
          END SOLIDE *THE END OF THE SOLIDE PROGRAM

```

SOLIDO

```

SOLIDO   START X'22C20'           *START OF THE SOLIDO CONTROL PROGRAM
        RESERVE 6,100000,5,1500
* &ADDL=6 IS THE NUMBER OF BYTES IN THE COMPOSITE ADDRESSES
* &LTHAYY=100,000 IS THE LENGTH OF THE PRINCIPAL DATA-ARRAY (YY)
* &TPCORD=5 IS THE NUMBER OF PERMANENT CORDS TO BE USED BY SANPAKC
* &LJBLIST=1500 IS THE LENGTH OF ARRAYS JBLIST AND JBWORK
NEWJOB   DEVICE INPXT,OUTPXT,RSKIPS,SLENGTH,LLENGTH,RNOS,TPCORD
* INPXT-----TAPE(0),,CARDS(1),TERMINATE(>1)
* OUTPXT-----TAPE(0),CARDS(1),PRINTER(>2 FOR OUTPXT-2 SKIPS).
* RSKIPS-----NUMBER OF STRINGS TO BE SKIPPED AT BEGINNING OF JOB.
* SLENGTH-----THE MINIMUM BYTES IN EACH SEGMENT (A TAPE COMMAND).
* LLENGTH-----THE NUMBER OF BYTES(<256) IN THE LABEL.
* RNOS-----THE NUMBER OF STRINGS TO BE PROCESSED(TAPE COMMAND).
* TPCORD-----THE NUMBER OF PERMANENT CORDS TO BE USED
STRING   STRING MODE,POSTOP,LEXCON,LEXMODE,LEXPCH
* MODE-----RETRIEVE(0),STORE(1),UPDATE(2,3,4).
* POSTOP-----TERMINATE(<1),NEWJOB(=1),STRING(0 OR <0).
* -----FOR POSTOP NEGATIVE THE LEXMODE AND LEXPCH COMMANDS ARE
* -----CHANGED TO THE FAST MODE AND NO PERMANENT CORDS ARE
* -----PUNCHED AFTER MINUS POSTOP STRINGS HAVE BEEN PROCESSED.
* -----THIS COMMAND IS FOR TAPE ONLY.
* LEXCON-----READ PERMANENT CORDS COMMAND (READ=1 ; NO CARDS=0).
* LEXMODE-----FAST-MODE (0); SLOW-MODE (1); EXTEND PCORDS TABLE (2).
* LEXPCH-----PUNCH PCORDS TABLE (0); DON'T PUNCH TABLE (NOT 0).
ITEMS    GETJLIST JLINPXT,JLRSKIP,JLTRAN,JLNORM,KLENGTH,NJOBS,NTASKS,X
        MVALUE,JVALUE,NUMDIAG,GENERATE
***ELEVEN INSTRUCTIONS ARE READ FROM TWO CARDS.
* JLINPXT DESIGNATES THE INPUT DEVICE (TAPE (0);CARDS(1);-GEN(16)).
* JLRSKIP IS THE NUMBER OF RECORDS TO SKIP ON TAPE(&JLINPXT=0).
* JLTRAN DESIGNATES THE TRANSLATOR(NONE(0),1,2,3,4, OR 5).
* JLNORM IS THE NORMALIZATION INDICATOR(NORM(1);DON'T(0)).
* KLENGTH IS THE NUMBER OF BYTES IN EACH KERNEL OR ELEMENT OF THE IR.
* NJOBS IS THE NUMBER OF BULK ITEMS TO BE STORED FOR EACH INFO. PATH.
***THE SECOND CARD HAS FIVE ITEMS FOR THE JOBLIST GENERATOR
* NTASKS IS THE NUMBER OF ITEMS IN THE JOBLIST.
* MVALUE IS THE VALUE OF 'M' FOR THE JOBLIST ITEM TO BE GENERATED.
* JVALUE IS THE MAXIMUM VALUE FOR EACH SMALL 'M' IN THE JOBLIST ITEM
* NUMDIAG IS THE TOTAL NUMBER OF ALL TYPES OF DIAGONALS IN THE ITEM.
* GENERATE IS THE ODD NUMBER TO BE USED BY RANDOM NUMBER GENERATORS.
***OVER-RIDE INFORMATION IS READ FROM CARDS NEXT.
LOOKFILE CALL2 SEARCH,RESULT,ANSWER
ANSWER   DISPENSE CARDREAD
CARDREAD CALL1 READC,TAPEREAD
* SUB-STRING COMMANDS ARE READ IN THE SREADC COMPONENT.
* NV IS THE NUMBER OF SUBSTRINGS PER STRING.
* SOS IS THE STATE-OF-SUBSTRING COMMAND.
* LSX IS THE "LOWEST LIMIT OF LEAST SIGNIFICANCE".
* SOS AND LSX TOGETHER DEFINE THE FORMAT OF SUBSTRINGS AND SIGNIFY THE
* PATH TAKEN THROUGH THE COMPRESSORS.

```

SOLIDO (CONT.)

```

* SOS<0 ;LSX#0 -----X FORMAT;DONT COMPRESS THE SUBSTRING.
* SOS<0 ;LSX=0 -----A FORMAT;DONT COMPRESS THE SUBSTRING.
* SOS=0 ;LSX=0 -----I FORMAT;COMPRESS WITH SNUPAK(INTERGER).
* SOS=0 ;LSX<0 -----E FORMAT;COMPRESS WITH SNUPAK(LEFT-SHIFT).
* SOS=0 ;LSX>0 -----E FORMAT;COMPRESS WITH SNUPAK(BIN METHOD).
* SOS>0 ;-----A FORMAT; DONT COMPRESS THE SUBSTRING.
TAPERREAD GLI INPXT+3,X'00'
          BNE COPAKST
          CALL1 READT,COPAKST
COPAKST  MVC RSKIPS(4),ZERO
          COPAK OUTPUTN
OUTPUTN CALL1 OUTPUT,NEWJOB
          SUBMG 100000,500,500,500,1500
* &LTHAYY=100,000 IS THE NUMBER OF BYTES IN THE PRINCIPAL DATA ARRAY.
* 4000 IS ABOUT THE MAXIMUM STRING LENGTH
* &LOVER1=500 IS THE LENGTH OF ARRAYS AOVER1, AOVER2, AND AOVER3
* &LOVER2=500 IS THE LENGTH OF THE UPDATING ARRAY(AOVER2R).
* &LOVER3=500 IS THE LENGTH OF THE GATE ARRAY AOVER3R.
* &LJBLIST=1500 IS THE LENGTH OF THE JBLIST AND JBWORK ARRAYS.
          END SOLIDO
*THE END OF THE SOLIDO PROGRAM

```

We claim:

1. Method of utilizing a digital computer system including memory and control sections comprising the steps of:

- a. storing in the memory a string comprising a set of multibit information units;
- b. identifying and storing in the memory a LEXICON table comprising Type 1 codes defined as units which are of the same format as the stored string units and which do not occur in the stored string;
- c. searching the stored string for the presence of a plurality of patterns of contiguous string units which patterns are repeated in the string, and identifying such repeated patterns;
- d. replacing each of the patterns of a plurality of repeated patterns identified in the preceding step by a unique Type 1 code from the LEXICON table;
- e. storing in the memory decompression information associated with the string and defining the replacement carried out in the preceding step; and
- f. storing in the memory a PCORD table containing one pattern from each plurality of identical patterns which have been replaced by a Type 1 code.

2. Method as in claim 1 including computing and storing in the memory a savings ratio indicative of the saving in the length of the stored string achieved through replacing by a Type 1 code string pattern identical to patterns stored in the PCORD table to thereby associate a savings ratio with each pattern stored in the PCORD table.

3. Method as in claim 2 including: defining a maximum value for the number of patterns that can be stored in the PCORD table; testing before storing a pattern in the PCORD table if the PCORD table is full; if the PCORD table is full, testing if the savings ratio of the pattern to be stored in the PCORD table is more favorable than the least favorable savings ratio of the patterns already in the PCORD table; and, if the answer is yes, removing the pattern with the least favorable savings ratio from the PCORD table and storing therein the pattern with the more favorable savings ratio.

4. Method of utilizing a digital computer having memory and control sections to decompress a string compressed by the method of claim 1 comprising:

- a. storing the compressed string in the decompression information in the memory;
- b. identifying from the decompression information a Type 2 code and the pattern replaced by it;
- c. locating in the compressed string Type 2 codes identical to the Type 2 code identified in the preceding step; and
- d. replacing each Type 2 code located in the preceding step by the pattern identified in the identifying step.

5. Method of utilizing a digital computer system including memory and control sections comprising the steps of:

- a. storing in the memory a string comprising a set of multibit information units;
- b. identifying and storing in the memory a LEXICON table comprising Type 1 codes defined as units which are of the same format as the stored string units and which do not occur in the stored string;
- c. storing in a PCORD table in the memory a list of PCORD patterns each composed of a plurality of units of the same format as the units of the stored string;
- d. searching the stored string for a plurality of patterns each composed of contiguous units and each identical to a PCORD pattern;
- e. identifying such patterns for replacement by Type 1 codes;
- f. replacing each of the patterns of a plurality of repeated patterns identified in the preceding step by a unique Type 1 code from the LEXICON table; and
- g. storing in the memory decompression information associated with the string and defining the replacement carried out in the preceding step.

6. Method as in claim 5 including defining prior to the searching step if the stored string is to be subjected to slow mode compression or to a fast mode compression, and — if slow mode compression is defined — searching in the searching step for a plurality of patterns each composed of contiguous units and identifying such patterns for replacement by Type 1 codes without reference to the PCORD patterns stored in the PCORD table, but — if fast mode compression is defined — searching in the searching step only for repeating patterns identical to PCORD patterns from the PCORD table.

7. Method of utilizing a digital computer system including memory and control sections comprising the steps of storing in the memory a string comprising a set of multibit information units; identifying and storing in the memory a LEXICON table comprising Type 1 codes defined as units which are of the same format as the stored string units and which do not occur in the stored string; searching the stored string for the presence of a plurality of patterns of contiguous string units which patterns are repeated in the string, and identifying such repeated patterns; replacing each of the patterns of a plurality of repeated patterns identified in the preceding step by a unique Type 1 code from the LEXICON table; storing in the memory decompression information associated with the string and defining the replacement carried out in the preceding step; including in said identifying and storing step the substep of identifying and storing in the LEXICON table Type 2 codes defined as units which are of the same format as the stored string units and which occur in the stored string at least a preselected number of times, and including the additional steps of: searching the stored string for string units identical to Type 2 codes stored in the LEXICON portion of the memory in the preceding step; constructing and storing in the memory a bit map identifying the locations in the stored string of units identical to a Type 2 code and occurring at least a preselected number of times in the string; removing from the string the units identified in the preceding step; and storing in the memory decompression information associated with the string and comprising the bit map and one of the removed string units.

8. Method of utilizing a digital computer system including memory and control sections comprising the steps of:

- a. storing in the memory a string comprising a set of multibit information units;
- b. identifying and storing in the memory a LEXICON table comprising Type 1 codes defined as units which are of the same format as the stored string units and which do not occur in the stored string;
- c. searching the stored string for the presence of a plurality of patterns of contiguous string units which patterns are repeated in the string, and identifying such repeated patterns;
- d. replacing each of the patterns of a plurality of repeated patterns identified in the preceding step by a unique Type 1 code from the LEXICON table; and
- e. storing in the memory decompression information associated with the string and defining the replacement carried out in the preceding step;
- f. storing in the memory a PCORD table containing preselected Type 2 codes each of the same format as the units of the stored string;
- g. searching the stored string for units which are identical to a Type 2 code and which occur in the string at least a preselected number of times;
- h. removing from the string units identified in the preceding step;
- i. constructing a bit map identifying the locations of the removed units; and
- j. storing in the memory decompression information associated with the string and comprising the bit map and one of the removed string units.

9. Method as in claim 8 including: defining prior to the step of searching the stored string for units identical to Type 2 codes if the stored string is to be subjected to fast mode com-

pression or to slow mode compression; if slow mode compression is defined identifying and storing in said identifying and storing step in the LEXICON table Type 2 codes defined as identical in format with the units of the stored string and occurring in the string at least a preselected number of times, searching the stored string for string units identical to a Type 2 code from the LEXICON table of the memory and occurring at least a preselected number of times in the stored string, and then proceeding to the bit map constructing step without recourse to the table of Type 2 codes; but if fast mode compression is defined, then searching the stored string for units identical to Type 2 codes from the PCORD table of Type 2 codes without recourse to the LEXICON portion of the memory.

10. Method of utilizing a digital computer having memory and control sections to decompress a string compressed by the method of claim 8 comprising the steps of:

- a. storing the compressed string and the decompression information associated with the string in the memory;
- b. identifying from the decompression information a Type 1 code and the pattern replaced by it;
- c. locating in the compressed string Type 1 codes identical to the Type 1 code identified in the preceding step;
- d. replacing each Type 1 code located in the preceding step by the pattern identified in step b;
- e. identifying from the decompression information a bit map and one of the string units removed in conjunction with constructing the bit map; and
- f. storing the removed string unit in the places in the string identified by the bit map and expanding the string accordingly.

11. Method of utilizing a digital computer system including memory and control sections comprising the steps of:

- a. storing in the memory a string comprising a set of multibit information units;
- b. storing in a PCORD table in the memory at least one PCORD pattern composed of a plurality of units which are of the same format as the units of the stored string;
- c. searching the stored string for the presence of a plurality of patterns each composed of contiguous string units and each identical to a PCORD pattern, and identifying such string patterns if any are found; and
- d. replacing each of the string patterns identified in the preceding step by a Type 1 code defined as a unit which is of the same format as the string units but which does not occur in the stored string.

12. Method of utilizing a digital computer having memory and control sections to decompress a string compressed by the method of claim 11 comprising the steps of:

- a. identifying a Type 1 code which has replaced a pattern and the pattern replaced by the code;
- b. replacing each Type 1 code identical to the identified code and occurring in the string by the pattern identified in the preceding step; and
- c. expanding the string by a number of units equal to the difference between the number of units in the replaced patterns and the number of Type 1 codes which have been replaced by patterns.

13. Method of utilizing a digital computer system including memory and control sections comprising the steps of:

- a. storing in the memory a string comprising a set of multibit information units;
- b. identifying and storing in a LEXICON table in the memory Type 2 codes defined as units which are of the same format as the stored string units and which occur in the stored string at least a preselected number of times;
- c. searching the stored string for units identical to a Type 2 code and identifying the locations in the string of such identical units;
- d. constructing and storing in the memory a bit map identifying the locations in the stored string of units identified in the preceding step;
- e. removing from the string the identified units; and

f. storing in the memory decompression information associated with the string and comprising the bit map and one of the removed string units.

14. Method of utilizing a digital computer having memory and control sections to decompress a string compressed by the method of claim 13 comprising the steps of:

- a. storing the compressed string and the decompression information in the memory;
- b. identifying from the decompression information a bit map and a string unit removed in conjunction with constructing the bit map; and
- c. inserting the string unit identified in the preceding step in the locations in the string identified by the bit map and expanding the string accordingly.

15. Method of utilizing a computer system including memory and control sections comprising the steps of:

- a. storing in the memory a string composed of a number of multibit information units;
- b. storing in a PCORD table in the memory Type 2 codes defined as units which are of the same format as the stored string units;
- c. searching the stored string for units which are identical to a Type 2 code from the PCORD table and which occur at least a pre-selected number of times, and identifying such units if any are found;
- d. constructing a bit map identifying the locations in the stored string of units identified in the preceding step; and
- e. removing from the string the identified units and storing in the memory together with the string the bit map and one of the removed string units.

16. Method of utilizing a digital computer having byte-oriented memory and control sections to compress information supplied in the form of a string comprising a set of bytes of information bits, comprising the steps of:

- a. using the value of each byte of the string to address a 256byte table in which each byte address corresponds to a unique one of the 256 possible bit configurations of a byte and each byte address contains a count of the number of times the byte address has been addressed;
- b. storing an indication of the address of each byte address of the 256 byte table that has not been addressed in the course of step (a) in a LEXICON table to compile thereby a set of Type 1 codes which are bytes that do not occur in the string;
- c. detecting the occurrence in the string of a group of non-overlapping patterns, if any, of R contiguous bytes (R is an integer greater than 1) which patterns are identical with each other;
- d. replacing each of the patterns detected in the course of step c. with an identical Type 1 code selected from available Type 1 codes in the LEXICON table and compressing the string to eliminate the space vacated because of the difference in length between each such replaced pattern of R bytes and the one byte Type 1 code replacing it;
- e. associating with the string decompression information comprising the Type 1 code used in the course of step d and one of the replaced patterns of R bytes;
- f. changing the value of R and repeating steps c through e for as long as both i the combined length of the Type 1 codes used as pattern replacements and the decompression information is less than the combined length of the replaced patterns and ii. previously unused Type 1 codes are available in the LEXICON table; and
- g. storing a pattern from each group of patterns of R bytes which has been deleted from the string in a PCORD table.

17. Method as in claim 16 including storing a pattern from each group of patterns of R bytes which has been deleted from the string in a PCORD TABLE; associating with each pattern in the PCORD table a savings ratio indicative of the degree of compression of the string resulting from the deletion of said pattern from the string.

18. Method as in claim 17 including: limiting the capacity of the PCORD table; checking whether the PCORD table is full

when an attempt is made to include therein a new pattern; and, in case the PCORD table is full, storing the new pattern in the PCORD table and deleting from the PCORD table the pattern having the lowest associated savings ratio, but only if said lowest savings ratio is lower than the savings ratio of the new pattern.

19. Method of utilizing a digital computer having memory and control sections to decompress a string compressed by the method of claim 16 comprising the steps of:

- a. storing the compressed string and the decompression information in the memory;
- b. identifying from the decompression information a Type 1 code and the pattern of R bytes replaced by it;
- c. locating in the compressed string Type 1 codes identical to the Type 1 code identified in the preceding step;
- d. replacing each Type 1 code located in the preceding step by the pattern of R contiguous bytes identified in step b; and
- e. expanding the string by a number of bytes equal to the difference between the number of bytes of the patterns replacing the Type 1 codes and the number of replaced Type 1 codes.

20. Method of utilizing a digital computer having memory and control sections to compress numeric information supplied in the form of a first string comprising a set of words A, B, C, D, E, . . . , each word containing a number, comprising the steps of:

- a. recursively differencing by
 - i. adding the absolute value of the contents of the words of the first string to obtain a first sum;
 - ii. generating from the first string a second string having the same number of words, each word except the first having a value equal to the difference between the correspondingly located word of the first string and the next preceding word of the first string, whereby the second string comprises words A, A-B, B-C, C-D, D-E, . . . , and adding the absolute values of the words of the second string to obtain a second sum;
 - iii. comparing the magnitudes of the first and the second sums and, continuing to step (b) if the first sum is less than the second sum, but continuing to substep (iv) if the magnitude of the first sum is greater than or equal to the magnitude of the second sum;
 - iv. generating from the second string generated in substep (ii) a third string in the same manner as in substep (ii), whereby the third string comprises words A, A-(A-B), (A-B)-(B-C), (B-C)-(C-D), (C-D)-(D-E), . . . , and repeating substeps i and ii by considering each newly generated string as a second string and considering the previously generated string as a first string;
- b. detecting sequences of identical words;
- c. determining if the replacement of such sequences by defined decompression information would result in saving in string length and proceeding to step (d) if saving is indicated but proceeding to step (e) if no saving is indicated;
- d. deleting each detected sequence of identical words and associating with the string decompression information comprising the value of the deleted word, the number of

deleted words and the address in the string prior to deletion at which the deleted sequence started, and compressing the string to take up the space vacated by the deleted words; and

- e. packing the words of the string into double words, each double word having a set of bits designating the total number of and a set of bits storing values of words packed in the double word, each word occupying a fixed number of bit positions, said fixed number determined by the number of significant bits of the highest value word packed in the double word.

21. Method as in claim 20 including converting floating point numbers contained in the words A, B, C, D, E, . . . , into integer numbers by a logical right shift truncation process.

22. Method as in claim 21 including: placing each integer number in the string into a four byte word; searching the string to find a minimum and maximum value of the words; storing said maximum and minimum values; determining the median value by dividing the sum of the minimum and maximum values by 2; recording said median value; subtracting the recorded median value from each word in the string; associating with the string decompression information defining the above steps; and storing the decompression information.

23. Method as in claim 21 including: providing a value LSX indicating the degree of accuracy at which the information stored in the string is to be maintained; dividing LSX by 2 and storing the result; finding the minimum value of the words in the string; subtracting from each word of the string the minimum value and dividing the difference by the value LSX over 2; and rounding by removing all digits to the right of the decimal point in the words leaving only the digits to the left of the decimal point.

24. Method as in claim 21 including: locating in the string composed of integer numbers each contained in a word any patterns of words which patterns are composed of a plurality of contiguous words identical to each other; replacing each such pattern by two new words one of which is a count of the number of the words repeated in the pattern and the other one of which is a copy of the word which is repeated in the pattern, and associating with the string a third new word indicating the location in the string of the pattern which is replaced by said two new words.

25. Method of utilizing a digital computer having memory and control sections to decompress strings compressed by the method of claim 20 comprising the steps of:

- a. storing in the memory the packed string and its decompression information;
- b. unpacking the packed string double words by reference to the set of bits designating the total number of words packed in the double words and to the set of bits storing the values of words to generate unpacked words;
- c. if sequences of words were deleted in the course of compressing the string, utilizing the decompression information to replace in the string the deleted words; and
- d. if second or subsequent string were generated during compression, carrying out the reverse of the second and subsequent string generating step to regenerate the original string of words A, B, C, D, E,

* * * * *

5
10
15
20
25
30
35
40
45
50
55
60
65
70
75