Licensed Material—Property of IBM

IBM System/3 Model 6
System/3 BASIC Logic Manual

Program Number 5703-XM1

LY34-0001-1
File No. S3-23

Program Product

Second Edition (November 1971)

This is a major revision of, and obsoletes, LY34-0001-0 and Technical Newsletters LN34-0020 and LN34-0032. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the changes.

This edition applies to version 1, modification level 4 of IBM System/3 Model 6 System/3 BASIC, Program Product 5703-XM1, and to all subsequent program changes until otherwise indicated in new editions or Technical Newsletters. Changes are periodically made to the information herein. Before using this publication in connection with the operation of IBM systems, refer to the latest System/3 Bibliography, Order No. GC20-8080, for the editions that are applicable and current.

Some illustrations in this manual have a code number in the lower corner. This is a publishing control number and is not related to the subject matter.

Requests for copies of IBM publications should be made to your IBM representative or the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, send your comments to IBM Corporation, Publications, Department 245, Rochester, Minnesota 55901. Comments become the property of IBM.

IBM Technical Newsletter

This Newsletter No.	LN21-7729
Date	November 15, 1973
Base Publication No.	LY34-0001-1
File No.	S3-23
Previous Newsletters	LN34-0075

IBM System/3 Model 6
System/3 BASIC Logic Manual

This Technical Newsletter, a part of version 01, modification 05 of *IBM System/3 Model 6, System/3 BASIC,* provides replacement pages for the subject publication. These replacement pages remain in effect for subsequent versions and modifications unless specifically altered. Pages to be inserted and/or removed are:

Cover, edition notice
3-165, 3-166
6-3, 6-4
7-3, 7-6
Reader's Comment Form

Changes to text and illustrations are indicated by a vertical line at the left of the changes; new or extensively revised illustrations are denoted by the symbol ● at the left of the caption.

Summary of Amendments

Miscellaneous changes.

Note: Please file this cover letter at the back of the manual to provide a record of changes.

IBM Corporation, Publications, Department 245, Rochester, Minnesota 55901

IBM Technical Newsletter

System System/3

Base Publ. No. LY34-0001-1

This Newsletter No. LN34-0075

Date January 1972

Previous Newsletter Nos. None

IBM System/3 Model 6 System/3 BASIC Logic Manual

This Technical Newsletter, a part of version 1, modification level 4, of IBM System/3 Model 6 System/3 BASIC, Program Product 5703-XM1, provides replacement pages for the subject publication. These replacement pages remain in effect for subsequent versions and modifications unless specifically altered. Pages to be inserted and/or removed are:

cover through iv	3-59, 3-60	3-203, 3-204
1-1 through 1-4	3-65, 3-66	3-215, 3-216
2-1 through 2-4	3-81, 3-82	3-222.1, 3-222.2 (added)
3-1, 3-2	3-91, 3-92	3-223, 3-224
3-21, 3-22	3-105, 3-106	4-1 through 4-6
3-27 through 3-30	3-139, 3-140	5-3 through 5-8
3-33, 3-34	3-179, 3-180	5-17, 5-18
3-34.1, 3-34.2 (added)	3-183, 3-184	6-5 through 6-8
3-40.1, 3-40.2 (added)	3-184.1, 3-184.2 (added)	7-3 through 7-6
3-46.1, 3-46.2 (added)	3-185 through 3-188	X-1 through X-18

A technical change to the text or to an illustration is indicated by a vertical line to the left of the change.

Summary of Amendments

- A procedure file has been added to save program statements, data file lines, system and utility commands, comment lines, and procedure file lines. This procedure file can be executed later by a CALL command.
- A string function has been added for use in LET and IF statements to allow characters in character data items to be extracted, concatenated, replaced, or compared.
- An IBM 129 Card Data Recorder can now be attached to a System/3 Model 6 to read and punch 80-column cards.
- Minor text maintenance changes have also been made throughout the manual.

Note. Please file this cover letter at the back of the manual to provide a record of changes.

IBM Corporation, Systems Publications, Boca Raton, Florida 33432

Contents

Preface

This publication describes the internal logic and specifications of System/3 BASIC, a program product, for the IBM System/3 Model 6. The manual is designed to satisfy the documentation requirements of support personnel responsible for maintenance of System/3 BASIC.

Section 1, "Introduction," contains a general description of the modes of operation, functions, and characteristics of the programming system and the machine configuration.

Section 2, "Method of Operation," describes the functional flow of the program logic and data. Illustrations and supporting text trace the functional flow of the stand-alone computing system from input, through processing stages, to desired results (output). The usage of primary data areas is emphasized.

Section 3, "Program Organization," describes how the programs and routines that comprise System/3 BASIC are interconnected, and describes the functions of components. Because of the interactive environment, function level flowcharts are used extensively to describe complex programs.

Section 4, "Directory," contains a cross-reference table of all system components, for quick reference to System/3 basic assembly listings on microfiche. This section also defines source module labeling conventions and system equates.

Section 5, "Data Area Formats," contains detailed layouts of system data areas (communications area, directory formats, record formats, error-recording formats, parameter formats, etc.).

Section 6, "Diagnostic Aids," describes the maintenance utility program, program temporary fix (PTF) commands, and other useful servicing information.

Section 7, "Object Program," describes the interpreter/compiler functions, including a method for laying out the contents of an execution-time disk dump of virtual memory, the method for determining the contents of an execution-time core dump, and pseudo-machine-language formats.

Appendix A, "System/3 Basic Assembler Language," contains mnemonic operation code lists, instruction format descriptions, and an assembler instruction reference table.

Other publications related to this manual are:

IBM System/3 Disk System Basic Assembler Manual, SC21-7509

IBM System/3 Model 6 Components Reference Manual, GA34-0001

IBM System/3 Model 6 System/3 BASIC Reference Manual, GC34-0001

IBM System/3 Model 6 System/3 BASIC Operator's Guide, GC34-0003

IBM System/3 Model 6 System/3 BASIC Reference Handbook, GX34-0001

IBM System/3 Model 6 System/3 BASIC Desk Calculator Reference Card, GX34-0002

Section 1. Introduction

System/3 BASIC is a conversational, stand-alone, programming system oriented toward mathematical problem solving. It offers two modes of system operation:

- BASIC
- Desk calculator (DCALC)

BASIC MODE OF OPERATION

BASIC mode is programmed with a conversational programming language which is also called BASIC. The user develops program, data, and procedure files in an interactive environment; that is, he communicates with the system programs by entering (through the keyboard or the data recorder) BASIC statements, data-file lines, system commands, utility commands, and procedure-file lines. BASIC statements form BASIC programs, data-file lines specify the content of a data file, system and utility commands request immediate system action (except when entered as lines of a procedure file), and procedure-file lines (composed of BASIC statements, data-file lines, and commands) specify system actions to be performed at a later time.

A BASIC statement is a single line identified by a line number. Lines may be entered in any order and are automatically collected into a program file and ordered with respect to line number. Each statement is syntax checked as it is entered. Syntax is the specified way in which words and characters are combined in program-statement lines, data-file lines, and command lines. When errors in syntax are detected, an error message is generated and printed on the system printer.

A BASIC program is completely compiled when execution is specified by a system command. Line numbers provide a simple program editing facility by allowing the replacement of previous lines with new or null lines, and the insertion of new lines into the file. Data files may also be created and modified in the same manner as program files. The user develops a procedure file by entering lines that begin with a line number and are followed by either a system or utility command, a BASIC statement, a data-file line or another procedure-file line. Procedure file lines are syntax checked when they are called for execution, not as they are entered into the work file.

System commands are one keyboard line and are distinguished by lack of a statement number. They may be intermixed with program or data file lines in any manner. Each command is a unique keyword and an optional parameter list in a free-form format. Keyword commands have the following system functions:

- File editing.
- Initialization/modification of program execution.
- File library creation and management (source programs and data files may be saved on disk in either a private or pooled library).
- Disk utility functions.

Programs in the system program file (system program area) analyze the file and command lines, that the user enters, for syntax errors. These programs also perform the operations specified by command lines and the command keys (located on the left side of the keyboard).

The user has several ways to correct errors he made while entering BASIC statements, data-file lines, procedure-file lines, system commands, and utility commands. Also, these same procedures can be used to correct BASIC program errors that the system finds during program execution.

System/3 BASIC provides the user with execution time debugging aids:

- Trace mode.
- Step (one statement at a time) execution mode.
- Display and change program variables during execution.
- Interrupt and suspend execution at any point, perform other system functions, and later resume execution.

System/3 BASIC also provides the user with several utility functions. These functions use utility commands and include operations such as system generation and disk initialization, and they assign space on disks for work areas and libraries.

System/3 BASIC provides to the IBM customer engineer a maintenance utility aid program with ten options for diagnosing and correcting problems in the system. Program temporary fix (PTF) commands (used to apply PTF patches), an I/O parameter list save area. and other maintenance features are discussed in Section 6.

DESK CALCULATOR (DCALC) MODE OF OPERATION

DCALC permits the user to add, subtract, multiply, divide, compute powers and roots, and perform many other mathematical functions without using a programming language. The numeric keys and the first eight command keys of the keyboard are used with DCALC. The mathematical functions in the system are requested by entering the name of the function through the typewriter keyboard.

MINIMUM MACHINE CONFIGURATION

The minimum machine configuration required to operate System/3 BASIC is as follows:

- An IBM 5406 Processing Unit Model B2 (8k main storage) and the first eight command keys.
- An IBM 5444 Disk Storage Drive Model 1 with one fixed disk and one removable disk containing a total storage capacity of 2,457,600 bytes.
- An IBM 5213 Printer Model 1 with a 13-inch carriage and 132 print positions or a 2222 Printer Model 1 with a 22-inch carriage and 220 print positions.

SUPPORTED OPTIONAL DEVICES

Optional IBM devices supported by System/3 BASIC are:

- 5406 Processing Unit Model B3 (12k main storage).
- 5406 Processing Unit Model B4 (16k main storage).
- 5444 Disk Storage Drive Model 2 with one fixed disk and one removable disk (4,915,200 bytes).
- Two 5444 Disk Storage Drives Model 2 with two fixed disks and two removable disks (9,830,400 bytes).
- One 5444 Disk Storage Drive Model 2 with one fixed disk and one removable disk, and one 5444 Disk Storage Drive Model 3 with one removable disk (7,372,800 bytes).
- 5213 Printer Model 2 (13-inch carriage, 132 print positions).
- 5213 Printer Model 3 (13-inch carriage, 132 print positions with bidirectional printing).

- 2222 Printer Model 1 (22-inch carriage, 220 print positions).
- 2222 Printer Model 2 (22-inch carriage, 220 print positions with bidirectional printing).
- 5496 Data Recorder Model 1 (with the System/3 Model 6 Attachment).
- 2265 Display Station Model 2 (this requires 12k main storage and eight additional command keys).
- 129 Card Data Recorder (with the Card Input/Output Attachment for the System/3 Model 6).

FLOWCHARTING TECHNIQUES

This PLM (program logic manual) has two flowcharting techniques:

1. Function level—Shows the sequence of major internal objectives of complex programs (an example is Figure 3-22). Process blocks are keyed to the program listing with a label, if a label exists at that logical point. Process blocks contain a list of functions executed to accomplish the major objectives within the logical flow of the program. The language within the block is understandable at a level external to the program, so the flowchart serves as an index to the program listing. No attempt is made to maintain internal linkage between program label and physical sequence of instructions if this would interfere with the most logical presentation of the program. Every attempt has been made to create logical linkages for major objectives and to display the program on one page or facing pages.
   The on-page connector symbol is self-evident (refer to Figure 3-22); the off-page connector symbol usage is simplified, because these symbols refer only to another page of the same figure number. Reference (linkage) to other programs is made using a terminal flowcharting symbol containing the entry label for that program.
2. Conventional—Shows the sequence of major internal objectives of complex subroutines, IOCS routines, interfaces, error-logging overlays, etc. (an example is Figure 3-9). A predefined process block indicates a subroutine which is flowcharted elsewhere in this publication. The subroutine label is in the upper left corner; the flowchart figure number for that subroutine is in the upper right corner. On-page connectors, off-page connectors, and terminal symbols are used the same as on the function-level flowcharts.

SYMBOLIC LABEL (PROGRAM COMPONENT NAME)

A symbolic label (usually six characters) is used to identify each major System/3 BASIC program component. This label appears in the heading of each page in the assembly listing and on the microfiche. All labels within the same source module are prefixed by a character that identifies the type of source module. Section 4 contains a listing and a description of System/3 BASIC components.

Section 2. Method of Operation

This section describes the functional flow of the program logic and data for System/3 BASIC. Illustrations trace the functional flow from input, through processing stages, to output, emphasizing the use of primary data areas.

LOGICAL DIVISION OF SYSTEM PROGRAMS AND COMPONENTS

In this manual, System/3 BASIC is logically divided into six major groups of programs:

- Control
- Keyword
- Utility
- Compiler/loader
- Interpreter
- Desk calculator

Figure 2-1 illustrates the primary relationships between these programs and the disk data files. This figure also illustrates primary I/O flow from system input to system output.

Control

This group of programs perform the following primary functions:

- System initialization, or initial program load (IPL).
- Linkage and services, for transient programs, that must be core-resident (system nucleus) at all times.
- Acceptance and syntax checking of all system input while in conversational mode of operation.
- Maintenance on the work file (#GUFUD).
- Analysis of system commands and initiation of their execution (#ECMAN).
- Display of messages for errors and for operator communications.
- Program interruptions.

Keyword and Utility

- Each system command is associated with a transient program that performs or initiates the particular function described by the keyword or utility command.
- The keyword and utility programs are invoked by the command analyzer when the corresponding keyword (ALLOCATE, CHANGE, etc.) is entered.

Compiler/Loader

- Compilation of a BASIC language program is invoked by certain system commands.
- The source program in the work file is compiled in a single pass over the source statements.

Page of LY34-0001-1
Revised January 1972
By TNL LN34-0075

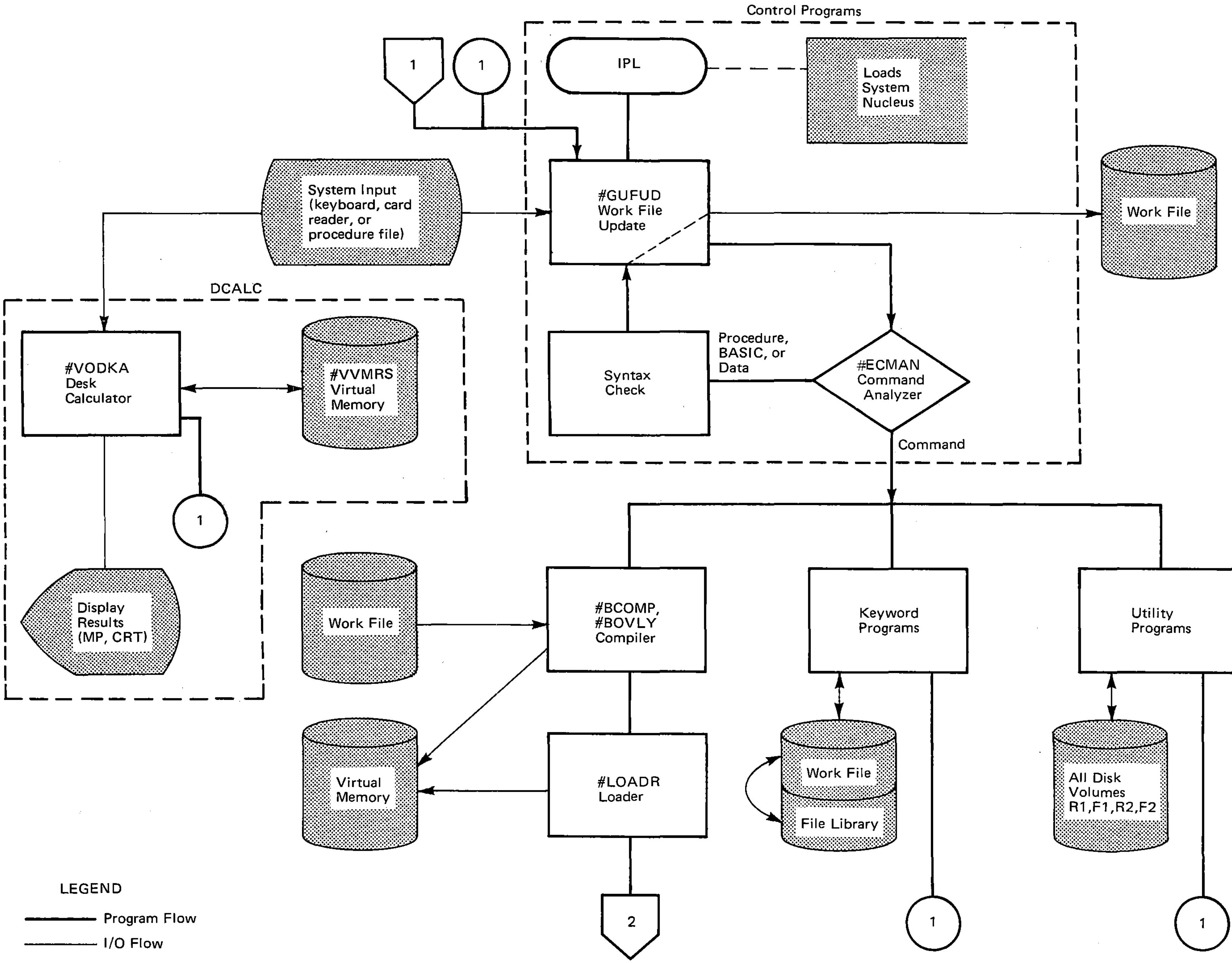


Figure 2-1. System Flow (Part 1 of 2)

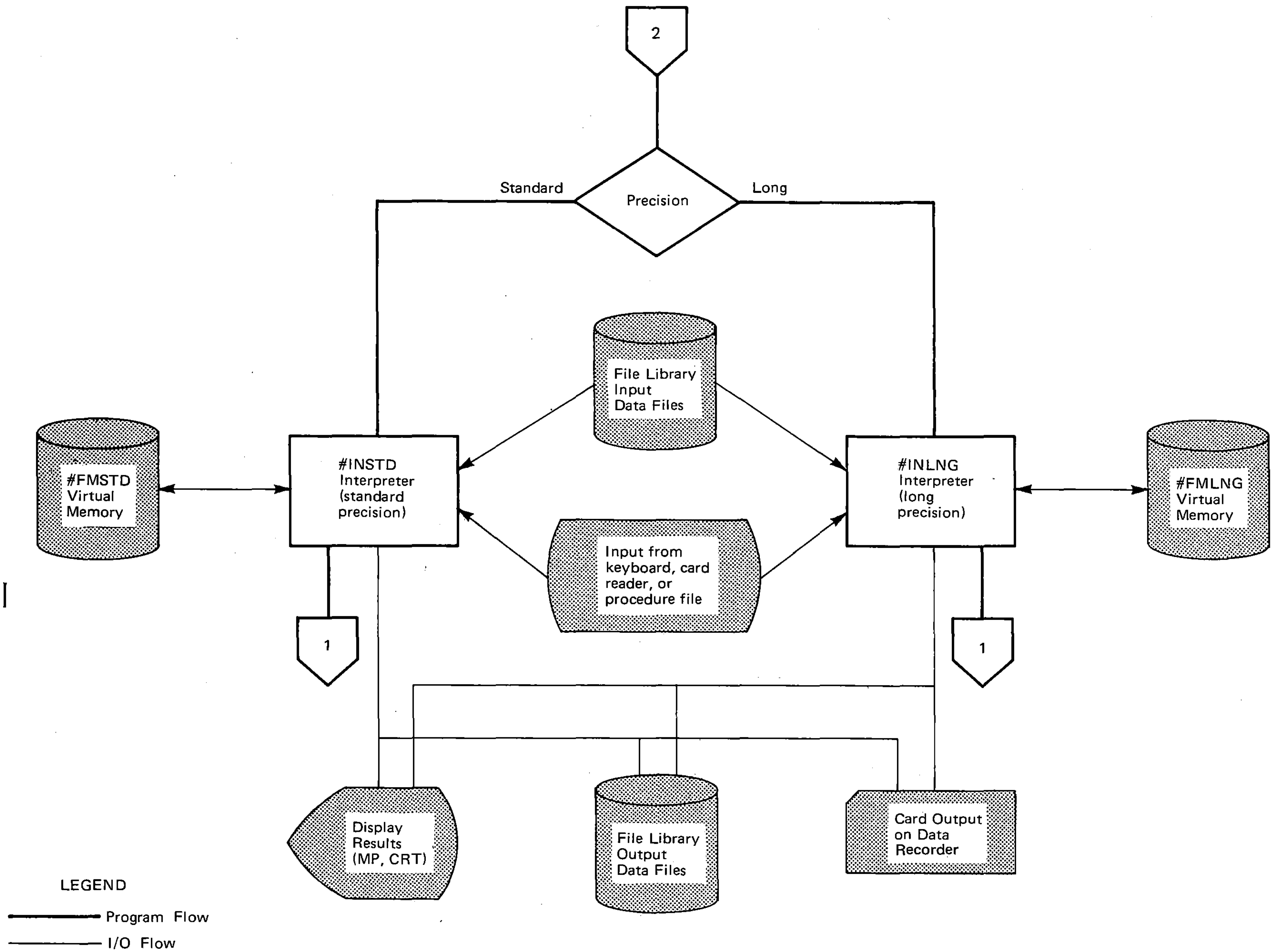


Figure 2-1. System Flow (Part 2 of 2)

Page of LY34-0001-1
Revised January 1972
By TNL LN34-0075

- The loader resolves addressing and allocations in virtual memory (disk) that cannot be resolved during the single compiler pass.
- Output from the compilation is a pseudo-machine-language program (object program) in virtual memory.

Interpreter

- This program produces the output from a BASIC language program by executing the pseudo-machine-language program in virtual memory.
- Each pseudo-machine-code (PMC) instruction within the pseudo-machine-language program is analyzed, one at a time.
- Subroutines perform the function specified by each PMC instruction.
- These subroutines reside in both core and virtual memory.
- A paging subroutine (part of the interpreter) performs a linkage function to load subroutines, PMC instructions, and data into core from virtual memory.

Desk Calculator (DCALC)

- This program accepts DCALC input from the keyboard, and then pages (transfers in sections) appropriate subroutines into core from virtual memory to execute the functions for the user.
- This program uses the same concepts as that of the interpreter.
- Error messages, operator communications, and I/O operations are provided by DCALC.

DISK ORGANIZATION

The first four cylinders on every disk volume are reserved for the volume information cylinder (cylinder 0) and alternate data tracks (cylinders 1, 2, and 3). The volume information cylinder contains the volume label and volume table of contents, used for volume identification, and other pertinent information about the volume (refer to Figures 5-9 and 5-10). A disk volume used for System/3 BASIC is optionally formatted with these primary disk areas (system files):

- System work file (system work area)
- System program file
- System library file (file library)
- Help text file
- PTF file (program temporary fix)

Refer to Figure 2-2 for an example of system file placement on disk and Figure 5-2 for the disk volume format. For details on file organization and data formats, refer to the Table of Contents or the Index for the particular subject you are interested in.

Volume Label

This reserved sector provides volume identification information, disk addresses, and size of System/3 BASIC system files on the volume. The sector is used by System/3 BASIC programs to locate these system files. The volume label also points to the location of the volume table of contents (VTOC). Refer to Figure 5-9 for the format of the volume label.

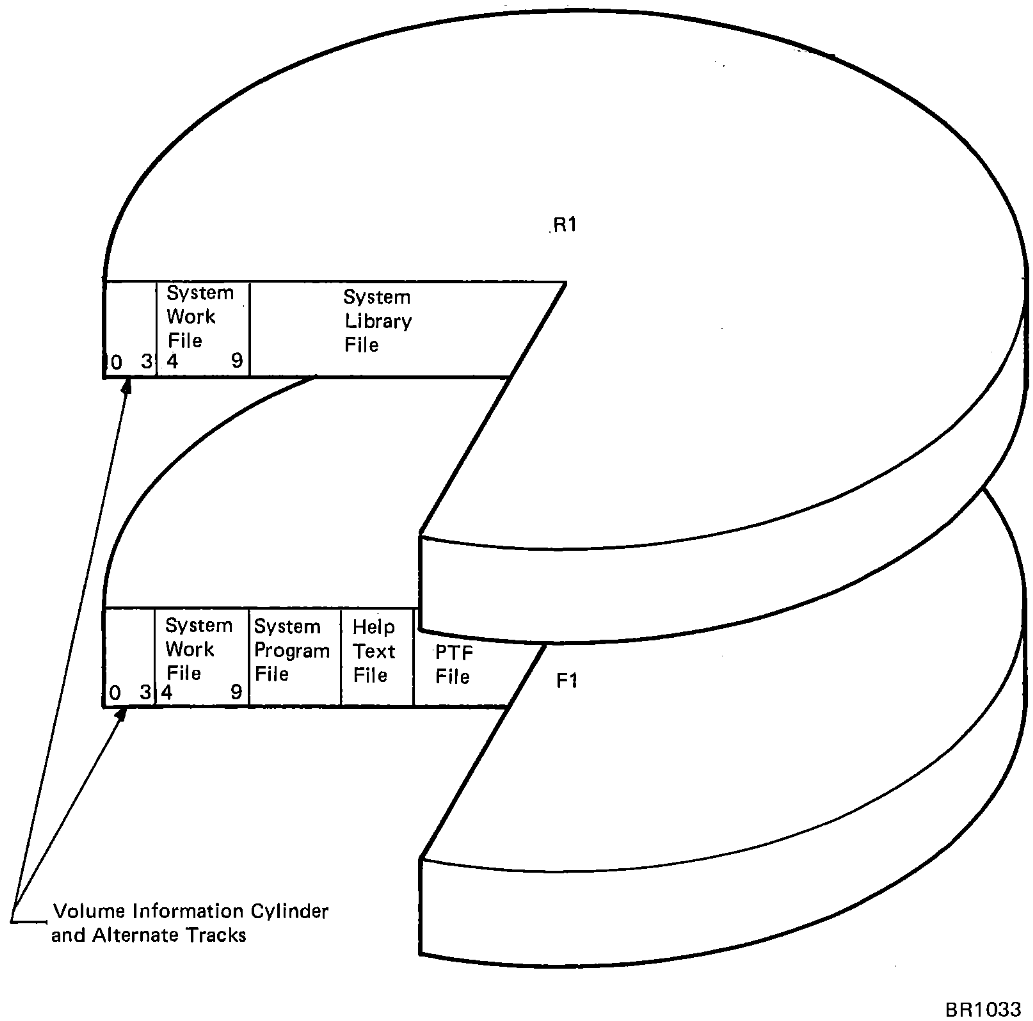


Figure 2-2. System Files, Example

Volume Table of Contents

The VTOC contains labels for all system files on the volume. Each label contains the name of the file, and disk extent information necessary to protect the area occupied by the file from other programming systems. Protection is handled by the track usage mask in the volume label. The VTOC is maintained by System/3 BASIC but it is not used to locate System/3 BASIC system files. Refer to Figure 5-10 for the format of the volume table of contents.

Note: Do not confuse system files with user files in the file library.

System Work File

This system file, also referred to as the system work area, is allocated on cylinders 4 through 9 on both volumes residing on drive 1. The file is accessed by system programs as a four-track logical file. When accessing the file, relative disk addresses are computed on the basis of 96 sectors per cylinder instead of 48. The system work area contains these four areas:

1. Selected system programs (cylinder 4)—Selected system programs are copied here from the system program file to reduce seek time.
2. Work file (cylinders 5 and 6)—This area is used for working with user program or data files.

3. Virtual memory (cylinders 7 and 8, and more than half of cylinder 9)—This area has a data length of 64k (256 sectors).
4. Temporary disk work area (last 32 sectors of cylinder 9)—This area provides programs with disk working storage.

The following selected system programs are copied to cylinder 4:

#ECMAN—Command analyzer
#GUFUD—Work file update/crusher program
#SFSYN—BASIC statement syntax checker
#SDSYN—Data syntax checker
#ERRPG—Error message program
#SFFIN } Execution-time disk I/O overlays
#SFLOA }
#BOVLY—Statement processor overlays

Refer to "System Work Area Equates (@WKAEQ)" in the program listings for disk addresses and sector counts associated with the system work file.

System Program File

This system file contains all system programs and related components, except those residing on the volume information cylinder (cylinder 0). All of the programs and components in this file are at fixed locations, relative to the first sector allocated for the file. None can be deleted from, or relocated in, the file. The first component in this file is a directory containing the relative disk address, sector count, and core load address of all components in the file, but this directory is not used by the system to locate the components. It is used for finding addresses of components when PTF commands are issued. (See Figure 5-29 for the format of a directory entry.) Relative disk addresses, sector counts and core load addresses of system components are assembled in the programs when and where they are needed. The starting disk address of this file is located in the volume label (see Figure 5-9).

System Library File

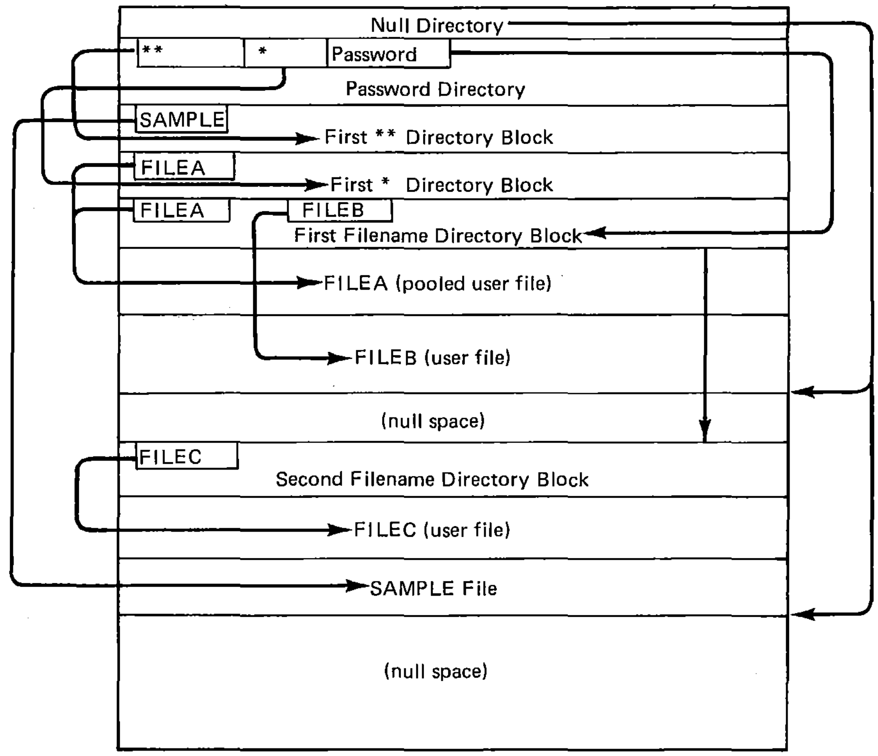
This system file, also referred to as the file library, contains space for storing user programs and data (see Figure 5-11).

Each grouping of user program statements or data statements stored in this library is called a user file and has an associated filename. The user files are accessed by the use of filename directories (Figure 5-14) and a single password directory (Figure 5-13). The password directory contains one password for each filename directory in the library.

Two reserved passwords are always present in the password directory. These reserved passwords are * (one-star) and ** (two-star). The user of the system will refer to these as one-star library (or pooled) and two-star library. These two passwords point to a directory of filenames as do the other passwords (see Figures 5-11 and 5-13).

The system library file also contains a null directory (Figure 5-12). This directory has entries pointing to all unused areas in the file. When the file is packed, there is only one entry, pointing to one null area at the end of the file. The null directory occupies the first sector allocated to the system library file. The starting disk address and size of this file are located in the volume label. Refer to Figure 2-3 for the organization of the system library file and its directories.

Refer to "Record Format" for the format of user program-generated and keyboard-generated files in the system library file. Program-generated files are considered as one record without a line number. This single record is written into sequential disk sectors as it is created by the program. No file index table (FIT) is generated for program-generated files.



Note: Refer to Figures 5-11 through 5-14 for directory formats.

BR1034

Figure 2-3. Organization of System Library File, Example

Help Text File

This system file (refer to Figures 5-21 and 5-22) contains all of the help text accessed by the HELP keyword program. This file is organized with an index starting in the first allocated sector. The general organization of this file is the same as that of the work file. The starting disk address of this file is located in the volume label (Figure 5-9). Refer to program listings #T1HEL, #T2HEL, etc., for the content of the help text file.

PTF File

This system file, if present, contains program temporary fixes to be applied to other system files as they are shipped from the IBM Program Information Department (PID). These PTF's may be applied by an IBM customer engineer. The starting disk address and size of this file are located in the volume label (Figure 5-9).

Work File

This is the work file referred to during system operations. It holds the current program or data file being entered or operated upon by the operator. The work file is logically addressed forward. All of cylinders 5 and 6, in the system work area, are allocated to the work file.

Both program-generated files and keyboard-generated data files have the same internal organization. A file consists of two parts:

1. File index table (FIT)—Used to randomly access the data records using the line number. This table can be from 1 to 3 sectors in length depending on the size of the data portion of the work file (see Figure 5-16). The first three sectors of the work file are always reserved for FIT.

2. Data area—Contains the data records (lines) in logical order with respect to line number. This portion of the file can be from 1 to 189 sectors in length.

All sectors of the work file are contiguous, including the index. The I/O information record (file directory 1, Figure 5-17) resides on cylinder 4, for a program file occupying the work file (see Figure 5-15), but is placed between the FIT and the data area if the file is copied (saved) to the file library.

All of the user program and keyboard-generated data files in the system library file were at one time saved from the work file. The format of user files in that library is the same as the format of the work file.

Note: The data portion of the file is organized into disk blocks (sectors). Record or line refers to a logical data segment as opposed to a physical disk block.

VIRTUAL-MEMORY CONCEPT

Virtual memory (VM) is a concept that uses disk to logically increase the size of the object program beyond the core capacity of the system. The VM concept also allows the core capacity of the system to be increased with no effect on the object program except for increased throughput of the system. The disk area occupied by virtual memory is cylinders 7 and 8, and more than half of cylinder 9 in the system work area. This area is a logical four-track file with a data length of 64k (256 sectors).

Sections of the object program are brought into available core from VM on an as-needed basis. These sections are referred to as pages. Each page is 256 bytes in length. The available core (core paging area) is divided into pages (Figure 2-4) which contain machine executable codes or data, as required for the execution of the object program.

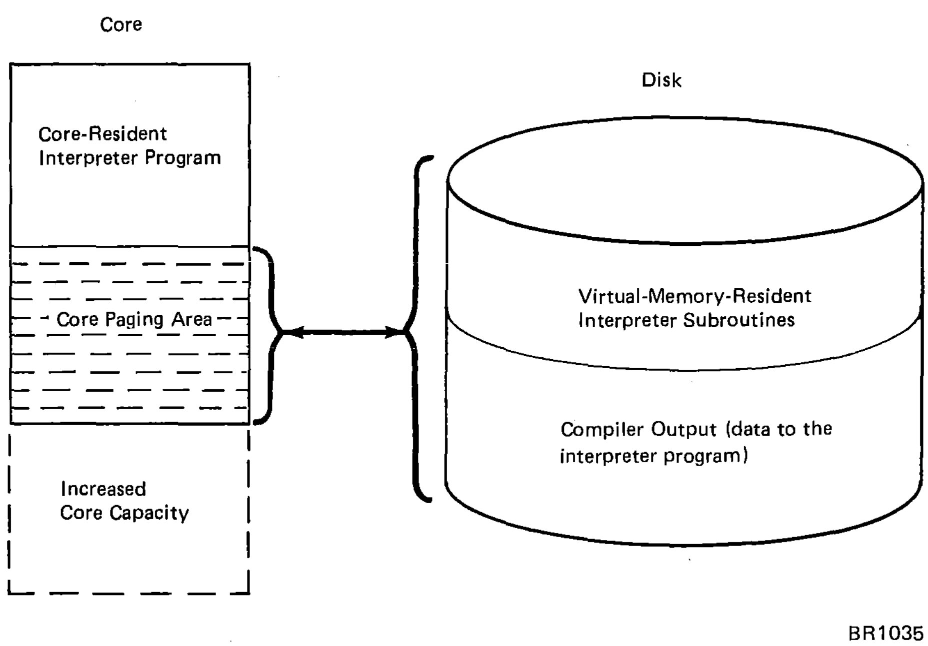


Figure 2-4. Virtual Memory Concept

The larger the core storage (core paging area), the less disk I/O activity occurs and the faster the object program is executed. Speed depends upon the actual size and the precision of the object program in relation to the size of the core paging area. Core configuration and relative paging area are discussed in detail in "Expanded Core Utilization" (Section 3). Detailed specifications of virtual memory and references to core paging are given in Section 7.

PSEUDO MACHINE LANGUAGE CONCEPT (PSEUDO OBJECT PROGRAM)

A pseudo machine language (object program) concept speeds the compilation time of a user program. It reduces the quantity of instruction output by the compiler and eliminates the necessity for an assembly pass or passes over the output instructions.

The pseudo instructions that make up the pseudo machine language (object program) invoke the execution of preassembled machine-language execution subroutines to perform the functions indicated by the pseudo instructions. This concept (Figure 2-5) is similar to the emulation of an instruction set foreign to the object machine, or the execution of machine instructions by hardware microprogramming.

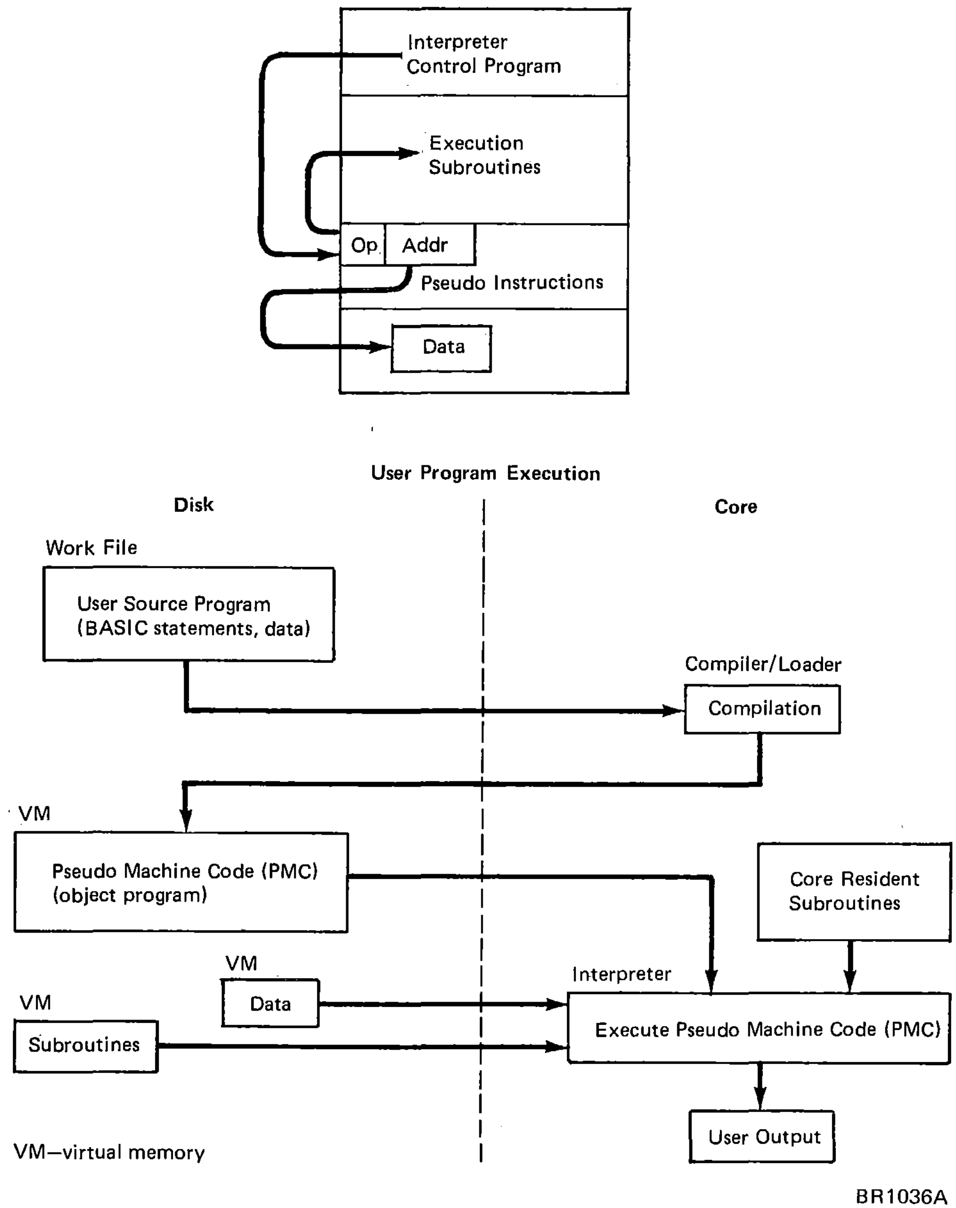


Figure 2-5. Pseudo Machine Language Concept

The pseudo machine language for System/3 BASIC contains pseudo instructions (Figure 3-169) to perform (1) arithmetic operations such as exponentiation, square root, trigonometric functions, logarithms, etc.; (2) array processing operations such as matrix multiply, inversion, transposition, determinant, etc.; and (3) I/O operations such as GET, PUT, PRINT, etc. All arithmetic operations are performed in either standard or long precision floating point arithmetic. Character (EBCDIC) data format is also processed by particular pseudo instructions.

The preassembled machine-language execution subroutines and their control program are referred to in this manual as the interpreter. The execution subroutines that are used least often are located in virtual memory (Figure 2-6). The pseudo machine language object program is generated by the compiler (and loader) and executed by the interpreter (Figure 2-5). Pseudo machine instructions are referred to in some areas of this manual as pseudo machine code (PMC) (refer to Figure 3-169). Detailed specifications of the object program are given in Section 7.

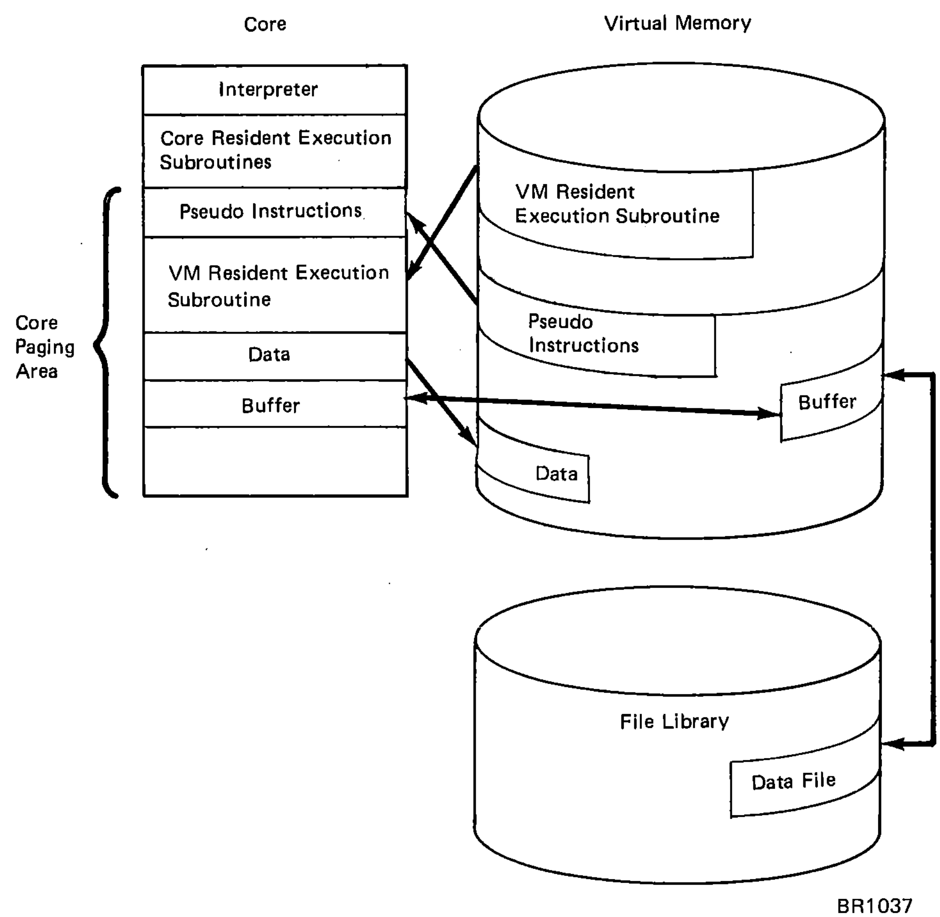


Figure 2-6. PMC and VM Concepts Combined

RECORD FORMAT

Data records in either the work file or the file library are variable-length records corresponding to one keyboard line. A record consists of one or more segments. A segment is the portion of a record contained in one disk block. Records are packed contiguously in the file and span disk block boundaries. A record spanning two disk blocks consists of two segments. Every segment is preceded by a segment descriptor field (SDF). Only the first segment contains the line number and statement type code. A program-generated data file has no line number or segment structure.

Refer to Figure 5-15 for the structure of a sample BASIC program file. A data file does not contain a file directory 1 record. Note that relative data blocks 04 and 05 are not in physical sequence. The line numbers (LINE) in the file index table entries are in ascending line number order allowing the relative data blocks (DB) to be referenced in logical order.

FILE DIRECTORY 1 (I/O INFORMATION RECORD)

This directory contains information, specified by ALLOCATE system commands, defining the data files referenced in the GET and PUT statements of a BASIC language program. This directory is associated only with a program file. For a program file in the work file, this directory resides on cylinder 4. When a program file is saved in the file library, this directory is placed between the FIT and the data blocks of the program file. Refer to Figure 5-17 for the file directory 1 format.

The first 8 entries of this directory occupy the first page in virtual memory during execution of the BASIC program. A maximum of 4 additional entries can be placed in virtual memory, by the program #LOADR, at a variable location.

Section 3. Program Organization

This section divides System/3 BASIC into these major groups of program components:

- Control programs
- Keyword programs
- Common subroutines
- Utility programs
- Maintenance utilities
- Compiler
- Loader
- Interpreter
- Desk calculator

For details on source module labeling conventions and system equates, refer to Section 4.

CONTROL PROGRAMS

System/3 BASIC control programs are defined under the following headings:

- System initialization–IPL
- System nucleus
- Error message program–#ERRPG
- Program interruption processor–#EXMSG
- Work file update/crusher–#GUFUD
- Command analyzer–#ECMAN
- Command key processor–#EFKEY
- BASIC statement syntax checker–#SFSYN
- Data syntax checker–#SDSYN
- Procedure line checker–#SPSYN
- Conversational I/O routines–#DPRIN, DPRINT, DEPRES
- Procedure file line processor–#GRAPR
- Card reader I/O routine–#DREAD
- Maintenance program load trace–#ZTRAC

System Initialization–IPL (Figure 3-1)

IPL is accomplished by three program components:

- Bootstrap loader–#MLOAD
- Interface routine; part of the system nucleus at IPL time–MOPPET
- Nucleus initialization program residing in the system program file–#MIPPE

IPL Bootstrap Loader—#MLOAD (Figure 3-1)

- This program is read from cylinder 0, head 0, sector 0 of disk when the program load switch is operated.
- #MLOAD first relocates itself to high core and then reads the system nucleus into low core (X'0000') from cylinder 0, head 1, sector 0.
- #MLOAD places a one-byte indicator at label $IPLDV, indicating the disk IPL'd (X'00' for R1 and X'01' for F1).
- If no system program file exists on the IPL'd volume, a hard halt occurs.

IPL Interface—MOPPET (Figure 3-1)

- This routine is loaded by #MLOAD as part of the system nucleus.
- The routine reads the volume label sector from the IPL'd disk, calculates the system program file address, and loads the main nucleus initialization program, #MIPPE.
- MOPPET resides immediately following the nucleus at label $ENDNU.

Nucleus Initialization Program—#MIPPE (Figure 3-1)

- This program loads #DPRIN—which consists of the matrix printer I/O control routine, or MP IOCR (DPRINT), and the keyboard IOCR (DEPRES)—at core address X'0700'. The keyboard input line buffer overlays MOPPET (IPL interface).
- System configuration is checked for validity by calling machine configuration (MCNFIG). This subroutine tests all devices specified in the configuration record for presence on the system.
- The core expansion factor is set in the nucleus communications area.
- The cathode-ray tube (CRT) IOCR is loaded into high core by the MCNFIG subroutine if the CRT is present in the configuration.
- The correct keyboard table is loaded by MCNFIG into the three keyboard IOCR's.
- Margin widths for the matrix printer are set to the hardware specifications by MCNFIG.
- The volume labels are read from all mounted disk volumes. IPL is terminated with hard halt 2345 if R1 does not have a standard System/3 volume label. Refer to "Halt 2345" in Section 6. All volumes other than R1 are assumed to require initialization if they do not have a standard volume label.
- The volume-ID table, which is located in the nucleus communications area, is built for all mounted volumes.
- The work area and bad-line buffer are cleared if they are present.
- Output is switched to the CRT if the matrix printer fails while the operator is requesting the configure option.
- All scratch file entries left in the VTOC by co-resident disk system management programs are deleted.

IPL

#MLOAD

READ SYSTEM NUCLEUS AND I/O ROUTINES INTO LOW CORE

1. Hardware reads #MLOAD from cylinder 0, head 0, sector 0, into the first 256 bytes of core and branches to address X'0000'.
2. #MLOAD relocates itself to X'IF00' and branches to X'IF00'.
3. Read system nucleus into core at X'0000'. A disk error causes a soft halt.
4. Branch to MOPPET, the IPL interface resident in the nucleus.

MOPPET

INTERFACE TO MAIN IPL PROGRAM

1. Set console interruption address to $CIENT routine (Figure 3-10).
2. Read volume lable from IPL'd disk using $DISKN (Figure 3-7).
3. Calculate disk address of system program file.
4. Exit to $RLOAD to load #MIPPE at X'0C00' from the system program file.

#MIPPE
Main IPL Program

#MIPPE

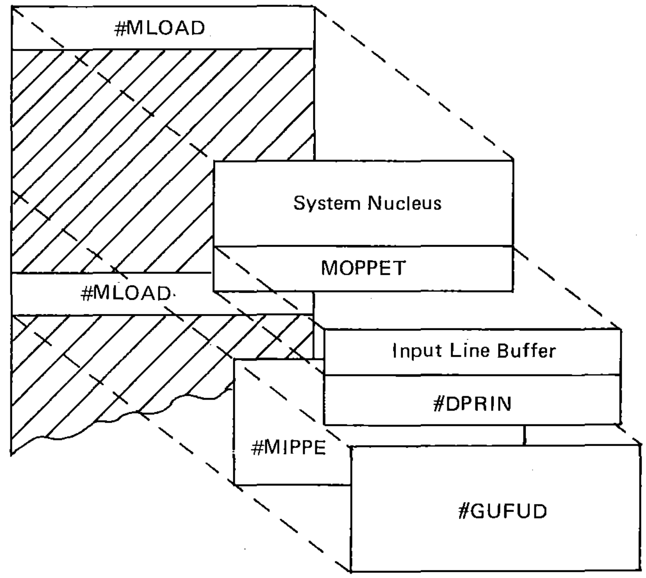
SYSTEM CONFIGURATION

1. Issue carriage return to initialize MP to the left margin.
2. Read #DPRIN (DPRINT and DEPRES) into core at X'0700'.
3. Configure system. Call $DISKN (Figure 3-7) to read configuration record from F1, and call $SPRNT (Figure 3-9) to print messages to operator.
4. Call DEPRES (Figure 3-30) to enable keyboard input.
5. Wait for CONFIGURE command or program start key.
6. Test configuration for validity and load #DSPLY (CRT IOCR) to high core if CRT is present.

READ ALL VOLUME LABELS

1. Call $DISKN (Figure 3-7) to read in all the volume labels of all mounted disks. Terminate IPL with a hard halt if the volume label on R1 does not have a valid label identifier (C'VOL' or X'ABCDEF').
2. Move the volume label and library file addresses to the nucleus communications area.
3. Check that the system work area is present on F1 and R1.

1



1

PROCESS DATE

1. Ask for date using $SPRNT (Figure 3-9).
2. Call DEPRES (Figure 3-30) to enable keyboard input.
3. Modify instruction at X'0000' for a branch to $PAUSD (FE aid).
4. Move user supplied date to nucleus communications area.
5. Set disk addresses for #GUFUD and #ERRPG in the nucleus.

System Work Area OK

No: #ERRPG Figure 3-17 Via $CAERK

Yes: Set Conversational Mode Indicator In Nucleus Communications Area.

#GUFUD
Figure 3-22
Via $CARPL

BR1038A

Figure 3-1. System Initialization (IPL) Flowchart

One of the following modes is entered at this point:

- Conversational mode (#GUFUD, work file update/crusher program) is entered, by #MIPPE, if the system work area is present on both volumes mounted on drive 1 and is set to the current release level. #GUFUD may be entered via the error program if an error message is printed (F2 not initialized, etc.).
- Utility mode (#ERRPG, error program) is entered if the system work area is not present.

System Nucleus

The system nucleus is the core-resident portion of System/3 BASIC. It contains a system communication area, the physical disk IOCR, and various interface routines for other system functions. Figure 3-2, a core map of the system nucleus, shows the components and their functions.

DKDISK ($DISKN) (physical disk IOCR)
NERLOG ($ERLOG) (error logging call section)
NUCLES (system communication area)
NSPRNT ($SPRNT) (interface to system printer IOCR)
NCAERK ($CAERK) (interface to error program)
NQUIRY ($CIENT, $UNMSK) (inquiry request routine)
NABORT ($CAIPL, $CARPL, $CABLD) (abort current operation routine)
NPAUSE ($PAUSD, $RSTR) (save/restore core)
NBLOAD ($BLOAD, $RLOAD, $LOADR) (system loader)
Patch Area

BR1039

Figure 3-2. System Nucleus Core Map

Resident Disk Physical IOCS–DKDISK, $DISKN (Figure 3-7)

- DKDISK is divided into two main sections:

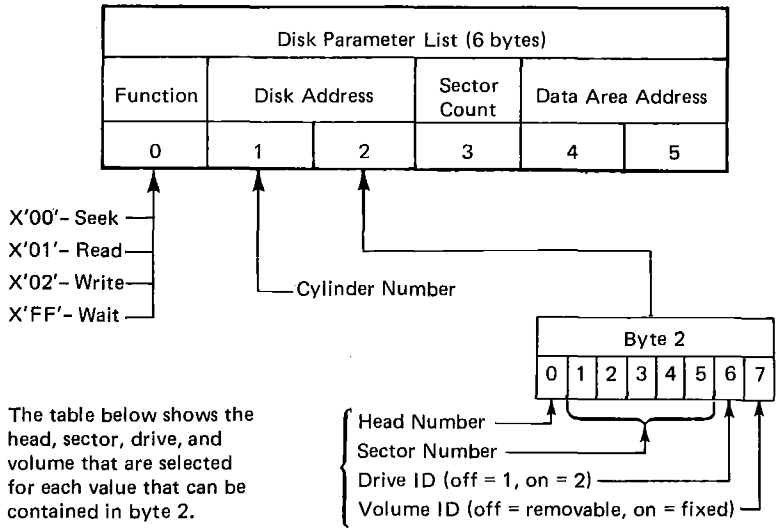
  1. Call–for normal I/O execution.
  2. ERP–error recovery procedure.

- DKDISK is core resident in the system nucleus and performs the physical disk operations of read, write, verify, and seek for both drives.
- A special wait function is provided which allows a calling program to be delayed until the last logical read or write operation for either drive is complete.
- The calling sequence for DKDISK is:

```
B     $DISKN
DC    AL2(DPL)        DPL is the address of the disk parameter list (Figure 3-3).
```

- No checks are made for validity of the DPL parameters. The calling program must ensure that the drive, disk address, etc., are valid.

- Hardware errors are automatically handled by error recovery procedures in the disk I/O control system (IOCS) (Figure 3-4). No error returns are made to the calling program.



The table below shows the head, sector, drive, and volume that are selected for each value that can be contained in byte 2.

Sector	Head 0				Head 1			
	R1	F1	R2	F2	R1	F1	R2	F2
0	00	01	02	03	80	81	82	83
1	04	05	06	07	84	85	86	87
2	08	09	0A	0B	88	89	8A	8B
3	0C	0D	0E	0F	8C	8D	8E	8F
4	10	11	12	13	90	91	92	93
5	14	15	16	17	94	95	96	97
6	18	19	1A	1B	98	99	9A	9B
7	1C	1D	1E	1F	9C	9D	9E	9F
8	20	21	22	23	A0	A1	A2	A3
9	24	25	26	27	A4	A5	A6	A7
10	28	29	2A	2B	A8	A9	AA	AB
11	2C	2D	2E	2F	AC	AD	AE	AF
12	30	31	32	33	B0	B1	B2	B3
13	34	35	36	37	B4	B5	B6	B7
14	38	39	3A	3B	B8	B9	BA	BB
15	3C	3D	3E	3F	BC	BD	BE	BF
16	40	41	42	43	C0	C1	C2	C3
17	44	45	46	47	C4	C5	C6	C7
18	48	49	4A	4B	C8	C9	CA	CB
19	4C	4D	4E	4F	CC	CD	CE	CF
20	50	51	52	53	D0	D1	D2	D3
21	54	55	56	57	D4	D5	D6	D7
22	58	59	5A	5B	D8	D9	DA	DB
23	5C	5D	5E	5F	DC	DD	DE	DF

Notes:

1. Bytes 3-5 are not used for a seek function.
2. Bytes 1-5 are not used for a wait function.

BR1041

Figure 3-3. Disk Parameter List (DPL)

Physical disk addresses are required, but translation of defective track addresses to the assigned alternate track address is performed automatically by the routine. Any initial seek required to access the specified cylinder is automatically performed by the IOCS. If a single logical read or write operation crosses a cylinder boundary, the IOCS automatically performs the seek to the second cylinder and completes the operation when the next call to the IOCS is made. Control remains in the IOCS during the succeeding cylinder operations. A read or write operation automatically crosses track boundaries on one cylinder without subsequent IOCS calls.

Error	Sense Byte	Sense Bit	Recovery Procedure
Unsafe Equipment check Intervention required	2 0 0	0 3 1	Hard halt; no recovery attempted. Retry operation once; then hard halt. Hangs on retry SIO until drive becomes ready.
Overrun No record found Track condition check Missing address marker Data check	1 0 0 0 0	5 5 6 2 4	Perform a read-ID to determine if: • Seeked to correct operative track-retry as read, verify, or write error (below). • Seeked to wrong track—recalibrate and retry operation. • Accessing a track flagged defective—seek to alternate and retry operation. • Accessing sectors beyond the end of an alternate track—seek to next sequential primary track and continue operation. Retry as read, verify, or write error. • Write error—retry write seven more times. • Verify error—rewrite and verify seven more times. • Read error—16 rereads performed with each seek and 16 seeks are tried (256 read errors before hard halt).
Seek check	0	7	Recalibrate and retry seek 15 more times—if seek is successful, retry as read or write error (above).
End of cylinder	1	2	Seek to next sequential cylinder and continue operation.
Unit check, but none of above			Hard halt; no recovery attempted.

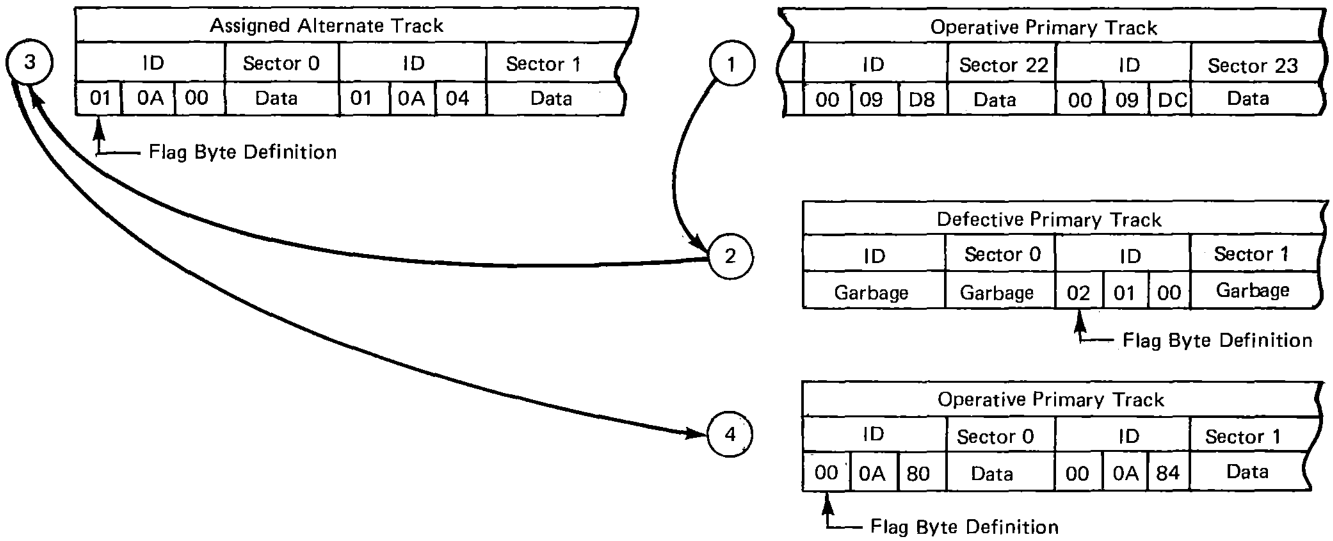
BR1042

Figure 3-4. Error Recovery Procedure (DKDISK)

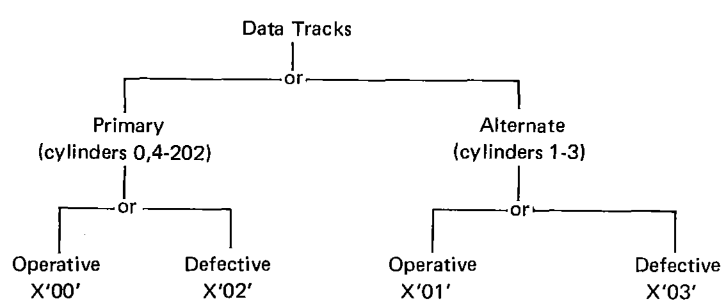
For example, consider a call specifying a read operation of 30 sectors starting at sector 22, track 0 of cylinder 43, when cylinder 10 is currently accessed. The IOCS initiates a seek to cylinder 43, queues the read operation, and returns control to the calling program. The read of sectors 22 and 23 (0-23 on cylinder 43) is automatically performed without returning control to the IOCS. A subsequent IOCS call allows the IOCS to perform the seek and read of the second cylinder.

At the termination of a write function, a verify is automatically performed by DKDISK.

Error Recovery Procedure (ERP) Section: An ERP section is contained within DKDISK. If the operation is unsuccessful after a specified number of retries, the routine comes to a hard halt and an error code is displayed. The system can be restarted only by an IPL. Figure 3-4 shows the error recovery procedures in DKDISK, Figure 3-5 shows an example of a switch to an alternate track in DKDISK, and Figure 3-6 shows the disk control field (DCF) format.



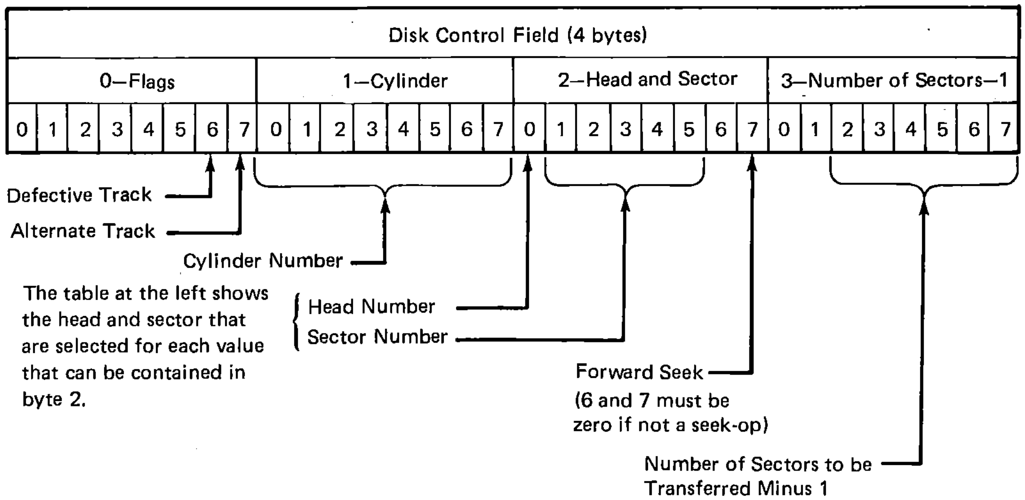
Events	Disk Address C	H	S	Sector Count	Comment
1 Seek to, and read, requested track. Unit check, end of cylinder.	09	1	22	28	Transfer 2 sectors to core. ERP seeks next cylinder.
2 Seek to, and read, next track. Unit check, no record found.	10	0	0	26	DKDISK does not know track is defective. ID of sector 0 is invalid.
Read ID and check flag.	10	0			Alternate assigned by disk initialize.
3 Seek to, and read, alternate track.	01	0	0	24	Transfer 24 sectors to core.
4 Seek to, and read, next primary track. Operation complete.	10	1	0	2	Transfer last 2 sectors to core.



BR1043A

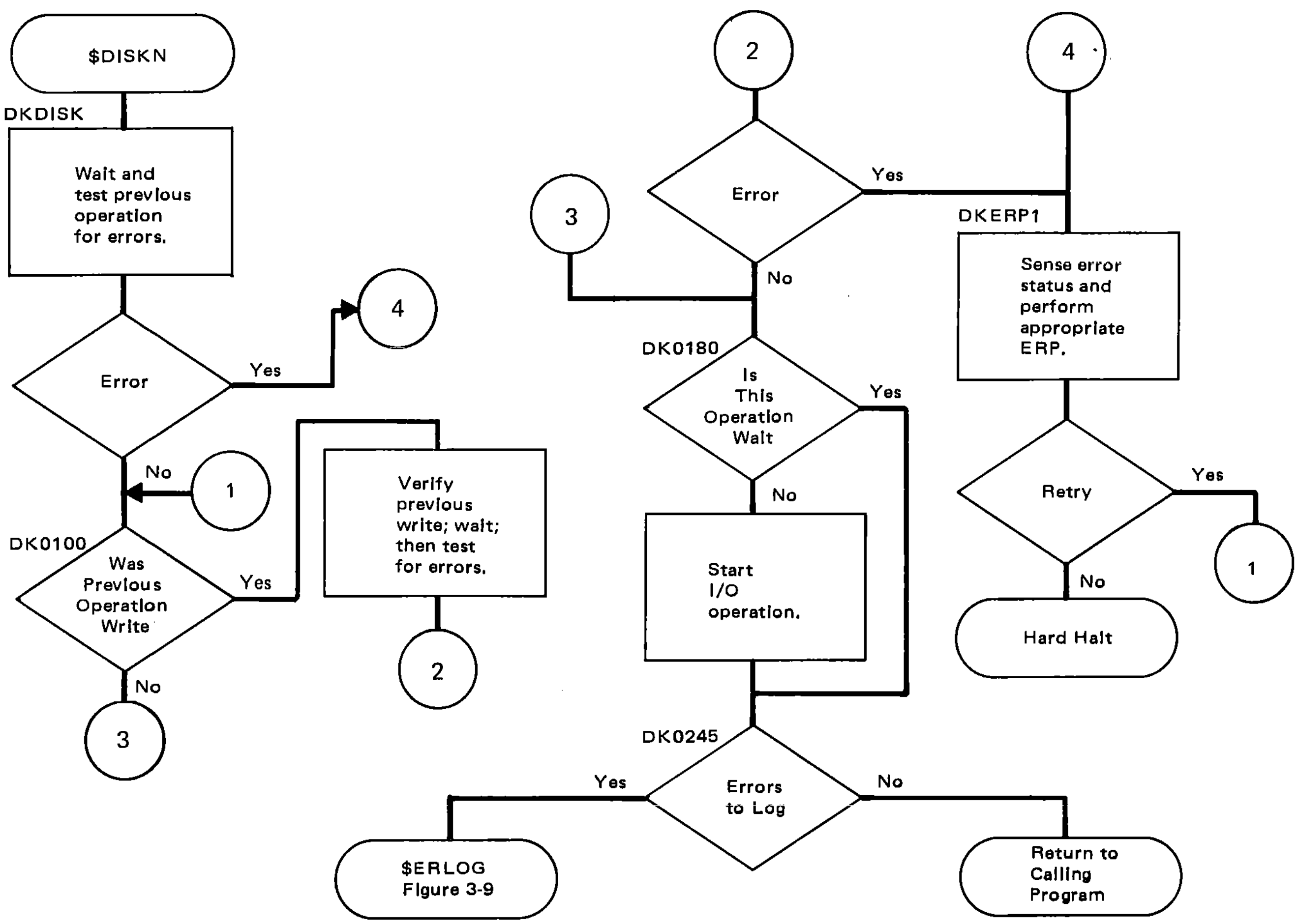
Figure 3-5. Switch to Alternate Track (DKDISK), Example

Sector	Head 0 (byte 3 value)	Head 1 (byte 2 value)
0	00	80
1	04	84
2	08	88
3	0C	8C
4	10	90
5	14	94
6	18	98
7	1C	9C
8	20	A0
9	24	A4
10	28	A8
11	2C	AC
12	30	B0
13	34	B4
14	38	B8
15	3C	BC
16	40	C0
17	44	C4
18	48	C8
19	4C	CC
20	50	D0
21	54	D4
22	58	D8
23	5C	DC



BR1044

Figure 3-6. Disk Control Field (DCF)



BR1045

Figure 3-7. Resident Physical Disk IOCS (DKDISK, $DISKN) Flowchart

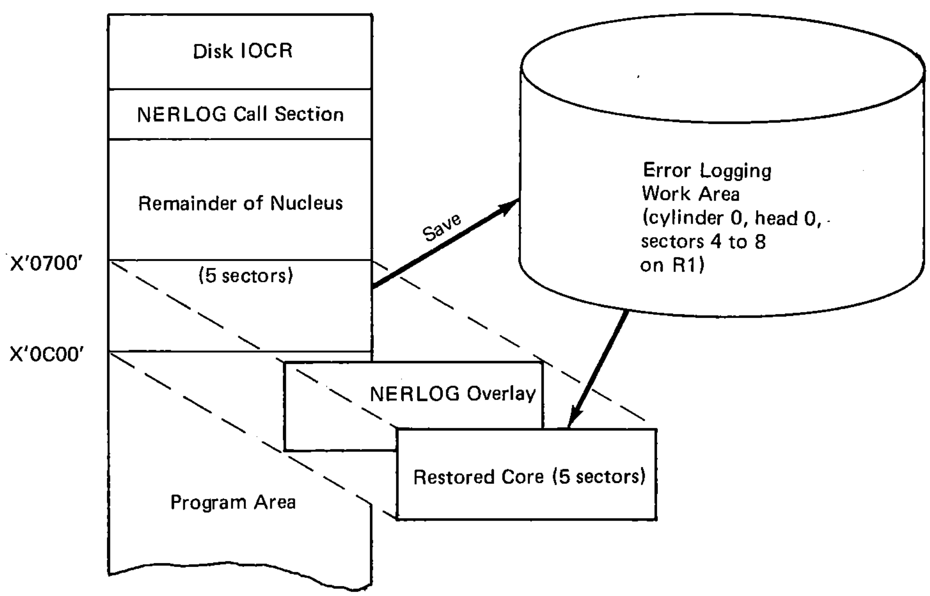
I/O Error Logging Routine—NERLOG, $ERLOG (Figure 3-9)

- NERLOG is used for recording I/O errors in the outboard record (OBR) and updating the statistical data record (SDR). Refer to Figure 3-8.
- The error history log entry ($HISTE) in the nucleus communications region must be set up by the calling IOCR. Refer to Figure 5-1.
- After setting the proper entry at $HISTE, the calling sequence to store the entry to disk is:

B	$ERLOG	Branch if disk error.
B	$DISKN	Branch if other than disk error.
DC	AL2($WAITF)	$WAITF is the address of a disk parameter list containing a wait function code (Figure 3-3).

- If the I/O error occurred while on the interruption level, $ERPND is set and the error is logged upon the next entry to DKDISK.
- NERLOG contains two sections:

    1. Call section—This is core resident within the system nucleus and used to modify DKDISK, save five sectors of core, and load the overlay section into this saved area.
    2. Overlay section—This is brought into core at the saved area to update the OBR and SDR, and generate a hard halt if a system unrecoverable I/O error is indicated.



BR1047

Figure 3-8. NERLOG Core Map

System Communication Area—NUCLES (see Figure 5-1)

- NUCLES provides for communication between system programs.
- It contains indicators, work areas, and core and disk addresses used by the entire system (refer to @FXDEQ in system equates).

Interface to System Printer IOCR—NSPRNT, $SPRNT (Figure 3-9)

- NSPRNT is used to call the device designated as the system printer (CRT or matrix printer).
- NSPRNT decides which device is to be used and branches to the corresponding IOCR.
- The calling sequence to print a line on the system printer is:

```
B     $SPRNT
DC    AL2(PPL)        PPL is the address of the print parameter list (Figure 5-23).
```

The source module consists of one load IAR instruction located in the nucleus. This instruction loads the address of the IOCR assigned as system printer from the $PRDEV field in the system communication area. This address is that of DSPLYN for CRT only, and CRT with matrix printer; or DPRINT for matrix printer. A branch to $SPRNT loads the IAR, effectively causing a branch to the IOCR. The calling sequence passes the address of the print parameter list (PPL). This list is detailed in Figure 5-23.

Error Program Interface—NCAERK, $CAERK (Figure 3-9)

- NCAERK is an interface to the error message program (#ERRPG).
- The error message program is loaded to core and executed.
- No control information is transferred to the routine.

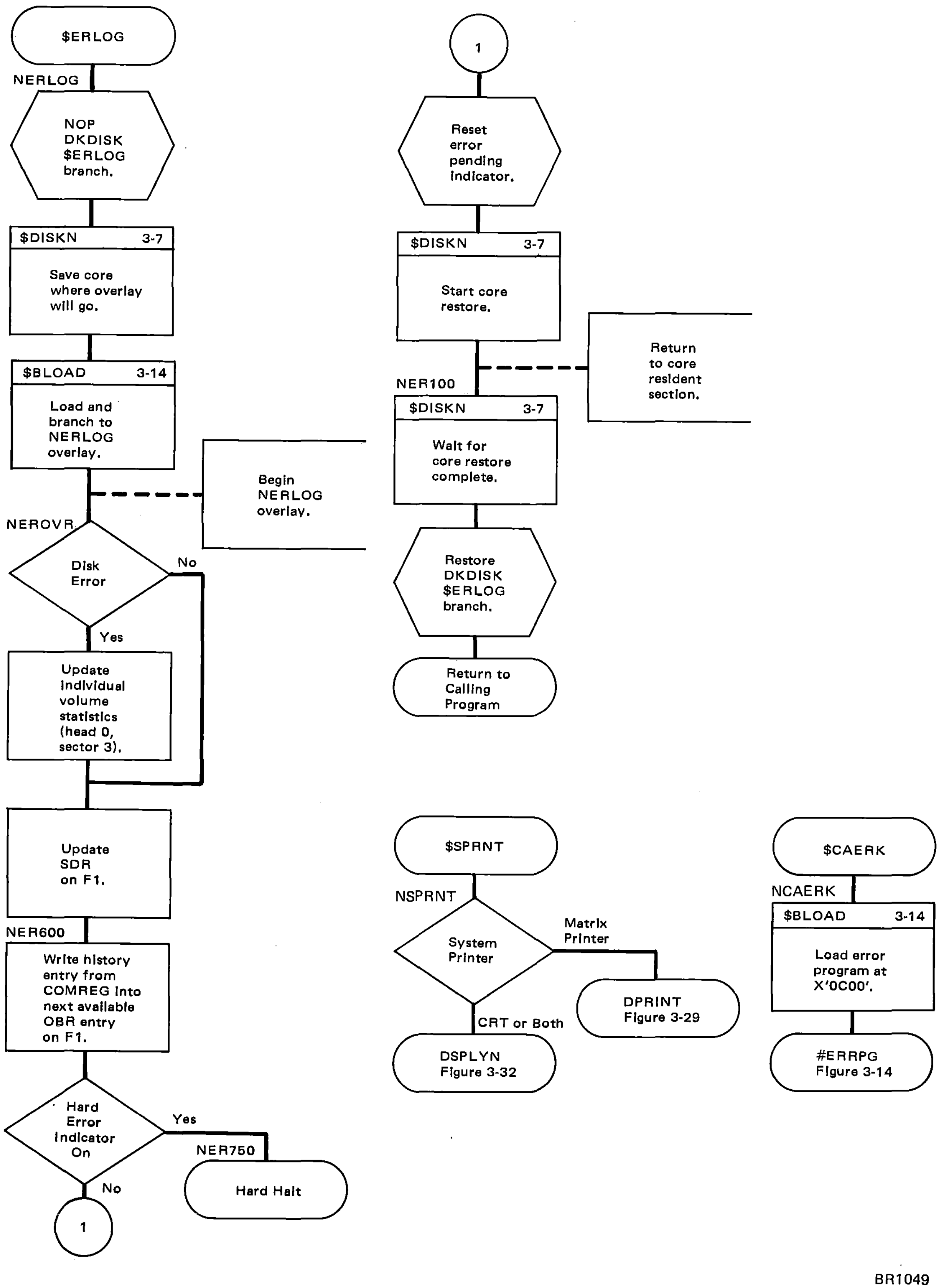


Figure 3-9. Printer and Error Program Interface ($ERLOG, $SPRNT, $CAERK) Flowchart

Inquiry Request Routine—NQUIRY, $CIENT, $UNMSK (Figure 3-10)

- This routine aborts the current operation (if the inquiry request is unmasked) and reloads the work file update/crusher program (#GUFUD).

- An entry point ($UNMSK) is provided for unmasking and aborting if an interruption occurred while masked.

- If the function is aborted, the program interrupted indicator ($INRPT) is set.

- Entry points:

  1. $CIENT—Entry for interruption processing (entered only when on interruption level).
  2. $UNMSK—Entry to unmask inquiry request (IR). To mask IR, it is necessary to move X'80' (equated to @NOP) to location $CIMSK within the IR routine.

- Exits:

  1. IR unmasked—Exit is to $CAIPL.
  2. IR masked—Condition is set for suspended IR and return is made to the interrupted program.
  3. $UNMSK finds no suspended IR—Return is to the calling program.
  4. Suspended IR—Exit is to $CAIPL.

- No error procedures are provided.

Abort Current Operation Routine—NABORT, $CAIPL, $CARPL, $CABLD (Figure 3-10)

- This routine aborts the current operation and/or reloads the work file update/crusher program (#GUFUD).

- If entry occurred during execution (via IR), the program interruption processor program (#EXMSG) is loaded instead of #GUFUD.

- Entry points:

  1. $CAIPL—Entry sets indicators for keyboard entry and no suspended IR and loads #GUFUD or #EXMSG.
  2. $CARPL—Entry sets indicator for no suspended IR and loads #GUFUD or #EXMSG.
  3. $CABLD—Entry loads #GUFUD only; no indicators are modified.

Entry to $CAIPL first resets the input from the keyboard and turns off the no-list indicator. $CAIPL then falls to $CARPL which enables IR. The execution indicator is then tested, and if on (indicating execution in process), #EXMSG is loaded and executed via $PAUSD. If execution is not in process, $CABLD is branched to, which calls $BLOAD to load and execute #GUFUD.

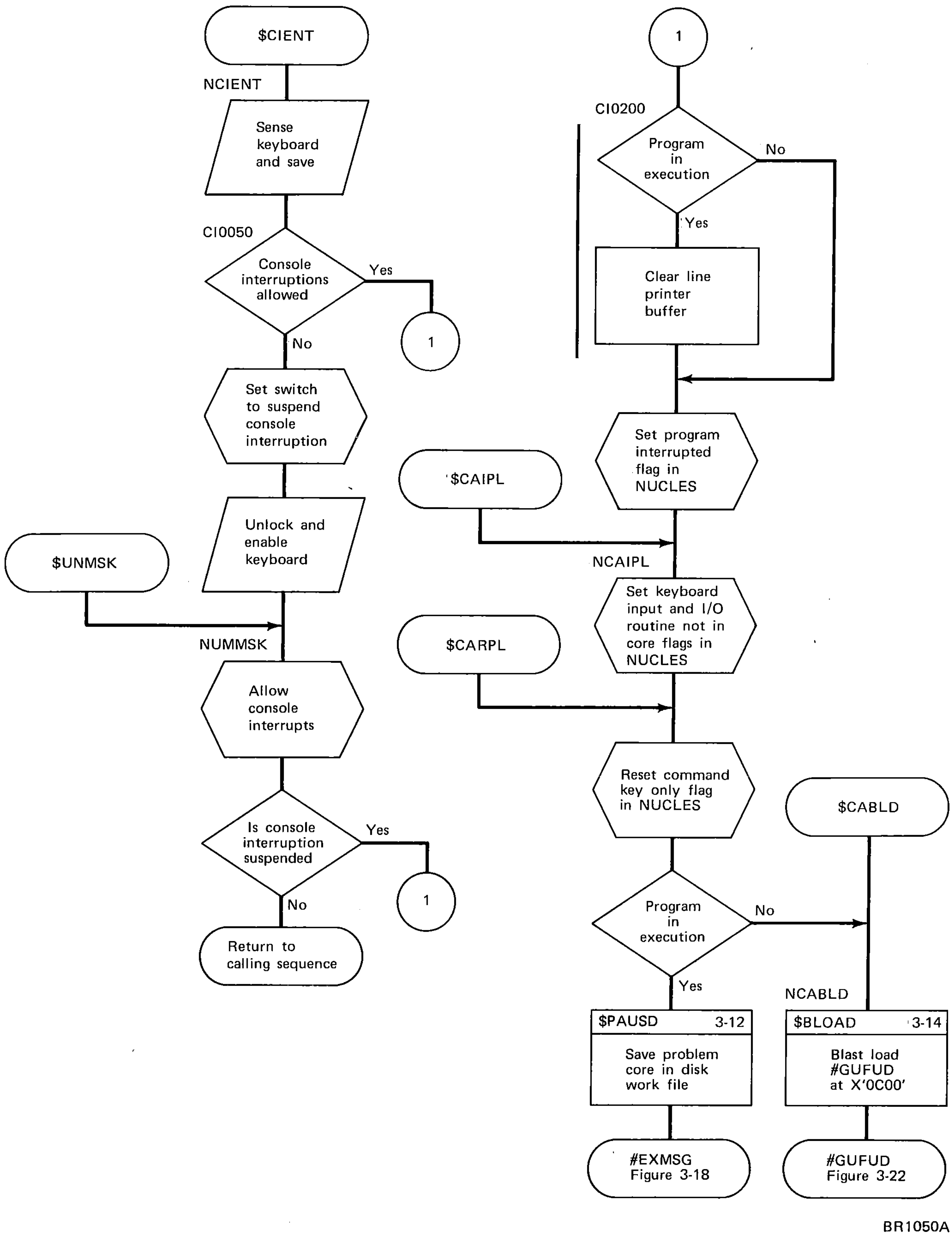


Figure 3-10. Abort (NABORT, $CAIPL, $CARPL, $CABLD) and Inquiry Request (INQUIRY, $CIENT, $UNMSK) Flowchart

Save/Restore Core—NPAUSE, $PAUSD, $RSTR (Figure 3-12)

- NPAUSE saves the contents of core in a disk save area when a program is put in an execution pause condition or the maintenance utility monitor (Figure 3-99) is called. The save area length is 58 sectors, starting at relative disk address X'0600' in the system program file. This area is labeled ##CORE.

- The core area saved is from the end of the nucleus (Figure 3-11) to the end of core or to the start of the CRT IOCS ($DSPLY).

- Upon reentry at the location $RSTR, the saved core is restored from the disk save area, and the program which was paused is ready to be continued.

- Entry points:

  1. $PAUSD—Normal entry to save core.
  2. $RSTR—Entry to restore core.

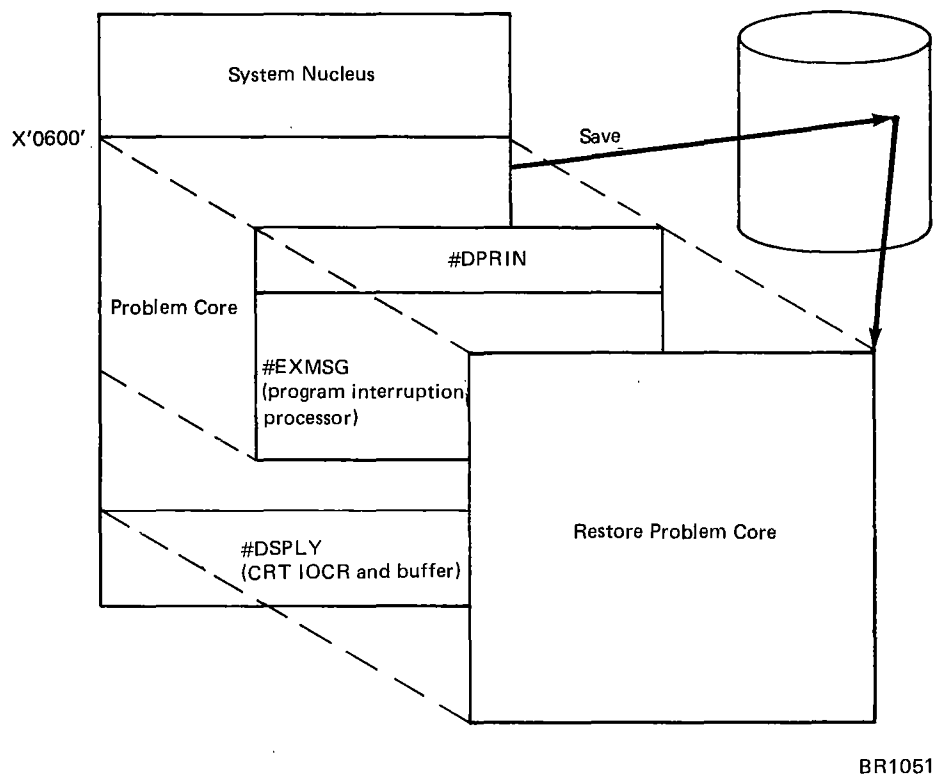


Figure 3-11. Save/Restore Core (NPAUSE) Core Map

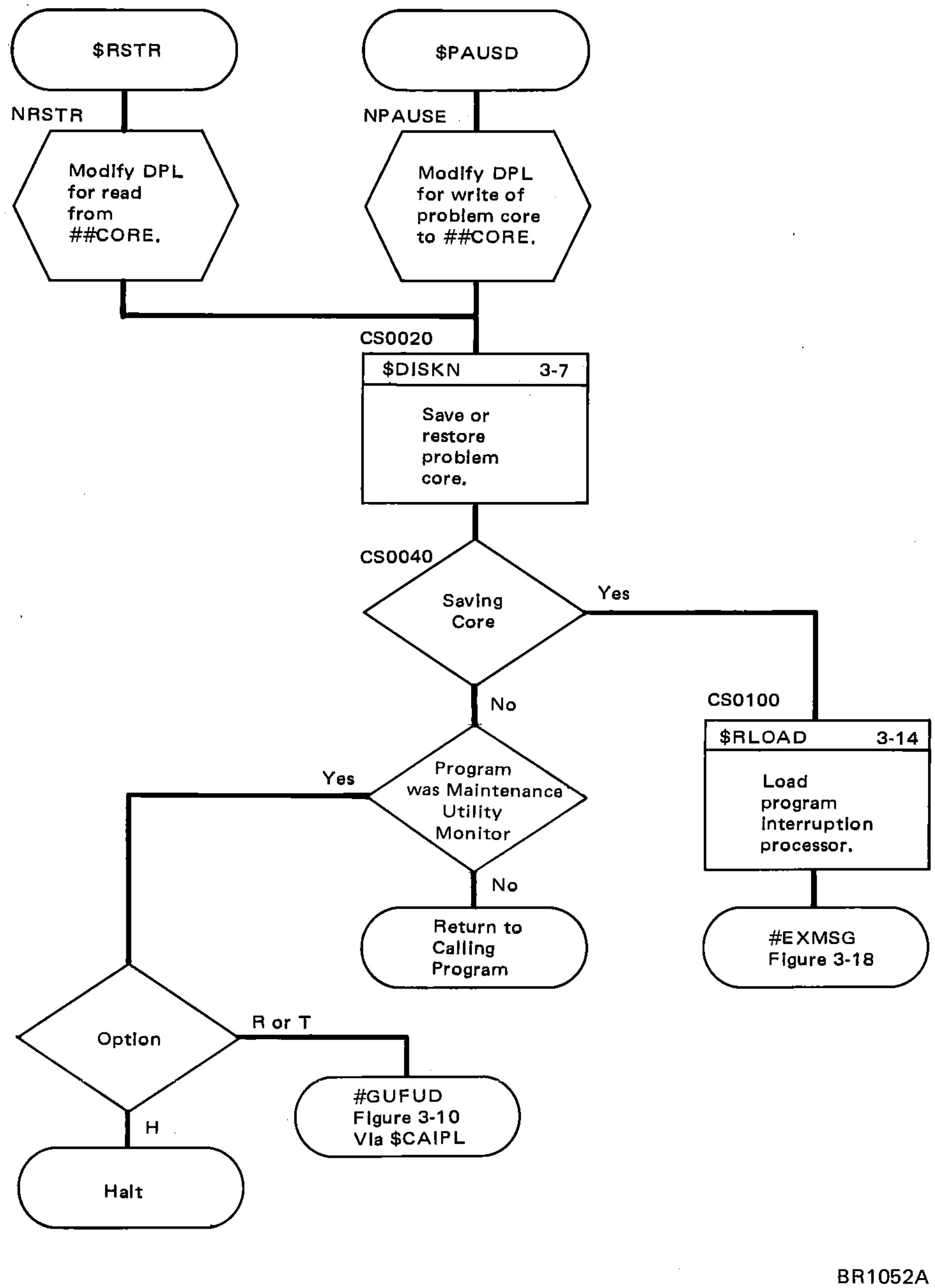


Figure 3-12. Save/Restore Core (NPAUSE, $RSTR, $PAUSD) Flowchart

System Loader—NBLOAD, $BLOAD, $RLOAD, $LOADR (Figure 3-14)

- NBLOAD is used for loading and executing a requested program (Figure 3-13).
- Three types of calling sequences are available to the calling program:

Calling sequence to load and execute a fixed-disk-address program:

```
B     $BLOAD
DC    AL2(DPL)      DPL is the address of the disk parameter list used to load the
                    program (Figure 3-3).
```

Calling sequence to load and execute a relocatable-disk-address program:

```
B     $RLOAD
DC    AL2(DPL)      DPL is the address of the disk parameter list used to load the
                    program (Figure 3-3).
```

Calling sequence to load a relocatable-disk-address program and return to the calling program:

```
B     $LOADR
DC    AL2(DPL)      DPL is the address of the disk parameter list used to load the
                    program (Figure 3-3).
```

- The disk address specification in the DPL, when using $RLOAD, is the base disk address for the program. It is added to the starting address of the system program file to find programs that are within the file at fixed displacements.
- No check is made to verify that the DPL address is correct.

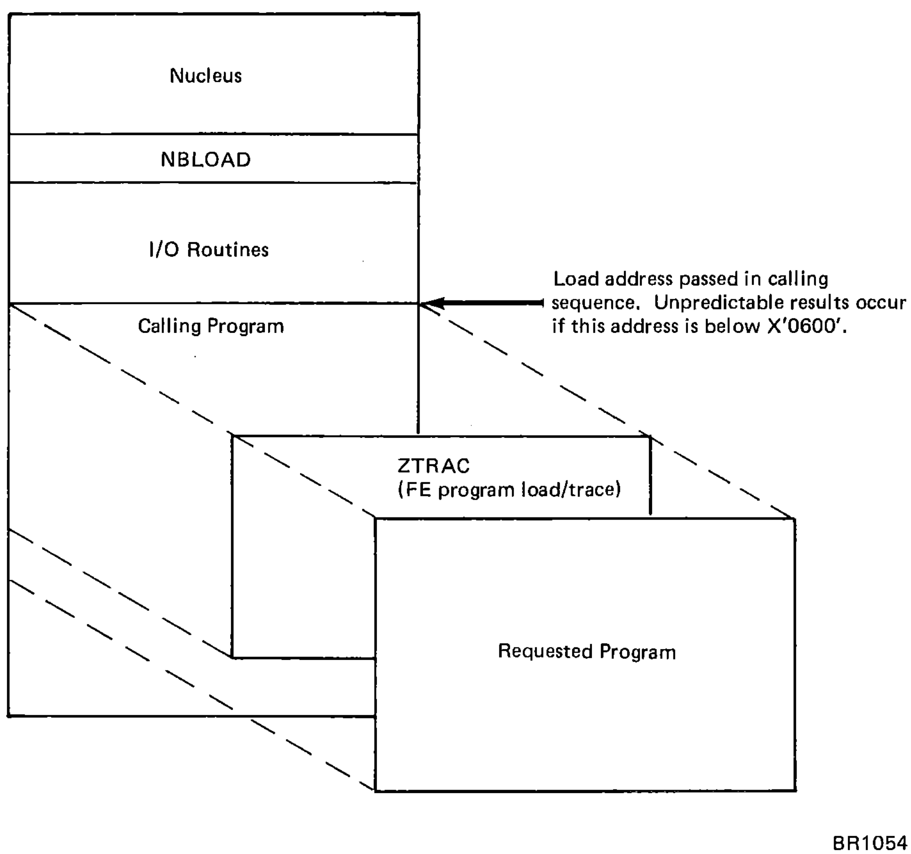


Figure 3-13. System Loader (NBLOAD) Core Map

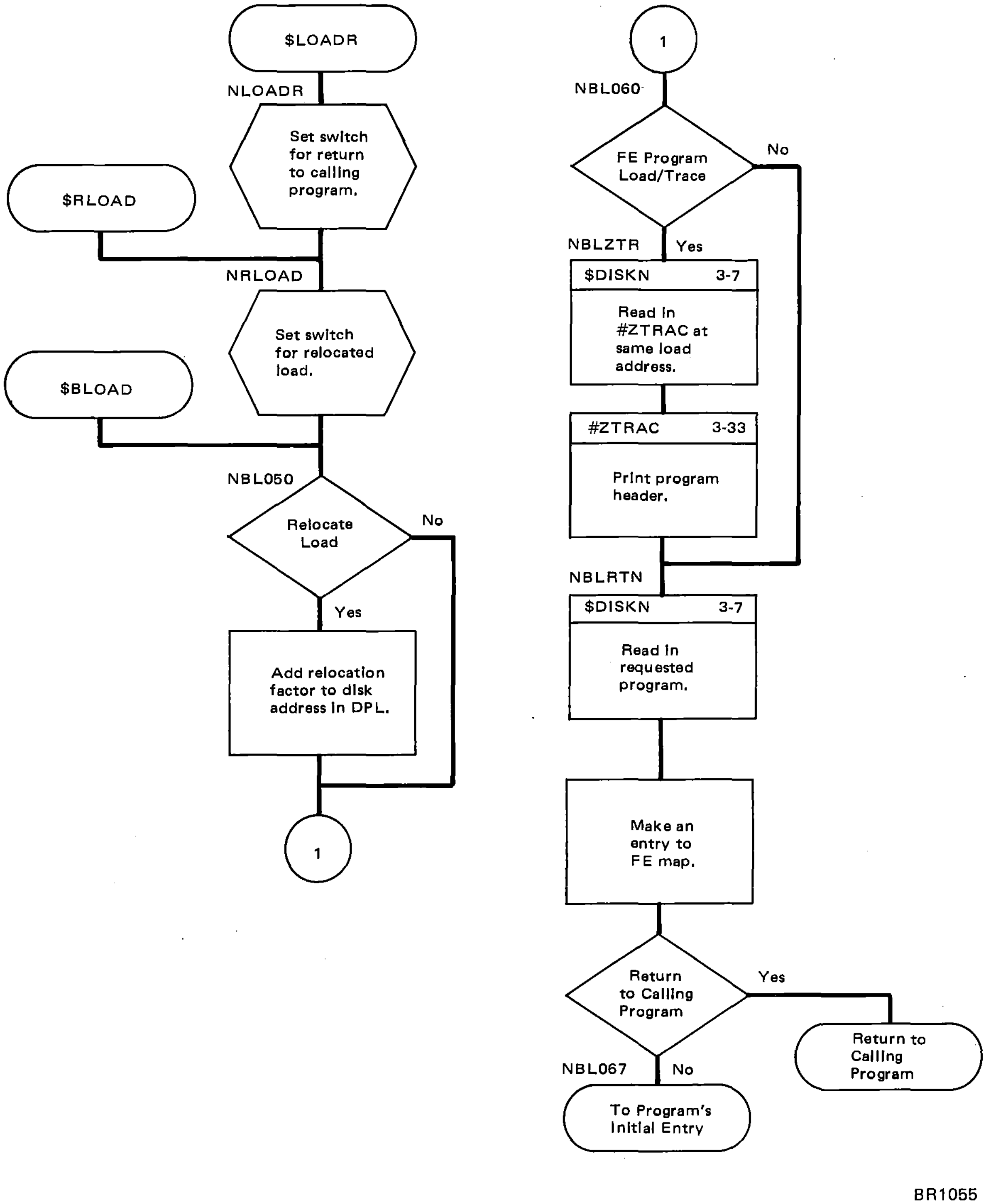


Figure 3-14. System Loader (NBLOAD, $BLOAD, $RLOAD, $LOADR) Flowchart

Error Message Program—#ERRPG (Figure 3-17)

- #ERRPG prints all terminal error messages (except those from copy disk) that occur during BASIC or utility modes of operation. For messages occurring in DCALC mode of operation, refer to "DCALC Error Messages—VERROR."
- The assembly of #ERRPG contains these major source modules:

  1. ERRPGM—Mainline logic, Figure 3-17.
  2. DL2ICS—Disk logical IOCS, Figure 3-70.

The error code for all messages except stacked (multiple) is obtained from the system communications area in the nucleus at label $CAERR (Figure 5-1). Stacked error codes (Figure 3-15) are located at label $$ERSK. The error codes, when present at these locations, are the message numbers within ##ERMS.

The message texts and table of relative displacements are located in the system program file. The assembly containing the messages has the name ##ERMS. Error codes passed to #ERRPG index these tables. The message text is read from disk with a two-sector read. A message can overlap one sector boundary. After the two sectors are read, the message is located in the buffer using the second byte of the table entry (Figure 3-16). The fourth byte of each message is the length of the message.

#ERRPG prints an up-arrow under the first improper character of input when a syntax error occurs. On entry, the index register points to this position in the input line buffer.

The bad line is stored in the bad-line buffer on cylinder 4 of the system work area. Refer to "System Work Area Equates—@WKAEQ" in the program listings for the disk address of the bad-line buffer.

3-Byte Error Entry		
1	2	3
Error code	Line number	

Note: Byte 2 is set to X'A0' when no line number exists.

BR1056

Figure 3-15. Stacked Error Entry at $$ERSK

2-Byte Error Message Entry	
1	2
Relative sector displacement	Relative displacement within sector

BR1057

Figure 3-16. Message Table Entry (#ERRPG)

When a syntax error occurs:

1. An up-arrow is printed under the first invalid character of input.
2. On entry, the index register points to this character in the input line buffer.
3. A full error message is printed if the enter-plus key is pressed.

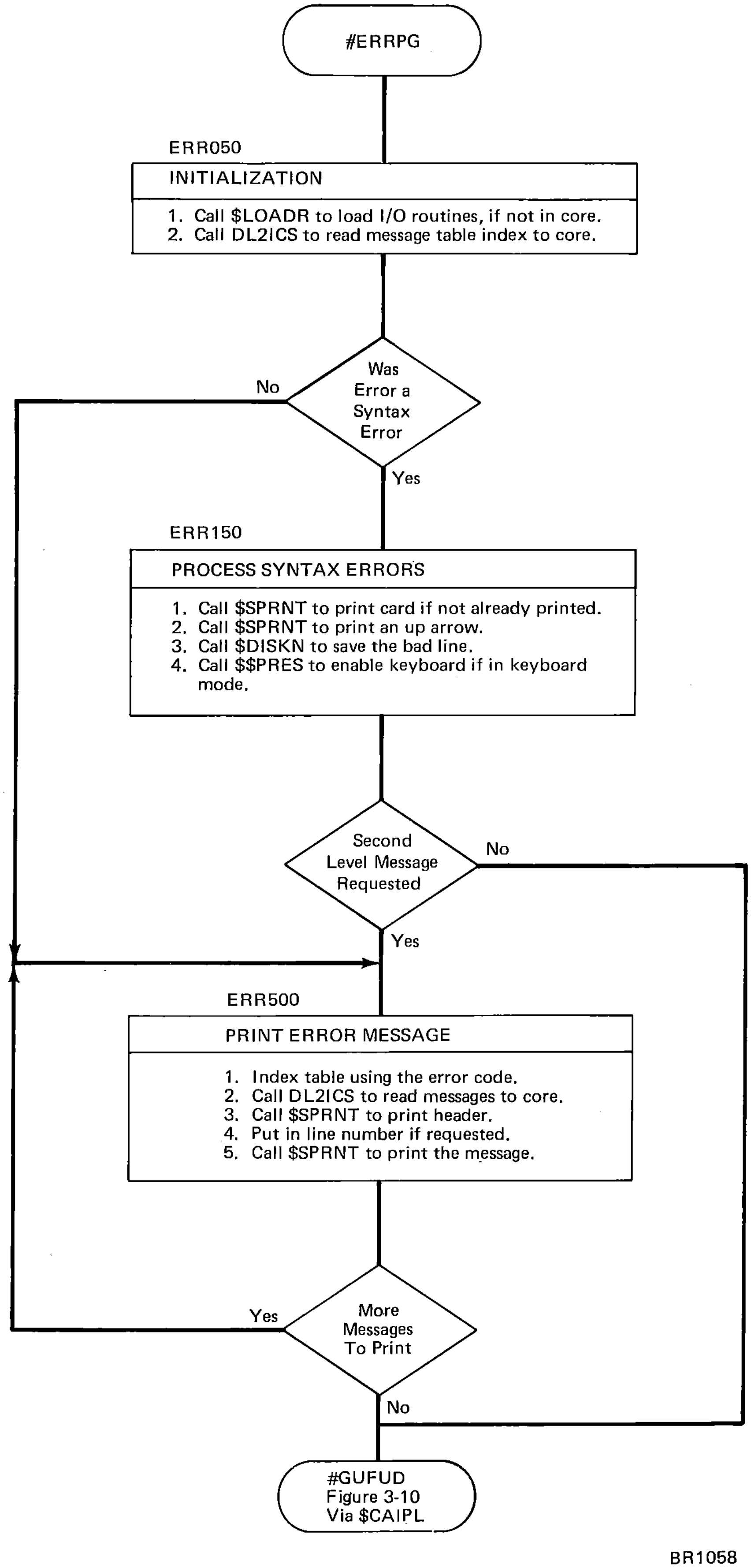


Figure 3-17. Error Message Program (#ERRPG) Flowchart

Program Interruption Processor—#EXMSG (Figure 3-18)

- #EXMSG prints a message on a program interruption, identifying the type of interruption, and the line number where the program was interrupted.
- The assembly of #EXMSG contains the major source module (EXMSGS).

#EXMSG is loaded (Figure 3-11) via the $PAUSD routine in the nucleus when one of the following conditions is present:

1. Console interruption—Printed line number refers to the statement last executed.
2. Pause statement—Printed line number refers to the pause statement being executed.
3. Step mode—Printed line number refers to the statement last executed.
4. System stop, system reset, system start—Invokes maintenance utility aids.

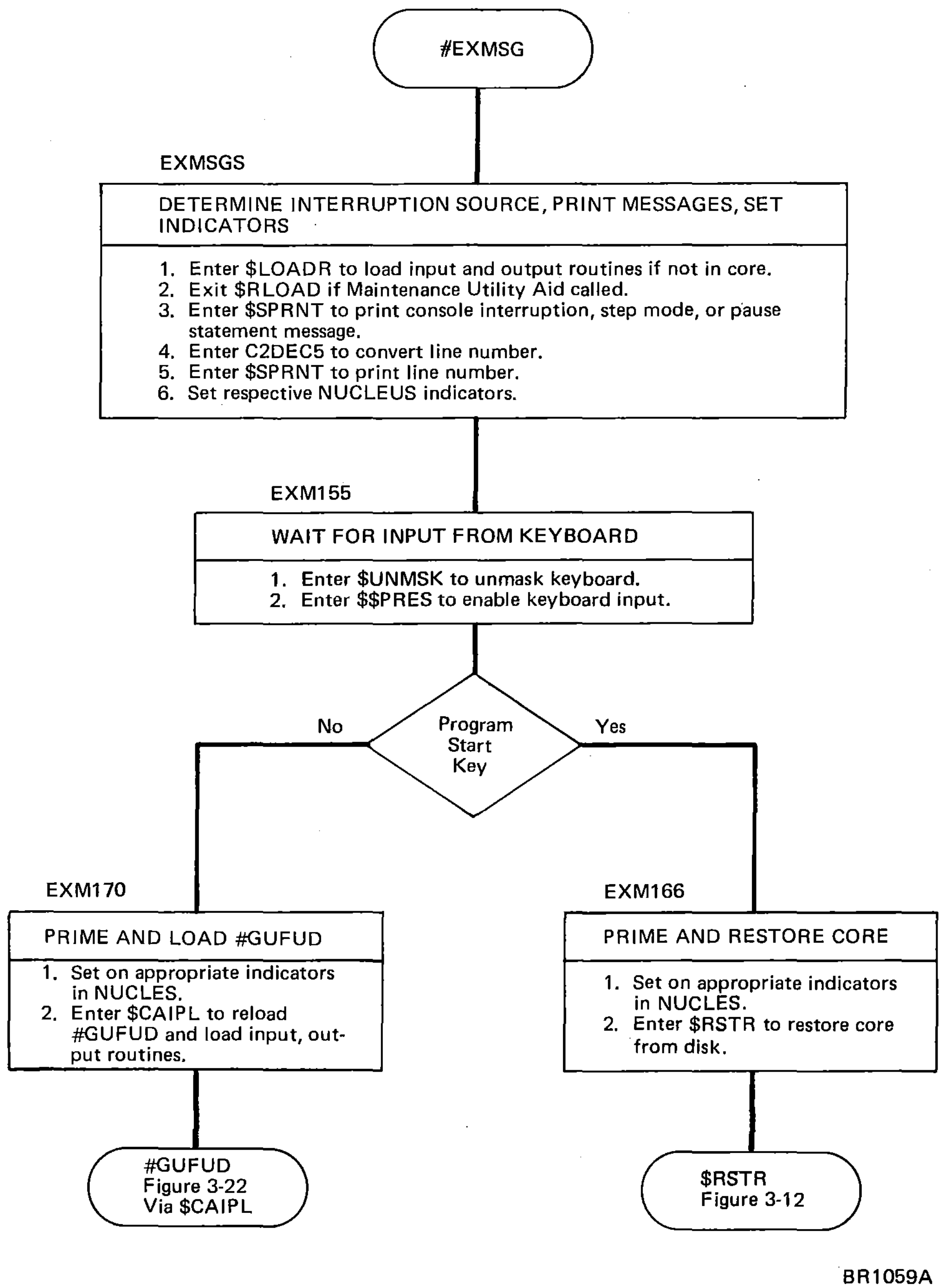


Figure 3-18. Program Interruption Processor (#EXMSG) Flowchart

Work File Update/Crusher—#GUFUD (Figure 3-22)

- #GUFUD updates the work file in the system work area and maintains the file in line-number order.

While #GUFUD is waiting for the operator to complete the next line of input, the crush and reorder portion packs the file by reorganizing the disk blocks that contain segments of the file. #GUFUD attempts to keep these disk blocks in physical order, utilizing as much space in each active block as possible by condensing the segments of the file. Either the keyboard IOCS (DEPRES), the card reader IOCS (DREADN), or the predefined procedure line fetch routine (#GRAPR) accepts an input statement or command from the operator and concurrently builds a line in the primary input line buffer.

The assembly of #GUFUD contains these major source modules:

1. GUFCSH—Work file crush and reorder, Figure 3-22.
2. GURDIN—Common disk read subroutine, no flowchart.
3. DL4ICS—Work file IOCS, Figure 3-70.
4. GUFPAK—Pack core buffers subroutine, Figure 3-22.
5. GUFENT—Initialization, Figure 3-22.
6. GCPACK—Pack BASIC program statement subroutine, no flowchart.
7. GUFUPD—Work file update, Figure 3-22.

Figure 3-19 illustrates the usage of core, initially containing initialization and file update routines, as disk I/O buffers. These buffers are referred to as CB1, CB2, CB3, and CB4. The fifth buffer is used by the subroutine that packs the contents of the first four buffers.

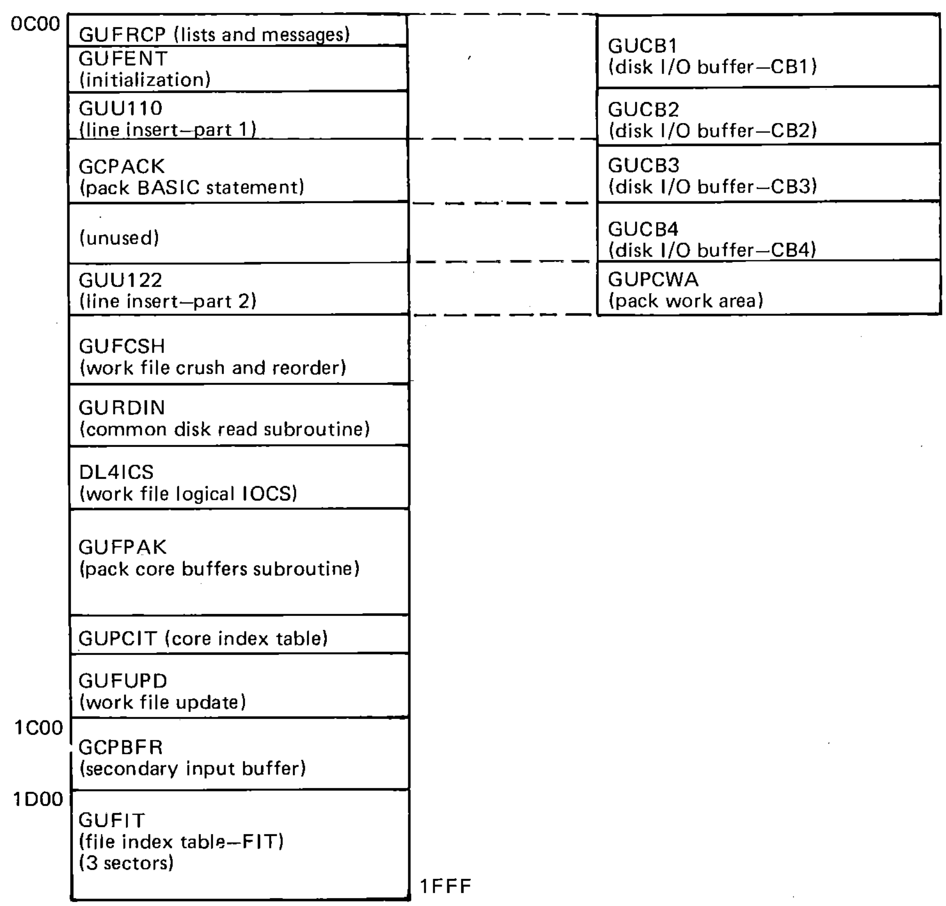


Figure 3-19. Work File Update/Crusher Core Map

The core index table (CIT) contains four 4-byte entries, one associated with each of the four core buffers. The content of each entry is:

1. Byte 1—Relative sector displacement into the work file of the disk block in this buffer.
2. Bytes 2 and 3—Highest line number in this buffer.
3. Byte 4—Unused or free bytes in this buffer.

Initialization—GUFENT

The initial entry to #GUFUD contains a branch to the initialization routine (GUFENT). The functions performed by this routine are detailed in Figure 3-22. The area occupied by GUFENT is used as disk I/O buffers by other sections of #GUFUD, after initialization is complete (Figure 3-19).

Three indicators in the system communications area (NUCLES) determine the operation to be performed on the input:

1. $FUIND—A new or replacement line is to be placed in the work file. #GUFUD expects a seven-byte statement header (Figure 3-21) in the primary input buffer (X'0600'). Following the header, in the eighth byte of the secondary input buffer (X'1C00'+7), #GUFUD expects either a syntax-checked BASIC program statement, a syntax-checked data file line, or a procedure file line.
2. $FDIND—A parameter list of line numbers is to be deleted. #GUFUD expects the presence of a delete parameter list (Figure 5-26) in the secondary input buffer (X'1C00').
3. $FCIND—A single line number is to be deleted. #GUFUD expects the two-byte binary line number, of the statement to be deleted, to be present in the fifth and sixth bytes of the primary input buffer (X'0600'+4).

If all three of the preceding indicators are off, only crushing and reorder operations are performed. Work file update operation is bypassed.

Pack BASIC Program Statements—GCPACK

If a BASIC program statement is being inserted (addition or replacement) in the work file, the GCPACK subroutine is executed to pack the statement. This packing operation is performed on the statement after it has been moved to the secondary input buffer.

Repetitions of characters in the statement are packed before the statement is written to the work file. When a character is repeated more than twice, all but the first character is replaced by a one-byte count of the additional repetitions of the character. This count byte can be recognized by the fact that it cannot equal or exceed X'1C' (end-of-statement code), the lowest valid functional character. The range of the repetition count byte is X'02' thru X'1B'. If the repetition count exceeds X'1B', more than one repetition sequence is generated (Figure 3-20). After the line is packed, the byte count of the packed line is stored in the statement header.

The core area occupied by GCPACK is also used as disk I/O buffers after initialization is complete (Figure 3-19).

Work File Update—GUFUPD

This routine adds, replaces, or deletes a single statement (line), or deletes lines specified by a list of line deletion parameters. The file index table (FIT) in high core is searched, by line number, to locate the first affected disk block. This disk block is read into CB1 and the next two logically sequential disk blocks are read into CB3 and CB4. The disk block in CB1 is searched for an equal or high line number. (Equal effects a line deletion or replacement; high effects a line addition.)

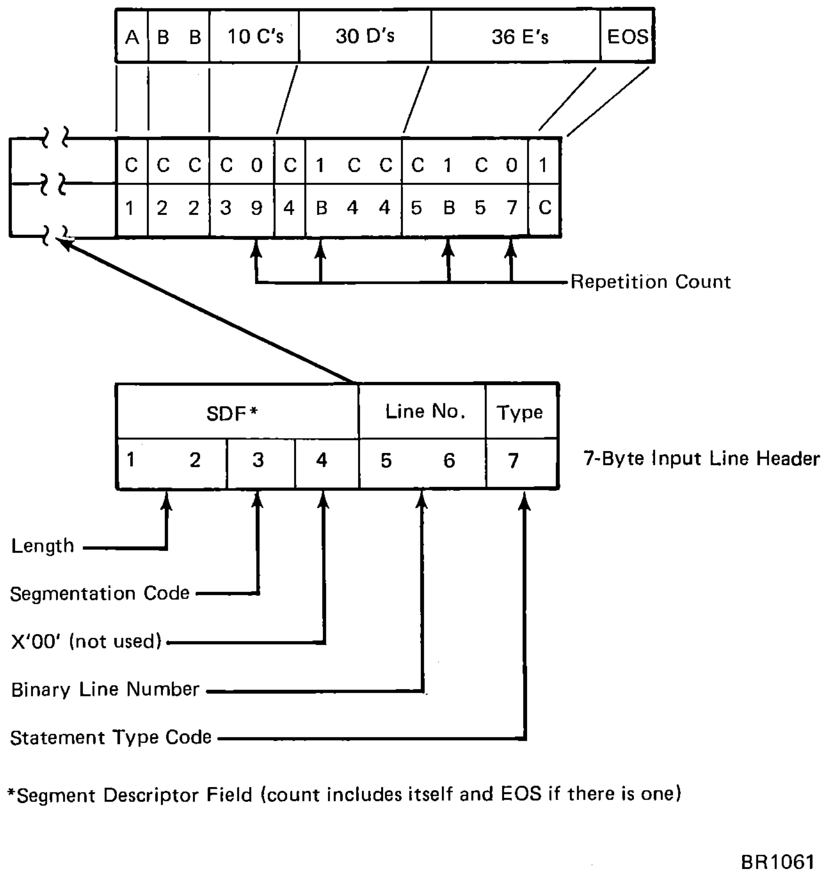


Figure 3-20. System Work File (Packed Data)

The four functional operations performed by the work file updater are:

- Single Line Deletion. Adjust the CIT to reflect the additional free space in CB1. Pass the core address of the first (primary) segment in the line to the pack core buffers subroutine. The packer physically deletes the line (all segments).
- Line Addition. Line segments are shifted from CB1 into CB2 to provide space for the new line. Each block is maintained in ascending line number order. The new line may require division into two segments—one in CB1 and one in CB2. In this case, the primary segment is moved to CB1, segments in CB2 are shifted to the right, and the secondary segment is moved to CB2. After the new segments are moved to the buffers and the CIT is adjusted to reflect their status, the buffers are packed by the pack core buffers subroutine.

  Note: Part 1 of the line insertion routine (GUU110) may be overlayed by data being moved to CB2 by part 2 of the routine (Figure 3-19).
- Line Replacement. Processed as a single line deletion followed by a line addition.
- Deletion of a Range of Lines. Deletion of a range of lines differs from single line deletion in these respects: The delete range indicator is set for the pack core buffers subroutine. Consecutive passes are made through work file update (GUFUPD) for each range of line numbers in the parameter list. When the parameter list is completely processed, #GUFUD is reloaded to make a second pass through initialization (GUFENT) and print the ready message.

Work File Crush and Reorder—GUFCSH

This routine is executed following the completion of a task by work file update and while the system is waiting for the completion of a statement or command input operation.

The FIT in high core is searched, always from the beginning, to locate the first active disk block containing more than eight bytes of free space. If such a block is located, that block and the next two logically sequential disk blocks are read into CB1, CB2, and CB3. After the CIT is updated, the buffers are packed by the pack core buffers subroutine. The preceding operation is referred to as a single crushing operation.

Successive crushing operations cause free space to be moved to the logical end of the work file. Any scan of the FIT that does not locate a disk block containing excessive free space indicates the work file is completely crushed (packed).

Note: Free space in the last logical block of the file is not considered.

When the work file is completely crushed, the reorder section (GUFRDR) is executed to resequence the active disk blocks into physical order. Physical order means that disk blocks are in ascending line number order at consecutive, ascending relative disk addresses within the work file. Logical order means that the disk blocks are chained together so that they can be accessed in ascending line number order. Chaining is provided by the FIT and by the linkage code in the first byte of each disk block.

To reorder the file, the FIT in high core is searched, always from the beginning, until two consecutive entries are found pointing to disk blocks out of ascending order. Four disk blocks are read into CB1 thru CB4. These disk blocks are those referenced by four consecutive FIT entries, where the two in the center (CB2 and CB3) are the two found out of sequence on the search.

The physical location references in the FIT, for the last three disk blocks read in, are sorted into ascending order. The physical location reference of the first disk block cannot be changed because it is referenced by a linkage code in a disk block which is not available at this point. All four disk blocks are written to the physical disk locations specified by the sorted FIT entries. As each is written, it is linked to the disk location of the block that follows it. The preceding operation is referred to as a single reorder operation.

Successive reorder operations cause the work file to be closer to being in physical order. Any scan of the FIT that does not locate an out-of-sequence condition in the file indicates the work file is completely reordered.

Checks for input line complete (GUFSCL) are made before each crushing or reorder operation. If the statement or command input is complete, the command analyzer (#ECMAN) is loaded via the system nucleus. If a blank line or card is encountered, input is reenabled, and crushing/reordering continues. Successive crushing or reorder operations continue until input is complete or the work file is completely reordered.

Figure 3-21 is a simplified flowchart of the crush and reorder operations.

Pack Core Buffers Subroutine—GUFPAK

The packing subroutine is used by both work file update and work file crush and reorder to pack the disk blocks in CB1 through CB4 and write them to the work file. Packing the core buffers means moving the free space from the first three buffers to the end of the fourth buffer by shifting line segments toward the first buffer. The disk blocks in the buffers are always in ascending line number order. The packing subroutine also updates entries in the file index table (FIT) to reflect changes made to the disk blocks.

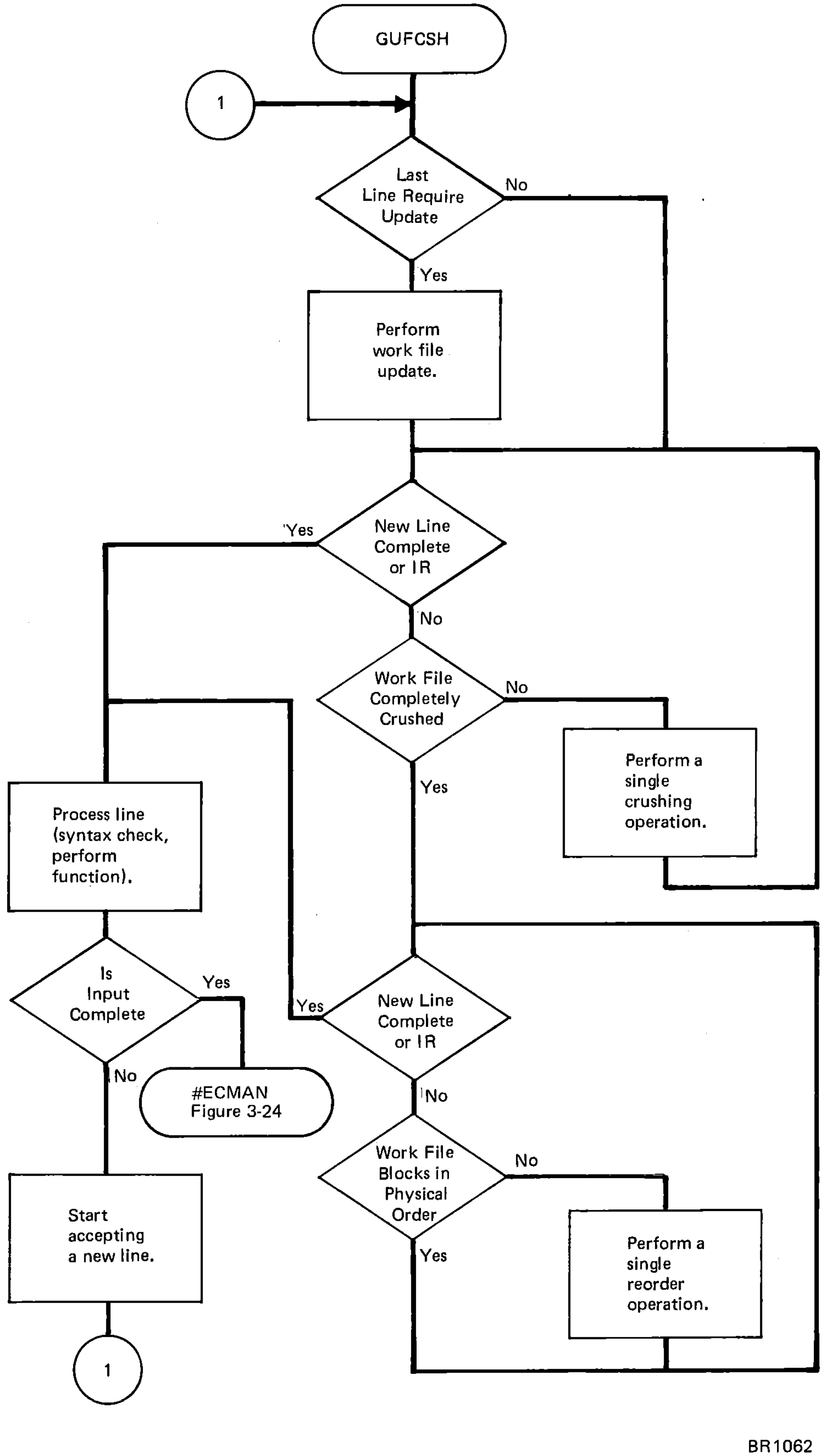


Figure 3-21. Work File Update, Crush, and Reorder Operations

Two address pointers are used to perform the packing operation. One address references the start of the first free space in the buffers. The other references the start of the next line segment following the free space. The second address is incremented past any secondary segments of a deleted line, or past deleted lines, when deleting a range. It may become necessary to read in more disk blocks, in logical line number order, to effect a deletion. A work area (GUFCWA) is used during the packing operation as an intermediate holding area as the buffers are being condensed (packed).

Following a single pass through the packing subroutine, CB1, CB2, CB3, and CB4 may contain line segments (always in ascending line number order). Only the CB's containing active segments are written back to the work file.

FIT entries may become null (due to line deletions), be activated (line addition; one entry only), or be modified (changes in line number and/or free space).

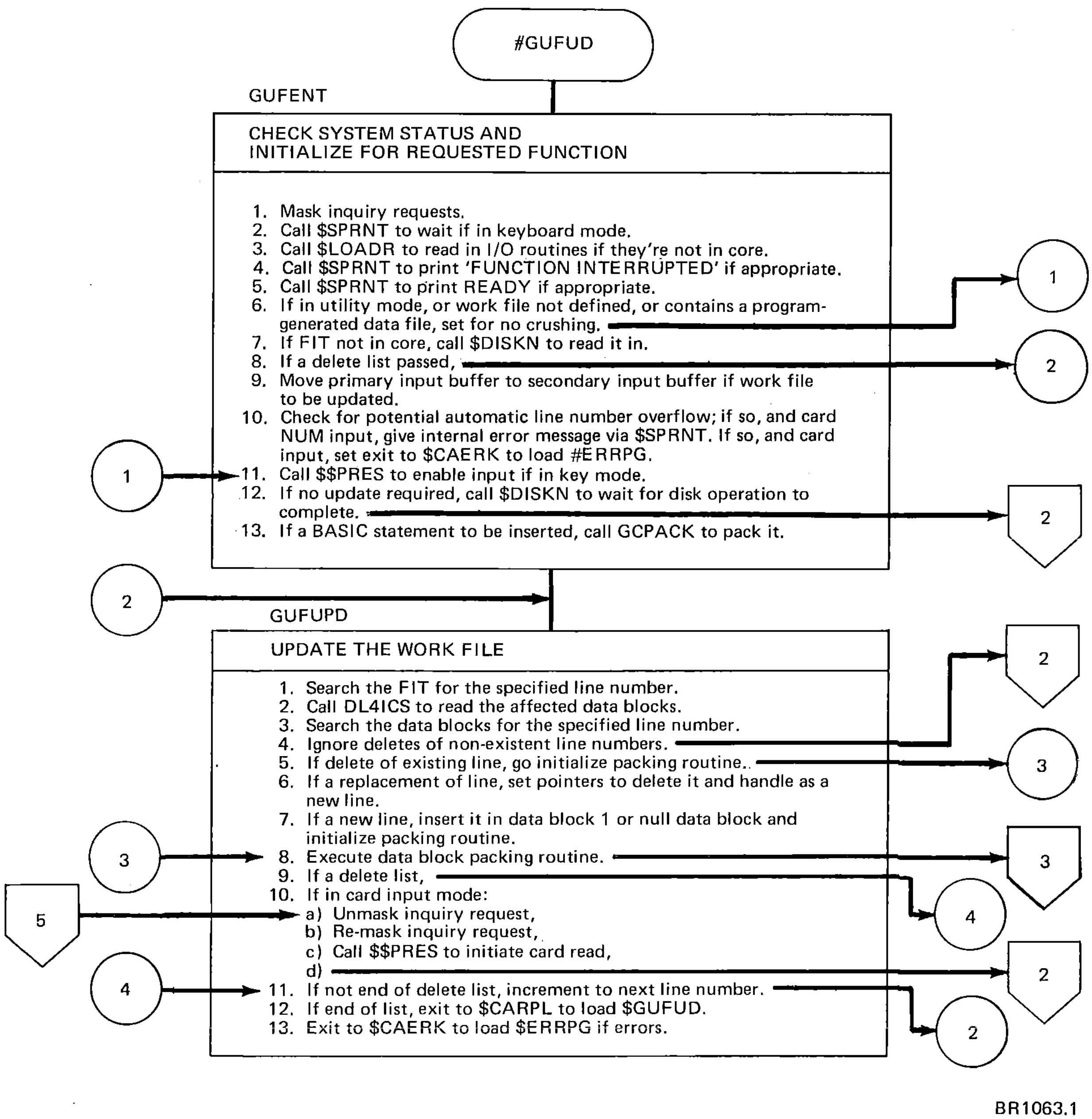
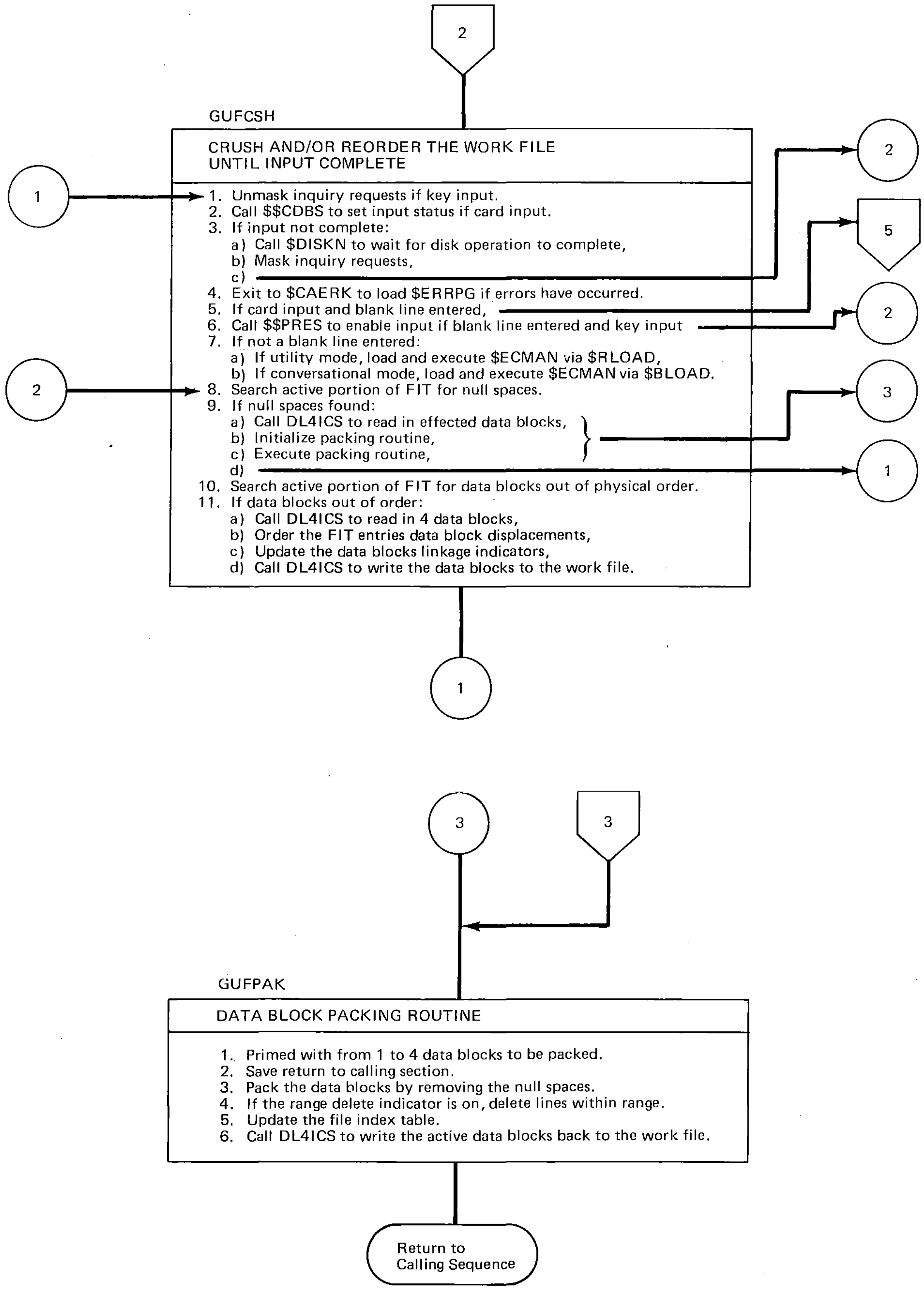


Figure 3-22. Work File Update/Crusher (#GUFUD) Flowchart (Part 1 of 2)



3R1063.2

Figure 3-22. Work File Update/Crusher (#GUFUD) Flowchart (Part 2 of 2)

Command Analyzer—#ECMAN (Figure 3-24)

- #ECMAN analyzes BASIC system input and loads the program required to process the requested function.
- The assembly of #ECMAN contains the major source module, ECMANL.

All input from the keyboard or card reader is analyzed by #ECMAN, except for blank lines, or input at execution time, to user written BASIC programs. #ECMAN is loaded by the file update program (#GUFUD) when a completed input record (EOS detected) exists in the input line buffer. The following actions are taken:

1. Input starting with a keyword causes the corresponding keyword program to be loaded and executed. #ECMAN scans a table, containing one entry for each keyword, for a match with the input line. The DPL used to load the keyword program is built from fields in this entry (Figure 3-23).
2. Input starting with a line number causes the appropriate syntax checker program (#SFSYN, #SPSYN, or #SDSYN) to be loaded and executed.
3. If the first character of input is a command key, #EFKEY (Figure 3-61) is loaded.
4. Invalid lines, and other error conditions, cause the error program (#ERRPG) to be loaded and executed.
5. If DCALC-requested code is on the first text byte of primary input buffer, DCALC is invoked.

Length	Field Name	Field Description
1	Keyword length	Count of letters in the keyword = n. n + 7 = length of this entry. A length of zero indicates end of table.
1	Indicators	X'80'—Work file can be program generated. X'40'—Work file can be protected. X'20'—Work file must not be empty. X'10'—Work file must be defined. X'08'—Non-pause state only. X'04'—Pause state only. X'02'—Conversational mode only. X'01'—Reserved.
1	Indicators	X'80'—Reserved. X'40'—Virtual memory must be intact. X'20'—Allowed in temporary utility mode. X'10'—Work file can be data file. X'08'—Virtual memory overlayed. X'04'—I/O routines overlayed. X'02'—Prime buffers with work file. X'01'—FIT overlayed.
2	Relative disk address	Displacement of the first sector in the keyword program relative to the start of the system program file.
1	Sector count	Count of sectors occupied by the keyword program.
1	Load address	High-order byte of two-byte core load address. Low-order byte is always X'00'.
n	Keyword	Actual keyword. This field is scanned for a match to the input line buffer.

Figure 3-23. Keyword Table Entry (#ECMAN)

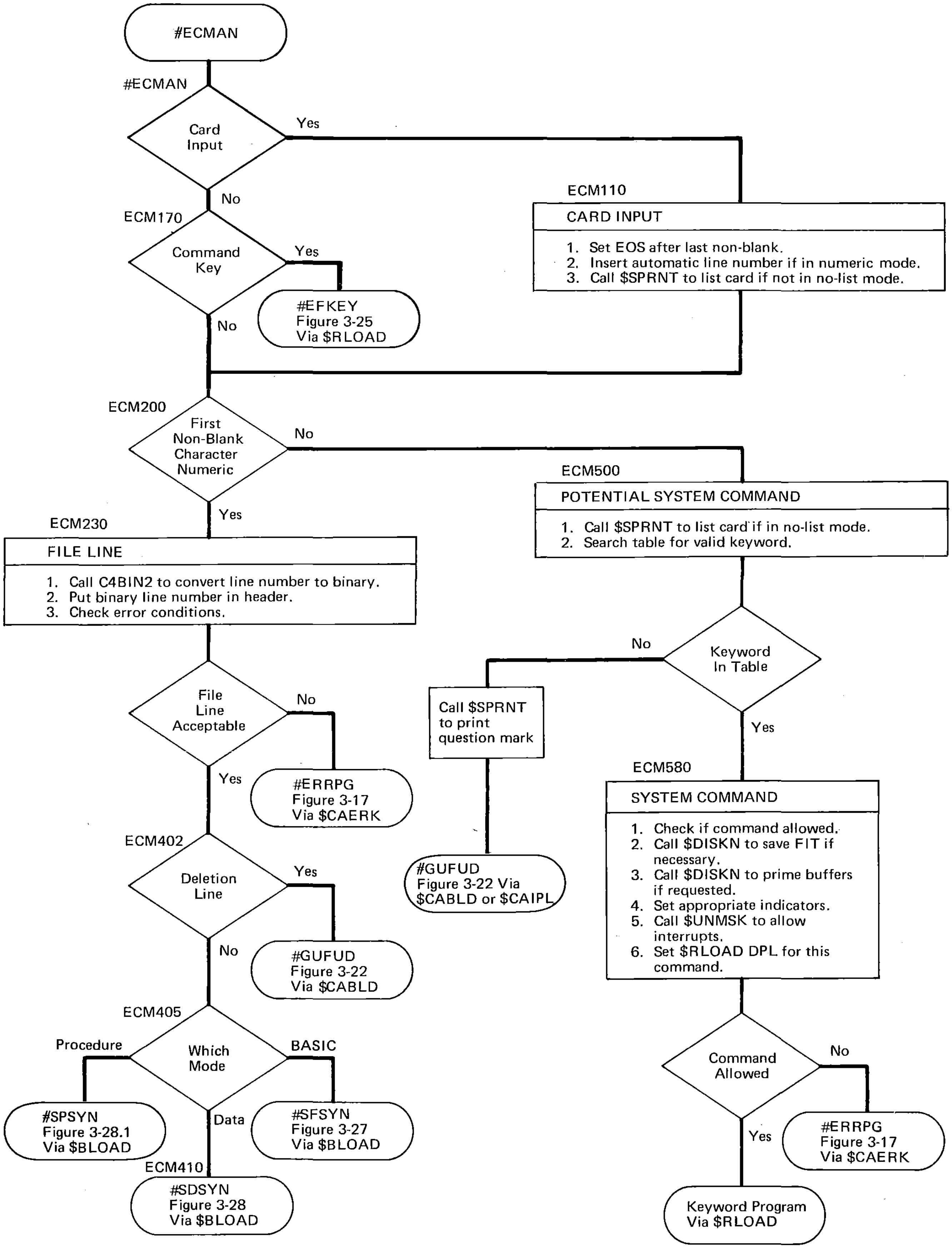


Figure 3-24. Keyword Table Entry (#ECMAN) Flowchart

Command Key Processor—#EFKEY (Figure 3-25)

- #EFKEY processes command keys 1 through 11
- #EFKEY resides in the system program file and is loaded behind the I/O routines in core by the command analyzer (#ECMAN).
- The command key table (##CKTB) contains commands that are either IBM assigned or assigned by the KEYS keyword program.

The command key table (##CKTB) has an entry for each of command keys 1 through 11. Commands in the table are either IBM assigned or assigned by the KEYS keyword program. Figure 5-28 is a list of the IBM assigned command key functions. See Figure 5-27 for the format of the command key table.

If the command length in the table is nonzero, the command text for the specified key is passed to the command analyzer (#ECMAN) in the input line buffer. If the command length is zero, the IBM assigned function for command key 1, 4, or 7 (whichever key is specified) is processed by routines in #EFKEY.

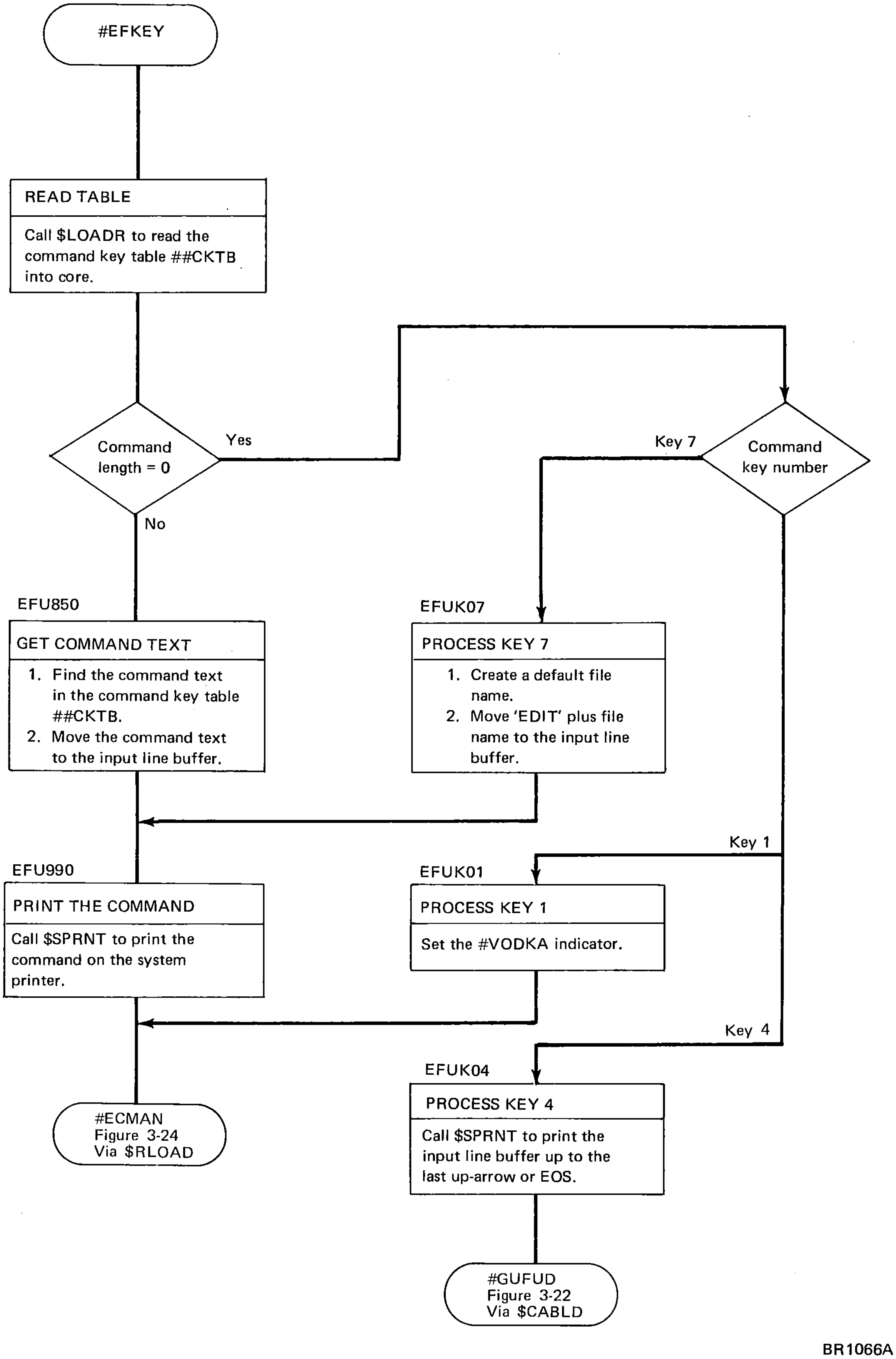


Figure 3-25. Command Key Processor (#EFKEY) Flowchart

BASIC Statement Syntax Checker—#SFSYN (Figure 3-27)

- #SFSYN examines every BASIC statement for valid syntax.
- The assembly of #SFSYN contains the source module, SFSYNC.

If a syntax error is detected, #ERRPG is called to print an up-arrow under the first invalid character of the statement. The index register is loaded with the address of this character. An error code is also loaded into $CAERR in case the user pressed the enter-plus key, requesting a full text message.

If no error is found, a one-byte type code is placed in the byte immediately preceding the BASIC statement in the input line buffer.

#SFSYN scans a statement branch table (Figure 3-26) for the address of one of 18 routines used to syntax check statements. The first two nonblank characters after the line number are used in this scan when a statement keyword is in evidence. For those exceptions where no keyword exists (IMAGE, non-LET assignments), a direct branch is taken to the proper syntax checking routine.

The arithmetic expression routine includes a search through an intrinsic function table that contains 23 entries. Each entry contains the three-byte name of an intrinsic function. The arithmetic expression routine also uses an eight-byte pushdown list to validate nested subexpressions. Each single-byte entry in the pushdown list indicates the validity of a comma appearing in the remainder of the subexpression.

4-Byte Statement Branch Table Entry			
1	2	3	4
Keyword prefix		Routine address	

BR1067

Figure 3-26. Statement Branch Table Entry (#SFSYN)

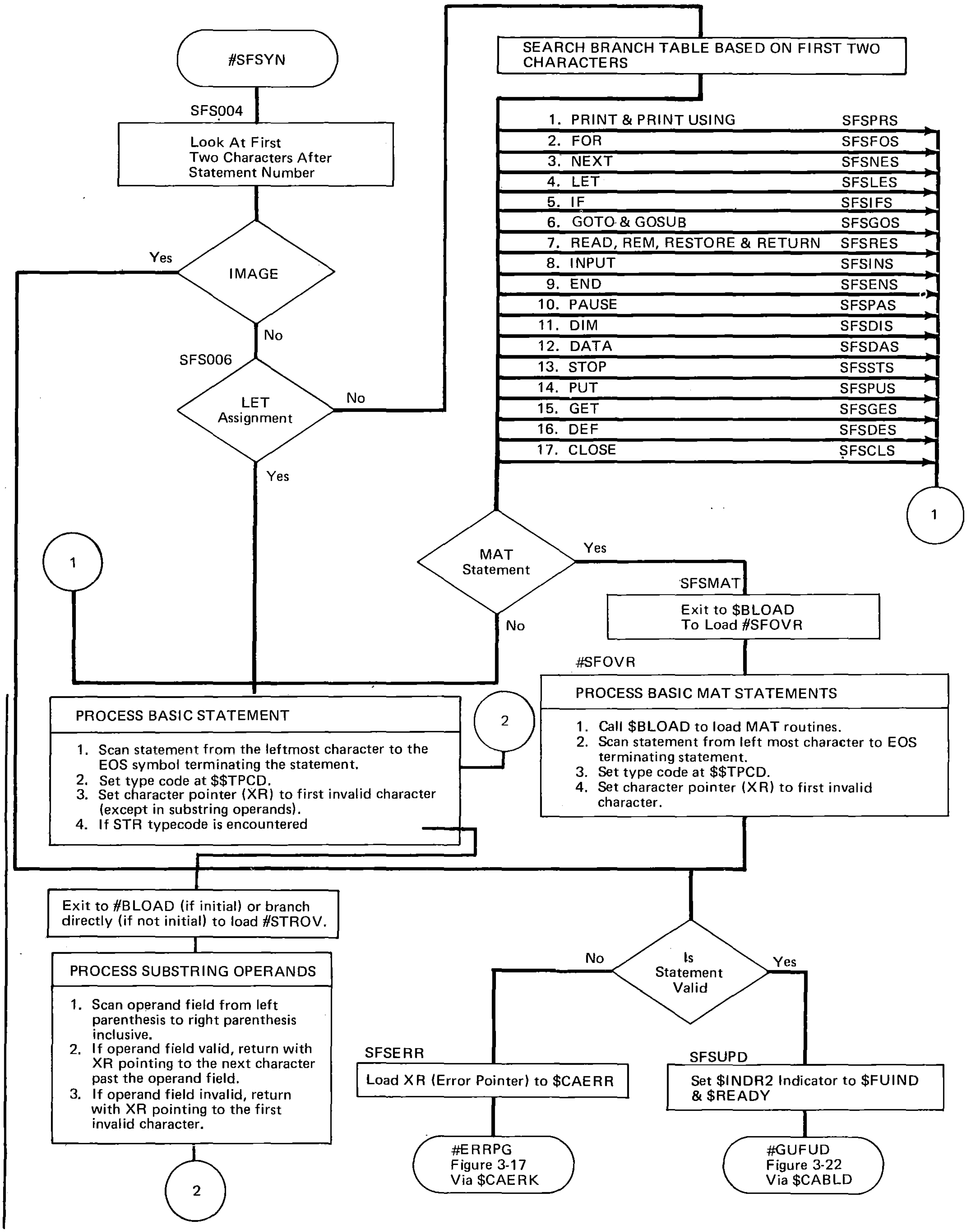


Figure 3-27. BASIC Statement Syntax Checker (#SFSYN) Flowchart

Data Syntax Checker—#SDSYN (Figure 3-28)

- #SDSYN examines data entered when operating under the EDIT DATA command.
- The assembly of #SDSYN contains the source module, SDSYNC.

If a syntax error is detected, #ERRPG is called to print an up-arrow under the first invalid character of the statement. The index register is loaded with the address of this character. An error code is loaded into $CAERR in case the user depresses the enter-plus key, requesting a full text message.

If no error is found, each character or numeric constant is converted to internal form in the secondary input buffer. This buffer is written to the work file by #GUFUD (Figure 3-22). The secondary input buffer contains a header preceding the statement. This header contains the length of the data and header in the secondary input buffer.

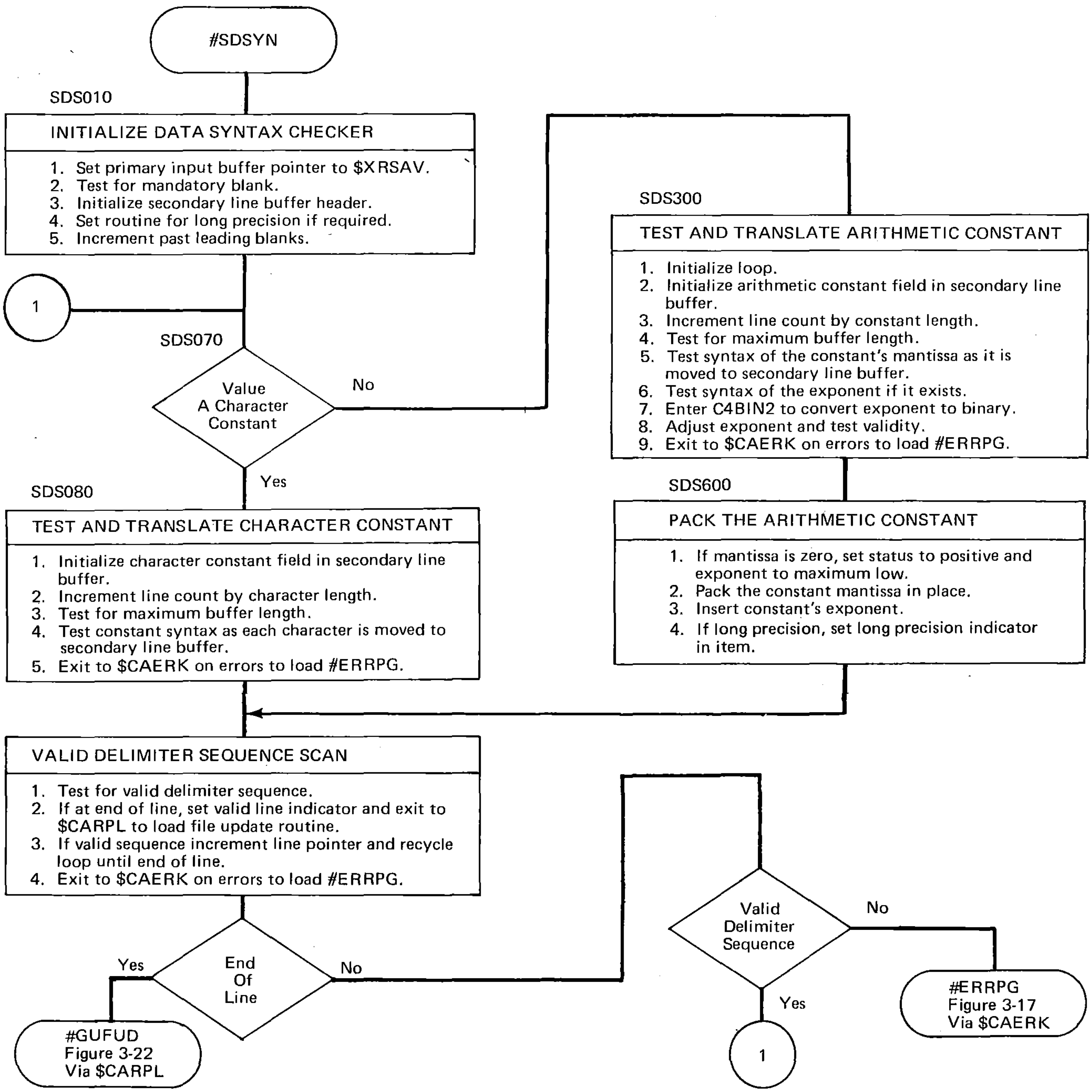


Figure 3-28. Data Syntax Checker (#SDSYN) Flowchart

Procedure Line Checker—#SPSYN (Figure 3-28.1)

- #SPSYN analyzes procedure lines when operating under the EDIT PROCEDURE command.
- The assembly of #SPSYN contains the major source module, SPSYNC.

If a format error is detected, #ERRPG is called to print an up-arrow under the invalid character of the statement. The index register is loaded with the address of this character. An error code is loaded into $CAERR in case the user depresses the enter + key to request a full text message.

If no error is found, each character is moved to the secondary input buffer. This buffer is written to the system work file by #GUFUD (Figure 3-22). The secondary input buffer contains a header preceding the statement. This header contains the length of the procedure line and header in the secondary input buffer.

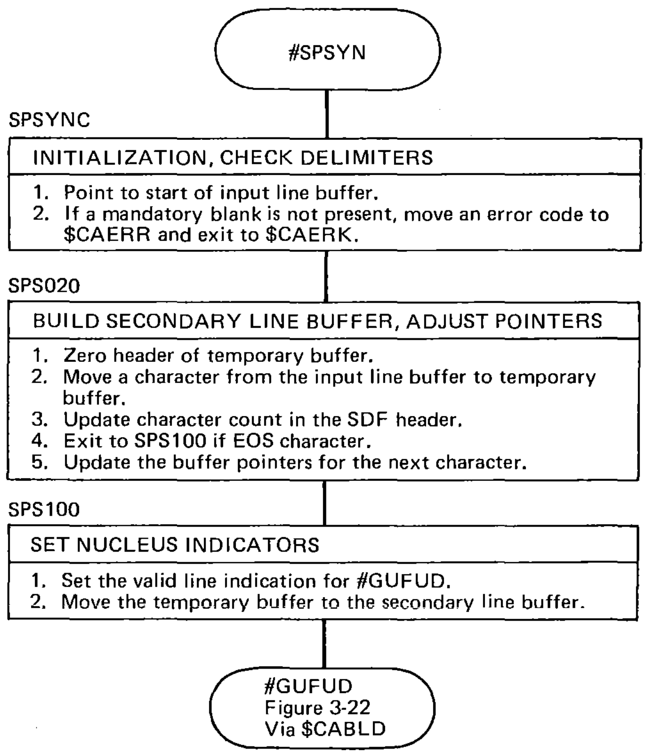


Figure 3-28.1. Procedure Line Checker (#SPSYN) Flowchart

This page is intentionally left blank.

Conversational I/O Routines—#DPRIN

- This program contains two I/O subroutines: DPRINT and DEPRES. Their functions are described in the following paragraphs.

Matrix Printer IOCR—DPRINT (Figure 3-29)

- This routine provides six print I/O functions.
- If an operation is not in progress when a call is made to this IOCR, the operation is started and a return is made to the calling program.
- If a previous operation is in progress, the IOCR does not return until that operation is completed error free and the new operation is started.
- The calling sequence for DPRINT is:

```
Calling sequence for system printer:
B     $SPRNT
DC    AL2(PPL)      PPL is the address of the print parameter list (Figure 5-23).
Calling sequence for a direct call to the matrix printer:
B     DPRINT
DC    AL2(PPL)      PPL is the address of the print parameter list (Figure 5-23).
```

- No checks are made for validity of the PPL.

I/O Functions—DPRINT

Print: The data to be printed must reside in core and be contiguous. Any length of data up to 256 characters can be printed by one call. The IOCR starts printing the data at the current print element position. If the programmed right margin is hit, the print element is returned to the programmed left margin and the form is advanced to the next line. Printing is then completed on the next entry to DPRINT. Upon completion of the print function, the print element is positioned at the next print position after the last character is printed.

Print and Return Element: This operation is the same as print, except the print element is positioned at the programmed left margin on the next line following the completion of print.

Return Element: The print element is positioned at the programmed left margin and the form is advanced to the next line.

Backspace and Index: This operation moves the print element left one print position and indexes (advances) the forms one line. If the left margin is hit, no more spacing is done.

Backspace: This operation is the same as backspace and index except no index is performed.

Wait and Check for Errors: To allow printer overlap, a special wait function is provided. The IOCR waits for the previous operation to be completed and then checks for errors. If the previous operation hit the programmed right margin, a new operation to continue printing on the next line(s) is started and completed before a return is made.

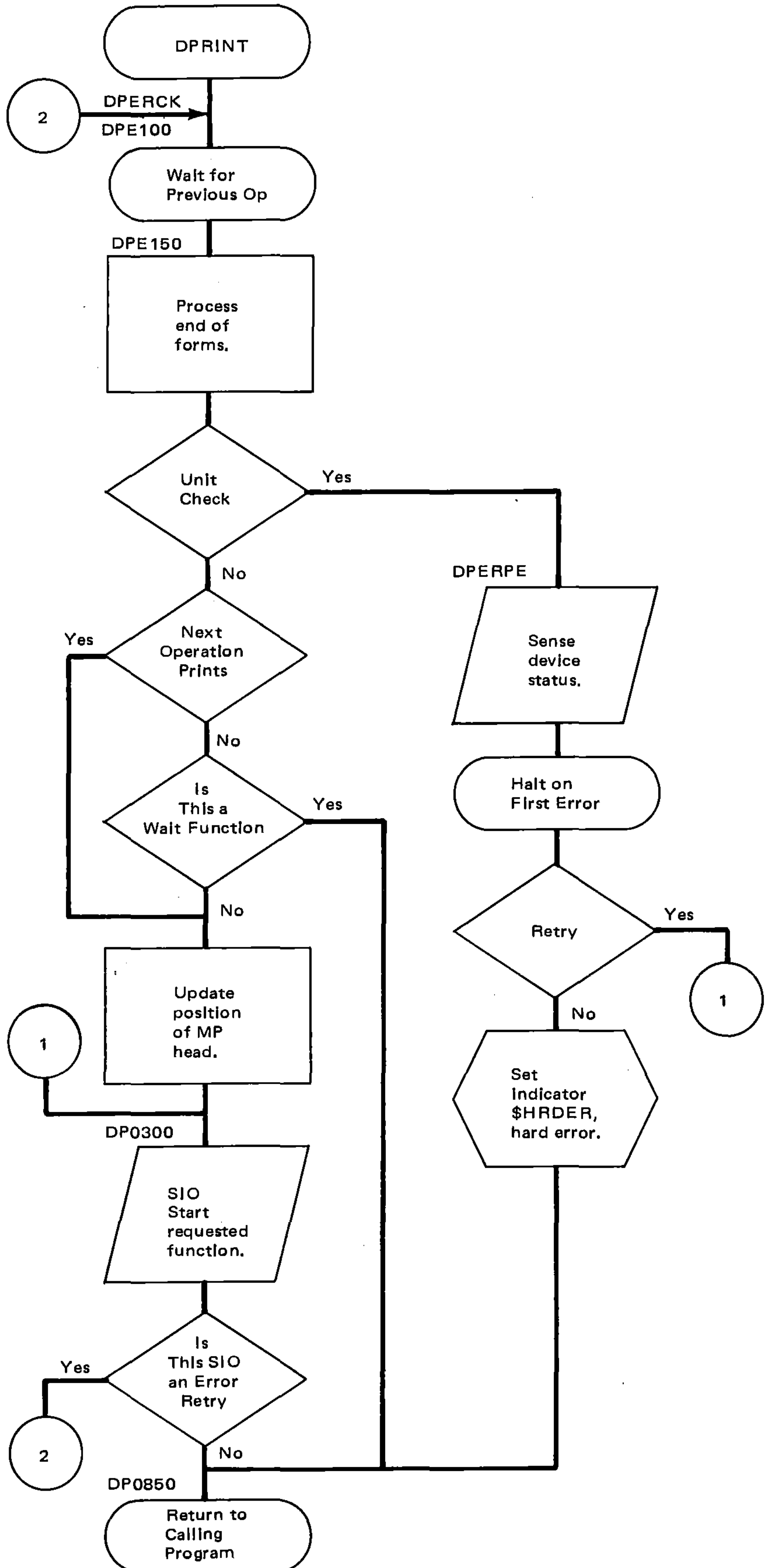


Figure 3-29. Matrix Printer IOCR (DPRINT) Flowchart

Keyboard IOCR—DEPRES (Figure 3-30)

- DEPRES handles input from the keyboard.
- DEPRES is divided into two sections:
  1. Call section—Enables interruptions and unlocks the keyboard in preparation for line input. It sets the interruption address to the interruption section. When a key is pressed, the interruption section is entered on the keyboard interruption level.
  2. Interruption section—Saves the system status (BR, XR, PSR, ARR, P1-IAR) and handles the data input from the keyboard. Upon completion of the input line, $KYBSY is set to zero, indicating the line is complete. The keyboard is then locked (inquiry request is never locked).
- Entry Points. When line input or a command key is desired by the calling program, the call section of DEPRES is called, unlocking the keyboard and setting the line input indicator ($KYBSY): B DEPRES. If only a command key or a function key is desired, $CMDKY is set on by the calling program, indicating to the keyboard IOCR that only command keys and interruption requests are to be recognized.
- Exits. Exit from the call section is to the calling program; exit from the interruption section is to the interrupted program.
- Data parity is checked.

Key Functions (DEPRES)

Data Keys: The character is placed in the input line buffer and printed on the system printer.

Tab Keys: If the current position in the line buffer is pointing within an existing line, the old character is printed. If it is not, a blank is printed. This positions the carrier one space to the right. If the key is held down, the typamatic feature is activated and the spacing operation is repeated until the key is released.

Backspace Key: If the system printer is the matrix printer, and if this was the first backspace for the current line, the carriage is indexed and backspaced one position. Otherwise, the index feature is not executed. If the key is held down, the typamatic feature is activated and the backspace operation is repeated until the key is released.

Return Key: The carriage is returned on the system printer and $KYBSY is set to zero, indicating the line is complete. The keyboard is then locked.

Erase Key: ERASE is printed, and the carriage is returned on the system printer, allowing the line to be reentered.

Inquiry Request Switch: Depending upon the mask status, the current operation is aborted. This switch, on the keyboard console, cannot be locked.

Program Start Key: The data is sensed and saved. If it is the start of a line, the auto line is printed. This key is also used to start execution when the system is in pause mode.

Enter-Minus Key: Printer (if in use) is indexed one line.

Enter-Plus Key: Used to invoke the second-level error message.

Command Keys 1 through 11: If the print element (or CRT cursor) is at the left margin, the command key indicator is placed in the input line buffer.

Other Command Keys: If the CRT is present, DSPLYN is called to perform the function requested.

Error Procedures

A data register parity error is retried once. The system halts upon such an error, indicating to the user that a parity error has occurred. The system start switch must be activated to continue. Two successive parity errors cause a system-generated hard halt. An IPL must be initiated to recover from a hard halt.

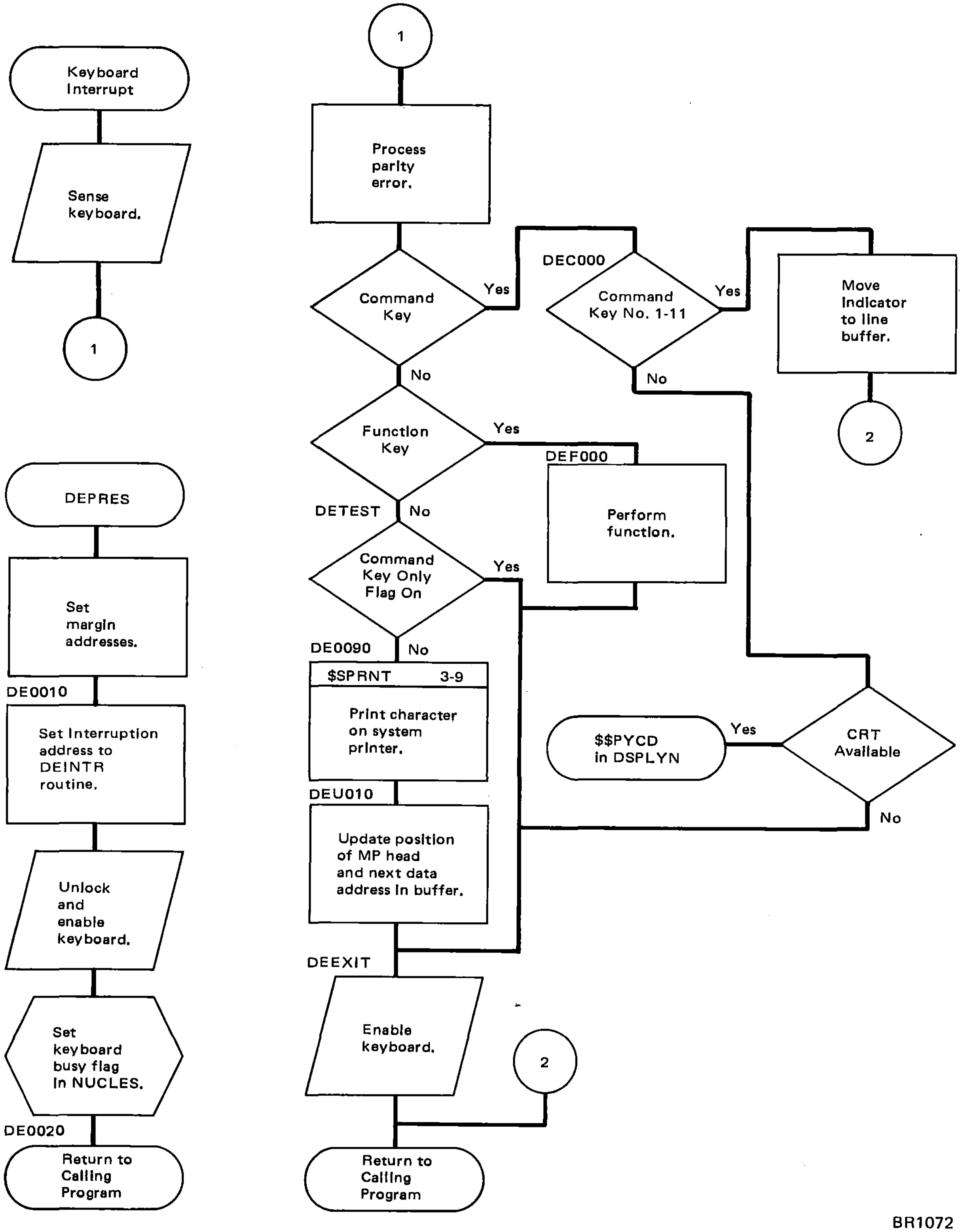


Figure 3-30. Keyboard IOCR (DEPRES) Flowchart

Card Reader I/O Routine—#DREAD (Figure 3-31)

- #DREAD provides two functions:

  1. Reads a card into the input line buffer.
  2. Tests to see if the card reader is busy.

- This routine overlays the keyboard routine (DEPRES) in core when the input mode is cards rather than keyboard.

- Entry points:

  1. DREADN—Initiates the reading of a card. This entry is the same as enabling and unlocking the keyboard when keyboard is desired (refer to "Keyboard IOCR—DEPRES").
  2. CRDBSY—This entry is to test completion of the card read function.

#DREAD is called into core by the work file update/crusher program (#GUFUD), overlaying the keyboard routine (DEPRES), when card input is specified.

The calling program branches to DREADN in #DREAD to read a card in the card reader. A check is made to see if the card reader is busy. If the card reader is busy, a branch is made to the card busy routine (CRDBSY). If the card reader is not busy, and is ready to operate, the reading of a card is started. #DREAD exits to the calling program while the card is being read. The calling program must then reenter #DREAD at entry point CRDBSY to test for successful completion of the card read function.

CRDBSY is entered to see if the card reader is busy and if an error is indicated in the card reader. If it is not busy and no error is indicated, #KYBSY indicator is set to 0, indicating completion of the card input, and return is made to the calling program. If the card reader is not busy and an error is indicated, the error pending indicator is set on, and the CRDBSY routine is reentered to retest for the error indication. A soft halt results if there is a compare error or a transport jam. The error test is made for a maximum of five times. A hard halt results after the fifth try if an error still exists.

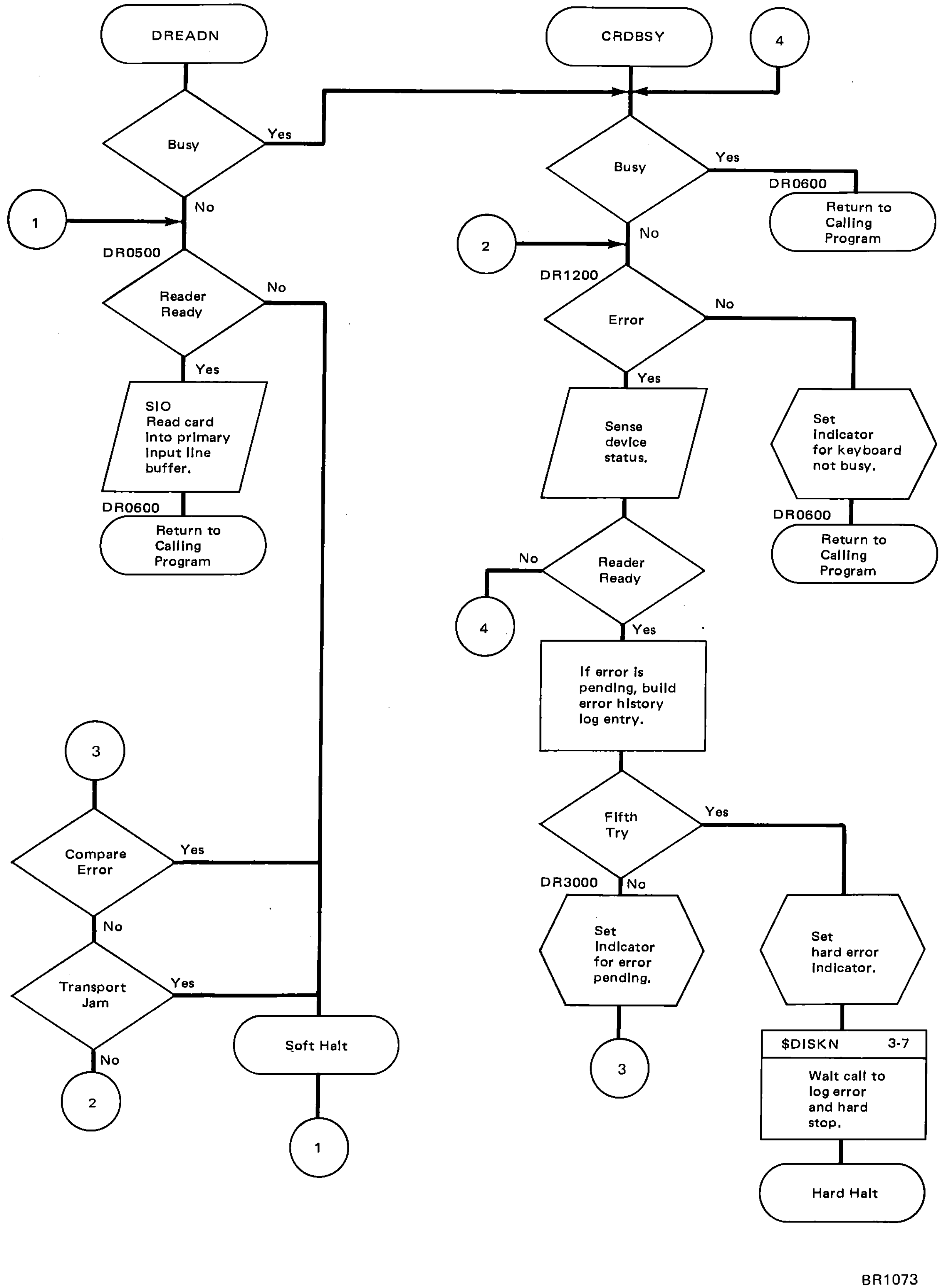


Figure 3-31. Card Reader IOCR (#DREAD) Flowchart

Procedure File Line Processor—#GRAPR (Figure 3-31.1)

- #GRAPR performs 3 functions:
  1. Reads a procedure file line into the input line buffer
  2. Simulates a card reader not busy condition
  3. Places READ KEY in the input line buffer after all the lines in the procedure are processed.
- This routine overlays the keyboard routine (DEPRES) when the input mode is procedure file rather than keyboard or cards.
- Entry points to #GRAPR are:
  1. GRAPRO—Reads one SDF unpacked line. This entry is the same as enabling and unlocking the keyboard when input is from the keyboard.
  2. $$CDBS—Simulates a test for completion of the card read (not busy).

#GRAPR is loaded to main storage by the work file update/crusher (#GUFUD) and overlays the keyboard routine (DEPRES) when PROCEDURE FILE is specified.

The calling program branches to GRAPRO to read a procedure file line. #GRAPR extracts sequential procedure text lines unpacked and stripped of SDF fields and puts them in the input line buffer. The index register (@XR) points to the next binary line number. #GRAPR returns to the calling program after the procedure line is in the input line buffer.

$$CDBS is entered to simulate a check for a card not busy condition. The indicator #KYBSY is reset to zero to indicate a not busy status. #GRAPR then returns to the calling routine.

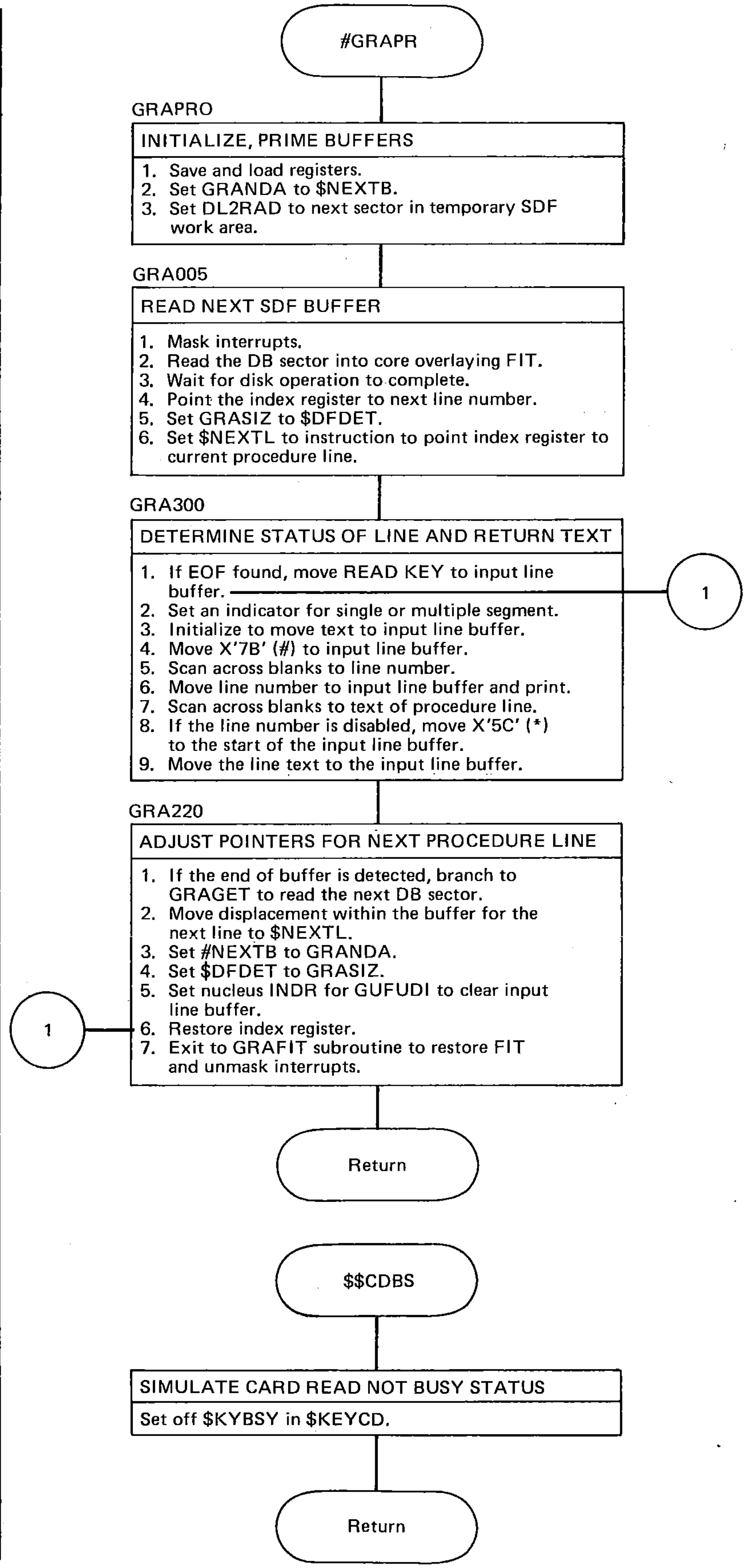


Figure 3-31.1. Procedure File Line Processor (#GRAPR) Flowchart

CRT I/O Routine—#DSPLY (Figure 3-32)

- DSPLYN is the IOCR used for displaying output to the CRT.
- It is used in place of (or with) DPRINT. When the CRT is designated as the system printer, #DSPLY is used. When both the matrix printer and the CRT are designated as the system printer, DPRINT and #DSPLY are used.
- Calling sequences to #DSPLY are:

Calling sequence for system printer:

```
B      $SPRNT
DC     AL2(PPL)        PPL is the address of the print parameter list (Figure 5-23).
```

Calling sequence for a direct call to print on the CRT:

```
B      DSPLYN
DC     AL2(PPL)        PPL is the address of the print parameter list (Figure 5-23).
```

Calling sequence for a direct call to print on both the CRT and matrix printer:

```
B      DSPYMP
DC     AL2(PPL)        PPL is the address of the print parameter list (Figure 5-23).
```

Calling sequence used to clear the CRT screen:

```
B      DSPCMD
```

- The address in the calling sequences must be relocated by the value in $EXFTR.

This routine is normally called via the nucleus interface $SPRNT, which decides the device to be used for output. DSPLYN handles all functions used by DPRINT plus additional features for the CRT. If these additional functions are used, the calling program must know that the CRT is being used.

Printer/CRT Functions

- The following functions can be performed on the matrix printer and the CRT:

Print: Data is displayed starting at the current display position and continuing, line by line, until all characters have been displayed.

Print and Return: This function is the same as print, except that the next position to be displayed is at the start of the next line.

Return: The next position to be displayed is at the start of the next line.

Tab Left/Tab Left and Index: The CRT cursor (next print position) is moved to the left (backspaced) one position. No indexing is done. If the cursor reaches the left position of the statement and another tab left is issued, the cursor remains there.

Tab Right: The cursor is moved right the desired number of positions. If the physical right margin is encountered, the cursor is moved to the left margin and the displayed lines are indexed.

Wait: This function tests the CRT for errors.

- The following function is for CRT use only:

Roll Down and Print: The displayed lines are rolled down and the new data is displayed on the top line. A maximum 64-byte character string can be used with this function.

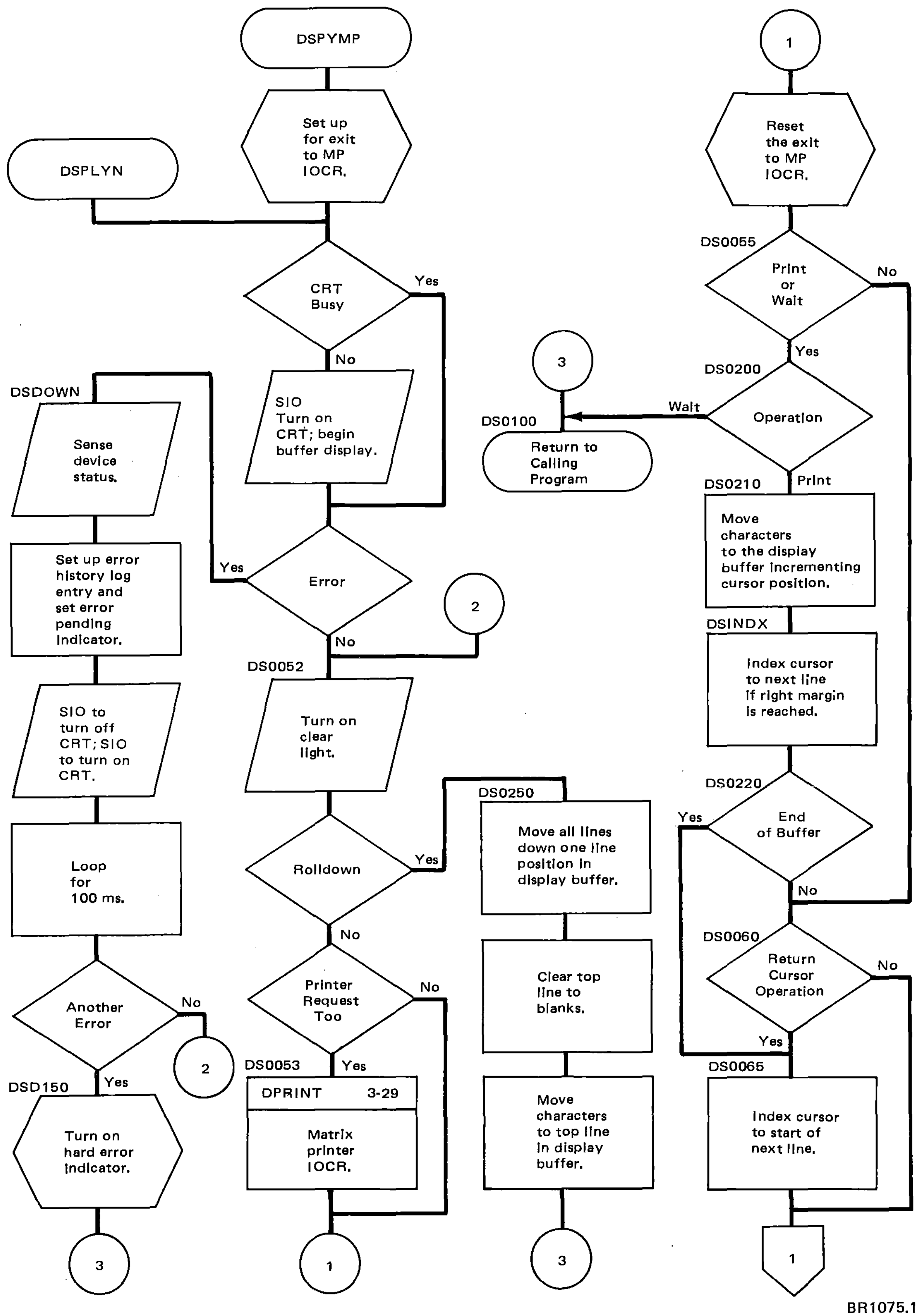


Figure 3-32. CRT IOCR (#DSPLY) Flowchart (Part 1 of 2)

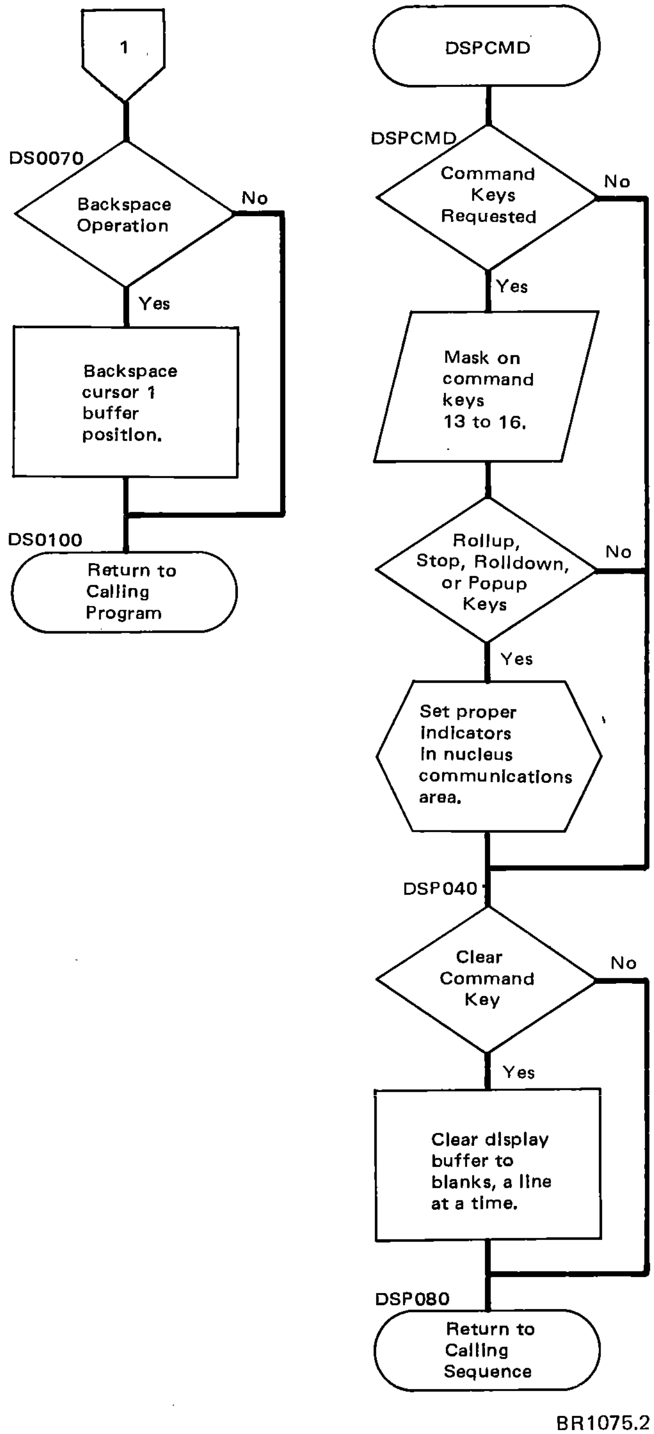


Figure 3-32. CRT IOCR (#DSPLY) Flowchart (Part 2 of 2)

Maintenance Program Load Trace—#ZTRAC (Figure 3-33)

- #ZTRAC is called to print the program header of every program loaded by the system nucleus (NBLOAD) when the branch in NBLOAD is active.
- To reverse the trace program to the opposite status, enter maintenance utility mode, key in a T, and press the return key.

#ZTRAC is loaded and executed on each entry to NBLOAD before the requested program is loaded to core (see Figure 3-33). #ZTRAC is loaded to the same core address as the requested program.

#ZTRAC reads the first sector of the program being loaded. The first six characters of this sector are displayed on the matrix printer at the physical left margin.

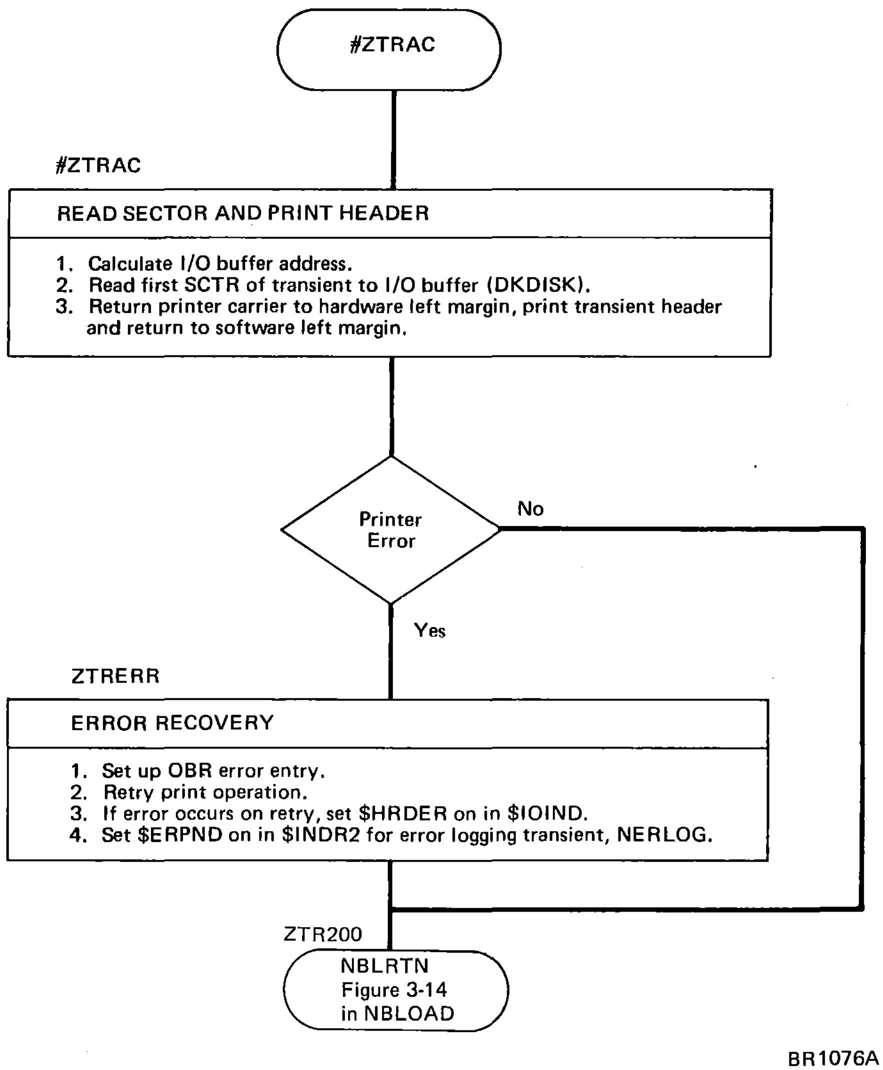


Figure 3-33. Maintenance Program Load Trace (#ZTRAC) Flowchart

KEYWORD PROGRAMS

- Keyword programs are described in alphabetical sequence.

ALLOCATE Keyword Program—#KALLO (Figure 3-35)

- #KALLO defines data file attributes for the BASIC program in the work file.
- The assembly of #KALLO contains these major source modules:

KALLOC—Mainline logic, Figure 3-35
SVOLID—Search volume-ID table, Figure 3-76
SGETDB—Search password directory, Figure 3-77
SRCHFN—Search user directory, Figure 3-78
SFINDF—Find library file, Figure 3-83
SURCHN—Search null directory, Figure 3-81
STUFID—User directory insert, Figure 3-80
DL2ICS—Disk logical IOCS, Figure 3-70

#KALLO will load #SPACK, Figure 3-86, when disk space can be obtained by packing the file library. #SPACK loads, and returns to, #KALLO.

Functions of #KALLO are:

1. Build a user directory entry to reserve space for NEW, PERMANENT, DISK files.
2. Update the work file I/O record with data file information from the command parameters. This record is used at execution time by the GET/PUT routines to define the I/O device and disk location, if the device is disk, for data files referenced by the BASIC program.

KALLOC builds an entry for the system work file I/O record (Figure 3-34).

Hexadecimal Displacement	Decimal Displacement	Length	Description
00	0	1	Device code
01	1	8	GET/PUT name
09	9	2	SCRATCH file size
09	9	6	Disk label
0F	15	8	Password
17	23	8	Filename
1F	31	1	Unused

Note: Each active entry must define a device code and GET/PUT name. The content and meaning of the other fields depend upon the device code.

BR1077

Figure 3-34. Entry for I/O Record

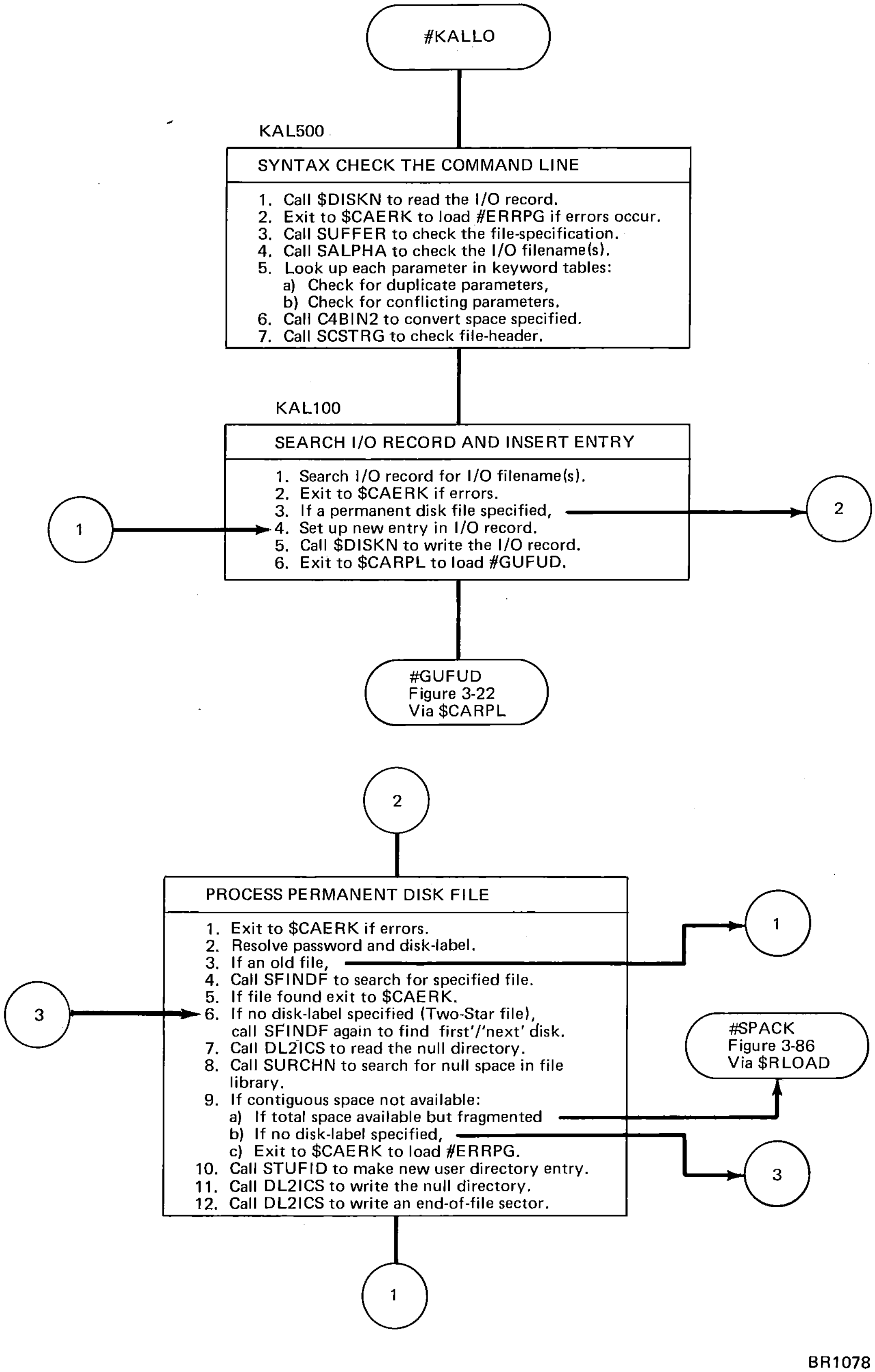


Figure 3-35. ALLOCATE Keyword Program (#KALLO) Flowchart

CALL Keyword Program—#KCALL (Figure 3-35.1)

- #KCALL calls procedure files from the user library.
- The assembly of #KCALL contains these major source modules:

  KCALLN—Mainline logic, Figure 3-35.1

  GRABIT—Work file input, Figure 3-74

  SFINDF—Find library file, Figure 3-75

  SVOLID—Search volume-ID table, Figure 3-76

  SGETDB—Search password directory, Figure 3-77

  SRCHFN—Search user directory, Figure 3-78

The functions of #KCALL are:

1. Syntax check the CALL command
2. Find a saved procedure file in the user library
3. Copy the procedure file to the temporary procedure save area on disk
4. Initialize indicators in the system nucleus for the call sequence

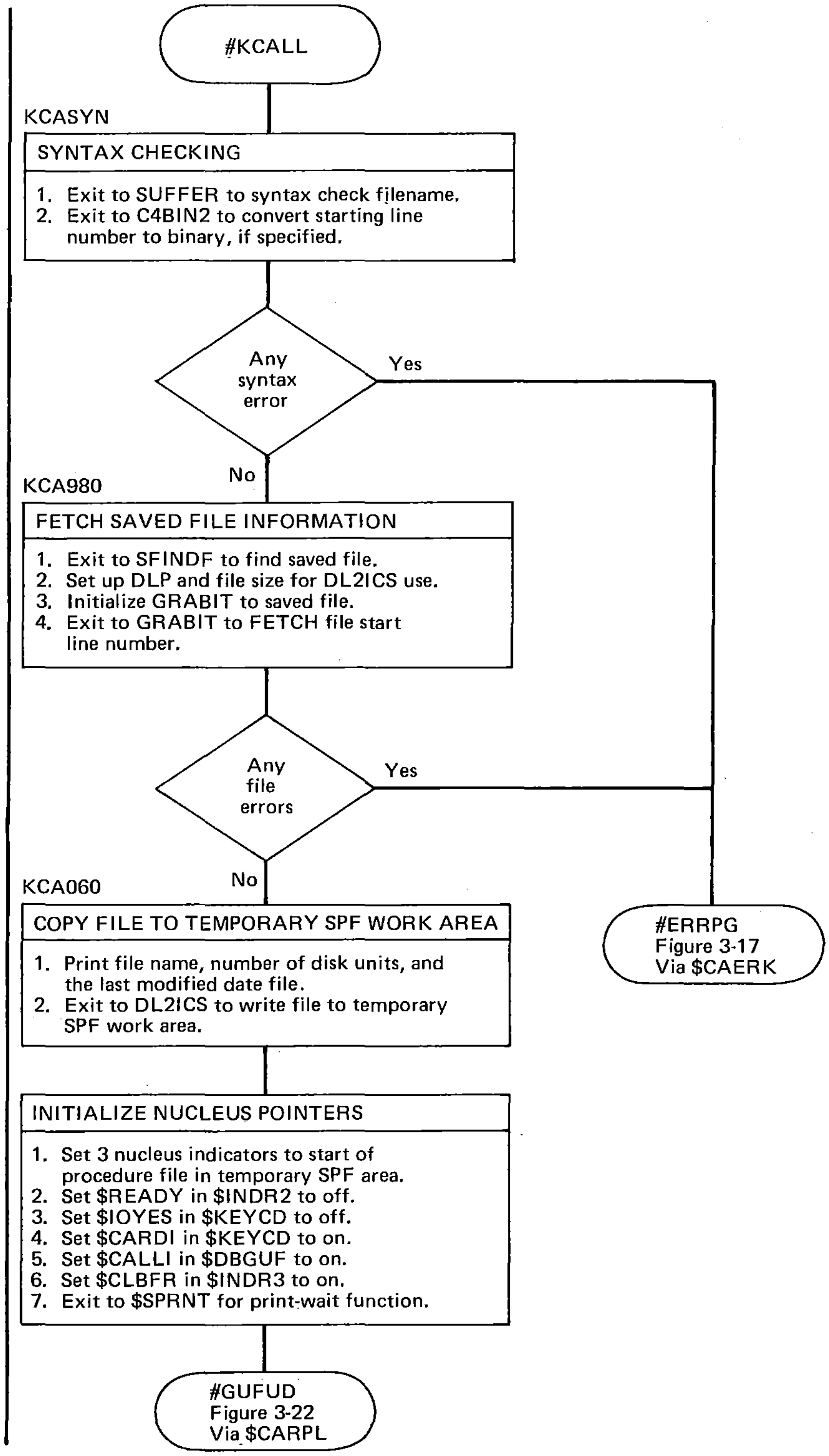


Figure 3-35.1. CALL Keyword Program (#KCALL) Flowchart

CHANGE Keyword Program—#KCHAN (Figure 3-36)

- #KCHAN alters a previously entered line without reentering the entire line.
- The assembly of #KCHAN contains these major source modules:

  KCHANG—Mainline logic, Figure 3-36
  GFINDN—Locate work file disk block, no flowchart
  GRABIT—Work file input, Figure 3-74
  SDLIST—List data files, no flowchart
  DL4ICS—System work file IOCS, Figure 3-70

The CHANGE command is used to alter a previously entered line without retyping the entire line. If a line number parameter is present, the specified line number from the work file is changed. If no line number is present, the last line entered containing a syntax error is changed. The line may be a file line or a system command.

#KCHAN performs the text replacement on the specified line and then prints the changed line. When the line is printed, the carriage is not returned. At this point, the system operates as if the user has just entered the line printed by the CHANGE command but has not yet entered the carriage return. The backspace and tabulate keys may now be used to modify what appears to be the original line. When the operation is terminated by a carriage return, the changed line is accepted as a normal keyboard input line and the appropriate action is taken. If the changed line exceeds the current width, an automatic carriage return is given. The same procedure is followed if the command is input from the data recorder.

The optional character string constant parameters define the text changing to be performed. In addition, further changes may be performed with the use of the FIRST or ALL parameter. Basically, the first occurrence in the line of the first character string constant is replaced by the second character string. If ALL is specified, all occurrences of the first character string are replaced by the second character string.

Character strings can be of different lengths. The portion of the original line following the text to be changed is moved to immediately follow the replacement string in the new line. If the second character string is missing, it is assumed to be a null string; the first string, and everything following, is eliminated from the line. (This is not the same as replacing the first string with blanks.) If the first character string parameter is the null string, the second string is inserted before the first character in the original line.

A line must be present at the disk address equated to #@#BAD (bad-line buffer) if no line number parameter is present. The line is assumed to be in the active work file if a line number parameter is present.

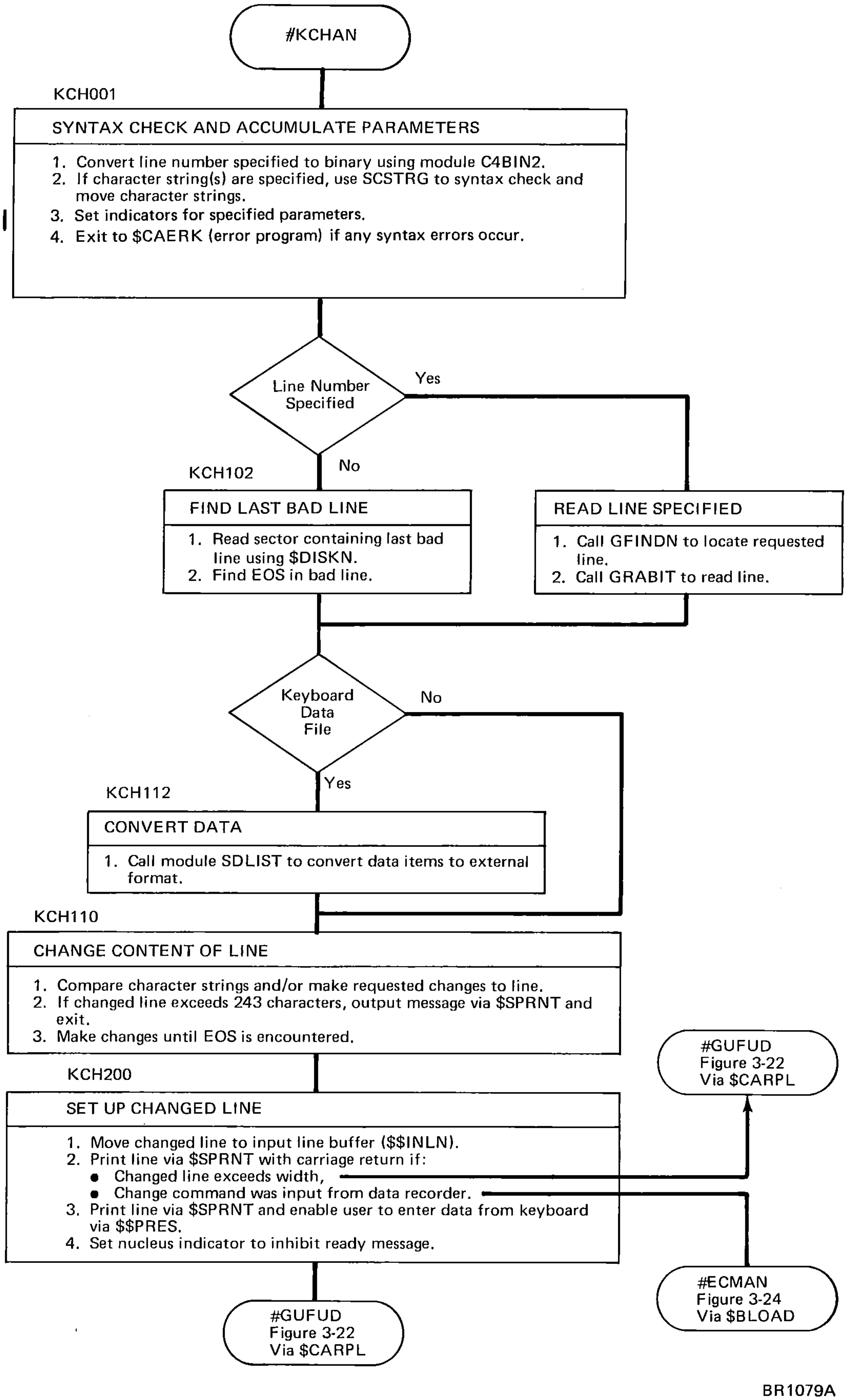


Figure 3-36. CHANGE Keyword Program (#KCHAN) Flowchart

CONDITION Keyword Program—#KCNDI (Figure 3-37)

- #KCNDI displays current system status information on the device assigned as system printer.
- The assembly of #KCNDI contains these major source modules:

  KCNDIT—Mainline logic, Figure 3-37
  DLPRNT—IOCS for output, Figure 3-71

#KCNDI displays the following current system status information derived from the contents of the system communication area (Figure 5-1) and disk areas (Figure 5-2).

1. Whether or not a password is logged-on.
2. Whether or not a disk label is logged-on.
3. Status of the disk-label table.
4. Date.
5. Left margin and width values for the printer.
6. System mode.
7. Name of suspended BASIC program (if any).
8. Status of the system work file.
9. Information about the file in the system work file (name, status, type, number of lines, number of disk units, etc.).
10. Status of the system configuration record.
11. Information concerning the files currently allocated (device, GET/PUT filename, etc.) if any exist.

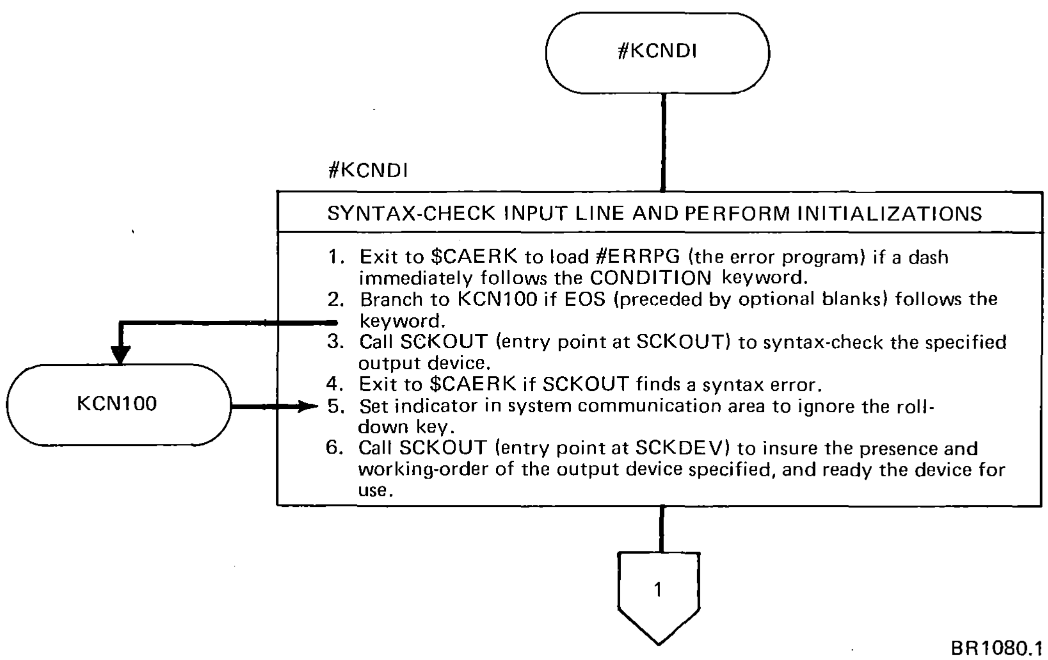


Figure 3-37. CONDITION Keyword Program (#KCNDI) Flowchart (Part 1 of 2)

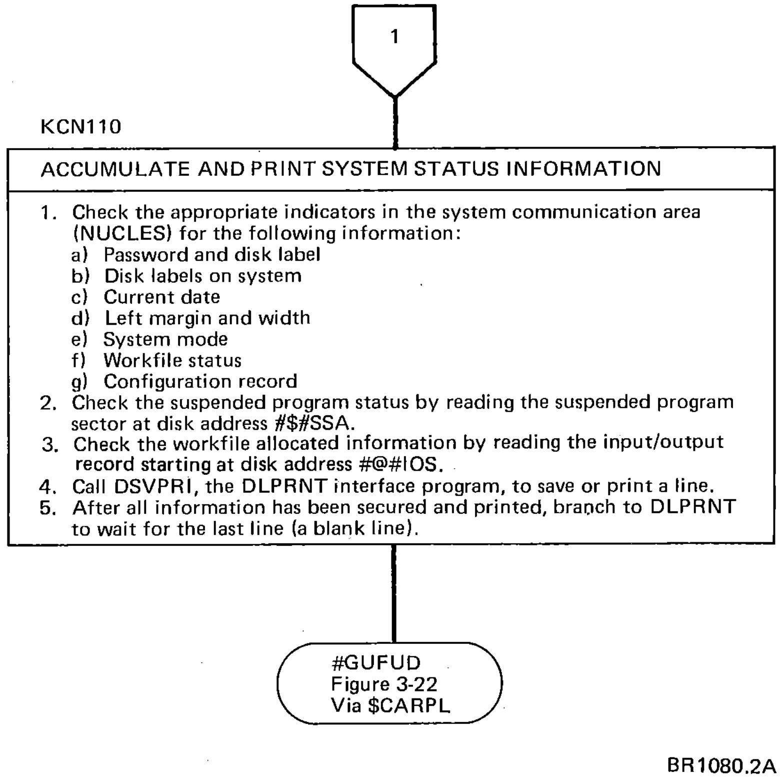


Figure 3-37. CONDITION Keyword Program (#KCNDI) Flowchart (Part 2 of 2)

DELETE Keyword Program—#KDELE (Figure 3-38)

- Three options that #KDELE can perform are:

  1. Delete a line number list from an active file in the work file (passes a delete parameter list, Figure 5-26, to #GUFUD to do this).
  2. Delete a file linked to a specified password.
  3. Delete all files, linked to a specified password, that are not pooled or protected. The password is also deleted if all files linked to it are deleted.

- The assembly of #KDELE contains these major source modules:

  KDELET—Mainline logic, Figure 3-38
  DL2ICS—Disk logical IOCS, Figure 3-70
  STORIN—Null directory insert, Figure 3-79
  SFINDF—Find library file, Figure 3-75
  SGETDB—Search password directory, Figure 3-77
  SRCHFN—Search user directory, Figure 3-78
  SVOLID—Search volume-ID table, Figure 3-76

As each file is deleted, the disk space occupied is linked into the null directory. #KDELE loads #SPACK, Figure 3-86, to pack the file library if the null directory is full. #SPACK loads, and returns to, #KDELE.

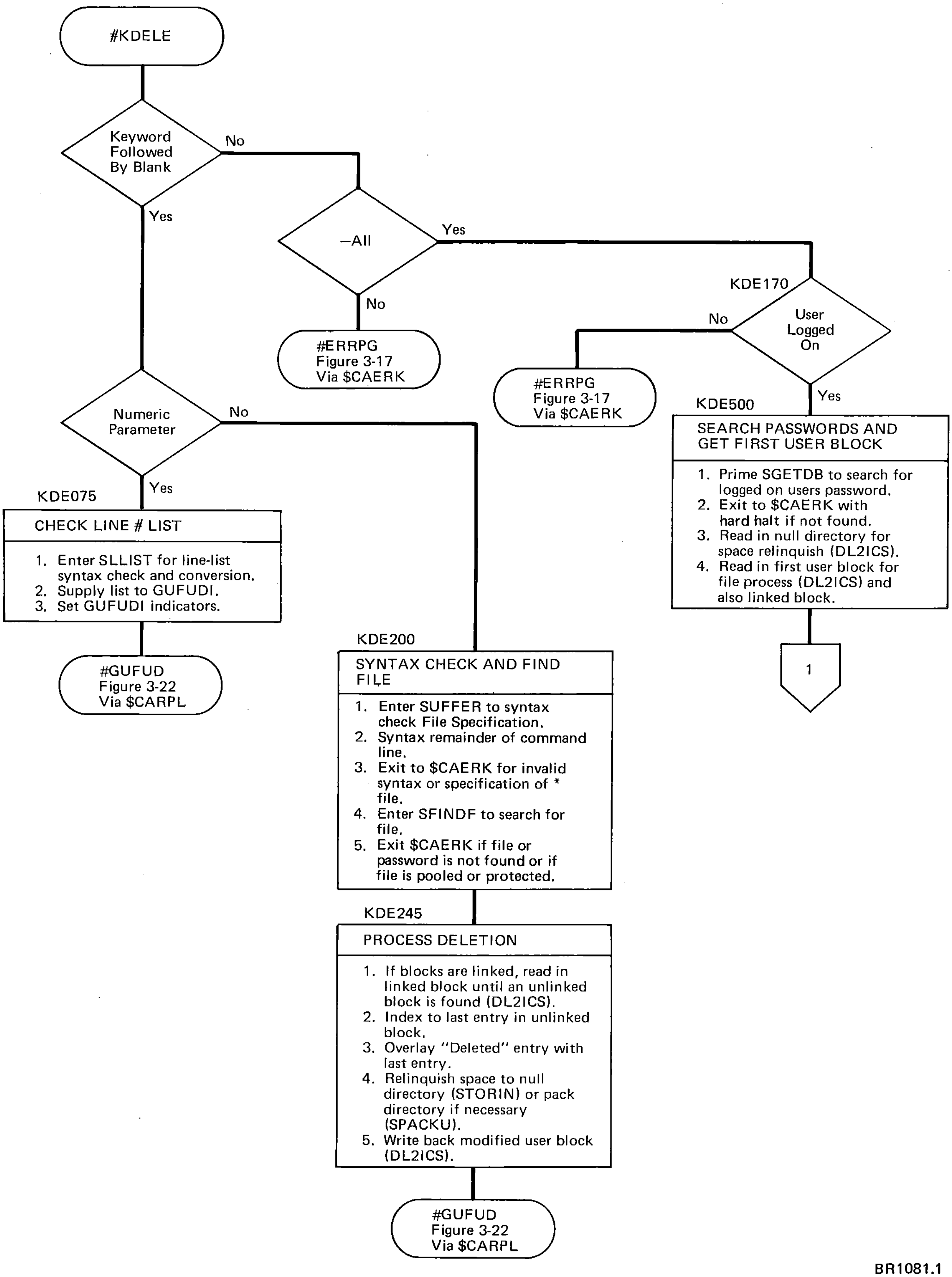


Figure 3-38. DELETE Keyword Program (#KDELE) Flowchart (Part 1 of 2)

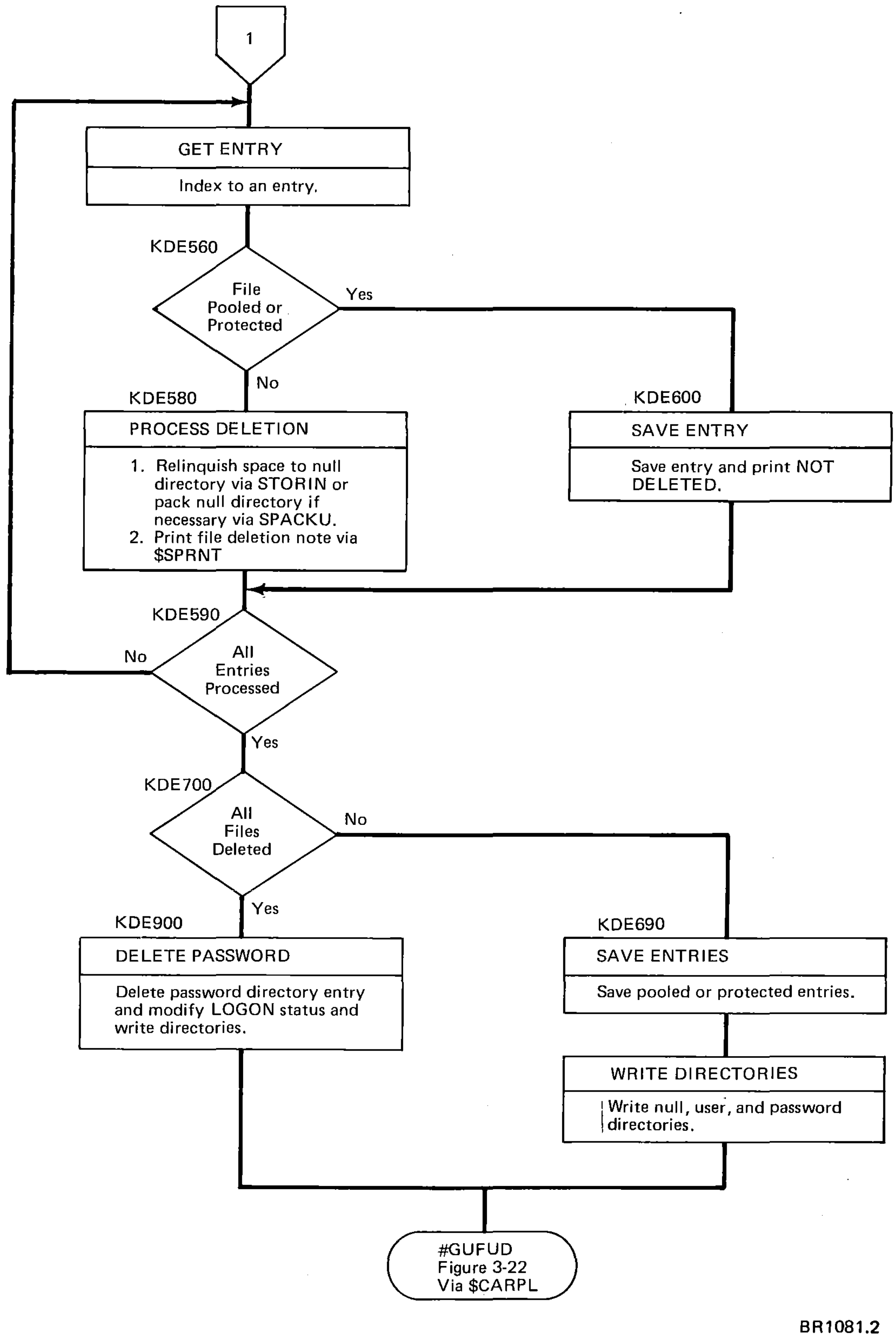


Figure 3-38. DELETE Keyword Program (#KDELE) Flowchart (Part 2 of 2)

DISPLAY Keyword Program—#KDISP, #KDOVR (Figure 3-39)

- #KDISP syntax checks the DISPLAY command line, assuring valid syntax for the DISPLAY overlay #KDOVR.
- #KDOVR displays the current values of program variables during a program execution pause state or following the termination of program execution.
- The assembly of #KDISP contains these major source modules:

  KDISPL—Mainline logic, Figure 3-39
  SCKOUT—Check output specification, no flowchart

- The assembly of #KDOVR contains these major source modules:

  KDOVRL—Mainline logic, Figure 3-39
  DL4ICS—System work file IOCS, Figure 3-70
  DLPRNT—IOCS for output, Figure 3-71

The DISPLAY command line is syntax checked. When correct syntax is assured, the overlay initialization is performed. If in pause mode, the virtual-memory pages in the paging module are returned to virtual memory. The symbol tables are placed in core, and the overlay #KDOVR is loaded.

#KDOVR converts each specified variable or array element symbol to a virtual address. The element value at this address is retrieved from virtual memory, converted to display format, and displayed on the matrix printer, CRT, or system printer. The All parameter causes each scalar variable to be displayed. Symbol format 'A(*)' causes each element in array A to be displayed according to current array dimensions. Symbol format 'A$(*)' causes each element in array A$ to be displayed.

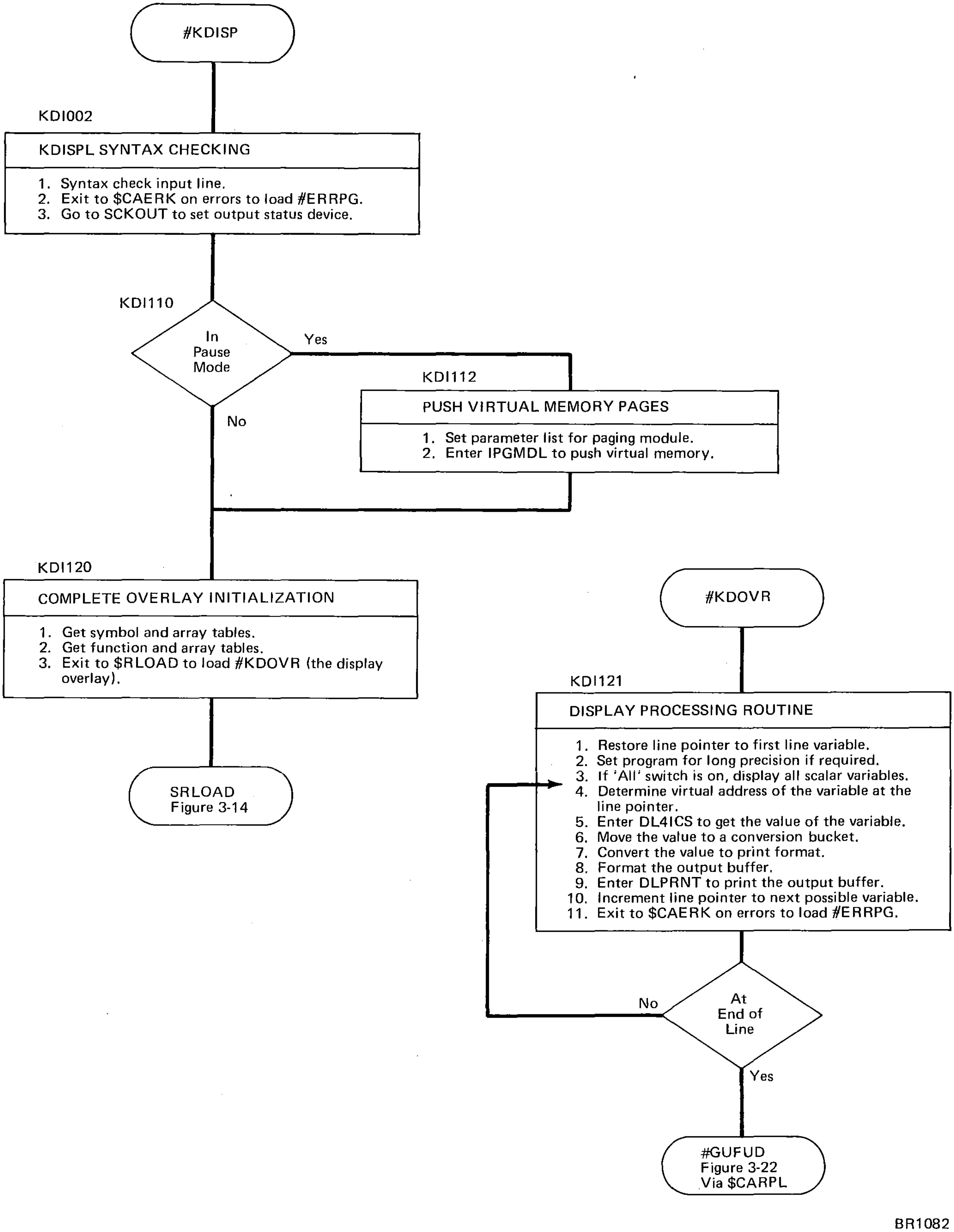


Figure 3-39. DISPLAY Keyword Program (#KDISP) Flowchart

EDIT Keyword Program—#KEDIT (Figure 3-40)

- #KEDIT places a specified file into, or clears, the work file.
- The assembly of #KEDIT contains these major source modules:

  KEDITN—Mainline logic, Figure 3-40
  SVOLID—Search volume-ID table, Figure 3-76
  SGETDB—Search password directory, Figure 3-77
  SRCHFN—Search user directory, Figure 3-86
  SFINDF—Find library file, Figure 3-75
  GCLEAR—System work file clear, no flowchart
  DL2ICS—Disk logical IOCS, Figure 3-70
  DL4ICS—System work file IOCS, Figure 3-70

Functions of #KEDIT are:

1. Move the specified file from the user, one-star, or two-star library file to the work file. The work file is cleared and #KEDIT exits if only a user filename is specified and cannot be found.
2. Set the work file status indicators, $INDR1 in the nucleus communications area, to reflect the status of the work file.
3. Load, and exit to, the compiler (#BCOMP) if the system command was RUN or STEP.
4. The data buffer, used to transfer the file, overlays routines in #KEDIT that were used to find the file.

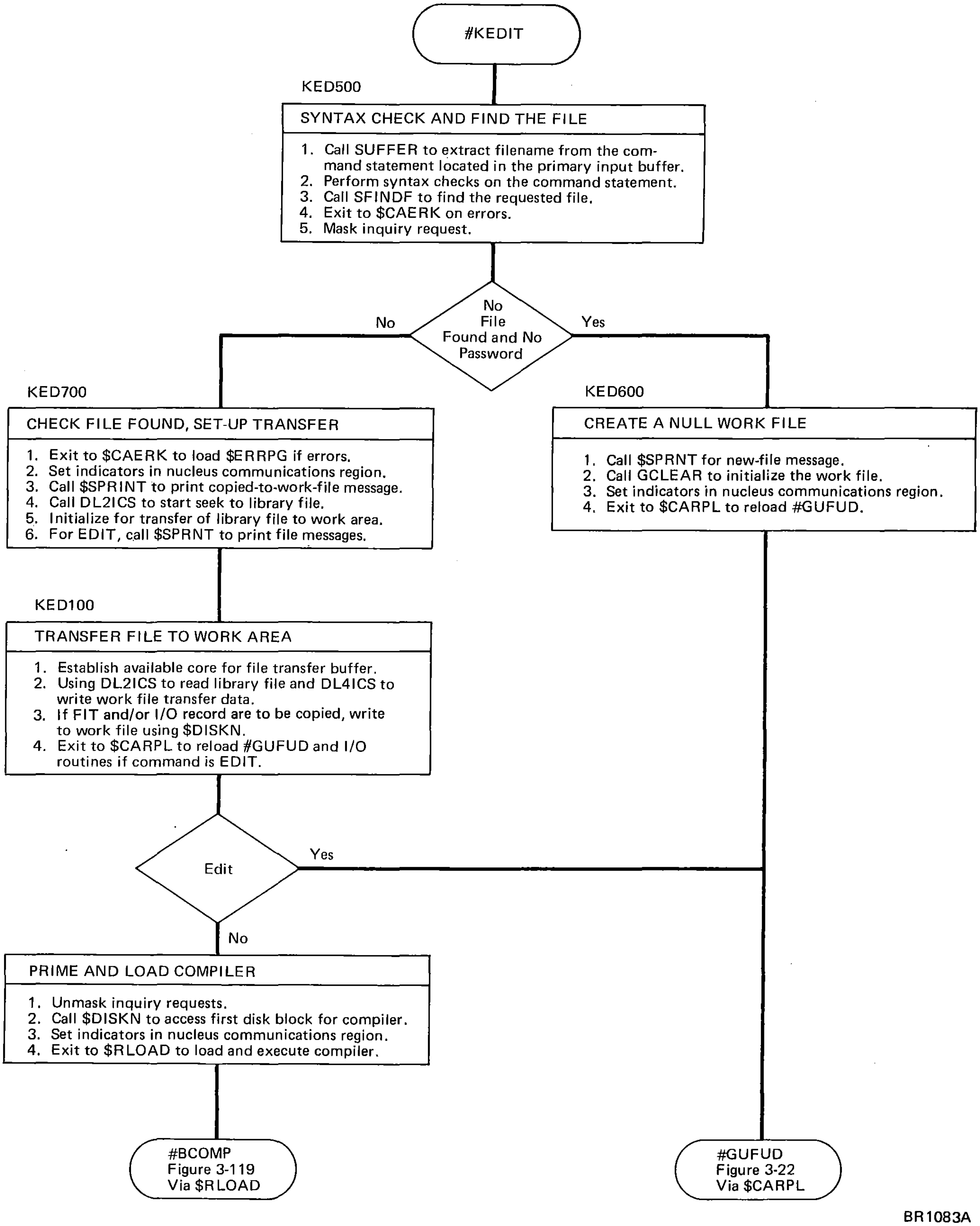


Figure 3-40. EDIT Keyword Program (#KEDIT) Flowchart

ENABLE/DISABLE Keyword Program—#KENAB (Figure 3-41)

- #KENAB modifies the type code of statements in the work file.
- The assembly of #KENAB contains these major source modules:

  KENABL—Mainline logic, Figure 3-41
  GRABIT—Work file input, Figure 3-74
  GFINDN—Locate work file disk block, no flowchart
  DL4ICS—System work file IOCS, Figure 3-70

If the DISABLE command is issued, KENABL modifies the type code of each statement in the line number list so that it is flagged and ignored in future compilations or input operations.

If the ENABLE command is issued, the type code of each statement in the line number list is modified so that previously disabled statements are again enabled for compilation or input. If the line number list is omitted from the ENABLE command, all previously disabled statements currently in the work area are enabled.

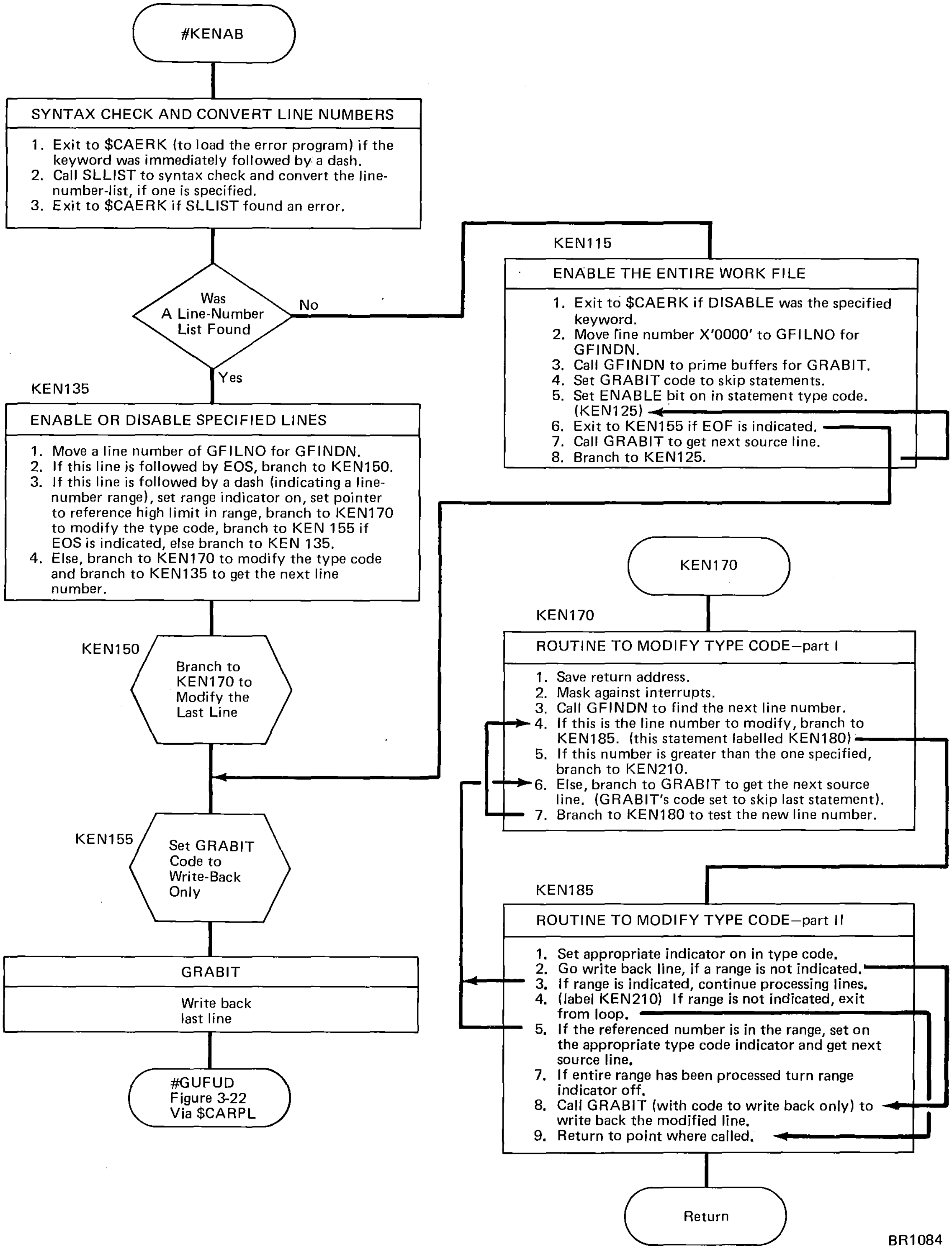


Figure 3-41. ENABLE/DISABLE Keyword Program (#KENAB) Flowchart

ENTER Keyword Program—#KDNTE (Figure 3-42)

- #KDNTE sets the system mode of operation to disk system management program, if it is available on the system.
- The assembly of #KDNTE contains these major source modules:

KDNTER—Mainline logic, Figure 3-42
SUPDAT—Statistical error recording, no flowchart

If the disk system management program (SCP) is specified, and it shares the same volume as the current BASIC system program area, the disk system management IPL bootstrap program is loaded from cylinder 0.

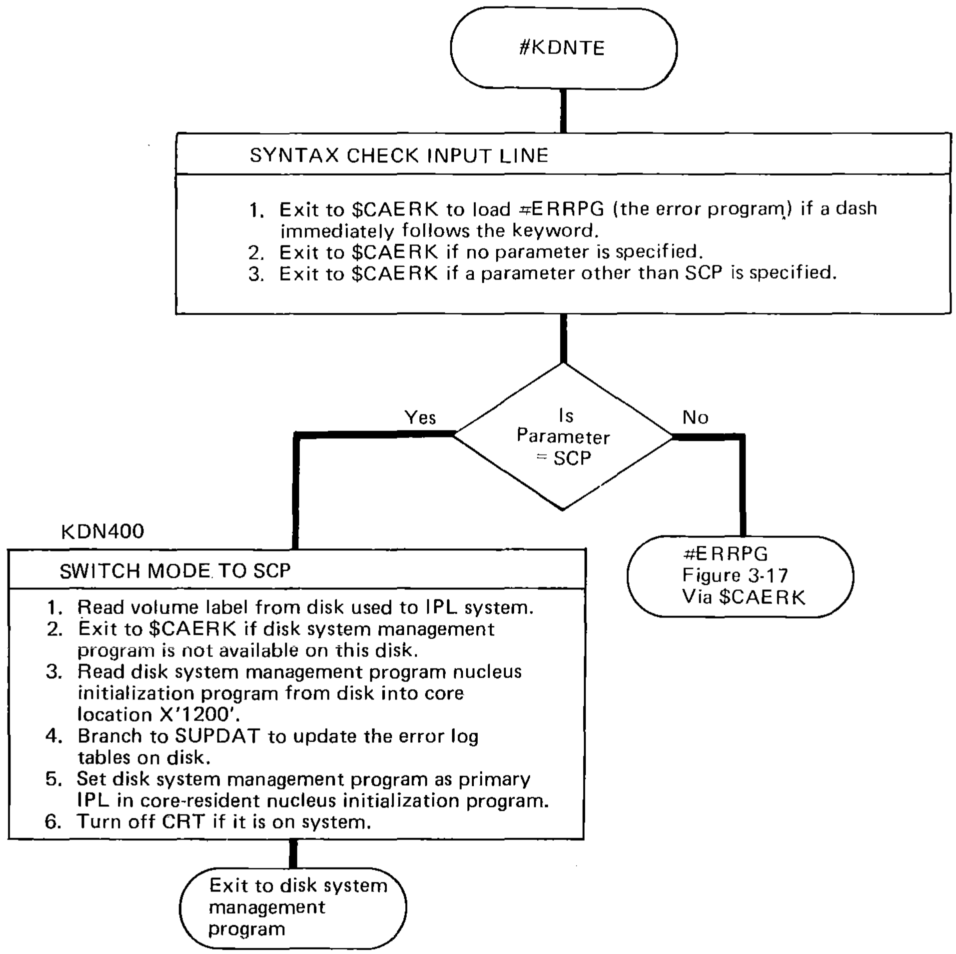


Figure 3-42. ENTER Keyword Program (#KDNTE) Flowchart

EXTRACT Keyword Program—#KEXTR (Figure 3-43)

- #KEXTR saves user specified line numbers in the work file.
- The assembly of #KEXTR contains this major source module:

KEXTRC—Mainline logic, Figure 3-43

#KEXTR retains the line number list in the active work file by deleting all unwanted line numbers. The line number list is converted to a delete parameter list (refer to Figure 5-26). The actual deletion of the lines from the work file is performed by #GUFUD, Figure 3-22.

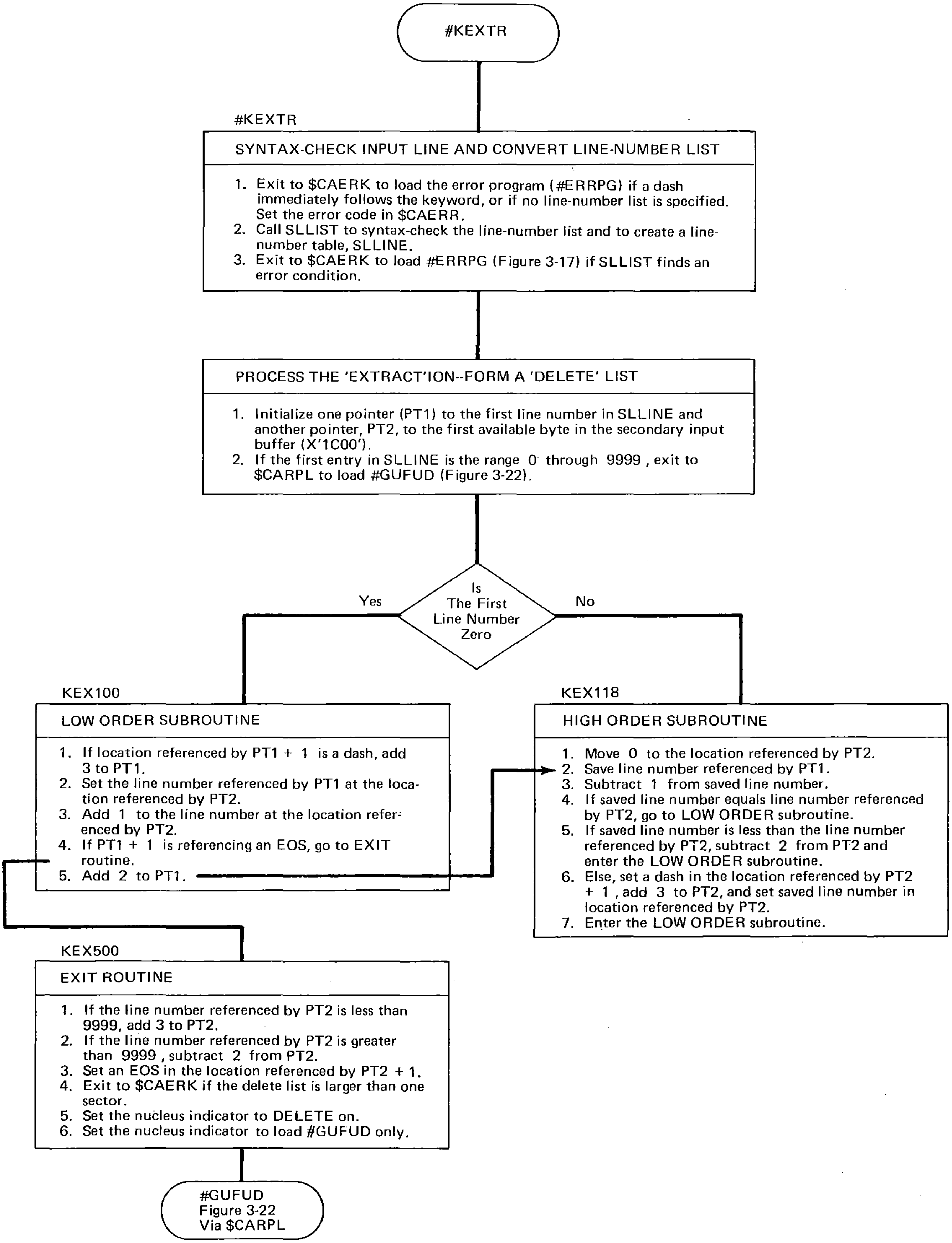


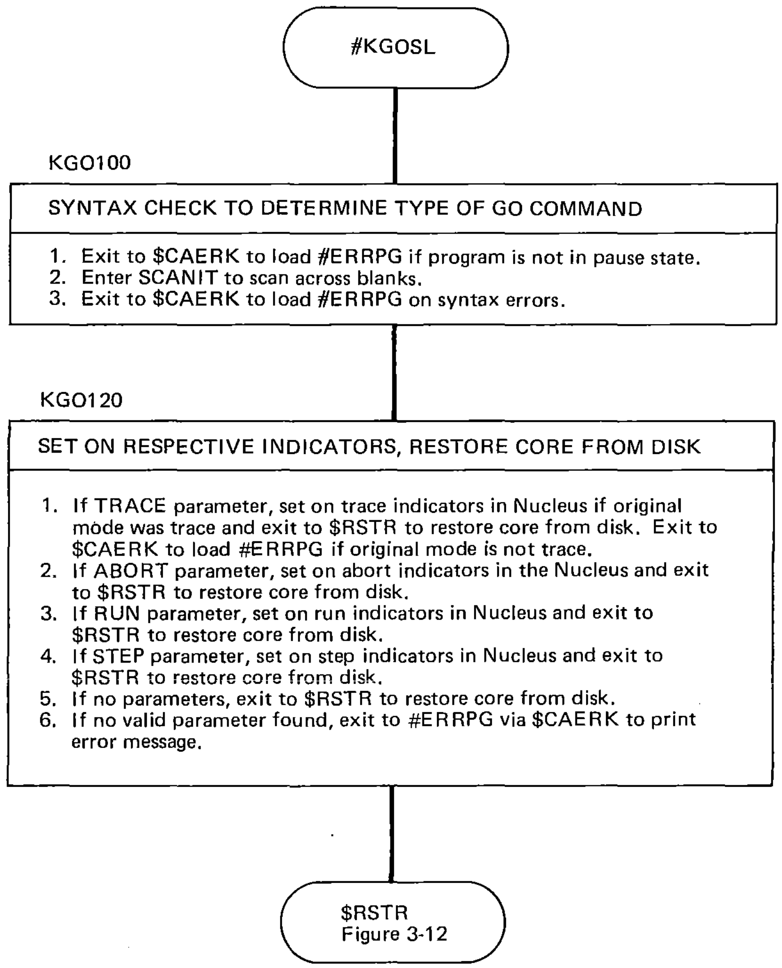
Figure 3-43. EXTRACT Keyword Program (#KEXTR) Flowchart

GO Keyword Program—#KGOSL (Figure 3-44)

- #KGOSL continues or aborts the execution of a BASIC program when the program is in an execution pause state.
- The assembly of #KGOSL contains this major source module:

  KGOSLO—Mainline logic, Figure 3-44

#KGOSL restores core from the execution save area via the restore function of the system nucleus ($PAUSD). Execution mode indicators are set in the system communication area as a result of user specified parameters.



BR1087

Figure 3-44. GO Keyword Program (#KGOSL) Flowchart

HELP Keyword Program—#KHELP (Figure 3-45)

- #KHELP displays text from the help text disk file on the matrix printer, CRT, or system printer.
- Checks release level of help text file against a built-in constant defining the expected release level.
- The assembly of #KHELP contains these major source modules:

  KHELPN—Mainline logic, Figure 3-45
  DLPRNT—IOCS for output, Figure 3-71
  GRABIT—Work file input, Figure 3-74
  DL2ICS—Disk logical IOCS, Figure 3-70

#KHELP searches a table of keywords (refer to Figure 5-21) which contains entries made up of (1) the length of a keyword, (2) a keyword, and (3) a relative address in the help text disk file. This address points to the text to be displayed for the corresponding keyword.

When there is no keyword parameter, a predetermined section of text is displayed and a choice of responses for further information is shown. Input is enabled and the input character is used to index the relative addresses in the EOF record. All nonterminating help files are handled in this manner. Help files are displayed until a terminal file is encountered via #KHELP or until the function is interrupted via an inquiry request.

Refer to Figure 5-21 for the format of help text.

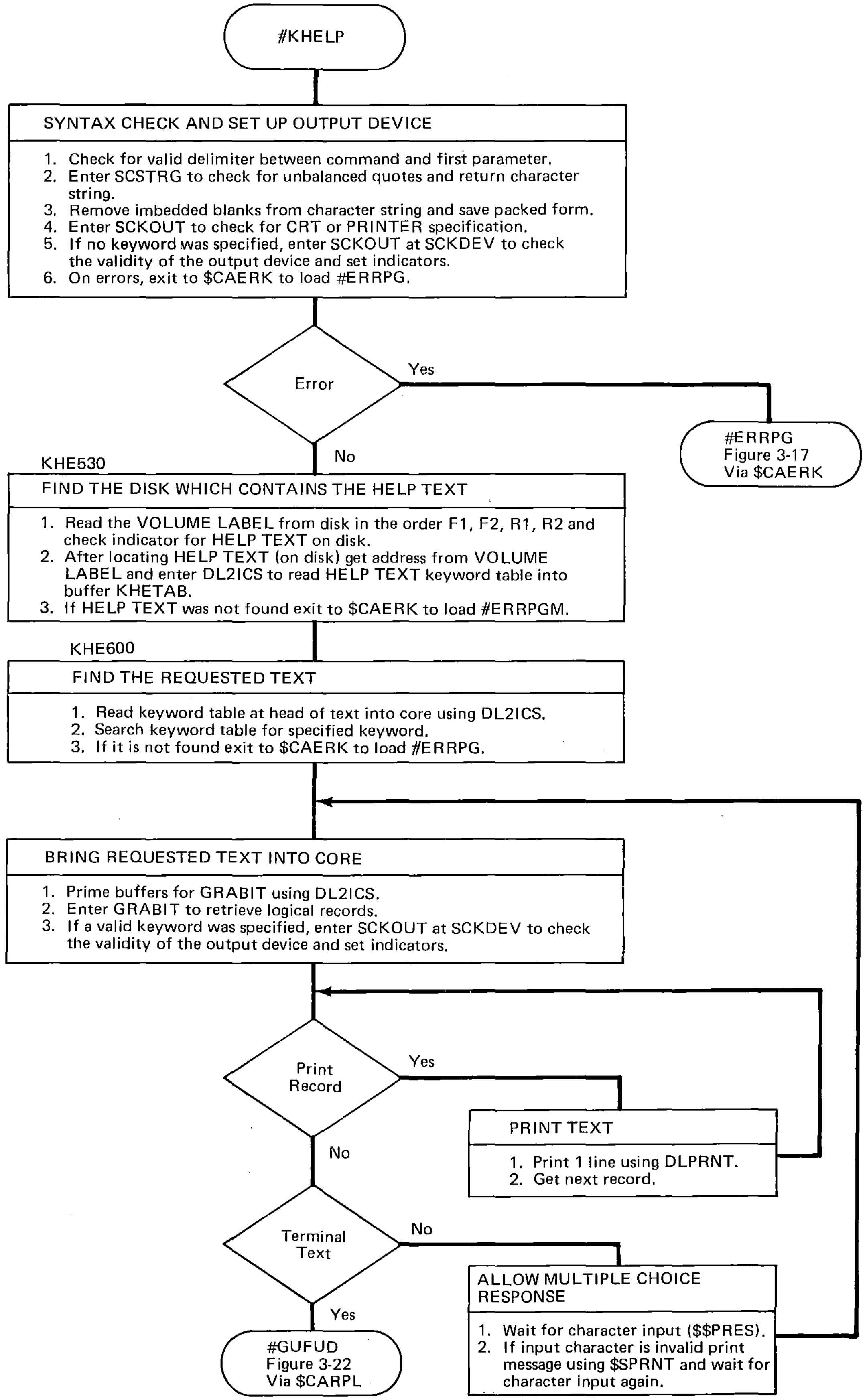


Figure 3-45. HELP Keyword Program (#KHELP) Flowchart

KEYS Keyword Program—#KKEYS (Figure 3-46)

- #KKEYS lists, assigns, or restores the functions of the command keys.
- The assembly of #KKEYS contains this major source module:

  KKEYSP--Mainline logic, Figure 3-46

Depending on the parameters specified, #KKEYS lists the functions currently assigned to the available command keys, assigns a function to an available command key, or restores one or all of the IBM-assigned functions to the available command keys. A command key is available if it is one of the first 11 command keys and is defined in the current configuration record.

The format of the command key table (##CKTB) is shown in Figure 5-27. This table resides in the system program file. A list of the IBM-assigned functions for command keys 1 through 11 is contained in Figure 5-28.

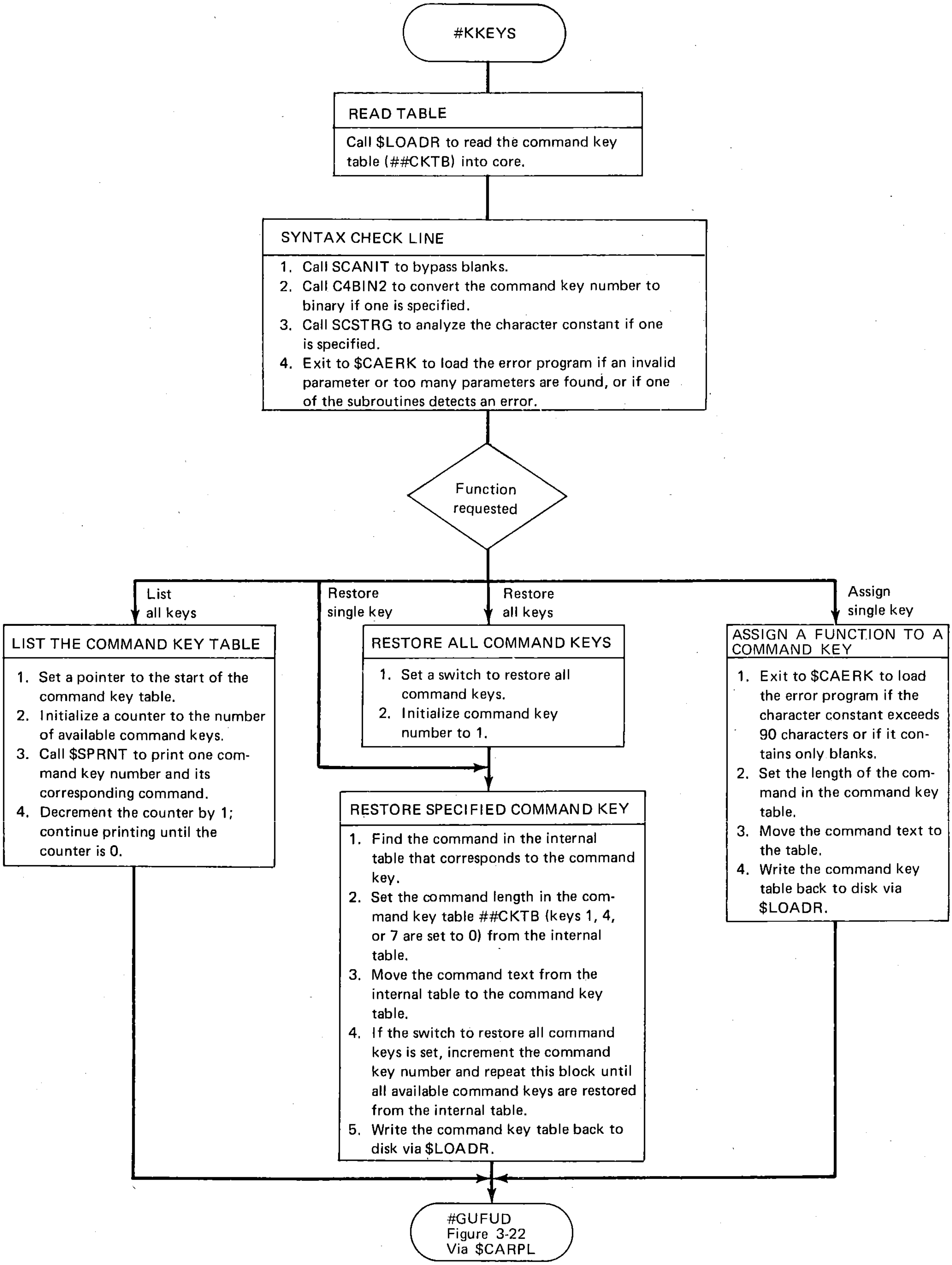


Figure 3-46. KEYS Keyword Program (#KKEYS) Flowchart

LIST Keyword Program—#KLIST, #KLLAY (Figure 3-48)

- #KLIST displays any type of file contained in the file library (system library file).
- The assembly of #KLIST contains these major source modules:

  KLISTN—Mainline logic, Figure 3-48
  GFINDN—Locate work file disk block, no flowchart
  GRABIT—Work file input, Figure 3-74
  SDLIST—List data file, no flowchart
  DLPRNT—IOCS for output, Figure 3-71
  DL4ICS—System work file IOCS, Figure 3-70

- The assembly of the overlay #KLLAY contains this major source module:

  DCDOUT—Card punch IOCR, Figure 3-72

#KLIST displays BASIC programs, keyboard-generated files, procedure files, or program-generated files on the printer, punch, or CRT.

If the file is listed on the CRT, the user may rollup, rolldown, or popup the file (this does not apply to program-generated files). The user may also specify line lists for starting and ending the LIST function (for CRT, only the initial line reference is used).

A list control block (LCB) (Figure 3-47), 20 bytes in length and containing all information necessary to control the output, is created from the parameters of the LIST statement. When the output device is the CRT, a CRT line segment table (Figure 3-47) is maintained from elements in the LCB. If the work file contains a program-generated file, logical lines are constructed and sent sequentially to the specified devices until end of file is encountered. A logical line is device dependent; for example:

Device	*Line Length*
5496 Data recorder	96
Matrix printer	(Right margin—left margin)
CRT	64
System printer	64

The rolldown key (command key 14) is not recognized for program-generated files.

If the work file contains a BASIC program, the file is sent to the specified devices under control of the line number list. When CRT is specified, only the initial line reference is used. However, the user can rollup, popup, or rolldown the file. Initially, the first 14 lines are placed on the screen. From this point, the file may be rolled as desired; interruptions are accepted after each line segment is displayed. If end of file on rollup or beginning of file on rolldown is encountered, the program waits for an interruption. The inquiry request key must be activated to terminate the listed function.

When the work file contains a keyboard-generated file, the data elements are converted from internal floating point to an optimum external format. Each line is then handled the same as a BASIC program line. Disabled BASIC statements and lines are shown with * preceding the line number.

No line number is punched if the NO-NUM parameter is specified, the output specified is CARD, and the work file contains a keyboard-generated file.

When the parameter CARD is specified, #KLLAY is read into the I/O portion of core. #KLLAY contains the card punch IOCR (DCDOUT) and overlays the card reader I/O routine #DREAD.

Hexadecimal Displacement	Name	Length (bytes)	Explanation
00	File condition code	1	Status indicator value: X'00'—Go X'01'—Line list exhausted X'02'—Beginning of file X'03'—End of file X'04'—No line list
01	Start line	2	Beginning line number of a loop.
03	Increment	2	+1 (X'0001'), except for rolldown, then −1 (X'FFFF')
05	Control character	1	X'4F'—Rolldown X'C0'—Rollup or print
06	File line length	1	Length of current work file line.
07	Buffer address	2	Current address, into line buffer area, that is used in PPL passed to DLPRNT.
09	CRT mode	1	Rolldown—X'02' Rollup—X'01'
0A	Current line	2	Current line number being analyzed.
0C	CRT line segments out	1	Count of number of CRT line segments displayed from current line.
0D	Maximum CRT line segments	1	Count of number of CRT line segments in file line being processed (length/64 + 1).
0E	CRT mode change	1	Indicates a change from rollup or popup to rolldown and the reverse.
0F	First line number	2	First line number in work file. Detect beginning of file when file is in rolldown mode.
11	Initial call indicator	1	X'01'—First time X'00'—Not first time
12	Stop line	2	Ending line number of a line loop.

CLST—One 5-Byte Entry per CRT Line (70 bytes total)													
1	2	3	4	5	6	7	8	9	10	11	12	13	14

CRT Line Segment Entry				
Mode	Current Line		CRT Line Segments Out	Maximum CRT Line Segments
0	1	2	3	4

Mode-From hexadecimal displacement 09 in LCB
Current Line-From hexadecimal displacement 0A in LCB
CRT Line Segments Out-From hexadecimal displacement 0C in LCB
Max CRT Line Segments-From hexadecimal displacement 0D in LCB

BR2671

Figure 3-47. List Control Block (LCB) and CRT Line Segment Table (CLST)

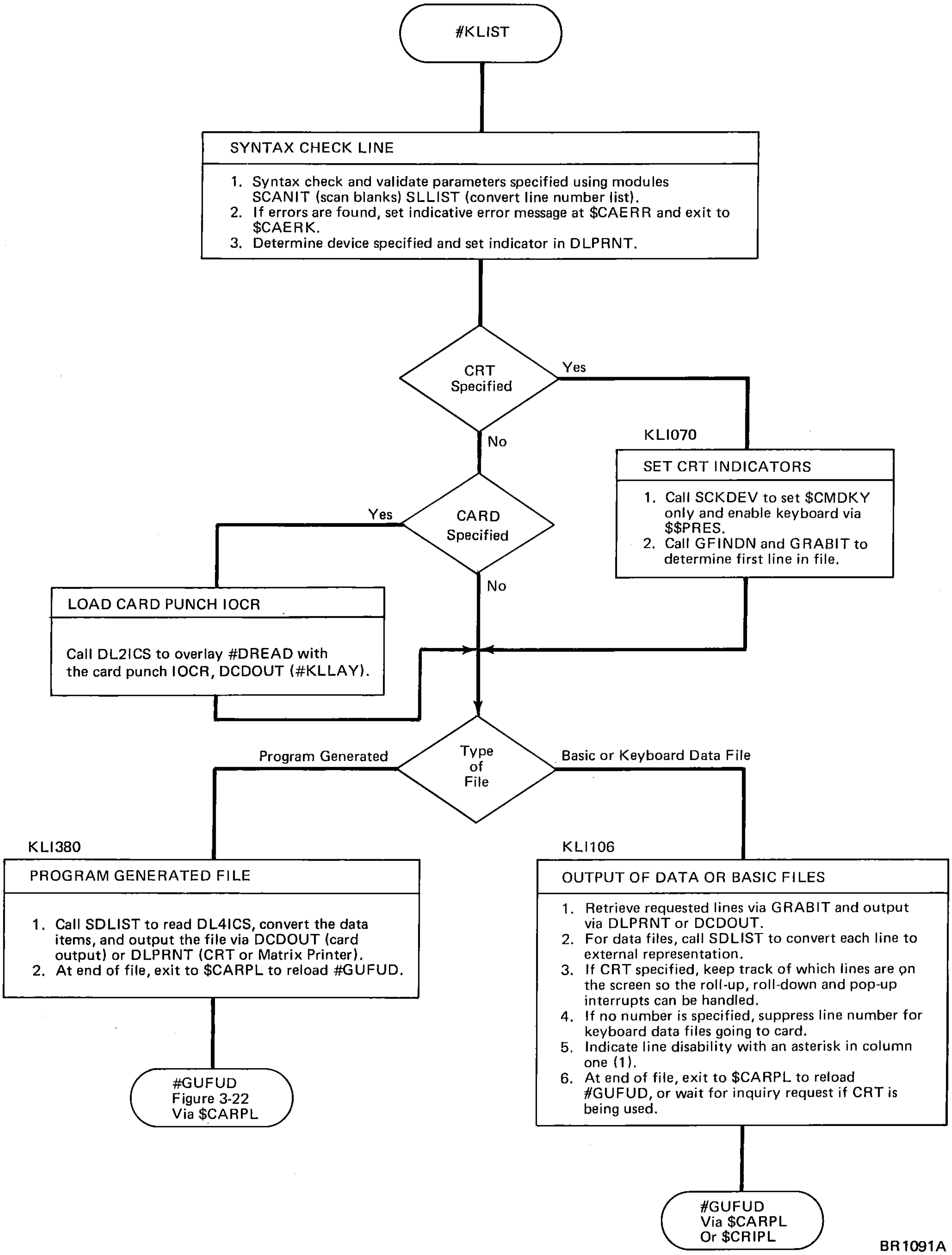


Figure 3-48. LIST Keyword Program (#KLIST) Flowchart

LISTCAT Keyword Program—#KCTLO (Figure 3-49)

- #KCTLO displays user-specified directory information from the file library (system library file).
- The assembly of #KCTLO contains these major source modules:

  KCTLOG—Mainline logic, Figure 3-49
  DL2ICS—Disk logical IOCS, Figure 3-70
  DLPRNT—IOCS for output, Figure 3-71
  SCKOUT—Check output specification, no flowchart
  DSVPRI—DLPRNT interface, no flowchart

#KCTLO displays the following directory information on the matrix printer, CRT, or system printer, if ALL is specified (otherwise, only the filename and file ID are displayed):

1. Filename—File ID
2. File type
3. Date the file was last modified
4. Count of lines contained in the file
5. Count of sectors the file occupies
6. Precision of the file
7. Pooled status
8. File protection status
9. Open/close status

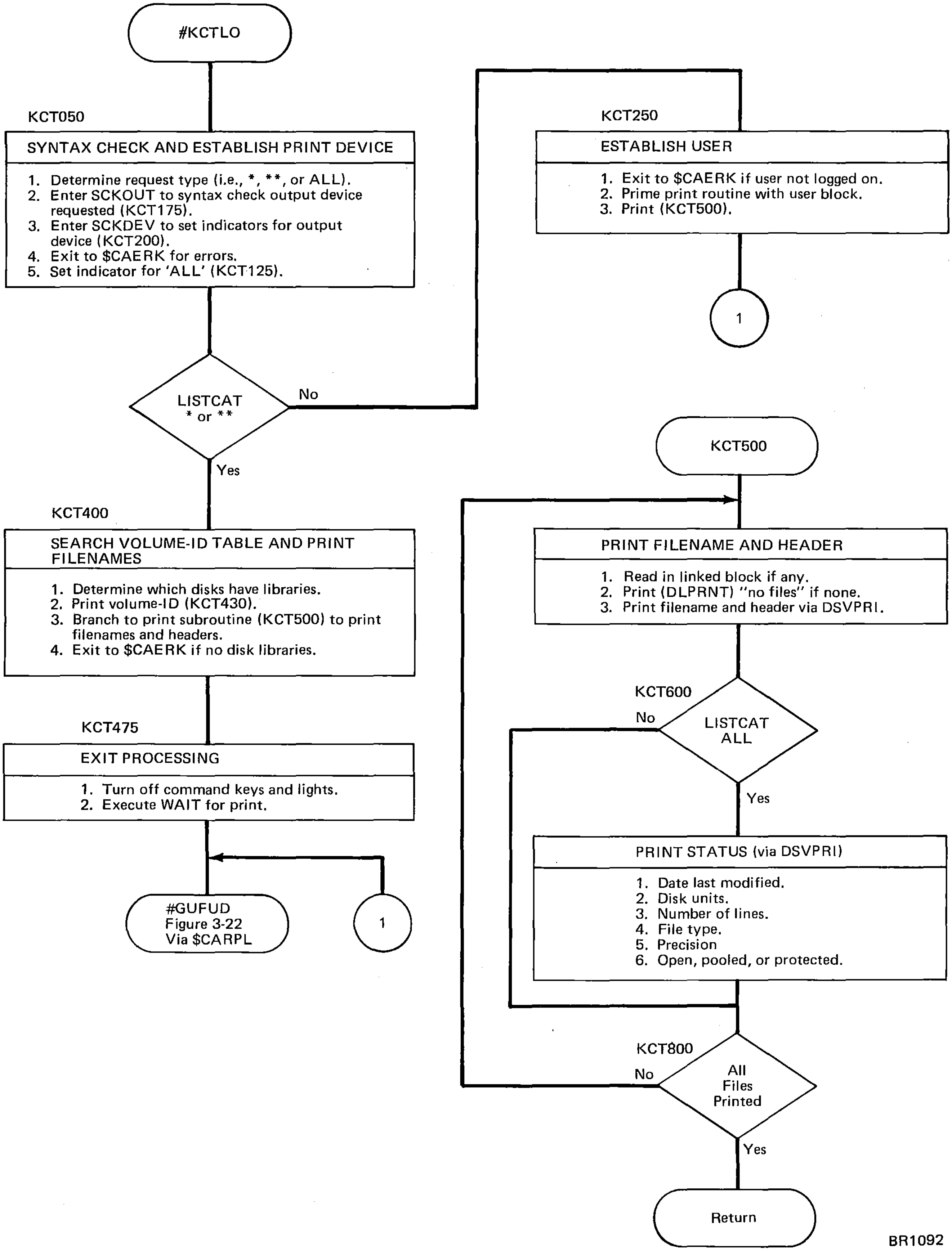


Figure 3-49. LISTCAT Keyword Program (#KCTLO) Flowchart

LOGON/OFF Keyword Program—#KLOGO (Figure 3-50)

- #KLOGO is used to define a password and volume (new or old) to be used for subsequent operations. OFF is used to cancel the current password and volume.
- The assembly of #KLOGO contains these major source modules:

  KLOGON--Mainline logic, Figure 3-50
  SVOLID--Search volume-ID table, Figure 3-76
  SGETDB--Search password directory, Figure 3-77
  DL2ICS—Disk logical IOCS, Figure 3-70
  SURCHN--Search null directory, Figure 3-81
  SUPDAT—Statistical error recording, no flowchart

#KLOGO clears the saved bad line area (used by CHANGE), deletes the file in the system work area, and updates the statistical data recorder on the logged-on volume. It also clears the CRT if it is configured.

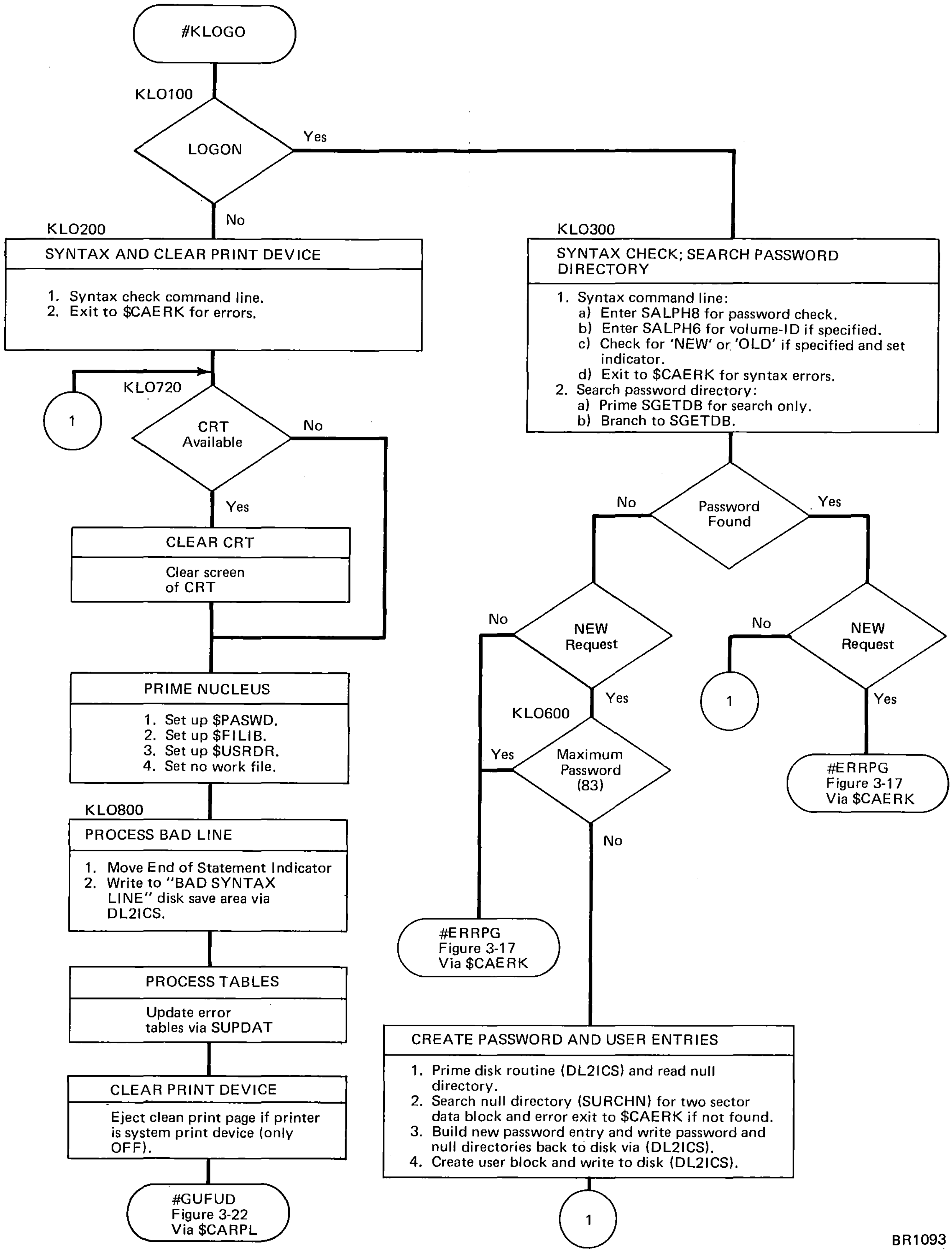


Figure 3-50. LOGON/OFF Keyword Program (#KLOGO) Flowchart

MERGE Keyword Program—#KMERG (Figure 3-51)

- #KMERG merges statements from a library file with the file in the system work file.
- The assembly of #KMERG contains these major source modules:

  KMERGE—Mainline logic, Figure 3-51
  GRABIT—Work file input, Figure 3-74
  GPUTIT—Work file output, Figure 3-73
  SVOLID—Search volume-ID table, Figure 3-76
  SGETDB—Search password directory, Figure 3-77
  SRCHFN—Search user directory, Figure 3-78
  SFINDF—Find library file, Figure 3-75
  DL2ICS—Disk logical IOCS, Figure 3-70
  DL4ICS—System work file IOCS, Figure 3-70

Functions of #KMERG are:

1. Merge statements from a saved library file with the active file in the system work file.
2. Write the merged file temporarily in virtual memory and build a line number table in core.
3. Load and exit to #KOVME, Figure 3-61, to renumber the merged file and write it back to the system work file.

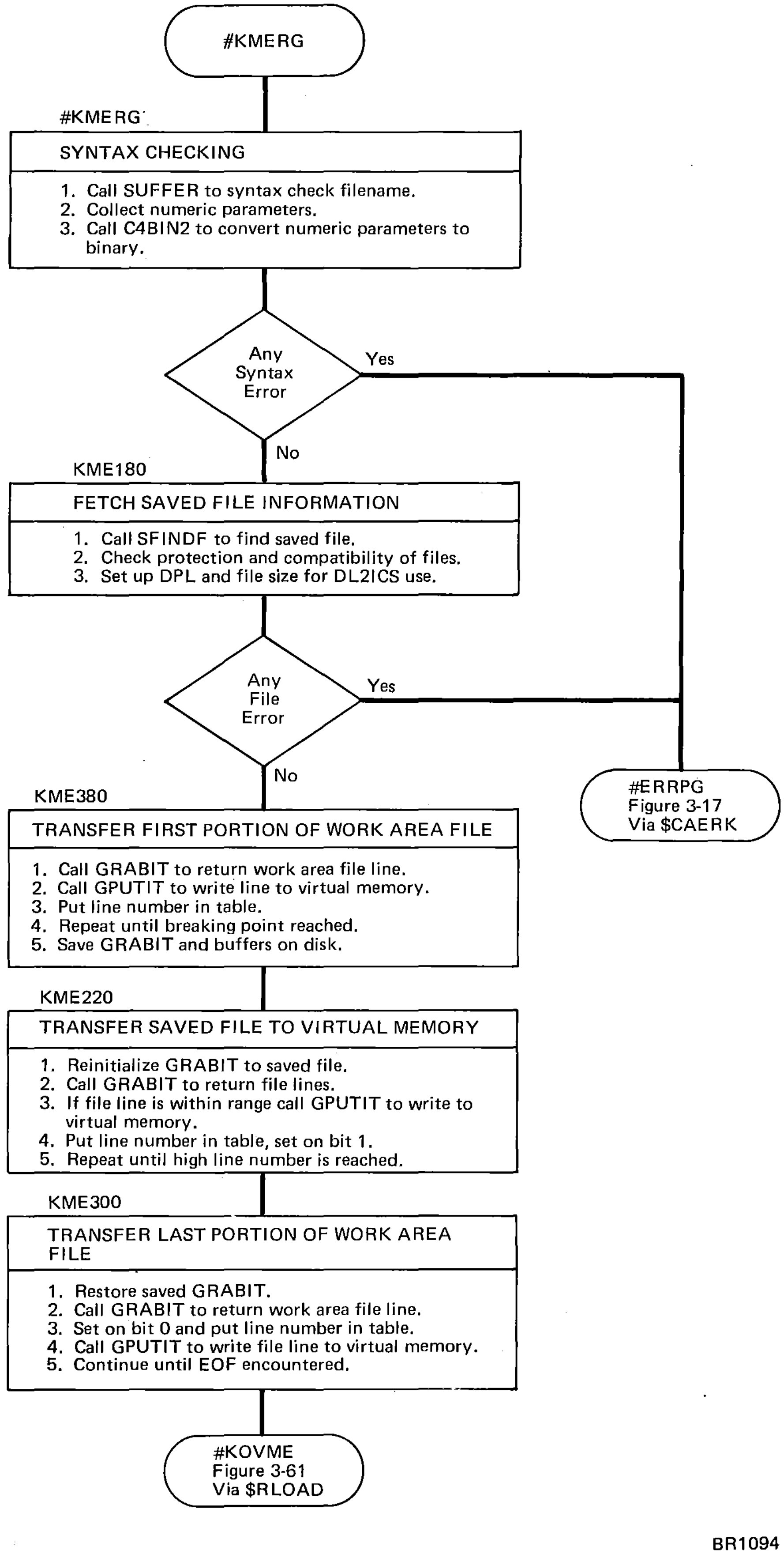


Figure 3-51. MERGE Keyword Program (#KMERG) Flowchart

MOUNT Keyword Program—#KMOUN (Figure 3-52)

- #KMOUN notifies the system that a different volume is mounted on R1 or R2.
- The assembly of #KMOUN contains these major source modules:

  KMOUNT—Mainline logic, Figure 3-52
  SUTOBA—Switch system mode, no flowchart
  MINITL—Read the disk label if the disk has been initialized, no flowchart

#KMOUN reads the volume label to verify that the mounted volume-ID matches the user specified volume-ID. If the drive is R1, and if R1 and F1 both contained valid system work areas before R1 was removed, the volume label of the disk specified in the MOUNT command must also contain a valid system work area; otherwise, an error message results. The volume-ID table in the nucleus communications area (refer to Figure 5-9) is updated. If any scratch file entries exist in the VTOC on the pack being mounted, #KMOUN deletes them.

The disk drive must be ready before #KMOUN can read the volume label.

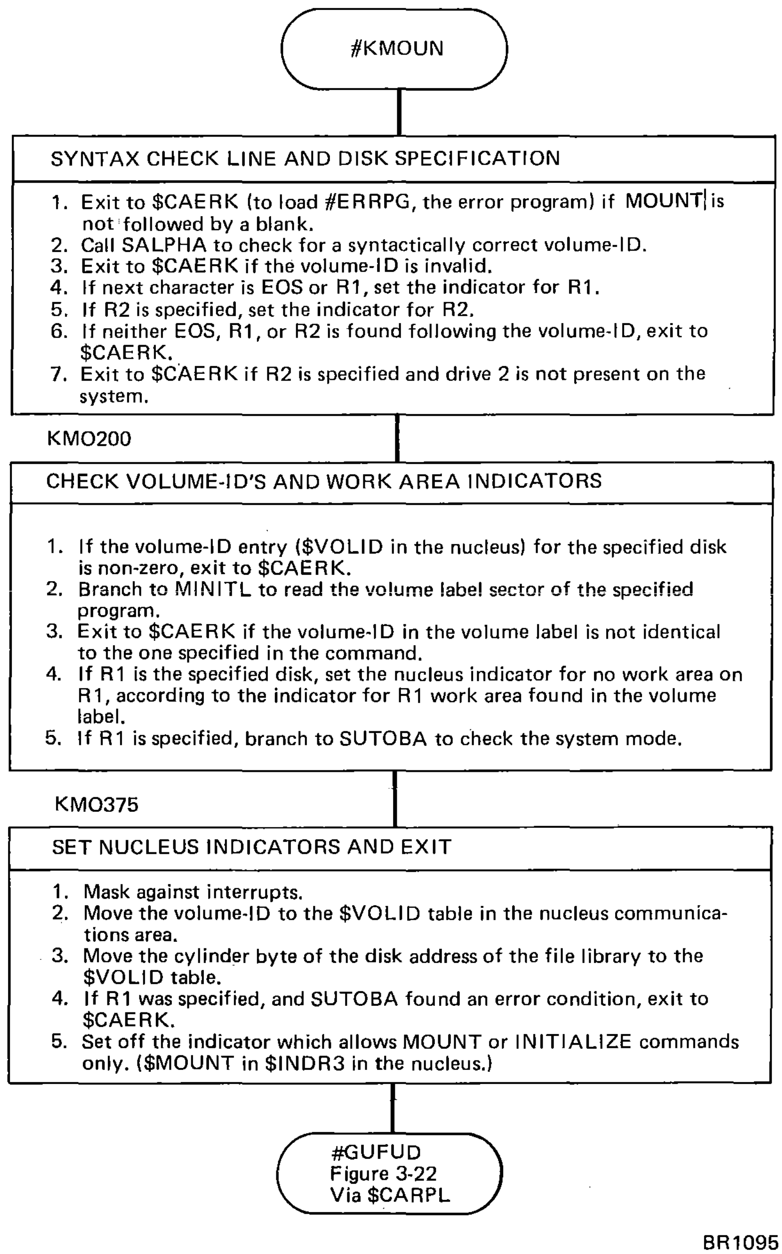


Figure 3-52. MOUNT Keyword Program (#KMOUN) Flowchart

PASSWORD Keyword Program—#KPASW (Figure 3-53)

- #KPASW changes the current password and the password directory to the password specified by the PASSWORD command.
- The assembly of #KPASW contains these major source modules:

  KPASWD—Mainline logic, Figure 3-53
  DL2ICS—Disk logical IOCS, Figure 3-70
  SGETDB—Search password directory, Figure 3-77

#KPASW searches the password directory, checking to see that the new password is not a duplicate, before updating and writing back the directory. The specified password replaces the current password entry in the password directory.

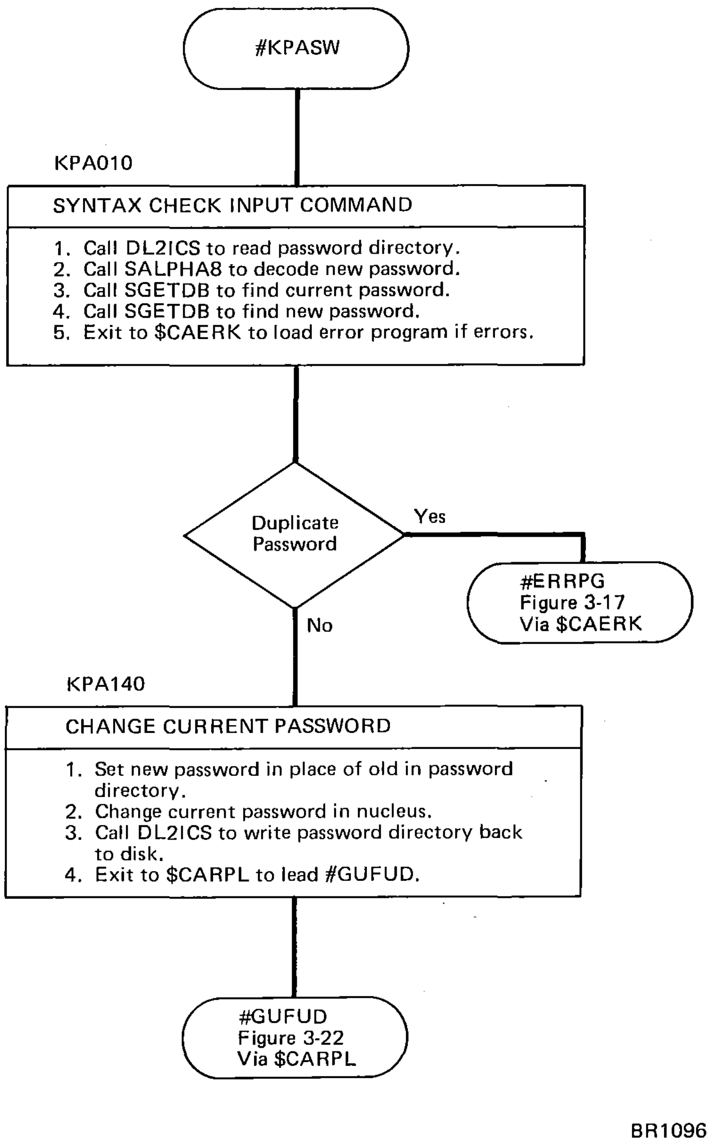


Figure 3-53. PASSWORD Keyword Program (#KPASW) Flowchart

PROTECT Keyword Program—#KPRTC (Figure 3-54)

- #KPRTC sets or cancels file protection on user library files or pooled files.
- One-star and two-star data files cannot be protected.
- The assembly of #KPRTC contains these major source modules:

  KPRTCT–Mainline logic, Figure 3-54
  DL2ICS–Disk logical IOCS, Figure 3-70
  SRCHFN–Search user directory, Figure 3-78
  SVOLID–Search volume-ID table, Figure 3-76
  SGETDB–Search password directory, Figure 3-77
  SFINDF–Find library file, Figure 3-55

#KPRTC sets the protect status bit in the user, pooled, or two-star filename directory. The selection of the status is determined by the user with the ON or OFF parameter, with ON being the default condition.

If a pooled filename is specified, a current user must be logged on and the filename must be in his user directory to qualify him as the creating user. If the name is in his directory, the protection of the entry in the pooled directory is changed (protected or unprotected).

When a file specification is entered, #KPRTC searches the password directory, and then the user directory, for the filename. The status bit is set if a match is made. When a two-star filename is specified, the protect status can only be set ON.

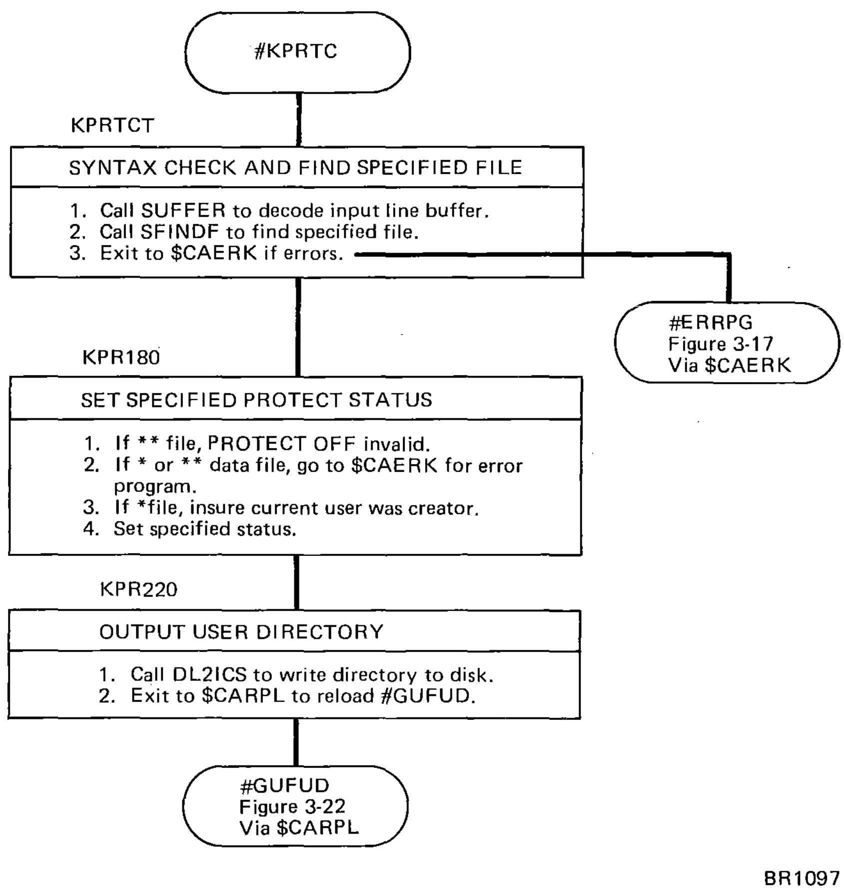


Figure 3-54. PROTECT Keyword Program (#KPRTC) Flowchart

PULL/POOL Keyword Program—#KPOOL (Figure 3-55)

- #KPOOL adds or removes a specified filename to or from the one-star library directory.
- The assembly of #KPOOL contains these major source modules:

  KPOOLN—Mainline logic, Figure 3-55
  SRCHFN—Search user directory, Figure 3-78
  SVOLID—Search volume-ID table, Figure 3-76
  SGETDB—Search password directory, Figure 3-77
  SFINDF—Find library file, Figure 3-75
  DL2ICS—Disk logical IOCS, Figure 3-70
  STORIN—Null directory insert, no flowchart
  STUFID—User directory insert, no flowchart
  SURCHN—Search null directory, Figure 3-81

#KPOOL loads #SPACK, Figure 3-86, to pack the file library if the null directory is full. #SPACK loads, and returns to, #KPOOL.

If the POOL keyword command is issued, KPOOLN inserts the specified user file into the one-star on the disk containing the user library, thus making it available to all of the system (i.e., an entry for the file is created in the one-star library and an indicator is set in the user library, allowing the "pooled" use of the file).

If the PULL keyword command is issued, KPOOLN removes the specified user file from the one-star library on the disk containing the user library (i.e., the one-star library entry for the file is deleted and the "pooled" indicator for the file is set off).

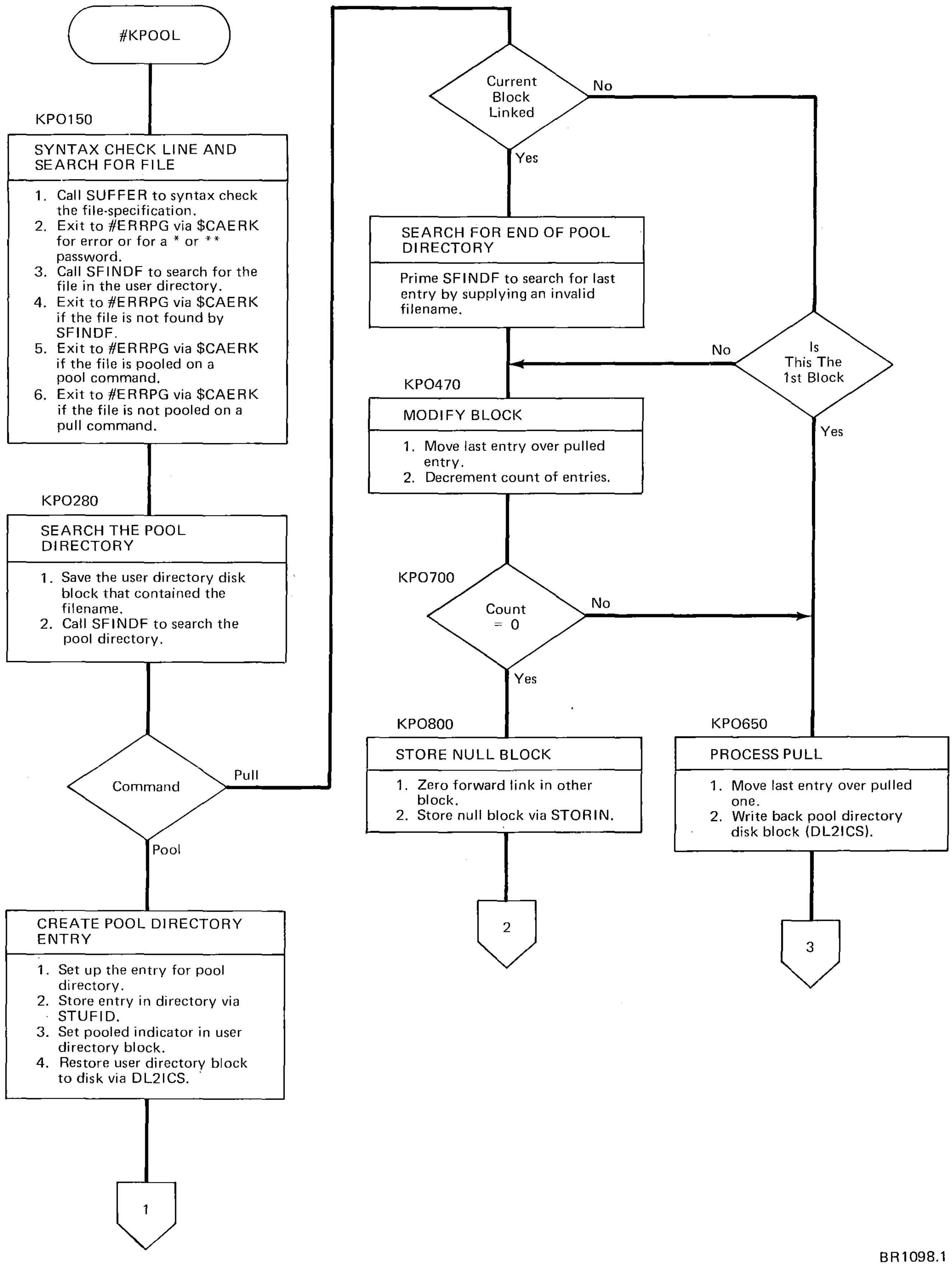


Figure 3-55. PULL/POOL Keyword Program (#KPOOL) Flowchart (Part 1 of 2)

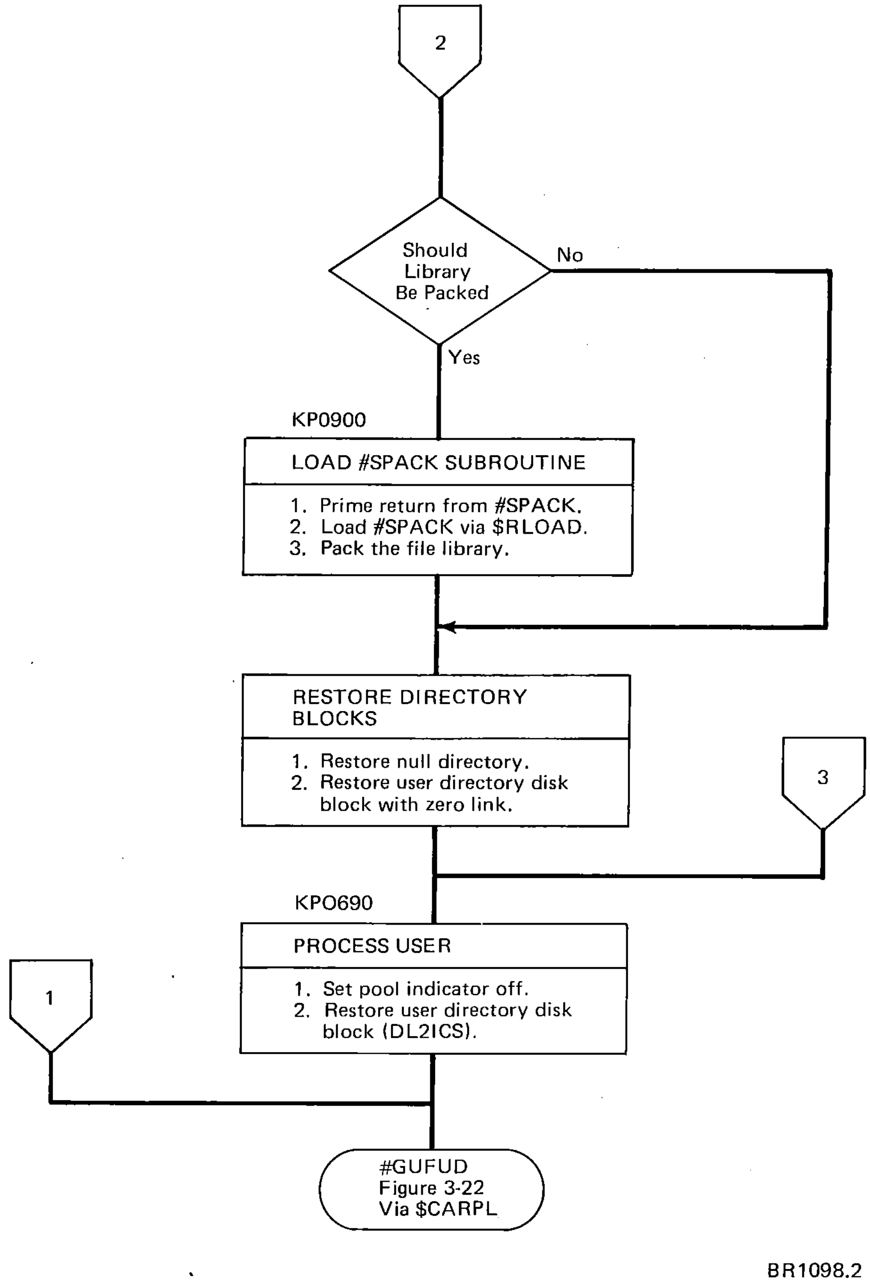


Figure 3-55. PULL/POOL Keyword Program (#KPOOL) Flowchart (Part 2 of 2)

READ Keyword Program—#KREAD (Figure 3-56)

- #KREAD changes the system input device to keyboard or card reader.
- The assembly of #KREAD contains this major source module:

  KREADN—Mainline logic, Figure 3-56

#KREAD sets input mode indicators in the nucleus communications area (refer to Figure 5-1) corresponding to parameters of the READ statement. #KREAD exits to the system nucleus which loads #GUFUD (Figure 3-22) to load the proper I/O routines.

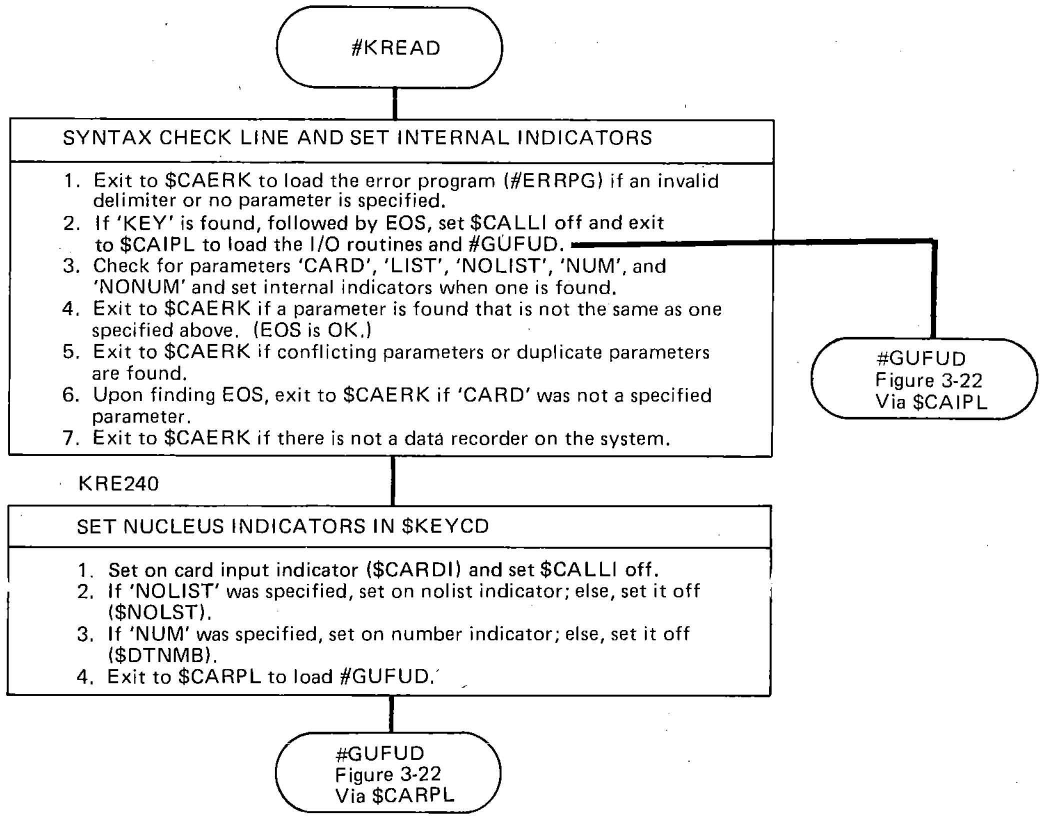


Figure 3-56. READ Keyword Program (#KREAD) Flowchart

RELABEL Keyword Program—#KRLAB (Figure 3-57)

- #KRLAB changes variable names in the system work area program according to user specified parameters.
- The assembly of #KRLAB contains these major source modules:

  KRLABL—Mainline logic, Figure 3-57
  GRABIT—Work file input, Figure 3-74
  GPUTIT—Work file output, Figure 3-73
  DL4ICS—System work file IOCS, Figure 3-70
  SVARAB—Variable scan, no flowchart.

#KRLAB evaluates and determines the validity of the parameters, which must be pairs of the same class of labels. If an error occurs, the program is terminated prior to any file alterations. Every statement is scanned for the first entry of a parameter pair. If a match is found, the second entry of that parameter pair is substituted in its place. The file line length is altered only when a different length label is substituted. The command is rejected if a data file is in the work file. #KRVLA is the second phase of the RELABEL keyword program.

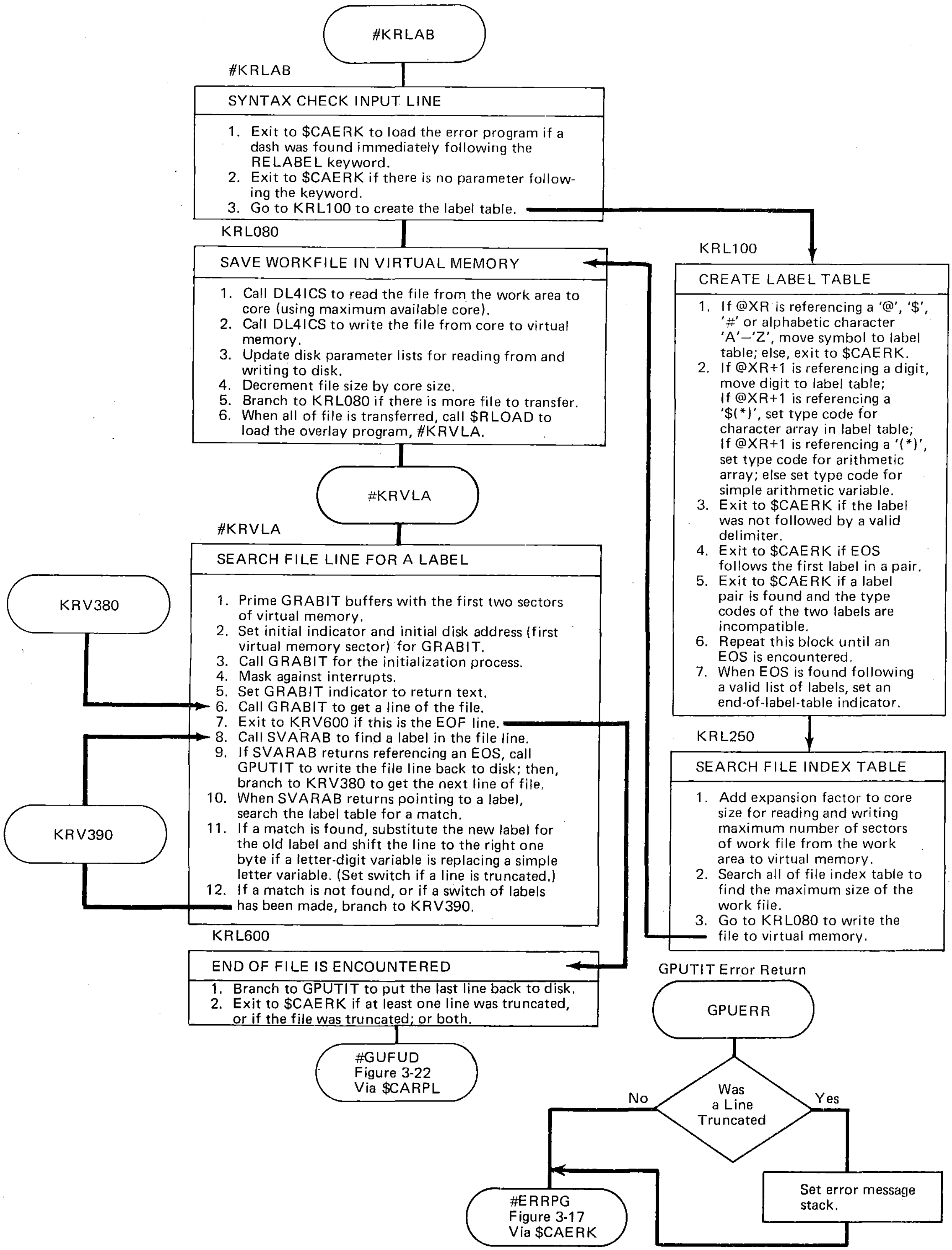


Figure 3-57. RELABEL Keyword Program (#KRLAB) Flowchart

REMOVE Keyword Program—#KRMOV (Figure 3-59)

- #KRMOV notifies the system that a volume is being removed from R1 or R2.
- The assembly of #KRMOV contains these major source modules:

  KRMOVE—Mainline logic, Figure 3-59
  SUPDAT—Statistical error recording, no flowchart
  SUTOBA—Switch system mode, no flowchart

#KRMOV reads the volume label to verify that the volume-ID matches the user specified volume-ID in the nucleus communications area (refer to Figure 5-1). If the drive is R1, an indicator is reset for no system work area available. The volume-ID table entry for R1 or R2 is reset to binary 0's (refer to Figure 5-10).

The volume must be ready until #KRMOV is terminated because the individual volume error statistics must be updated. If R1 is removed and it contains the current system program file, a hard halt is generated after the appropriate message is printed. An error also occurs and a warning error message is printed if the user is logged onto the disk he is removing.

#KRMOV is loaded by #ECMAN (Figure 3-24) at 0C00 (see the core map, Figure 3-58).

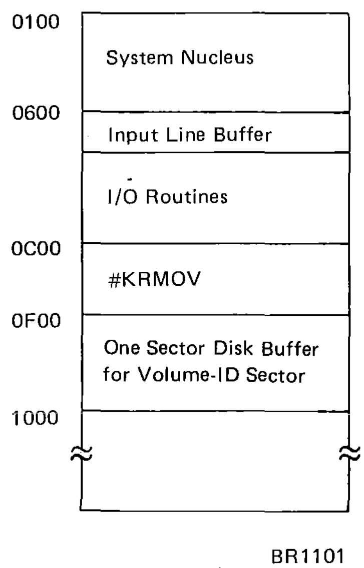
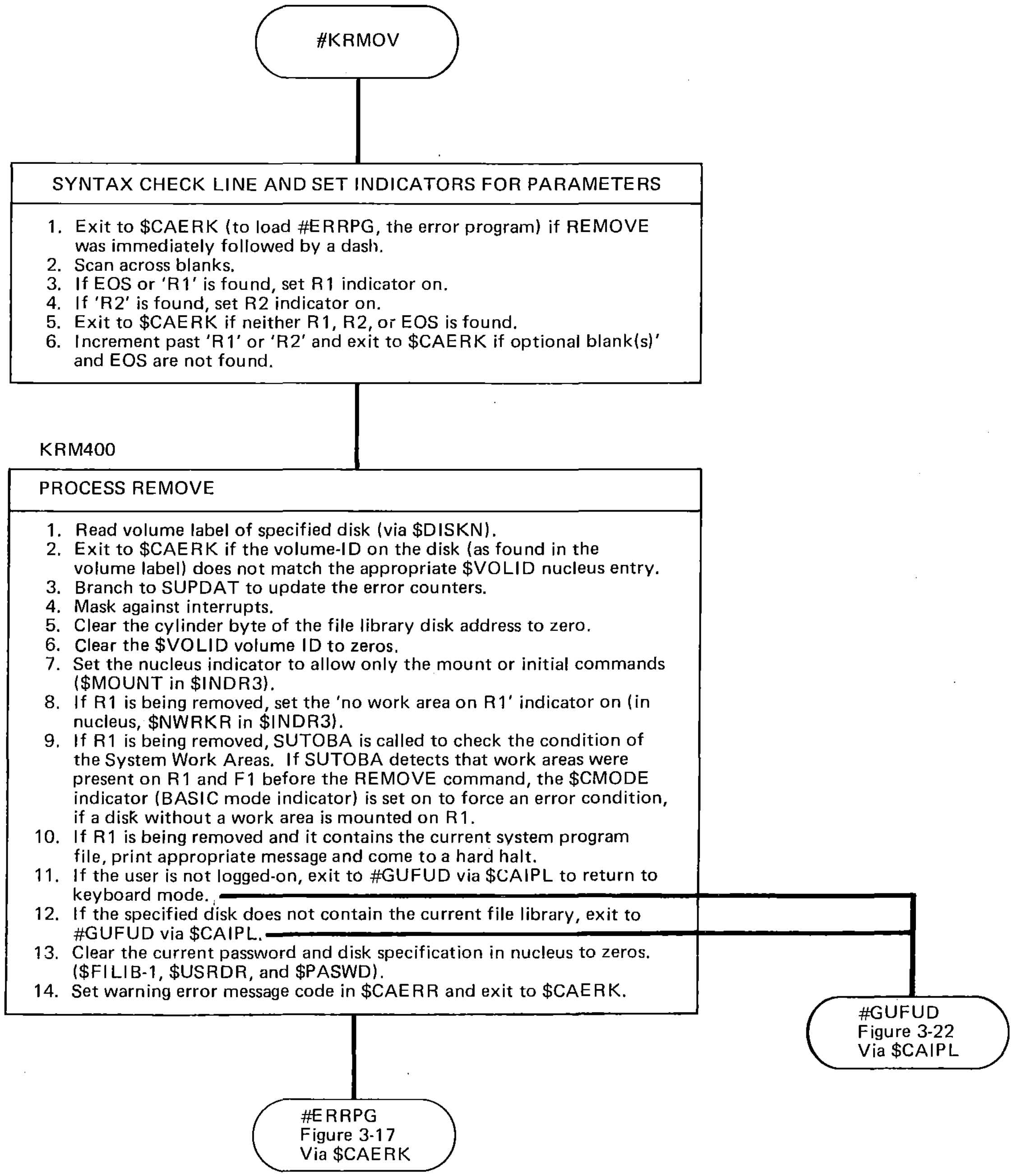


Figure 3-58. #KRMOV Core Map



BR1102

Figure 3-59. REMOVE Keyword Program (#KRMOV) Flowchart

RENAME Keyword Program—#KNAME (Figure 3-60)

- #KNAME assigns a new filename to the work file or to a file in the file library (pooled and/or user file).
- The assembly of #KNAME contains these major source modules:

  KNAMES—Mainline logic, Figure 3-60
  DL2ICS—Disk logical IOCS, Figure 3-70
  SFINDF—Find library file, Figure 3-75
  SGETDB—Search password directory, Figure 3-77
  SRCHFN—Search user directory, Figure 3-78
  SVOLID—Search volume-ID table, Figure 3-76

#KNAME assigns the user filename to the file specified by the user file specification.

The user directory and the pooled directory are searched to ensure that the new filename is not a duplicate.

If the name is valid, the entry in the directory is changed by writing back that sector of the directory.

If the user file specification is not present, the user filename is assigned to the currently active file in the system work file.

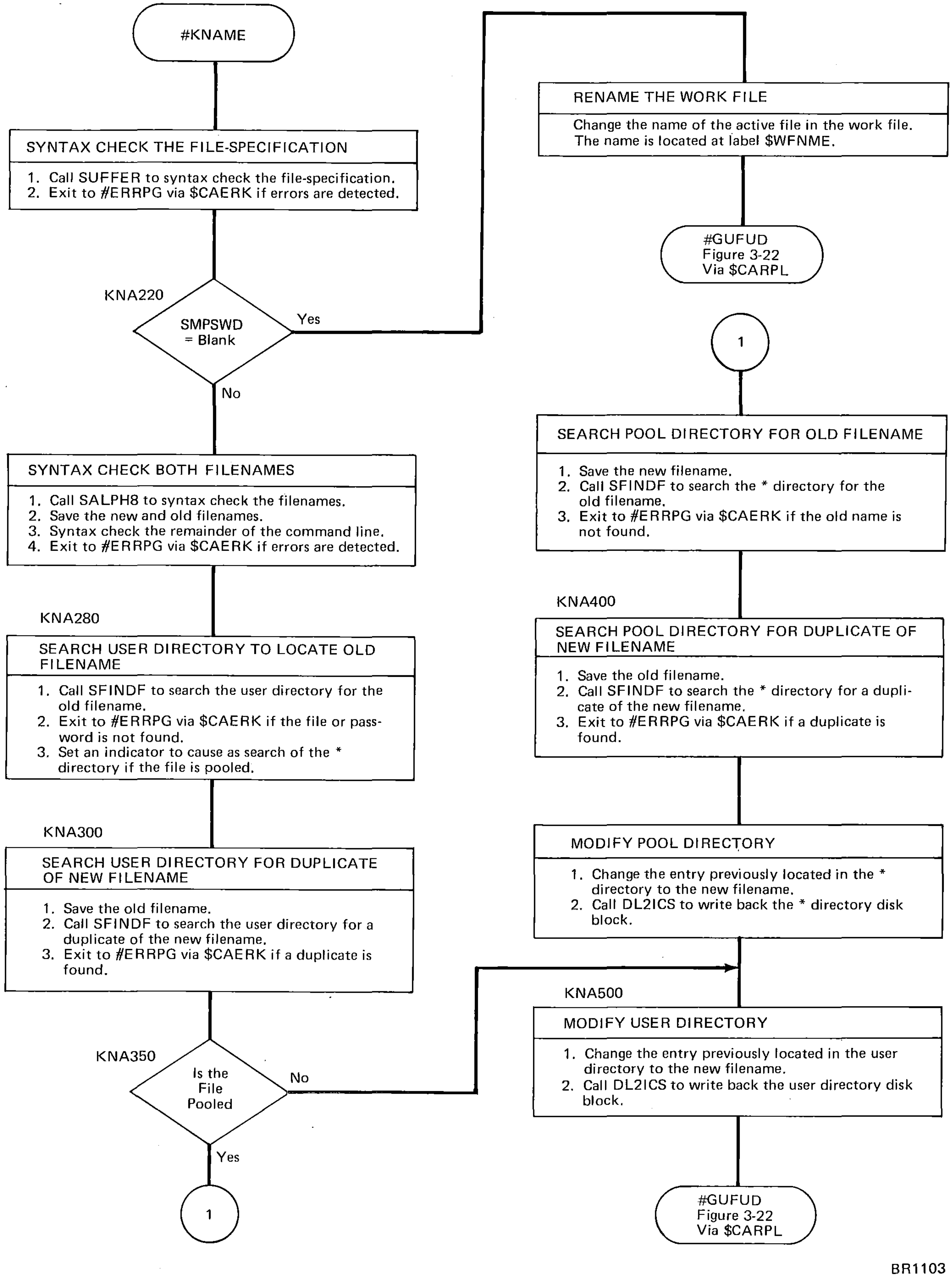


Figure 3-60. RENAME Keyword Program (#KNAME) Flowchart

RENUMBER Keyword Program—#KRNUM (Figure 3-61)

- #KRNUM renumbers the statements of the active file in the system work area.
- The assembly of #KRNUM contains these major source modules:

  KRNUMB—Mainline logic, Figure 3-61
  KROVLY—Mainline logic, Figure 3-61
  GRABIT—Work file input, Figure 3-74
  GPUTIT—Work file output, Figure 3-73
  DL4ICS—System work file IOCS, Figure 3-70

#KOVME is an entry point used only by #KMERG (MERGE keyword program, Figure 3-51). This entry assumes that the file has been written in virtual memory and a line number table is in core.

The first line of the current file to be renumbered is specified by the second parameter. The line number assigned to it is the first parameter. All succeeding lines of the work area file are renumbered, using the third parameter as an increment.

If the file in the work file is a program, all line number references in the program are changed to reflect the new numbering, with each line number occupying four positions. (Imbedded blanks in a line number are removed.)

Parameters can be omitted only in descending order. Default values for the three parameters are 100, 0, and 10.

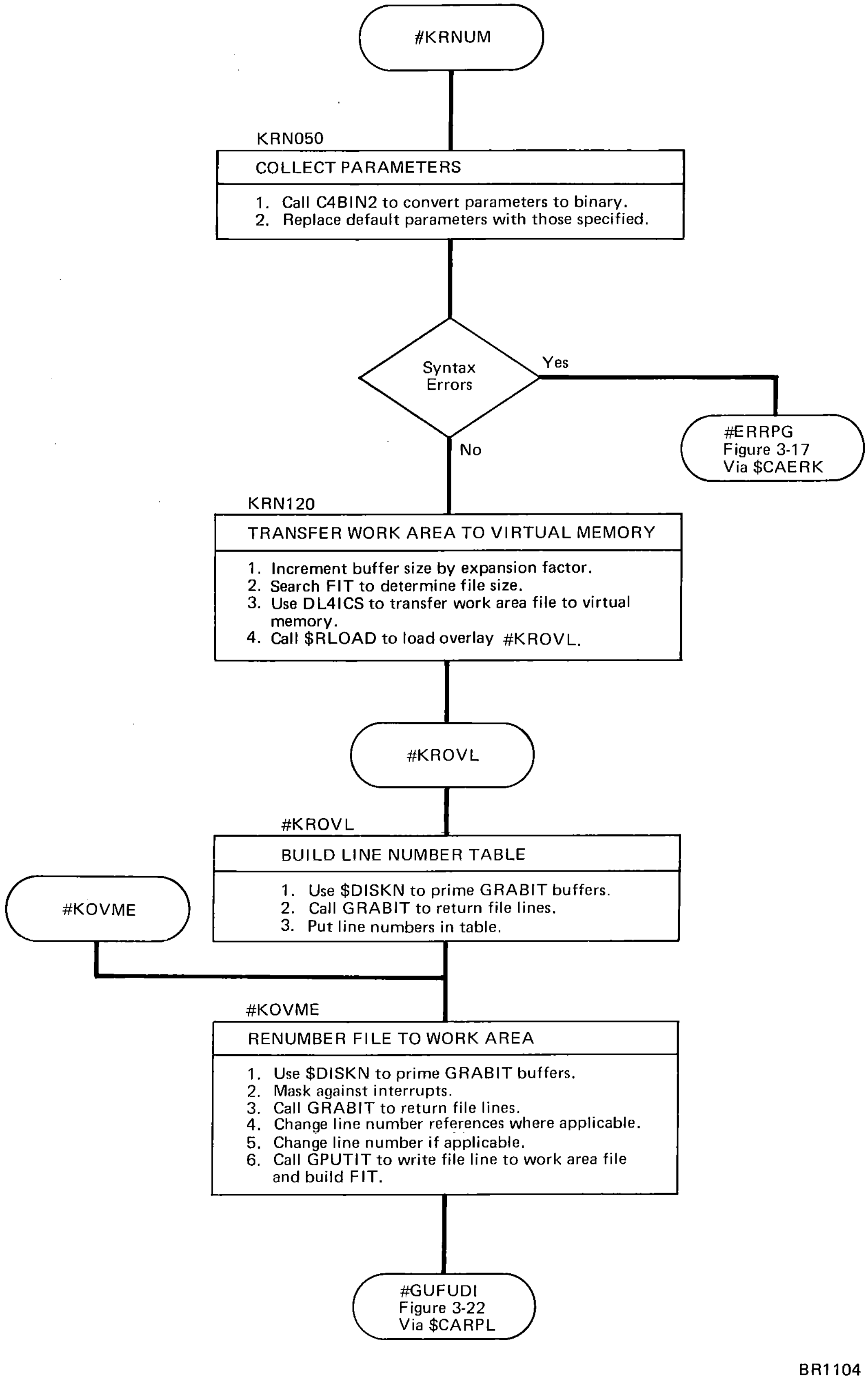


Figure 3-61. RENUMBER Keyword Program (#KRNUM) Flowchart

RESUME Keyword Program—#KRSUM (Figure 3-62)

- #KRSUM returns the suspended program to the execution pause state.
- Running of the returned suspended program is aborted if an "open" file is gone or was modified.
- The suspended program is aborted without running, if the configuration was altered, to allow the user to reconfigure.
- The assembly of #KRSUM contains these major source modules:

  KRSUME—Mainline logic, Figure 3-62
  SVOLID—Search volume-ID table, Figure 3-76
  SFINDF—Find library file, Figure 3-75
  SGETDB—Search password directory, Figure 3-77
  SRCHFN—Search user directory, Figure 3-78
  DL2ICS—Disk logical IOCS, Figure 3-70
  DL4ICS—System work file IOCS, Figure 3-70

The RESUME command restores the currently suspended program (if one exists), along with its associated status information, to the execution pause state so that execution can resume when the GO command is issued.

The program deletes the suspended program file and sets an indicator for the system, enabling a user to suspend another program; and prints the name of the program that is restored to the pause state.

The existence of any of the following conditions results in an error condition when the RESUME command is issued:

1. An operand of any sort with the keyword (a syntax error).
2. Nonexistence of a program in a suspended state.
3. Nonexistence of a file that the program expects (i.e., was deleted).
4. Open indicator in a file is not set on when KSSUME goes out to shut it off.
5. Modified configuration.

Note: In conditions 3 and 4, the suspended program is lost without restoration to a pause state. All conditions result in an error and an error code is set in $CAERR, followed by a branch to $CAERK.

Input to RESUME is the suspended program and its associated status information. Output is the restoration of the program to the execution pause condition.

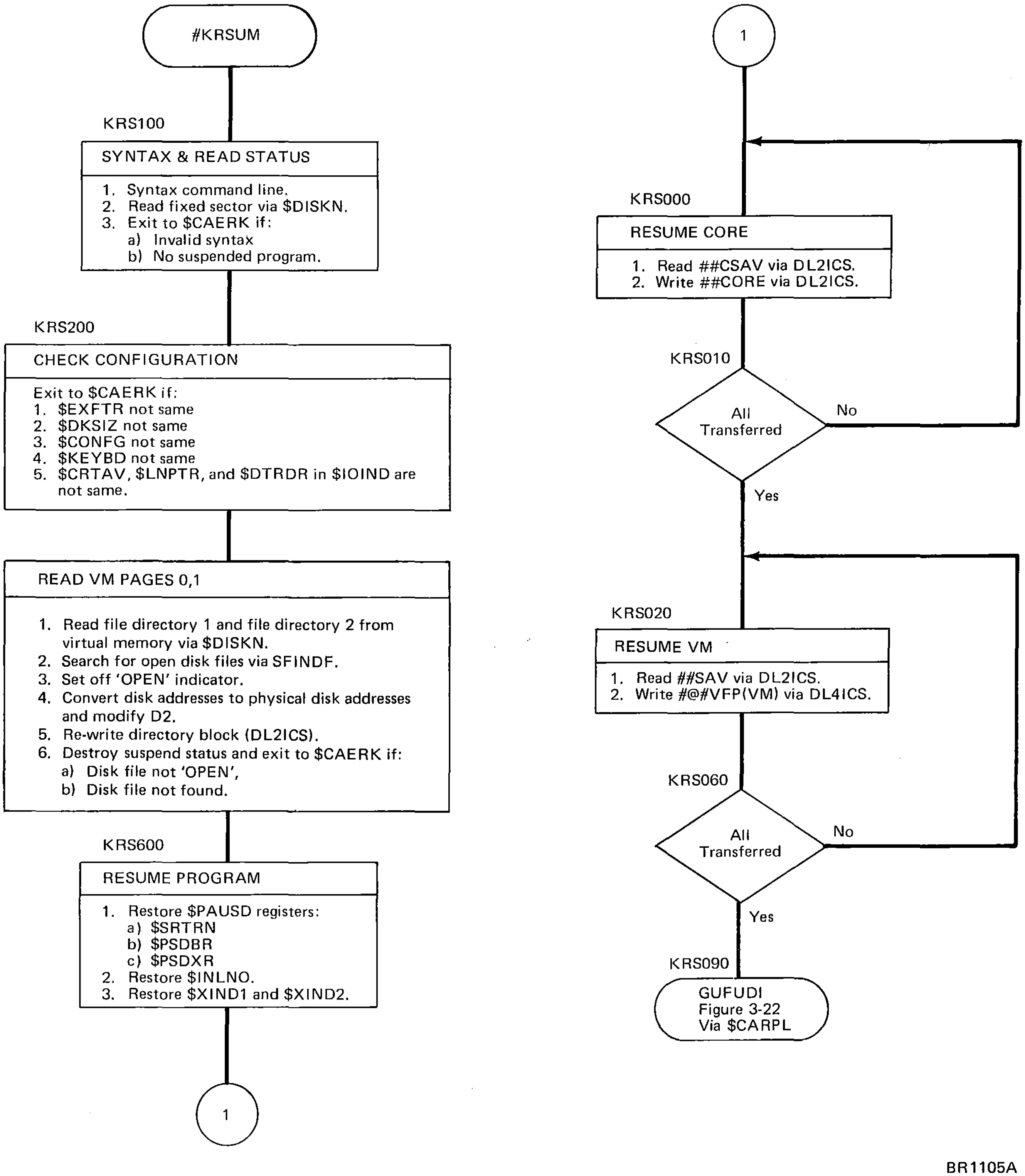


Figure 3-62. RESUME Keyword Program (#KRSUM) Flowchart

RUN/STEP/TRACE Keyword Program—#KRUNI (Figure 3-63)

- #KRUNI provides linkage to the compiler.
- The assembly of #KRUNI contains this major source module:

  KRUNIT—Mainline logic, Figure 3-63

To compile the BASIC program active in the system work file, the compiler (#BCOMP) is loaded directly. To compile a BASIC program from the file library, #KEDIT (Figure 3-40) is loaded to edit the file into the system work file and then load the compiler.

For TRACE, if a list of BASIC identifiers is present, the list is written to virtual memory for use by the compiler.

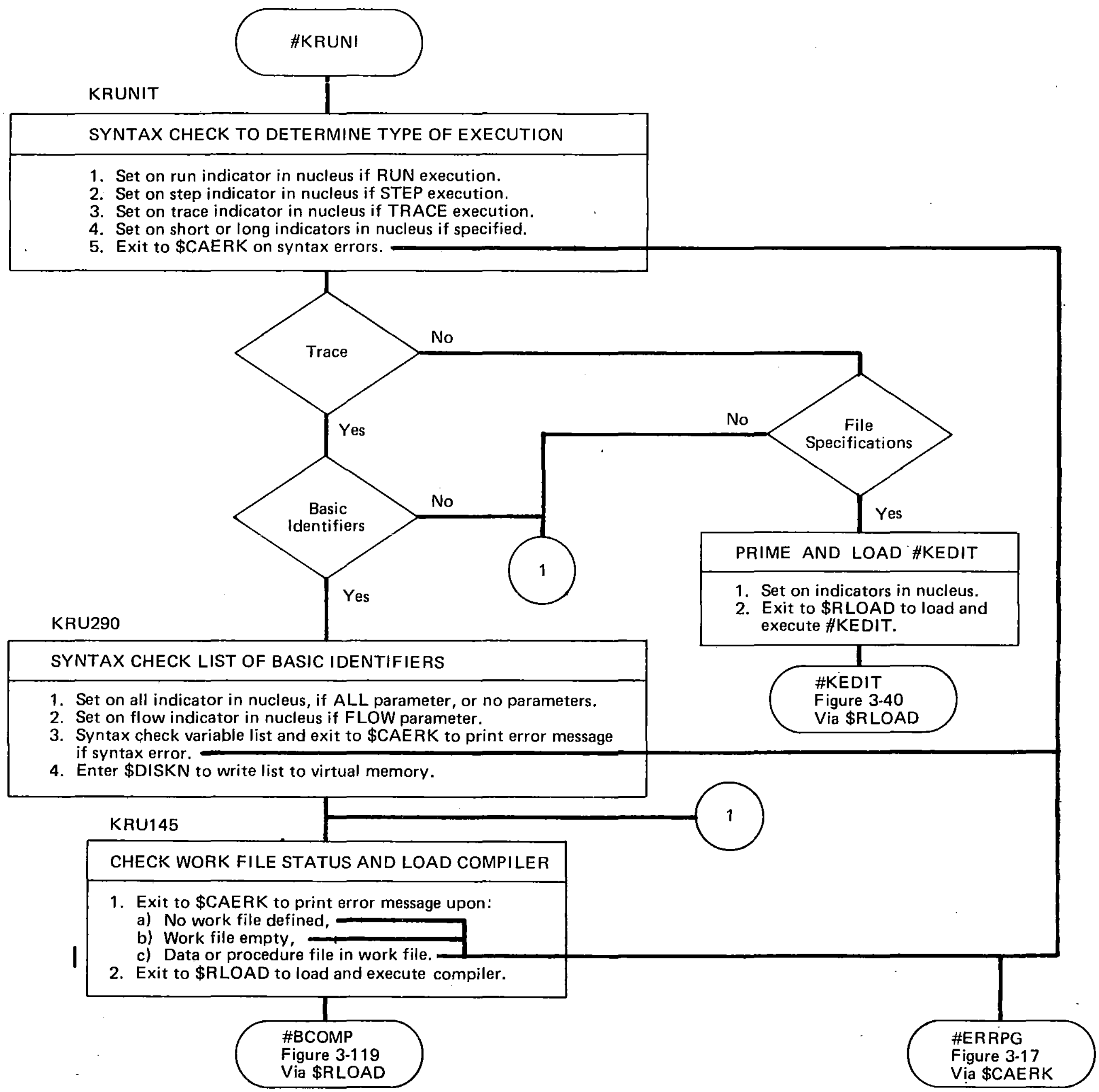


Figure 3-63. RUN/STEP/TRACE Keyword Program (#KRUNI) Flowchart

SAVE Keyword Program—#KSAVE (Figure 3-64)

- #KSAVE stores the active file from the system work file to the file library (system library file).
- The assemblies of #KSAVE and #KSVLA contain these major source modules:

  KSAVEN—Mainline logic, Figure 3-64
  DL2ICS—Disk logical IOCS, Figure 3-70
  DL4ICS—System work file IOCS, Figure 3-70
  STORIN—Null directory insert, Figure 3-79
  STUFID—User directory insert, Figure 3-80
  SRCHFN—Search user directory, Figure 3-78
  SFINDF—Find library file, Figure 3-75
  SGETDB—Search password directory, Figure 3-77
  SVOLID—Search volume-ID table, Figure 3-76
  SURCHN—Search null directory, Figure 3-81

The new file is stored on the same volume as the old file when the filenames match. The new file does not necessarily occupy the same physical disk space. The old file physical disk space may be placed in the null directory. A file is not replaced if it is pooled or protected.

#KSAVE loads #SPACK (Figure 3-86) when disk space can be obtained by packing the file library. #SPACK loads, and returns to, #KSAVE.

#KSAVE

KSAVEN

SYNTAX CHECK AND FIND SPECIFIED FILE

1. Mask interrupts and set pointer for input buffer.
2. Save disk block count from FIT.
3. Call SUFFER to decode file-specification.
4. Call SCSTG to decode character constant header.
5. Call SFINDF to locate current password and file.
6. Call $RLOAD to load #KSVLA.
7. Exit to $CAERK if any errors.

Password Found

No → #ERRPG Figure 3-17 Via $CAERK

Yes

KSA007

CHECK FILE STATUS AND TEST IF FILE**

1. If file name was found and file is pooled or protected, take the error exit.
2. If ** file, call SFINDF to find first available space in the disk searching order.
3. If space not available, call SPACKU to pack the library area.
4. If space found for ** file, return it to null directory.
5. Call DL4ICS to read in FIT.

KSA030

CALCULATE FILE LENGTH AND BUILD READ TABLE

1. If program generated data file disk block count is file size, calculate FIT size and add to disk block count.
2. If BASIC file, add the length of the I/O record.
3. Build read table of displacements and number of contiguous sectors.

1

1

KSA140

SELECT SPACE TO USE

1. Call SURCHN to look for space to use.
2. If space not found, go to $CAERK, if total null space not enough.
3. If total null space required, call #SPACK to pack library area.
4. If file name and new space found, find closest to start of library.
5. If new space closer, use it and call STORIN to send old space to null directory.
6. If null directory full, call #SPACK to pack library.
7. If old space is closer, but new file is larger, use new space.
8. If old space closer, and new file is smaller, call STORIN to return new space and remainder of old space to null directory.

KSA210

UPDATE USER DIRECTORY AND START I/O

1. Update old entry or build new entry for user directory.
2. Call STUFID to make entry and output the directory.
3. Set up DPL to output FIT and I/O sector, if file is not a program generated data file.
4. Set FIT entry displacements in physical/logical order.
5. Call DL2ICS to output FIT and I/O sectors.

KSA229

COPY FILE FROM WORK AREA TO LIBRARY

1. Pick up entries from read table and call DL4ICS to fill the buffer.
2. Call DL2ICS to write the buffer to the library area.
3. On last entry, call DL2ICS to empty the buffer.
4. Go to $CARPL to reload #GUFUD.

#GUFUD Figure 3-22 Via $CARPL

BR1107A

Figure 3-64. SAVE Keyword Program (#KSAVE) Flowchart

SET Keyword Program—#KSETI, #KSOVR (Figure 3-65)

- #KSETI syntax checks the SET command line, assuring valid syntax for the SET overlay #KSOVR.
- The assembly of #KSETI contains this major module:

  KSETIT—Mainline logic, Figure 3-65
- #KSOVR assigns a value to an existing program variable during a program execution pause state.
- The assembly of #KSOVR contains this major module:

  KSOVRL—Mainline logic, Figure 3-65

The SET command line is syntax checked. When correct syntax is assured, initializing operations are performed to load the overlay #KSOVR.

The specified variable or array element symbol is converted to a virtual address. The specified constant is converted to a form suitable for storage in virtual memory and then moved to the virtual memory address associated with the symbol.

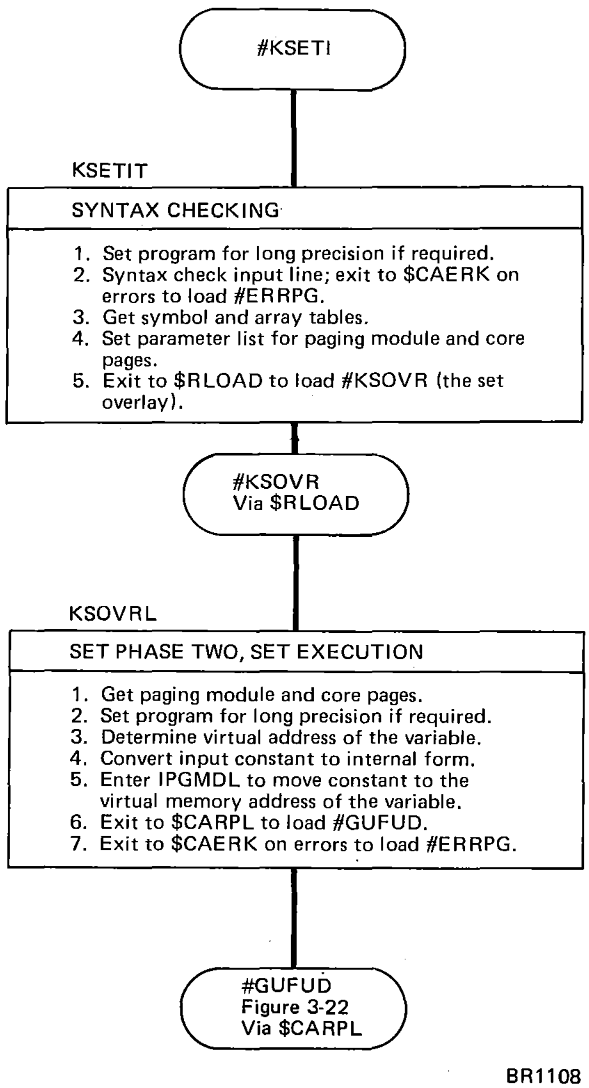


Figure 3-65. SET Keyword Program (#KSETI, #KSOVR) Flowchart

SUSPEND Keyword Program—#KSSPN (Figure 3-66)

- When the SUSPEND command is issued, the current program in an execution pause condition is saved for future completion of execution.
- The assembly of #KSSPN contains these major source modules:

  KSSPND—Mainline logic, Figure 3-66
  SVOLID—Search volume-ID table, Figure 3-76
  SFINDF—Find library file, Figure 3-75
  SGETDB—Search password directory, Figure 3-77
  SRCHFN—Search user directory, Figure 3-78
  DL2ICS—Disk logical IOCS, Figure 3-70
  DL4ICS—System work file IOCS, Figure 3-70

The SUSPEND command causes the program that is currently in an execution pause condition (if one exists) to be saved, along with its associated status information, for future completion of execution. This enables the user to execute other programs, or certain system functions, without affecting the suspended program. If the RESUME command is issued and a program is in the suspended state, the program is returned to an execution pause condition. If two SUSPEND commands are issued in succession, the first suspended program is replaced by the second suspended program if the optional filename of the first program is specified in the second command. If any active data files are modified while the program is in the suspended state, the suspended program is aborted.

The associated status information suspended with the program includes the 64k of virtual memory that is unique for this program, a six-sector symbol table, register data for return to the calling point, and other indicators.

Any of the following conditions results in an error when the SUSPEND command is issued:

1. An operand of any sort, other than the optional filename, with the keyword (a syntax error).
2. Any program already in a suspended state, if the optional filename is not specified.
3. The nonexistence of a program in an execution pause state.
4. An active disk scratch file for the program.
5. The nonexistence of a file associated with the program for suspension.

Note: This error causes a hard halt after a message is displayed.

Each of the preceding conditions results in an error; and an error code is set in $CAERR, followed by a branch to the error exit routine at $CAERK.

Input information to SUSPEND is (1) the program in an execution pause state and (2) its associated status information. Output is the transfer of this program and information to the suspend save area.

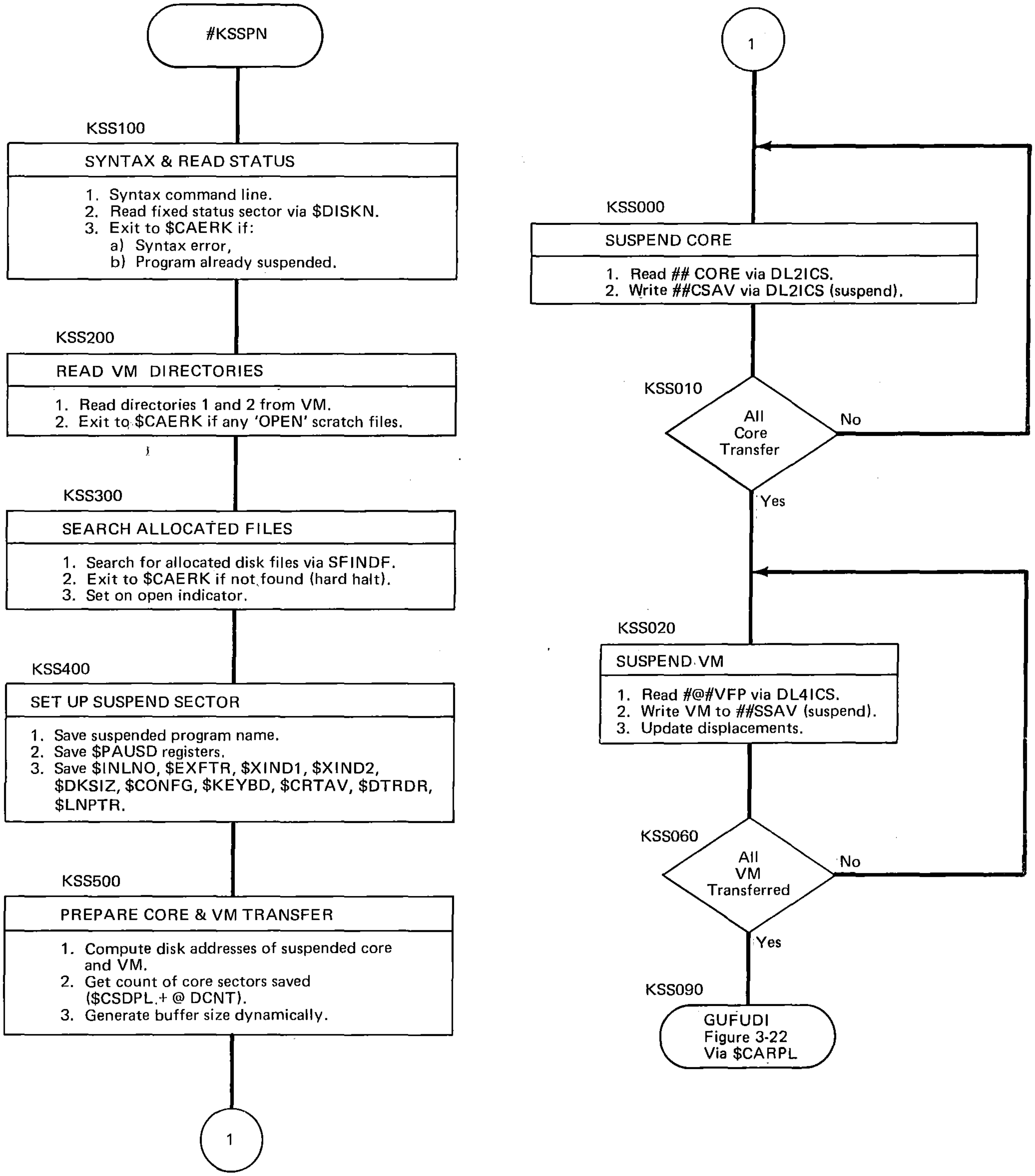


Figure 3-66. SUSPEND Keyword Program (#KSSPN) Flowchart

SYMBOLS Keyword Program—#KSYMB (Figure 3-67)

- #KSYMB displays all variable names used in the system work area program.
- The assembly of #KSYMB contains these major source modules:

  KSYMBL—Mainline logic, Figure 3-67
  GRABIT—Work file input, Figure 3-74
  DL4ICS—System work file IOCS, Figure 3-70
  DLPRNT—IOCS for output, Figure 3-71
  SVARAB—Variable scan, no flowchart.

#KSYMB scans the lines of the program in the system work area to locate the variable names used.

A symbol table is built, using one byte for each possible variable name. If a variable is referenced in a disabled line, an indicator for this is set in the appropriate symbol table byte also. When all variables have been scanned, this symbol table is printed with each symbol occupying a seven-character field of which the last character is always blank. If the variable was in a disabled line, * is printed in the first character position of the output field. Nine variables are printed on one line, giving a print line 63 characters long. Output can be specified to go to the matrix printer or CRT; otherwise, the system printer is assumed to be the output device.

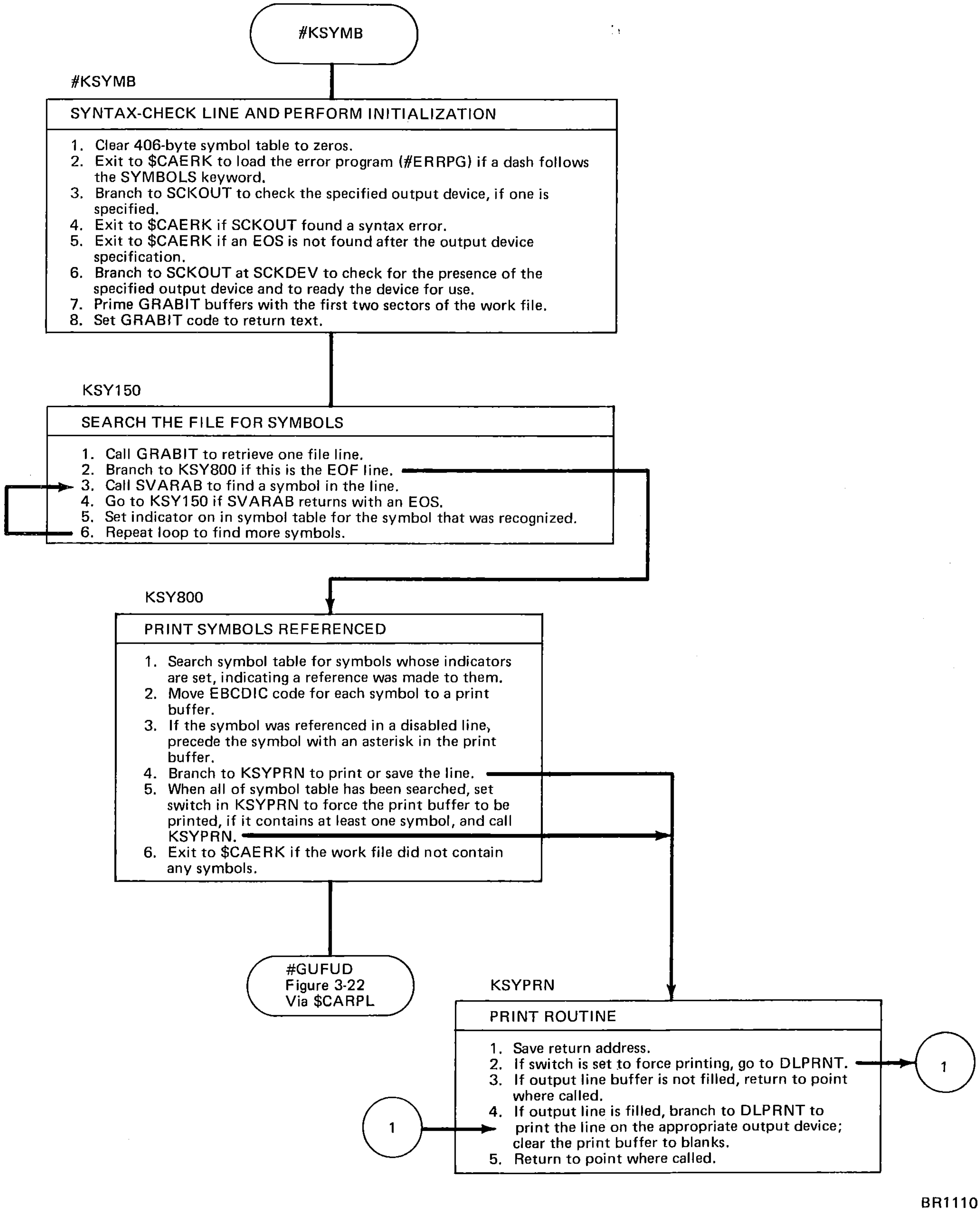


Figure 3-67. SYMBOLS Keyword Program (#KSYMB) Flowchart

WIDTH Keyword Program—#KWIDT (Figure 3-68)

- #KWIDT changes the margin values for the system printer in the nucleus communications area (refer to Figure 5-1).
- The assembly of #KWIDT contains this major source module:

  KWIDTH—Mainline logic, Figure 3-68.

#KWIDT

SYNTAX CHECK LINE AND ACCUMULATE PARAMETERS

1. Initialize new left margin to old left margin, in case a new one is not specified.
2. If WIDTH is immediately followed by a dash, exit to $CAERK to load #ERRPG, the error program.
3. Exit to $CAERK if the line contains no parameters.
4. Branch to C4BIN2 to convert the width parameter to binary.
5. Exit to $CAERK if the width was not an integer, if it contained more than four digits, and if it was not validly delimited (i.e., followed by a comma, blank(s), or EOS).
6. If EOS does not follow (after optional blanks) the width parameter, branch to C4BIN2 to convert the left margin to binary.
7. Exit to $CAERK if the left margin was not a number, if it contained more than four digits, or it was not followed (optional blanks) by EOS.

KWI500

CHECK FOR VALID SPECIFICATION(S)

1. If a left margin value was specified, subtract '1' from it; if negative number results, exit to $CAERK.
2. If specified width is less than '18', exit to $CAERK.
3. Compute the right margin by adding the left margin to the width.
4. Exit to $CAERK if the right margin exceeds the physical capacity of the printer (i.e., it is greater than 132 or 220).
5. If the current print position is less than the new left margin,
   a) Calculate the difference between the current print position and the new left margin, in KWIHLD.
   b) Set the value of KWIHLD in the count of the print parameter list.
   c) Set the right margin in nucleus ($RMRGN) = 220, temporarily.
   d) Branch to $$PRNT to print blanks over the new left margin (i.e., move the print position over).

KWI700

STORE THE NEW MARGINS IN THE NUCLEUS

1. Store the new left margin in the nucleus at $LMRGN.
2. Move a carrier return to position the print head. ($$PRNT)
3. Set the 'I/O routines in core' indicator off.
4. Store the new right margin in the nucleus at $RMRGN.

#GUFUD
Figure 3-22
Via $CARPL

BR1111

Figure 3-68. WIDTH Keyword Program (#KWIDT) Flowchart

WRITE Keyword Program—#KWRIT (Figure 3-69)

- #KWRIT changes the device used as system printer to CRT, matrix printer, or both.
- The assembly of #KWRIT contains this major source module:

  KWRITE—Mainline logic, Figure 3-69

#KWRIT stores these addresses at $PRDEV in the nucleus:

DPRINT for matrix printer IOCR.
DSPLYN for CRT IOCR.
DSPYMP for matrix printer IOCR and CRT IOCR; this label is an entry point to the CRT IOCR.

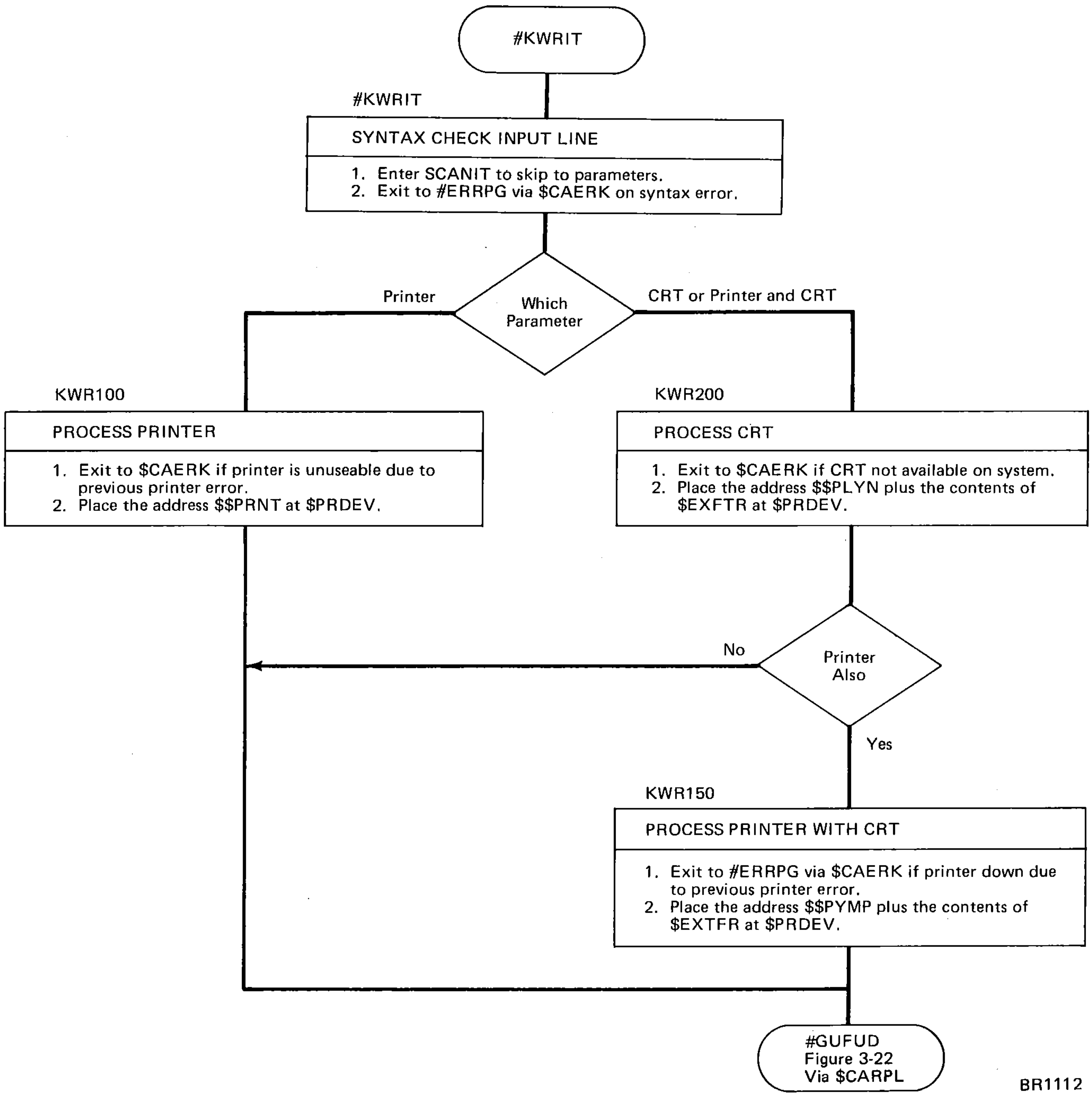


Figure 3-69. WRITE Keyword Program (#KWRIT) Flowchart

System Work File IOCS—DL4ICS (Figure 3-70)

- DL4ICS converts relative disk addresses to physical disk addresses within the work file or virtual memory. It calls DKDISK to perform the disk I/O operation.
- The calling sequence for DL4ICS is:

```
B    DL4ICS
DC   AL2(DPL)     DPL is the address of the disk parameter list (Figure 3-3). The second
                  byte of the disk address is a relative sector displacement.
```

The disk address is specified as a physical cylinder, and a single-byte sector displacement relative to sector 0 on the specified cylinder. If a multiple-sector operation is required, DL4ICS splits the operation and makes multiple calls to DKDISK if necessary to properly cross cylinder boundaries.

Disk Logical IOCS—DL2ICS (Figure 3-70)

- DL2ICS converts relative disk addresses to physical disk addresses within a two-track file, and calls DKDISK to perform the disk I/O operation.
- The calling sequence for DL2ICS is:

```
B    DL2ICS
DC   AL2(DPL)     DPL is the address of the disk parameter list (Figure 3-3). The disk
                  address is a two-byte relative displacement.
```

The disk address is specified as a two-byte cylinder and sector displacement relative to a predefined disk address. This predefined disk address (two-byte physical address) must be stored at label DL2RAD prior to the first call to this IOCS. Files accessed by this IOCS are logically on one volume; therefore, the disk ID and drive number do not change from those specified in the predefined starting address.

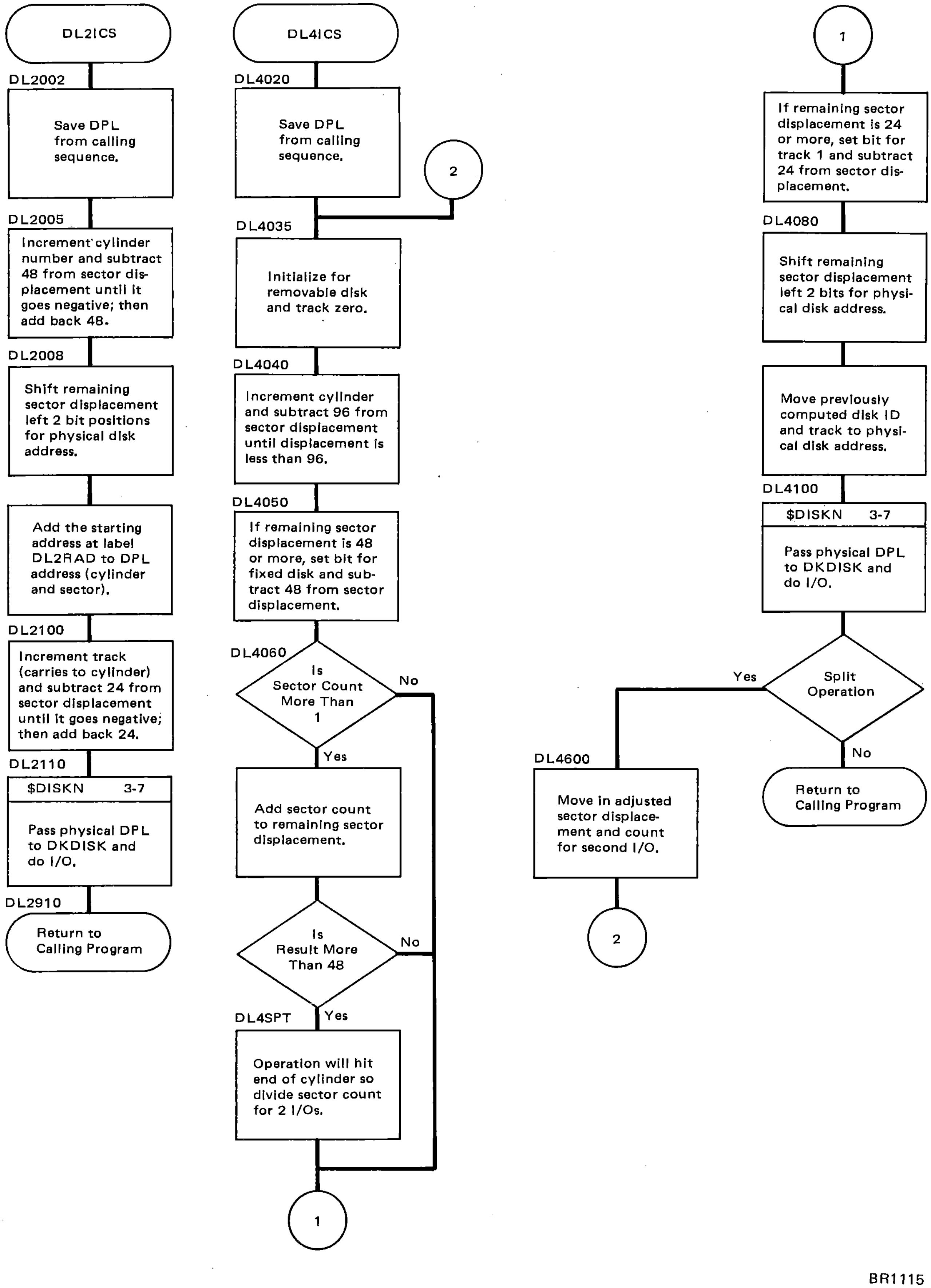


Figure 3-70. Disk IOCS Routines (DL2ICS, DL4ICS) Flowchart

Line Printer Interface—DLPRNT (Figure 3-71)

- DLPRNT allows device independence when listing lines on the CRT or matrix printer.
- If the CRT is to be used, the speed of the displayed lines is controlled, and the roll-stop and popup commands are recognized.
- If the bidirectional printer is used, printing is done in both directions.
- The calling sequence for DLPRNT is:

```
B     DLPRNT
DC    AL2(PPL)      PPL is the address of the print parameter list (Figure 5-23).
```

To control which device receives the output, a device type code may be placed at the label DLPTYP. The device type code is the displacement (from DLPRNT) to the routine for interface to the proper device. DLPTYP is initially set for output on the system printer. Values at label DLPTYP define the displacement equated to the label and its associated device:

Label	*Device*
DLPMPR	For output on MP
DLPCRT	For output on CRT
DLPSPT	For output on system printer

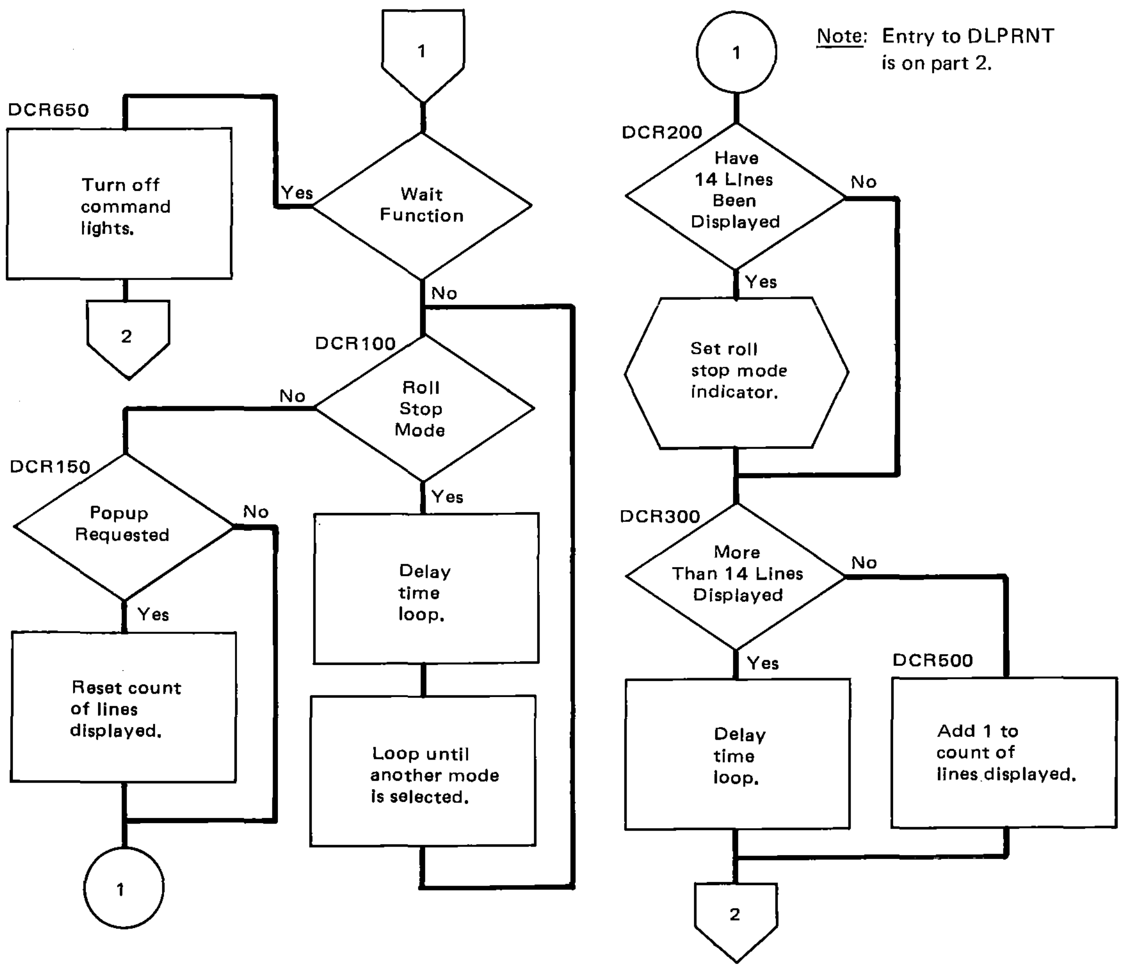


Figure 3-71. Line Printer Interface (DLPRNT) Flowchart (Part 1 of 2)

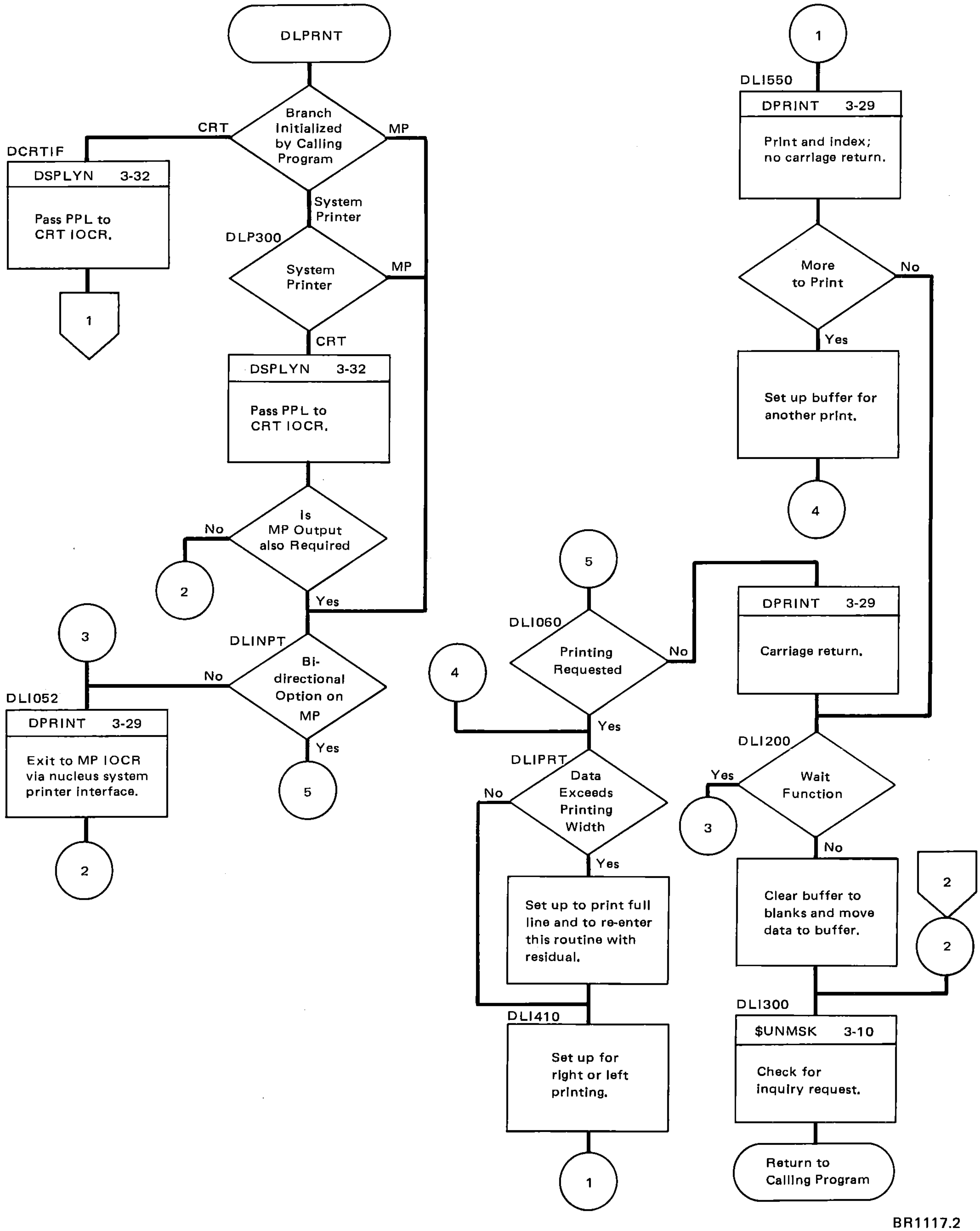


Figure 3-71. Line Printer Interface (DLPRNT) Flowchart (Part 2 of 2)

Card Punch IOCR–DCDOUT (Figure 3-72)

- DCDOUT performs punching I/O for the data recorder.
- When the call is made to this routine, the previous punching operation is checked for errors before starting the new request.
- The routine then exits, allowing continued processing while the card is being punched.
- The calling sequence for DCDOUT is:

```
B     DCDOUT
DC    AL2(PPL)      PPL is the address of the print parameter list (Figure 5-23).
```

I/O Routines

Two I/O functions are provided by DCDOUT:

1. Punch–96 bytes of data are punched (80, if configured for the 129), starting at the core address specified in the PPL.
2. Wait and check for errors–This function allows the punching operation to complete error-free before returning to the calling routine.

Error Recovery Procedures (ERP's)

No error returns are made to the calling program. All ERP's are included within the IOCR. Not-ready conditions cause a soft halt. Off-line and hopper full/empty conditions cause the CPU to loop on the TIO until the problem is corrected by the operator. Once the problem has been corrected, the SIO sent to the device is executed automatically.

Data compare errors are retried once. Incorrect card code is accepted from the system, but the resulting punched card is bad.

If five compare errors or hopper jams occur in one operation, the system comes to a hard halt, requiring a re-IPL. Errors are logged on the fixed disk.

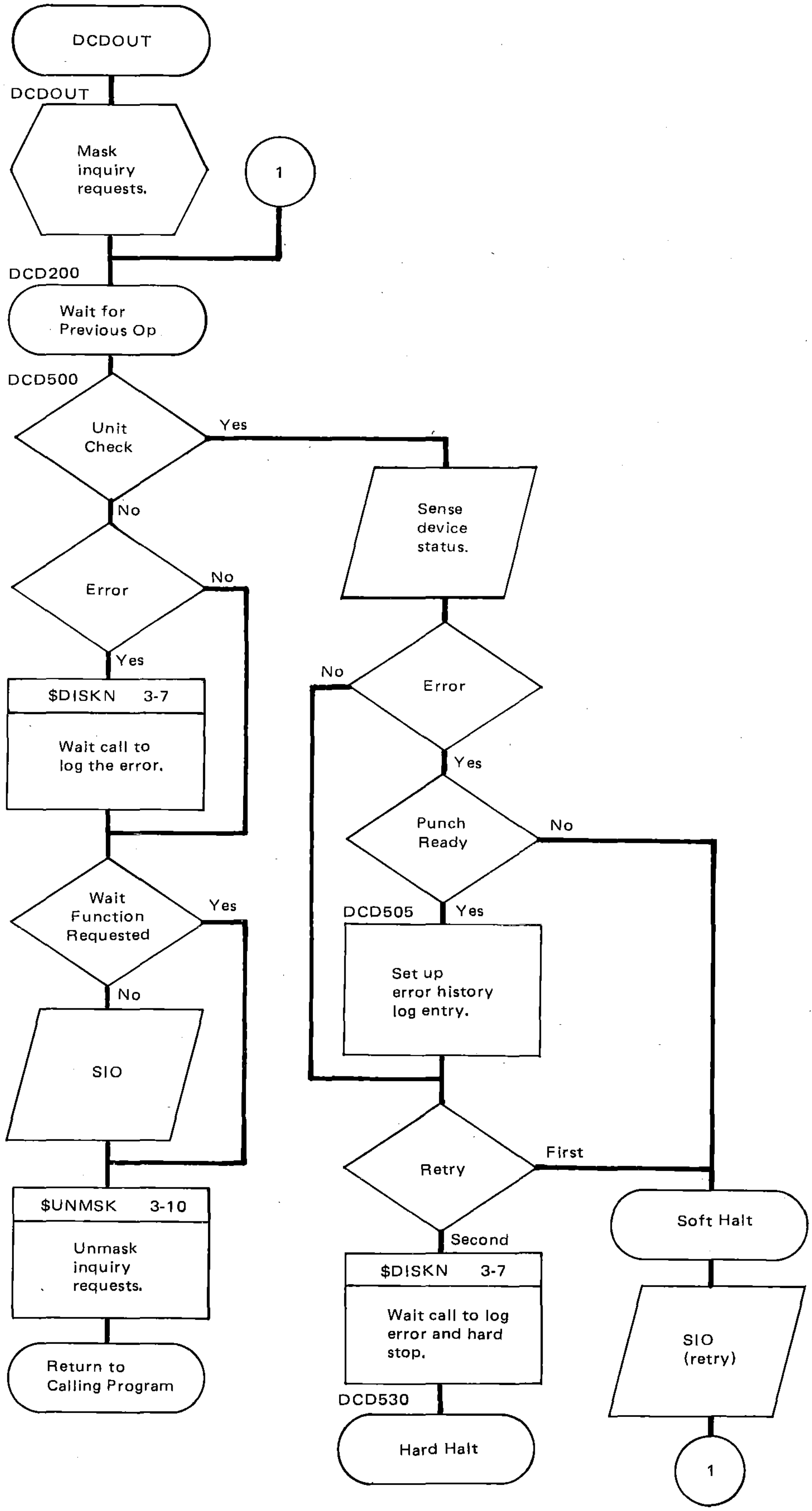


Figure 3-72. Card Punch IOCR (DCDOUT) Flowchart

Work File PUT Subroutine—GPUTIT (Figure 3-73)

- GPUTIT is a routine used to place single statements in the work file or in a temporary VM file, in ascending order.

When this routine is first called, it initializes the file index table and places the statement passed to the routine in a core buffer as the first statement of a new file. Each statement passed via a subsequent call to GPUTIT is placed in the core buffers, following the previous statement. As a statement is placed in a core buffer, the file index table (refer to Figure 5-16) is adjusted to reflect the inclusion of that statement unless GPUTIT = 1 (set on).

When a core buffer is filled to capacity, it is written to disk, and file building continues in the alternate core buffer. When the last statement of the file has been placed in a core buffer, it is followed by the end-of-file record. The last core buffer is then written to disk.

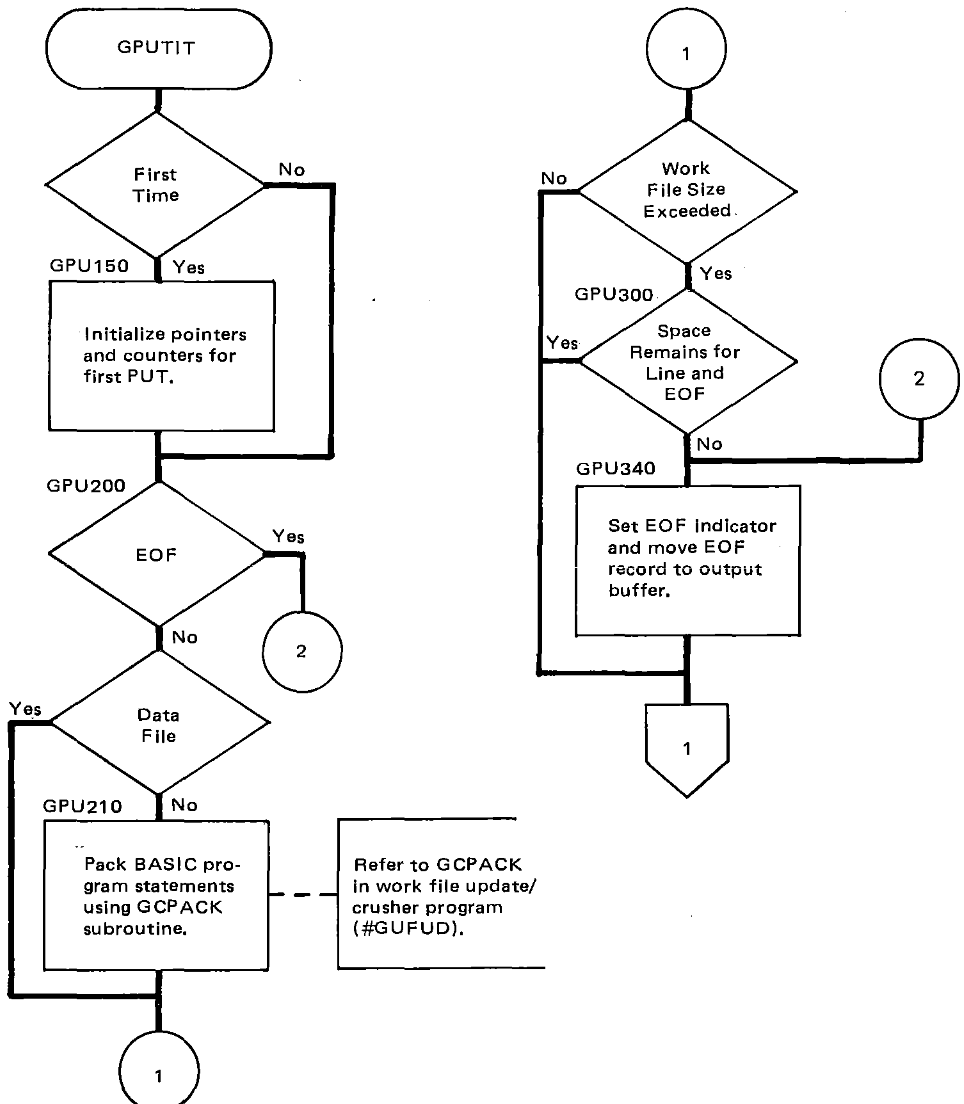


Figure 3-73. Work File PUT Subroutine (GPUTIT) Flowchart (Part 1 of 2)

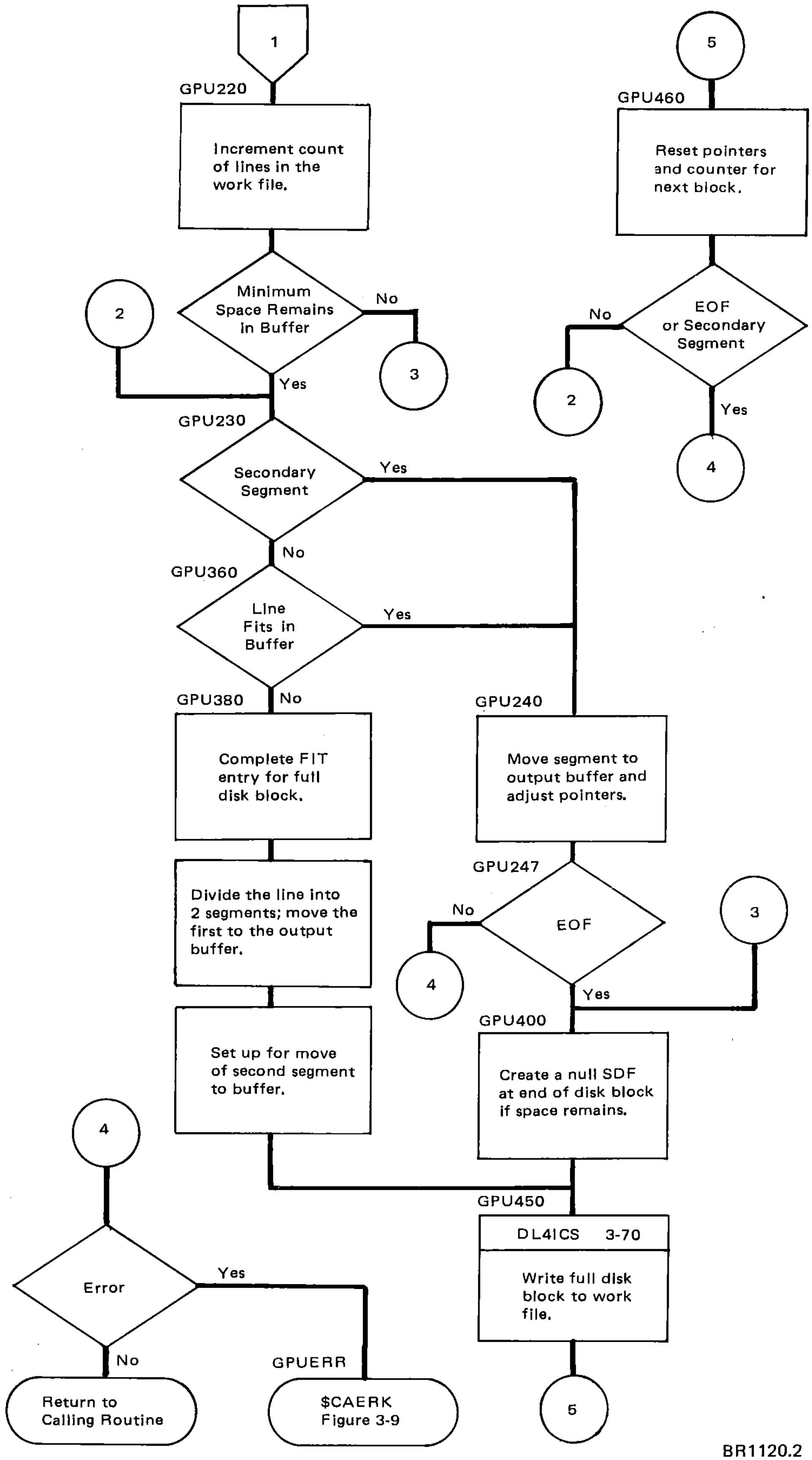


Figure 3-73. Work File PUT Subroutine (GPUTIT) Flowchart (Part 2 of 2)

Work File Retrieval Subroutine—GRABIT (Figure 3-74)

- GRABIT locates sequential statements in the file specified by the user, and, depending upon the option chosen, passes back the statement or skips to the next.

After being primed by the calling program, GRABIT reads logically consecutive blocks of segmented statements, from the file specified by the user, into core. GRABIT returns with @XR pointing to the binary line number of the next statement.

In addition to @XR, GRABIT parameters can be set to cause the binary line number; the type code; and the unpacked, non-segmented text of the next statement to be placed in areas defined by the user. If GRABIT is used to skip through the statements without unpacking them or changing their length or segmented condition, GRABIT can be instructed to return the blocks to their original disk address if the specified file is accessed by DL4ICS.

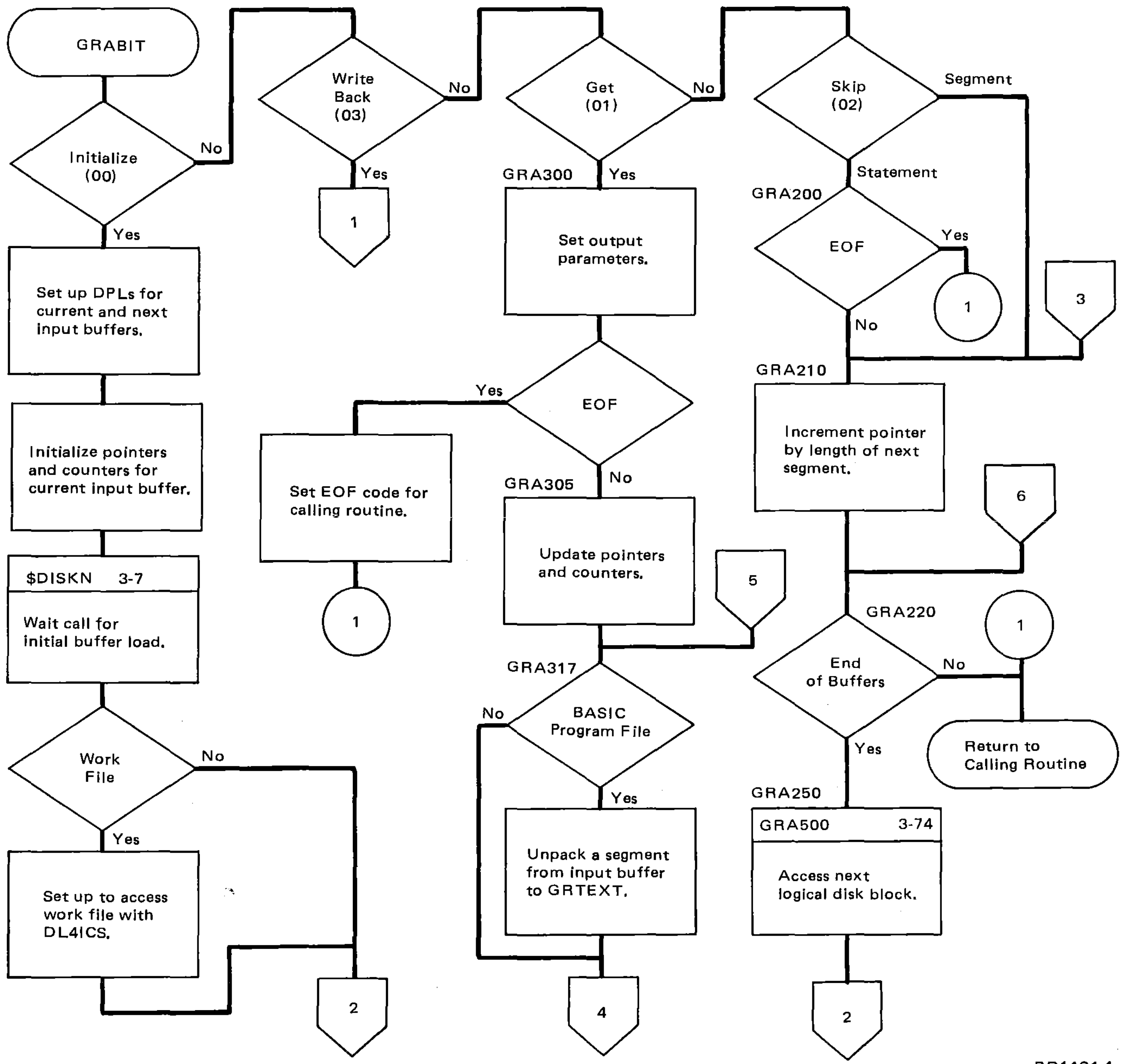
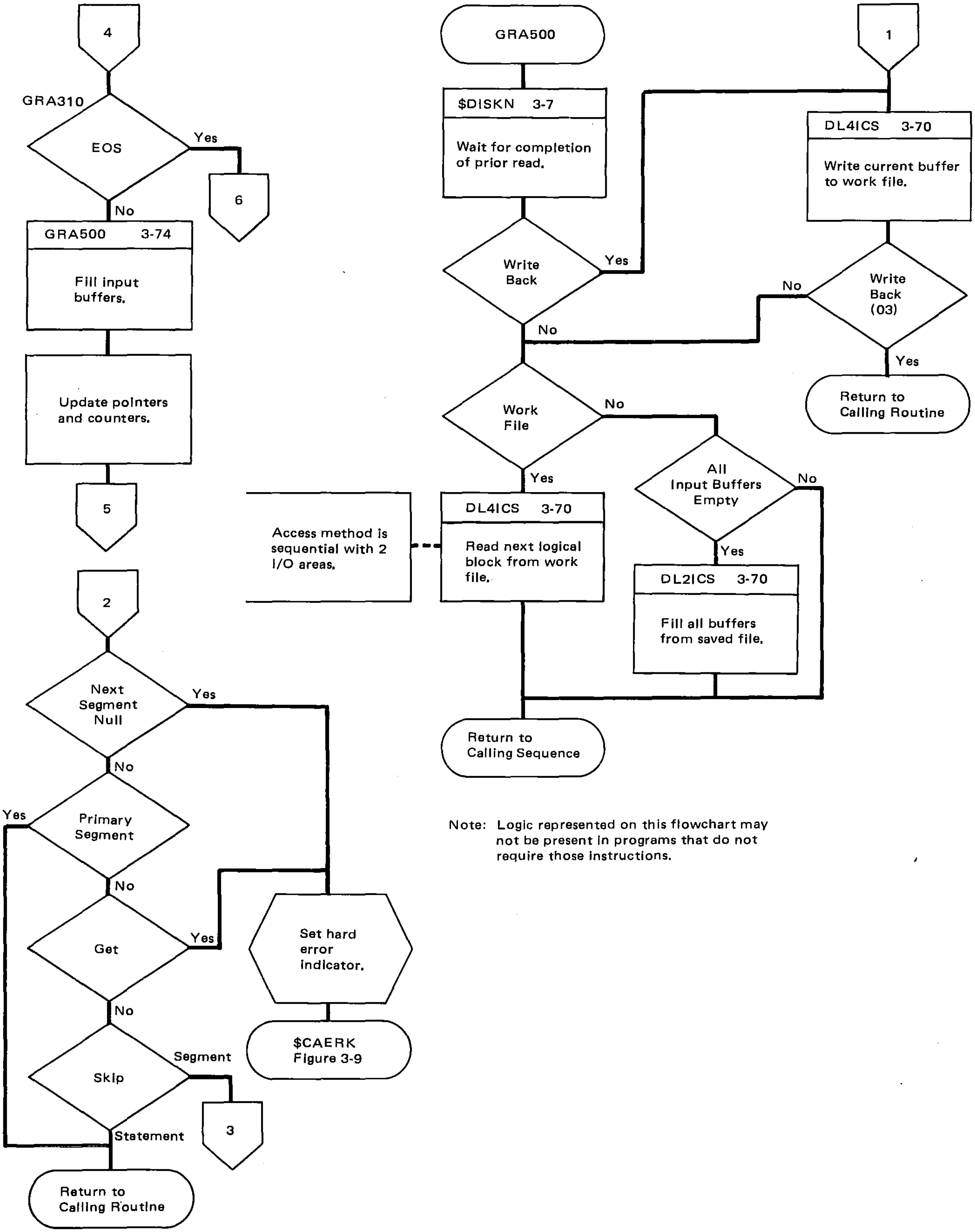


Figure 3-74. Work File Retrieval Subroutine (GRABIT) Flowchart (Part 1 of 2)



Note: Logic represented on this flowchart may not be present in programs that do not require those instructions.

BR1121.2

Figure 3-74. Work File Retrieval Subroutine (GRABIT) Flowchart (Part 2 of 2)

Find Specified File Subroutine—SFINDF (Figure 3-75)

- SFINDF is a control subroutine used to locate a specified password and/or filename.

The function of SFINDF depends upon the way the file is specified:

1. If a filename, password, and volume-ID are all explicitly specified, SFINDF issues calls to SVOLID, SGETDB, and SRCHFN to search the appropriate file library directories to find the specified file.
2. If the password or volume-ID is not explicitly defined, SFINDF defaults to the current user specifications, if they exist, for the missing parameters and then issues the required calls to SGETDB and/or SRCHFN to locate the file.
3. If a one-star (*) or two-star (**) filename is specified, SFINDF either searches the specified disk if a volume-ID was specified or searches every disk on the system for the file if a volume-ID was not specified. Parameters may be set to terminate the search after processing a specified number of disks containing file libraries.

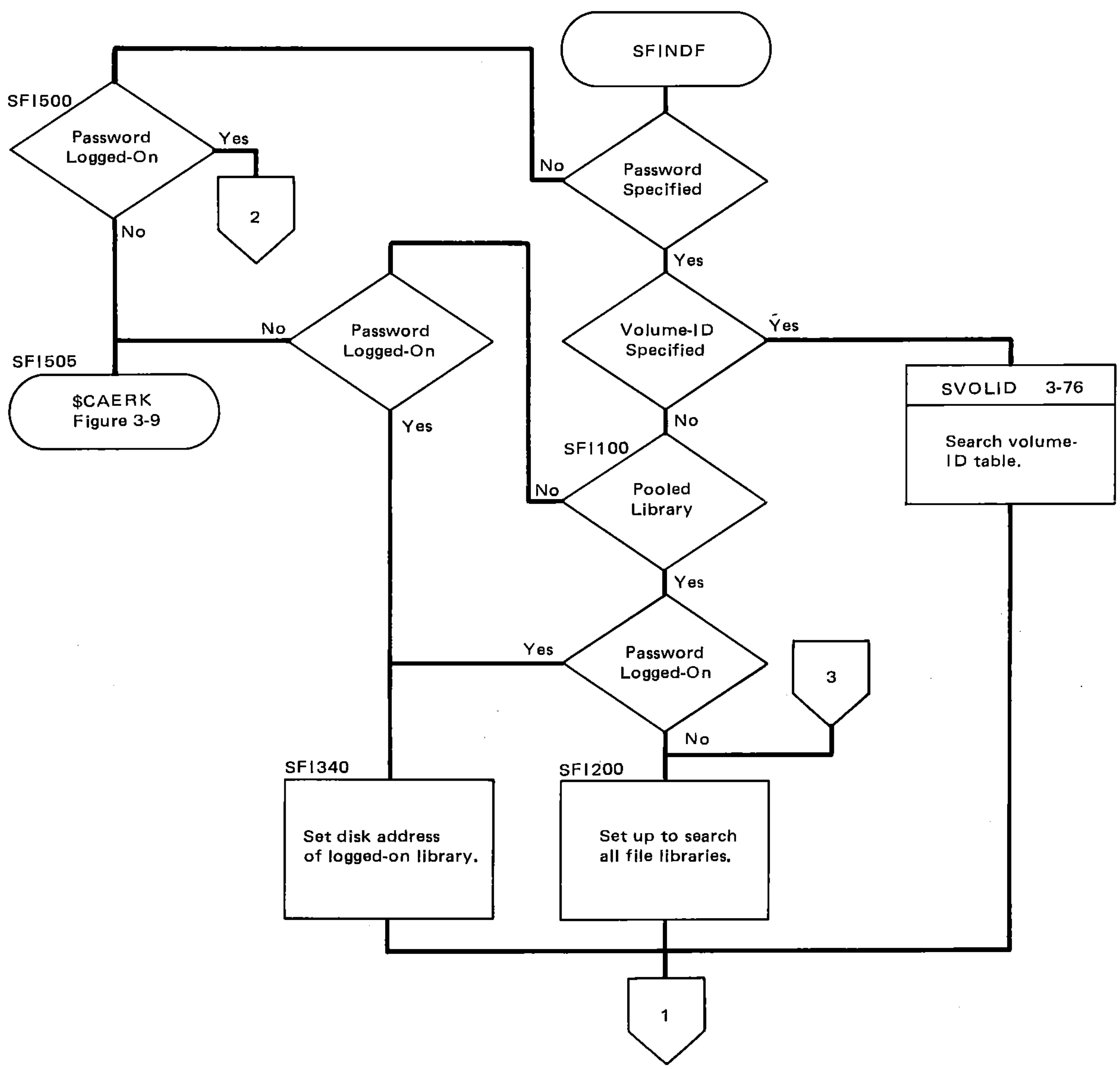


Figure 3-75. Find Specified File Subroutine (SFINDF) Flowchart (Part 1 of 2)

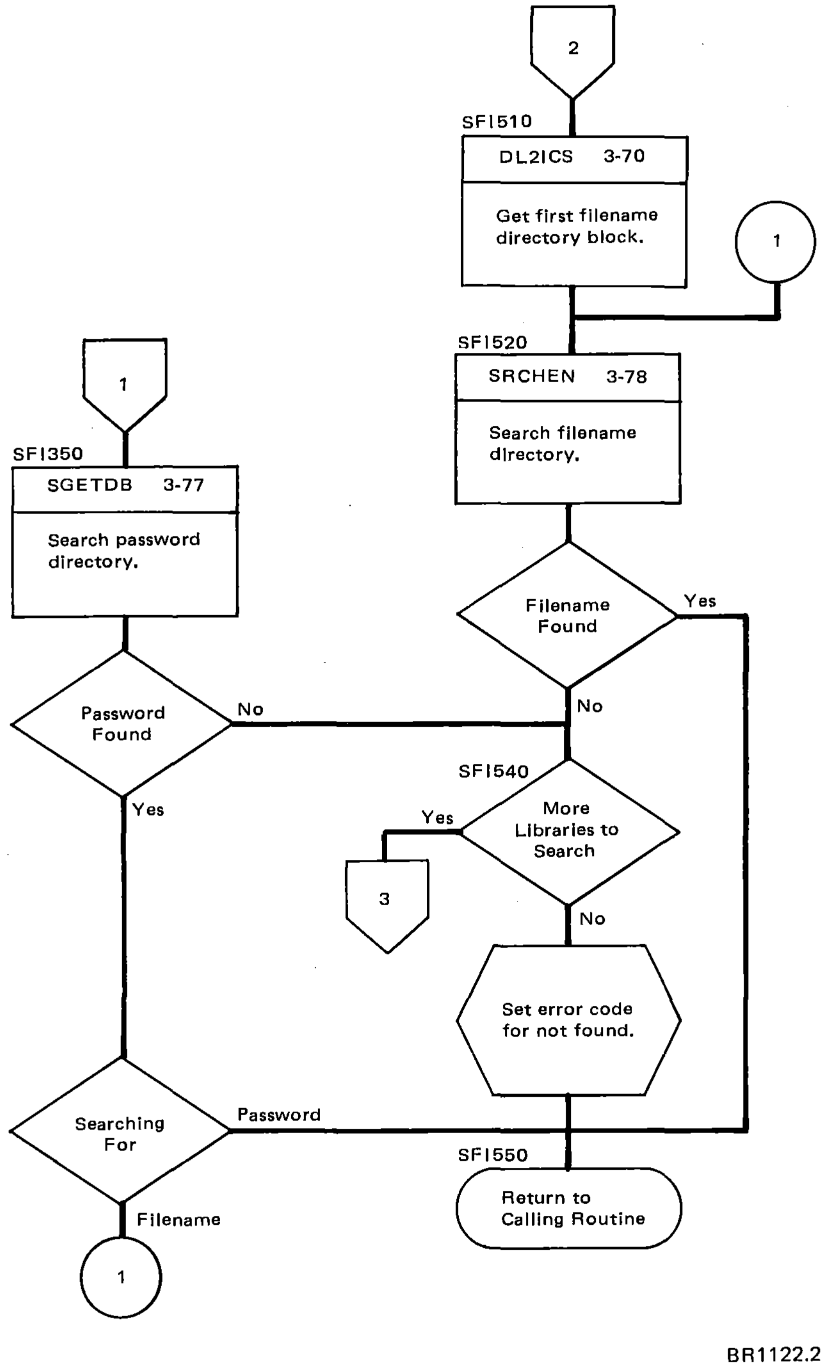


Figure 3-75. Find Specified File Subroutine (SFINDF) Flowchart (Part 2 of 2)

Find Volume-ID Subroutine—SVOLID (Figure 3-76)

- SVOLID searches the volume-ID table in the nucleus communications area for a specified volume-ID.

SVOLID scans the volume-ID table for a specified volume. If the volume is not found, an error code is put in $CAERR and an exit to SVOERR in the using program is taken. If more than one volume with the same volume-ID is found, the user is requested to indicate which drive and disk is to be used. If the user is unable to resolve the conflict, the current system command is rejected. If the system input device is the card reader, and duplicate volume-ID's have been found, the current system command is rejected.

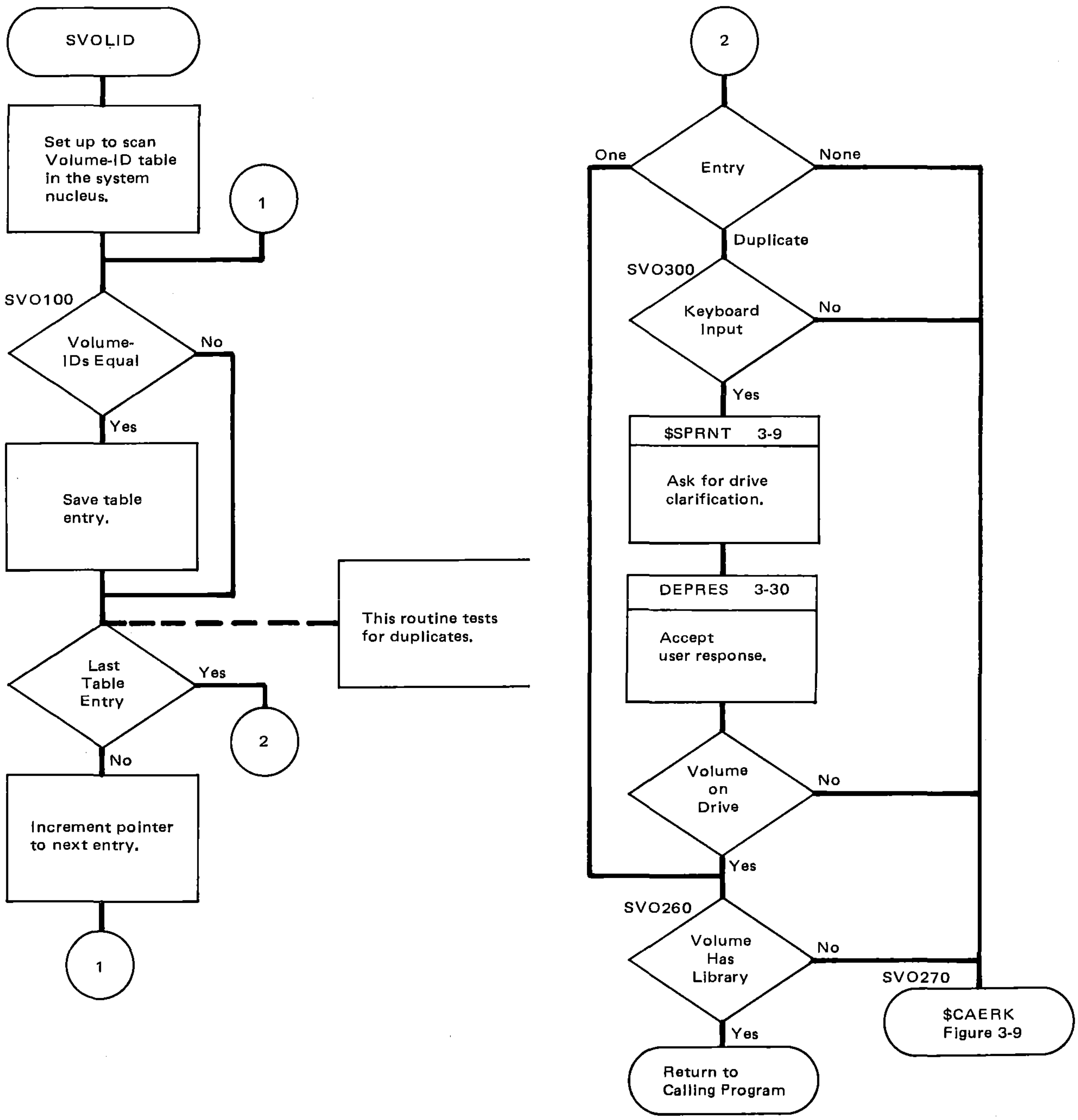


Figure 3-76. Find Volume-ID Subroutine (SVOLID) Flowchart

Search Password Directory Subroutine—SGETDB (Figure 3-77)

- SGETDB searches the password directory for a specified password or reads into core the first directory block of the file-specification password.

SGETDB searches the password directory for a specified password and reads into core the first directory block associated with that password. If SM1PDS is set, only the entry address of the password is passed to the caller in SMPEAD. If the directory block is requested and the password is not found, the error code is placed in $CAERR, SM1PNF is turned on in SMIND1, and a normal return is taken. If only the password is requested and the password is not in the directory, the address for the next entry is passed in SMPEAD, SM1PNF is turned on, and the return is taken.

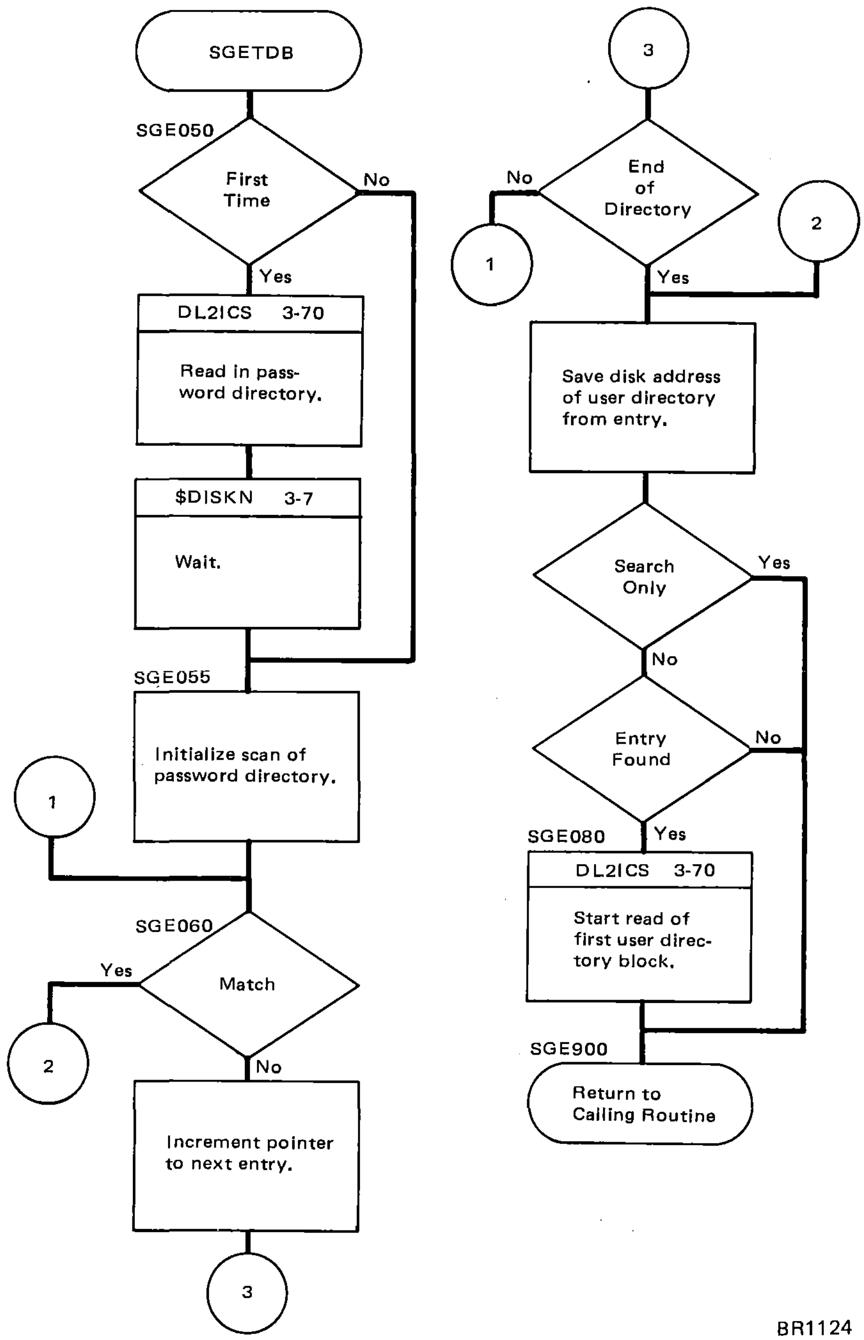


Figure 3-77. Search Password Directory Subroutine (SGETDB) Flowchart

Search Filename Directory Subroutine—SRCHFN (Figure 3-78)

- SRCHFN searches the filename directories for a specified filename.

SRCHFN searches a filename directory (USER, POOL, or **) for the filename in SMFNAM. The directory buffers and work areas are assumed to be available in TSMLES. The calling routine starts the disk operation to read the first directory block.

If the name is found, the address of the left byte of the entry is stored in SMUDEA and the SM1FNE bit of SMIND1 is set off. If the name is not found, the address where the next entry is placed is stored in SMUDEA and the SM1FNE bit of SMIND1 is set on. In both cases, SMUDBA contains the left byte address of the active block.

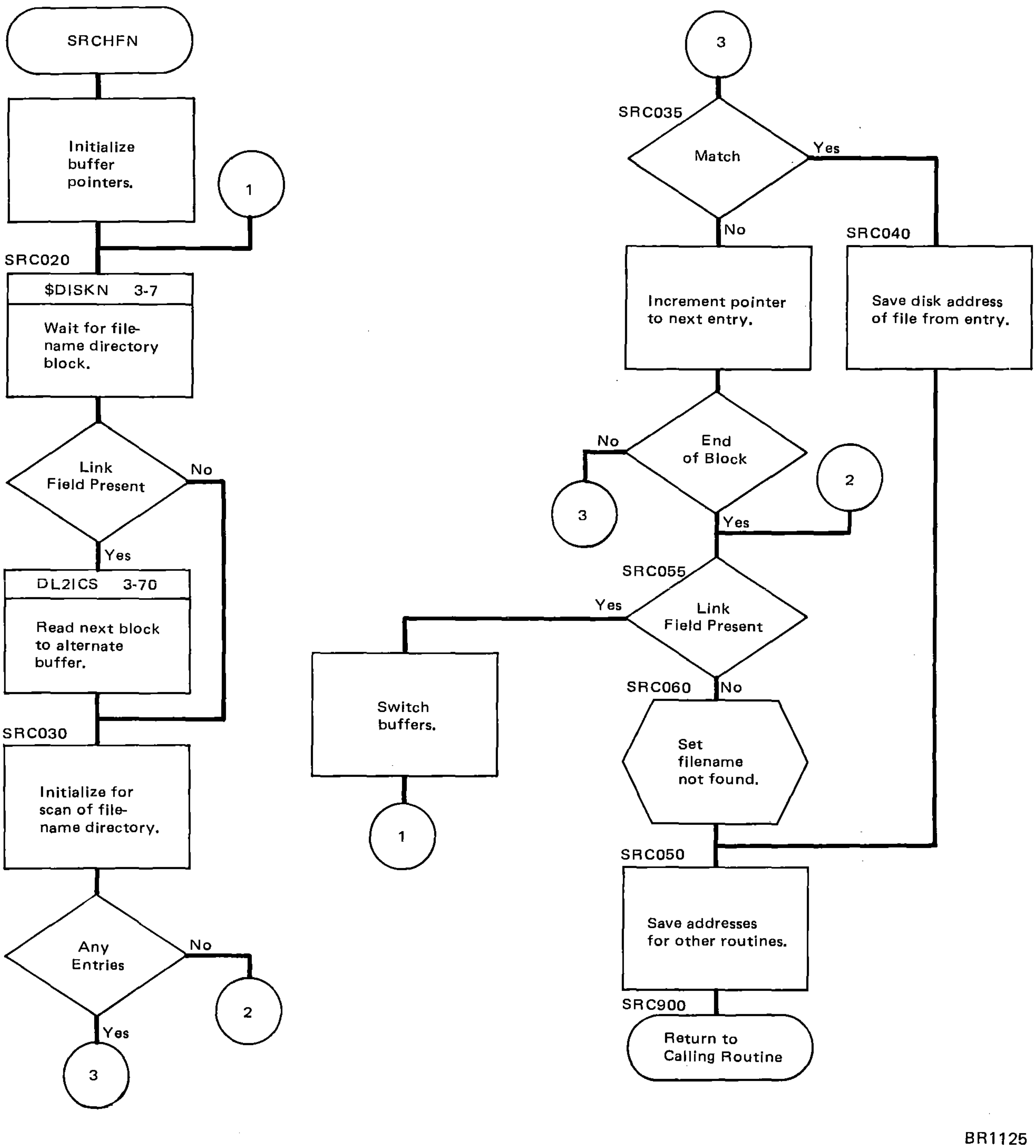


Figure 3-78. Search Filename Directory Subroutine (SRCHFN) Flowchart

Null Directory Entry Subroutine—STORIN (Figure 3-79)

- STORIN creates an entry in the null directory.

If the entry cannot be created, an indicator is set to note that the file library should be packed. If the null space is contiguous to that of any other entries in the directory, STORIN adjusts that directory entry to include the space.

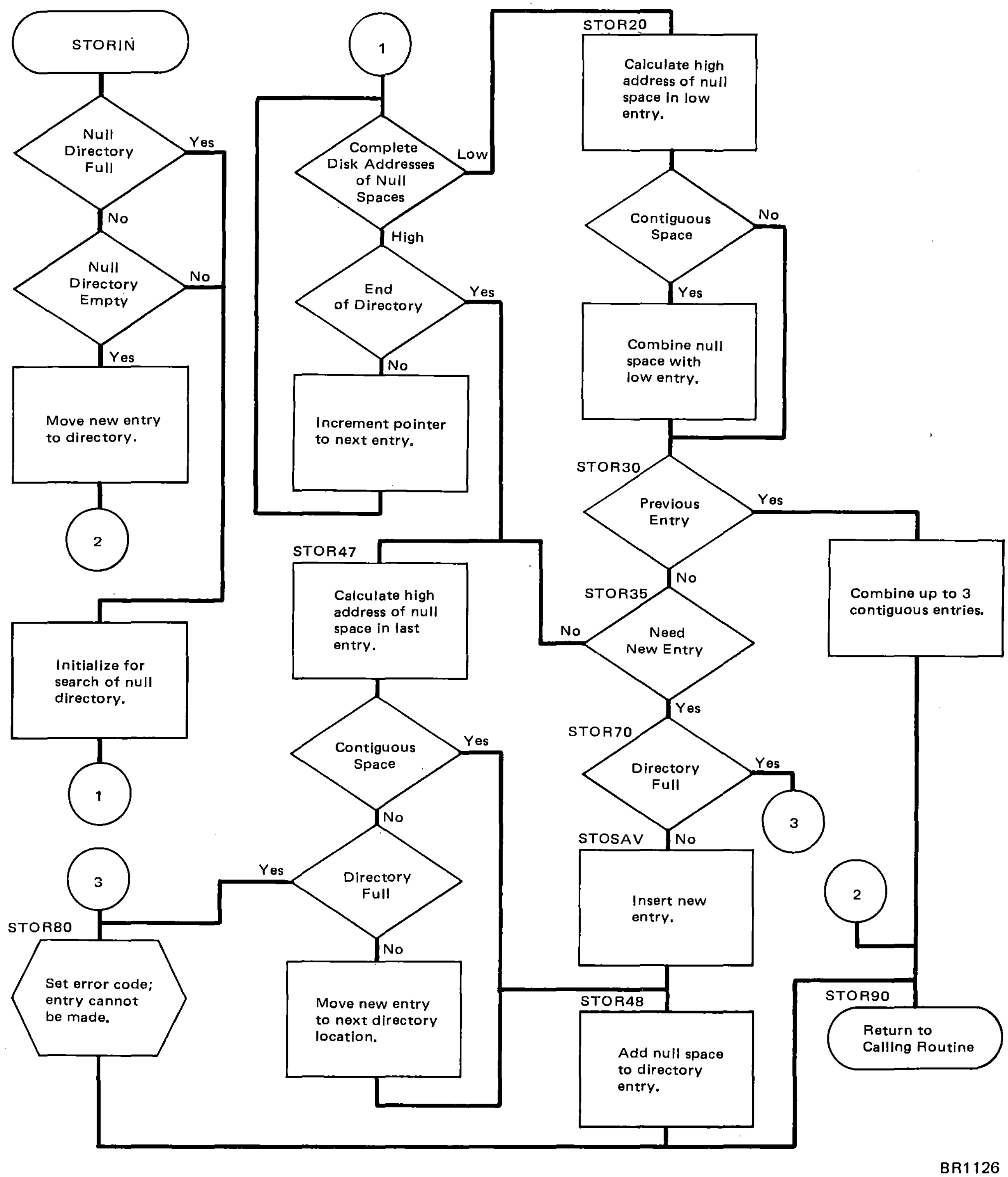


Figure 3-79. Null Directory Entry Subroutine (STORIN) Flowchart

Filename Directory Entry Subroutine—STUFID (Figure 3-80)

- STUFID inserts one entry in a filename directory. If the directory is full, STUFID tries to create a new block automatically.

STUFID adds a filename to a filename directory in the file library. If the directory is full, STUFID searches the null directory for a two-sector space to create a new directory block. If a space cannot be found, an error indicator is set in $CAERR and an exit to STUERR is taken. If the space is found, the new block is created. The write operation is started, to restore the affected directory block.

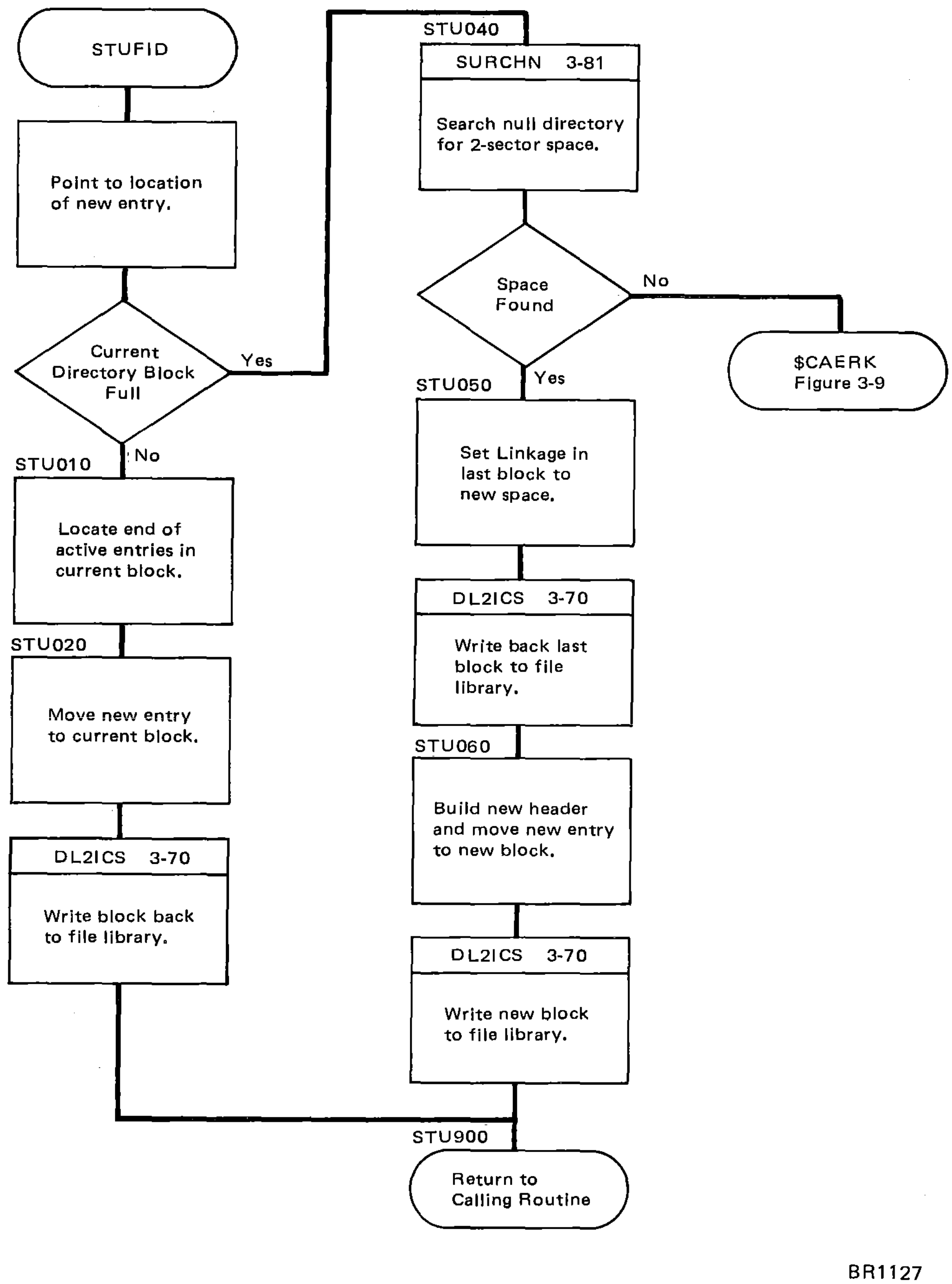


Figure 3-80. Filename Directory Insert Subroutine (STUFID) Flowchart

Search Null Directory Subroutine—SURCHN (Figure 3-81)

- SURCHN searches the null directory for an entry of at least N sectors in size, where N is specified by the calling routine in SMNSCT.

An attempt is made to find an entry in the directory of at least N sectors in length. If a directory entry is not large enough, it is added to SMNULT, which is an accumulated total of all available space for the file library. If the space required cannot be found, the calling program determines if the file library will be packed, by testing if SMNULT is equal to or greater than N. If the space is not found, a relative address of zero is returned in SMNDEA. If space is found, the relative address of the space is returned in SMNDEA.

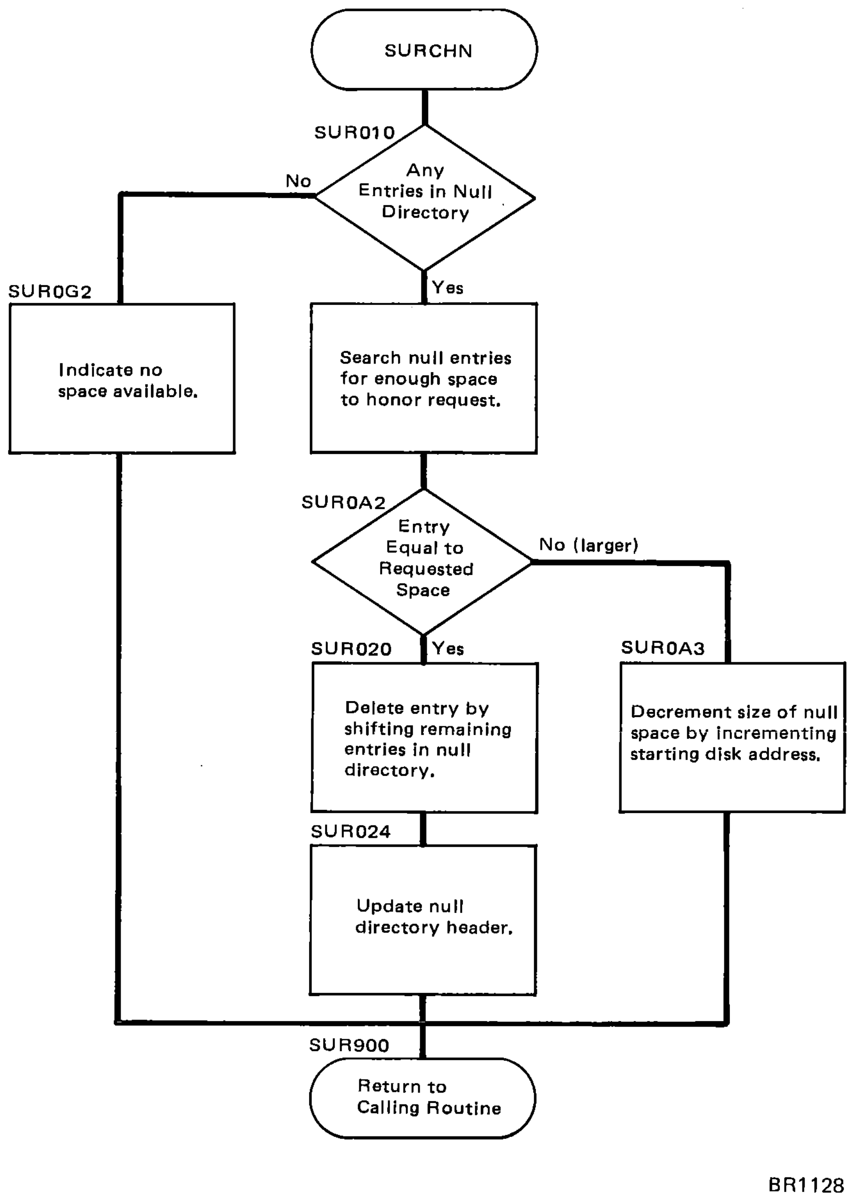


Figure 3-81. Search Null Directory Subroutine (SURCHN) Flowchart

Track Usage Mask Utility Subroutine—UTKUSE (Figure 3-83)

- UTKUSE tests and updates the track usage mask in the volume label (refer to Figure 5-9).
- The calling source module, assembled with UTKUSE, passes parameters via labels located within UTKUSE (Figure 3-82).
- Entries to UTKUSE are:

  UTKINP—Reads in volume label.
  UTKPRC—Bypass reading of volume label.

The calling source module can test for space as close to cylinder 10 as possible by moving UTKFLG to TKSYLN, causing the initial cylinder number to default to 10. The function code (in this case) moved to UTKTYP would be UTKTBF (Figure 3-82). This subroutine scans the track usage mask for the first available and consecutive space (TKSCYL).

Method Used to Displace into Track Usage Mask

The cylinder number divided by 4 equals the byte displacement into the track usage mask. The remainder is used to displace into a table of bit masks:

If Remainder Is	*Mask Is*
0	00000011
1	00001100
2	00110000
3	11000000

Label	Length	Description
TKSYLN	1	Initial cylinder number. If set to UTKLIM (X'FF'), the initial cylinder defaults to 10.
TKSCYL	1	Number of cylinders.
TKSADR	2	Core address of volume label.
TKSDSK	2	Disk address of volume label.
UTKTYP	1	Function codes: UTKSBN (X'3A')—Assign space. UTKSBF (X'3B')—Release space. UTKTBF (X'39')—Test for space available. UTKTBN (X'38')—Test for space not available.

BR1129

Figure 3-82. Parameters Passed to UTKUSE Subroutine

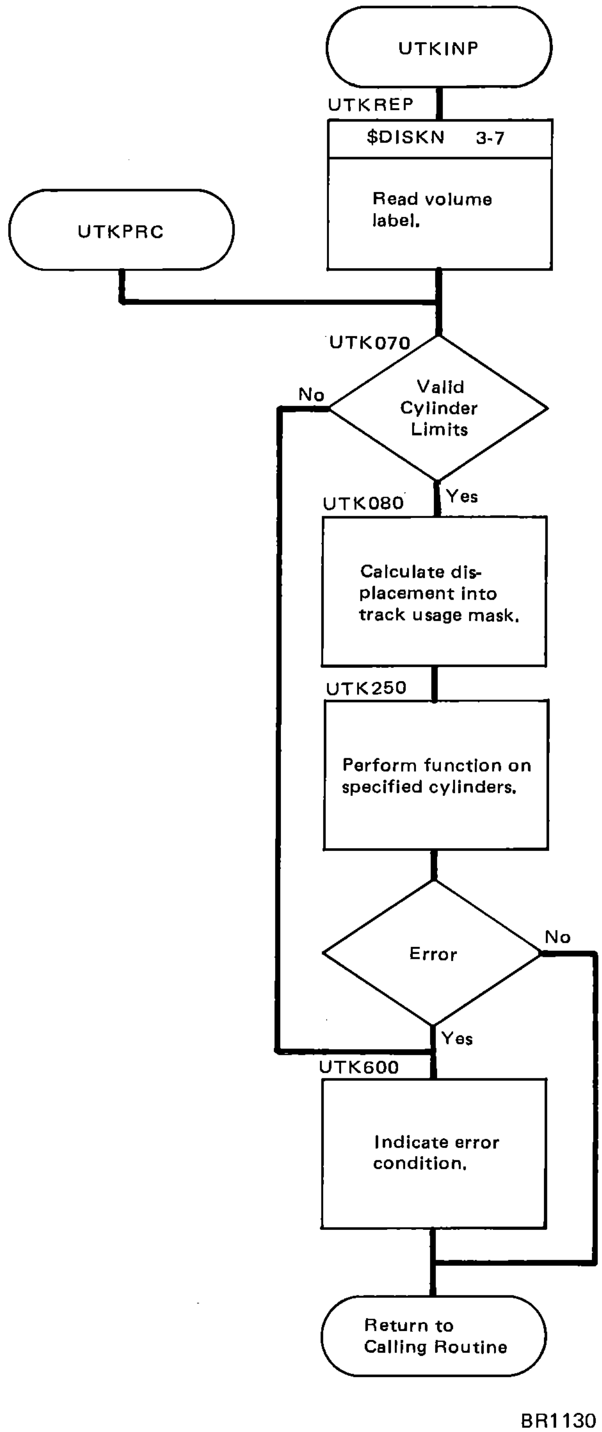


Figure 3-83. Track Usage Mask Utility Subroutine (UTKUSE) Flowchart

VTOC Utility Subroutine—UTVTOC (Figure 3-85)

- UTVTOC performs maintenance on the VTOC (refer to Figure 5-10) and volume label (refer to Figure 5-9). In version 1, modification 0, the ending disk address of the file is the address of the last track used, while in version 1, modification 1, the ending disk address of the file is the address of the next available track. After successful modification of any modification 0 file, UTVOC modifies the ending disk address of that file.
- The calling source module, assembled with UTVTOC, passes parameters via labels within UTVTOC (Figure 3-84). An assembly that contains UTVTOC also contains UTKUSE (Figure 3-83) to test and update the track usage mask in the volume label.
- Entries to UTVTOC are:

  UTVDEL—Delete a file.
  UTVEXP—Increase file size.
  UTVSHK—Decrease file size.
  UTVIST—Allocate a new file at specified location.
  UTVDFT—Allocate a new file as close to cylinder 10 as possible.
  UTVINF—Obtain information about file.

Refer to the functions in the description of the calling source module for functions provided by this subroutine.

Label	Length	Parameter Name	Note
TKSBFI	1	System files indicator	Same as in volume label, Figure 5-9.
TVSFIL	8 .	Filename	Not required for increasing, decreasing, or deleting system files.
TVSDSK	2	VTOC disk address	Physical disk address of VTOC index: X'0024'—R1 X'0025'—F1 X'0026'—R2 X'0027'—F2
TKSCYL	1	Number of cylinders	Used to increase, decrease, or allocate a file.
TKCYLN	1	Initial cylinder number	Used when allocating a file at a specific location.

BR1131

Figure 3-84. Parameters Passed to UTVTOC Subroutine

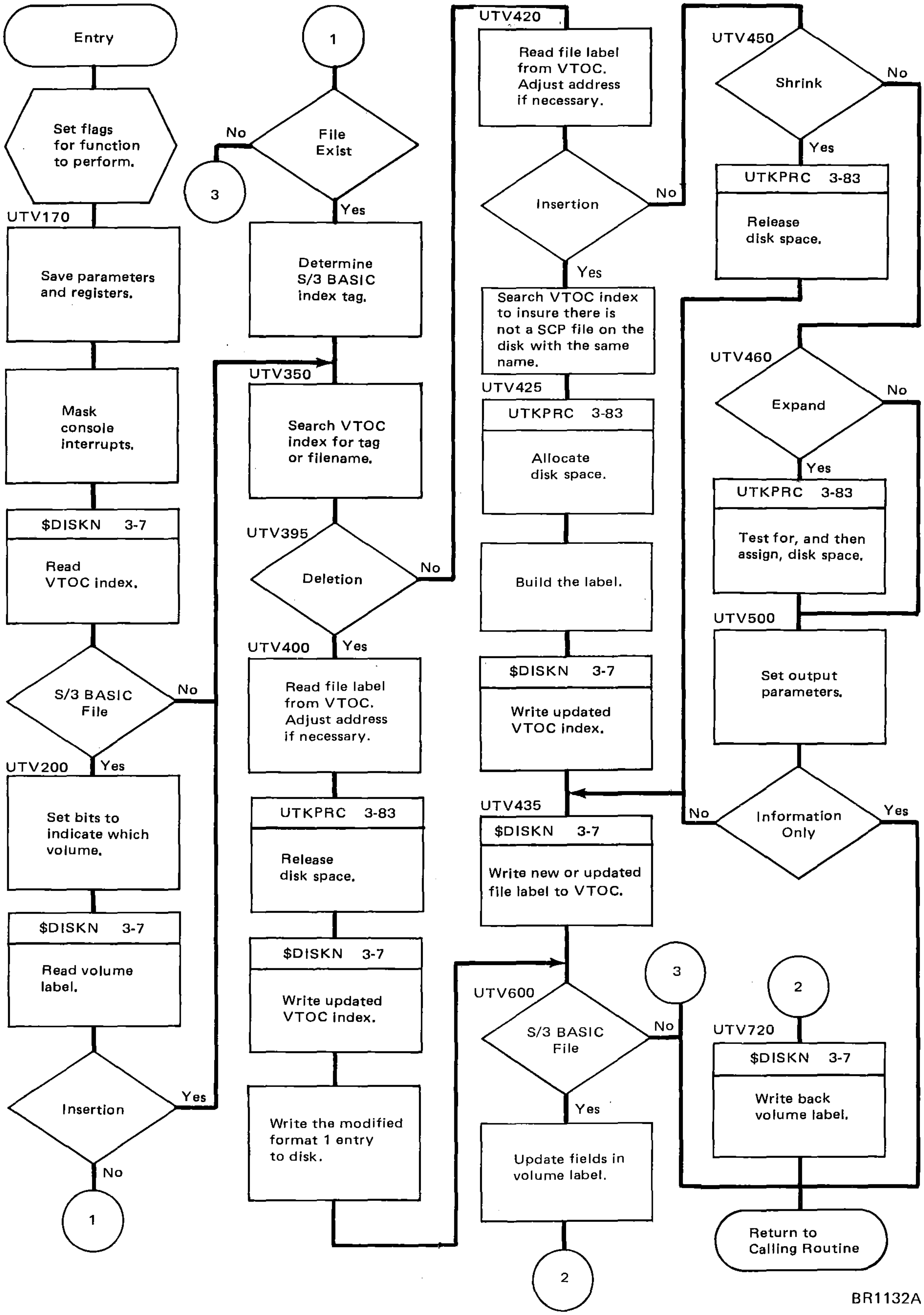


Figure 3-85. VTOC Utility Subroutine (UTVTOC) Flowchart

Pack File Library Subroutine—#SPACK (Figure 3-86)

- #SPACK reorganizes the file library and eliminates imbedded null sectors.

The null directory is referenced to determine where there are null sectors in the file library. All files and directories are moved up the disk to eliminate imbedded null sectors until all null sectors are located at the end of the file library. All pointers are updated to the new location. The sequence of the records in the file library is not changed.

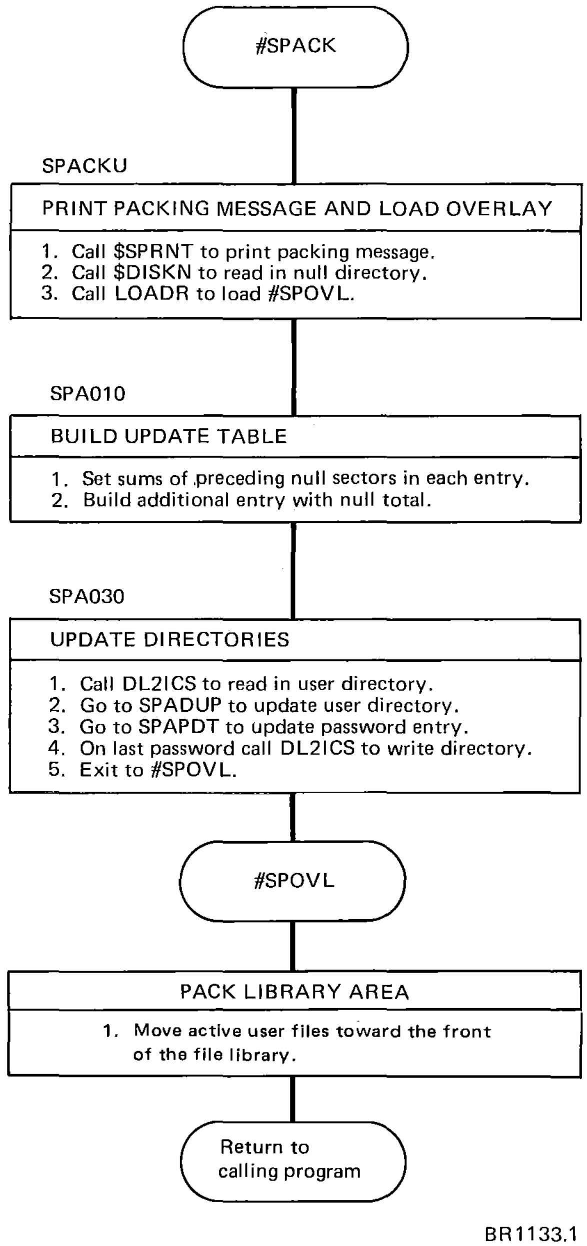


Figure 3-86. Pack File Library Subroutine (#SPACK) Flowchart (Part 1 of 2)

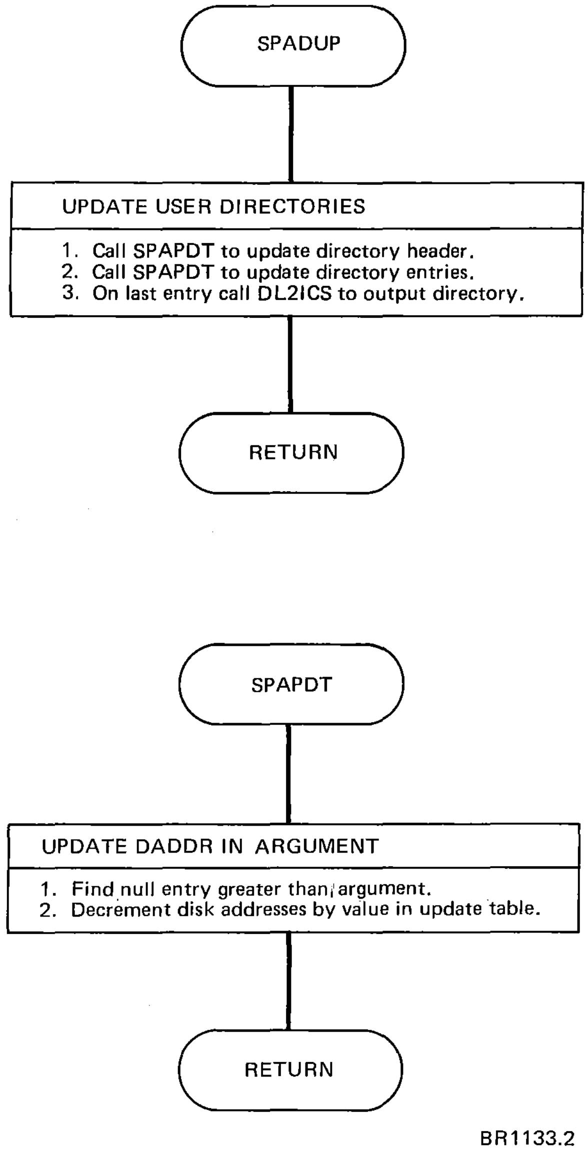


Figure 3-86. Pack File Library Subroutine (#SPACK) Flowchart (Part 2 of 2)

ALTERNATE-TRACK Utility Program—#UATRC (Figure 3-87)

- #UATRC tests, assigns, and unassigns alternate data tracks.
- The assembly of #UATRC contains this major source module:

  UATRCK—Mainline logic, Figure 3-87

Two alternatives are available in determining the suspected defective track: (1) specify a physical track, or (2) default to the tracks logged in the suspect track log in the volume-label sector. Either TEST or ASSIGN is valid in this case.

#UATRC performs one of four functions:

1. A suspected operative data track can be unconditionally flagged defective and assigned an alternate.
2. A suspected operative data track can be tested. The track is flagged defective and assigned an alternate based on the results of the test.
3. A flagged data track can be unconditionally restored to operative status. The alternate is unassigned.
4. A flagged data track can be tested. The track is restored to operative status based on the results of the test. The alternate is unassigned.

Data is transferred if it can be read without an unrecoverable error.

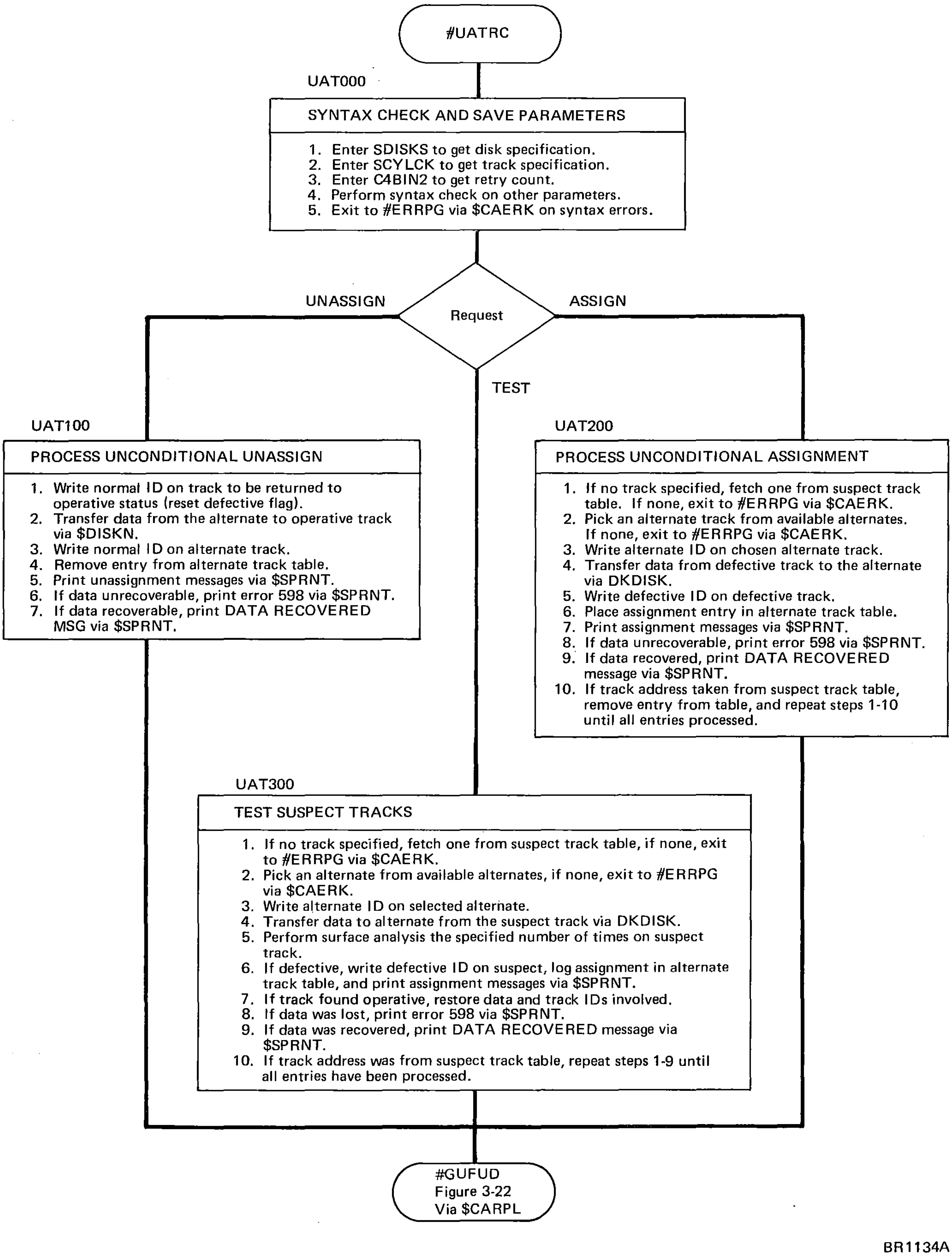


Figure 3-87. ALTERNATE-TRACK Utility Program (#UATRC) Flowchart

ASSIGN Utility Program—#UALLO (Figure 3-88)

- #UALLO allocates disk space for a system library file or a system work area.
- The assembly of #UALLO contains these major source modules:

  UALLOC–Mainline logic, Figure 3-88
  UTVTOC–VTOC subroutine, Figure 3-85
  UTKUSE–Track usage mask subroutine, Figure 3-83
  DL2ICS–Disk logical IOCS, Figure 3-70

Functions of #UALLO are:

1. Check the track usage mask in the volume label (Figure 5-9) for contiguous space.
2. Reset bits in the track usage mask that correspond to the tracks being allocated.
3. Update other required fields in the volume label.
4. Create an entry in the VTOC index and a label in the VTOC (refer to Figure 5-10).
5. Create the null directory, password directory, and * and ** directories for the files (refer to Figure 5-11).

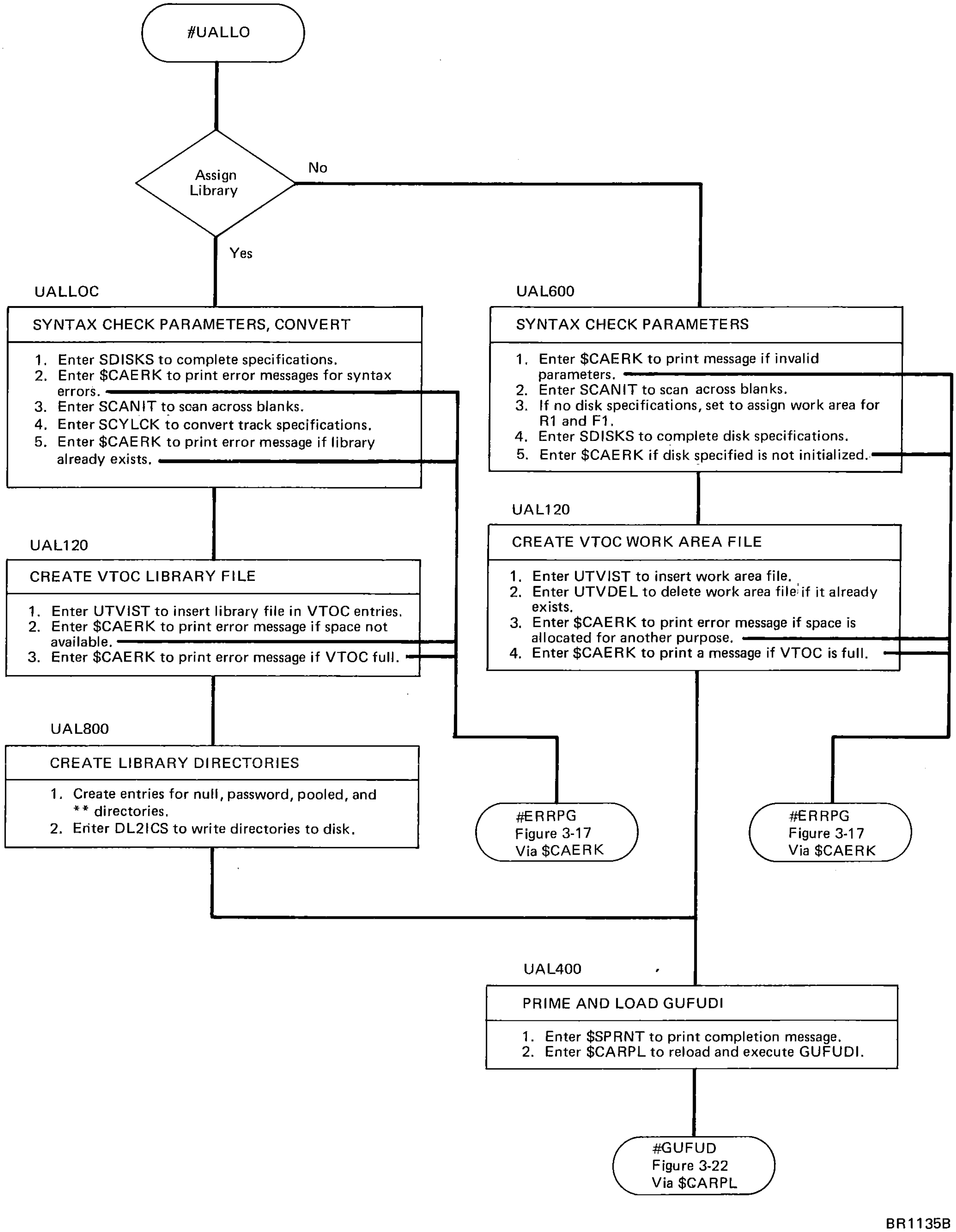


Figure 3-88. ASSIGN Utility Program (#UALLO) Flowchart

CONFIGURE Utility Program—#UCNFI (Figure 3-89)

- #UCNFI creates or modifies the configuration record on cylinder 0 (refer to Figure 5-3).
- The assembly of #UCNFI contains this major source module:

  UCNFIG—Mainline logic, Figure 3-89

Each device present in the new configuration record is issued a test command before the record is written on the IPL'd volume. When configuring up to 3D or 4D, any VTOC entries that exist (on the new packs being configured) for scratch files are deleted from the VTOC.

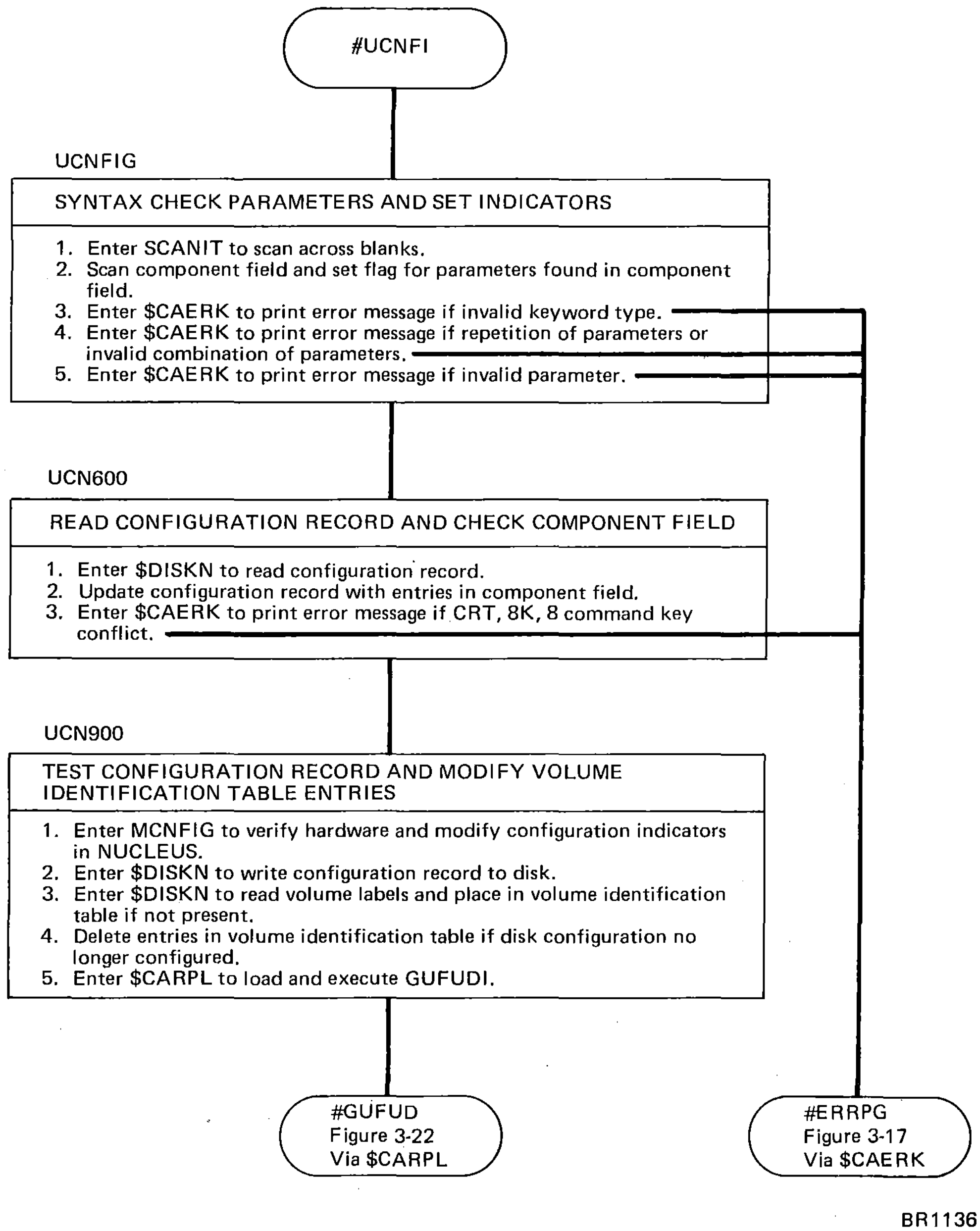


Figure 3-89. CONFIGURE Utility Program (#UCNFI) Flowchart

COPY File Utility Overlay—#UCPLI (Figure 3-90)

- #UCPLI copies a file defined by a label in the VTOC to another volume, or repositions the file on the same volume.
- #UCPLI is loaded by #UCDIS (Figure 3-91) when the command is either COPY-SYSTEM, COPY-LIBRARY, or COPY-HELPTEXT.
- The assembly of #UCPLI contains these major source modules:

  UCPLIB—Mainline logic, Figure 3-90
  DL2ICS—Disk logical IOCS, Figure 3-70

If #UCPLI copies the file to another volume, a new label is created in the VTOC and the volume label is updated. If the file is repositioned on the same volume, the existing label in the VTOC is deleted, a new label is created, and the volume label is updated. When a file is repositioned on the same volume, the old area is no longer accessible, except by a disk dump, even if the disk areas did not overlap.

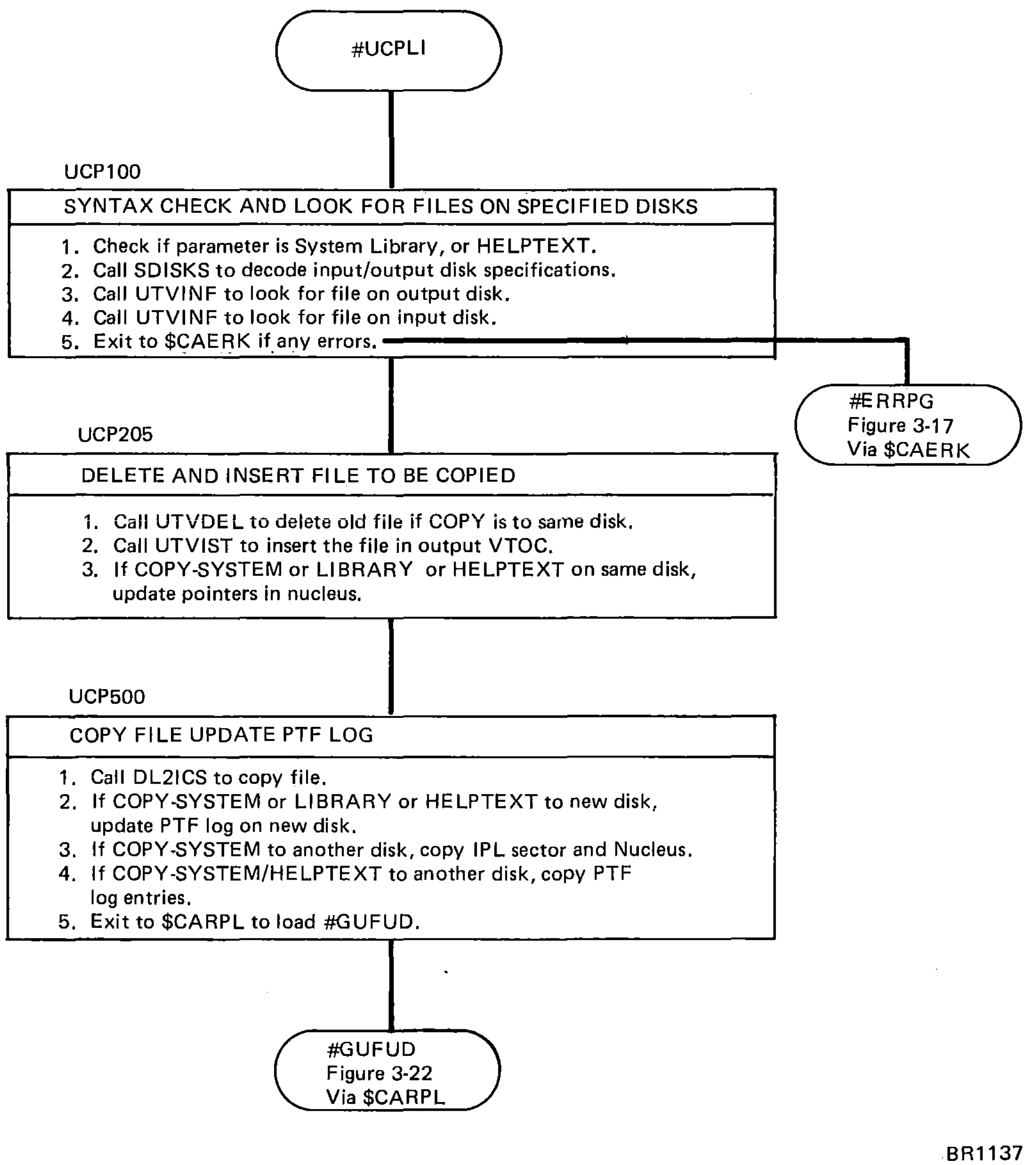


Figure 3-90. COPY File Utility Overlay (#UCPLI) Flowchart

COPY Volume Utility Program—#UCDIS (Figure 3-91)

- #UCDIS copies the entire contents of a disk volume to another volume.
- The assembly of #UCDIS contains these major source modules:

  UCDISK—Mainline logic, Figure 3-91
  UTKUSE—Track usage mask subroutine, Figure 3-82

The DISK parameter causes a track-for-track copy of the entire input volume to the output volume with the exception of the error log statistics and program protection sectors on cylinder 0 (refer to Figures 5-3 and 5-4). The volume-ID's of both volumes are verified prior to each copy operation.

The following parameters in the volume label (refer to Figure 5-9) are not copied:

1. Alternate tracks
2. Cylinder count
3. Suspected defective tracks
4. Volume label
5. Owner I/D
6. Track usage mask in its entirety (except in copying the contents of one 200-cylinder disk to another 200-cylinder disk)

On one disk read or write, 12 sectors are transferred on an 8k system and 24 sectors are transferred on a 12k or 16k system (12 sectors if 12k and CRT).

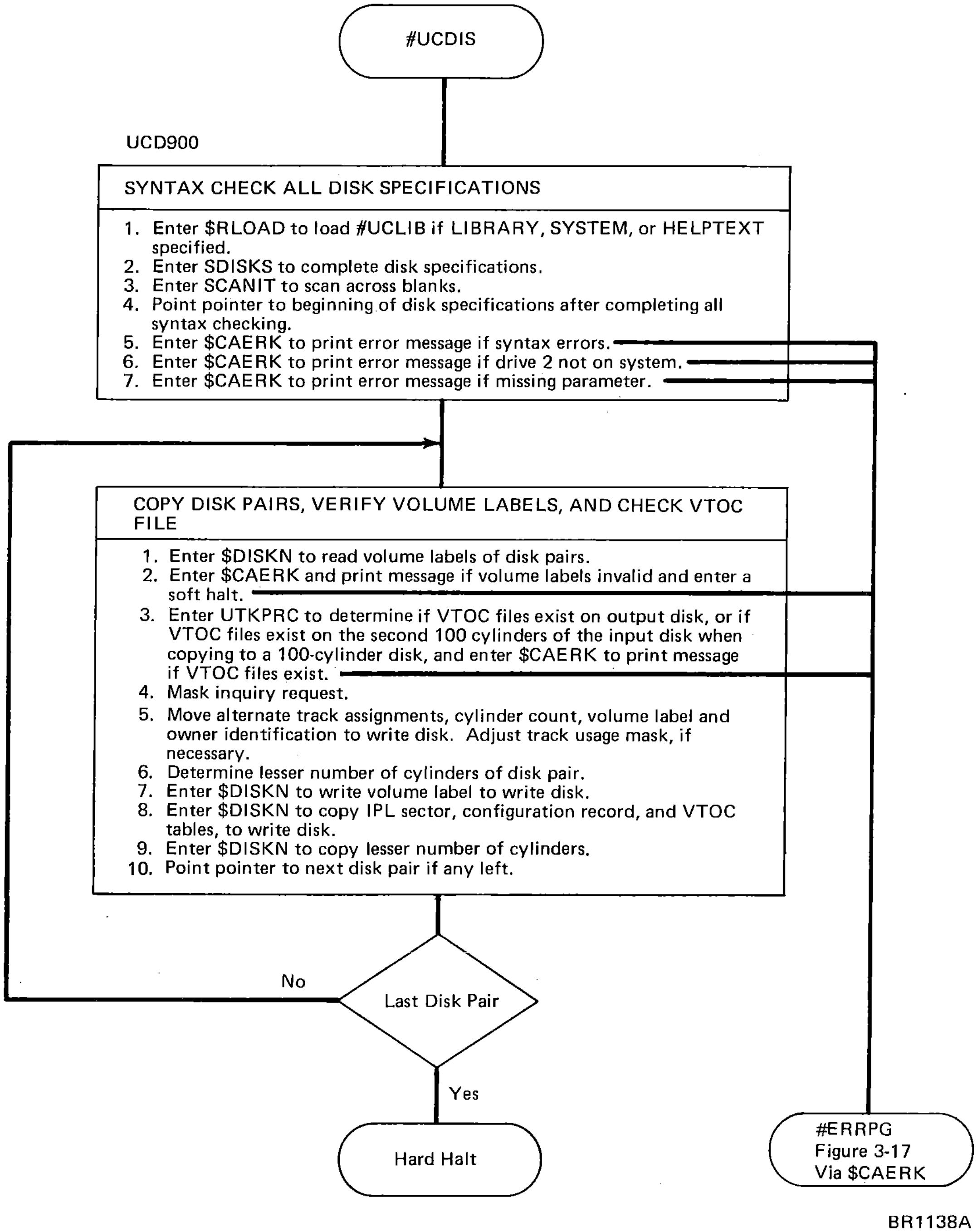


Figure 3-91. COPY Volume Utility Program (#UCDIS) Flowchart

EXPAND Utility Program—#UEXLI (Figure 3-92)

- #UEXLI changes the disk space allocated to a library file.
- The assembly of #UEXLI contains this major source module:

  UEXLIB—Mainline logic, Figure 3-92.

Functions of #UEXLI are:

1. Check the track usage mask in the volume label (refer to Figure 5-9) for available tracks when enlarging the library.
2. Set or reset bits in the track usage mask to reflect the change.
3. Update other required fields in the volume label; update the VTOC index and file labels.
4. Update the null directory for the library.

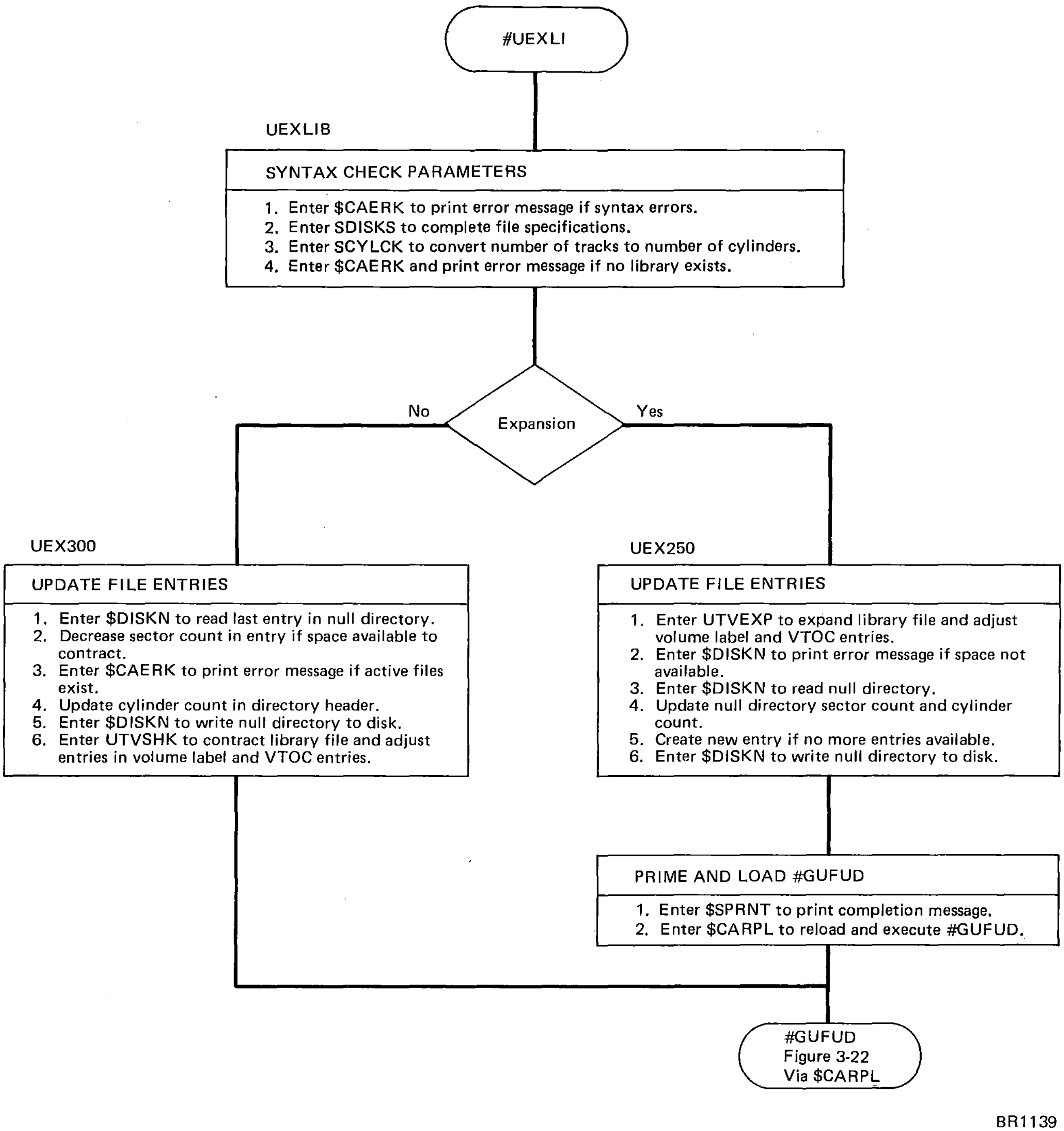


Figure 3-92. EXPAND Utility Program (#UEXLI) Flowchart

INITIALIZE Disk Utility Program—#UINIT (Figure 3-93)

- #UINIT formats and tests all tracks, including alternate tracks. Data tracks found defective are flagged and alternate tracks are assigned.
- The assembly of #UINIT contains this major source module:

  UINITL—Mainline logic, Figure 3-93

Functions of #UINIT are:

1. Clear the data field of all sectors to binary 0's.
2. Flag the addresses on tracks found defective and assign an alternate (refer to Figure 3-5)
3. Create the volume label (refer to Figure 5-9)
4. For PRIMARY initialization, write instructions on cylinder 0, head 0, sector 0, that will cause a hard halt if IPL is attempted on the volume.
5. Change the volume-ID with the CHANGE option.
6. Extend the initialization of an existing pack initialized to 103 cylinders.
7. Write VTOC index initial error logs.
8. For secondary initialization, delete all scratch file entries that may have been left in the VTOC by the co-resident disk system management programs.
9. Test for a valid system work area. If an invalid system work area is found, an error message results.

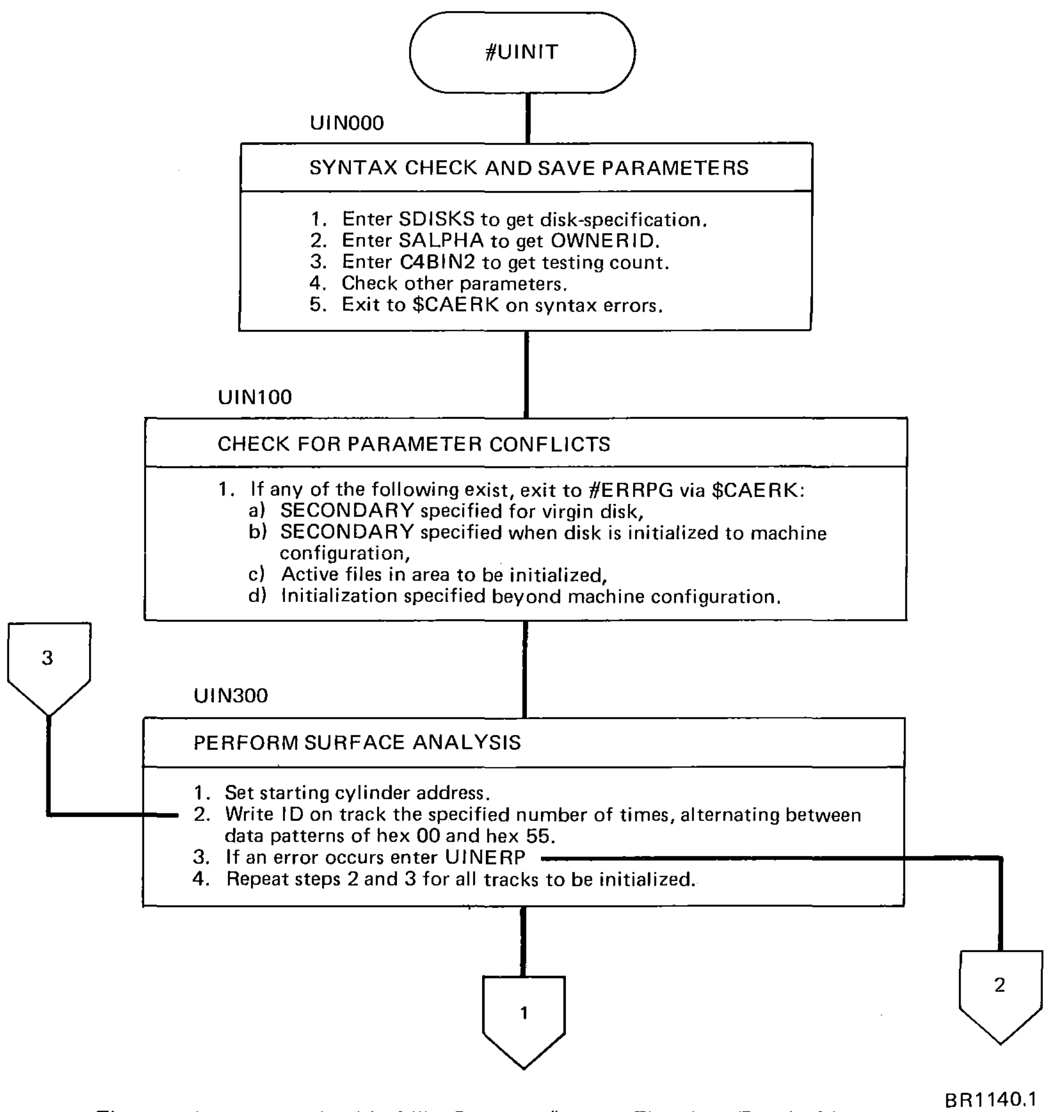


Figure 3-93. INITIALIZE Disk Utility Program (#UINIT) Flowchart (Part 1 of 2)

1

UINSEK

VERIFY CORRECT ID ON ALL TRACKS

1. Read back track ID from all tracks via DKDISK.
2. Check all defective ID's against alternate track table.
3. If incorrect ID is read, repeat block UIN300 once — If second failure, exit to #ERRPG via $CAERK.

UIN450

FORMAT CYLINDER ZERO

1. If primary initialization, perform the following:
   a) Set up initial volume label SCTR with VOLID and OWNERID,
   b) Place track usage mask, set on all bits beyond initialized area,
   c) Initialize VOL-LABEL indicators (including CE cylinder status),
   d) Write initial VTOC index and error log.
2. Update alternate track table.
3. Update track usage mask.
4. Write volume label sector via DKDISK.
5. MOUNT disk.
6. Enter SUTOBA to check work areas.

#GUFUD
Figure 3-22
Via $CARPL

2

UINERP

ASSIGN ALTERNATE TRACKS

1. Retest suspect track ten times; if no failure, return to calling routine.
2. If the track is not on the CE cylinder, select an alternate from available alternates; if none, exit to #ERRPG via $CAERK.
3. If the track is on the CE cylinder, do not attempt to assign an alternate to it.
4. Write defective ID on bad track.
5. Write alternate ID on selected alternate.
6. Place assignment entry in alternate track table
7. Print assignment messages via $SPRNT.
8. Return to process next track

3

BR1140.2B

Figure 3-93. INITIALIZE Disk Utility Program (#UINIT) Flowchart (Part 2 of 2)

PACK Utility Program—#UPACK (Figure 3-94)

- #UPACK analyzes the disk specification and loads #SPACK (Figure 3-86) to pack the library file.
- The assembly of #UPACK contains this major source module:

  UPACKU—Mainline logic, Figure 3-94.

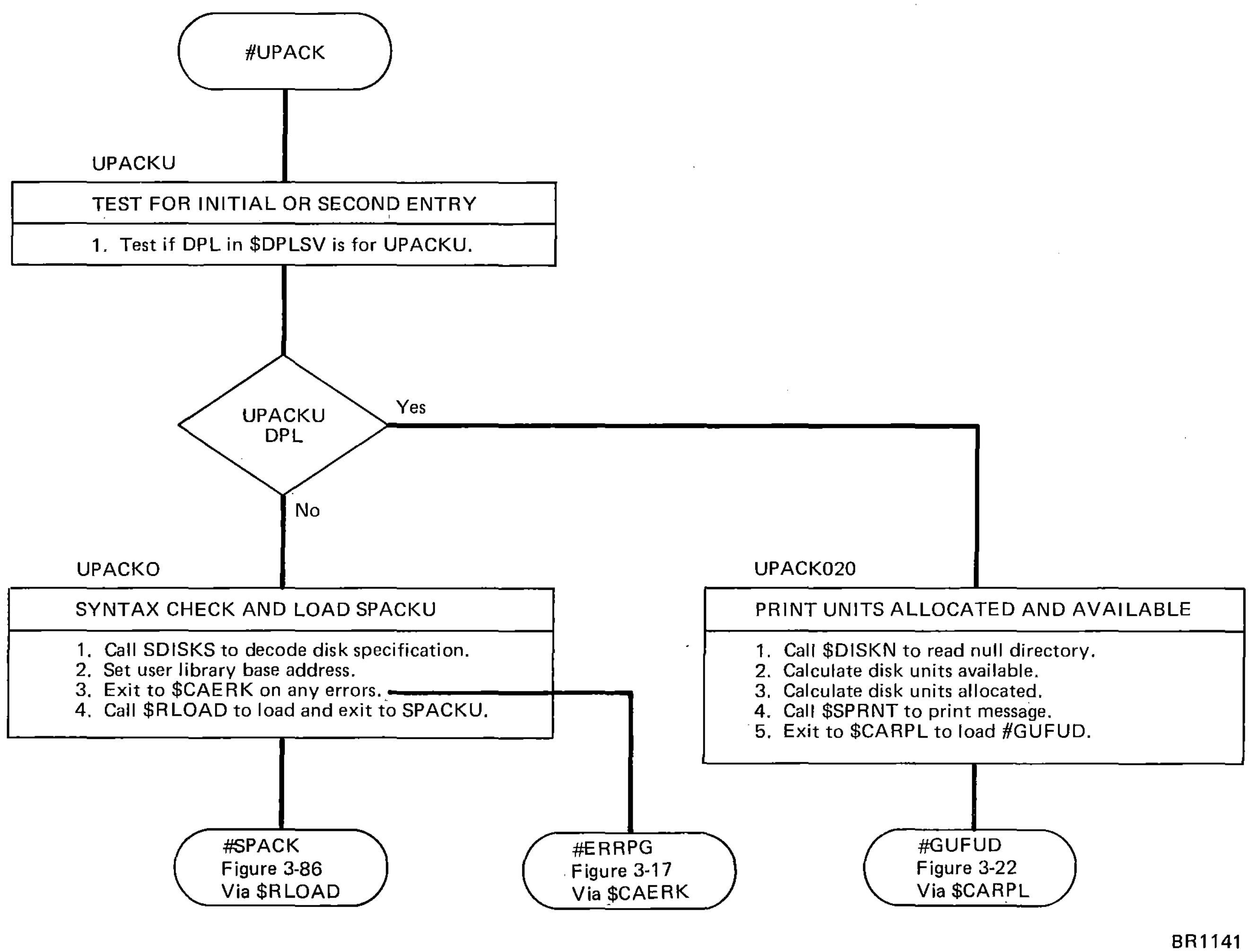


Figure 3-94. PACK Utility Program (#UPACK) Flowchart

PTF Utility Program—#UPTFI (Figure 3-95)

- #UPTFI applies program temporary fixes to components residing in the system program file or to the help text file.

- For PTF operating procedures, refer to "PTF Commands" in Section 6.

- The assembly of #UPTFI contains these major source modules:

  UPTFIX–Mainline logic, Figure 3-95
  DL2ICS–Disk logical IOCS, Figure 3-70

The PTF HDR (header) statement specifies the PTF identification, the disk to which the PTF is to be applied, and the disk from which the PTF will come, if it is from disk. Next, the PTF statement specifies the program name or help text component name and the system or help text release level. Then, one or more DATA statements are entered, specifying the core address and data of the patch or patches to be made. Multiple PTF's may be applied by specifying a new PTF statement and DATA statement(s) for the program or help text component to be fixed.

The components are updated when the PTF END statement is issued. This update consists of:

1. Modifying the specified locations within the component(s).
2. Updating the PTF log (refer to Figure 5-2).
3. Deleting the system work area (if there is one) to force the modification of a program in the system work file.

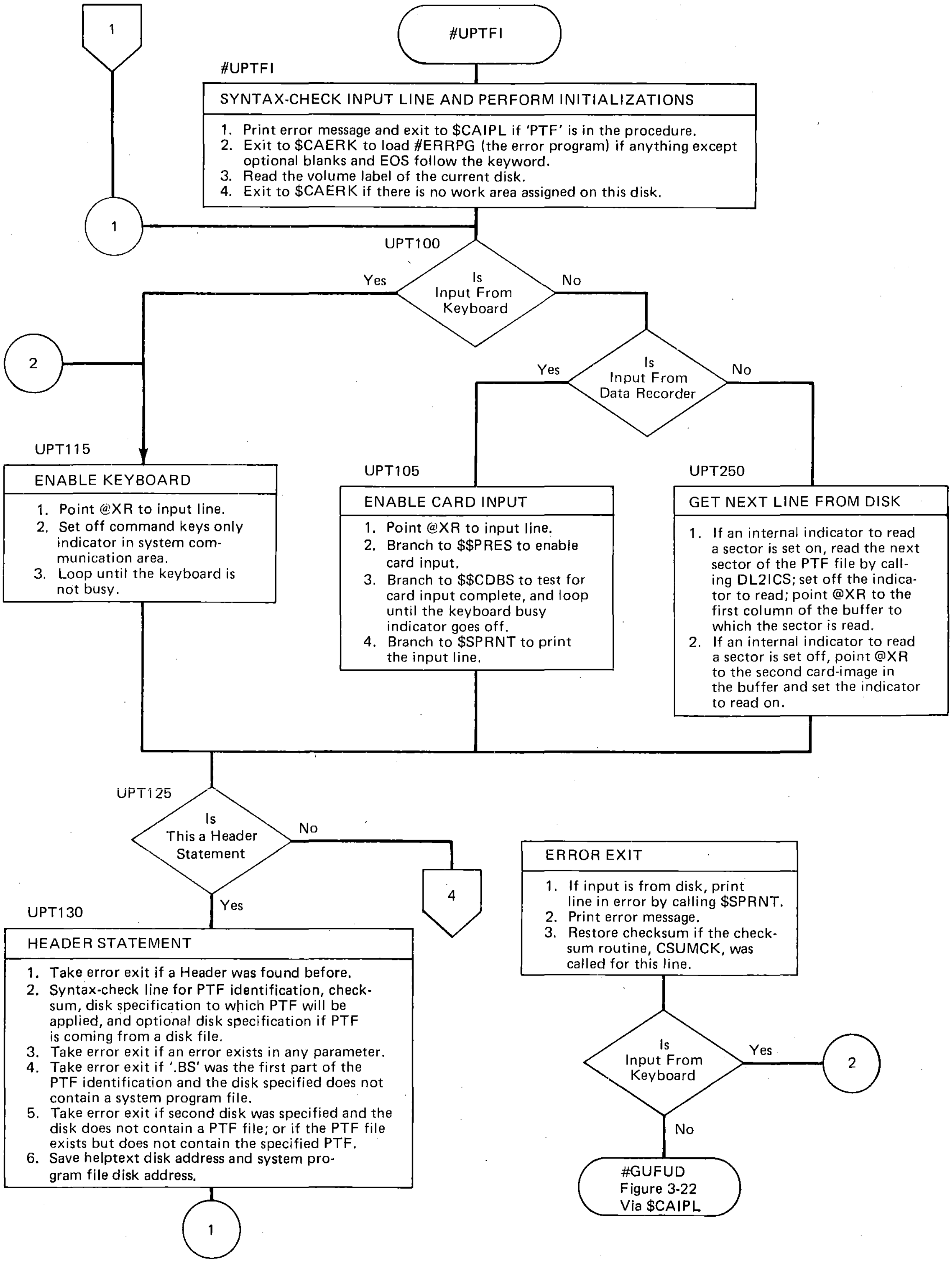


Figure 3-95. PTF Utility Program (#UPTFI) Flowchart (Part 1 of 2)

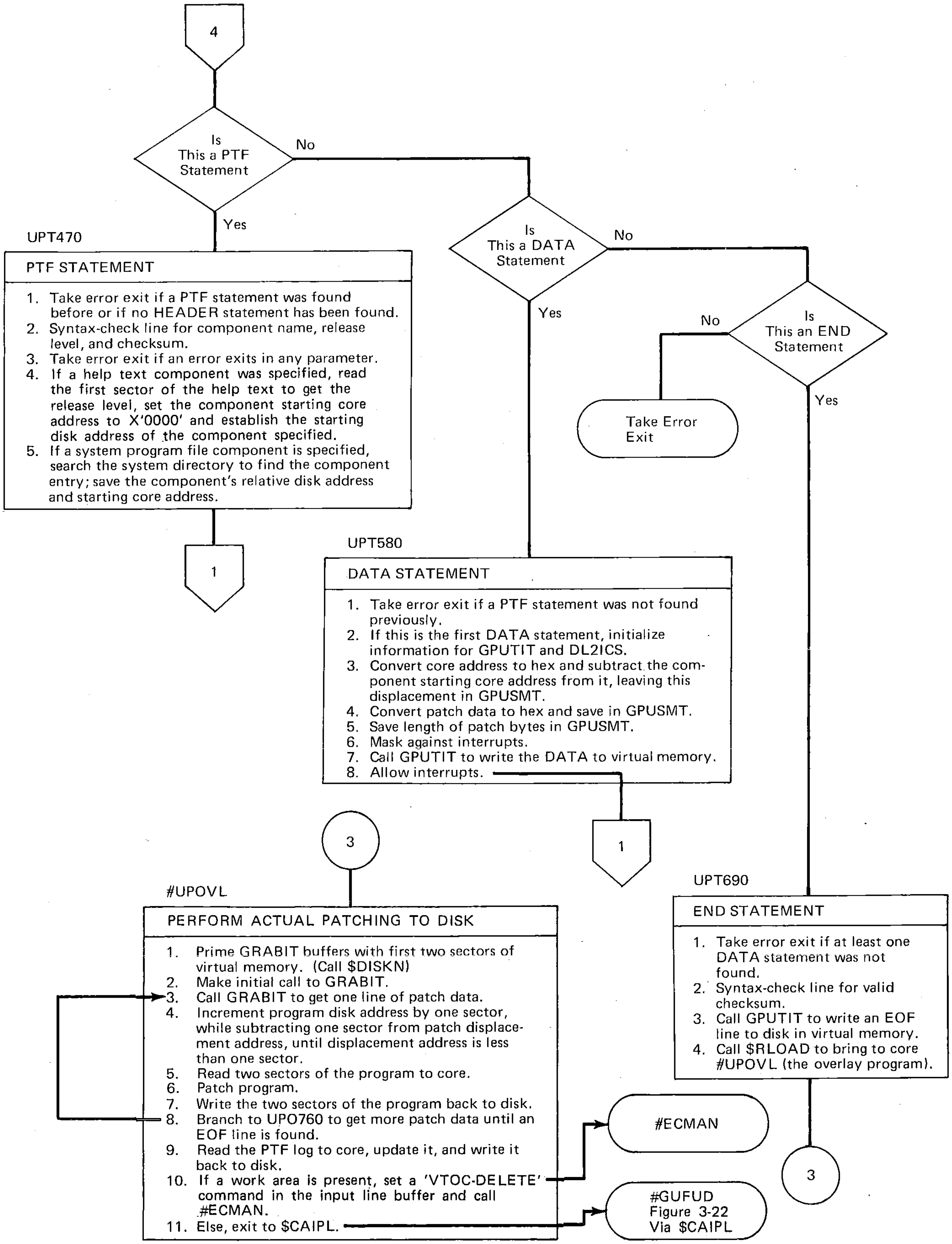


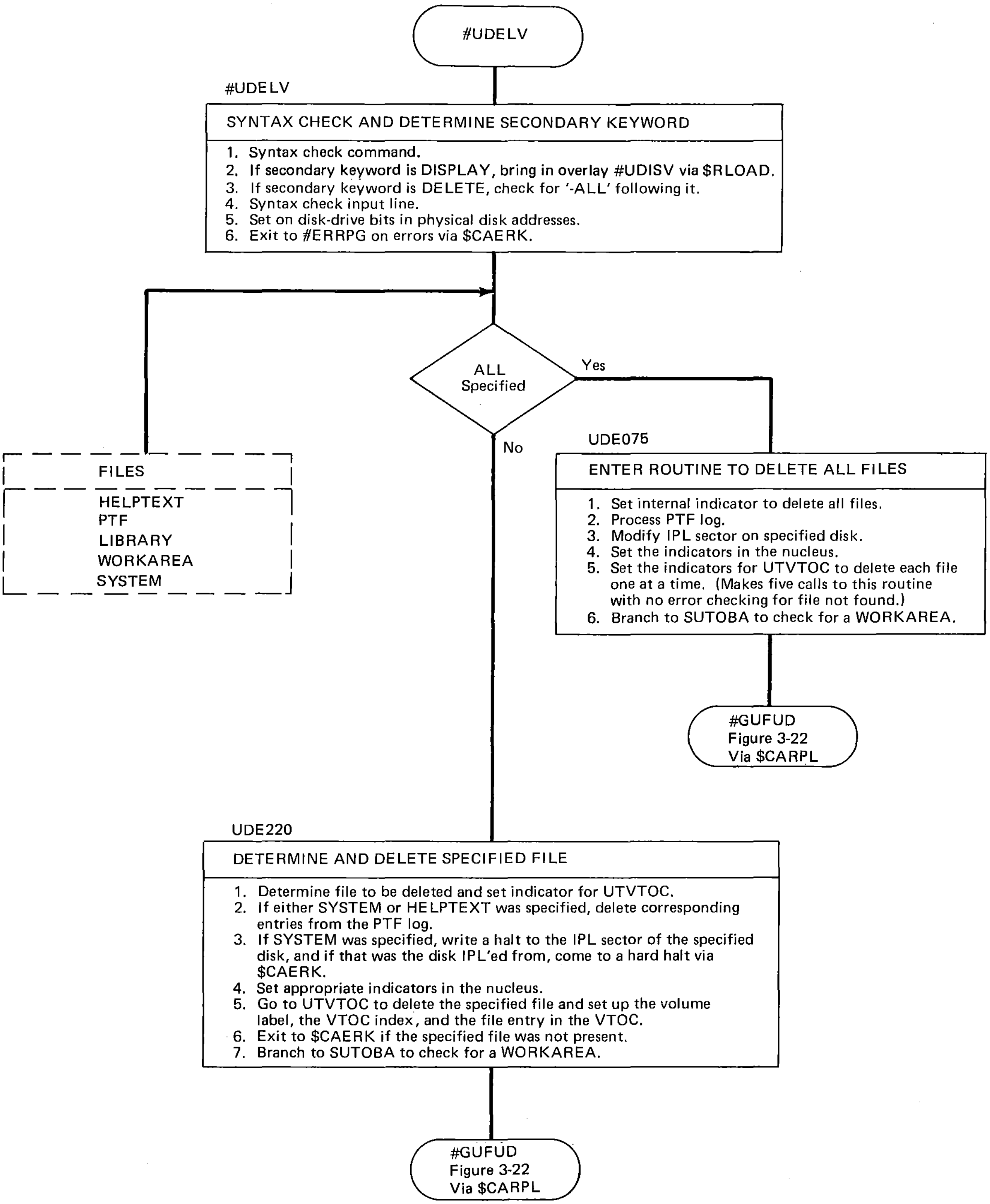
Figure 3-95. PTF Utility Program (#UPTFI) Flowchart (Part 2 of 2)

VTOC-DELETE Utility Program—#UDELV (Figure 3-96)

- #UDELV deletes System/3 BASIC files from a specified volume. If VTOC-DISPLAY is specified, the program #UDISV (Figure 3-97) is loaded.

Functions of #UDELV are:

1. Delete the file label in the VTOC and the VTOC index entry (refer to Figure 5-10).
2. Set bits in the track usage mask in the volume label (refer to Figure 5-9), releasing the tracks occupied by the file.
3. Update other required fields in the volume label.
4. Update required fields in the nucleus.



BR1143A

Figure 3-96. VTOC–DELETE Utility Program (#UDELV) Flowchart

VTOC-DISPLAY Utility Overlay—#UDISV (Figure 3-97)

- #UDISV displays VTOC label information from a specified volume on the system printer.
- #UDISV is loaded by #UDELV when the command is VTOC-DISPLAY.
- The assembly of #UDISV contains this major source module:

  UDISVT—Mainline logic, Figure 3-97

#UDISV displays the following information:

1. Volume-ID.
2. Owner identification.
3. Alternate track assignments.
4. Filename, starting address, size, and file type for each file in the VTOC.
5. Initialized disk size.
6. Number of unused VTOC file entries.
7. Co-resident disk system management program files (starting address, size, and file type).

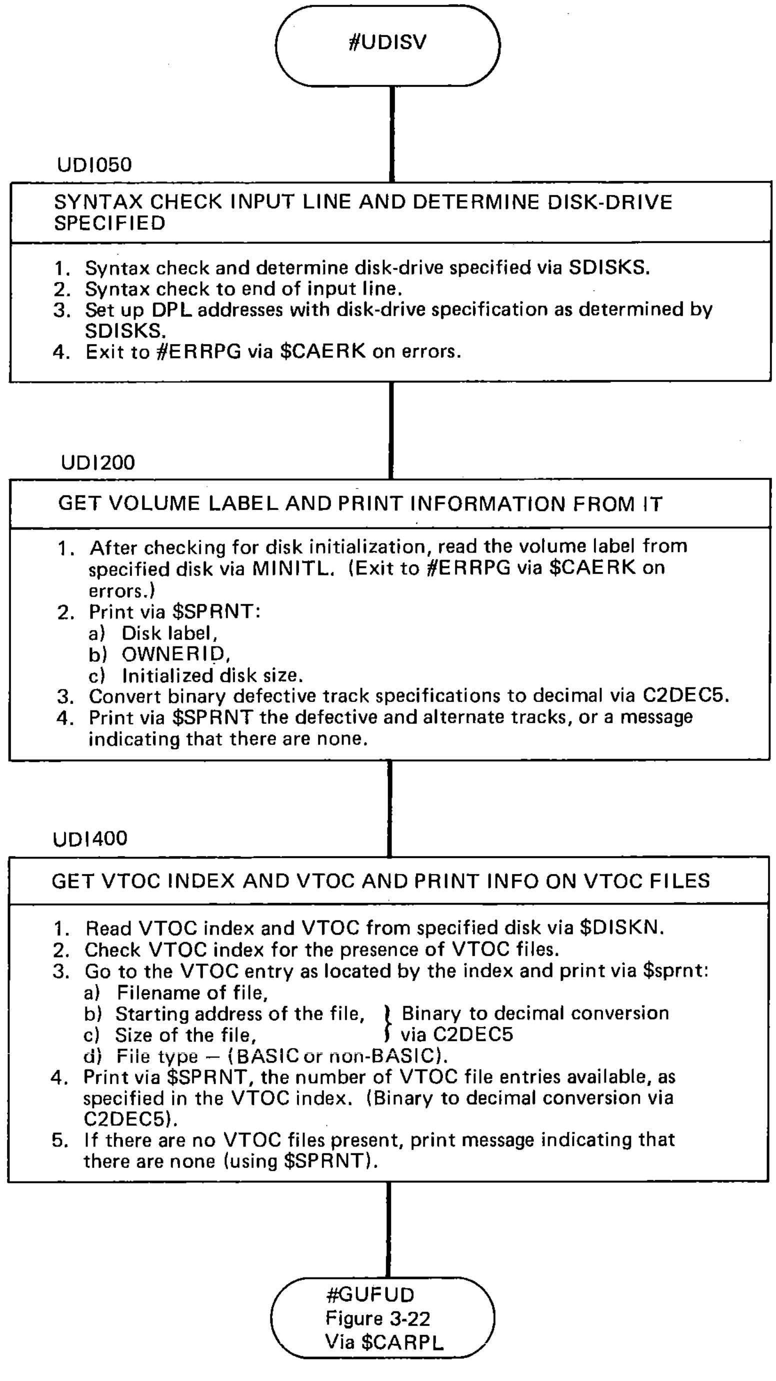


Figure 3-97. VTOC-DISPLAY Utility Program (#UDISV) Flowchart

MAINTENANCE UTILITIES

Refer to "Maintenance Utility Aid Program—#ZUTMO" in Section 6 for maintenance utility aid operating procedures.

Maintenance Utility Monitor—#ZUTMO (Figure 3-99)

- #ZUTMO performs these service aid functions:

  CD—Core dump
  DD—Disk dump
  VM—Virtual memory dump (accomplished by #ZDUMP, Figure 3-101)
  CP—Core patch
  DP—Disk patch
  DC—Disk compare
  DW—Disk write (copy sector)
  H—Restore core and halt
  R—Restore core and return to #GUFUD
  T—Reverse the program load trace option
  M—Library map and test

- The assembly of #ZUTMO contains these major source modules:

  ZUTMON—Monitor and all functions except CP and M, Figure 3-99
  DL2ICS—Disk logical IOCS, Figure 3-70

#ZUTMO is loaded by doing a system stop, system reset, and a system start, or by branching to core address 0. A branch to NPAUSD in the system nucleus is made at location 0 to save core. The CD and CP functions reference the saved core via the DPL used by NPAUSD. If the high limit of saved core is on a 4k boundary, it is assumed this address is the end-of-core address. Figure 3-98 is a sample core map showing #ZUTMO.

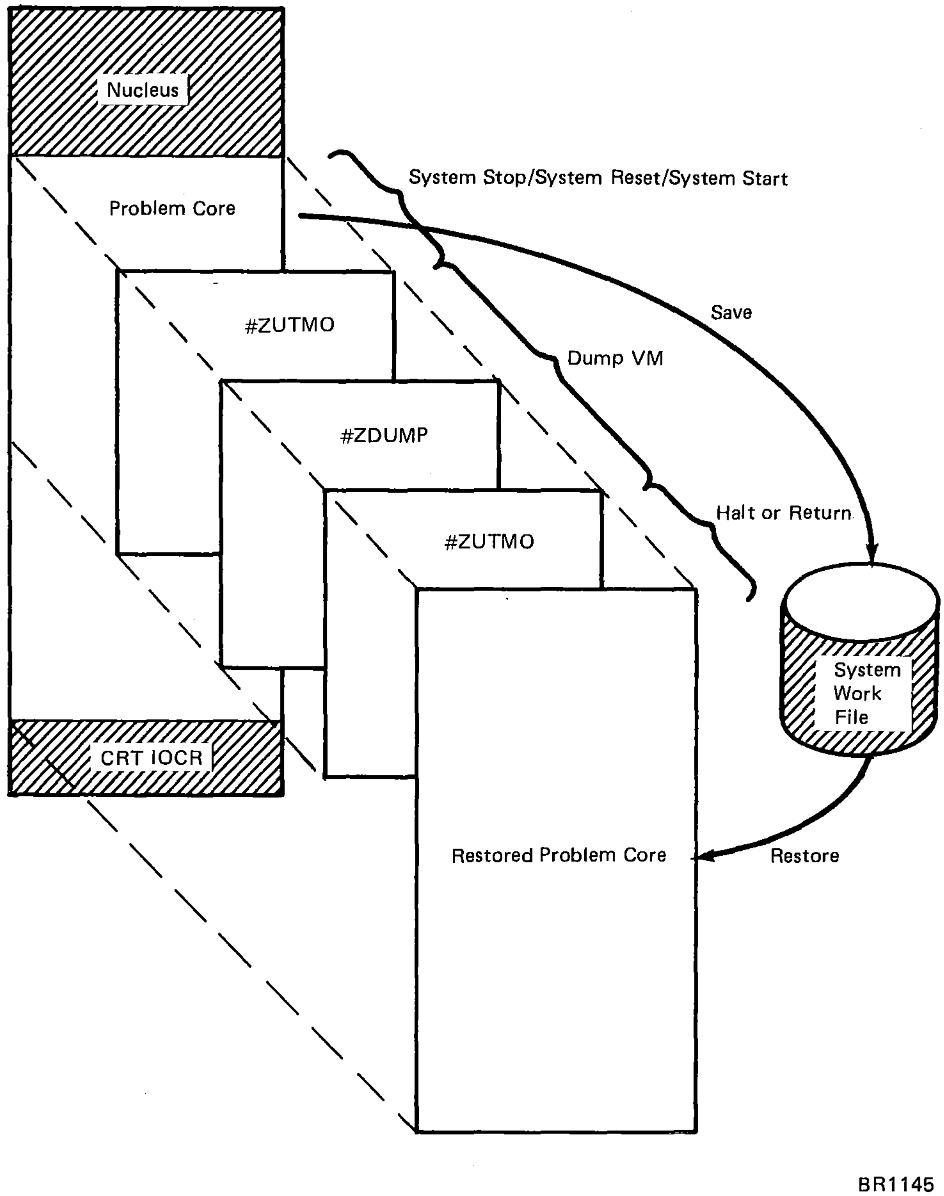
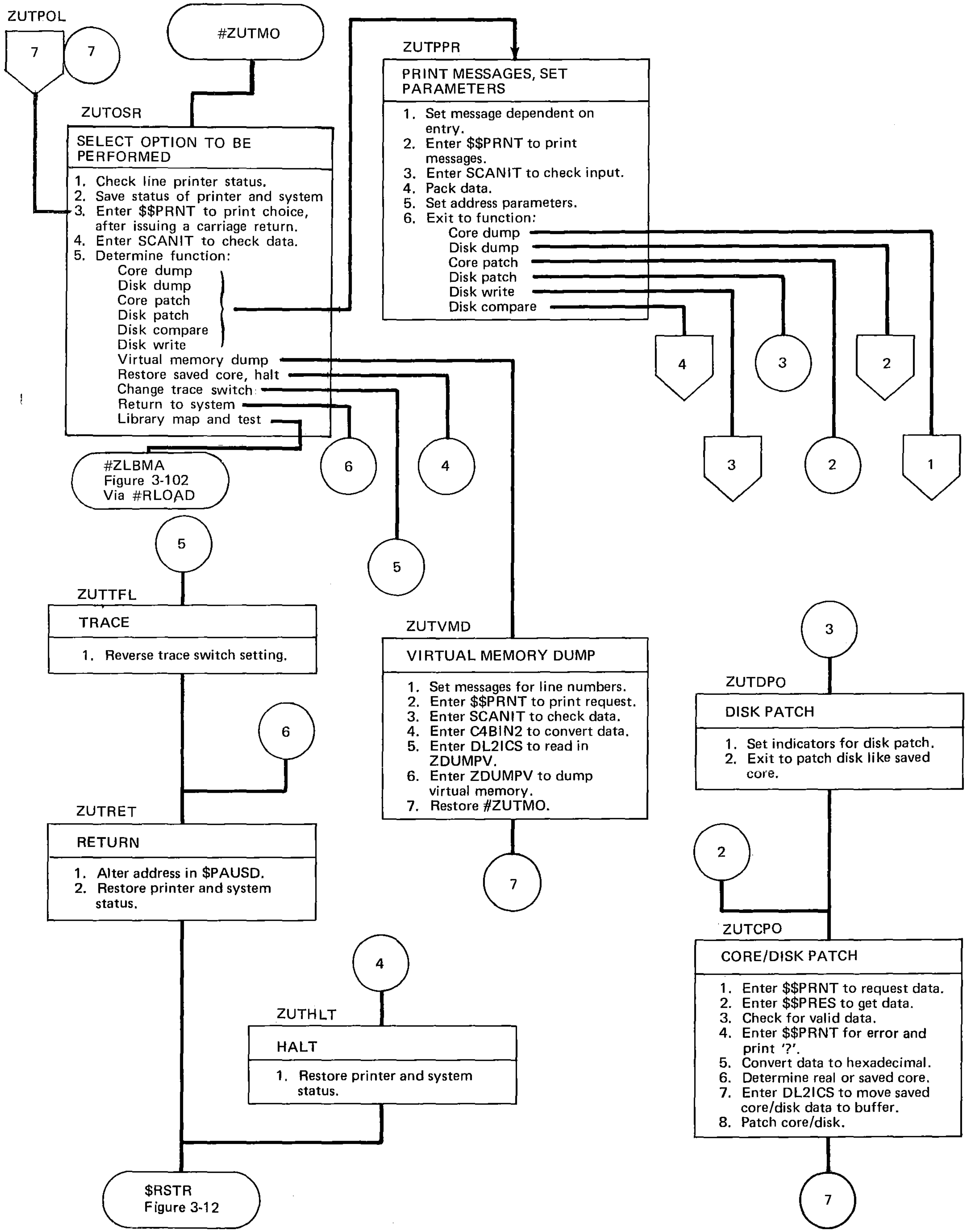
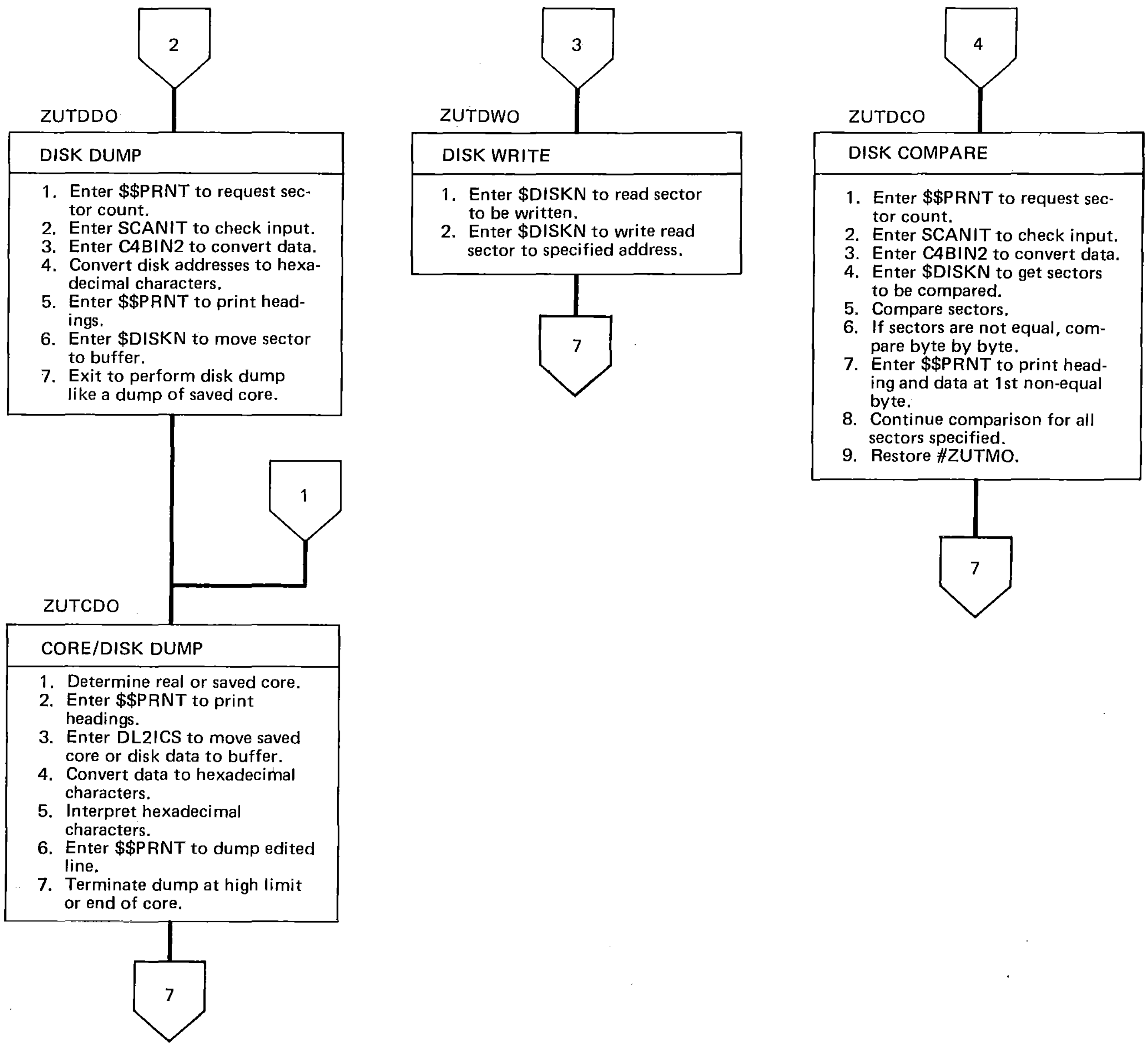


Figure 3-98. Maintenance Utility Core Map, Example



BR1146.1B

Figure 3-99. **Maintenance Utility Monitor** (#ZUTMO) Flowchart (Part 1 of 2)



BR1146.2A

Figure 3-99. Maintenance Utility Monitor (#ZUTMO) Flowchart (Part 2 of 2)

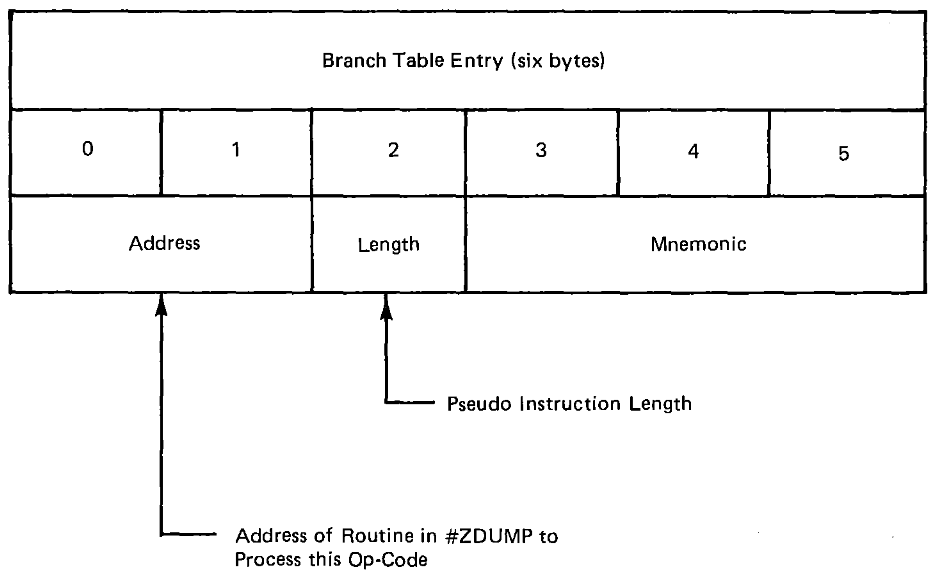
VM Dump Overlay—#ZDUMP (Figure 3-101)

- #ZDUMP interprets and lists the pseudo machine code in virtual memory.
- The assembly of #ZDUMP contains these major source modules:

  ZDUMPV—Mainline logic, Figure 3-101
  DL4ICS—System work file IOCS, Figure 3-70
- #ZDUMP is loaded by the maintenance utility monitor (Figure 3-99). The return entry in #ZUTMO reloads the overlay portion of #ZUTMO after completing the virtual-memory dump (Figure 3-98).

A validity check is made on the pseudo op-codes although it is assumed the pseudo machine code is correct in virtual memory. Output is on the system printer, one line for each pseudo machine instruction. Each line contains the virtual-memory address, a mnemonic op-code and operand, and the actual hexadecimal pseudo instruction. The statement and image header op-codes also generate the BASIC statement line number in the output line.

#ZDUMP contains a pseudo op-code branch table (Figure 3-100). The actual op-code indexes the table.



Note: The table contains one entry for each pseudo op-code.

BR1147A

Figure 3-100. #ZDUMP Branch Table

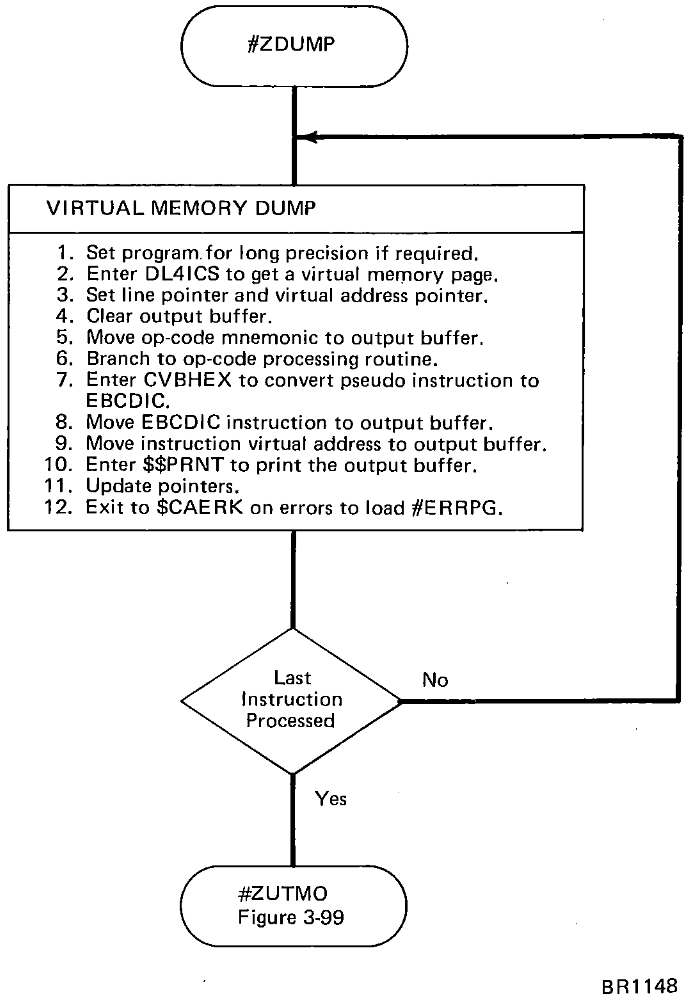


Figure 3-101. VM Dump Overlay (#ZDUMP) Flowchart

Library Mapping Overlays (Figure 3-102)

- These maintenance utility overlays map and test the directories and files in the File Library.

- There are five overlays in this group:

  #ZLBMA–Mainline entry routine. This routine calls one of the option overlays.
  #ZL1MA–Option 1 overlay. Maps null and password directories.
  #ZL2MA–Option 2 overlay. Maps a specified password.
  #ZL3MA–Option 3 overlay (part 1). } Maps the entire library.
  #ZLVRL–Option 3 overlay (part 2). }

- #ZLBMA is loaded by the maintenance utility monitor (Figure 3-99). Each option reloads the maintenance utility monitor after completion of the option or if the option overlay is interrupted.

- All output (maps and error messages) is displayed on the matrix printer.

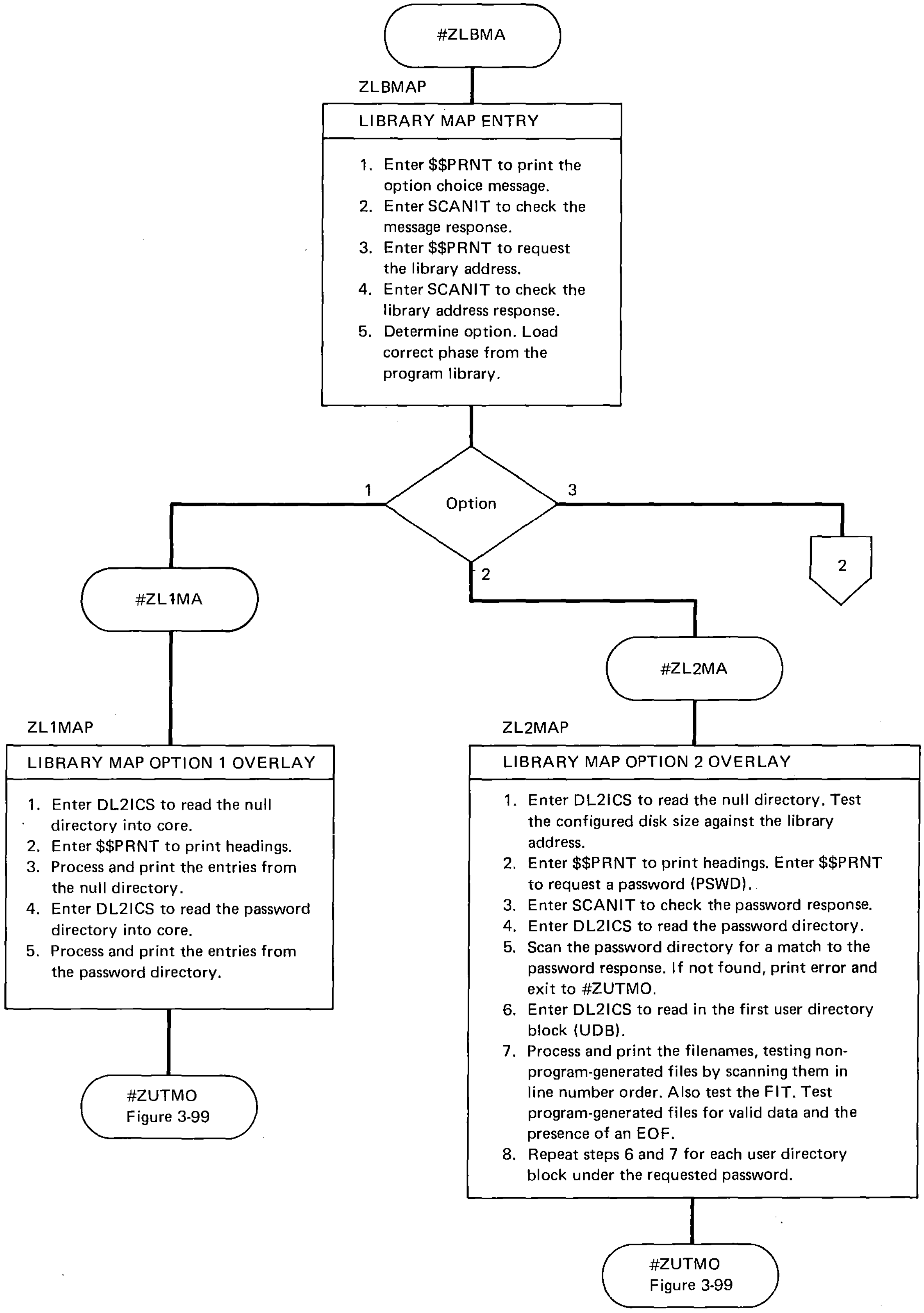
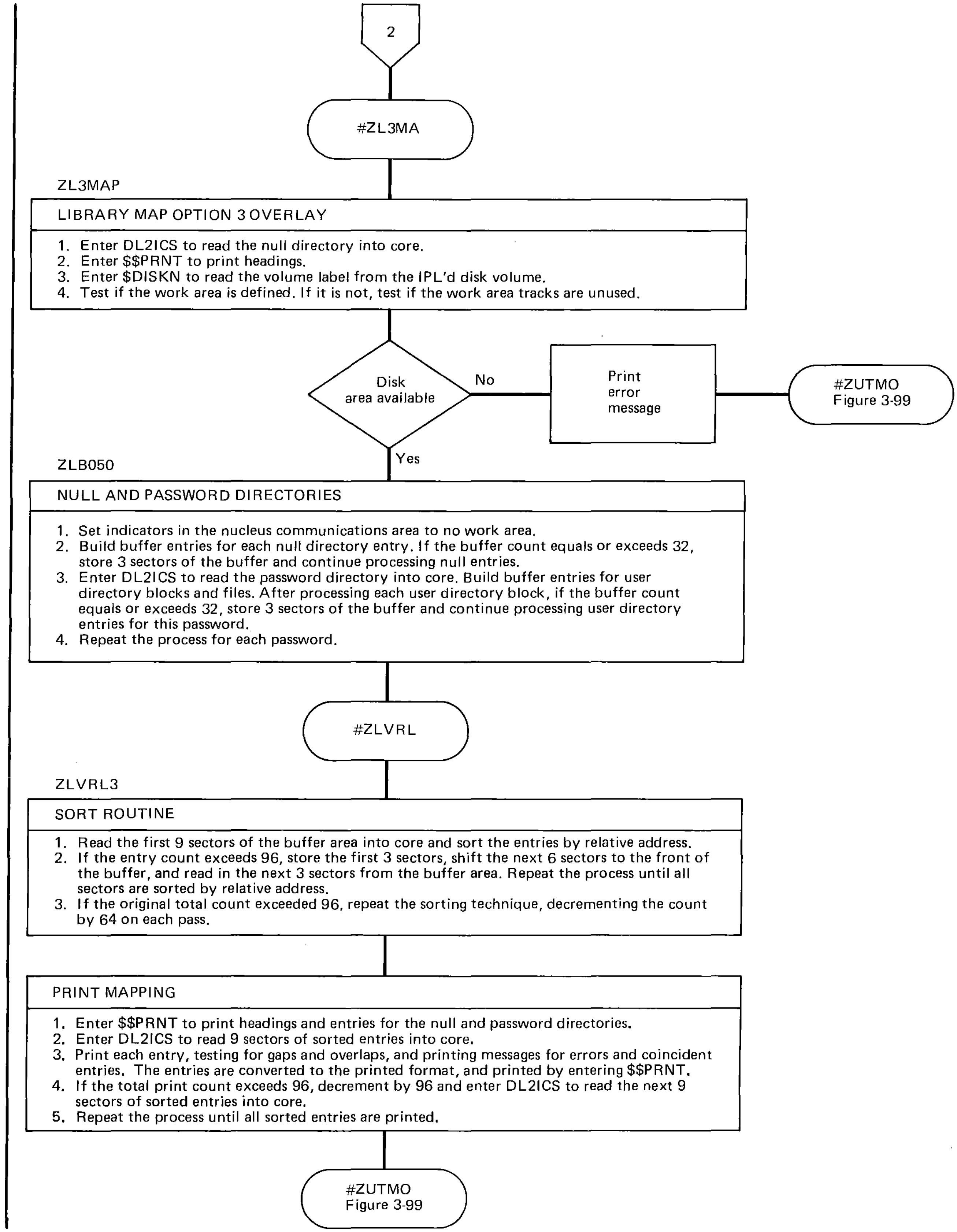


Figure 3-102. Library Mapping Overlays (Part 1 of 2)



BR2672.2

Figure 3-102. Library Mapping Overlays (Part 2 of 2)

COMPILER–#BCOMP, #BOVLY (Figure 3-119)

Compiler input is a sequence of BASIC statements in the work file. Output is a sequence of pseudo machine code (PMC), constants, and run-time indicators in virtual memory. Both the work file and virtual memory reside in the system work area on disk.

Refer to Section 7 for:

1. How to take a sequential disk dump of virtual memory.
2. Disk address specifications for the utility dump.
3. Conversion of virtual addresses to disk addresses (Figure 7-1).
4. Virtual memory map (Figure 7-2).
5. How to lay out virtual memory (standard precision).
6. Example of pseudo instruction references to virtual memory (Figure 7-3).
7. How to lay out virtual memory (long precision).
8. How to lay out an execution-time core dump.
9. Fixed core addresses in execution-time core dump (Figure 7-4).

Compiler Cycle

1. Retrieve one BASIC statement from the work file.
2. Use the type code in the statement as a displacement into the statement branch table.
3. Using the entry in the table, access a PMC generator and branch to it.
4. Generate a sequence of pseudo machine code (PMC), constants, and/or indicators and write these to virtual memory.
5. Perform steps 1 through 4 until the BASIC statements are depleted.

PMC in virtual memory is in the same sequential order as the BASIC statements in the work file.

Organization of Assembly Listings

All modules of the compiler are contained in these two assemblies:

1. Core resident routines–#BCOMP
2. PMC generator (statement processor) overlays–#BOVLY

Core Resident Routines–#BCOMP

This assembly contains these executable source modules in this physical order:

BGINIT–Compiler initiator
BHDIST–Compiler distributor
BAGETC–Statement input subroutine
BBPUTC–VM output subroutine
BCFCON–Constant generator subroutine
BDSYMB–Symbol translator subroutine
BECSCN–Character expression PMC subroutine
BFSCAN–Arithmetic expression PMC subroutine
BLISTA–Assignment list PMC subroutine
BMATXR–Matrix reference PMC subroutine
BRATAB–Branch table subroutine
BUZDBN–Convert decimal to binary subroutine
BVDL4T–Disk logical IOCS interface
BPALET–LET (arithmetic simple) statement processor
BNRMRK–REM statement processor

PMC Generator (Statement Processor) Overlays—#BOVLY

This assembly contains these executable source modules in this physical order:

BNDATA–DATA statement
BNFDEF–DEF statement
BPMLET–LET (arithmetic multiple) statement
BPCLET–LET (character) statement
BXPUTX–PUT statement
BKFORX–FOR statement
BKNEXT–NEXT statement
BXGETX–GET statement
BKARIF–IF (arithmetic) statement
BTPAUS–PAUSE statement
BKCRIF–IF (character) statement
BTSTOP–STOP statement
BKGOTO–GOTO (simple) statement
BKMGTO–GOTO (multiple) statement
BKSUBG–GOSUB statement
BXRSET–RESET statement
BKRTRN–RETURN statement
BXCLOS–CLOSE statement
BPREAD–READ statement
BPXRSR–RESTORE statement
BXINPT–INPUT statement
BNADIM–DIM statement
BXDPRT–PRINT statement
BXUPRT–PRINT USING statement
BNIMAG–Image (:) statement
BMMATA–MAT statement
BMGETX–MAT GET statement
BMINPT–MAT INPUT statement
BMREAD–MAT READ statement
BMPUTX–MAT PUT statement
BMDPRT–MAT PRINT statement
BMUPRT–MAT PRINT USING statement
BTRMNT–Compiler terminator (END statement)

Compiler Labeling Conventions

Because disk-resident statement processors must communicate with the core-resident compiler, a fixed equate module (BEQU) has been developed to reference core-resident instructions and areas. In addition, the compiler common module (BZCOMN) contains an equate section which has been developed to assist in defining the fixed addresses in BEQU. Essentially, BZCOMN equates reference the same core addresses as BEQU, except BZCOMN addresses are derived from the assembled code while BEQU addresses are manually adjusted constants.

Core-resident modules are coded to reference other core-resident modules, using the following conventions:

- Module Entry Points. Actual entry point label.
- Module Data/Instruction Fields. Equivalent BZCOMN label.

Disk-resident statement processor modules are coded to reference core-resident modules using the following conventions:

- Module Entry Points. Equivalent BEQU label.
- Module Data/Instruction Fields. Equivalent BEQU label.

Virtual-memory references are always specified using the appropriate VEQU label.

Program descriptions use the following conventions, with respect to both core-resident and disk-resident modules, for consistency:

- Module Entry Points. Actual entry point label.
- Module Data/Instruction Fields. Equivalent BEQU label.

For example:

- Actual core-resident entry point label– BCFCON

  Referenced from core as– BCFCON
  Referenced from statement processor as–B$FCON

- Actual core-resident data field label– BFSBKT

  Referenced from core as– BZBCKT
  Referenced from statement processor as–B$BCKT

Compiler Initialization

Entry: #BCOMP is loaded, at X'0600', via the nucleus loader function. #BCOMP is called directly by the RUN/STEP/TRACE keyword program (#KRUNI) or via the EDIT keyword program (#KEDIT). Figure 3-103 is an example of a core map with an 8k system.

Compiler Initiator: Functions of the compiler initiator (BGINIT) are:

- When long precision is indicated, floating point data length and virtual-memory addresses are changed in the following core-resident compiler subroutines:

  BBPUTC–Virtual memory output subroutine
  BCFCON–Constant generator subroutine
  BDSYMB–Symbol translator subroutine
  BFSCAN–Arithmetic expression PMC subroutine

- As many sectors from #BOVLY as possible are loaded into expanded core (12k or 16k). (The entire contents of #BOVLY can be loaded into 16k.) Entries in the processor address table (Figure 3-104) are modified to indicate the overlays resident in expanded core.
- Set compile-time indicators (data file pointer, primary input buffer clear switch).
- Seek to first cylinder of virtual memory.

BGINIT exits to the compiler distributor (BHDIST) and is overlayed by disk-resident PMC generators during the compilation. BGINIT is not overlayed if 16k expanded core is present.

Accessing PMC Generators

The pseudo machine code (PMC) generator required to process a BASIC statement can be:

1. A PMC generator overlay that is not in core.
2. A PMC generator overlay that is presently in core.
3. A PMC generator that is permanently in core.

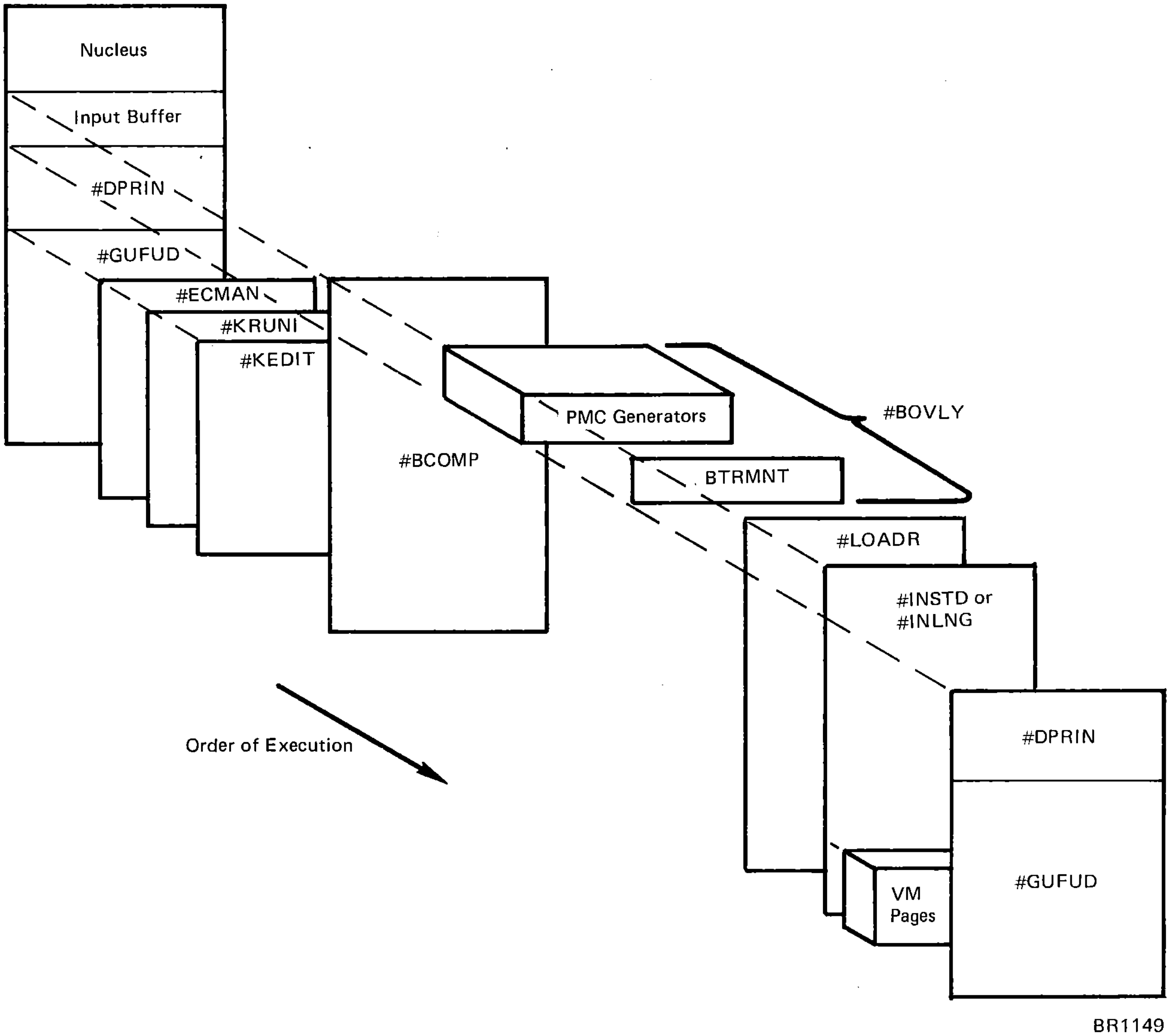


Figure 3-103. RUN Program Name Core Map (8k System), Example

The compiler distributor (BHDIST) contains a processor address table (Figure 3-104). The type code of the current BASIC statement indexes this table and the table contains information necessary to access the PMC generators.

When the required PMC generator is in core, a branch to the generator entry point is executed. When the required PMC generator is not in core, the appropriate disk sector is loaded into the transient area initially occupied by BGINIT.

All disk-resident PMC generators reside on the same disk track in the system work area. The PMC generator assembly (#BOVLY) is constructed so that each generator is contained within a sector boundary (every X'0100' bytes) where possible. Multiple generators may occupy the same sector, and a large generator can be segmented into two sectors (Figure 3-105).

Resolving Virtual-Memory Addresses

As sequences of PMC instructions are being generated, situations occur where an instruction references a line number or virtual-memory location that is currently unknown. When these situations occur, an instruction image with missing operand (X'0000') is generated. That is, a "hole" is temporarily left in virtual memory.

Two tables are maintained by the compiler for resolving these virtual-memory addresses. Both tables have the same format except for the content of the entries.

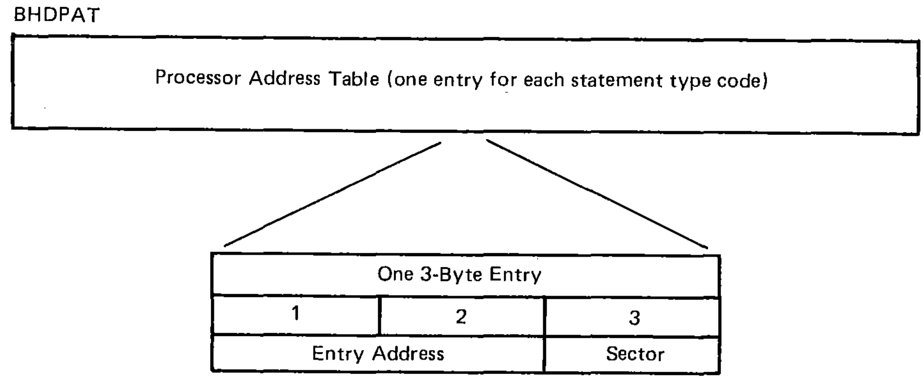
1. A statement address table (Figure 3-106) is created by BHDIST to relate statement line numbers to the virtual addresses of statement header pseudo instructions (STH/IMH). An entry is made in the table prior to processing each new statement.
2. A branch address table (Figure 3-107) is created by BRATAB to relate unresolved operands (holes in virtual memory) to line numbers or virtual-memory locations.

Four types of unresolved virtual addresses can occur in the PMC generators. The unknown operand references:

1. A line number that has been previously processed.
2. A line number that has not yet been processed.
3. The next sequential statement.
4. Another pseudo instruction that is not associated with a line number.

These situations are discussed individually in the following paragraphs.

When the unknown operand references a line number (STH or IMH pseudo instruction) that has not yet been encountered or that is already written in VM, the virtual address of the hole and the line number are passed as parameters to BRATAB. BRATAB creates an entry in the branch address table. Entries in this table are not resolved by the compiler. This table, along with the completed statement address table, is passed to the loader (#LOADR) for resolution.

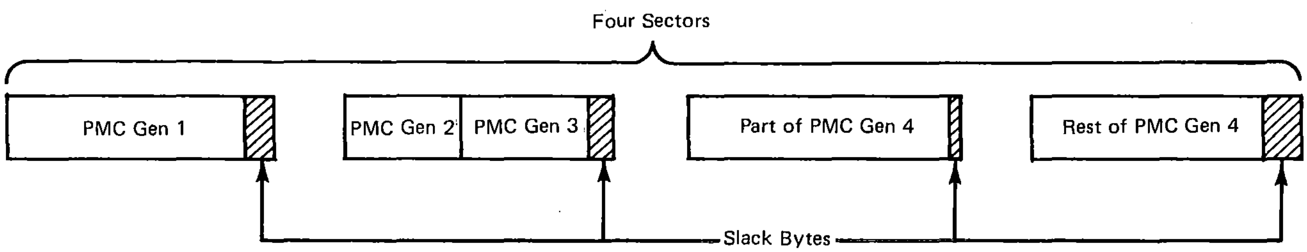


Notes:

1. "Entry address" is the entry point to the PMC generator (equal to X'0600' plus the displacement into the sector for disk-resident generator overlays).
2. "Sector" is the sector byte used in the DPL (Figure 5-24) when reading from the PMC generator track. A value of X'FF' indicates the generator is core-resident. The sector byte of the sector currently in the transient area is saved at label BHDDSA in case the same sector is required on consecutive compiler cycles.
3. The transient area where all overlay sectors are loaded is X'0600' to X'06FF'.

BR1151

Figure 3-104. Processor Address Table



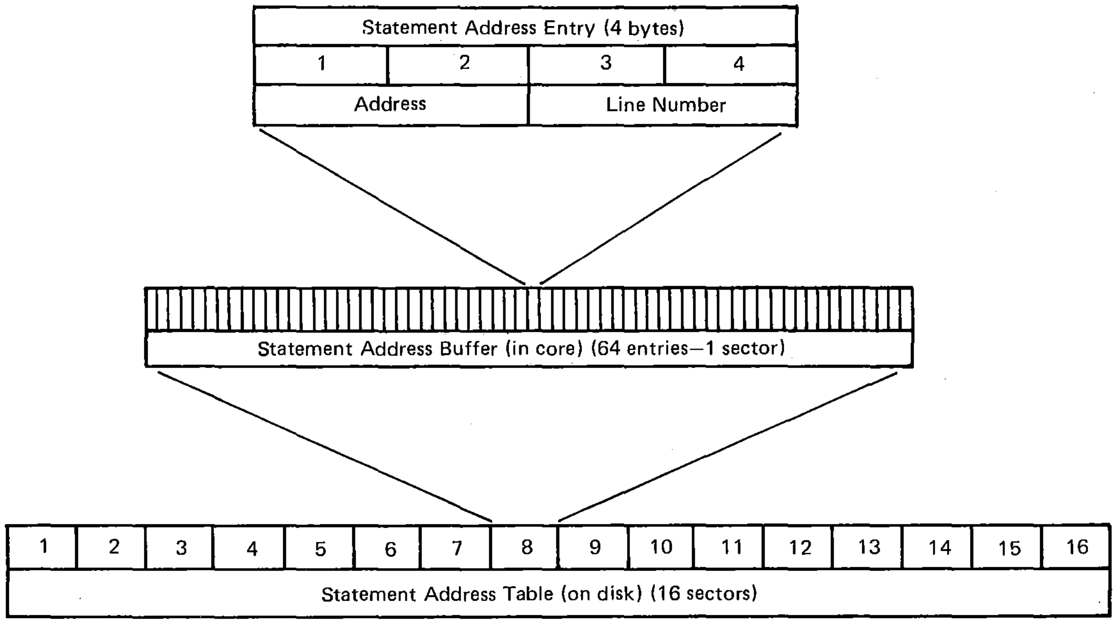
Notes:

1. PMC generator 1 is contained entirely on one sector. The unused area at the end of the sector is too small to contain another generator.
2. PMC generators 2 and 3 occupy the same sector.
3. PMC generator 4 is too large for one sector. Since only the first sector is loaded by the compiler distributor, generators that occupy more than one sector branch to label BHDST2 in the distributor, causing the distributor to load the next sector.

BR1152

Figure 3-105. Organization of PMC Generators on Disk

When the unknown operand references the next sequential statement (branch-to-next-statement), the next-address switch (B$NXSW) is set in BHDIST and the virtual address of the hole is saved as a parameter for BRATAB. BHDIST determines the line number of the next statement during normal processing and, because the switch is set, calls BRATAB to create an entry in the branch address table.



Notes:

1. The address field contains the virtual address of the statement header instructions (STH or IMH) associated with the line number.
2. Entries are always in ascending line number order.
3. Vacant entries contain binary 0's.
4. The 16-sector disk area can contain 1024 entries but never has more than 990 (the maximum BASIC program size).
5. X'FFFFFFFF' is inserted in the last (64th) entry of the last table sector before it is written to disk by the compiler terminator (BTRMNT). All preceding sectors are written by the branch table subroutine (BRATAB).

BR1153

Figure 3-106. Statement Address Table

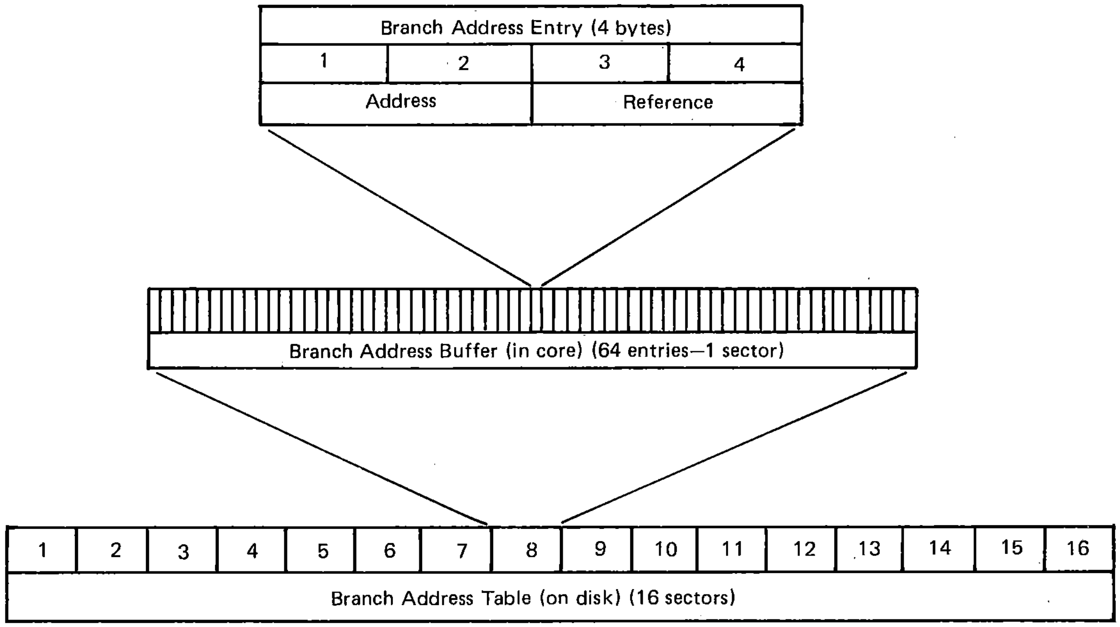
When the unknown operand references another pseudo instruction in the same PMC sequence (statement instruction group), the virtual address of the hole and the virtual address of the referenced instruction are passed, as parameters to BRATAB. The virtual address of the referenced instruction is always determined by the PMC generator on the same compiler cycle (see Figure 3-108 for example). BRATAB processes the virtual address the same as when a line number is referenced.

The lowest possible pseudo instruction virtual address is always greater than the highest possible binary line number. This is how the loader (#LOADR) differentiates between the two types of entries in the branch address table.

Core Resident Routines

Compiler Distributor—BHDIST

BHDIST passes control to the individual pseudo machine code (PMC) generators. (Refer to "Accessing PMC Generators.") Since each PMC generator completes processing for a single statement, BHDIST expects the next source text character to be the beginning of a new BASIC statement. The statement is scanned for the first nonnumeric character which should be the statement keyword. BHDIST performs this scan using BAGETC. The binary line number is saved for reference by the PMC generators. The statement type code is saved to index the processor address table (Figure 3-104). BHDIST bypasses disabled statements; bit 0 of the statement type code is on. A statement-header pseudo instruction (STH) is generated in virtual memory for each enabled statement. For image statements, this STH is later modified to be an image statement header (IMH).



Notes:

1. The address field contains the virtual address of an unresolved pseudo instruction operand (hole).
2. The reference field contains either a line number or the actual virtual address to be inserted in the hole.
3. Vacant entries contain binary 0's.
4. The 16-sector disk area can contain 1024 entries. If this limit is exceeded, compilation is aborted.

BR1154

Figure 3-107. Branch Address Table

Each PMC generator, except for the compiler terminator (BTRMNT), returns to BHDIST to complete the statement processing cycle. Some generators return to BHDIST via the REM statement processor (BNRMRK).

Statement Input Subroutine—BAGETC

BAGETC reads blocks of packed, segmented BASIC source text from the system work file. Core addresses of sequential text characters within these blocks are determined. These disk blocks are read in a logically sequential order by following the linkage fields in the work file data blocks (refer to Figure 5-15). The text is always presented in ascending line number order. The calling routine sets parameters for BAGETC in order to access a character position in a text segment.

Input parameters to BAGETC are:

1. B$NUMC—Character skip count. This field contains the relative displacement between accessed characters. A value of X'FF' accesses (skips to) the terminating character (EOS) of the current segment (BASIC statement). A value of X'01' accesses the next consecutive text character. A value of X'00' returns the address of the previously accessed character and does not advance to a new character. This parameter defaults to X'01' if it is not explicitly set prior to entry.
2. B$GBSW—Bypass blanks switch. On (X'01') ignores blanks when advancing to a new character. Off processes blanks the same as other text characters.

Example: The PMC sequence below contains three branch instructions, two of which require resolution by the loader (#LOADR). The arrow at 1 refers to the next sequential statement. The arrow at 2 refers to another pseudo instruction not associated with a line number. The arrow at 3 refers to a pseudo instruction whose virtual address is known (since it was previously established within the same compiler cycle) at the time when the BRA instruction is generated; no entry in the branch address table is required in this case.

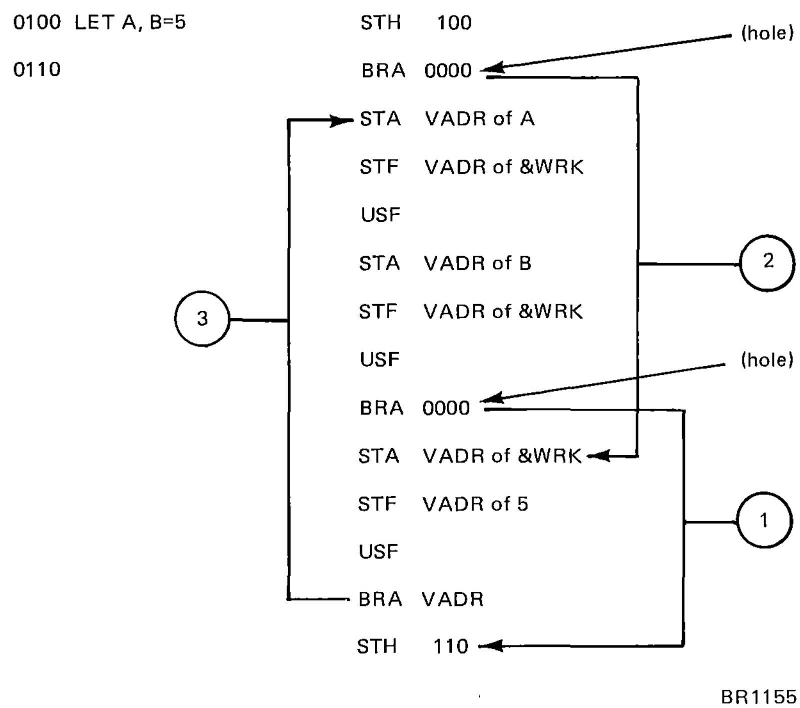


Figure 3-108. PMC Sequence (Branch Instructions)

Output parameters from BAGETC are:

1. Index register @XR–Character core address. This register contains a pointer to the selected text character as requested by the calling routine.
2. B$LINE–Line number. This two-byte field contains the binary line number of the BASIC statement currently being processed.
3. B$TYPE–This one-byte field contains the statement type code from the statement currently being processed.
4. B$GPTR–Address of selected character. This two-byte field contains the core address of the selected text character, and is used as a backup for register @XR.

Virtual Memory Output Subroutine–BBPUTC

BBPUTC puts pseudo machine code strings of 1 to 255 bytes into sequential virtual-memory locations or stores 256-byte blocks of constants into sequentially descending virtual memory pages. BBPUTC is called to perform one of four functions:

1. Add record–This function code is set by default.
2. Write page–Function code equated to B$PFWP.
3. Add error–Function code equated to B$PFAE.
4. Close–Function code equated to B$PFCL.

Each function is performed by setting parameter B$PFNC with one of these codes. The add record function is performed by default unless B$PFNC is specifically set to an alternate code prior to the subroutine call.

Add Record: Single PMC instructions or sequences are loaded into consecutive locations in the output buffer. Full buffers are written to sequential, ascending sectors (pages) in virtual memory. Buffers are padded with at least one EOP pseudo instruction before they are written (refer to virtual memory map, Figure 7-2). The core address (B$PCAD) and the length minus 1 (B$PNBY) of the PMC string are required input parameters for this function.

Write Page: One full virtual memory page is written to disk from the compiler constant buffer. This function is used to write data blocks containing generated constants into sequentially decreasing virtual memory pages beginning with the base constant address (see BCFCON).

Add Error: This function is used to record compiler-generated errors. At the first execution of this function, virtual address pointers are reset, the compiler error switch (B$ERSW) is set, and the add-record and write-page functions are disabled. Each add-error function puts a three-byte error entry into virtual memory using the method described for the add-record function. These error entries are written over PMC sequences previously generated. The three-byte error entry consists of an error definition code (message number in hexadecimal) followed by the associated BASIC statement line number. The error code is passed as an input parameter at label B$PERC. The line number is taken from area B$LINE where it is normally stored.

Close: This function fills the PMC instruction output buffer with EOP pseudo instructions and writes the full buffer to virtual memory as in the add-record function, closing compile-time PMC generation.

The lowest page number referenced by a write-page function is compared to the page currently in the PMC instruction output buffer on each execution of BBPUTC. If an overlap occurs, compilation is aborted.

Constant Generator Subroutine—BCFCON

BCFCON is called to convert a BASIC source statement constant to internal format, put it into virtual memory, and return the virtual address of the first byte of the constant to the calling routine via label B$BCKT. The type of constant is passed as an input parameter via label B$CTYP. Three types of constants can be processed:

1. Arithmetic constant—Type code set by default.
2. Character constant—Type code equated to label B$CCON.
3. Character string constant—Type code equated to label B$SCON.

On entry, the index register (@XR) points to the first character of the constant in the statement input buffer. On exit, this register points to the first non-blank character after the constant. (Refer to "Statement Input Subroutine—BAGETC.")

Each constant is generated into a 19-byte work area in a form suitable for virtual memory. Constants are loaded in descending order in the constant output buffer. For arithmetic or character constants, the constants in the current output buffer are scanned and duplicates are not created. No check is made for duplicates of character strings. Full buffers are written to contiguous, descending sectors (pages) in virtual memory (refer to Figure 7-2).

Arithmetic Constants: These constants are found in algebraic expressions or data lists, and are converted from EBCDIC to unpacked-decimal, floating-point format in the 19-byte work area (Figure 3-109). They are converted to packed-decimal, floating-point format before they are moved to the constant output buffer. For long precision, modifications have been made to the packing and output routines by the compiler initiator (BGINIT). (Refer to "Floating-Point Arithmetic.")

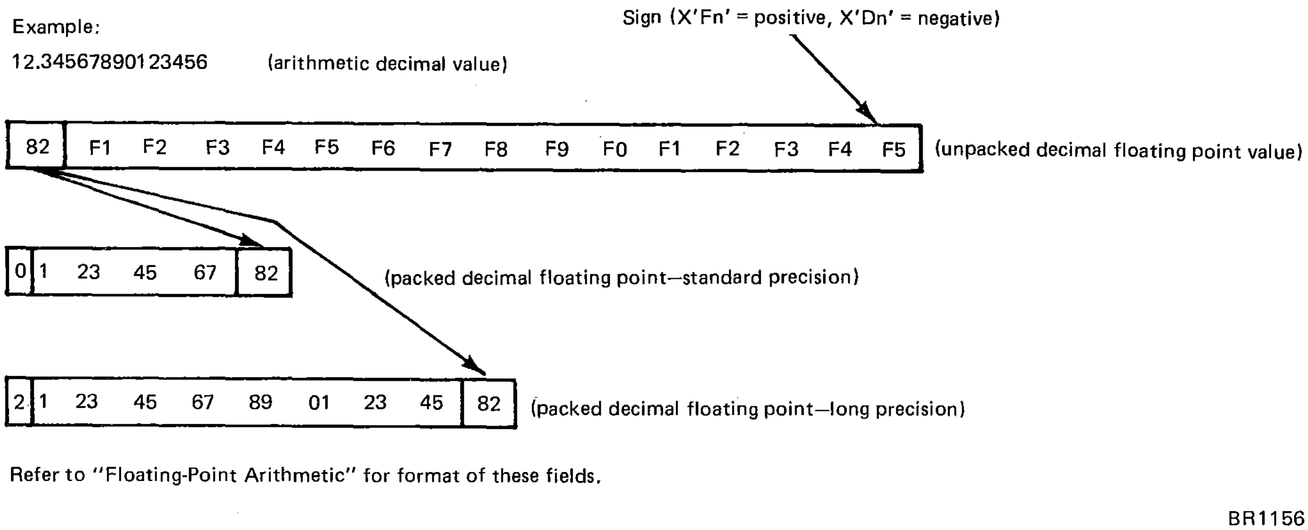


Figure 3-109. Conversion of an Arithmetic Constant to Unpacked Floating Point and then to Packed Floating Point

Character Constants: These constants (Figure 3-110) are character strings tailored to fit 18-byte character constant fields, and are associated with character variables or character array elements. The first character in the input string is a delimiter. The next single occurrence of this delimiter is the end of the string. Any paired occurrence of the delimiter character is interpreted as a single character in the string. The source character string is scanned, checking for delimiters and moving characters to the work area.

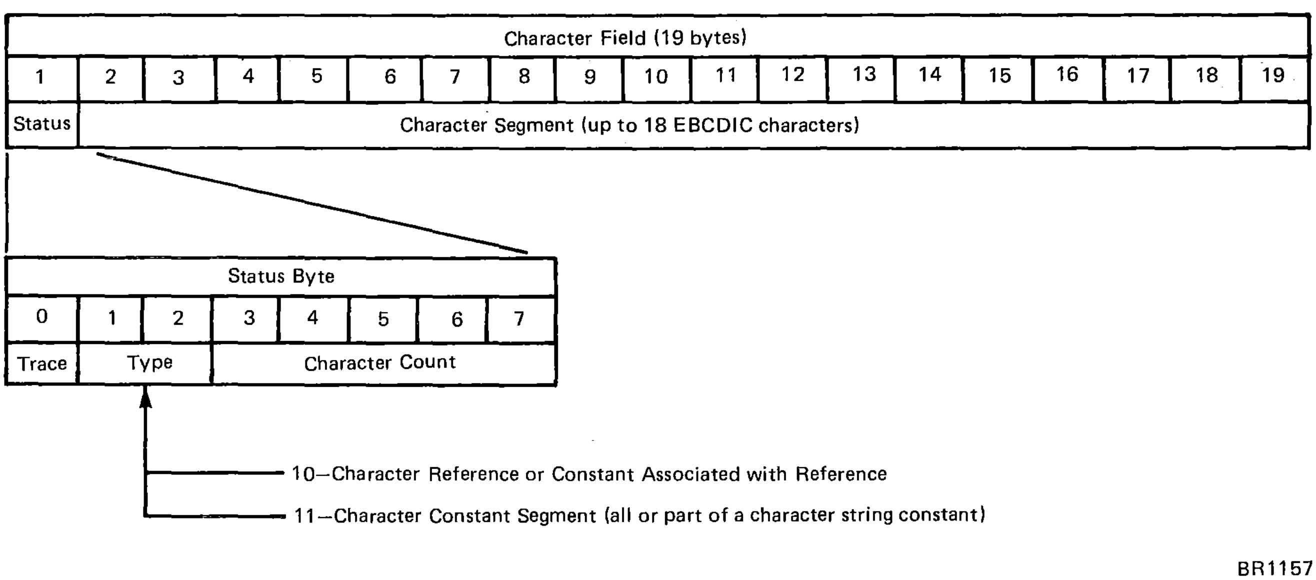


Figure 3-110. Character Field Format

Character String Constants: These constants (Figure 3-110) are variable length and are not associated with character variables or character array elements. If the length of a string exceeds the size of the work area, more than one string segment is constructed and moved to the output buffer. Delimiters are processed in the same manner as for character constants.

Symbol Translator Subroutine—BDSYMB

BDSYMB is called to analyze all symbols encountered in BASIC statements, allocate space in virtual memory, and return the virtual address of the allocated space (or entry point of an intrinsic function) via label B$BCKT. The symbol type is analyzed as belonging to one of the eight categories listed in Figure 3-111. No values are written in virtual memory by this subroutine.

	Allocated Element and Length	Returned Virtual Address	Table	Table Format
Arithmetic (letter) variables	Packed floating-point value; 5 bytes for standard, or 9 bytes for long precision.	Virtual address of first byte of element.	B$SLVT	29, 2-byte virtual addresses assigned to symbols $, #, @, A–Z.
Arithmetic (letter-digit) variables			B$SLDT	290, 2-byte virtual addresses assigned to ($, #, @, A–Z)*(0-9)
Arithmetic array reference	Arithmetic array dope vector; 8 bytes.		B$SNAT	29, 6-byte entries. First 2 bytes contain virtual address assigned to symbols $, #, @, A–Z. Last 4 bytes contain specified array dimensions (two 2-byte values).
Character variable	Character variable field; 19 bytes.		B$SCVT	29, 2-byte virtual addresses assigned to symbols $$, #$, @S, A$–Z$.
Character array reference	Character array dope vector; 4 bytes.		B$SCAT	29, 4-byte entries. First 2 bytes contain virtual address assigned to symbols $$, #$, @S, A$–Z$. Last 2 bytes contain specified array dimension (2-byte value).
User function reference	User function (subroutine) virtual address execution entry point (2 bytes).		B$SFNT	29, 4-byte entries. First 2 bytes contain virtual address assigned to functions FN$, FN#, FN@, FNA-FNZ. Last 2 bytes contain virtual address of associated DEF statement execution entry point.
Intrinsic function reference	None	Entry point to VM-resident intrinsic funtion subroutine.	BDSIFT	24, 5-byte entries containing a 3-byte function name and a 2-byte virtual address.
Secondary (delimiting) keyword	None	Virtual address at label B$BCKT is not changed.	BDSKWT	Four 2-byte entries containing first 2 bytes of each secondary keyword (TH, etc.). An additional check is made to insure that 'ST' is actually the beginning of keyword 'STEP'.

BR1158B

Figure 3-111. Symbol Processing in BDSYMB

In the following examples, the characters $, (, and FN are identifiers for the symbol A:

A—Arithmetic (letter) variable
A1—Arithmetic (letter-digit) variable
A$—Character variable
A(—Arithmetic array reference
A$(—Character array reference
FNA—User function
LOG—Intrinsic function
THEN—Delimiting keyword

Symbol Tables: The BASIC language has an absolute number of usable symbols. Symbol tables in BDSYMB contain entries (initially binary 0's) for every possible symbol (Figure 3-111). A relative displacement, to the entry corresponding to a symbol, is determined by scanning the alphabet reference table (BDSART) for equal or low.

The first time a symbol is referenced in a BASIC program, space for the associated element is allocated in virtual memory. The virtual address of the element is moved to the corresponding entry in one of the symbol tables, and also returned to the calling routine. Subsequent references to the same symbol return the virtual address from the table entry.

Space in virtual memory for elements is assigned using two virtual-memory address pointers:

B$SFAB—User function addresses and array dope vectors.
B$SVRB—Variable elements (arithmetic and character).

The initial value of B$SFAB is X'0000' and is decremented as each user function address or array dope vector is allocated space. The resulting virtual address references the first byte of the element (example: X'0000' - 8 = X'FFF8').

The initial value of B$SVRB is X'F536' (X'F049' for long precision) and is incremented as each variable element is allocated space. This address also points to the first byte of the variable element.

The virtual-memory area defined by the initial values of B$SVRB to B$SFAB accommodates all possible elements the user can define in a single BASIC program. Any unused area between these addresses, at the end of compilation, is available to the loader (#LOADR) for the allocation of small arrays (refer to Figure 7-2).

Intrinsic Functions: The intrinsic function table is scanned for a match to the BASIC name of the function. The virtual address (fixed entry point) from the table is returned to the calling routine.

Input parameters to BDSYMB are:

1. Index register (@XR)—Text character pointer. This register contains the core address of the first character in the identifier of the symbol to be processed.
2. B$MRSW—Matrix reference switch. When this switch is on, references that would otherwise be interpreted as simple letter variables are interpreted as arithmetic array references.
3. B$FSSW—Function scan switch. When this switch is on, all arithmetic variable references are matched against a user-function, dummy-argument identifier. Matching references are assigned the dummy argument virtual address rather than that derived from a symbol table.
4. B$FSC1—Function scan identifier (first character). This parameter contains the first character of the user-function, dummy-argument identifier during a function scan.
5. B$FSC2—Function scan identifier (second character). This parameter contains the digit portion of the user-function, dummy-argument identifier, if present, during a function scan. When none exists, the value at B$FSC2 is X'40' (EBCDIC blank).
6. B$FSVA—Function scan virtual address. This parameter contains the virtual address of a user-function-dummy-argument assigned during a function scan.

Output parameters from BDSYMB are:

1. Index register (@XR)—Text character pointer. If the symbol is a secondary keyword or a letter variable immediately followed by a delimiting keyword, this register points to the second character in the keyword. In all other cases, the register points to the character following the complete identifier. Blanks are ignored.
2. B$BCKT—Identifies virtual address bucket. This contains the virtual address of the leftmost byte of the element associated with the processed identifier.

3. B$ADSW—Address available switch. This switch is on when a virtual address is stored in B$BCKT.
4. B$IFSW—Intrinsic function switch. This switch is on when the symbol is an intrinsic function reference.
5. B$FRSW—Function reference switch. This switch is on when the symbol is either a user or intrinsic function reference.
6. B$CRSW—Character reference switch. This switch is on if the symbol is either a character variable or character array reference.
7. B$KWSW—Expression keyword switch. This switch is on when the symbol is a secondary keyword (alone or following a letter variable).
8. B$HRSW—Matrix reference switch. This switch is on when a matrix-directly intrinsic function is encountered.

Character Expression PMC Subroutine—BECSCN

BECSCN is called to generate pseudo instructions, in virtual memory, that will stack one of the following character expressions:

1. Character variable—Generates a stack-character-field (STC) pseudo instruction (Figure 3-112).
2. Character array element—Generates a stack-character-array-element (SC1) pseudo instruction preceded by a stack-arithmetic-expression-value (Figure 3-112).
3. Character literal—Generates a stack-character-field (STC) pseudo instruction (Figure 3-112).

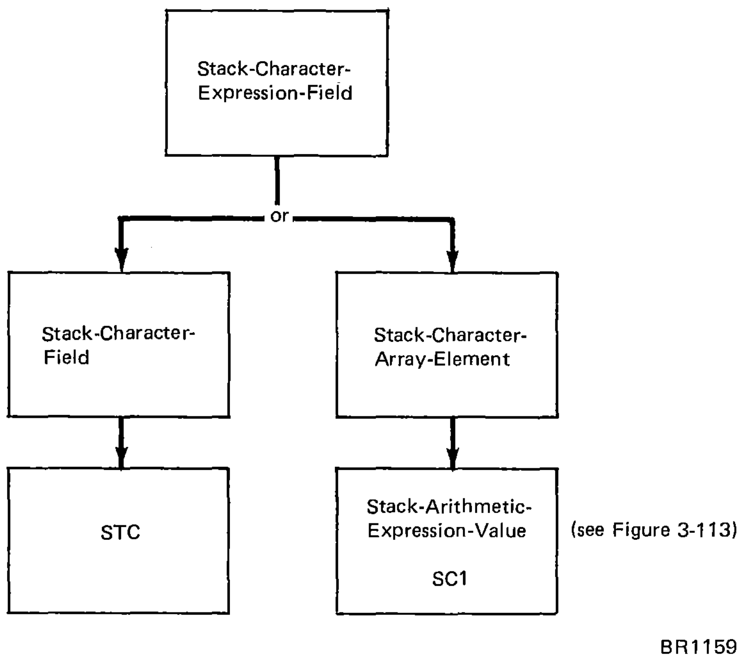


Figure 3-112. Stack-Character-Expression-Field

Input Text Pointer: Index register @XR contains the core address of the character preceding the first character of the character expression unless B$NUMC = 0 or switch B$CSSW is on. If B$NUMC = 0, the compiler input subroutine (BAGETC) is effectively disabled and the text pointer references the first character of the character expression. If switch B$CSSW is on, the arithmetic expression PMC subroutine (BFSCAN) was called to process the expression, and encountered a $ identifier in the expression. When switch B$CSSW is on, the text pointer references the character following the $ identifier, and the character reference symbol virtual address is stored in B$BCKT.

Arithmetic Expression PMC Subroutine—BFSCAN

BFSCAN is called to generate stack-arithmetic-expression-values, composed of value stacking and arithmetic pseudo instructions, in virtual memory (Figure 3-113). One entry to this subroutine generates all pseudo instructions necessary to stack the value represented by a single arithmetic expression. An arithmetic expression can be a single symbol, or symbols separated by arithmetic operators (+, –, *, /, ↑, or **). The expression can contain signed symbols (unary – or + sign).

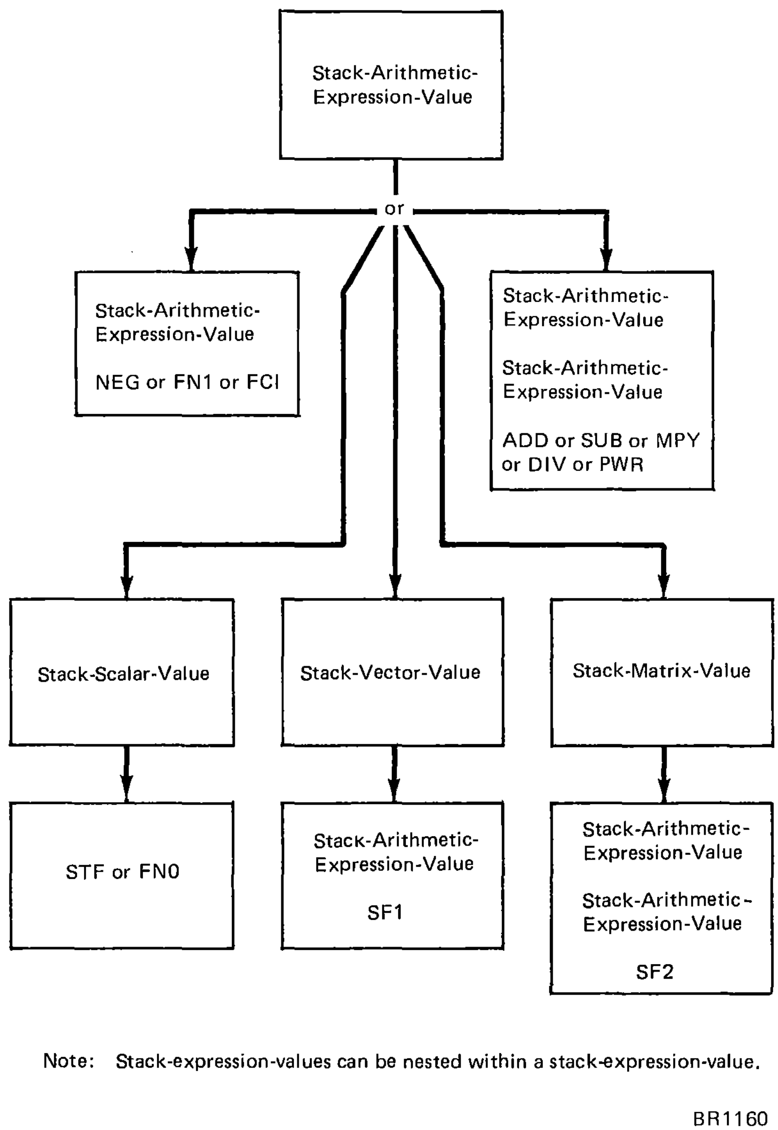


Figure 3-113. Stack-Arithmetic-Expression-Value

The unary minus (negative) sign and its quantity must normally be enclosed in parentheses (–B). If, however, the unary operator applies to the leading term of an expression, parentheses are unnecessary (–A means negative of A, +A means A).

The operands in the expression can be any of the following types:

1. Arithmetic variables.
2. Arithmetic array elements.
3. Arithmetic (numeric) constants.

4. Arithmetic (internal) constants.
5. Intrinsic functions.
6. User-defined functions.
7. Subexpressions (those enclosed in parentheses).

The following major work areas and tables are used in BFSCAN to process arithmetic expressions:

1. Compile-time stack (BFSSTK)—This stack operates as a first-in/last-out queue. It has a maximum capacity of 53 two-byte entries (arithmetic operation pseudo instructions require two bytes, function pseudo instructions require four bytes, and array pseudo instructions require six bytes).
2. Operand address bucket (B$BCKT)—The virtual address of the last encountered operand is saved at this location until it can be output in a pseudo instruction or placed in the compile-time stack. The available address switch (B$ADSW) indicates when this location contains a usable address.
3. Current entry (BFSCEN)—This location holds a pseudo op-code and the priority of the current arithmetic operation while they are being processed.
4. Scan routine branch table (BFSTBL)—This table contains a five-byte entry for each valid BASIC arithmetic expression character except letters and numbers (A-Z and 0-9). Each entry contains the EBCDIC character, the address of the routine within BFSCAN that processes the character, the hexadecimal value of the pseudo op-code (characters that do not generate a pseudo instruction contain X'00'), and the hexadecimal value of the priority code. Refer to this table for the priorities of arithmetic operators.

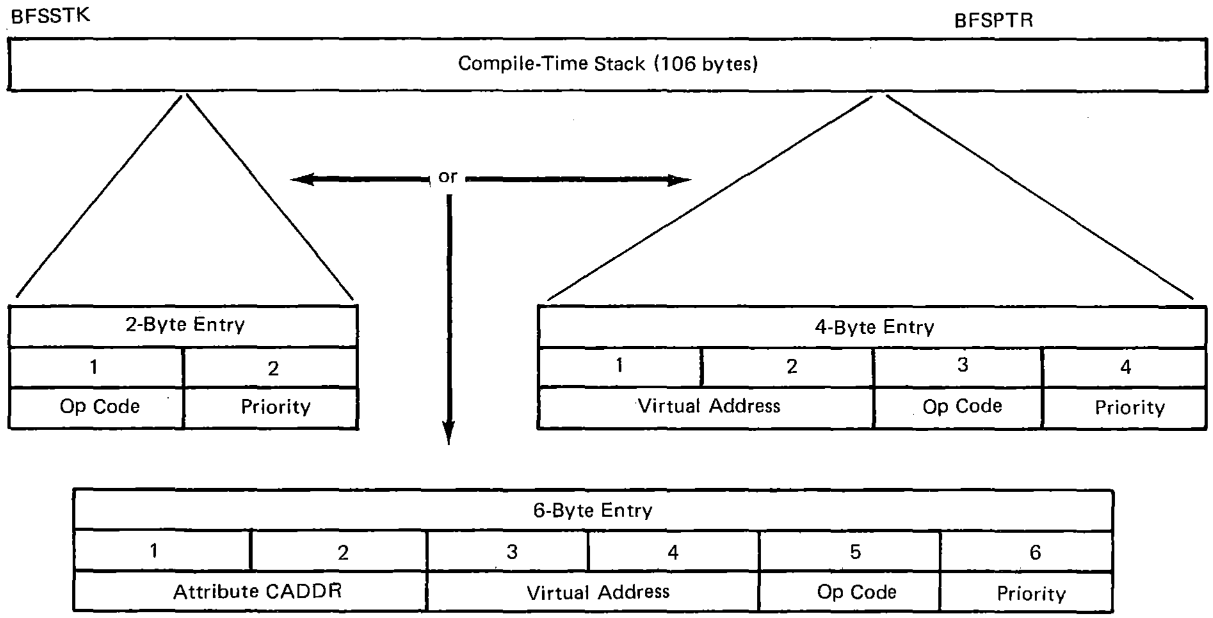
Priority: Pseudo instructions are generated to conform with the priority of the operations within the arithmetic expression. BFSCAN scans the arithmetic expression, from left to right, one character at a time. An entry is loaded into the top of the compile-time stack for each operational or function/array pseudo instruction that is generated (Figure 3-114).

If the priority of the current entry (BFSCEN) changes to a value lower than or equal to that of the last entry loaded into the stack, the stack popper (BFS160) is entered. This routine unloads an entry from the top of the stack, builds a pseudo instruction from the entry, and deletes the entry from the stack. Entries are unloaded, one at a time, as long as the priority of each stack entry is higher than or equal to that of the current entry (BFSCEN).

All entries active in the stack, when the end of the arithmetic expression is reached, are unloaded by the stack popper. The virtual-memory output subroutine (BBPUTC) is called to move each generated pseudo instruction to the output buffer and write it into virtual memory.

Input Text Pointer: Index register @XR contains the core address of the character preceding the first character of the arithmetic expression except when B$NUMC is set = 0. With B$NUMC = 0, the register contains the first character of the expression. The address in the register on output depends on the type of expression processed:

1. Arithmetic expression without a delimiting keyword—The pointer references the first nonblank character after the expression.
2. Arithmetic expression with a delimiting keyword—The pointer references the second character of the delimiting keyword.
3. Character variable—The pointer references the character following the $ identifier.
4. Character constant—The pointer references the leading delimiter (quote mark).



Notes:

1. The op code field contains the pseudo instruction operation code.
2. The priority field contains the priority of the current arithmetic operation.
3. The virtual address field is present in the stack for 3-byte function or array pseudo instructions. This address is determined by the symbol translator subroutine (BDSYMB).
4. The stack pointer (BFSPTR) contains the location of the next entry to be stacked. BFSPTR marks the top of the stack; BFSSTK marks the bottom of the stack.
5. The attribute CADDR field contains the core address of the attribute (array usage flags) in an array dope vector.

BR1161

Figure 3-114. Entries in Compile-Time Stack

Converting Arithmetic Expressions to Pseudo Instruction Sequences

The logical rules for conversion are:

1. Perform all conversions from left to right.
2. Begin conversions at the innermost subexpression level and then proceed outward. "Subexpression" refers only to normal parenthetical inner expressions, and does not include array reference subscript expressions or function reference argument expressions.
3. Convert all subscript and function argument expressions within the current subexpression level, treating these as independent expressions.
4. Convert all array and function references within the current subexpression level.
5. Within the current subexpression level, perform all highest-priority operations first, then the next highest, etc., until all operations have been completed. The priority of operations is, from high to low: ↑ or **, unary - or + sign, * or /, - or +. The unary minus (negative of) sign and its quantity must normally be enclosed in parentheses (-B). If however, the unary operator applies to the leading term of an expression, parentheses are unnecessary (-A means negative of A, +A means A).
6. The resultant pseudo instruction sequence represents an expression value in the next outer subexpression level. Go to number 3 to resolve the next outer subexpression until all levels are resolved.

Example: Convert the following arithmetic expression to a pseudo instruction sequence (the circled numbers, such as (1), represent expressions developed as the operation progresses):

```
A+(A+C(A+1,B)-B)/D(A↑B)
```

Step 1. Using rule 2, there are two subexpression levels. The innermost is (A+C(A+1,B)-B). The next outer subexpression level is A+expression/D(A↑B).

Step 2. Using rule 3, array reference subscript A+1 is converted first. Use items 1 through 38 in Figure 3-115 (as they apply) for these conversions. A+1 is the same form as item 7 in Figure 3-115. Therefore, it is converted to:

```
STF  A
STF  1   } stack-expression-value (1)
ADD
```

Step 3. Using rule 4, array reference C (expression (1),B) is converted (item 30 in Figure 3-115) to:

```
stack-expression-value (1)
STF B
SF2 C(
```

Step 4. Insert the resultant pseudo instructions from step 2 (shown within broken line) and the sequence is:

```
STF  A
STF  1
ADD       } stack-expression-value (2)
STF  B
SF2  C(
```

Step 5. Using rule 5, there are two operations in the current subexpression level: A+expression (2) -B. The priority of operations is, from high to low: ↑ or **, unary - or + sign, * or /, - or +. From left to right (rule 1), convert A+expression (2); then expression (3)-B will resolve this subexpression level. A+expression (2) is converted (item 17 in Figure 3-115) to:

```
STF A
stack-expression-value (2)
ADD
```

Step 6. Insert the resultant pseudo instructions from step 4 (shown within broken line) and the sequence is:

```
STF  A
STF  A
STF  1
ADD       } stack-expression-value (3)
STF  B
SF2  C(
ADD
```

Step 7. Expression–B is converted (item 13 in Figure 3-115) to:

```
stack-expression-value (3)
STF  B
SUB
```

Step 8. Insert the resultant pseudo instructions (shown within broken line) from step 6 and the sequence is:

```
┌─────┐
│STF A│
│STF A│
│STF 1│
│ADD  │
│STF B│  stack-expression-value (4)
│SF2 C(│
│ADD  │
└─────┘
 STF B
 SUB
```

Step 9. Resolution of this subexpression level is complete. The next higher subexpression level is A+expression (4) /D(A↑B). Using rule 3, array reference subscript A↑B is converted (item 11 in Figure 3-115) first:

```
STF  A
STF  B   stack-expression-value (5)
PWR
```

Step 10. Using rule 4, array reference D is converted (item 28 in Figure 3-115) to:

```
stack-expression-value (5)
SF1  D(
```

Step 11. Insert the resultant pseudo instructions (shown within broken line) from step 9 and the sequence is:

```
┌─────┐
│STF A│
│STF B│  stack-expression-value (6)
│PWR  │
└─────┘
 SF1 D(
```

Step 12. Using rule 5, there are two operations in this subexpression level: A+expression (4) [step 8] /expression (6) (step 11). The priority of the / is higher than the + so convert expression (4) /expression (6); then A+expression (7) will resolve this subexpression level. Expression (4) /expression (6) is converted (item 25 in Figure 3-115) to:

```
stack-expression-value (4) (step 8)
stack-expression-value (6) (step 11)
DIV
```

Step 13. Insert the resultant pseudo instructions (shown within broken lines) from step 8 and step 11, and the sequence is:

```
STF A
STF A
STF 1
ADD
STF B
SF2 C(
ADD          stack-expression-value (7)
STF B
SUB
STF A
STF B
PWR
SF1
DIV
```

Step 14. A+expression is converted (item 17 in Figure 3-115) to:

```
STF A
stack-expression-value (7)
ADD
```

Step 15. Insert the resultant pseudo instruction (shown within broken line) from step 13 and the sequence is:

```
STF A
STF A
STF A
STF 1
ADD
STF B
SF2 C(
ADD          stack-expression-value (8)
STF B
SUB
STF A
STF B
PWR
SF1 D(
DIV
ADD
```

Step 16. Resolution of this subexpression level is complete.

Item	Subexpression	Pseudo Instruction Sequence
1	A	STF A
2	+A	STF A
3	−A	STF A NEG
4	expression	stack-expression-value
5	+expression	stack-expression-value
6	−expression	stack-expression-value NEG
7	A+B	STF A STF B ADD
8	A−B	STF A STF B SUB
9	A*B	STF A STF B MPY
10	A/B	STF A STF B DIV
11	A↑B or A**B	STF A STF B PWR
12	expression+B	stack-expression-value STF B ADD
13	expression−B	stack-expression-value STF B SUB
14	expression*B	stack-expression-value STF B MPY
15	expression/B	stack-expression-value STF B DIV
16	expression↑B or expression**B	stack-expression-value STF B PWR
17	A+expression	STF A stack-expression-value ADD
18	A−expression	STF A stack-expression-value SUB
19	A*expression	STF A stack-expression-value MPY
20	A/expression	STF A stack-expression-value DIV
21	A↑expression or A**expression	STF A stack-expression-value PWR

BR1177.1

Figure 3-115. Conversions of Subexpressions to Pseudo Instruction Sequences (Part 1 of 2)

Item	Subexpression	Pseudo Instruction Sequence
22	expression+expression	stack-expression-value stack-expression-value ADD
23	expression−expression	stack-expression-value stack-expression-value SUB
24	expression*expression	stack-expression-value stack-expression-value MPY
25	expression/expression	stack-expression-value stack-expression-value DIV
26	expression↑expression or expression**expression	stack-expression-value stack-expression-value PWR
27	C(A)	STF A SF1 C(
28	C(expression)	stack-expression-value SF1 C(
29	C(A,B)	STF A STF B SF2 C(
30	C(expression,B)	stack-expression-value STF B SF2 C(
31	C(A,expression)	STF A stack-expression-value SF2 C(
32	C(expression,expression)	stack-expression-value stack-expression-value SF2 C(
33	FNC(A)	STF A FCI FNC(
34	FNC(expression)	stack-expression-value FCI FNC(
35	IFN(A)	STF A FN1 IFN(
36	IFN(expression)	stack-expression-value FN1 IFN(
37	RND	FN0 RND
38	DET(C)	SD0 C(MF1 DET STF &WRK

Notes:

1. A and B are simple scalar references or constants.
2. IFN may be any intrinsic function requiring a scalar argument.
3. C(is the virtual address of an array dope vector.
4. FNC(is a virtual address pointing to the virtual address of a user-defined function.
5. IFN(and RND are virtual addresses of intrinsic functions.
6. &WRK is the virtual address of a work area.

BR1177.2

Figure 3-115. Conversions of Subexpressions to Pseudo Instruction Sequences (Part 2 of 2)

Assignment List PMC Subroutine—BLISTA

BLISTA generates stack-variable-address PMC sequences (Figure 3-116), for single variable references in a list, in virtual memory. Typical BASIC statements that can contain these lists are:

[LET]
READ
INPUT
GET

The lists reference these types of elements:

1. Arithmetic scalar variable
2. Arithmetic vector element
3. Arithmetic matrix element
4. Character variable
5. Character array element

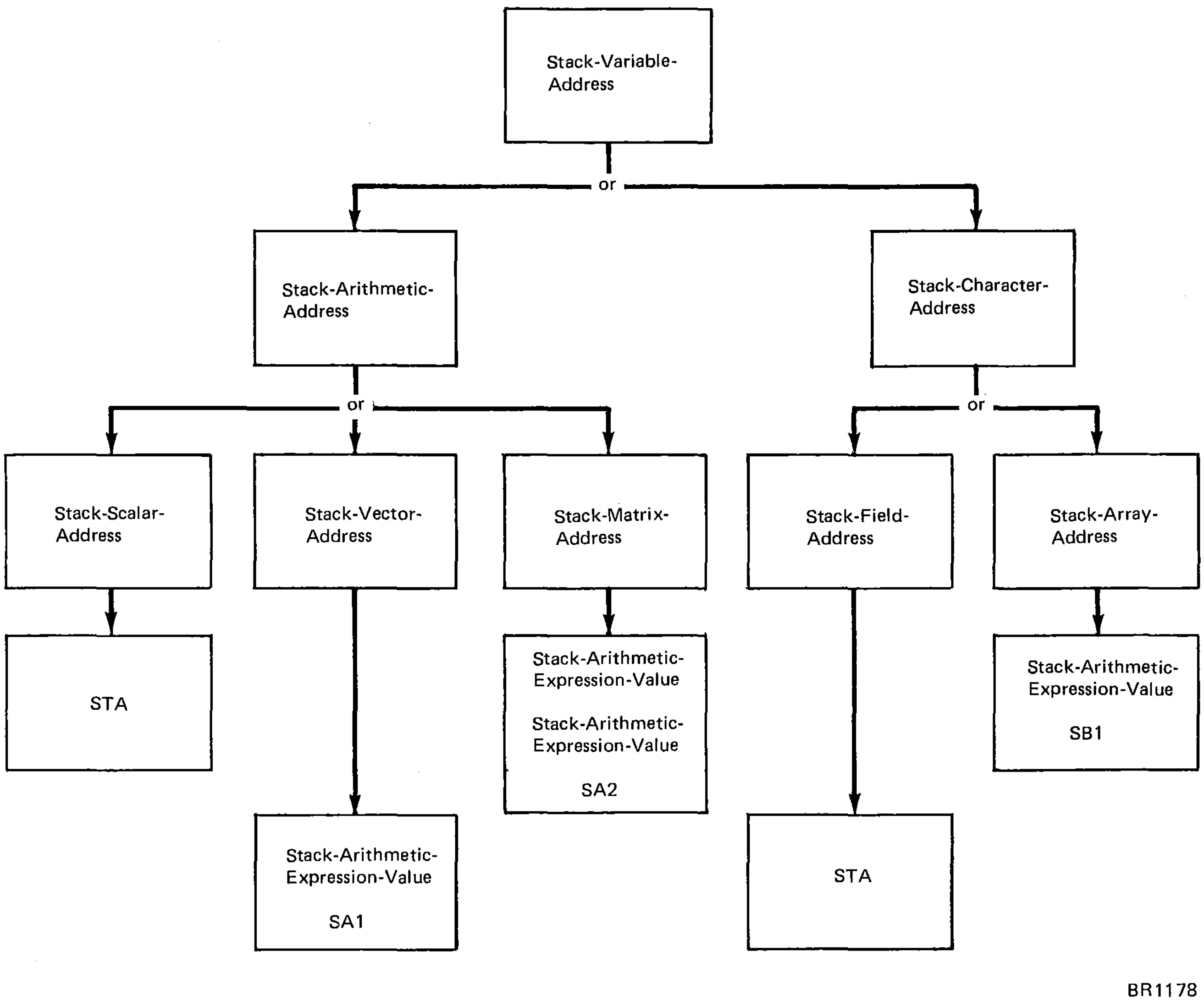


Figure 3-116. Stack-Variable-Address

The symbol translator subroutine (BDSYMB) determines the virtual address to be appended to each address stacking op code. The arithmetic expression PMC subroutine determines the pseudo instructions for all array subscript expressions. The pseudo instructions are written to virtual memory by BBPUTC.

Input Text Pointer: Index register @XR contains the core address of the first character in the list variable to be processed; following processing, it contains the core address of the first nonblank character after the variable reference.

B$LTYP—List reference type. This indicator is set to X'01' when the list contains character references, and is set to X'00' when the list contains arithmetic references.

Matrix Reference PMC Subroutine—BMATXR

BMATXR generates stack-update-matrix-descriptor PMC sequences (Figure 3-117) for the processing of arithmetic array references appearing in all MAT statements. These PMC sequences are written to virtual memory. The array reference can be a simple array name or an array name redimensioned by one or two dimension expressions.

The virtual address of the array name is determined by the symbol translator subroutine (BDSYMB); and stack-expression-values for dimension expressions, if present, are generated by the arithmetic expression subroutine (BFSCAN).

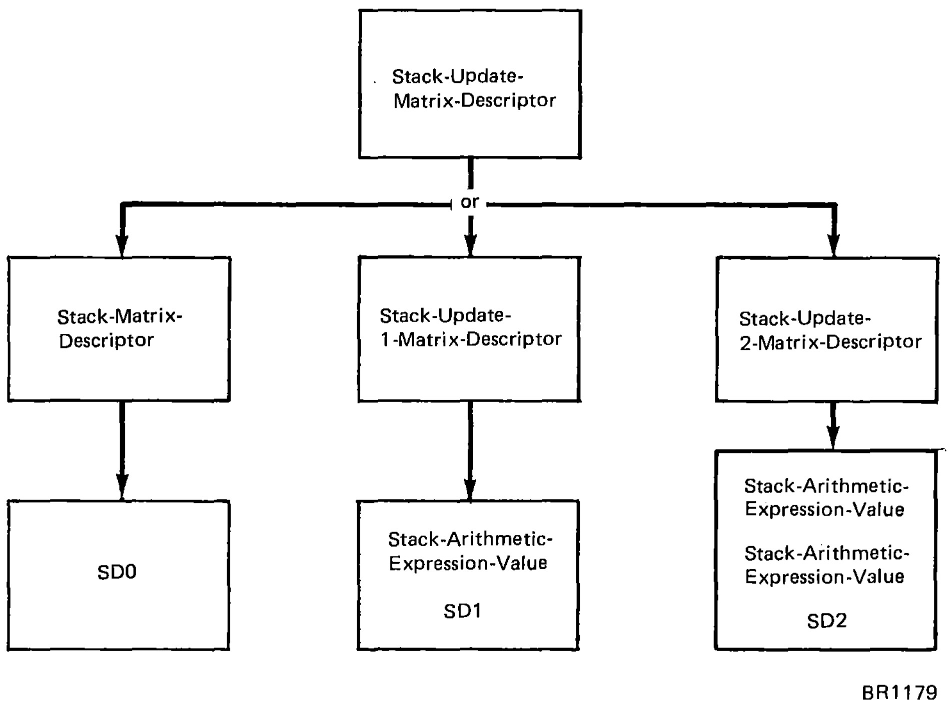


Figure 3-117. Stack-Update-Matrix-Descriptor

Input Text Pointer: Index register @XR contains the core address of the character preceding the first character of the array reference on entry. This register contains the core address of the character that delimits the array reference on exit.

Branch Table Subroutine—BRATAB

BRATAB resolves virtual addresses for branch pseudo instructions and builds the branch address table (Figure 3-107). (Refer to "Resolving Virtual Memory Addresses.")

Input parameters to BRATAB are:

1. B$BRVA–Contains the virtual address that points to the location of the unresolved operand.
2. B$BRLN–Contains a line number or actual virtual address referenced by the pseudo instruction with the unresolved operand.

Disk Four-Track Logical IOCS Interface–BVDL4T

BVDL4T is called by the statement input subroutine (BAGETC) to read blocks of source text from the work file and by the virtual memory output subroutine (BBPUTC) to write pages of PMC to virtual memory. This I/O subroutine converts relative disk addresses to physical disk addresses and calls DKDISK in the system nucleus to perform the disk I/O operation. The calling sequence, disk parameter list (DPL) format, and functions of BVDL4T are the same as for DL4ICS (system work file IOCS). Refer to DL4ICS (Figure 3-70) and disk parameter list (Figure 3-3).

PMC Statement Processors (General Specifications)

Statements are assumed to be free of syntax errors, since they are checked and assigned type codes as they are entered into the system work file. Each source BASIC program statement is scanned, character by character, in the compiler input buffer. This buffer is managed by the statement input subroutine (BAGETC). Index register @XR, updated by all modules of the compiler, generally contains the core address of the current character to be inspected in the input buffer. On entry to a PMC generator overlay, the index register normally references the first character of the statement keyword. When returning to the compiler distributor (BHDIST), the index register references the character terminating the processed statement.

Most PMC generator overlays generate a series of pseudo instructions and assign locations for data in virtual memory. Pseudo object code sequences for each BASIC statement are stored, as generated, contiguously in a 256-byte output buffer. This buffer is managed by the virtual-memory output subroutine (BBPUTC).

The pseudo instructions and/or data area assignments for the following group of BASIC statement syntactical units are generated by core-resident subroutines called by the PMC generator overlays:

1. Constant; constant generator subroutine (BCFCON).
2. Arithmetic-variable; symbol translator subroutine (BDSYMB).
3. Arithmetic-expression; arithmetic expression PMC subroutine (BFSCAN).
4. List of variable-references (arithmetic or character); assignment list PMC subroutine (BLISTA).
5. Character-expression; character expression PMC subroutine (BECSCN).
6. Array-dimension-specification; symbol translator subroutine (BDSYMB).
7. Array-reference; matrix reference PMC subroutine (BMATXR).

The core-resident subroutines call other core-resident subroutines to process lesser elements of the syntactical unit (example: BFSCAN calls BDSYMB to process each symbol in an arithmetic expression). BASIC statement syntactical units, not in the preceding list, are generated by the PMC generator overlays (example: branch, compare, and unstack instructions). Exceptions to this are the STH and EOP instructions. The statement header (STH) is generated by the compiler distributor (BHDIST). End-of-page (EOP) instructions are inserted by the virtual-memory output subroutine (BBPUTC).

A stack-basic-element (Figure 3-118) can be either a stack-arithmetic-expression-value or a stack-character-expression-field. Refer to Figures 3-146 and 3-148 for the stack-basic-element syntax used in the PRINT and PUT keyword statements.

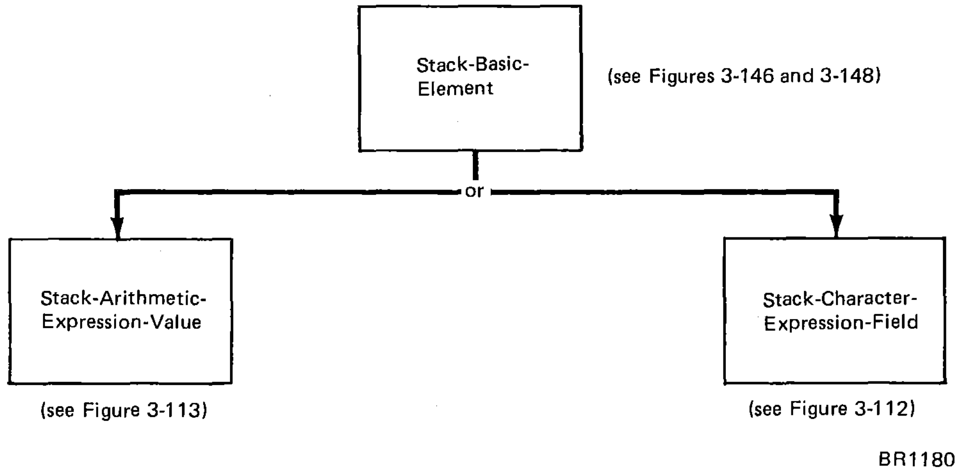


Figure 3-118. Stack-Basic-Element

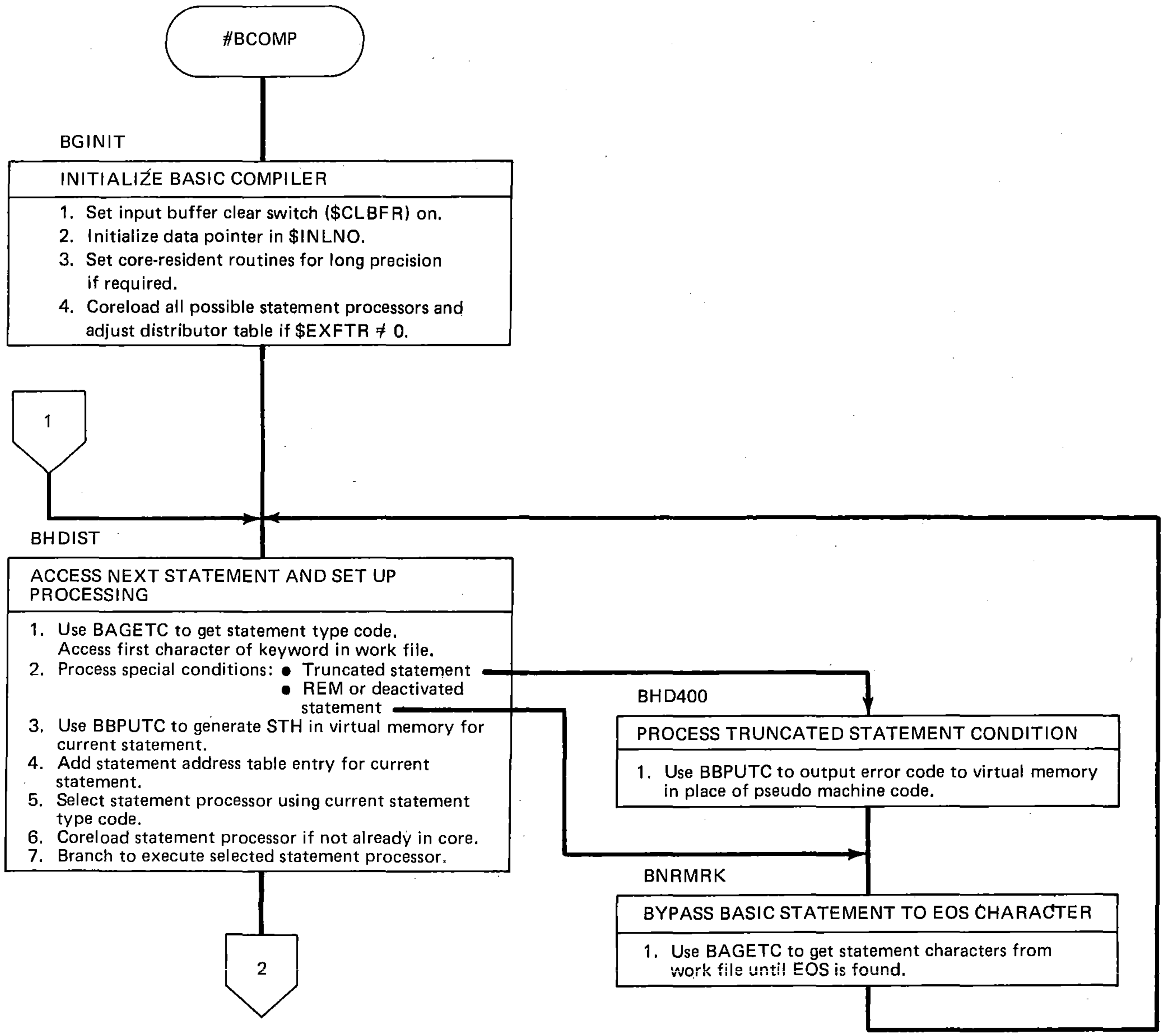


Figure 3-119. Compiler (#BCOMP) Flowchart (Part 1 of 2)

2

Statement	Routine
CLOSE	BXCLOS
DATA	BNDATA
DEF	BNFDEF
DIM	BNADIM
FOR	BKFORX
GET	BXGETX
GOSUB	BKSUBG
GOTO (Simple)	BKGOTO
GOTO (Multiple)	BKMGTO
IF (Arithmetic)	BKARIF
IF (Character)	BKCRIF
IF (Character, String)	BSTRIF
Image (:)	BNIMAG
INPUT	BXINPT
LET (Arithmetic, Simple)	BPALET
LET (Arithmetic, Multiple)	BPMLET
LET (Character)	BPCLET
LET (Character, String)	BSTRLT
MAT	BMMATE
MAT GET	BMGETX
MAT INPUT	BMINPT
MAT PRINT	BMDPRT
MAT PRINT USING	BMUPRT
MAT PUT	BMPUTX
MAT READ	BMREAD
NEXT	BKNEXT
PAUSE	BTPAUS
PRINT	BXDPRT
PRINT USING	BXUPRT
PUT	BXPUTX
READ	BPREAD
RESET	BXRSET
RESTORE	BPXRSR
RETURN	BKRTRN
STOP	BTSTOP
END (or EOF)	

BTRMNT

TERMINATE COMPILER PMC GENERATION PHASE

1. If any compiler-generated errors in virtual memory, coreload error codes and exit to $CAERK to load #ERRPG.
2. If incomplete FOR loop or virtual memory capacity exceeded, exit to $CAERK to load #ERRPG.
3. Close PMC output to virtual memory.
4. Output residual constants to virtual memory.
5. Output residual branch table entries to disk.
6. Output residual statement table entries to disk.
7. Establish parameters and symbol tables in core transfer area for use by #LOADR.
8. Exit to $RLOAD to load and execute #LOADR.

#LOADR
Figure 3-161

SCAN BASIC STATEMENT TO EOS CHARACTER

1. Use BAGETC to get statement characters from work file.
2. Use BBPUTC to output generated PMC to virtual memory.
3. Use BCFCON to generate and output constants to virtual memory.
4. Use BDSYMB to create variable symbol addresses.
5. Use BECSCN to generate PMC for character expression.
6. Use BFSCAN to generate PMC for arithmetic expression.
7. Use BLISTA to generate PMC for assignment list.
8. Use BMATXR to generate PMC for matrix references.
9. Use BRATAB to add unresolved addresses to branch table.
10. Use BUZDBN for decimal to binary conversion.
11. Use BVDL4T as logical interface to disk IOCS.
12. Use BBPUTC to output error codes to virtual memory rather than PMC when minor compiler error occurs.
13. Exit to $CAERK to load #ERRPG if virtual memory or branch table capacity is exceeded.

1

PMC=pseudo machine code

Figure 3-119. Compiler (#BCOMP) Flowchart (Part 2 of 2)

Pseudo Instruction Sequences

Pseudo instruction sequences are detailed in Figures 3-120 through 3-154. These figures are in alphabetical order by BASIC statement keyword. As illustrated in the figures, a syntactical unit of PMC (pseudo machine code) is normally generated for each syntactical unit of the BASIC statement.

Input to BXCLOS (BASIC Statement Syntax)	Output from BXCLOS	
	Syntax of PMC Sequences	PMC Mnemonics
CLOSE { 'filename' / character-variable }	statement-header stack-character-field perform-file-activation close-file	STH STC ADF CLS
[, { 'filename' / character-variable }] . . .	[stack-character-field perform-file-activation close-file] . . .	[STC ADF CLS] . . .
Note: When 'filename' has not been defined (cannot be located in compile filename table), no ADF/CLS sequence is generated for that file reference.		

BR1182A

Figure 3-120. CLOSE PMC Syntax

Input to BNDATA (BASIC Statement Syntax)	Output from BNDATA	
	Syntax of PMC Sequences	PMC Mnemonics
DATA constant	statement-header branch-next-statement define-constant-address	STH BRA DCA
[,constant] . . .	[define-constant-address] . . .	[DCA] . . .
	define-data-linkage	DDL

BR1183

Figure 3-121. DATA PMC Syntax

Input to BNFDEF (BASIC Statement Syntax)	Output from BNFDEF	
	Syntax of PMC Sequences	PMC Mnemonics
DEF user-function	statement-header branch-next-statement branch-return-address	STH BRA BRA
(arithmetic-variable)	(packed-floating-parameter-area)	DWA
= arithmetic-expression	stack-arithmetic-expression-value branch-and-delete-function-entry	(see Figure 3-113) BRD

BR1184

Figure 3-122. DEF PMC Syntax

Input to BNADIM (BASIC Statement Syntax)	Output from BNADIM	
	Syntax of PMC Sequences	PMC Mnemonics
DIM array-dimension-specification	statement-header array-dope-vectors	STH
[,array-dimension-specification] . . .	[array-dope-vectors] . . .	

Notes:

1. No pseudo machine instructions, except a statement-header, are generated for the DIM statement.
2. Refer to Figures 3-156 and 3-157 for descriptions of array dope vectors. Partial array dope vector images remain core-resident during compilation and are tagged or filled as the array is referenced during execution of the BASIC program. Completed dope vectors are stored in virtual memory by #LOADR.

BR1185

Figure 3-123. DIM PMC Syntax

Input to BTRMNT (BASIC Statement Syntax)	Output from BTRMNT	
	Syntax of PMC Sequences	PMC Mnemonics
END [comment]	statement-header call-supervisor define-program-end	STH SVC EOF

Note: End of work file (condition) generates the same sequence.

BR1186

Figure 3-124. END PMC Syntax

Input to BKFORX (BASIC Statement Syntax)	Output from BKFORX	
	Syntax of PMC Sequences	PMC Mnemonics
FOR arithmetic-variable	statement-header	STH
= arithmetic-expression	stack-arithmetic-expression-value	(see Figure 3-113)
TO arithmetic-expression	stack-arithmetic-expression-value	(see Figure 3-113)
[STEP arithmetic-expression]	stack-arithmetic-expression-value	(see Figure 3-113)
	initialize-for-loop perform-next-step define-work-area (unpacked-floating-parameter-area)	FOR NXT DWA (see note)

Note: Each unpacked-floating-point-parameter-area is 8 bytes in length for standard precision and 16 bytes in length for long precision.

BR1187

Figure 3-125. FOR PMC Syntax

Input to BXGETX (BASIC Statement Syntax)	Output from BXGETX	
	Syntax of PMC Sequences	PMC Mnemonics
GET {'filename' / character-variable}	statement-header stack-character-field perform-file-activation	STH STC ADF
,variable reference	stack-variable-address get-file-element	(see Figure 3-116) GET
[,variable reference] . . .	[stack-variable-address get-file-element] . . .	[(see Figure 3-116) GET] . . .

Figure 3-126. GET PMC Syntax

Input to BKSUBG (BASIC Statement Syntax)	Output from BKSUBG	
	Syntax of PMC Sequence	PMC Mnemonics
GOSUB line-number	statement-header stack-return-address branch-unconditionally	STH STA BRA

Figure 3-127. GOSUB PMC Syntax

Input to BKMGTO (BASIC Statement Syntax)	Output from BKMGTO	
	Syntax of PMC Sequences	PMC Mnemonics
GOTO line-number	statement-header stack-bypass-address stack-line-address	STH STA STA
[,line-number] . . .	[stack-line-address] . . .	[STA] . . .
ON arithmetic-expression	stack-arithmetic-expression-value compute-stacked-address branch-stacked-address	(see Figure 3-113) CSA BRS

Figure 3-128. GOTO (Multiple) PMC Syntax

Input to BKGOTO (BASIC Statement Syntax)	Output from BKGOTO	
	Syntax of PMC Sequence	PMC Mnemonics
GOTO line-number	statement-header branch-unconditionally	STH BRA

Figure 3-129. GOTO (Simple) PMC Syntax

Input to BKARIF (BASIC Statement Syntax)	Output from BKARIF	
	Syntax of PMC Sequences	PMC Mnemonics
IF arithmetic-expression	statement-header stack-arithmetic-expression-value	STH (see Figure 3-113)
relational-operator arithmetic-expression	stack-arithmetic-expression-value compare-stacked-values	(see Figure 3-113) CMF
THEN GO TO line-number	branch-on-condition	BRC

Figure 3-130. IF (Arithmetic) PMC Syntax

Input to BKCRIF (BASIC Statement Syntax)	Output from BKCRIF	
	Syntax of PMC Sequences	PMC Mnemonics
IF character-expression	statement-header stack-character-expression-field	STH (see Figure 3-112)
relational-operator character-expression	stack-character-expression-field compare-stacked-values	(see Figure 3-112) CMC
THEN GO TO line-number	branch-on-condition	BRC

Figure 3-131. IF (Character) PMC Syntax

Input to BSTRIF (BASIC Statement Syntax)	Output from BSTRIF	
	Syntax of PMC Sequences	PMC Mnemonics
IF character-expression	statement-header stack-character-expression-field [stack-arithmetic-expression-value stack-arithmetic-expression-value function-call-no-argument]	STH (see Figure 3-112) [STF STF FN0]
relational-operator character-expression	stack-character-expression-field [stack-arithmetic-expression-value stack-character-expression-value function-call-no-argument] compare-stacked-values	(see Figure 3-112) [STF STF FN0] CMC
THEN GO TO line-number	branch-on-condition	BRC

Figure 3-131.1. IF (Character, String) PMC Syntax

<table>
<tr><th rowspan="2">Input to BNIMAG
(BASIC Statement Syntax)</th><th colspan="2">Output from BNIMAG</th></tr>
<tr><th>Syntax of PMC Sequences</th><th>PMC Mnemonics</th></tr>
<tr><td></td><td>statement-header
branch-next-statement</td><td>IMH
BRA</td></tr>
<tr><td>[character-string
or
print-image] . . .</td><td>

set-print-image
branch-stacked-address</td><td>

(see note)
BRS</td></tr>
<tr><td colspan="3">Note: Set-print-image is either a set-null-image (PRU) or a sequence of the following form:
stack-character-field (STC)
set-initial-image (PRU)
[stack-character-field [STC
set-image-field] . . . PRU] . . .</td></tr>
</table>

Figure 3-132. IMAGE (:) PMC Syntax

<table>
<tr><th rowspan="2">Input to BXINPT
(BASIC Statement Syntax)</th><th colspan="2">Output from BXINPT</th></tr>
<tr><th>Syntax of PMC Sequences</th><th>PMC Mnemonics</th></tr>
<tr><td>INPUT variable-reference</td><td>statement-header
stack-return-address
branch-unconditionally
stack-variable-address
input-data-element</td><td>STH
STA
BRA
(see Figure 3-116)
GET</td></tr>
<tr><td>[,variable-reference] . . .</td><td>[stack-variable-address
input-data-element] . . .
branch-next-statement
stack-execution-code
[stack-execution-code] . . .
initiate-data-input
branch-stacked-address</td><td>[(see Figure 3-116)
GET] . . .
BRA
STX
[STX] . . .
INI
BRS</td></tr>
</table>

Figure 3-133. INPUT PMC Syntax

<table>
<tr><th rowspan="2">Input to BPMLET
(BASIC Statement Syntax)</th><th colspan="2">Output from BPMLET</th></tr>
<tr><th>Syntax of PMC Sequences</th><th>PMC Mnemonics</th></tr>
<tr><td>[LET] arithmetic-reference</td><td>statement-header
branch-unconditionally
stack-arithmetic-address
stack-scalar-value
unstack-scalar-value</td><td>STH
BRA
(see Figure 3-116)
STF
USF</td></tr>
<tr><td>[,arithmetic-reference] . . .</td><td>[stack-arithmetic-address
stack-scalar-value
unstack-scalar-value] . . .</td><td>[(see Figure 3-116)
STF
USF] . . .</td></tr>
<tr><td>= arithmetic-expression</td><td>branch-to-next-statement
stack-scalar-address
stack-expression-value
unstack-expression-value
branch-unconditionally</td><td>BRA
STA
(see Figure 3-113)
USF
BRA</td></tr>
</table>

Figure 3-134. LET (Arithmetic, Multiple) PMC Syntax

Input to BPALET (BASIC Statement Syntax)	Output from BPALET	
	Syntax of PMC Sequences	PMC Mnemonics
[LET] arithmetic-reference	statement-header stack-arithmetic-address	STH (see Figure 3-116)
= arithmetic-expression	stack-arithmetic-expression-value unstack-arithmetic-expression-value	(see Figure 3-113) USF

Figure 3-135. LET (Arithmetic, Simple) PMC Syntax

Input to BPCLET (BASIC Statement Syntax)	Output from BPCLET	
	Syntax of PMC Sequences	PMC Mnemonics
[LET] character-reference	statement-header stack-character-address	STH (see Figure 3-116)
[,character-reference] . . .	[stack-character-address] . . .	[(see Figure 3-116)] . . .
= character-expression	stack-character-expression-field unstack-character-expression-field	(see Figure 3-112) USC

Figure 3-136. LET (Character) PMC Syntax

Input to BSTRLT (BASIC Statement Syntax)	Output from BSMLET	
	Syntax of PMC Sequences	PMC Mnemonics
[LET] { character-reference / string-function }	statement-header branch-unconditionally stack-character-address [stack-character-expression-field stack-arithmetic-expression-value stack-arithmetic-expression-value] stack-character-field [function-call-no-argument] unstack-character-elements	STH BRA (see Figure 3-116) [(see Figure 3-112) STF STF] STC [FNO] USC
[{ character-reference / string-function }] . . .	[stack-character-address [stack-character-expression-field stack-arithmetic-expression-value stack-arithmetic-expression-value] stack-character-field [function-call-no-argument] unstack-character-elements]	[(see Figure 3-116) [(see Figure 3-112) STF STF] STC [FNO] USC]
= character-expression	branch-to-next-statement stack-character-address stack-character-expression-field [stack-arithmetic-expression-value stack-arithmetic-expression-value function-call-no-argument] unstack-character-elements branch-unconditionally	BRA STA (see Figure 3-112) [STF STF FNO] USC BRA

Figure 3-136.1. LET (Character, Multiple, String) PMC Syntax

Input to BMMATA	Output from BMMATA	
MAT matrix-name = {matrix-name / matrix-expression}	Syntax of PMC Sequences	PMC Mnemonics
MAT Statement Example		
MAT C = A + B	statement-header stack-matrix-descriptor stack-matrix-descriptor stack-matrix-descriptor perform-3-matrix-function	STH SD0 SD0 SD0 MF3
MAT C = INV(M)	statement-header stack-matrix-descriptor stack-matrix-descriptor perform-2-matrix-function	STH SD0 SD0 MF2
MAT C = CON(10)	statement-header stack-arithmetic-expression-value stack-update-1-matrix-descriptor perform-1-matrix-function	STH (see Figure 3-113) (see Figure 3-117) MF1
MAT C = (E1)*M	statement-header stack-matrix-descriptor stack-arithmetic-expression-value stack-matrix-descriptor perform-scalar-matrix-multiply	STH SD0 (see Figure 3-113) SD0 MSM

Figure 3-137. MAT PMC Syntax

Input to BMGETX (BASIC Statement Syntax)	Output from BMGETX	
	Syntax of PMC Sequences	PMC Mnemonics
MAT GET {'filename' / character-variable}	statement-header stack-character-field perform-file-activation	STH STC ADF
,array-reference	stack-update-matrix-descriptor perform-get-matrix-function	(see Figure 3-117) MF1
[,array reference] . . .	[stack-update-matrix-descriptor perform-get-matrix-function] . . .	[(see Figure 3-117) MF1] . . .

Figure 3-138. MAT GET PMC Syntax

Input to BMINPT (BASIC Statement Syntax)	Output from BMINPT	
	Syntax of PMC Sequences	PMC Mnemonics
MAT INPUT array-reference	statement-header stack-update-matrix-descriptor perform-input-matrix-function	STH (see Figure 3-117) MF1
[,array reference] . . .	[stack-update-matrix-descriptor perform-input-matrix-function] . . .	[(see Figure 3-117) MF1] . . .

Figure 3-139. MAT INPUT PMC Syntax

Input to BMDPRT (BASIC Statement Syntax)	Output from BMDPRT	
	Syntax of PMC Sequences	PMC Mnemonics
MAT PRINT matrix-name	statement-header stack-matrix-descriptor perform-print-matrix-unformatted	STH SD0 MF1
[{; ,} matrix-name] . . . [{; ,}]	[stack-matrix-descriptor perform-print-matrix-unformatted] . . .	[SD0 MF1] . . .

Figure 3-140. MAT PRINT PMC Syntax

Input to BMUPRT (BASIC Statement Syntax)	Output from BMUPRT	
	Syntax of PMC Sequences	PMC Mnemonics
MAT PRINT USING line-number	statement-header stack-return-address branch-set-image	STH STA BNX
,matrix-name	stack-matrix-descriptor perform-print-matrix-formatted	SD0 MF1
[,matrix-name] . . .	[stack-matrix-descriptor perform-print-matrix-formatted] . . .	[SD0 MF1] . . .
	release-image	PRU

Figure 3-141. MAT PRINT USING PMC Syntax

Input to BMPUTX (BASIC Statement Syntax)	Output from BMPUTX	
	Syntax of PMC Sequences	PMC Mnemonics
MAT PUT {'filename' / character-variable}	statement-header stack-character-field perform-file-activation	STH STC ADF
,array reference	stack-matrix-descriptor perform-put-matrix-function	SD0 MF1
[,array reference] . . .	[stack-matrix-descriptor perform-put-matrix-function] . . .	[SD0 MF1] . . .

Figure 3-142. MAT PUT PMC Syntax

Input to BMREAD (BASIC Statement Syntax)	Output from BMREAD	
	Syntax of PMC Sequences	PMC Mnemonics
MAT READ array-reference	statement-header stack-update-matrix-descriptor perform-read-matrix-function	STH (see Figure 3-117) MF1
[,array-reference] . . .	[stack-update-matrix-descriptor perform-read-matrix-function] . . .	[(see Figure 3-117) MF1] . . .

Figure 3-143. MAT READ PMC Syntax

Input to BKNEXT (BASIC Statement Syntax)	Output from BKNEXT	
	Syntax of PMC Sequence	PMC Mnemonics
NEXT arithmetic-variable	statement-header branch-unconditionally	STH BRA

Figure 3-144. NEXT PMC Syntax

Input to BTPAUS (BASIC Statement Syntax)	Output from BTPAUS	
	Syntax of PMC Sequence	PMC Mnemonics
PAUSE [comment]	statement-header halt-execution	STH HLT

Figure 3-145. PAUSE PMC Syntax

Input to BXDPRT (BASIC Statement Syntax)	Output from BXDPRT	
	Syntax of PMC Sequences	PMC Mnemonics
PRINT [print-references] . . .	statement-header print-unformatted. . .	STH (see note)
Note: Print-unformatted is either a position-carrier (PRS) or a print-position-carrier (PRS) preceded by a stack-basic-element (Figure 3-118).		

BR1208

Figure 3-146. PRINT PMC Syntax

Input to BXUPRT (BASIC Statement Syntax)	Output from BXUPRT	
	Syntax of PMC Sequences	PMC Mnemonics
PRINT USING line-number	statement-header stack-return-address branch-set-image	STH STA BNX
	print-image-only	PRU (see Note 1)
,scalar-reference [,scalar-reference] . . .	print-formatted [print-formatted] . . .	STC PRU (see Note 2) [STC PRU] . . .
Notes: 1. This instruction is not generated when at least one scalar-reference is specified. 2. These instructions are generated only when at least one scalar-reference is specified.		

BR1209

Figure 3-147. PRINT USING PMC Syntax

Input to BXPUTX (BASIC Statement Syntax)	Output from BXPUTX	
	Syntax of PMC Sequences	PMC Mnemonics
PUT { 'filename' | character-variable }	statement-header stack-character-field perform-file-activation	STH STC ADF
,scalar-reference	stack-basic-element put-file-element	(see Figure 3-118) PUT
[,scalar-reference] . . .	[stack-basic-element put-file-element] . . .	[(see Figure 3-118) PUT] . . .

BR1210A

Figure 3-148. PUT PMC Syntax

Input to BPREAD (BASIC Statement Syntax)	Output from BPREAD	
	Syntax of PMC Sequences	PMC Mnemonics
READ	statement-header	STH
variable-reference	stack-variable-address read-data-element	(see Figure 3-116) GET
[,variable-reference] . . .	[stack-variable-address read-data-element] . . .	[(see Figure 3-116) GET] . . .

BR1211

Figure 3-149. READ PMC Syntax

Input to BNRMRK (BASIC Statement Syntax)	Output from BNRMRK	
	Syntax of PMC Sequence	PMC Mnemonics
REM [comment]	statement-header	STH

BR1212

Figure 3-150. REM PMC Syntax

Input to BXRSET (BASIC Statement Syntax)	Output from BXRSET	
	Syntax of PMC Sequences	PMC Mnemonics
RESET {'filename' | character-variable}	statement-header stack-character-field perform-file-activation reset-file-pointer	STH STC ADF RST
[{'filename' | character-variable}] . . .	[stack-character-field perform-file-activation reset-file-pointer] . . .	[STC ADF RST] . . .
Note: When 'filename' has not been defined (cannot be located in compile filename table), no ADF-RST sequence is generated for that file reference.		

BR1213A

Figure 3-151. RESET PMC Syntax

Input to BPXRSR (BASIC Statement Syntax)	Output from BPXRSR	
	Syntax of PMC Sequence	PMC Mnemonics
RESTORE [comment]	statement-header restore-data-pointer	STH RSR

BR1214

Figure 3-152. RESTORE PMC Syntax

Input to BKRTRN (BASIC Statement Syntax)	Output from BKRTRN	
	Syntax of PMC Sequence	PMC Mnemonics
RETURN [comment]	statement-header branch-stacked-address	STH BRS
Note: The last executed GOSUB stacked the address operated on by BRS.		

BR1215

Figure 3-153. RETURN PMC Syntax

Input to BTSTOP (BASIC Statement Syntax)	Output from BTSTOP	
	Syntax of PMC Sequence	PMC Mnemonics
STOP [comment]	statement-header call-supervisor	STH SVC

BR1216

Figure 3-154. STOP PMC Syntax

Compiler Termination

Compiler Terminator–BTRMNT: The compiler overlay is called by the compiler distributor (BHDIST) when an END statement or a work file end-of-file record is processed.

Functions performed by the compiler terminator are:

1. Generate the PMC sequence for the END statement (Figure 3-124).
2. Write the last page of pseudo instructions to virtual memory by calling BBPUTC (CLOSE function).
3. Write the last page of constants to virtual memory by calling BBPUTC (WRITE PAGE function).
4. Write the last statement address table and branch address table buffers to disk.
5. Build the common parameter area (Figure 3-155) in high-core.
6. Load and exit to the loader (#LOADR) if nothing occurred to abort execution of the BASIC program.

The following error conditions abort execution of the BASIC program and call the error program (#ERRPG) via $CAERK in the system nucleus:

1. BASIC program errors have been detected by the compiler. Switch B$ERSW is on, and the errors are recorded beginning in the first pseudo instruction page of virtual memory. This page and the two pages following are read into core at X'1C00', the location of the error stack for the error program (#ERRPG). These pages contain up to 255 stacked error records.
2. The capacity of the branch address table file on disk is exceeded.
3. The FOR loop table contains an unresolved entry (a FOR statement was not paired with a matching NEXT statement).

Core Address of Leftmost Parameter Byte	Decimal Length	Loader Input Parameters
1A00	2	Starting virtual address for the allocation of arrays (equal to the last pseudo instruction page + 1).
1A02	2	Last virtual address available in the first area for the allocation of arrays (equal to the last, or lowest, page constants).
1A04	2	First virtual address available in the second area for the allocation of arrays (equal to the last page of variables +1).
1A06	2	Ending virtual address for the alloca^ion of arrays (equal to the last, or lowest, page containing array dope vectors).
1A08	2	Starting virtual address of the internal constants.
1A0A	2	Starting virtual address of the internal variables.
1A0C	58	Arithmetic (letter) variable symbol table (from label B$SLVT).
1A46	580	Arithmetic (letter-digit) variable symbol table (from label B$SLDT).
1C8A	58	Character variable symbol table (from label B$SCVT).
1CC4	58	Arithmetic array symbol table (from label B$SNAT).
1CFE	58	Character array symbol table (from label B$SCAT).
1D38	58	User function symbol table (from label B$SFNT).
1D72	406	Array dope vector images and user function entry addresses. This area contains all array descriptors defined in the program, including dimensions specified in DIM statements and tags to define the arithmetic arrays as vector or matrix arrays. This area also includes virtual address entry points for all functions defined with a DEF statement.
1F07		Last address occupied by the loader parameters.

Notes:

1. For clarification of the areas for the allocation of arrays, refer to the virtual memory map (Figure 7-2).
2. Symbol tables and array dope vectors are generated in the symbol translation subroutine (BDSYMB). Refer to symbol processing in BDSYMB (Figure 3-111).

BR1217

Figure 3-155. Compiler/Loader Common Parameter Area

LOADER—SECOND PHASE OF COMPILATION—#LOADR (Figure 3-161)

#LOADR is called by the compiler terminator (BTRMNT), via $RLOAD in the system nucleus, upon completion of the first phase of the compilation. The loader performs the following functions in preparation for execution of the BASIC program:

1. Allocation of arrays in virtual memory
2. Allocation of data file buffers in virtual memory
3. Initialization of elements in virtual memory
4. Resolution of all entries in the branch address table
5. Loading of VM-resident execution subroutines of the specified precision
6. Loading the interpreter to begin execution of the BASIC program

The assembly of #LOADR contains these major source modules:

LALLOC—Allocate arrays
LDFILE—Allocate data file buffers
LVINIT—Initialize elements
LRADDR—Resolve branch address table
LSORTA—Sort branch address table subroutine
DL2ICS—Disk logical IOCS, Figure 3-70
DL4ICS—System work area IOCS, Figure 3-70

The loader references parameters and tables accumulated by the compiler to perform the functions described in the following paragraphs. The loader does not access any of the source information in the work file (BASIC statements). The following list of figure references will aid in determining the input to, and output from, this phase of the compile:

1. Compiler/loader common parameter area, Figure 3-155
2. Virtual memory map, Figure 7-2
3. RUN program name core map, Figure 3-102
4. Arithmetic array dope vector, Figure 3-156
5. Character array dope vector, Figure 3-157
6. Symbol tables in BDSYMB, Figure 3-111
7. Directory-1 (work file I/O record), Figure 5-17
8. Directory-2, Figure 5-20

Allocation of Arrays in Virtual Memory—LALLOC

LALLOC allocates all arithmetic and character arrays, specified by entries in the respective array symbol tables, into the remaining available pages of virtual memory. Reference is made to the following parameters in the common parameter area (Figure 3-155):

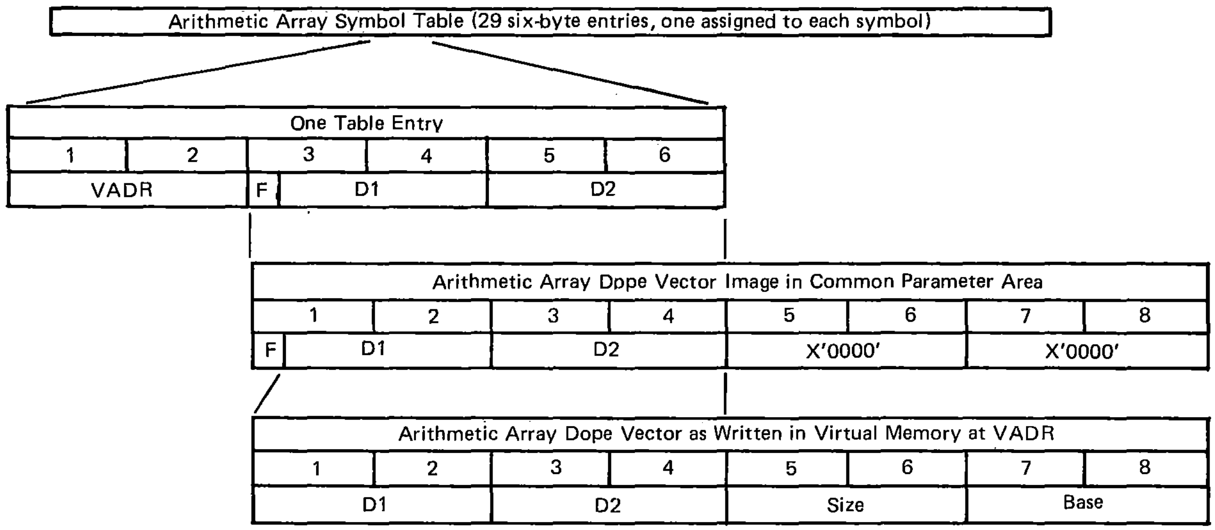
1. The first four parameters define the two areas available for the allocation of arrays. These parameters are updated, as arrays are allocated, so that they always reflect the limits of the remaining available area.
2. The arithmetic array symbol table contains a pointer to an array dope vector image, also in the common area, for each arithmetic matrix or vector array to be allocated. The array dope vector defines the type and size of the array.
3. The character array symbol table contains a pointer to an array dope vector image, also in common area, for each character array to be allocated. The character array dope vector defines the size of the array.

Default values are used if the array dope vector is flagged as undefined. All fields of the array dope vectors are completed in the common parameter area and that portion of the area is written to virtual memory after all arrays are allocated.

The length of each element in the array is:

1. 5 bytes for arithmetic arrays for standard precision
2. 9 bytes for arithmetic arrays for long precision
3. 19 bytes for character arrays

B$SNAT



Notes:

VADR— The virtual address of the space allocated by virtual memory for the array dope vector assigned to this symbol. Until the symbol is referenced at compile-time (arithmetic array reference of DIM statement), this field contains binary 0's.

F— Array usage flags (bits 0 and 1).
00—Array undefined.
10—Vector usage; one dimension; field D1 contains binary 0's.
Field D2 contains either a specified or a default single dimension.
11—Matrix usage; two dimensions; both fields D1 and D2 contain a specified or a default dimension.

D1— First Dimension. This field defaults to a value of 10 when dimensions of a matrix array are not defined.

D2— Second Dimension. This field defaults to a value of 10 when dimension(s) of a matrix array are not defined.

Size— Total number of elements in this array. This field defaults, to a value of 10 for vector usage or 100 for matrix usage, when the dimension(s) of the array are not defined.

Base— Base virtual address for this array. This address is assigned by the loader (#LOADR). The first element in the array is located at base plus 5 bytes (9 bytes for long precision).

BR1218

Figure 3-156. Arithmetic Array Dope Vector

Allocation of Data File Buffers in Virtual Memory—LDFILE

LDFILE reads the first sector of file directory 1 into storage and determines if there is a second sector. If file directory 1 is two sectors long, virtual memory space is allocated for the second sector. The first four parameters of the common parameter area define the available pages in virtual memory.

LDFILE must be able to allocate at least one page for each card and disk file referenced in the BASIC program or execution of the program is aborted. The buffers are allocated from the remaining available pages defined by the first four parameters of the common parameter area. One page is allocated for each card file and the remaining pages are divided equally among the disk files, if specified, to a maximum of eight pages for each disk file.

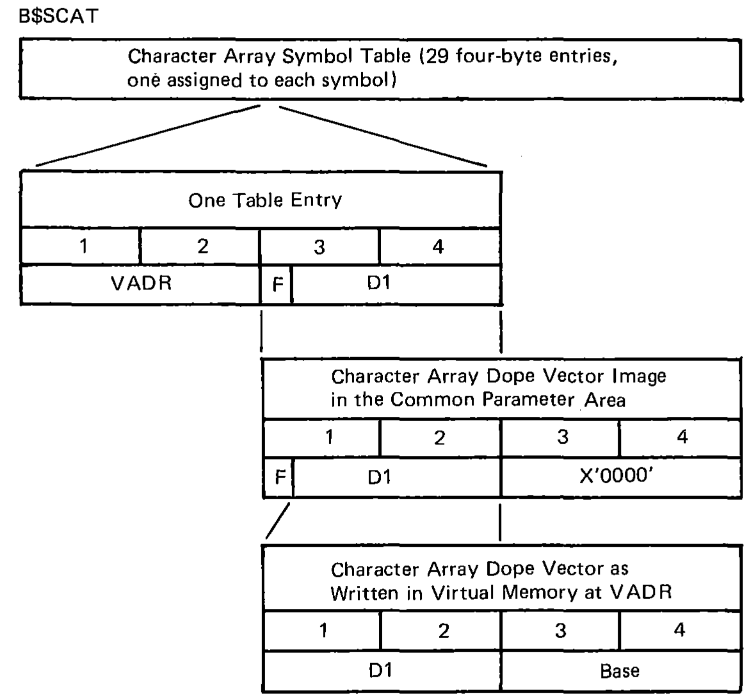
The files are defined in file directory 1 (work file I/O record). The device type code in each entry in file directory 1 is checked and file directory 2 is created as the buffers are allocated. File directory 1 and file directory 2 are stored in virtual memory.

Initialization of Elements in Virtual Memory—LVINIT

LVINIT scans the following tables in the common parameter area and initializes each item that is referenced:

1. Arithmetic array symbol table
2. Character array symbol table

3. Character variable symbol table
4. Arithmetic (letter) variable symbol table
5. Arithmetic (letter-digit) variable symbol table



Notes:

VADR— The virtual address of the space allocated in virtual memory for the array dope vector assigned to this symbol. Until the symbol is referenced (character array reference or DIM statement), this field contains binary 0's.

F— Array usage flag (bit 0).

D1— Dimension; number of character elements in the array. Only single dimension references (vector) are valid. This field defaults to a value of 10 when the dimension of a character array is not defined.

Base— Base virtual address of the array. This address is assigned by the loader (#LOADR). The first character element in this array is located at base plus 19 bytes.

BR1219

Figure 3-157. Character Array Dope Vector

Trace Mode: A trace reference list (256 bytes) contains an image of the input parameters from the TRACE keyword statement. This list is passed against each symbol table listed above. Bits are set in an internal trace table (Figure 3-158) for symbols to be traced, if the symbol is referenced in its corresponding symbol table. The internal trace table is used to set trace bits (X'80' in the first byte) as the elements to be traced are initialized.

Initializing Elements: All arithmetic elements (including each array element) are initialized to a value of 0^{-98}(X'00 00 00 00 1E' in short precision). (Refer to "Floating-Point Arithmetic" for the format of an arithmetic element in virtual memory.) All character elements are initialized to blanks (X'40'). (Refer to Figure 3-110 for the format of a character field in virtual memory.)

Trace Table (29 two-byte entries assigned to symbols $, #, @, A-Z in that order)

One Two-Byte Entry	
Mask	Symbol Type
X'8000'	Arithmetic (letter-digit) variable (digit = 0)
X'4000'	Arithmetic (letter-digit) variable (digit = 1)
X'2000'	Arithmetic (letter-digit) variable (digit = 2)
X'1000'	Arithmetic (letter-digit) variable (digit = 3)
X'0800'	Arithmetic (letter-digit) variable (digit = 4)
X'0400'	Arithmetic (letter-digit) variable (digit = 5)
X'0200'	Arithmetic (letter-digit) variable (digit = 6)
X'0100'	Arithmetic (letter-digit) variable (digit = 7)
X'0080'	Arithmetic (letter-digit) variable (digit = 8)
X'0040'	Arithmetic (letter-digit) variable (digit = 9)
X'0020'	Arithmetic (letter) variable
X'0010'	Character variable
X'0008'	Arithmetic array
X'0004'	Arithmetic array element*
X'0002'	Character array
X'0001'	Character array element*

*Trace reference list is rescanned to determine individual elements to be traced.

BR1220

Figure 3-158. Trace Table

Resolution of the Branch Address Table—LRADDR

LRADDR resolves all entries in the branch address table (Figure 3-159). (Refer to "Resolving Virtual Memory Addresses.") Resolution involves passing the branch address table against the statement address table (Figure 3-159), replacing line numbers in the branch address table with virtual addresses from the statement address table, and then updating the unresolved operands in virtual memory as specified in the branch address table. Both of these tables were written in the system work area on disk by the compiler.

To efficiently replace the line numbers, the branch address table is sorted into ascending line number order (last two bytes of entry) by the sort subroutine (LSORTA). One sector is processed at a time. The statement address table is created in ascending line number order; therefore, it need not be sorted. Each line number in the branch address table is located in the statement address table, and the line number in the branch address table is replaced with its associated virtual address from the statement address table.

It may be necessary to scan more than one sector of the statement address table to locate a line number. If the range of line numbers in the statement address table buffer is higher than the unresolved line number, the scan starts with the first sector of the statement address table. If the range of line numbers in the buffer is lower, the next sequential sector is read from disk.

After all line numbers in one sector of the branch address table have been replaced, the updated sector is again sorted (LSORTA), this time to arrange the entries in ascending virtual-memory-location order (first two bytes of entry). After the sort, the virtual-memory page required by the first entry in the branch address table is read from disk. This page is updated at the displacements indicated by all entries in that range of virtual addresses and then written back to virtual memory.

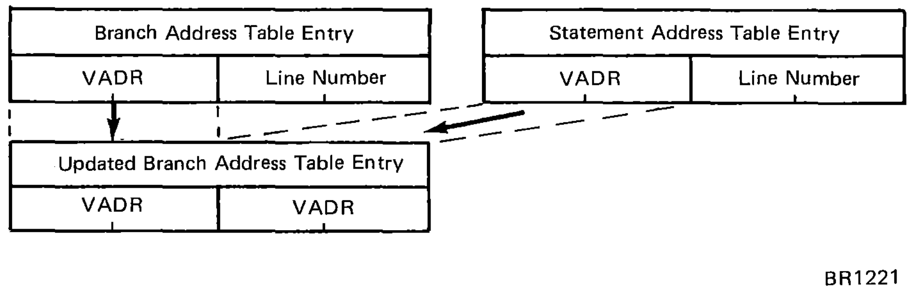


Figure 3-159. Branch and Statement Address Tables

When all entries on one sector of the branch address table have been processed, the next sequential sector is read from disk, sorted by line number, updated from the statement address table, sorted by virtual-memory location, virtual-memory updated, etc. This process continues until all entries in the branch address table (16 sectors maximum) have been resolved.

When the last entry in the table is resolved, the interpreter (#INSTD or #INLNG) is loaded via $RLOAD in the system nucleus. #INSTD is loaded if execution is to be in standard precision; #INLNG is loaded to execute the BASIC program in long precision.

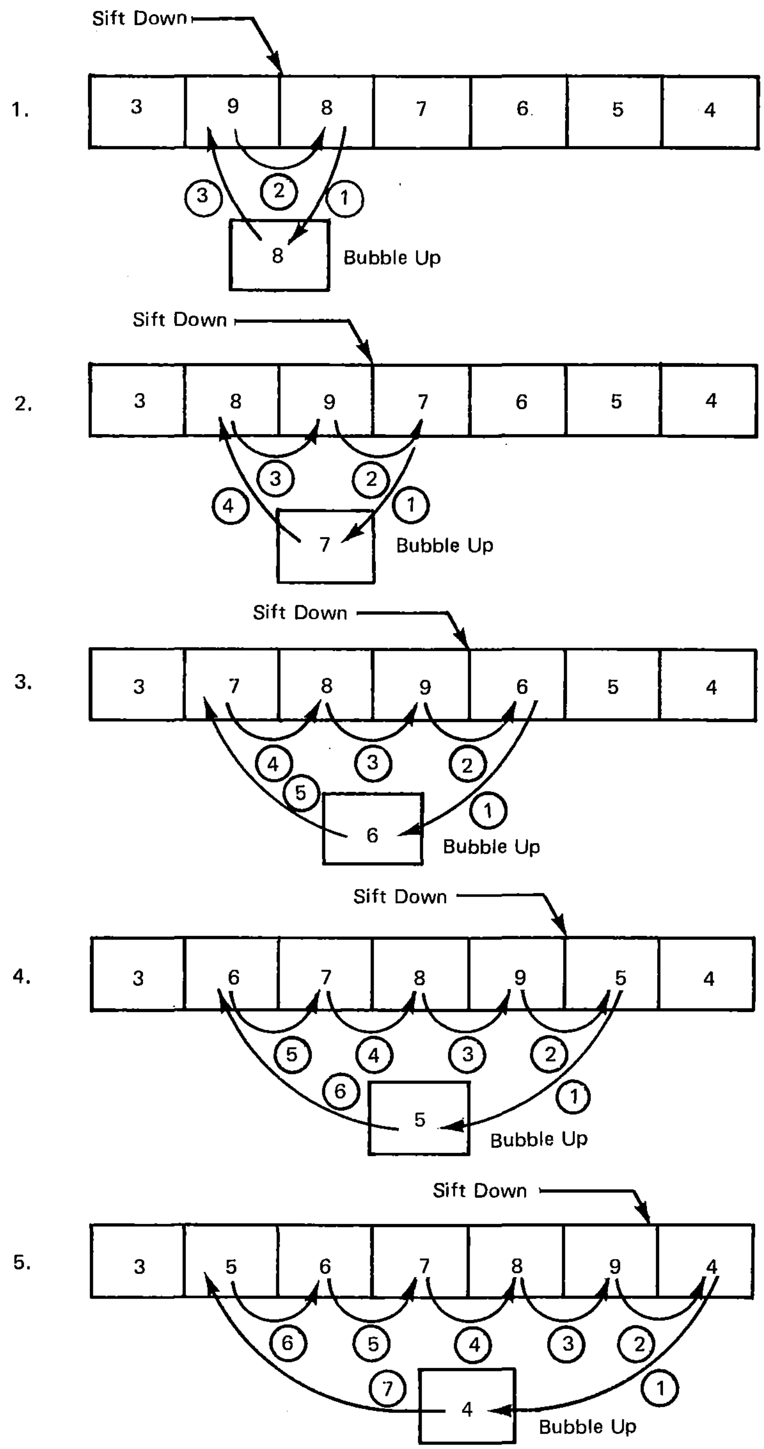
Sorting the Branch Address Table—LSORTA

One 256-byte buffer containing four-byte entries is sorted on each call to LSORTA. The entries are sorted in place, and in ascending order by either the first two or last two bytes of the entries.

Input parameters to LSORTA are:

1. The core address of the two-byte sort field (register @XR).
2. The core address of the buffer to be sorted (register @BR).
3. The core address of the next to the last two-byte sort field in the buffer (LSBOTM). The number of entries in a buffer is variable.

The method used by LSORTA is called sifting down and bubbling up (Figure 3-160). The entries are scanned, from the top, until two sort fields are found that are not in ascending order. This is called sifting down. When out-of-sequence entries occur, they are reversed. The scan of the entries reverses, and entries are swapped until the out-of-sequence entries are in the correct sequence. This is called bubbling up. Sifting down continues from the point where the out of sequence was detected. Only one full forward pass is made over the entries in the buffer. When all entries are in ascending order, this subroutine returns to LRADDR.



6. Continue sifting and bubbling until all entries are in sequence.

BR1222

Figure 3-160. Sift and Bubble Sort (Worst Case)

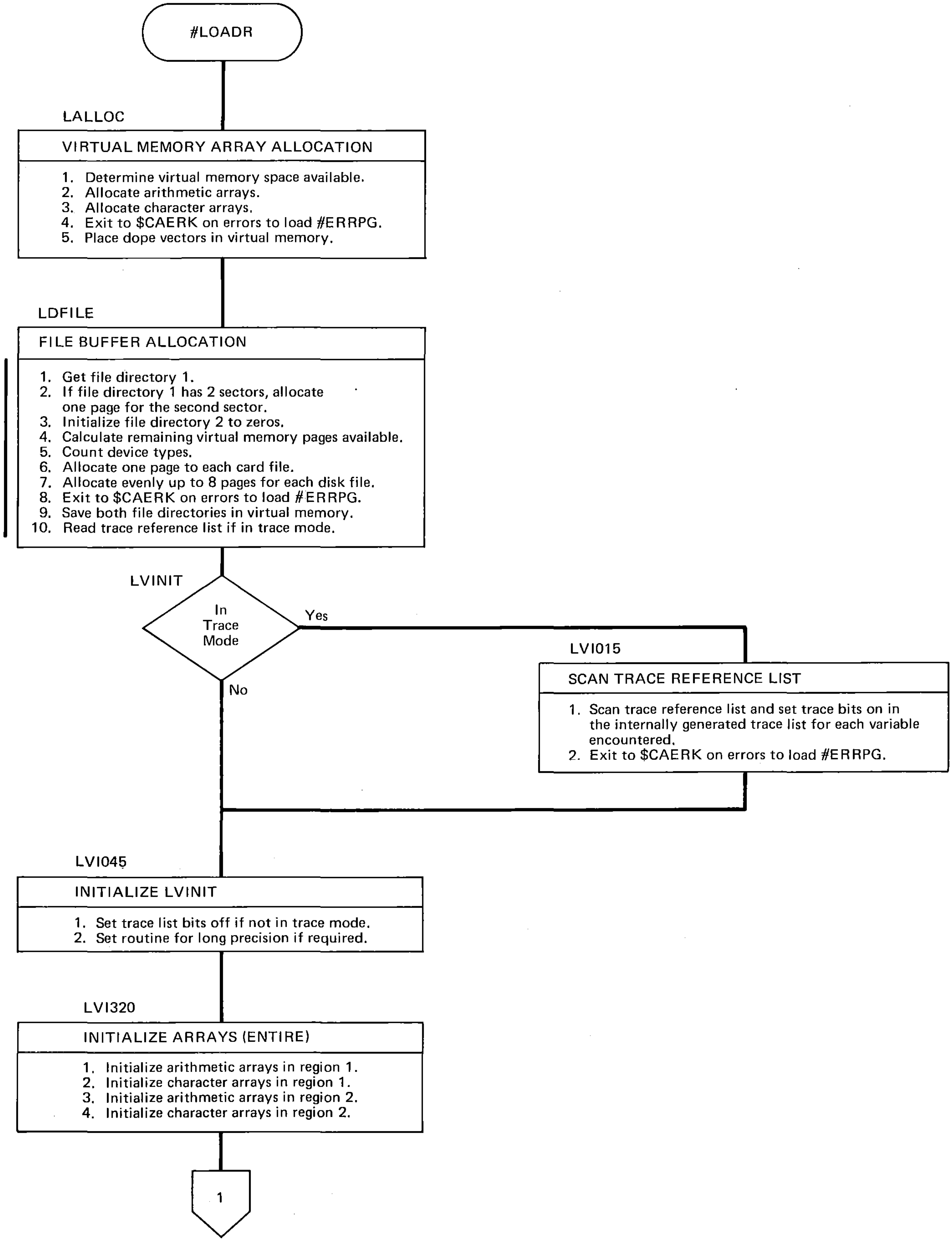


Figure 3-161. Loader (#LOADR) Flowchart (Part 1 of 2)

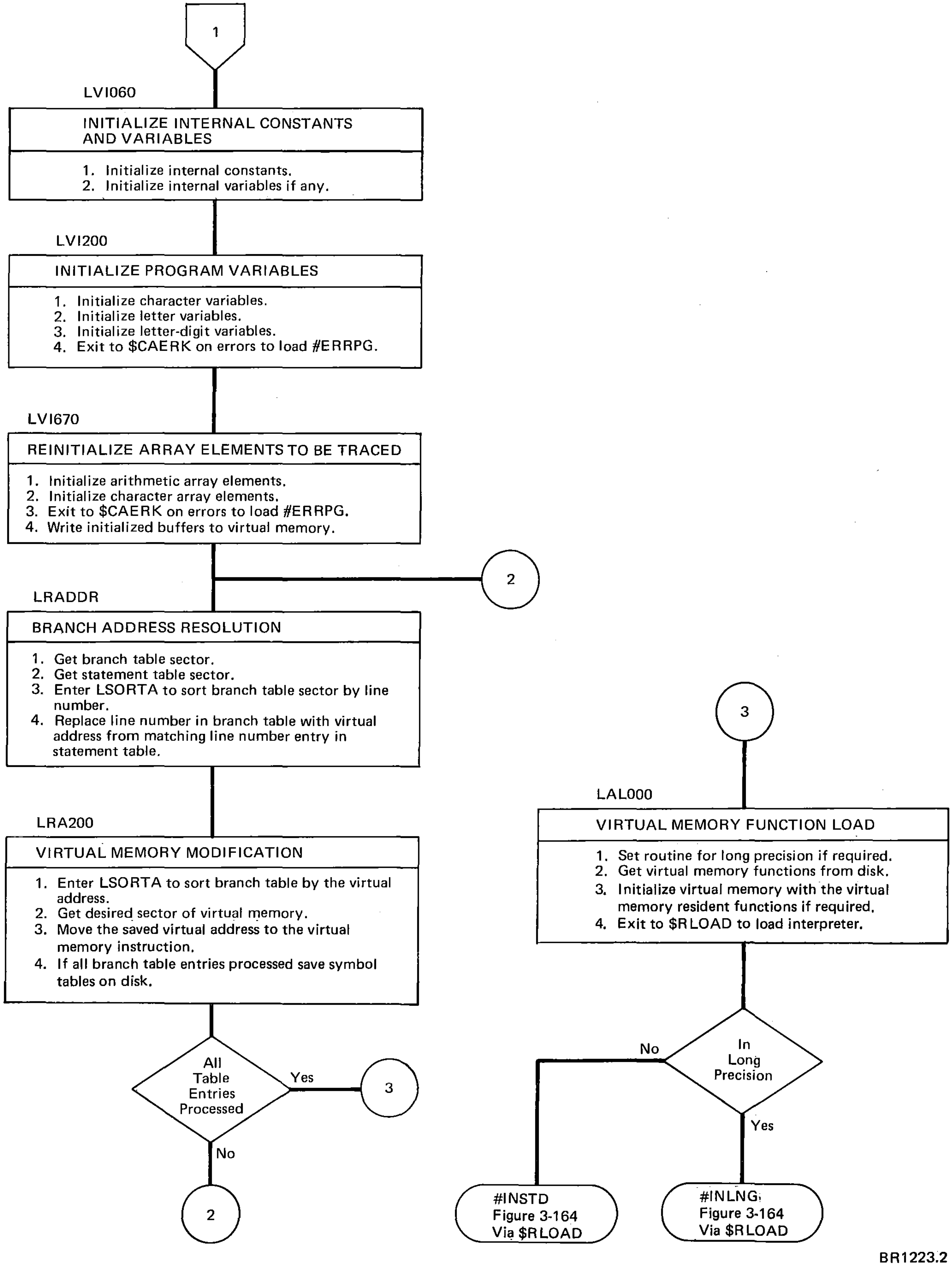


Figure 3-161. Loader (#LOADR) Flowchart (Part 2 of 2)

INTERPRETER (Figure 3-164)

Input to the interpreter is an object program composed of pseudo machine instructions. The interpreter executes these instructions, one at a time, to produce output for the user.

Interpreter Cycle

1. The pseudo instruction address register (PIAR) points to the op code of the next pseudo instruction to be executed. If the virtual-memory page that contains that pseudo instruction is not in core, it is read from virtual memory into the paging area.
2. The op code is used as a displacement into the PMC execution branch table. The core address located in the table is used to branch to a core-resident execution subroutine.
3. The core-resident subroutine may interface to an execution subroutine resident in virtual memory. The page containing the subroutine is read into the paging area if it is not already there.
4. The pseudo instruction is executed. Pages containing required data elements are read into the paging area if they are not already there.
5. The PIAR is incremented by the instruction length to point to the next sequential pseudo instruction; or, if branching, the branch virtual address is used to set the PIAR.
6. Steps 1 through 5 are performed until a terminating pseudo instruction is encountered and then a return is made to conversational mode.

Organization of Assembly Listings

All modules of the interpreter are contained in these four assemblies:

1. Standard precision core resident routines–#INSTD
2. Long precision core resident routines–#INLNG
3. Standard precision virtual memory resident execution subroutines–#FMSTD
4. Long precision virtual memory resident execution subroutines–#FMLNG

Interpreter Core Resident Routines–#INSTD, #INLNG

Two interpreter programs reside in the system program file. Either program is loaded into core for execution at X'0600', immediately following the system nucleus. These two programs are:

#INSTD–Standard precision interpreter
#INLNG–Long precision interpreter

The assembly of either #INSTD or #INLNG contains the same modules except for different interpreter execution equates. Each module assembles to the same byte length regardless of the precision, the differences being reflected in the execution characteristics of the coding. Each assembly contains the following modules arranged in the order listed:

@SYSEQ–General system equates
@FXDEQ–Fixed address equates
@CANEQ–Command analyzer equates
@ERMEQ–Error message equates
VEQU–Virtual address equates
$B@EQU–Compiler system equates
IEQU–Interpreter fixed equates
$I@SEQ–Interpreter system equates (The long-precision interpreter contains $I@LEQ instead of $I@SEQ. This is the only difference in the assembly listings.)

IMINIT–Initiator (overlayed with the run-time stack and work areas)
FDIADD/FDISUB–Floating-point add/subtract
FZIMPY–Floating-point multiply
FFIDVD–Floating-point divide
CPUFLT–Convert floating-point element to unpacked-decimal
CUPFLT–Convert floating-point element to packed-decimal
CAFPBS–Convert floating-point element to binary subscript
ISTACK–Element stacking subroutine
IUSTAK–Element unstacking subroutine
INTERP–Interpreter executive
ICFLTA–Arithmetic pseudo instruction execution
ICMATF–Matrix function pseudo instruction execution
ICELST–Element stacking pseudo instruction execution
ICARST–Array element stacking pseudo instruction execution
ICTEST–Logical pseudo instruction execution
ICBRAN–Branch pseudo instruction execution
ICLOOP–FOR/NXT pseudo instruction execution
ICVMEX–Interface to pseudo instruction execution subroutines in virtual memory
IPGMDL–Virtual-memory paging subroutine
IZCOMN–Interpreter common equates

Interpreter Virtual-Memory-Resident Execution Subroutines–#FMSTD and #FMLNG

Two interpreter components, containing virtual memory resident execution subroutines, reside in the system program file. Both components contain the same modules except for those marked with * on the symbolic label in Figure 3-162; coding varies in those modules due to precision differences. Each component assembles so that there is no difference between standard-precision and long-precision subroutine entry points.

Virtual Address	Disk Address	Symbolic Label	Pseudo Mnemonic	Synopsis
0200	0708	*FKSLGT	FN1	LGT intrinsic function (log base 10)
020B	0708	*FKSLTW	FN1	LTW intrinsic function (log base 2)
0219	0708	*FKSLOG	FN1	LOG intrinsic function (log base e)
0470	0710	CENXZD	*	Convert exponent to zoned decimal
04AD	0710	CCZDFP	*	Convert zoned decimal to floating point
0500	0714	*FGSEXP	FN1	EXP intrinsic function (exponential)
0800	0720	FNBPWR	PWR	Floating-point exponentiate
0900	0724	FRBSQR	FN1	SQR intrinsic function (square root)
0A00	0728	*FSSCOS	FN1	COS intrinsic function (cosine)
0A1A	0728	*FSSIN	FN1	SIN intrinsic function (sine)
0C70	0730	CBFPZD	*	Convert floating point to zoned decimal
0CB2	0730	CDBNZD	*	Convert binary number to zoned decimal
0D00	0734	*FWSCOT	FN1	COT intrinsic function (cotangent)
0D28	0734	*FWSTAN	FN1	TAN intrinsic function (tangent)
1100	0744	*FBSATN	FN1	ATN intrinsic function (arctangent)
1400	0750	*FCSACS	FN1	ACS intrinsic function (arcosine)
1413	0750	*FCSASN	FN1	ASN intrinsic function (arcsine)
1500	0754	*FHSHCS	FN1	HCS intrinsic function (hyperbolic cosine)

BR1224.1

Figure 3-162. Contents of Virtual Memory (Interpreter) (Part 1 of 3)

Virtual Address	Disk Address	Symbolic Label	Pseudo Mnemonic	Synopsis
1557	0754	*FHSHSN	FN1	HSN intrinsic function (hyperbolic sine)
1593	0754	*FHSHTN	FN1	HTN intrinsic function (hyperbolic tangent)
1700	075C	*FTSSEC	FN1	SEC intrinsic function (secant)
1725	075C	*FTSCSC	FN1	CSC intrinsic function (cosecant)
1761	075C	FABABS	FN1	ABS intrinsic function (absolute value)
176C	075C	FJBINT	FN1	INT intrinsic function (integer value)
17A7	075C	FUBSGN	FN1	SGN intrinsic function (sign of value)
17CB	075C	FPBRAD	FN1	RAD intrinsic function (degrees to radians)
17DA	075C	FPBDEG	FN1	DEG intrinsic function (radians to degrees)
1800	0780	*FQSRND	FN0 or FN1	RND intrinsic function (random-number generator)
1900	0784	IDDVST	*	Entry for all stack array dope vector pseudo instructions
191F	0784	IDDSD0	SD0	Stack array dope vector (no redimensioning)
192A	0784	IDDSD1	SD1	Stack array dope vector (redimension as a vector array)
1930	0784	IDDSD2	SD2	Stack array dope vector (redimension as a matrix array)
1A00	0788	IDFILE	*	Entry for all I/O pseudo instructions
1A40	0788	IDFGET	GET	Input data element
1A75	0788	IDFPUT	PUT	Output data element
1A87	0788	IDFINI	INI	Initiate data input
1A95	0788	IDFADF	ADF	Activate external data file
1AAB	0788	IDFPRS	PRS	Print and position carrier
1ABA	0788	IDFPRU	PRU	Print using image
1ACD	0788	IDFRSR	RSR	Restore internal data file pointer
1AD6	0788	IDFRST	RST	Reset external data file pointer
1ADF	0788	IDFCLS	CLS	Close external data file
1B00	078C	IDIFNC	FCI	User function call (indirect)
1C00	0790	SFADFR	ADF	Activate external data file
1D00	0794	SFPUTR	PUT	Output element to external data file
2100	07A4	SFGETR	GET	Input element from external data file
2400	07B0	SFRCAL	CLS	Close all external data files
2406	07B0	SFRCLS	CLS	Close a specified external data file
2409	07B0	SFRSET	RST	Reset external data pointer
2500	07B4	DFKEYN	GET	Keyboard physical IOCS (actual I/O)
25C0	07B4	DEPTBL	GET	Keyboard character table
2800	07C0	DFPRNT	PRS or PRU	System printer physical IOCS (actual I/O)
2A00	07C8	DFRDIN	GET	Card reader physical IOCS (actual I/O)
2A96	07C8	DFCOUT	PUT	Card punch physical IOCS (actual I/O)
2B00	07CC	FZXINP	GET	Keyboard input
2B00	07CC	FZXIP1	INI	Initiate data input from keyboard
2B66	07CC	FZXIP2	GET	Convert and move to virtual memory, one keyboard input data element

BR1224.2

Figure 3-162. Contents of Virtual Memory (Interpreter) (Part 2 of 3)

Virtual Address	Disk Address	Symbolic Label	Pseudo Mnemonic	Synopsis
3300	070D	FZREAD	GET	Read internal data file
3400	0711	FZSPRT	PRS	Print and carrier positioning
3800	0721	FZUPRT	PRU	Print using image
3D00	0735	FZDMIP	MF1	Data input to a matrix via the keyboard (MAT INPUT)
3E00	0739	FZAMIO	*	Matrix I/O routines
3E00	0739	FZAMRD	MF1	Read internal data file to a matrix (MAT READ)
3E06	0739	FZAMGT	MF1	Get data from external data file to a matrix (MAT GET)
3E0C	0739	FZAMPT	MF1	Put data from a matrix to an external data file (MAT PUT)
3F00	073D	FZCMPR	*	Matrix print routines
3F00	073D	FZCMPS	MF1	Print (packed) contents of a matrix (MAT PRINT)
3F06	073D	FZCMPL	MF1	Print (full) contents of a matrix (MAT PRINT)
3F13	073D	FZCMPU	MF1	Print (using image) contents of matrix (MAT PRINT USING)
4000	0741	FEBMSB	MF3	Matrix subtraction (MAT C=A–B)
4007	0741	FEBMAD	MF3	Matrix addition (MAT C=A+B)
4100	0745	FMBMPY	MF3	Matrix multiplication (MAT C=A*B)
4264	0749	FYBSMM	MSM	Matrix scalar multiplication (MAT C=(E1)*M)
4300	074D	FZBIDN	MF1	Matrix identity (MAT C=IDN)
4324	074D	FZBCON	MF1	Matrix unity (MAT C=CON)
432B	074D	FZBZER	MF1	Matrix zero (MAT C=ZER)
43A0	074D	FLBMAS	MF2	Matrix assignment (MAT A=B)
4400	0751	FXBTRN	MF2	Matrix transposition (MAT C=TRN (M))
4500	0755	FVBINV	MF2	Matrix inversion (MAT C=INV (M))
4540	0755	FVBDET	MF1	Matrix determinant (DET (C))
4600	0759	FZLINT	*	Trace line numbers subroutine
4700	075D	FZVART	*	Trace variables subroutine
4C00	0791	FZZVMP	*	Virtual-memory push/pull subroutine
4C00	0791	FZZVPS	*	Virtual-memory push
4C06	0791	FZZVPL	*	Virtual-memory pull
4D00	0795	DLFPRT	*	Line printer physical IOCR
5000	07A1	SFADF2	ADF	Activate external data file (part 2)
5100	07A5	SUBSTR	*	Substring routine

BR1224.3B

Figure 3-162. Contents of Virtual Memory (Interpreter) (Part 3 of 3)

Either component is copied to virtual memory by the loader (#LOADR) from the system program file. Individual pages are read into the paging area of core and executed under control of the virtual-memory paging subroutine (IPGMDL). Both components contain subroutines to perform the functions listed in Figure 3-162.

The following list contains explanations of the column entries in Figure 3-162:

1. "Virtual address" is the virtual entry point to perform the function.
2. "Disk address" is the physical disk address of the virtual-memory page containing the entry point.

3. "Symbolic label" is the symbolic name of the entry point in the assembly listings of either #FMSTD or #FMLNG. An * indicates that coding in the subroutine varies due to precision differences.
4. "Pseudo mnemonic" is the mnemonic of the pseudo instruction associated with the execution of that subroutine. An * indicates that multiple pseudo instructions are associated with the subroutine.

Interpreter Labeling Conventions

Because virtual-memory-resident routines must communicate with the core-resident interpreter, a fixed equate module (IEQU) is used to reference core-resident instructions and areas. In addition, equate module IZCOMN is used to assist in defining the fixed addresses in IEQU. Essentially, IZCOMN references the same core addresses as IEQU, except IZCOMN addresses are derived from the assembled code while IEQU addresses are manually adjusted constants.

Core-resident modules are coded to reference other core-resident modules using the following conventions:

- Module entry points–Actual entry point label.
- Module data/instruction fields–Equivalent IZCOMN label.

Virtual-memory modules are coded to reference core-resident modules using the following conventions:

- Module entry points–Equivalent IEQU label.
- Module data/instruction fields–Equivalent IEQU label.

Virtual-memory-resident module entry points are always referenced using the appropriate VEQU label.

Program descriptions use the following conventions, with respect to both core-resident and VM-resident modules, for consistency:

- Module entry points–Actual entry point label.
- Module data/instruction fields–Equivalent IEQU label.

For example:

• Actual core-resident entry point label–	INTERR
Referenced from core as–	INTERR
Referenced from virtual memory as–	I$XERR
• Actual core-resident data field label–	INTERC
Referenced from core as–	IZERRC
Referenced from virtual memory as–	I$ERRC
• Actual VM-resident entry point label–	FZREAD
Referenced from core as–	V$XSRD
Referenced from virtual memory as–	V$XSRD

Interpreter Initiator–IMINIT

IMINIT modifies the core-resident interpreter for an expanded core configuration, initializes the core virtual-memory page region, and sets run-time indicators prior to entering the interpreter executive (INTERP).

Expanded Core Utilization

When the system core configuration exceeds 8k and core beyond 8k is available for increased operational efficiency, IMINIT performs appropriate adjustments to the paging subroutine (IPGMDL) such that all usable core space is dedicated to expanding the core page region. The 8k system (Figure 3-163) operates on 10 core pages. When extra core is available, one of these page areas is used to expand tables in IPGMDL. The remaining nine-page region, combined with the additional core, is used to contain virtual-memory pages. The size of the core paging area is:

Core Size	*Pages*
8k	10
12k (with CRT)	18
12k (no CRT)	25
16k (with CRT)	34
16k (no CRT)	41

After core allocation, the core page region is loaded from virtual memory with consecutive pages beginning with page number 00. The page reference table in IPGMDL is initialized to define this condition.

Interpreter Executive—INTERP

The primary function of INTERP is to translate a pseudo instruction op code into the entry point of a core-resident PMC processing routine and then branch to that routine. INTERP also contains certain housekeeping routines and work areas that are central to interpreter operations and PMC routines to process the following pseudo machine instructions:

STH—Statement header
IMH—Image statement header
HLT—Halt execution
EOP—End of page
SVC—Supervisor call

Entry points to INTERP are:

1. INTERP—Begin execution. The first virtual-memory PMC page is locked into core. The first pseudo instruction in that page is accessed and control is passed to the appropriate PMC processing routine.
2. INTPAG—Transfer control. The virtual-memory PMC page specified in I$VADR is locked into core. The pseudo instruction referenced by I$VADR is accessed and control is passed to the appropriate PMC processing routine.
3. INTAD1—The pseudo instruction address register (I$XIAR) is incremented by one byte. The next instruction is accessed.
4. INTAD2—I$XIAR is incremented by two bytes and the next instruction is accessed.
5. INTAD3—I$XIAR is incremented by three bytes and the next instruction is accessed.
6. INTAD4—I$XIAR is incremented by four bytes and the next instruction is accessed.
7. INTXEC—The pseudo instruction referenced by I$XIAR is accessed and control is passed to the appropriate PMC processing routine.
8. INTADS—The run-time stack pointer (I$STAK) is incremented by the value in parameter I$STKI. An error condition occurs when I$STAK is incremented beyond the stack data limit.
9. INTERR—The error code in I$ERRC is stored as a parameter to the error program (#ERRPG), all active external data files are closed, all modified pages in core are written back to virtual memory, and control is passed to the error program, via $CAERK in the system nucleus.

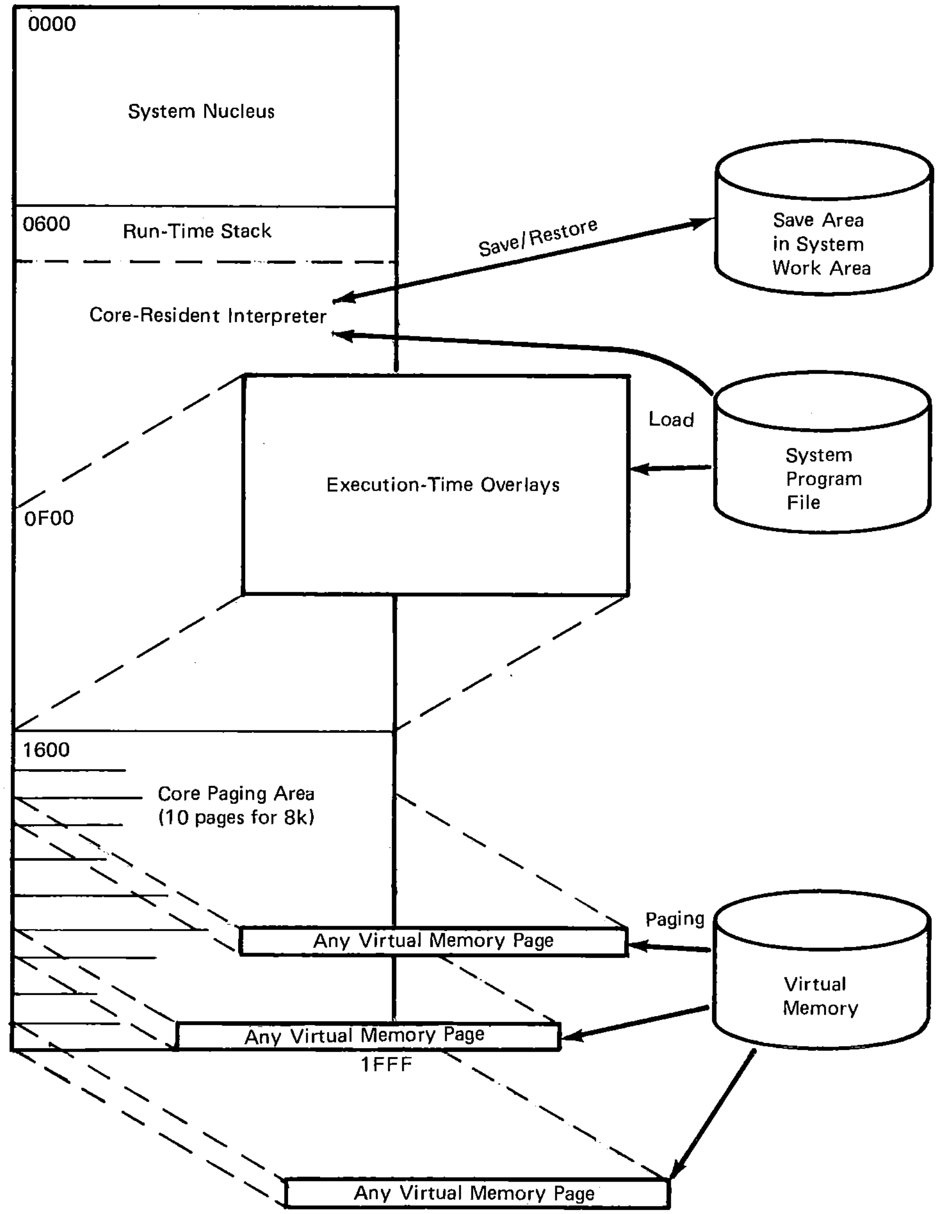


Figure 3-163. Interpreter Core Map (8k System)

Input parameters to INTERP are:

1. I$XPAG (entry INTPAG)–One byte, for the execution page number. This contains the virtual page number of the PMC page to which control is to be transferred.
2. I$VADR (entry INTPAG)–Two bytes, for the paging routine virtual address parameter. This contains the virtual address of the pseudo instruction to which control is to be transferred.
3. I$XIAR (entries INTAD1, INTAD2, INTAD3, INTAD4)–Two bytes, for the pseudo instruction address register. This contains the core address of the op code byte of the pseudo instruction.
4. I$XIAR (entry INTXEC)–Contains the core address of the op code byte in the pseudo instruction to be executed.

5. I$STKI (entry INTADS)—One byte, for the run-time stack pointer increment. This contains the value of the increment to be added to I$STAK.
6. I$ERRC (entry INTERR)—One byte, for the interpreter error code. This contains the code associated with the error message to be displayed by the system error program on exit to $CAERK.

Output parameters from INTERP are:

1. I$XIAR (entry INTPAG)—Contains the core address of the op code byte in the pseudo instruction to which control is transferred.
2. I$XIAR (entries INTAD1, INTAD2, INTAD3, INTAD4)—Contains the core address of the op code byte of the next pseudo instruction to be executed.
3. I$STAK (entry INTADS)—Two bytes, for the run-time stack pointer. This has been incremented by the value in parameter I$STKI.
4. $CAERR (INTERR execution)—One byte, for the system error program parameter. This is set equal to the value in I$ERRC.
5. $INLNO (STH execution)—Two bytes, for the system execution line number. This is set to contain the binary line number operand in the STH instruction.
6. I$STHA (STH execution)—Two bytes, for the statement header virtual address. This is set to contain the virtual address of the op code in the currently executed STH instruction.
7. I$IRSW (IMH execution)—One byte, for the image reference switch. This switch is set off (code @NOP) during IMH instruction execution.
8. I$IRSW (STH/IMH execution)—One byte, for the image reference switch. This switch, normally set to code @NOP, is set to code @UCB when the statement header to be executed must be an IMH rather than an STH.
9. I$RESW (STH execution)—One byte, for the recursion error switch. This is set to code @NOP when line number recursion is permitted during STH execution; unless specifically set prior to each STH instruction execution, I$RESW contains code @UCB which causes an error condition when line number recursion occurs.
10. I$TFSW (STH execution)—One byte, for the trace flow switch. This is set to code @NOP when TRACE FLOW is specified, and causes line number display when $TRACE in $XIND1 is also on. When TRACE mode processing has not been specified, I$TFSW is set to code @UCB.
11. $INLNO (STH execution)—Two bytes, for the system execution line number. This contains the binary line number of the statement just executed, or the value X'FFFF' when the first STH instruction is to be executed.
12. $XIND2 (INTERR, SVC execution)—One byte, for system execution indicator 2. Bit $EXCMD is set off, indicating termination of execution mode.

INTERP contains the following interpreter common work areas. Where applicable, the external label is given along with the internal area name:

1. INTDT1 (I$DAT1)—Two bytes, for the internal data file base pointer.
2. INTPAR (I$PARM)—Two bytes, for the interpreter common parameter field.
3. INTWK1 (I$WRK1)—Two bytes, for interpreter common work area 1.
4. INTWK2 (I$WRK2)—Two bytes, for interpreter common work area 2.
5. INTDAT (I$DATA)—Two bytes, for the internal data file pointer.
6. INTPIN (PRINT USING communication area)—Twelve bytes, for interpage information transfer during PRINT USING operations.
7. INTFAT (user function activity table)—Used as a push-down stack to control the execution of nested user functions. The first table entry is set equal to X'0000' to indicate the bottom of the stack. Each two-byte entry in the table contains the virtual address of an active user function.

8. INTBAT (PMC execution branch table)—Used to translate pseudo instruction op codes to PMC execution routine core address entry points. Each two-byte entry contains the core address entry point of a PMC execution routine defined by the relative position of the entry in the table. The op code value is used as an index to this table. This table contains entries for all pseudo instructions except DCA, DDL, DWA, and EOF.

Paging Subroutine—IPGMDL

The paging module interfaces between core routines (including virtual memory pages presently in core) and virtual memory. It provides the capability of addressing virtual memory directly and provides subroutine communication within VM. Several options give user control over the replacement process.

The paging module has eight entry points which are described as follows:

1. I$CVAD or IPGCVA—Convert address. Keeps all counters, usage value, and other page information up to date as well as reading and writing VM pages when necessary. The basic external function is to provide the caller with a core address (at label IPGCAD) when called with a virtual address (at label IPGVAD). When return is made, the page containing the byte referenced by IPGVAD is in core and the byte address is the value at location IPGCAD.
2. I$MDFY or IPGMOD—Page modify. Performs all the functions of IPGCVA as well as setting the read-only bit for the referenced page. This bit indicates that the page must be written back to virtual memory when modifications have been made to it. If the read-only bit has not been set for a page at replacement time, the paging subroutine assumes that the core page is still an exact copy of the disk virtual-memory page and a write operation is not performed.
3. I$LOCK or IPGLOK—Page lock. Performs all the functions of IPGCVA as well as setting the page locked bit for the referenced page. This function is used so that future references to the page can be made using core addresses. The page unconditionally remains at the same core location until the lock bit is reset.
4. I$UNLK or IPGULK—Page unlock. Performs all the functions of IPGCVA as well as resetting the page locked bit. This means that the page is subject to being replaced by future paging operations.
5. I$LDBR or IPGLBR—Convert address and load @BR. Performs all the functions of IPGCVA as well as setting @BR to point to the first byte of the page in core. @BR may then be used as the referenced page base register as well as allowing the calling page to reference any byte of the page by using the proper page displacement.
6. I$LDXR or IPGLXR—Convert address and load @XR. Performs all the functions of IPGCVA as well as setting @XR to the converted core address. @XR may then be used to directly reference the byte referred to by the virtual address.
7. I$CALL or IPGCAL—Call pageable subroutines. Performs all the functions of IPGLBR as well as locking the referenced page in core and stacking the return address and base register of the calling page for future return. A branch is made to the specified address.
8. I$RTRN or IPGRTN—Return from pageable subroutine. Unlocks the returning page, and then unstacks the next available return address and base register (previously stacked by IPGCAL) and returns to the original calling program.

There are two major work areas in the paging subroutine. One area is centrally located so that location will be within the base register range. The other area consists of tables and follows the paging subroutine code. The core page area follows the tables beginning with the next even 256-byte core address. The paging module is arranged in core so that the 8k version tables end immediately before the first core page (X'15FF').

The central work area contains:

1. IPGVAD—Virtual address storage location (three bytes). The first byte is always 00, the second byte is the virtual page number, and the third byte is the page displacement.
2. IPGCAD—Core address storage location (two bytes). The first byte equals the core page number (IPGCPG), and the second byte equals the page displacement.
3. IPGUVL—Reference counter for setting page usage value (two bytes).

The tables at the end of the paging subroutine code are:

1. IPGUVT—Usage values table; two bytes per entry, indexed from the low end by (IPGCPG)*2.
2. IPGLRT—Lock, read-only bit table; one byte per entry, indexed from the low end by PGNO. Only two bits of each entry are used.
3. IPGTBL—Page table; one byte per entry, indexed from the low end by IPGVPG. If a page is in core, its entry is equal to IPGCPG. If a page is not in core, its entry equals 00.
4. IPGSTK—Page call stack (four bytes per entry). This stack is used in IPGCAL/IPGRTN functions to save @BR and return addresses.

Element Stacking Subroutine—ISTACK

ISTACK moves a variable-length data field from virtual memory to the core location (normally within the run-time stack) referenced by index register @XR. The field is referenced in virtual memory using paging parameter I$VADR, and may extend across a single virtual page boundary. Field length is specified in a one-byte parameter to the subroutine, and remains available after subroutine execution. Register @XR is not modified during execution, but the virtual address in I$VADR is subject to modification when a page boundary condition exists.

Input parameters to ISTACK are:

1. Register @XR—For the destination core location pointer. This contains the core address of the leftmost byte of the core area into which the data element is to be moved.
2. I$VADR—Two bytes, for the paging routine virtual address parameter. This contains the virtual address of the leftmost byte of the data element that is to be moved.
3. I$SLNG—One byte, for the data element length code. This contains a value that is one less than the actual length of the data element. Unless specifically set prior to subroutine execution, I$SLNG automatically contains the length code required to move a packed floating-point decimal value (five bytes for standard precision, nine bytes for long precision).

Element Unstacking Subroutine—IUSTAK

IUSTAK moves a variable-length data field from the core location (normally within the run-time stack) referenced by index register @XR to virtual memory. The destination field is referenced in virtual memory using paging parameter I$VADR, and may extend across a single core page boundary. Field length is specified in a one-byte parameter to the subroutine. Register @XR is returned to the calling program intact, but the virtual address in I$VADR is subject to modification when a page boundary condition exists.

Depending on a subroutine parameter setting, the source data type may be compared with the data type contained in the destination field (arithmetic or character); inconsistent data types cause execution to be aborted on an error condition.

Also, depending on the current execution mode of the system, the new value of an element whose destination field is tagged for tracing is displayed on the system output device.

Input parameters to IUSTAK are:

1. Register @XR—For the source core location pointer. This contains the core address of the leftmost byte of the core area from which the data element is to be moved.
2. I$VADR—Two bytes, for the paging routine virtual address parameter. This contains the virtual address of the leftmost byte of the destination field in virtual memory.
3. I$ULNG—One byte, for the data element length code. This contains a value that is one less than the actual length of the data element. Unless specifically set prior to subroutine execution, I$ULNG automatically contains the length code required to move a packed floating-point decimal value (five bytes for standard precision, nine bytes for long precision).
4. I$DMSW—One byte, for the unstacking routine data matching switch. This contains code @NOP when matching is to be performed, or code @UCB when matching is not required.
5. $XIND1—One byte, for system execution indicator-1. This indicator contains a bit (mask $TRACE) which is set on when TRACE mode execution has been specified.

Output parameters from IUSTAK are:

1. Unstacked data element—(I$ULNG+1) bytes, located with leftmost byte stored in virtual memory at the address originally specified in I$VADR.
2. Traced variable—When TRACE mode has been specified and the destination field has been tagged for variable trace, the unstacked value is displayed, in association with the BASIC identifier corresponding to the destination field, on the system output device.
3. I$ERRC—One byte, for the error condition code. This contains a null code (I@NERR) when no error condition exists, or an error code specifying the particular error condition discovered.

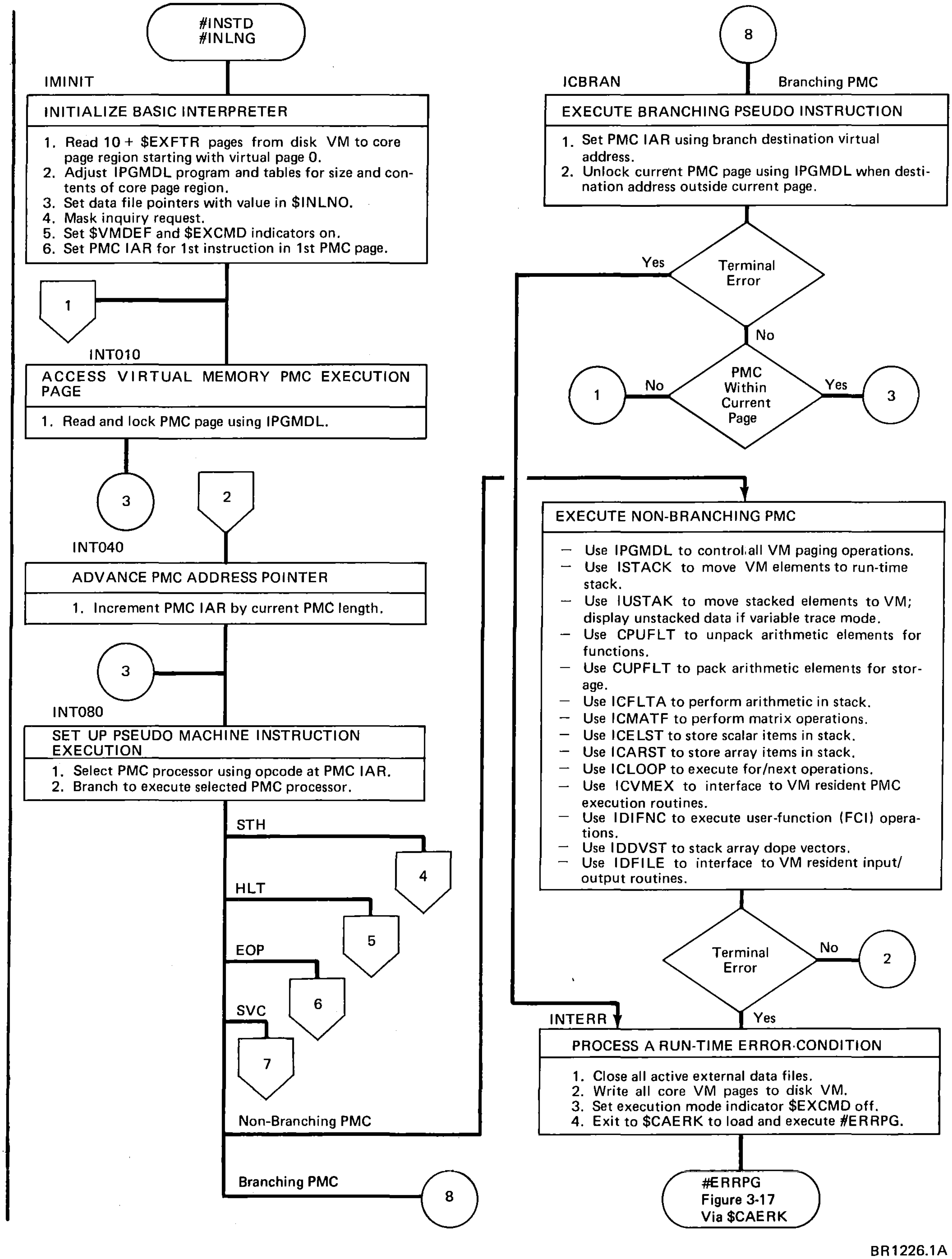


Figure 3-164. Interpreter (#INSTD, #INLNG) Flowchart (Part 1 of 2)

4

INTSTH

PROCESS STATEMENT HEADER PSEUDO INSTRUCTION

1. Execute INTERR if invalid statement line number recursion occurs.
2. Call $UNMSK to honor pending inquiry request; remask inquiry request on return.
3. Call $PAUSD to execute #EXMSG if step mode.
4. Execute INTSVC if GO ABORT specified.
5. Store new statement line number in $INLNO.
6. Display $INLNO if trace flow mode.

2

5

INTHLT

PROCESS HALT PSEUDO INSTRUCTION

1. Call DLFPRT if there is something to print and the system is in line-printer mode.
2. Call $PAUSD to execute #EXMSG for BASIC program PAUSE statement.
3. Disable inquiry request/step mode processing during next STH execution only

2

6

INTEOP

PROCESS END-OF-PAGE PSEUDO INSTRUCTION

1. Unlock current PMC page using IPGMDL.
2. Set PMC IAR for 1st instruction in next PMC page.

1

7

INTSVC

PROCESS SUPERVISOR CALL PSEUDO INSTRUCTION

1. Call DLFPRT if there is something to print and the system is in line-printer mode.
2. Close all active external data files.
3. Write all core VM pages to disk VM.
4. Set execution mode indicator $EXCMD off.
5. Exit to $CARPL to load and execute #GUFUD.

#GUFUD
Figure 3-22
Via $CARPL

BR1226.2A

Figure 3-164. Interpreter (#INSTD, #INLNG) Flowchart (Part 2 of 2)

I/O Execution Subroutines

Keyboard Physical IOCS—DFKEYN

DFKEYN reads from the keyboard to an input buffer and displays the input on the system printer, via a call to DFPRNT, through the paging subroutine (IPGMDL). All actual I/O to the keyboard during user program execution is executed by this subroutine. All function and command keys, except the enter-plus and program start keys, are processed. The call to this I/O subroutine includes the passing in @XR of the input buffer address.

System Printer Physical IOCS—DFPRNT

DFPRNT (Figure 3-165) prints on the matrix printer and performs carrier positioning operations. All actual I/O to the matrix printer during user program execution is executed by this subroutine. Waits for I/O to complete are executed by DFPRNT after the SIO, prior to returning to the calling routine (no I/O overlap is possible). The call to this I/O subroutine includes the address of the printer parameter list (Figure 5-23) in @XR. This subroutine assumes that the print parameter list is valid.

Line Printer Physical IOCR—DLFPRT (Figure 3-165)

DLFPRT prints bidirectionally on the line printer and performs carrier positioning operations. All actual I/O to the line printer during user program execution is executed by this subroutine together with DFPRNT. Waits for I/O to complete are executed after the SIO command and prior to printing another line or returning to the calling routine (no I/O overlap). The address of the printer parameter list (Figure 5-23) is passed to this I/O subroutine in @XR. This subroutine assumes that the print parameter list is valid.

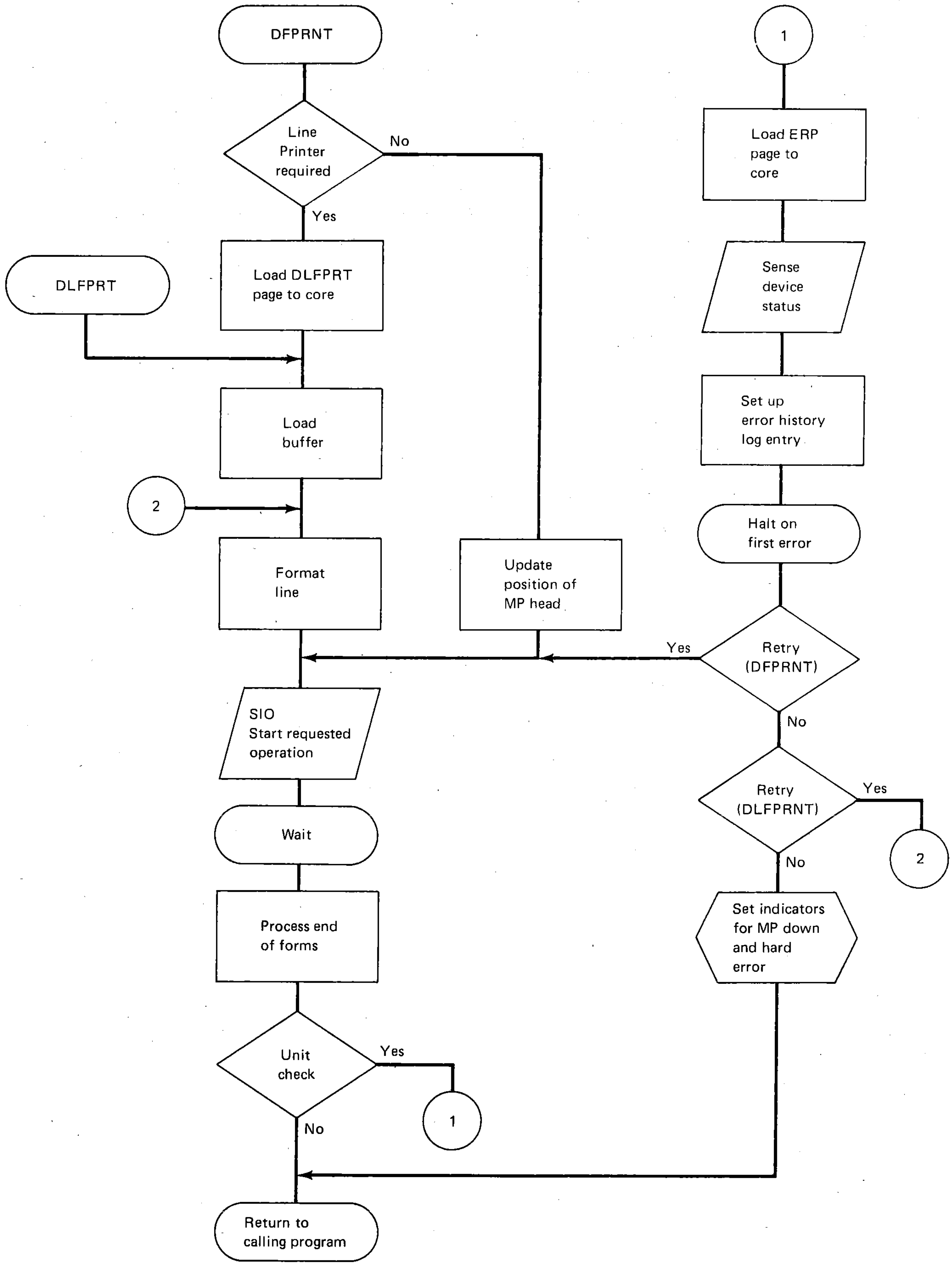


Figure 3-165. System Printer Physical IOCS (DFPRNT) and Line Printer Physical IOCS (DLFPRT) Flowchart

Card Reader Physical IOCS—DFRDIN (Figure 3-166)

DFRDIN fills the input buffer (located at the address in register @XR) with blanks, and then reads the card image (80 bytes for the 129 Card Data Recorder and 96 bytes for the 5496 Data Recorder) into the buffer with no I/O overlap and no truncation. All actual I/O for input from the data recorder, during user program execution, is executed by this subroutine. The call to this I/O subroutine includes the passing in @XR of the input data buffer address. Error procedures in DFRDIN are the same as those in #DREAD.

Card Punch Physical IOCS—DFCOUT (Figure 3-166)

DFCOUT punches 96 bytes of data from a buffer located at the address in @XR, with no I/O overlap. All actual I/O for output to the data recorder, during user program execution, is executed by this subroutine. The call to this I/O subroutine includes the passing in @XR of the output data buffer address.

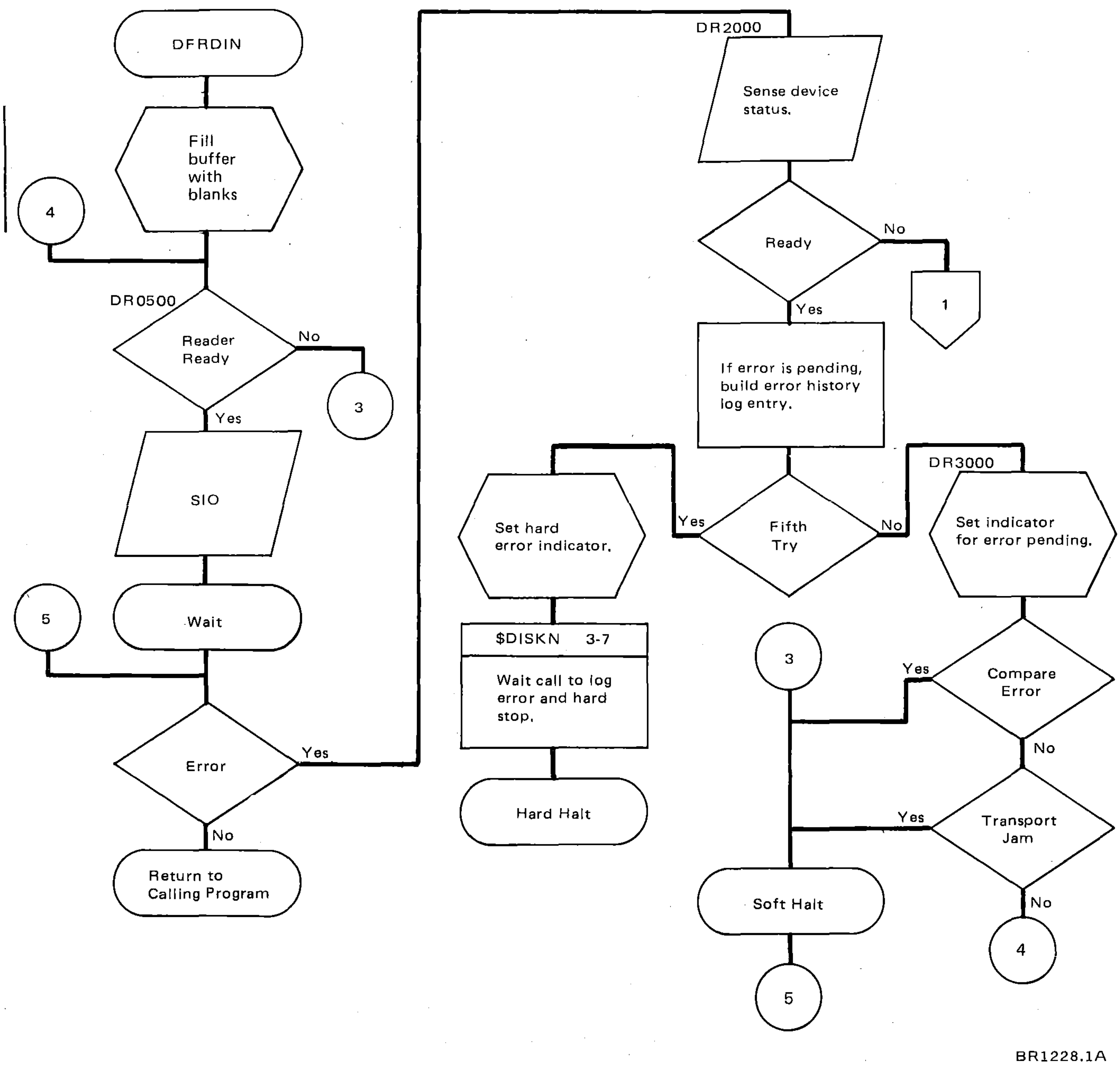


Figure 3-166. Card Reader Punch Physical IOCS (DFRDIN, DFCOUT) Flowchart (Part 1 of 2)

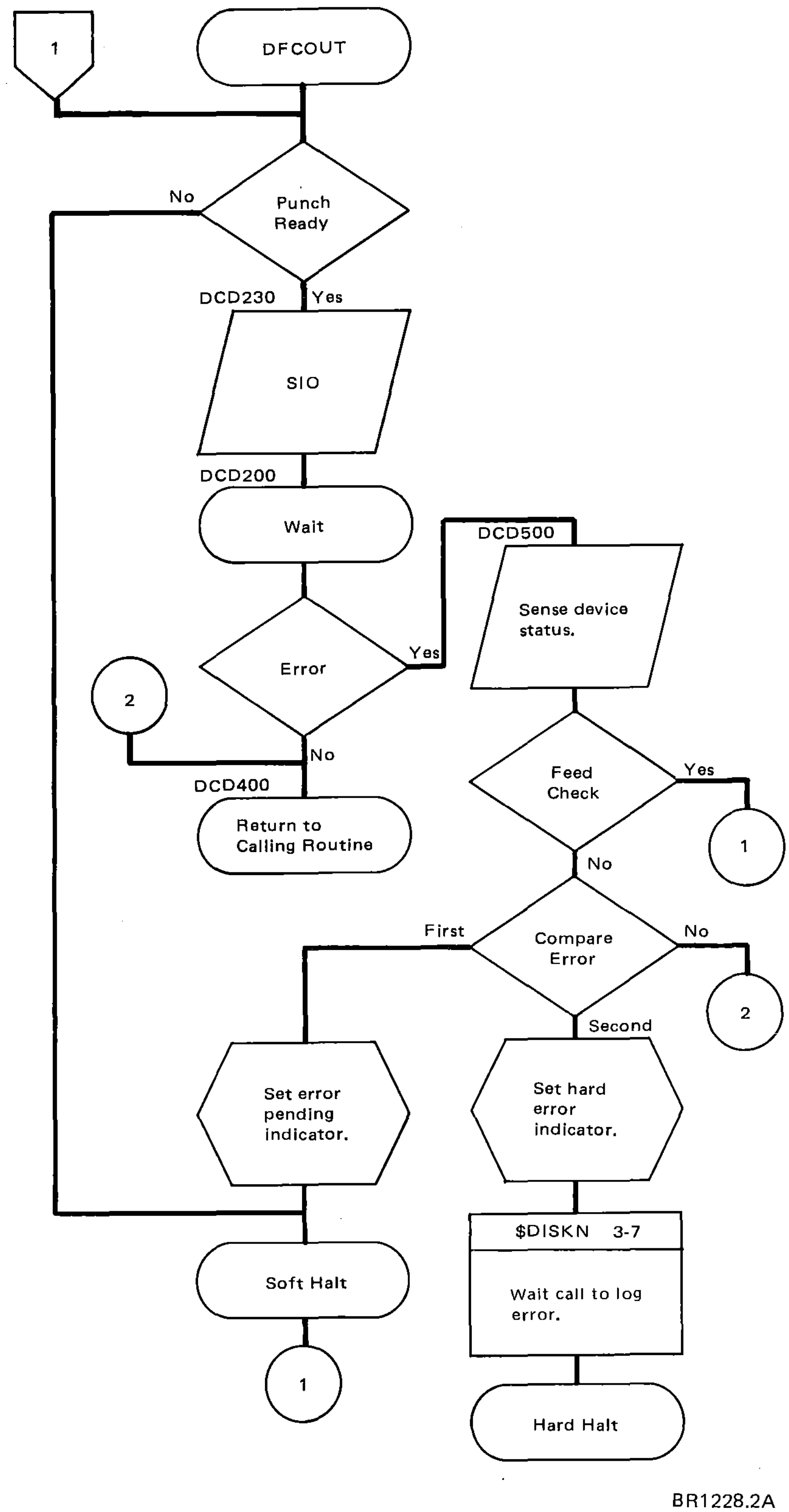


Figure 3-166. Card Reader Punch Physical IOCS (DFRDIN, DFCOUT) Flowchart (Part 2 of 2)

Activate External Data File—SFADFR

SFADFR is an execution-time file checker. Prior to any logical or physical I/O operation on an allocated file, SFADFR is called to open the file or to verify that the file is already open. Depending upon the status of the referenced file, SFADFR performs one of two functions. If the file is already open, the displacement within directory 2 (page X'01'), to the referenced entry, is stored in the directory 2 header for later use by SFGETR, SFPUTR or SFRSET. If the file had not been previously accessed, it must now be opened. The directory 1 (page X'00') record of allocated information is accessed and the file is found if it is a disk file. The directory 2 entry is initialized and the entry displacement is stored in the header.

Output an Element to an External Data File—SFPUTR

SFPUTR outputs a single arithmetic or character element to a sequential data file. This data file may be to disk, card, printer, or CRT. The specific action taken by this subroutine depends upon the device type:

1. For a disk file, the data element is placed in a buffer that is allocated for the disk file. When the buffer is full, the overlay program #SFLOA transfers the full buffer to the external data file in the file library. Following each transfer, an end-of-file record is generated and written to the data file, following the data. This EOF record is written over by the next transfer of data.
2. For a card output file, the data element is converted and placed in the buffer that is allocated for the card file. When the buffer is full, DFCOUT is called to punch the contents of the buffer.
3. For output to the printer or CRT, the data element passed is converted to external notation and DFPRNT is called to output the data element.

Input an Element from an External Data File—SFGETR

SFGETR is called to input the next sequential data element from an external data file. This data file may be on either disk or card. The next sequential data element, arithmetic or character, is accessed and placed in the run-time stack area. If input is from the card reader, the data element must be converted to internal notation before it is passed. When all data elements in the buffers allocated to the file are depleted, a call is made to the appropriate routine to refill the buffers in virtual memory. Refer to "Label Trace for GET Pseudo Instruction."

Close or Reset External Data Files—SFRSET

For disk or card output files, SFRSET outputs the last data elements (current contents of the buffers). For either input or output files, the current usage is set undefined (close only), the current buffer pointer is set to zero, and the displacement to the next sector of data within the file library (disk files only) is set to zero.

Keyboard Input—FZXINP

FZXINP execution causes keyboard data entry to be enabled during program operation. Entered data is syntax checked with respect to form and type, and valid elements are converted to internal format and placed in the run-time stack on an individual basis.

FZXINP performs the primary function of supporting the execution of INPUT statements. On a secondary level, the message printing, syntax checking, and data conversion facilities required for INPUT mode are also used for card file input operations. The first entry point (FZXIP1) operates in conjunction with stacked data type codes and a count parameter in I$PARM to allow keyboard data input and data line validity checking. The second entry point (FZXIP2) operates on the validity-checked data line to convert and stack sequentially occurring data elements.

Six alternate entry points are provided for use with MAT INPUT and GET (card) operations:

- Entry points FZXPQ1, FZXPQ2, and FZXPEM print question mark(s) or error messages on the system print device(s).
- Entry point FZXGCS syntax checks an entire GET (card) input line (into which comma delimiters have been inserted where they did not originally exist).
- Entry point FZXMIS validity checks a partial or entire array row.
- Entry point FZXCNV converts and stacks individual input line elements after the line has been syntax or validity checked.

Print and Carrier Positioning—FZSPRT

FZSPRT execution causes data output and/or carrier positioning on the system printer under control of codes developed from the format specified in a PRINT statement. FZSPRT performs the following functions depending on the code stored in interpreter parameter I$PARM:

1. Code X'01'—Print and no space. The data element at the top of the run-time stack is converted to output format and printed; if the element is arithmetic, the carrier is returned to the start of the next line, before printing, when the current line cannot contain the formatted value. The carrier is left positioned at the end of the printed value.
2. Code X'02'—Print and space full zone. The data element at the top of the run-time stack is converted to output format and printed; if the element is arithmetic, the carrier is returned to the start of the next line, before printing, when the current line cannot contain the formatted value; if the element is a character reference, the carrier is returned to the start of the next line, before printing, when the current line does not contain a full print zone (18 spaces). At the end of printing, the carrier is spaced to the end of the full print zone.
3. Code X'03'—Print and space packed zone. The data element at the top of the run-time stack is converted to output format and printed; if the element is arithmetic, the carrier is returned to the start of the next line, before printing, when the current line cannot contain the formatted value. After an arithmetic element is printed, the carrier is spaced to the end of the packed print zone; after a character element is printed, the carrier is left positioned at the end of the printed element.
4. Code X'04'—Print and return carrier. The data element at the top of the run-time stack is converted to output format and printed; if the element is arithmetic, the carrier is returned to the start of the next line, before printing, when the current line cannot contain the formatted value. After the element is printed, the carrier is returned to the start of the next line.
5. Code X'05'—Space full zone. The carrier is spaced 18 characters. If no more than 18 characters remain in the current line, the carrier is returned to the start of the next line.
6. Code X'06'—Space packed zone. The carrier is spaced three characters. If no more than three characters remain in the current line, the carrier is returned to the start of the next line.
7. Code X'07'—Return carrier. The carrier is returned to the start of the next line.
8. Code X'08'—Return carrier on condition. When the current line does not contain more than 18 characters, the carrier is returned to the start of the next line.

When required, element conversion and output are performed in the run-time stack, so that the stacked value is not recoverable after printing. Arithmetic element output format depends on the magnitude and fractional characteristics of the value; character reference formatting involves truncation of trailing blanks; character constants are printed as specified in the PRINT statement.

Either the matrix printer or the CRT (or both) may be used for output, depending on the current definition of the system printer. CRT output is based on a fixed display width of 64 characters, while printer line width is based on that assigned through the WIDTH system command.

Print Using Image—FZUPRT

FZUPRT execution causes a print image to be established in virtual-memory buffers and data elements to be output on the system printer under format control of image conversion specifications. FZUPRT performs the following functions depending on the code stored in interpreter parameter I$PARM:

1. Code X'00'—Release image. Virtual-memory pages containing the currently established image are unlocked for replacement during normal paging operations.
2. Code X'01'—Null image specification. This code causes a null image indicator to be set for future PRINT USING operations; no image buffers are established.
3. Code X'02'—Null print list specification. This code causes the currently established image to be printed, up to the character preceding the first conversion specification or end of image, and the carrier returned to the start of the next line; a null image results in a simple carrier return.
4. Code X'03'—Null character constant. This code causes the next available conversion specification in the image work buffer to be filled with blanks.
5. Code X'04'—First image segment. This code causes the character constant segment at the top of the run-time stack to be established as the first image segment in the image save buffer.
6. Code X'05'—Secondary image segment. This code causes the character constant segment at the top of the run-time stack to be added to the existing image segments in the image save buffer.
7. Code X'06'—Primary data element. A primary data element is defined as a floating point value, a character element, or the first segment of a multisegment character constant. This code causes the primary data element at the top of the run-time stack to be converted and placed in the image work buffer according to the next available conversion specification.
8. Code X'07'—Secondary data element. A secondary data element is defined as any segment (except the first) of a multisegment character constant. This code causes the secondary data element at the top of the run-time stack to be converted and placed in the image work buffer according to the currently referenced conversion specification (i.e., added to the current contents of the conversion specification).

Operations involving the "next available" conversion specification imply the following actions:

1. When no unfilled conversion specification remains in the image work buffer, the filled image is printed and the carrier is returned to the start of the next line.
2. When an image is to be printed, the carrier is returned to the start of the next line (before printing occurs) when not already positioned at the start of the current line.
3. Following step 1, all conversion specifications in the image become available, with the "next available" specification being the first contained in the image.

In conjunction with the codes, these indicators may be set in I$PARM:

1. Mask X'10'—Terminate print using. This indicator causes the image to be printed, up to the character preceding the next conversion specification or end of image, following the activity specified by the control code itself. All image buffers are released from core VM.

2. Mask X'20'—Matrix end of row. This indicator causes the image to be printed, up to the character preceding the next conversion specification or end of image, following the activity specified by the control code itself. Image buffers remain locked in core VM, and step 3 in the previous paragraph becomes effective.

Either the matrix printer or the CRT (or both) may be used for output, depending on the current definition of the system printer.

Keyboard Input to a Matrix—FZDMIP

FZDMIP contains the run-time routine which executes matrix operations for an array referenced in a MAT INPUT statement. FZDMIP performs INPUT operations for each element of the matrix referenced by the arithmetic array dope vector at the top of the run-time stack. Elements are entered on a row-by-row basis, each data line consisting of an entire partial array row. Partial array rows are terminated with a comma preceding the keyboard carriage return; the end of a row is signified with a carriage return without a preceding comma.

A single question mark is printed to request entry of the first array row. Thereafter, two question marks are printed to request data line entry until the array is completely assigned. Input errors in any single line cause a request (??) for the reentry of the entire row associated with that line (after an appropriate error message has been printed). Input is automatically terminated when each array element has been assigned a value.

Inquiry request may be invoked whenever the keyboard has been enabled for input. This results in reexecution of the STH pseudo instruction associated with the current MAT INPUT statement.

Matrix I/O Routines—FZAMIO

FZAMIO contains the run-time routines which execute matrix operations for an array referenced in a MAT READ, MAT GET, or MAT PUT statement. FZAMIO performs operations for each element of the matrix referenced by the arithmetic array dope vector at the top of the run-time stack:

1. READ—Successive elements from the program DATA file are assigned, beginning at the DATA file element currently referenced by I$DATA, to elements in the referenced matrix on a row-by-row basis; I$DATA is left referencing the first unused DATA element.
2. GET—Successive elements from the currently active external input file are assigned, beginning at the element currently referenced by the file pointer, to elements in the referenced matrix on a row-by-row basis; the file pointer is left referencing the first unused file element.
3. PUT—Elements from the referenced matrix are assigned, on a row-by-row basis, to successive element positions in the currently active external output file beginning at the element position currently referenced by the file pointer. The file pointer is left referencing the first unused file element position.

Matrix Print Routines—FZCMPR

FZCMPR contains the run-time routines which execute matrix operations for an array referenced in a MAT PRINT or MAT PRINT USING statement. FZCMPR performs PRINT (full zone format), PRINT (packed zone format), or PRINT USING operations for each element of the matrix referenced by the arithmetic dope vector at the top of the run-time stack:

1. PRINT (full zone format)—Successive elements from the referenced matrix are printed, on a row-by-row basis, on the system print device; each element is printed as specified for full zone output. (Refer to "Print and Carrier Positioning—FZSPRT.")

2. PRINT (packed zone format)–Successive elements from the referenced matrix are printed, on a row-by-row basis, on the system print device; each element is printed as specified for packed zone output. (Refer to "Print and Carrier Positioning–FZSPRT.")
3. PRINT USING–Successive elements from the referenced matrix are printed, on a row-by-row basis, on the system print device; each element utilizes the "next available" conversion specification in the currently active image. (Refer to "Print Using Image–FZUPRT.") The printer carrier is positioned, prior to output of the first array element, such that two blank lines exist between the first matrix row and the previous printed line. Each matrix row is separated from the previous row with a blank line, and the carrier is returned following output of the final matrix row.

Miscellaneous Execution Subroutines

Trace Line Numbers Subroutine–FZLINT

FZLINT is called during the execution of every STH and IMH pseudo instruction when execution is in trace line number mode. The binary line number at label $INLNO is converted to a four-digit decimal integer and displayed on the system printer (matrix printer and/or CRT).

Trace Variables Subroutine–FZVART

FZVART is called when execution is in trace variables mode and the trace bit is on in a referenced arithmetic element or character field. Using the virtual address located at label I$PARM, the compiler symbol tables are searched to locate the variable name (symbol) assigned to the element or field.

The variable name along with the current value or contents of the element or field is displayed on the system printer (matrix printer and/or CRT). If the element or field is within an array, the subscripts of the element or field are also displayed.

The subscripts of the element or field are developed by this subroutine by incrementing the array's base virtual address by the element or field length until it is equal to the virtual address of the element or field (I$PARM).

The compiler symbol tables are searched in this order:

1. Arithmetic variable (letter) symbol table (LVT)
2. Character variable symbol table (CVT)
3. Arithmetic variable (letter-digit) symbol table (LDT)
4. Character array symbol table (CAT)
5. Arithmetic array symbol table (NAT)

Virtual Memory Push/Pull Subroutine–FZZVMP

- FZZVMP has two entry points: FZZVPS and FZZVPL.

Entry FZZVPS: This entry causes all modified virtual memory pages in core to be written back to disk. All pages in core referenced with a modify switch in the lock and read only indicator table (located in the paging subroutine, IPGMDL) are written back to their respective locations in virtual memory.

Entry FZZVPL: This entry causes all unlocked virtual memory pages in core to be re-read from disk virtual memory. All pages in core referenced with a lock switch in the lock and read only indicator table (located in the paging subroutine, IPGMDL) are read into core at their respective locations.

Both procedures are automatically adjusted to process an expanded table and core paging area for 12k or 16k systems.

Interpreter Execution Overlay Programs

Matrix Inversion/Determinant—#FISTD and #FILNG

Two interpreter execution overlays reside in the system program file. Either one overlays the core-resident interpreter at X'0E00' to perform matrix inversion or determinant functions during execution of the BASIC program. #FISTD performs these functions in standard precision and #FILNG performs them in long precision. These overlays are called by the virtual-memory-resident execution subroutine, FVBINV/FVBDET (VM page X'45').

Random Number Generator—FQSRND and FQLRND (Figure 3-166.1)

The random number generator is a routine contained in #FMSTD (#FMLNG). It is paged into any available page in main storage above the interpreter.

The following algorithm is used to generate the primary sequence of numbers:

$U_0=(U_2 + U_3)\mathrm{mod}P$

where U_0, U_2, and U_3 are the numbers being calculated now, 2 times ago, and 3 times ago, respectively, and P is the prime number.

A subsequence is then obtained by taking every fourth element of the primary sequence. This subsequence provides the mantissa of the numerator in the expression:

R=U/P

where R is the output random number. The period for standard precision is approximately 10^{15} and the period for long precision is approximately 10^{29}.

The initial values for the variables in Figure 3-166.1 are:

Standard precision	*Long precision*
P = 6684673	P = 820678790827111
X = 3926991	X = 109050773266576
Y = 1442695	Y = 797882384626433
Z = 8414709	Z = 832795028878064

Find Disk Data File—#SFFIN (Figure 3-167)

#SFFIN is a program called from the system program file, and overlays part of the core-resident interpreter (#INSTD or #INLNG). The calling routine must save the core-resident interpreter in the system work area prior to loading #SFFIN.

Using file directory 1, #SFFIN locates disk data files when they are first accessed at program execution time. For a permanent file, #SFFIN searches all disks on the system for the filename, password, and volume-ID specified. The status of the file is checked and the necessary information is placed in file directory 2. For a scratch file, the space specified in the ALLOCATE command is sought for in all the null directories on the system and necessary information is returned in file directory 2.

Before returning to the calling routine, #SFFIN starts I/O to begin the restore of the core-resident interpreter. Refer to the interpreter core map (Figure 3-163).

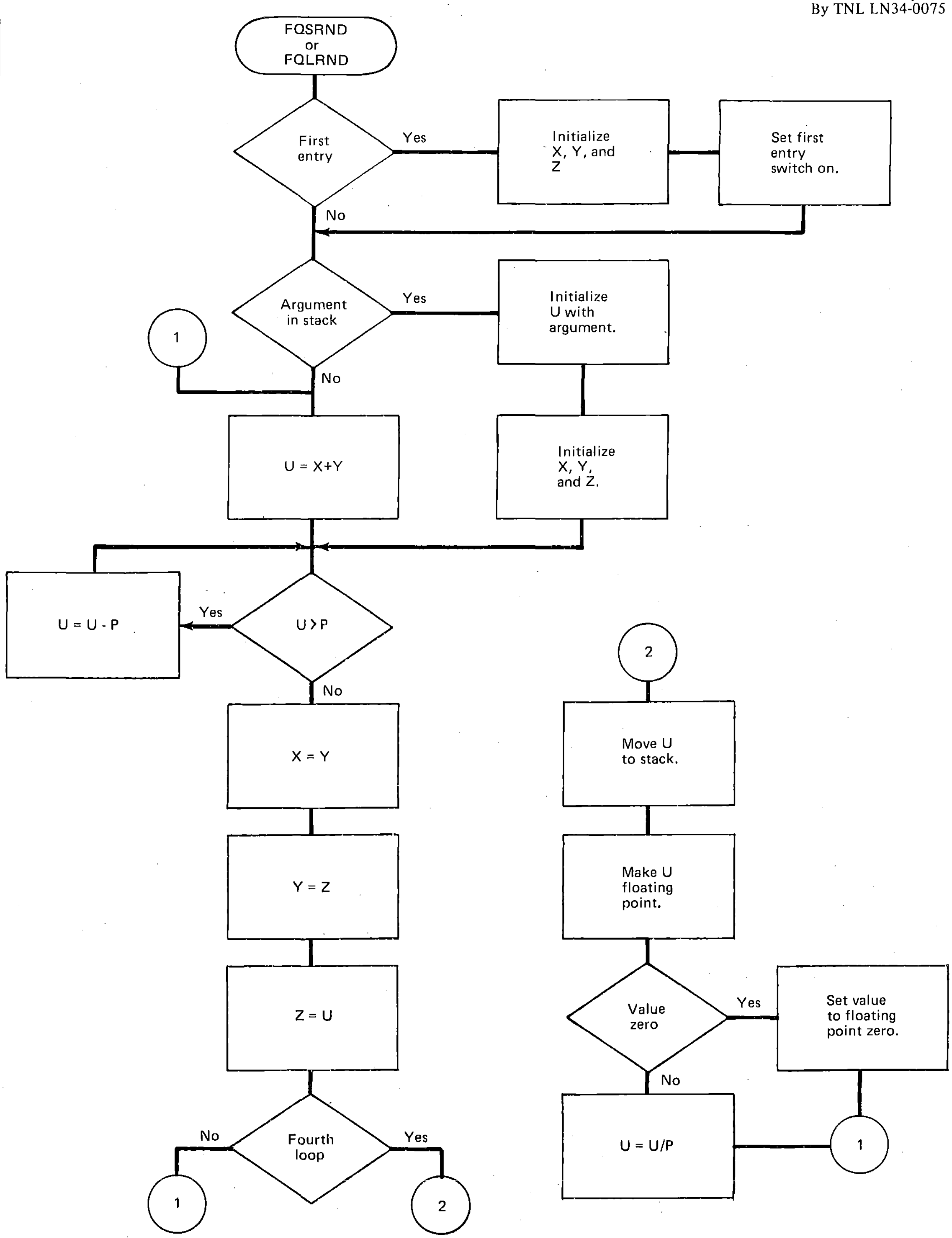


Figure 3-166.1. Random Number Generator (FQSRND, FQLRND) Flowchart

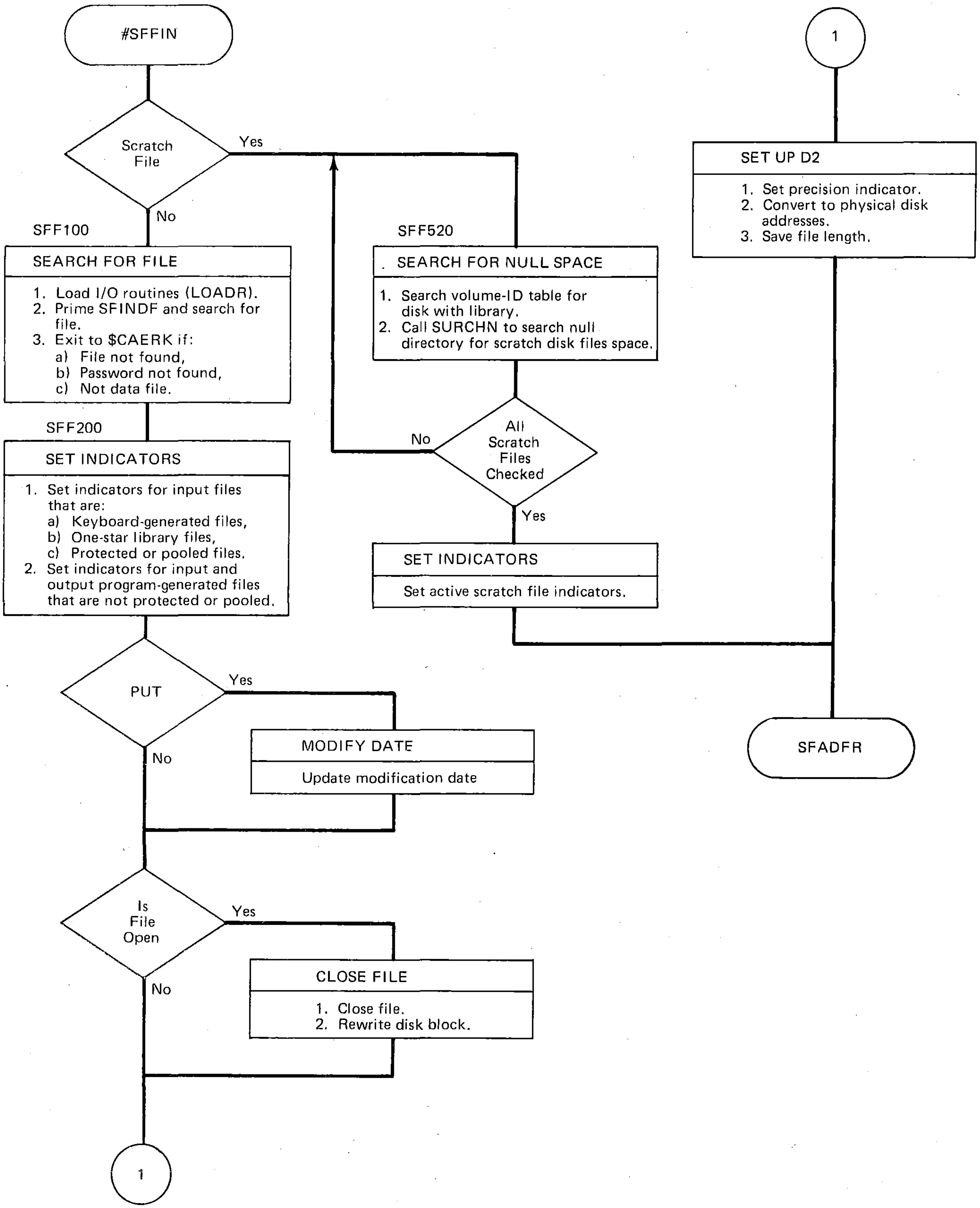


Figure 3-167. Find Disk Data File (#SFFIN) Flowchart

Logical IOCS for Disk Data Files—#SFLOA (Figure 3-168)

#SFLOA is a program called from the system program file to overlay the core-resident interpreter (#INSTD or #INLNG). The calling routine must save the core-resident interpreter in the system work area prior to loading #SFLOA.

#SFLOA executes multiple sector transfer operations between allocated buffers in virtual memory and the data file located in the file library. Output is transferred to the file library; input is transferred to virtual memory. Before returning to the calling routine, #SFLOA starts I/O to begin the restore of the core-resident interpreter. All actual disk I/O is performed by branching to $DISKN in the system nucleus. Refer to the interpreter core map (Figure 3-163).

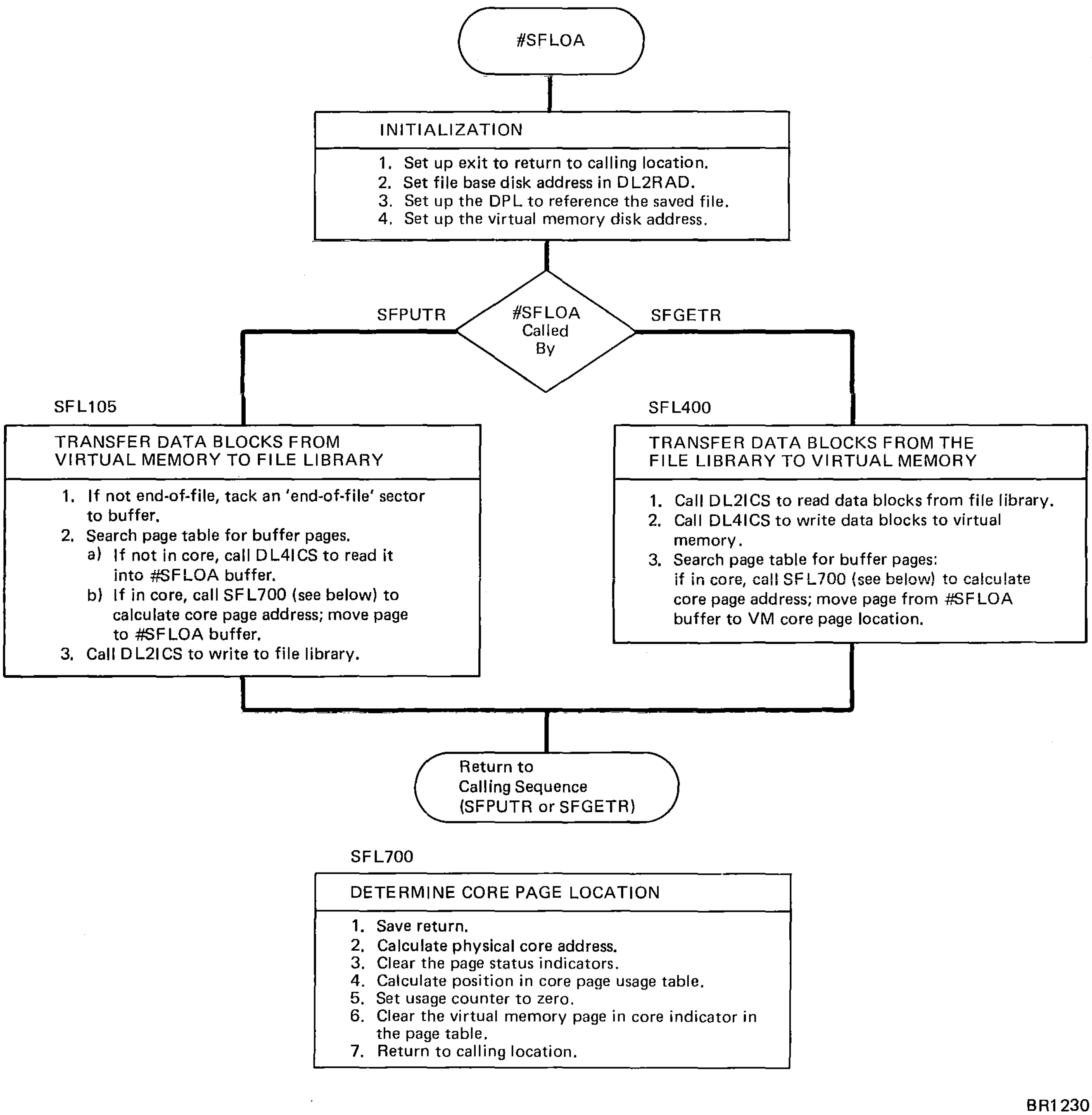


Figure 3-168. Logical IOCS for Disk Data Files (#SFLOA) Flowchart

Label Trace for ADD Pseudo Instruction

The following labels trace the execution of the ADD pseudo instruction. This trace illustrates an instruction executed entirely by core-resident routines.

1. INTXEC—The op code value for the ADD pseudo instruction (X'06') indexes the op code execution branch address table (INTBAT).
2. INT100—Pass control to a core-resident execution subroutine. In this case, the entry point is ICFADD.
3. ICFADD—Pass control to the core-resident floating point add subroutine (FDIADD).
4. FDIADD—Perform floating-point addition of the top two run-time stack elements. IZSTAK references the run-time stack.
5. ICF020—Pass control back to the interpreter executive at INTAD1.
6. INTAD1—Increment the pseudo instruction address register (INTIAR) by one byte (length of ADD instruction) in preparation for the next sequential pseudo instruction.
7. INTXEC—Access the next pseudo op code.

Label Trace for GET Pseudo Instruction

The following labels trace the execution of the GET pseudo instruction. Prior to this GET, an external data file was activated by an ADF pseudo instruction. This trace illustrates an instruction that requires paging of subroutines from virtual memory. This example also includes a save/overlay/restore of the core-resident interpreter. Labels marked with * are located in the core-resident interpreter. Unmarked labels are located in virtual-memory-resident execution subroutines.

1. *INTXEC—The op code value for the GET pseudo instruction (X'52') indexes the op code execution branch address table (INTBAT).
2. *INT100—Pass control to a core-resident execution subroutine. In this case, the entry point is ICVFIO.
3. *ICFVIO—Branch to the paging subroutine. The DC following the branch instruction is the virtual entry point in the required virtual memory page.
4. *IPGCAL—Read and lock page X'1A' into the paging area. This page contains the execution subroutine IDFILE.
5. IDFILE—Pass control to the routine at label IDFGET.
6. IDFGET—Branch to the paging subroutine. I$CALL is equated to IPGCAL. The virtual address operand of the GET pseudo instruction was stored in the DC following the branch to I$CALL. In this case, the virtual address operand equals X'2100'.
7. *IPGCAL—Read and lock page X'21' into the paging area. This page contains the execution subroutine SFGETR.
8. SFGETR—Check file usage and device type.
9. SFG290—Branch to the paging subroutine (I$CALL).
10. *IPGCAL—Read and lock page X'22' (second page of SFGETR).
11. SFGBS2—Assuming the input buffer is empty, branch to the paging subroutine (I$CALL).
12. *IPGCAL—Read and lock page X'23' (third page of SFGETR).
13. SFGBS3—Assume the input buffer must be filled.
14. SFG780—Save the core-resident interpreter on cylinder 9 of the system work area.
15. SFG790—Load #SFLOA at X'0F00', via $BLOAD in the system nucleus.
16. SFLOAD—This subroutine copies data, in blocks, from the user's external data file in the file library to the pages in virtual memory assigned as input buffers for this file. (See Figure 3-168.) The core-resident interpreter is restored to core.

17. SFG795–Wait for I/O complete on the interpreter restore operation; then determine the data file type.
18. SFG900–Branch to the paging subroutine. I$RTRN is equated to IPGRTN.
19. *IPGRTN–Unlock page X'23' and return to the calling page, X'22'.
20. SFG450–Move the data item to the run-time stack.
21. SFG695–Branch to the paging subroutine (I$RTRN).
22. *IPGRTN–Unlock page X'22' and return to the calling page, X'21'.
23. SFG295–Branch to the paging subroutine. I$UNLK is equated to IPGULK.
24. IPGULK–Unlock directory 2 (page number X'01') and return to the same page, X'21'.
25. SFG295+9–Branch to the paging subroutine (I$RTRN).
26. *IPGRTN–Unlock page X'21' and return to the calling page, X'1A'.
27. IDF120–Establish the virtual address destination and the data element type.
28. IDF140–Branch to the core-resident element unstacking subroutine. I$USTK is equated to IUSTAK.
29. *IUSTAK–Branch to the paging subroutine. The destination virtual address in the run-time stack is referenced by I$VADR. The data element, also in the run-time stack, is referenced by @XR.
30. *IPGMOD–Read the page referenced by I$VADR into the paging area. Read-only bit is set for the page.
31. *IUS012–Move the data element from the run-time stack to the referenced displacement (second byte of I$VADR).
32. *IUS150–Return to page X'1A'.
33. IDF150–Load @XR with the return address in the interpreter executive. I$XAD3 is equated to INTAD3.
34. IDF990–Branch to the paging subroutine (I$RTRN).
35. *IPGRTN–Unlock page X'1A' and return to the interpreter executive.
36. *INTAD3–Increment the pseudo instruction address register (INTIAR) by three bytes (length of the GET instruction) in preparation for the next sequential pseudo instruction.
37. *INTXEC–Access the next pseudo op code.

PSEUDO INSTRUCTION SET

Pseudo instructions make up the pseudo machine language and invoke the execution of preassembled machine language execution subroutines to perform the functions indicated by the pseudo instruction name. Figure 3-169 contains a table of the mnemonic operation codes, in alphabetic order, for all pseudo machine instructions. Figure 3-170 shows the pseudo instruction formats.

Detailed descriptions of the pseudo instructions follow these two figures, which should be used as references in following the descriptions. The instructions are described in order as follows:

1. Arithmetic operations
2. Function call operations
3. I/O operations
4. Logical operations
5. Stack and unstack operations
6. Miscellaneous operations
7. Nonexecutable operations

Mnemonic	Length (bytes)	Operand	Hexadecimal Op Code	Name
ADD	1	*	06	Add
ADF	2	XX	58	Activate external data file
BNX	3	VADR	4A	Branch and suppress execution
BRA	3	VADR	46	Branch unconditionally
BRC	4	VADR CC	44	Branch on condition
BRD	3	VADR	48	Branch and delete function entry
BRS	1	*	4C	Branch to stacked address
CLS	1	*	5E	Close external data file
CMC	1	*	42	Compare character elements
CMF	1	*	40	Compare floating point values
CSA	2	NN	3E	Compute stacked address
DCA	3	VADR	6A	Define constant address
DDL	3	VADR	6C	Define data linkage
DIV	1	*	0C	Divide
DWA	2	NN	6E	Define work area
EOF	1	*	70	End of program
EOP	1	*	68	End of page
FCI	3	VADR	16	Function call—indirect
FN0	3	VADR	12	Function call—no argument
FN1	3	VADR	14	Functional call—one argument
FOR	3	VADR	4E	Initiate FOR loop
GET	3	VADR	52	Input data element
HLT	1	*	04	Halt execution
IMH	3	LINE	66	Image statement header
INI	2	NN	56	Initiate keyboard input
MF1	3	VADR	18	Single matrix function call
MF2	3	VADR	1A	Double matrix function call
MF3	3	VADR	1C	Triple matrix function call
MPY	1	*	0A	Multiply
MSM	3	VADR	1E	Matrix scalar multiply
NEG	1	*	10	Negate
NXT	3	VADR	50	Perform next step
PRS	2	XX	60	Print and space carrier
PRU	2	XX	62	Print using image
PUT	2	XX	54	Output data element
PWR	1	*	0E	Exponentiate
RSR	1	*	5A	Restore internal data file pointer
RST	1	*	5C	Reset external data file pointer
SA1	3	VADR	36	Stack vector array element address
SA2	3	VADR	38	Stack matrix array element address
SB1	3	VADR	3A	Stack character array element address
SC1	3	VADR	2A	Stack character array field

BR1231.1A

Figure 3-169. Pseudo Instruction Reference List (Part 1 of 2)

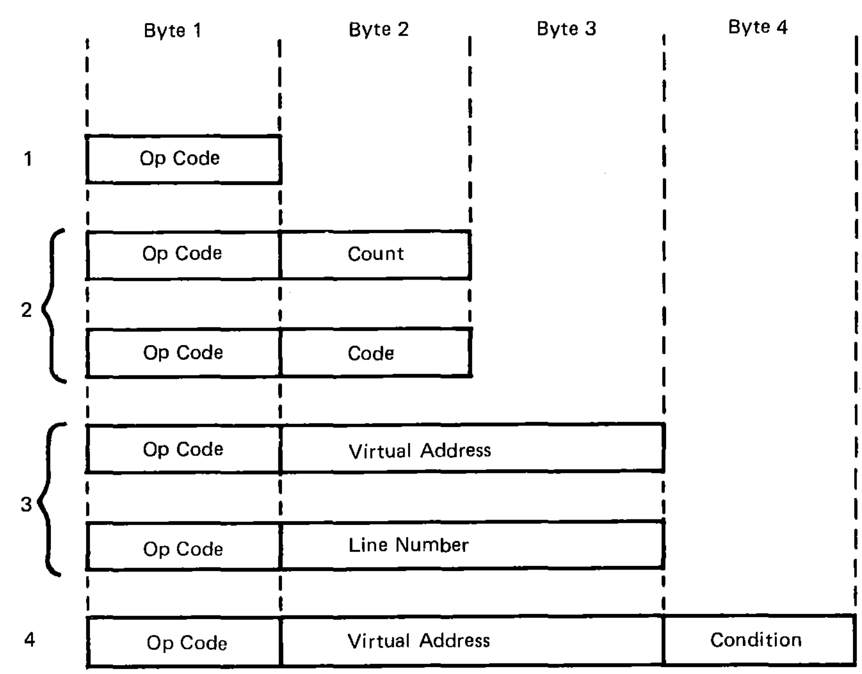
Mnemonic	Length (bytes)	Operand	Hexadecimal Op Code	Name
SD0	3	VADR	2E	Stack arithmetic array descriptor
SD1	3	VADR	30	Stack arithmetic array descriptor
SD2	3	VADR	32	Stack arithmetic array descriptor
SF1	3	VADR	22	Stack arithmetic vector element
SF2	3	VADR	24	Stack arithmetic matrix element
STA	3	VADR	34	Stack virtual address
STC	3	VADR	28	Stack character field
STF	3	VADR	20	Stack floating point value
STH	3	LINE	64	Statement header
STX	2	XX	3C	Stack execution control code
SUB	1	*	08	Subtract
SVC	1	*	02	Supervisor call
USC	2	NN	2C	Unstack character element
USF	1	*	26	Unstack floating point element

Notes:

*— no operands
VADR— 2-byte virtual address
XX— 1-byte execution control code
NN— 1-byte binary execution count
LINE— 2-byte binary statement line number
CC— 1-byte branch condition code.

BR1231.2A

Figure 3-169. Pseudo Instruction Reference List (Part 2 of 2)



BR1232

Figure 3-170. Pseudo Instruction Formats

Arithmetic Operations

Add

ADD

X'06'

0 7

BR1233

The floating-point value at the top stack location (the top of the run-time stack) is added to the floating-point value at the second stack location. Both values are deleted from the stack and the sum (in floating-point notation) is placed at the top stack location.

Example:

PMC Sequence

.

STF VADR of A

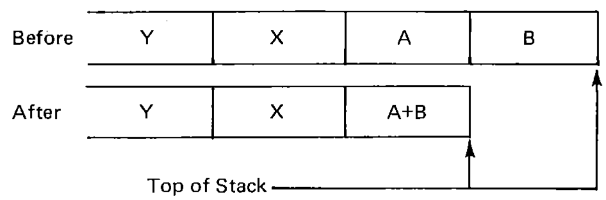
STF VADR of B

➤ADD

.

.

Run-Time Stack



BR1234

Subtract

SUB

X'08'

0 7

BR1235

The floating-point value at the top stack location is subtracted from the floating-point value at the second stack location. Both values are deleted from the stack and the difference (in floating-point notation) is placed at the top stack location.

Example:

PMC Sequence

.

STF VADR of A

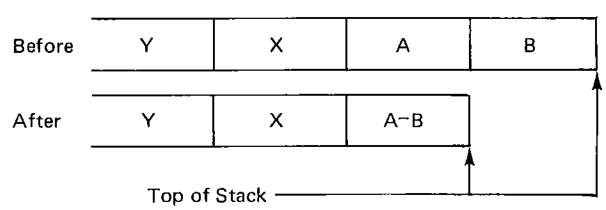
STF VADR of B

➤SUB

.

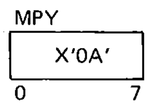
.

Run-Time Stack



BR1236

Multiply



BR1237

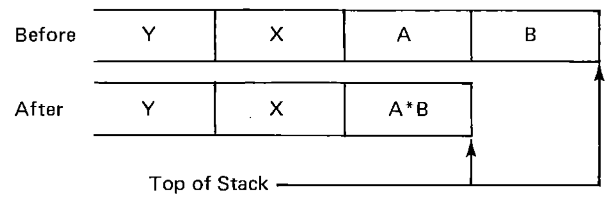
The floating-point value at the second stack location is multiplied by the floating-point value at the top stack location. Both values are deleted from the stack and the product (in floating-point notation) is placed at the top stack location.

Example:

PMC Sequence

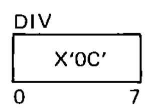
.
STF VADR of A
STF VADR of B
➤MPY
.
.

Run-Time Stack



BR1238

Divide



BR1239

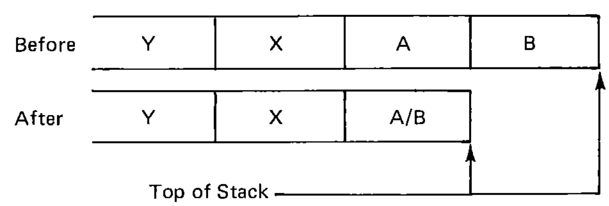
The floating-point value at the second stack location is divided by the floating-point value at the top stack location. Both values are deleted from the stack and the quotient (in floating-point notation) is placed at the top stack location.

Example:

PMC Sequence

.
STF VADR of A
STF VADR of B
➤DIV
.
.

Run-Time Stack



BR1240

Exponentiate (Power)

PWR

X'0E'

0 7

BR1241

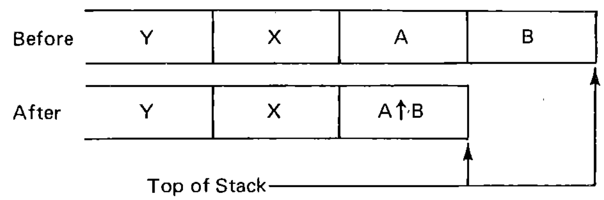
The floating-point value at the second stack location is raised to the power specified by the floating-point value at the top stack location. Both values are deleted from the stack and the result is placed at the top stack location.

Example:

PMC Sequence

.
STF VADR of A
STF VADR of B
➤PWR
.
.

Run-Time Stack



BR1242

Negate

NEG

X'10'

0 7

BR1243

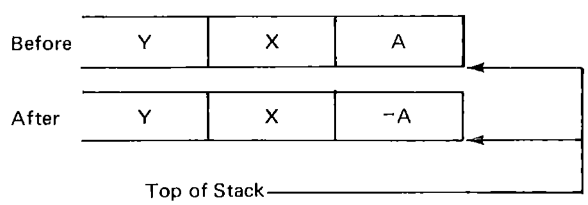
The floating-point value at the top stack location is negated. The original value at the top stack location is deleted and the negated value is placed at the top stack location.

Example:

PMC Sequence

.
STF VADR of A
➤NEG
.
.

Run-Time Stack



BR1244

Matrix Scalar Multiply

MSM

X'1E'	VADR
0 7	0 7

BR1245

The third stack location contains an arithmetic array descriptor that defines the matrix to contain the product elements. These elements are the result of multiplying the matrix defined by the arithmetic array descriptor at the first (top) stack location, by the floating-point value at the second stack location. VADR is the virtual entry point to a subroutine in virtual memory that performs the operation. The multiplier value and both array descriptors are deleted from the stack after the function is executed.

Example:

BASIC Statements	PMC Sequence
0100 MAT C = (A) * M	STH 0100
	SD0 VADR of Descriptor for C
	STF VADR of A
	SD0 VADR of Descriptor for M
	➤MSM VADR of Subroutine
0110 (statement)	STH 0110

Run-Time Stack

Before	Y	X	Desc (C)	A	Desc (M)
After	Y	X			

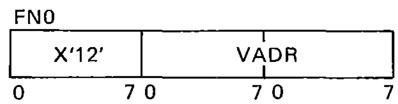
Top of Stack

Desc—Array Descriptor (array dope vector)

BR1246

Function Call Operations

Function Call—No Argument



BR1247

No argument is required for the execution of this pseudo instruction. VADR is the virtual entry point to a subroutine in virtual memory that performs the function. The floating-point value (R), resulting from execution of the function, is placed at the top stack location. Refer to the intrinsic function virtual address equates in the program listing "System Equates" (VEQU in #TEQU2). An example of a function performed by this instruction is the "no argument" form of intrinsic function RND.

Example:

PMC Sequence

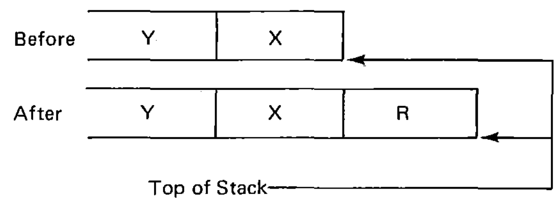
.

➤FN0 VADR of Subroutine

.

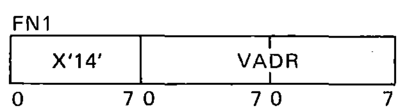
.

Run-Time Stack



BR1248

Function Call—One Argument



BR1249

The floating-point value at the top stack location is used as the argument for the function. VADR is the virtual entry point to a subroutine in virtual memory that performs the function. The floating-point value (R), resulting from execution of the function, replaces the argument (A) at the top stack location. Refer to the intrinsic function virtual address equates in the program listing "System Equates" (VEQU in #TEQU2). An example of a function performed by this instruction is computation of the tangent (TAN) of the argument, the argument being expressed in radians.

Example:

PMC Sequence

.

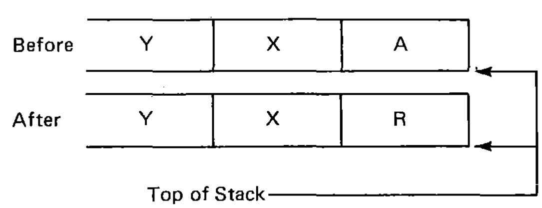
STF VADR of A

➤FN1 VADR of Subroutine

.

.

Run-Time Stack



BR1250

Function Call—Indirect

FCI

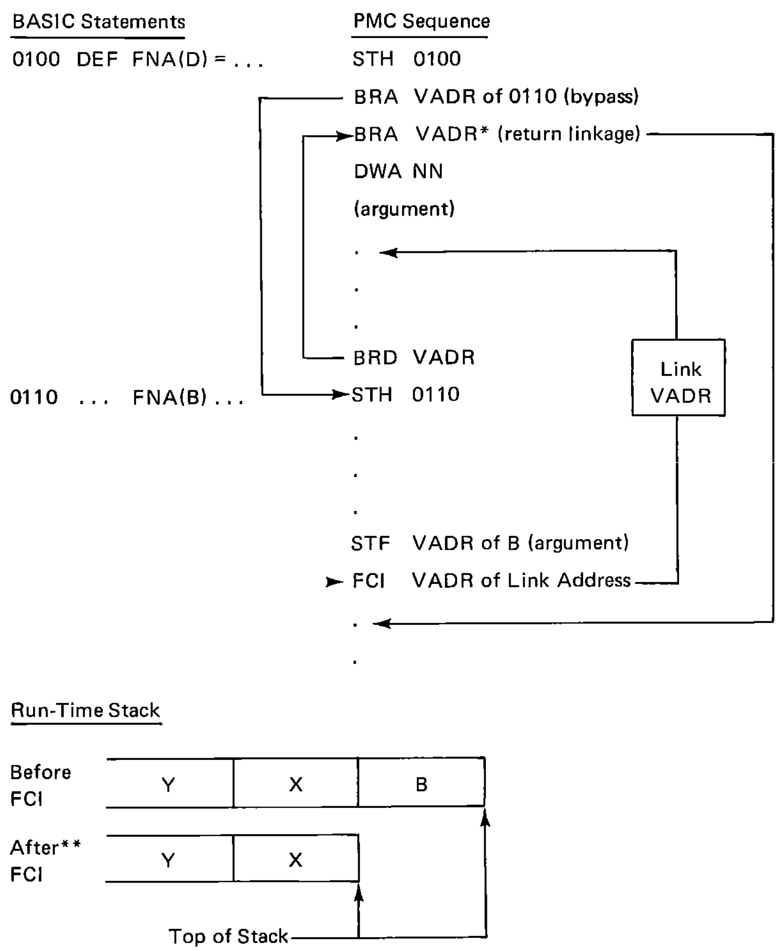
X'16'	VADR
0 7	0 7 0 7

BR1251

The floating-point value at the top stack location is used as the argument for the user function whose linking address is defined at VADR. The value at the top stack location is deleted, and control is transferred to the pseudo instruction which begins the user function execution. Linkage is established such that the function execution sequence returns control to the pseudo instruction following the FCI.

Prior to user function execution, the user function activity table is searched for an entry that matches VADR. When no match occurs, VADR is added to the table. When a match does occur, or when the table size is exceeded, a terminal error condition is indicated. A match in the table occurs when user function is referenced within the definition of that same function.

Example Using FCI, BRD, and DWA Pseudo Instructions:



* This VADR (return linkage) is established at execution time by the FCI function execution subroutine.

** "After" refers to the logical stack condition immediately after the FCI instruction execution, but before the execution of the DEF statement expression.

BR1252

Single Matrix Function Call

MF1

X'18'	VADR
0 7	0 7 0 7

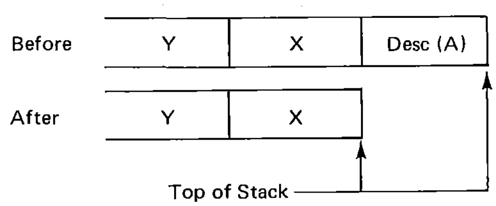
BR1253

The arithmetic array descriptor at the top stack location references the single matrix argument for the matrix function to be performed. VADR is the virtual entry point to a subroutine in virtual memory that performs the function. The array descriptor is deleted from the top stack location after the function is executed. Refer to matrix function virtual address equates in the program listing "System Equates" (VEQU in #TEQU2). An example of a function performed by this instruction is matrix I/O operations.

Example:

BASIC Statements	PMC Sequence
0100 MAT INPUT A	STH 0100
	SD0 VADR of Descriptor for A
	➤ MF1 VADR of Subroutine
0110 (statement)	STH 0110

Run-Time Stack

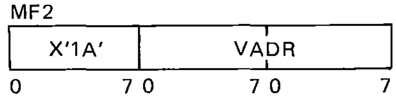


Desc—Array Descriptor (array dope vector)

BR1254

Double Matrix Function Call

MF2



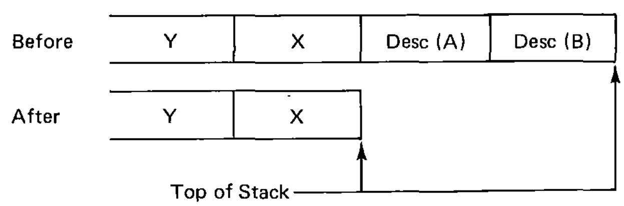
BR1255

The arithmetic array descriptors at the second and top stack locations reference the two matrix arguments for the matrix function to be performed. VADR is the virtual entry point to a subroutine in virtual memory that performs the function. Both array descriptors are deleted from the stack after the function is executed. Refer to matrix function virtual address equates in the program listing "System Equates" (VEQU in #TEQU2). An example of a function performed by this instruction is matrix assignment.

Example:

BASIC Statements	PMC Sequence
0100 MAT A = B	STH 0100
	SDO VADR of Descriptor for A
	SDO VADR of Descriptor for B
	► MF2 VADR of Subroutine
0110 (statement)	STH 0110

Run-Time Stack



Desc—Array Descriptor (array dope vector)

BR1256

Triple Matrix Function Call

MF3

X'1C'	VADR
0 7	0 7 0 7

BR1257

The arithmetic array descriptors at the third, second, and top stack locations reference the three matrix arguments for the matrix function to be performed. VADR is the virtual entry point to a subroutine in virtual memory that performs the function. All three array descriptors are deleted from the stack after the function is executed. Refer to matrix function virtual address equates in the program listing "System Equates" (VEQU in #TEQU2). An example of a function performed by this instruction is matrix subtraction.

Example:

BASIC Statements	PMC Sequence
0100 MAT C = A–B	STH 0100
	SDO VADR of Descriptor for C
	SDO VADR of Descriptor for A
	SDO VADR of Descriptor for B
	➤ MF3 VADR of Subroutine
0110 (statement)	STH 0110

Run-Time Stack

Before	Y	X	Desc (C)	Desc (A)	Desc (B)
After	Y	X			

Top of Stack

Desc—Array Descriptor (array dope vector)

BR1258

Input/Output Operations

Input Data Element

GET

X'52'	VADR
0 7	0 7 0 7

BR1259

The next sequential data element entered from a file of data elements is stored in virtual memory at the virtual address at the top stack location. VADR is the virtual entry point to a subroutine in virtual memory that performs the input operation. The referenced virtual address is deleted from the stack.

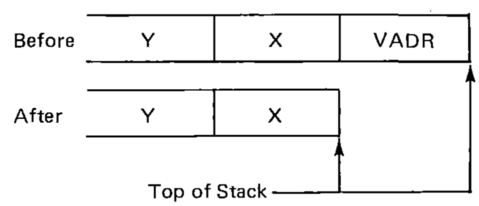
This pseudo instruction is generated for GET, READ, and INPUT BASIC program statements. If the GET is to reference an external data file, it must be preceded by an ADF (activate external data file) pseudo instruction. If the GET is to reference the internal data file or the system keyboard, it need not be preceded by an ADF instruction.

Example:

PMC Sequence

```
 .
 STA  VADR
►GET  VADR of Subroutine
 .
 .
```

Run-Time Stack



BR1260

Output Data Element

PUT

X'54'	XX
0 7	0 7

BR1261

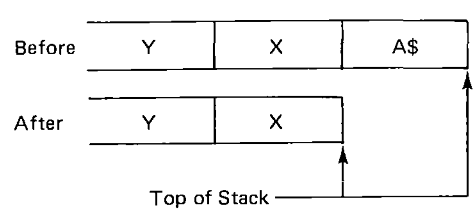
The data element or field at the top stack location is written in the next sequential location in the currently active data file. This external data file was activated by the last executed ADF pseudo instruction. XX defines the type of data (X'02' = arithmetic element; X'04' = character field). The data element or field is deleted from the top stack location.

Example:

PMC Sequence

```
 .
 ADF  (see ADF)
 STF  VADR of A$
►PUT  04
 .
 .
```

Run-Time Stack



BR1262

Initiate Keyboard Input

INI

X'56'	NN
0 7	0 7

BR1263

This pseudo instruction is generated for the INPUT BASIC statement to initiate an I/O operation for input from the keyboard. The execution control codes contained in stack locations 1 (top of the stack) through NN are parameters to the initiate input subroutine in virtual memory. They are used to verify the data type and number of elements entered by the user on the keyboard. Each of the referenced execution control codes is deleted from the stack.

The format of the execution control code, in the STX instructions preceding the INI, is:

Bit 0 = 0 for arithmetic elements; 1 for character fields.
Bits 1-7 = count of the consecutive elements of the same type.

Example:

BASIC Statement	PMC Sequence
0100 INPUT A, B, C, A$, B$, D	.
	.
	STX 03 (A, B, C)
	STX 82 (A$, B$)
	STX 01 (D)
	➤INI 03 (count of preceding STX instructions)

Run-Time Stack

Before	Y	X	03	82	01
After	Y	X			

Top of Stack

BR1264

Activate External Data File

ADF

X'58'	XX
0 7	0 7

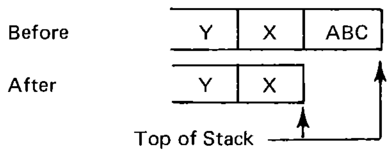
BR1265

The external data file referenced by the character literal in the top of the run-time stack is activated. The displacement to the file directory 2 entry for the referenced file is calculated, the file is tested for validity, and prepared for either input or output. XX equals X'01' when the referenced file is activated for output and equals X'00' when the referenced file is activated for input.

Example:

BASIC Statement	PMC Sequence
0100 GET 'ABC', X	.
	.
	STC VADDR of 'ABC'
	► ADF 00

Run-Time Stack



BR2674

Restore Internal Data File Pointer

RSR

X'5A'
0 7

BR1266

The internal data file pointer is restored to reference the first data element or field in the internal data file. Refer to "Define Constant Address" (DCA) and "Define Data Linkage" (DDL). These instructions define the data elements and/or fields in the internal data file. The next GET (to the internal data file) that is executed references the first data element or field in the internal data file.

The contents of the run-time stack are unaffected by the execution of this pseudo instruction.

Reset External Data File Pointer

RST

X'5C'
0 7

BR1267

The external data file pointer, for the currently activated external data file, is reset to reference the first data element or field in that file. This external data file was activated by the last executed ADF pseudo instruction. The next GET or PUT (to that external data file) that is executed references the first data element or field in that external data file.

The contents of the run-time stack are unaffected by the execution of this pseudo instruction.

Close External Data File

CLS

X'5E'
0 7

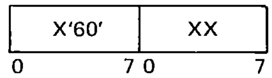
BR1268

The currently activated external data file is closed. The associated external data file pointer is reset to reference the first data element or field in that file. Refer to "Reset External Data File Pointer" (RST). Closing an external data file allows that file to be activated for input or for output.

The contents of the run-time stack are unaffected by the execution of this pseudo instruction.

Print and Space Carrier

PRS



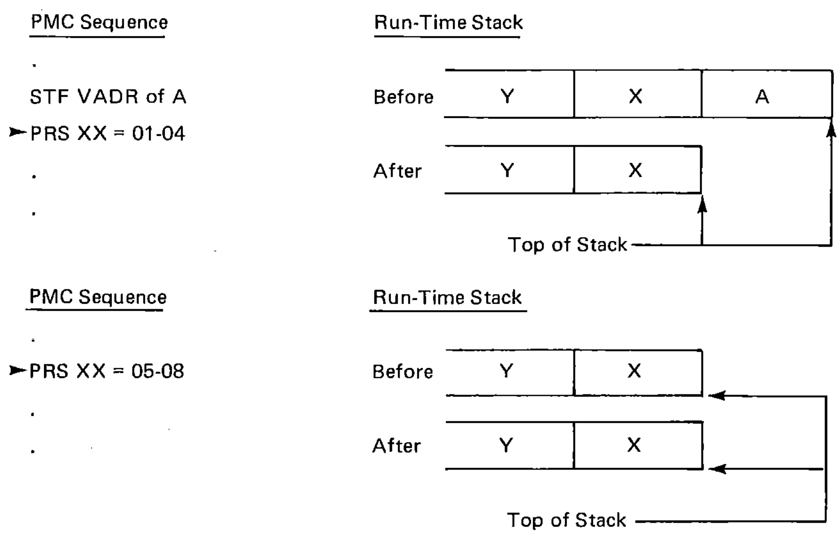
BR1269

The data element at the top stack location is output on the system printer, or the system printer carrier is positioned, under control of parameter XX. When XX specifies data element output, that element is deleted from the top stack location.

The possible XX codes (hexadecimal) and the functions they perform are:

XX	*Function*
01	Print and space suppress
02	Print and space to long zone
03	Print and space to short zone
04	Print and carrier return
05	Space to long zone
06	Space to short zone
07	Carrier return
08	Conditional carrier return

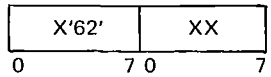
Example:



BR1270

Print Using Image

PRU



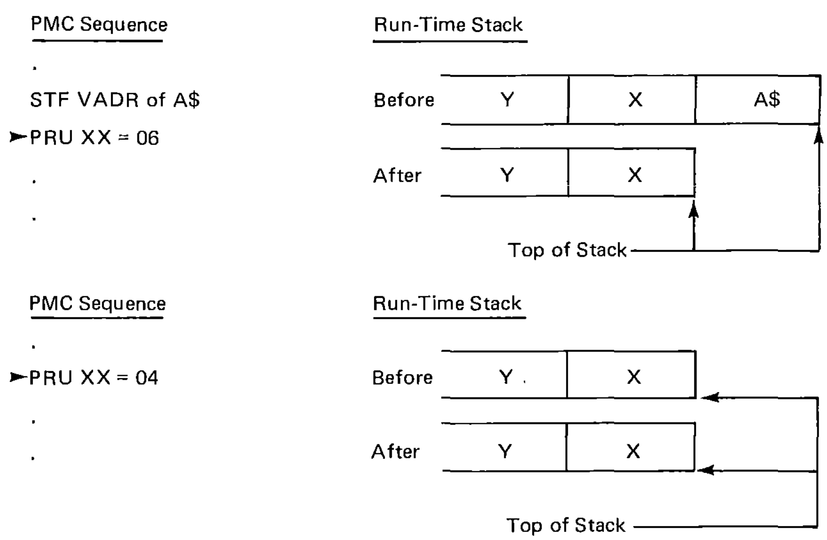
BR1271

The data element at the top stack location is output according to the current image, or the current image is output, on the system printer under control of parameter XX. When XX specifies data element output, that element is deleted from the top stack location.

The possible XX codes (hexadecimal) and the functions they perform are:

XX	*Function*
01	Establish null image specification.
04	Establish first image character string segment.
05	Establish secondary image character string segment.
02	Statement contains no data list.
06	Print arithmetic or character expression, including first constant established for a character string but excluding a null character string ('').
07	Print any constant established for a character string except for the first constant in that string series.
03	Print a null character string ('').
12	Same as code 02 except indicates final PRU instruction for this list.
16	Same as code 06 except indicates final PRU instruction for this list.
17	Same as code 07 except indicates final PRU instruction for this list.
13	Same as code 03 except indicates final PRU instruction for this list.

Example:



BR1272

Logical Operations

Compare Floating Point Values

CMF

X'40'

0 7

BR1273

The floating-point value at the second stack location is compared algebraically to the floating-point value at the top stack location. A compare condition code is set specifying greater than, equal to, or less than. Both of the floating-point values are deleted from the stack.

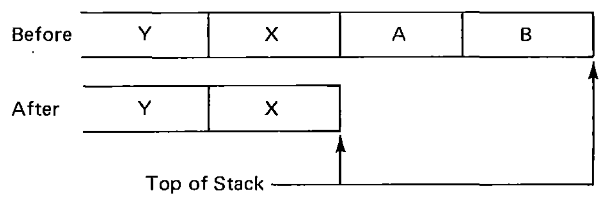
Example:

If A = B . . . .

PMC Sequence

.
STF VADR of A
STF VADR of B
➤CMF
.
.

Run-Time Stack



BR1274

Compare Character Elements

CMC

X'42'

0 7

BR1275

The character field at the second stack location is compared with the character field at the top stack location. A compare condition code is set specifying a collating sequence greater than, equal to, or less than. Both of the character fields are deleted from the stack.

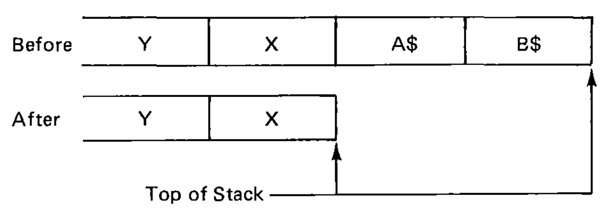
Example:

If A$ = B$ . . .

PMC Sequence

.
STC VADR of A$
STC VADR of B$
➤CMC
.
.

Run-Time Stack



BR1276

Branch On Condition

BRC

X'44'	VADR		CC
0 7	0 7	0 7	0 7

BR1277

Control is transferred to that pseudo instruction which begins at VADR when code CC agrees with the current compare condition. If the compare condition is not met, control is passed to the next sequential pseudo instruction.

The possible CC codes (hexadecimal) and the functions they perform are:

CC	*Function*
82	Branch low
84	Branch equal
88	Branch high
92	Branch not low
94	Branch not equal
98	Branch not high

The BRC pseudo instruction always follows a CMF or CMC pseudo instruction. The contents of the run-time stack are unaffected by the execution of the BRC pseudo instruction.

Branch Unconditionally

BRA

X'46'	VADR	
0 7	0 7	0 7

BR1278

Control is transferred unconditionally to the pseudo instruction that begins at VADR.

The contents of the run-time stack are unaffected by the execution of this pseudo instruction.

Branch and Delete Function Entry

BRD

X'48'	VADR	
0 7	0 7	0 7

BR1279

The entry at the top of the user function activity table is deleted, and control is transferred to the pseudo instruction that begins at VADR. Refer to "Function Call–Indirect" (FCI).

The contents of the run-time stack are unaffected by the execution of this pseudo instruction.

Branch and Suppress Execution

BNX

X'4A'	VADR
0 7	0 7 0 7

BR1280

Control is transferred to the pseudo instruction that begins at VADR. The first BRA instruction encountered after the transfer of control is suppressed (not executed).

The contents of the run-time stack are unaffected by the execution of this pseudo instruction.

Example:

BASIC Statements	PMC Sequence
0100 PRINT USING 120 . . .	STH 0100
	STA VADR of (A)
	➤BNX VADR of 0120
	(A)
0110 . . .	STH 0110
	.
0120 : (image)	STH 0120
	*BRA VADR of 0130
	.
	.
	BRS (VADR of (A) is stacked)
0130 . . .	STH 0130
	.

* This BRA is deactivated by the BNX instruction.

BR1281

Branch to Stacked Address

BRS

X'4C'	VADR
0 7	0 7 0 7

BR1282

Control is transferred to the pseudo instruction that begins at the virtual address at the top stack location. The virtual address is deleted from the top stack location.

Example:

PMC Sequence	Run-Time Stack			
.				
STA VADR	Before	Y	X	VADR
.				
.	After	Y	X	
➤BRS				
.			Top of Stack	
.				

BR1283

Initiate FOR Loop

FOR

X'4E'	VADR
0 7	0 7 0 7

BR1284

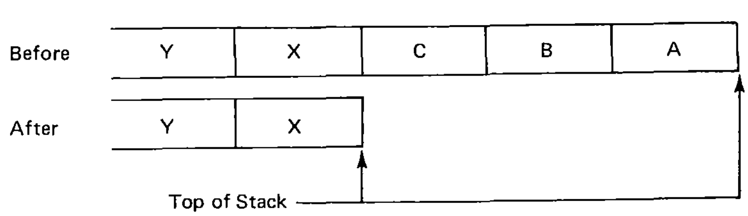
This instruction is always paired with a trailing NXT instruction. VADR is the virtual address of the loop control variable. The floating-point value at the third stack location (the loop control initial value) is saved in a control variable work area. The floating-point values at the second and top stack locations (the final value and increment, respectively) are stored in a DWA-defined work area following the NXT instruction in the PMC sequence. The three floating-point values are deleted from the stack and control is transferred to the NXT instruction such that control variable retrieval and incrementation are bypassed.

The following example illustrates two nested levels of FOR-NEXT BASIC statement pairs.

Example Using FOR, NXT, and DWA Pseudo Instructions:

BASIC Statements	PMC Sequence
0100 FOR D = C TO B STEP A	STH 0100
	STF VADR of C
	STF VADR of B
	STF VADR of A
	FOR VADR of D
	NXT VADR of Loop D Exit
	DWA 16
	(8 bytes; limit B)
	(8 bytes; increment A)
0110	STH 0110
(statements)	.
0170	.
0180 FOR H = G TO F STEP E	STH 0180
	STF VADR of G
	STF VADR of F
	STF VADR of E
	FOR VADR of H
	NXT VADR of Loop H Exit
	DWA 16
	(8 bytes; limit F)
	(8 bytes; increment E)
0190	STH 0190
(statements)	.
0220	.
0230 NEXT H	STH 0230
	BRA VADR to Continue Loop H
0240	STH 0240
(statements)	.
0280	.
0290 NEXT D	STH 0290
	BRA VADR to Continue Loop D
0300 (statement)	STH 0300
	.
	.
	.

Run-Time Stack (first FOR instruction)



BR1285

Perform Next Step

NXT

X'50'	VADR
0 7	0 7 0 7

BR1286

This instruction is always paired with a preceding FOR instruction, and always precedes a DWA instruction that defines a work area containing the final value and increment for the loop. Refer to "Initiate FOR Loop" (FOR). The loop control variable stored at the VADR of the FOR instruction is placed in a control variable work area and modified using the loop increment. When the working value of the control variable exceeds the final value, control is transferred to the pseudo instruction that begins at the VADR of the NXT instruction (exit). If the working value of the control variable does not exceed the final value, it is stored at the VADR of the FOR instruction (loop control variable) and control is passed to the STH instruction of the next sequential statement.

The contents of the run-time stack are unaffected by the execution of the NXT pseudo instruction.

Stack and Unstack Operations

Stack Floating Point Value

STF

X'20'	VADR
0 7	0 7 0 7

BR1287

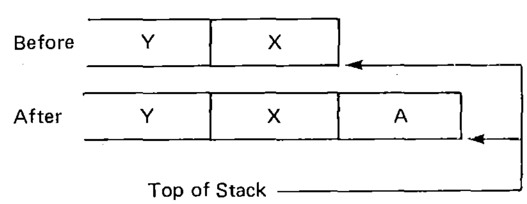
The floating-point value referenced by VADR is moved from virtual memory to the top stack location. The length of the element is precision dependent. The actual data element is moved to the stack.

Example:

PMC Sequence

.
.
➤STF VADR of A
.
.

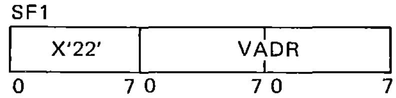
Run-Time Stack



BR1288

Stack Arithmetic Vector Element

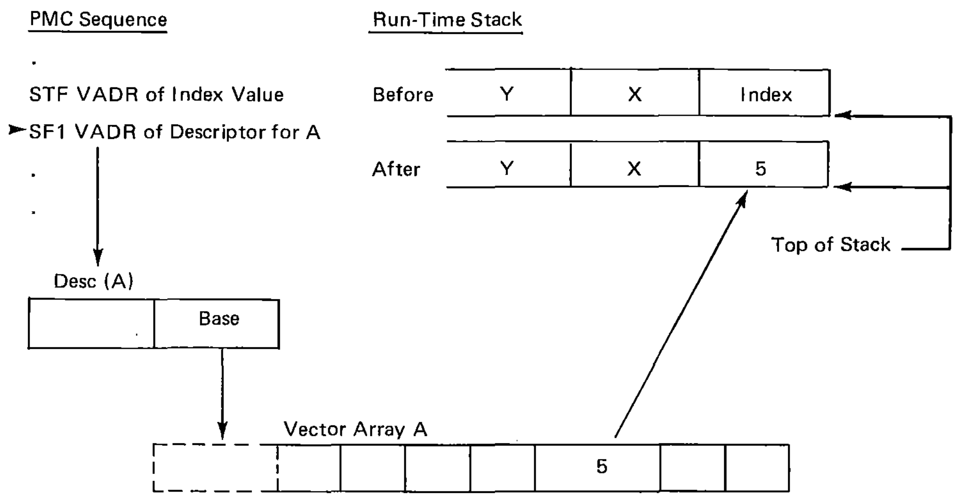
SF1



BR1289

The floating-point value at the top stack location is truncated and converted to a binary indexing value which is used to reference an element in the one-dimensional arithmetic array. VADR is the virtual address of the descriptor (array dope vector) for this array. The indexing value at the top stack location is replaced by the floating-point array element. The length of the element is precision dependent. The actual data element is moved from the array to the stack.

Example:

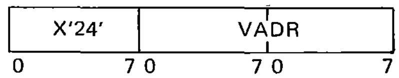


Element Address = Base + (Index * Element Length)

BR1290

Stack Arithmetic Matrix Element

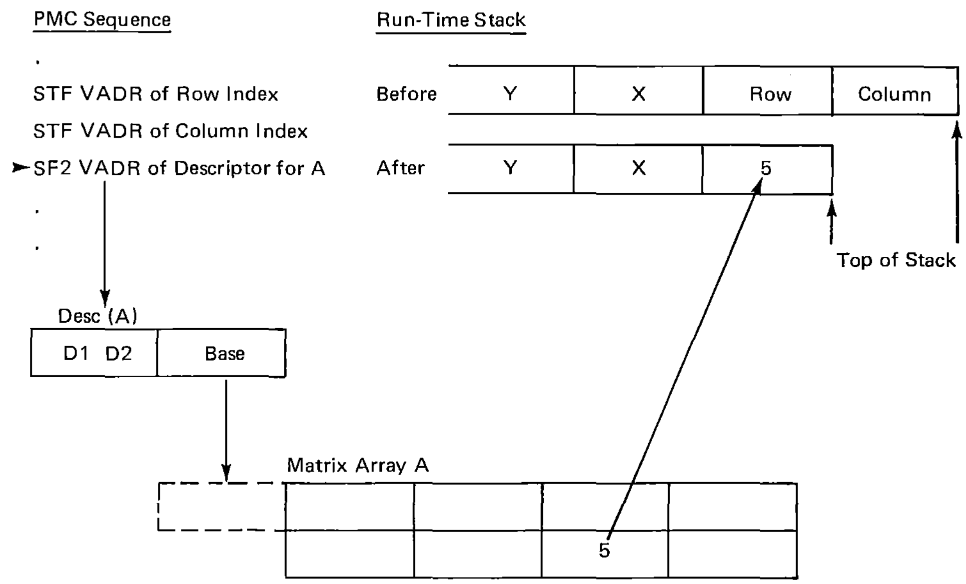
SF2



BR1291

The floating-point values at the second stack location and at the top stack location are truncated and converted to binary indexing values. Respectively, these two indexing values define row and column values required to reference a single element in a two-dimensional arithmetic array. VADR is the virtual address of the descriptor (array dope vector) for this array. The two indexing values at the top stack location are replaced by the floating-point array element.

Example:

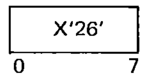


Element Address = Base + ((Row − 1) * D2 + Column) * Element Length, where D2 is the array column dimension and element length is precision dependent.

BR1292

Unstack Floating Point Element

USF



BR1293

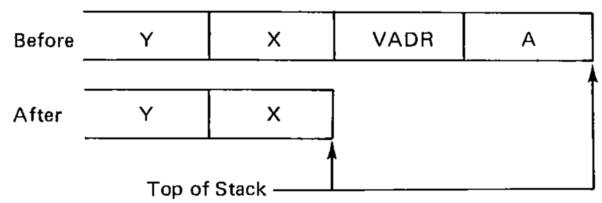
The floating-point value at the top stack location is stored in virtual memory at the address contained in the second stack location. The value and the referenced address are deleted from the stack.

Example:

PMC Sequence

.
STA VADR of VM Location
.
STF VADR of A
➤USF
.

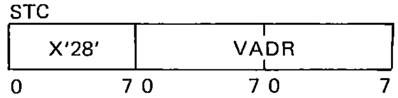
Run-Time Stack



BR1294

Stack Character Field

STC



BR1295

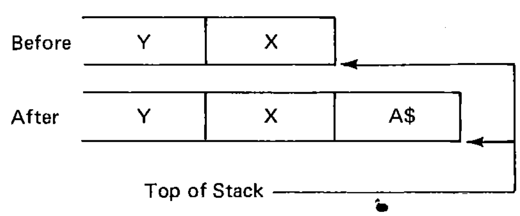
The character field referenced by VADR is moved from virtual memory to the top stack location. The length of a character field is 19 bytes. The actual content of the field is moved to the stack.

Example:

PMC Sequence

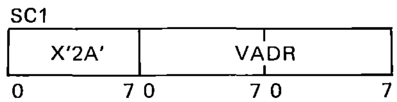
.
.
➤STC VADR of A$
.
.

Run-Time Stack



BR1296

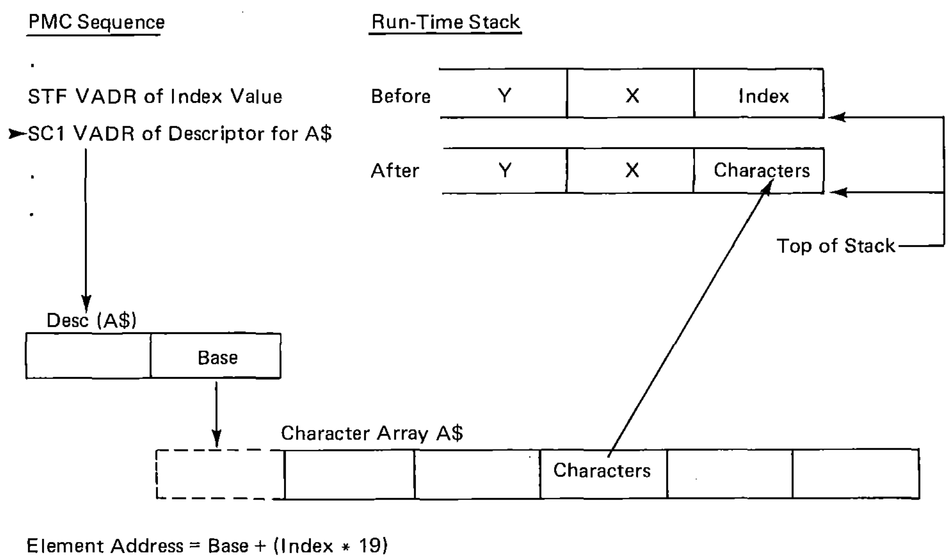
Stack Character Array Field



BR1297

The floating-point value at the top stack location is truncated and converted to a binary indexing value which is used to reference a field in a character array. VADR is the virtual address of the descriptor (array dope vector) for this character array. The indexing value at the top stack location is deleted and the character array field is moved to the top stack location. The length of the character array field is 19 bytes. The actual content of the field is moved to the stack.

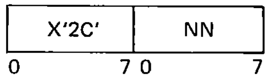
Example:



BR1298

Unstack Character Element

USC



BR1299

The character field at the top stack location is stored in virtual memory at the virtual addresses contained in stack locations 2 through NN+1. The character field and all of the referenced addresses are deleted from the stack.

Example:

0100 LET A$, B$, C$ = D$

<u>PMC Sequence</u>

STH 0100

STA VADR of A$

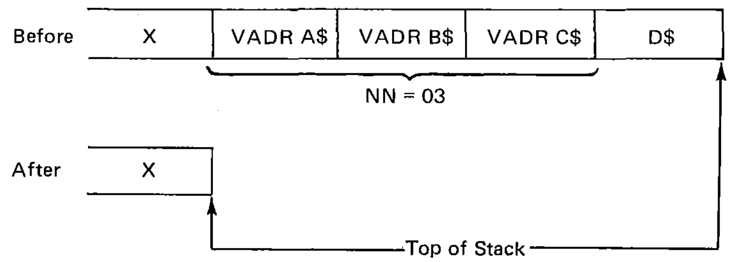
STA VADR of B$

STA VADR of C$

STC VADR of D$

➤USC 03

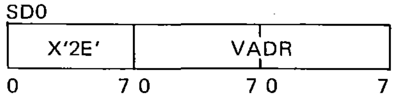
<u>Run-Time Stack</u>



BR1300

Stack Arithmetic Array Descriptor

SDO



BR1301

The arithmetic array descriptor (array dope vector) referenced by VADR is moved to the top stack location. Arithmetic array dope vectors are eight bytes in length. The actual contents of the dope vector are moved to the stack.

Example:

<u>PMC Sequence</u>

.

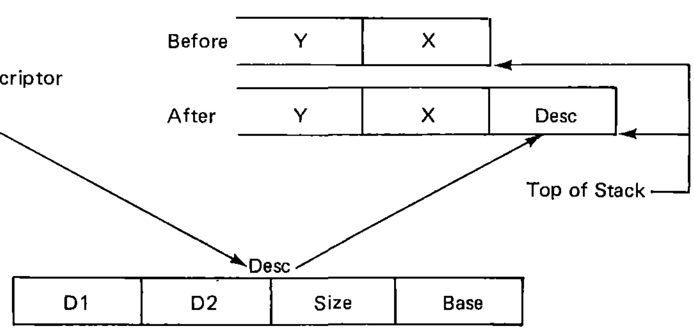
.

➤SDO VADR of Descriptor

.

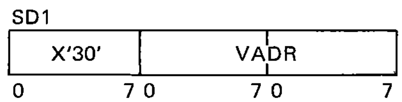
.

<u>Run-Time Stack</u>



BR1302

Stack Arithmetic Array Descriptor (Redimension 1)

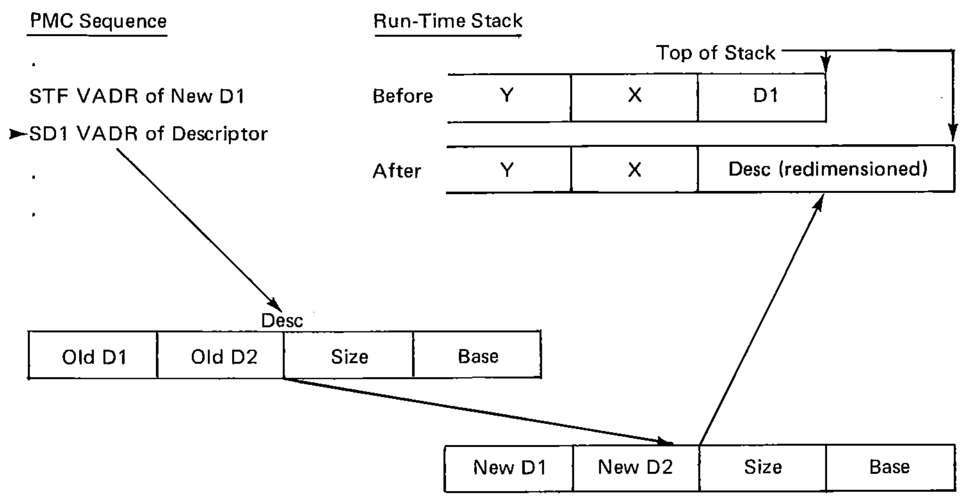


BR1303

The floating-point value at the top stack location is truncated and converted to a binary array dimension. This new dimension replaces the single dimension (D1) in the arithmetic array descriptor (array dope vector) referenced by VADR. The binary array dimension value is deleted from the stack. The redimensioned arithmetic vector descriptor is moved to the top stack location. (The descriptor is also redimensioned in virtual memory.)

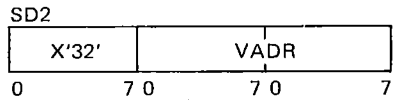
Arithmetic array dope vectors are eight bytes in length. The actual contents of the redimensioned dope vector is moved to the stack.

Example:



BR1304

Stack Arithmetic Array Descriptor (Redimension 2)

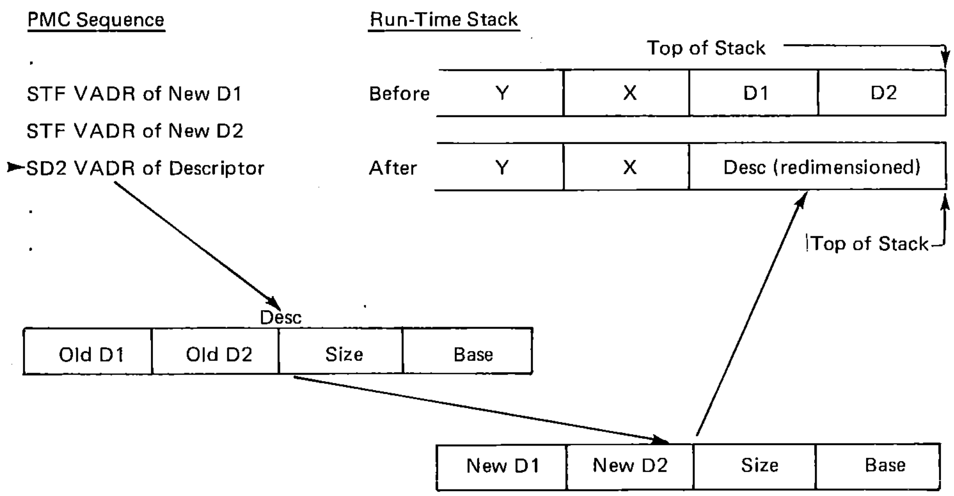


BR1305

The floating-point value at the second stack location is truncated and converted to a binary array row dimension (D1). The floating-point value at the top stack location is truncated and converted to a binary array column dimension (D2). These new dimensions replace D1 and D2 in the arithmetic array descriptor (array dope vector) referenced by VADR. Both binary array dimension values are deleted from the stack. The redimensioned arithmetic matrix descriptor is moved to the top stack location. (The descriptor is also redimensioned in virtual memory.)

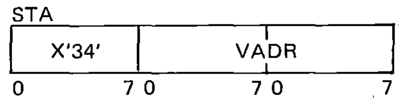
Arithmetic array dope vectors are eight bytes in length. The actual contents of the redimensioned dope vector are moved to the stack.

Example:



BR1306

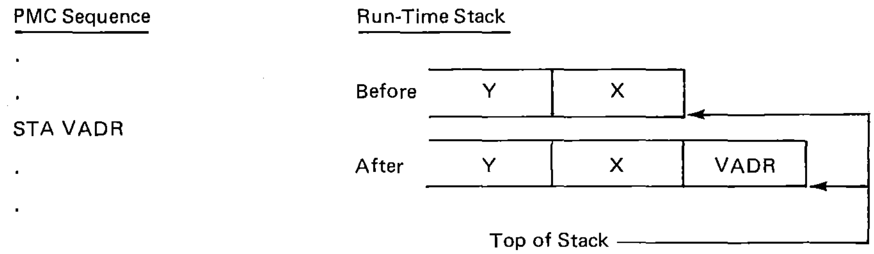
Stack Virtual Address



BR1307

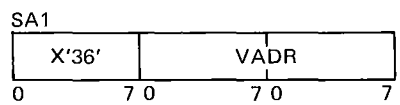
The virtual address in the second and third bytes (VADR) is moved to the top stack location.

Example:



BR1308

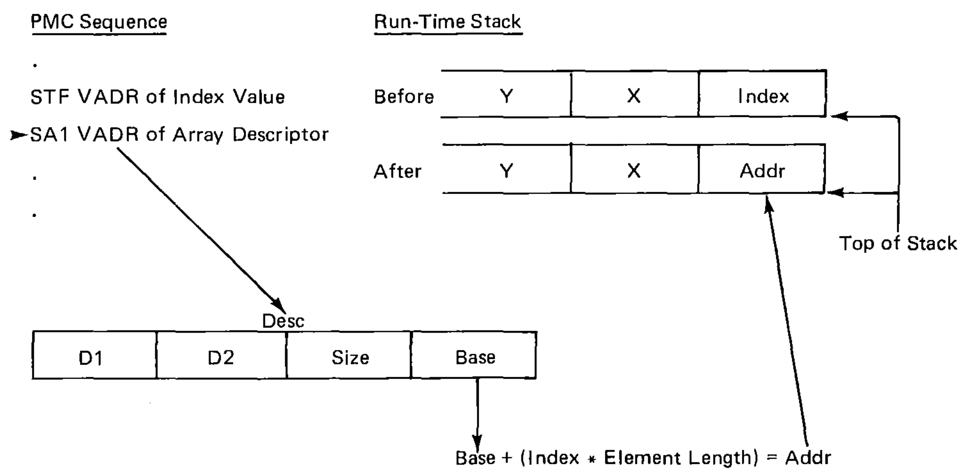
Stack Vector Array Element Address



BR1309

The floating-point value at the top stack location is truncated and converted to a binary indexing value. This indexing value is used to determine the virtual address of a single element in a one-dimensional arithmetic array. VADR is the virtual address of the descriptor (array dope vector) for this array. The indexing value at the top stack location is replaced by the virtual address of the referenced array element.

Example:



BR1310

Stack Matrix Array Element Address

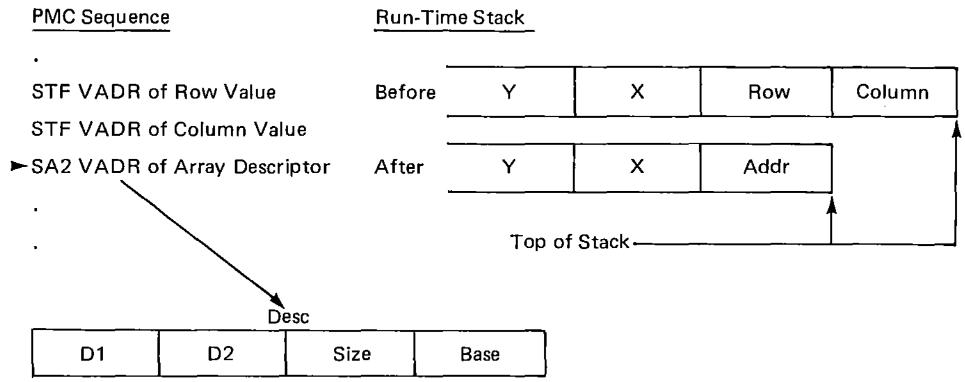
SA2

X'38'	VADR	
0 7	0 7	0 7

BR1311

The floating-point values at the second stack location and at the top stack location are truncated and converted to binary indexing values. Respectively, these two indexing values define row and column values required to determine the virtual address of a single element in a two-dimensional arithmetic array. VADR is the virtual address of the descriptor (array dope vector) for this array. The two indexing values at the top stack location are replaced by the virtual address of the referenced array element.

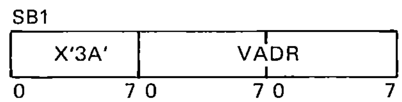
Example:



Element Address = Base + ((Row − 1) * D2 + Column) * Element Length, where D2 is the array column dimension and element length is precision dependent.

BR1312

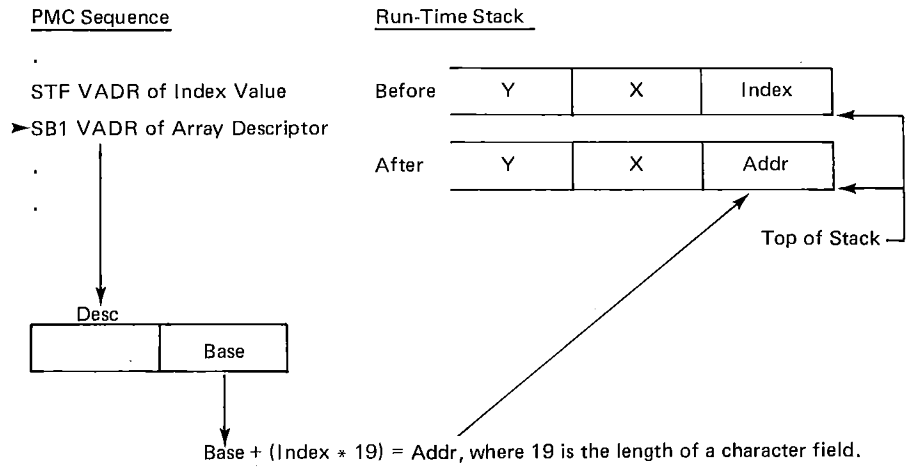
Stack Character Array Element Address



BR1313

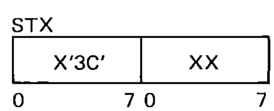
The floating-point value at the top stack location is truncated and converted to a binary indexing value. This indexing value is used to determine the virtual address of a single field in a character array. VADR is the virtual address of the descriptor (array dope vector) for this array. The indexing value at the top stack location is replaced by the virtual address of the field referenced in the character array.

Example:



BR1314

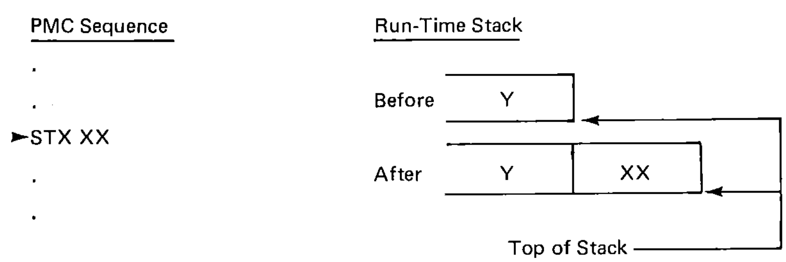
Stack Execution Control Code



BR1315

The execution control code in byte 2 (XX) is moved to the top stack location.

Example:



BR1316

Compute Stacked Address

CSA

X'3E'	NN
0 7	0 7

BR1317

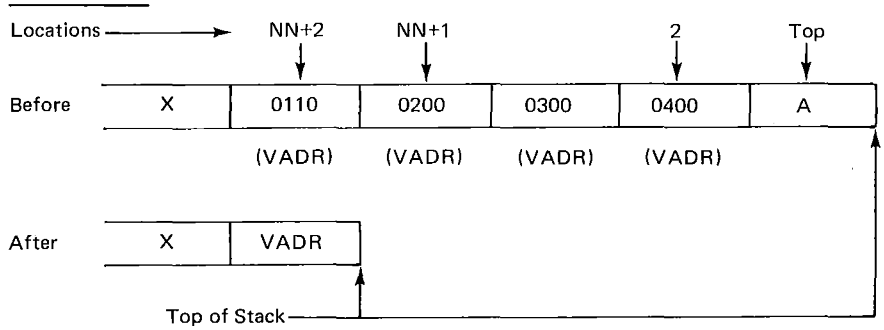
The floating-point value at the top stack location is truncated and converted to a binary index value. This index value references a virtual address previously placed in the stack (i.e., if the index value is I, the (NN+2–I)th stack entry is referenced). If the index value is in the range 1 through NN, the referenced virtual address is selected. If the index value is outside the range 1 through NN, the virtual address at stack location NN+2 is selected. The binary index value, at the top of the stack, and the series of virtual addresses, in stack locations 2 through NN+2, are deleted from the stack and the selected virtual address is placed at the top stack location.

Example:

BASIC Statements	PMC Sequence
0100 GOTO 200,300,400 ON A	STH 0100
	STA VADR of 0110
	STA VADR of 0200
	STA VADR of 0300
	STA VADR of 0400
	STF VADR of A
	CSA 03
	*BRS
0110	STH 0110
.	.
0200	STH 0200
.	.
0300	STH 0300
.	.
0400	STH 0400

*Branch depends on value of A.

Run-Time Stack



For example, if A = 2, VADR = VADR 0300; if A = 99, VADR = VADR 0110.

BR1318

Miscellaneous Operations

Supervisor Call

SVC

X'02'
0 7

BR1319

This pseudo instruction:

1. Closes all activated external data files (those that were activated by an ADF instruction, but not closed by a CLS instruction).
2. Writes all modified pages in the core paging area back to virtual memory.
3. Resets the execution mode indicator in the system communication area.
4. Causes the interpreter to pass control to the system control program (#GUFUD), via $CARPL in the system nucleus.

This pseudo instruction marks the termination of the System/3 BASIC user-program execution.

Halt Execution

HLT

X'04'
0 7

BR1320

This instruction initiates a program-requested interruption. Execution of the System/3 BASIC program is halted and control is passed to $PAUSD in the system nucleus, placing the interpreter program in the execution pause state. If the interpreter program is resumed, execution continues with the next sequential pseudo instruction following the HLT.

Statement Header

STH

X'64'	Line
0 7	0 7 0 7

BR1321

With the exception of the image (:) statement, the pseudo instruction sequence for each translated BASIC statement begins with an STH instruction. The compiler distributor (BHDIST) generates the STH pseudo instruction in virtual memory, preceding any pseudo machine code generated specifically for statement execution.

Execution is interrupted if an interrupt condition is in effect or if execution is in STEP mode. The STH instruction identifies the beginning of a statement and its line number reference. "Line" contains the binary line number reference.

The STH pseudo instruction performs no logical operation and makes no modifications to either the run-time stack or the contents of virtual memory. Control is passed to the next sequential pseudo instruction.

Image Statement Header (:)

IMH

X'66'	Line
0 7	0 7 0 7

BR1322

The pseudo instruction sequence, for each translated image (:) statement, begins with an IMH instruction.

Execution of the IMH pseudo instruction is identical to that of the STH pseudo instruction with this exception—when the pseudo instruction executed immediately preceding the IMH instruction is a BNX instruction, the IMH instruction becomes a no-op.

The IMH instruction performs no logical operation and makes no modifications to either the run-time stack or the contents of virtual memory. Control is passed to the next sequential pseudo instruction.

End of Page

EOP

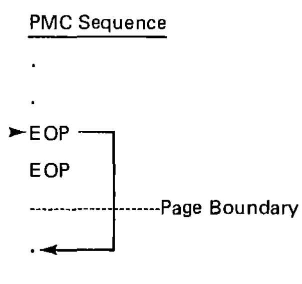
X'68'
0 7

BR1323

Each pseudo machine code virtual page is terminated with at least one EOP instruction. EOP execution results in control being passed to the first pseudo instruction that appears in the next sequential virtual page.

When more than one EOP terminates a page, only the first is executed. (Only the first is displayed in a maintenance utility dump of virtual memory.) The contents of the run-time stack are not affected by the execution of this instruction.

Example:



BR1324

Nonexecutable Operations

Define Constant Address

DCA

X'6A'	VADR	
0 7	0 7	0 7

BR1325

The single arithmetic element or character field at VADR is defined as a data element in the internal data file. The position of the element in the file is directly related to the position of the DCA instruction with respect to other DCA instructions.

All DCA pseudo instruction sequences are chained together by DDL pseudo instructions to form the internal data file. Refer to "Define Data Linkage" (DDL) for an example.

All sequences of DCA and DDL pseudo instructions have a BRA (branch unconditional) preceding them to bypass these nonexecutable instructions.

Define Data Linkage

DDL

X'6C'	VADR	
0 7	0 7	0 7

BR1326

The DDL pseudo instruction always follows a string of one or more DCA pseudo instructions. VADR is the virtual address that provides the linkage to the next sequential DCA instruction in the internal data file chain. A DDL instruction, with a VADR containing X'0000', marks the end of the internal data file.

All sequences of DCA and DDL pseudo instructions have a BRA (branch unconditional) preceding them to bypass these nonexecutable instructions.

Example Using DCA and DDL Pseudo Instructions:

Basic Statements	PMC Sequence	
0100 DATA 1, 2, 3	STH 0100	
	BRA VADR of 0110	
	DCA VADR of 1	Internal Data File Pointer
	DCA VADR of 2	
	DCA VADR of 3	
	DDL VADR of Next DCA	
0110 DATA 4, 5	STH 0110	
	BRA VADR of 0120	
	DCA VADR of 4	
	DCA VADR of 5	
	DDL VADR of Next DCA	
0120	STH 0120	
.	.	
.	.	
0200 DATA 6	STH 0200	
(last data statement)	BRA VADR of 0210	
	DCA VADR of 6	
	DDL 0000	End of Internal Data File
0210	STH 0210	
.	.	
.	.	

BR1327

Define Work Area

DWA

X'6E'	NN
0 7	0 7

BR1328

NN is equal to the number (in binary) of bytes in the work area. The work area initially contains binary 0's.

The DWA pseudo instructions are preceded by either a NXT or a BRA (branch unconditional) instruction. Either of these instructions provides insurance that the DWA cannot be executed.

Example:

PMC Sequence

.

BRA VADR

➤DWA 09

0000|0000|0000|0000|00

.

.

BR1329

End of Program

EOF

X'70'
0 7

BR1330

The EOF instruction marks the end of the functional pseudo instructions. This instruction is not executed. This is the last instruction generated for a System/3 BASIC user program with the exception of EOP pseudo instructions which are padded to the end of the page. (These EOP instructions are not displayed by the maintenance utility VM dump.) The EOF pseudo instruction is always preceded by a SVC pseudo instruction. The SVC ensures that the EOF cannot be executed. This sequence is always generated.

Example:

PMC Sequence

.

.

SVC

➤EOF

EOP

EOP

.

EOP

-------------------Page Boundary

BR1331

FLOATING-POINT ARITHMETIC

Floating-point arithmetic automatically maintains decimal point placements (scaling) during computations in which the range of values used varies widely or is unpredictable (Figure 3-171).

The key to floating-point data representation is the separation of the significant digits of a number from the size (scale) of the number. Thus, the number is expressed as a fraction times a power of 10. A floating-point number has two associated sets of values. One set represents the significant digits of the number and is called the fraction. The second set specifies the power (exponent) to which 10 is raised and indicates the location of the decimal point in the number.

These two numbers (the fraction and exponent) are recorded in a single field. Since each of these two numbers is signed, some method must be employed to express two signs in the field. A negative fraction is indicated by the presence of a sign bit in the field. The sign of the exponent is expressed in excess 128 arithmetic; that is, the exponent is added as a signed number of 128. The resulting number is called the characteristic. Since the decimal range of the exponent is −98 through 0 to +99, the range of the characteristic is 30 to 227 (X'1E' to X'E3'). (Refer to Figure 3-172 to convert characteristics to exponents, or the reverse.)

The number is always normalized to provide a fraction with the greatest possible precision. The number is normalized when the decimal point is immediately to the left of the first significant digit. The exponent is raised or lowered until the decimal point is positioned. (Example: 1234.56 is normalized to a fraction of 0.123456 with an exponent of 10^4.)

Floating-point data is recorded in either standard (short) or long precision. Standard precision provides for 7 significant digits and long precision provides for 15. The significant digits, when represented in packed-decimal format, occupy a five-byte field for standard precision. The field is extended to nine bytes for long precision. Refer to Figure 3-173 for the format of these fields. Arithmetic unpacked-decimal format is shown on Figure 3-174.

Conversion Example: Convert a standard-precision, packed-decimal, floating-point value (1 | 1 23 00 00 | 7E) to an unnormalized decimal number:

1. Disregarding the first half-byte and the last byte, the packed-decimal, normalized fraction is 0.1230000.
2. The exponent is developed from the characteristic (last byte):

   Characteristic − Base = Exponent

   X'7E' = 126 − 128 = −2
3. Bit 3 of the first half-byte indicates that the fraction is negative: −0.1230000.
4. The normalized decimal number in floating-point notation is therefore −0.1230000 x 10^{-2}.
5. The unnormalized decimal number is −0.00123.

Number	Normalized Value		Internal Floating-Point		
12.345	0.1234500	$\times 10^{2}$	0	1 23 45 00	82
-12.345	-0.1234500	$\times 10^{2}$	1	1 23 45 00	82
12.345×10^{20}	0.1234500	$\times 10^{22}$	0	1 23 45 00	96
12.345×10^{-20}	0.1234500	$\times 10^{-18}$	0	1 23 45 00	6E
12.345×10^{96}	0.1234500	$\times 10^{98}$	0	1 23 45 00	E2
1.2345×10^{-99}	0.1234500	$\times 10^{-98}$	0	1 23 45 00	1E
0.00005	0.5000000	$\times 10^{-4}$	0	5 00 00 00	7C
0.5	0.5000000	$\times 10^{0}$	0	5 00 00 00	80
5000	0.5000000	$\times 10^{4}$	0	5 00 00 00	84
0	*0.0000000	$\times 10^{-98}$	0	0 00 00 00	1E
12.3456789012345	0.123456789012345	$\times 10^{2}$	2	1 23 45 67 89 01 23 45	82

*Special form used to display a standard-precision, packed, floating-point zero.

BR1332A

Figure 3-171. Floating-Point Numbers, Example

X'85'

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1_	Hexadecimal value is outside valid exponent limits.														-98	-97
2_	-96	-95	-94	-93	-92	-91	-90	-89	-88	-87	-86	-85	-84	-83	-82	-81
3_	-80	-79	-78	-77	-76	-75	-74	-73	-72	-71	-70	-69	-68	-67	-66	-65
4_	-64	-63	-62	-61	-60	-59	-58	-57	-56	-55	-54	-53	-52	-51	-50	-49
5_	-48	-47	-46	-45	-44	-43	-42	-41	-40	-39	-38	-37	-36	-35	-34	-33
6_	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17
7_	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
8_	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
9_	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
A_	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
B_	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
C_	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
D_	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
E_	96	97	98	99	Hexadecimal value exceeds exponent limits.											

BR1333A

Figure 3-172. Exponent Conversions (Internal Format to Decimal)

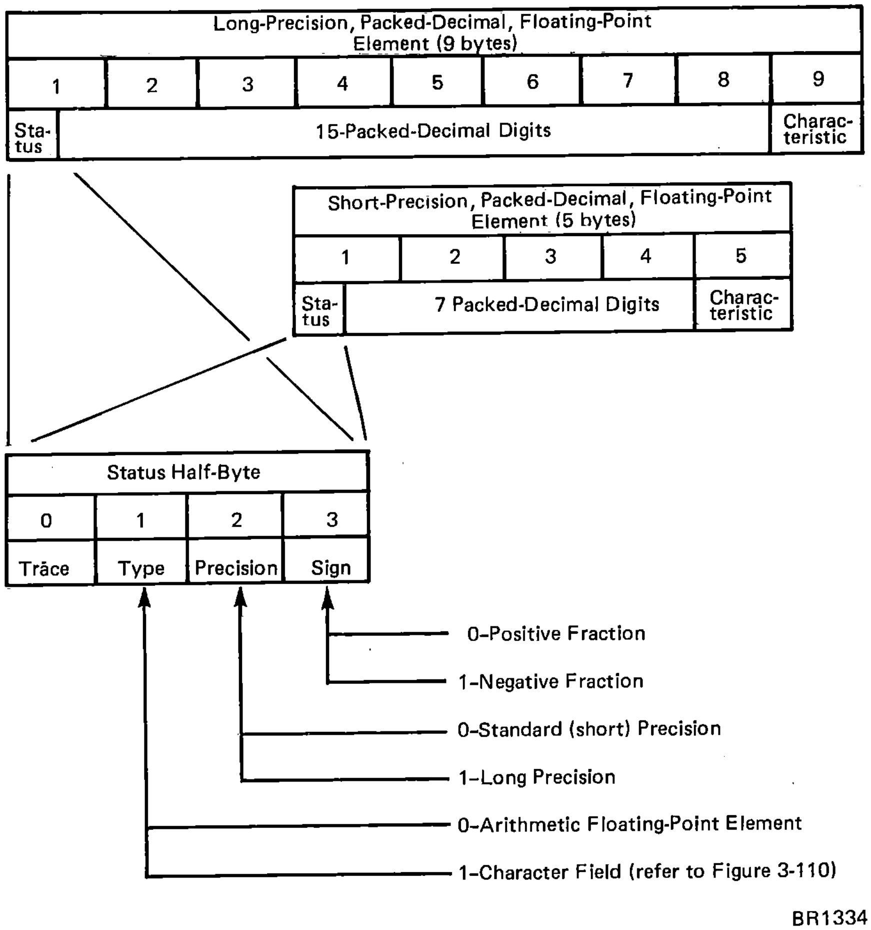
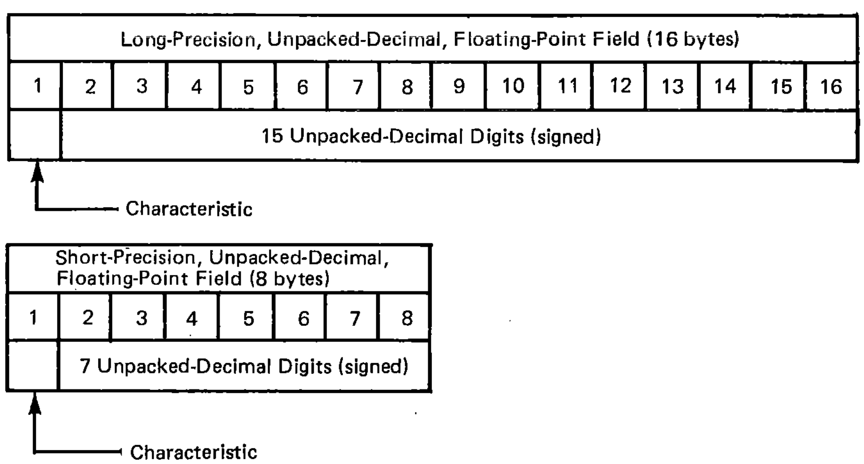


Figure 3-173. Arithmetic Packed-Decimal Format



Note:

1. The zone bits, of the last significant digit, indicate the sign of an unpacked-decimal field; F = positive, D = negative.

2. Refer to "Floating-Point Arithmetic" to determine the exponent from the characteristic.

BR1335

Figure 3-174. Arithmetic Unpacked-Decimal Format

DESK CALCULATOR–DCALC (Figure 3-179)

The first phase of DCALC is called by the command analyzer (#ECMAN) after command key 01 is pressed by the operator. DCALC processes operator commands from the keyboard, one command at a time, until the operator transfers control to #GUFUD via INQUIRY REQUEST (INQ REQ). The concepts of virtual memory and paging are the same for DCALC as for the interpreter. Many of the subroutines used by DCALC (including the paging subroutine, IPGMDL) are identical to those used in the interpreter.

DCALC Cycle

1. DCALC accepts input from the keyboard (command keys, function keys, and data keys).
2. The operator command is interpreted and/or syntax checked.
3. Core-resident and virtual-memory-resident subroutines are called to execute the operator's command.
4. Steps 1 through 3 are performed until DCALC is terminated (INQ REQ).

Organization of Assembly Listings

All modules of the desk calculator are contained in these six assembly listings:

1. DCALC loader–#VLOAD
2. Core-resident routines–#VODKA
3. Virtual-memory-resident subroutines–#VVMRS
4. Virtual-memory-resident subroutines and procedures–##VUFA
5. CRT physical IOCS–#VCRTI
6. DCALC terminator–#VXITI

Core Resident Routines–#VODKA

#VODKA resides in the system program file and is loaded to core at X'0600' immediately following the system nucleus. #VLOAD copies #VVMRS to virtual memory prior to loading #VODKA. The assembly listing of #VODKA contains the following modules arranged in this physical order:

VSVARA–Save areas and push-down (PM) registers
VOTCON–Convert floating point to output
FDIADD/FDISUB–Floating point add/subtract (long precision)
FZIMPY–Floating point multiply (long precision)
FFIDVD–Floating point divide (long precision)
VODKAL–DCALC monitor (control module)
VODIPT–Alpha input table
VENABL–Return next input character
VENDTB–Data key character table
VOUTPT–Output control routine
VSYNTX–Control computation routine
VCONVT–Convert input to floating point (long precision)
TVAREG–Addressable (AM) registers
DPRINT–Matrix printer physical IOCS (actual I/O) (refer to "Conversational I/O Routines–#DPRIN")
DVPRSC–Keyboard physical IOCS (actual I/O)
IPGMDL–Paging subroutine (refer to "Paging Subroutine–IPGMDL")
VINITI–DCALC initialization

Virtual-Memory-Resident Subroutines—#VVMRS and ##VUFA

These components of the desk calculator are copied from the system program file to the first 72 sectors of virtual memory, by the DCALC loader (#VLOAD). Individual pages are read into the core paging area and executed under control of the paging subroutine (IPGMDL). ##VUFA starts at virtual address X'3200'. #VVMRS and ##VUFA contain subroutines and data areas to perform the functions listed in Figure 3-175.

Virtual Address	Disk Address	Symbolic Label	Input Command	Synopsis
0000	0700	VERROR		Error message routine.
0100	0704	VPRINT	PRINT	Print AM and PM registers.
01B0	0704	VPRTBL		Core addresses and virtual addresses for AM and PM registers.
0200	0708	VPRBAA		Virtual memory buffers for all AM and PM registers.
0500	0714	VPOINT	POINT	Change decimal point location.
0600	0718	VNWDEF	PROC	Define a new procedure.
0661	0718	VNWCRD	CARD	Read a procedure from the data recorder.
068D	0718	VDELET	(01)	Delete a procedure.
0700	071C	VNDPRC	END	End a procedure.
0800	0720	VREADI	CARD	Read a procedure from the data recorder.
0900	0724	VRUNIT	(PROG START)	Execute a procedure.
090D	0724	VRUEXC	EXEC	List a procedure as it is executed.
0920	0724	VRULST	LIST	List procedure steps without execution.
0A00	0728	VPLIST		List procedure input buffer.
0A8C	0728	VPUNCH	PUNCH	Punch a procedure on data recorder.
0BA0	072C	VPUBUF		Virtual memory buffer for punch output.
0C00	0730	VSFONE	SF1	Perform statistical function 1.
0C06	0730	VSFTWO	SF2	Perform statistical function 2.
1100	0744	VSFT01		Text messages for SF1 and SF2.
1200	0748	*FKLLGT	LTW	Log base 10.
120B	0748	*FKLLTW	LGT	Log base 2.
1219	0748	*FKLLOG	LOG	Log base e.
1470	0750	*CENXZD		Convert exponent to zoned decimal.
14AD	0750	*CCZDFP		Convert zoned decimal to long-precision floating-point.
1500	0754	*FGLEXP	EXP	Exponentiate.
1800	0780	*FNBPWR	EXP	Floating-point exponentiate.
1900	0784	*FRBSQR	SQR	Square root.
1A00	0788	*FSBCOS	COS	Cosine.
1A1A	0788	*FSBSIN	SIN	Sine.
1D00	0794	*FQLRND	RND	Random number generator.
1E70	0798	*CBFPZD		Convert floating point to zoned decimal.
1EB2	0798	*CDBNZD		Convert binary number to zoned decimal.

BR1336.1

Figure 3-175. Contents of Virtual Memory (DCALC) (Part 1 of 2)

Virtual Address	Disk Address	Symbolic Label	Input Command	Synopsis
1F00	079C	*FTLSEC	SEC	Secant.
1F25	079C	*FTLCSC	CSC	Cosecant.
1F61	079C	*FJBINT	INT	Integer.
1F9C	079C	*FPBRAD	RAD	Convert degrees to radians.
1FAB	079C	*FPBDEG	DEG	Convert radians to degrees.
1FCB	079C	*FABABS	ABS	Absolute value.
1FD6	079C	*FUBSGN	SGN	Sign.
2000	07A0	*FWLCOT	COT	Cotangent.
2028	07A0	*FWLTAN	TAN	Tangent.
2400	07B0	*DFRDIN	READ	Card reader physical IOCS (actual I/O).
2496	07B0	*DFCOUT	PUNCH	Card punch physical IOCS (actual I/O).
2500	07B4	*FHLHCS	HCS	Hyperbolic cosine.
2557	07B4	*FHLHSN	HSN	Hyperbolic sine.
2593	07B4	*FHLHTN	HTN	Hyperbolic tangent.
2700	07BC	*FFBLATN	ATN	Arctangent.
2A00	07C8	*FCLACS	ACS	Arccosine.
2A13	07C8	*FCLASN	ASN	Arcsine.
2B00	07CC			Not used by DCALC (5 pages).
3000	0701	V@VEXT		DCALC terminator (#VXITI).
3200	0709	VSATBL		Procedure address table.
3228	0709	VSARCH		Returns the virtual address of a procedure.
3300	070D	VSAWRT	INQ REQ	Write back modified VM pages and load DCALC terminator (#VXITI).
3400	0711	VSAPRQ+1		Procedures Q through Z (2 pages per procedure—20 pages total).
4800	0781	V@VOVL		Three sectors of saved core starting at label FDIADD in #VODKA.
4B00	078D			The remainder of virtual memory is not used by DCALC.

BR1336.2

Figure 3-175. Contents of Virtual Memory (DCALC) (Part 2 of 2)

The following list contains explanations of the column entries in Figure 3-175:

1. "Virtual Address" is the virtual address entry point to perform the function or the virtual address of a data area or table.
2. "Disk Address" is the disk address of the virtual memory page containing the entry point.
3. "Symbolic Label" is the symbolic name of the entry point in the assembly listing of #VVMRS. An * indicates that the subroutine is identical to a subroutine in #FMLNG (interpreter), both having the same symbolic name.
4. "Input Command" is the user command associated with the execution of that subroutine. Parentheses indicate a keyboard function or command key (example: (PROG START) or (01)).

DCALC Initialization–#VLOAD and VINITI

#VLOAD is the first phase of the desk calculator to be loaded into core. This phase copies #VVMRS, #VXITI, and ##VUFA to virtual memory from the system program file, and then loads the mainline phase of the desk calculator (#VODKA) to core. #VODKA overlays #VLOAD. (Refer to Figure 3-176.)

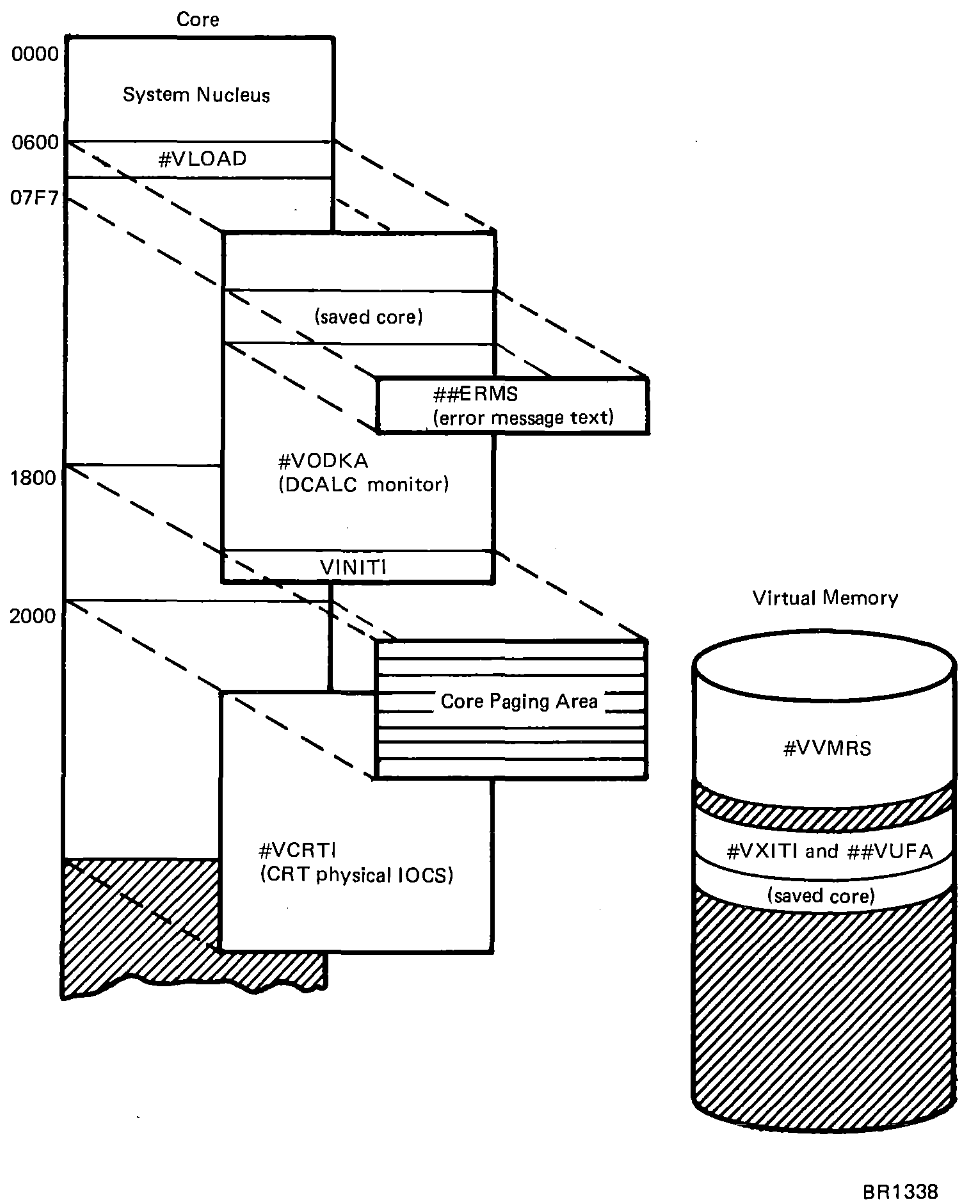


Figure 3-176. DCALC Core and VM Map (with CRT)

The first executable instruction in #VODKA is a branch to VINITI. This initialization routine (VINITI) writes a three-sector area of the core-resident desk calculator to virtual memory. These three sectors are used for error message text blocks (##ERMS) during DCALC error message processing (in routine VERROR). After error message processing, the area is restored from virtual memory. VINITI also loads #VCRTI (CRT physical IOCS and CRT buffer) to high core, if a CRT device is configured. VINITI initializes the core paging area to a size of eight pages (starts at X'1800').

Initialization of the desk calculator is complete when VINITI branches to the label VODONE, in #VODKA.

DCALC Error Messages—VERROR

VERROR is a pageable subroutine in virtual memory (#VVMRS). This page is called to display error messages to the operator. VERROR reads the first two sectors of ##ERMS from the system program file into core, overlaying part of the core-resident desk calculator (#VODKA). These two sectors contain an index to the message text within ##ERMS. The error code (VERERC) is used as a search argument against the index. The entry that is located contains the relative disk address of the message text within ##ERMS. The message text is read from the system program file, overlaying the index.

The message is displayed on the matrix printer by DPRINT. The core area, used as a buffer for the index and message text blocks, is restored from virtual memory. An image of the core-resident code normally occupying this area was written to virtual memory during DCALC initialization.

Label Trace for ENTER+ Function

The following labels trace the execution of the ENTER+ function key. This function places the numeric value just entered, into PM1. The numeric value is converted to a long-precision, unpacked-decimal, floating-point field. This trace illustrates a function executed entirely by core-resident routines.

1. VODONE—This label is the normal return point after the execution of each function.
2. VOD050—Branch to VENABL and set up return linkage.
3. VENABL—Get a single input character from the keyboard.
4. VEN200—Branch to DVPRSC for keyboard physical I/O.
5. DVPRSC—Read one input character from the keyboard and perform error checks.
6. VENRET—Return from keyboard IOCS; input was the ENTER+ function key.
7. VOD060—Check type of input character.
8. VOD110—Character is ENTER+. Branch to VSYNTX to perform syntax check. Calculation is desired.
9. VOD110+10—Branch to VODPSH.
10. VODPSH—Push down PM registers 1 through 9.
11. VOD110+14—Jump to VOD840.
12. VOD840—If no error, branch to VOD890.
13. VOD890—Branch to VOUTPT for output.
14. VOD060—Check type of input character.
15. VOU300—Branch to VSPRNT (interface to DPRINT and/or #VCRTI) to tab (space carrier and/or cursor).
16. VOU300+18—Branch to VOTCON to convert the contents of PM1 to printable format.
17. VOU300+25—Branch to VODPRT (interface to DPRINT and/or #VCRTI) to print PM1.
18. VOU802—Return to DCALC monitor.
19. VOD900—Return to VODONE to process the next operator input.

Label Trace for SIN Function

The following labels trace the execution of the SIN function. This trace illustrates a function that requires paging of subroutines from virtual memory. Those labels marked with an (*) are located in virtual memory subroutines (#VVMRS). Those that are unmarked are located in the core-resident desk calculator (#VODKA).

1. VODONE—This label is the normal return point after execution of each function.
2. VOD050—Branch to VENABL.
3. VENABL—Get a single input character from the keyboard.

4. VEN200–Branch to DVPRSC for keyboard physical I/O.
5. DVPRSC–Read one input character from the keyboard and perform error checks.
6. VENRET–Return from keyboard IOCS; input was the S data key.
7. VEN290–Move the first character into the input buffer at label VSVIPC.
8. VOD060–Check type of input character.
9. VOD300–S is an allowable first input data character.
10. VOD310–Call VENABL to get the second input character. Move it to the input buffer at label VSVIP2.
11. VOD330–Call VENABL to get the third input character. Move it to the input buffer at label VSVIP3. Now the input buffer contains SIN.
12. VOD340–Search the alpha input table (VODIPT) for the SIN entry and branch to the core address in the entry located.
13. VOD600–Save the virtual address of the SIN function subroutine from the alpha input table. Branch to the paging subroutine (IPGCAL). The parameter following the branch instruction is the virtual entry point in the required virtual-memory page (X'1A1A').
14. IPGCAL–Read and lock page X'1F' into the core paging area. This page contains the execution subroutine FPBRAD.
15. *FPBRAD–Convert the contents of PM1 from degrees to radians.
16. *FPB600–Branch to the paging subroutine. I$RTRN is equated to IPGRTN.
17. IPGRTN–Unlock page X'1F' and return to the DCALC monitor at label VOD600+18.
18. VOD660–Branch to the paging subroutine (IPGCAL). The parameter following the branch contains the virtual entry point to the SIN function subroutine. This address was saved previously at label VOD600.
19. IPGCAL–Read and lock page X'1A' (FSBSIN).
20. *FSBSIN–Compute the sine of the value in PM1 and place the result in PM1. Branch to the paging subroutine (I$RTRN).
21. IPGRTN–Unlock page X'1A' and return to the DCALC monitor at label VOD680+1.
22. VOD840–If no error, branch to VOD890.
23. VOD890–Branch to VOUTPT for output.
24. VOUTPT–Request is to print PM1.
25. VOU300–Branch to VSPRNT (interface to DPRINT and/or #VCRTI) to tab (space carrier and/or cursor).
26. VOU300+18–Branch to VOTCON to convert the contents of PM1 to printable format.
27. VOU300+25–Branch to VODPRT (interface to DPRINT and/or #VCRTI) to print PM1.
28. VOU802–Return to DCALC monitor.
29. VOD900–Return to VODONE to process the next operator input.

Keyboard Physical IOCS–DVPRSC (Figure 3-177)

DVPRSC is called when input of one key (function, command, or data) is to be read from the keyboard. The keyboard is enabled and this routine waits for a key to be pressed by the operator.

When the operator presses a key, the data is sensed and then passed to the calling routine. INQ REQ and hard parity errors return to VENEXT. All other keys return to VENRET with the sensed data at label VENKEY. If a typamatic key is pressed, DVPRSC waits until the key is released before branching to VENRET.

The keyboard is locked and enabled on INQ REQ. For all other keys, the keyboard is unlocked and disabled.

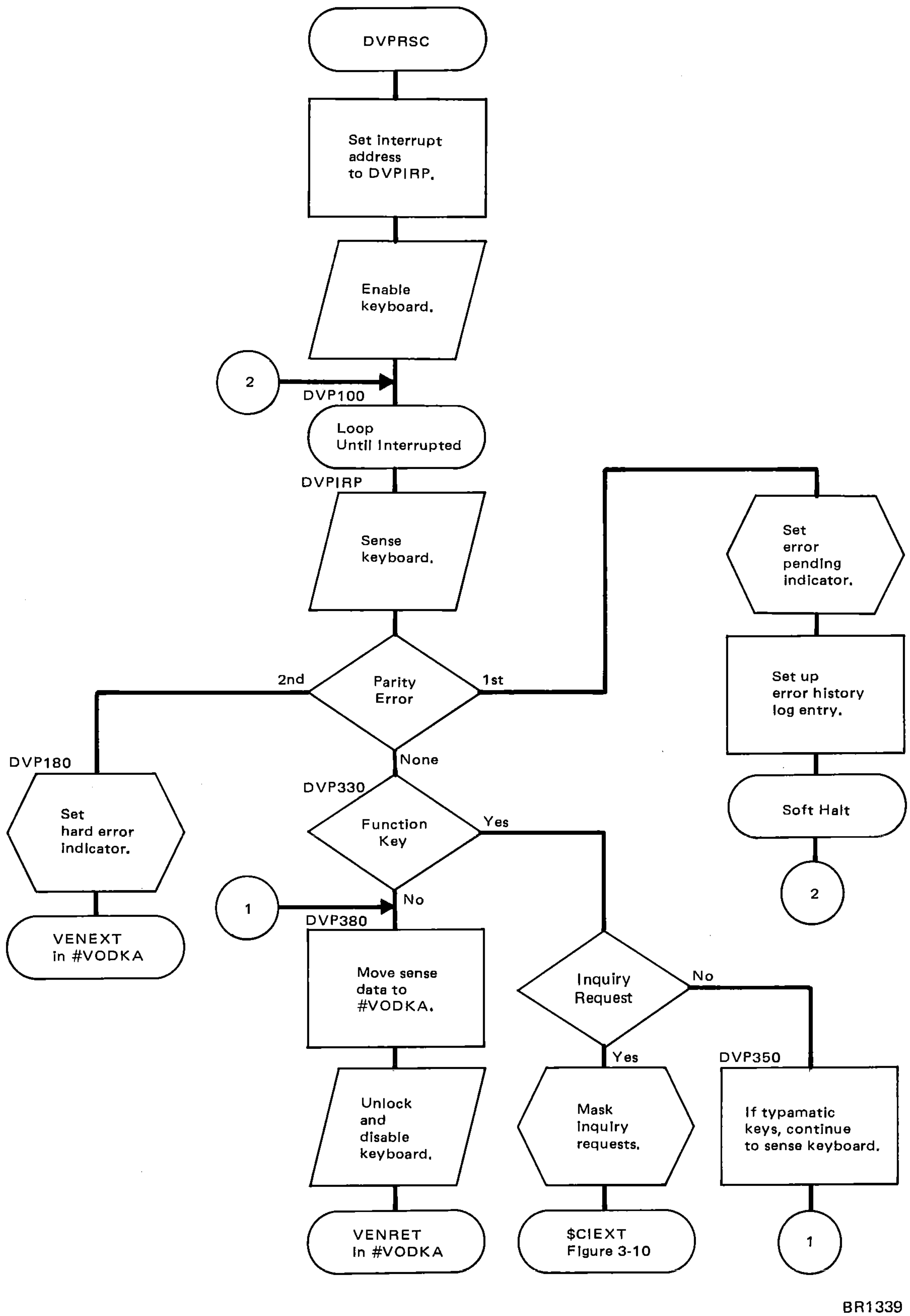


Figure 3-177. Keyboard Physical IOCS (DVPRSC) Flowchart

CRT Physical IOCS—#VCRTI

This program handles all output on the CRT. It is loaded to high core only if the CRT is configured on the system. The CRT output buffer (in the assembly listing of #VCRTI) contains the initial formatted output for the standard DCALC CRT display.

Changes in the display are moved to the output buffer by #VCRTI. Physical I/O and error recovery procedures on the CRT are accomplished by this program.

DCALC Termination—VSAWRT, #VXITI

VSAWRT is a pageable subroutine in virtual memory (##VUFA). This page is called when the operator depresses INQ REQ. VSAWRT writes back all modified pages from the core paging area to virtual memory (these pages may contain modifications to procedures). VSAWRT loads, via $BLOAD in the system nucleus, #VXITI and ##VUFA (procedures) from virtual memory.

#VXITI writes back all procedures (##VUFA) to the system program file before returning, via $CAIPL in the system nucleus, to BASIC mode of operation. This action reflects all modifications that the user may have made, to the procedures, on this activation of the desk calculator. (Refer to Figure 3-178.)

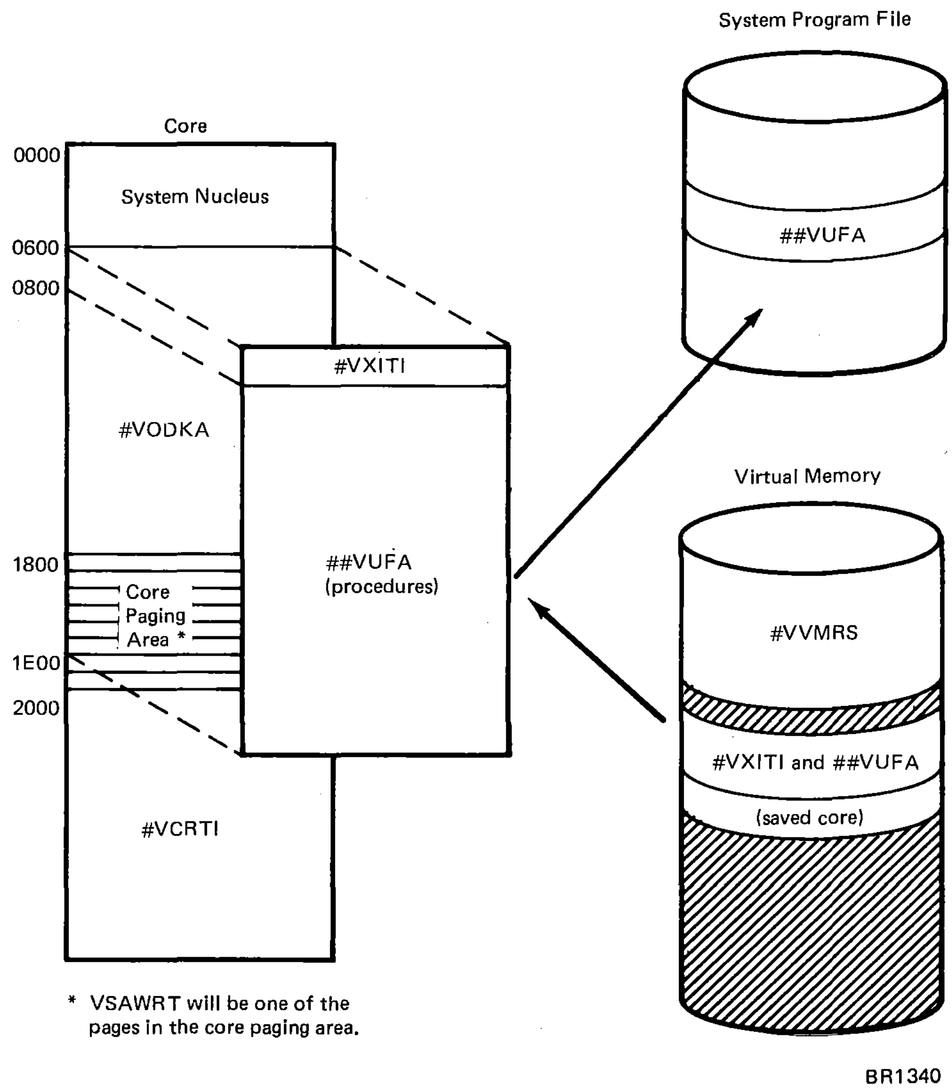


Figure 3-178. DCALC Termination Core Map

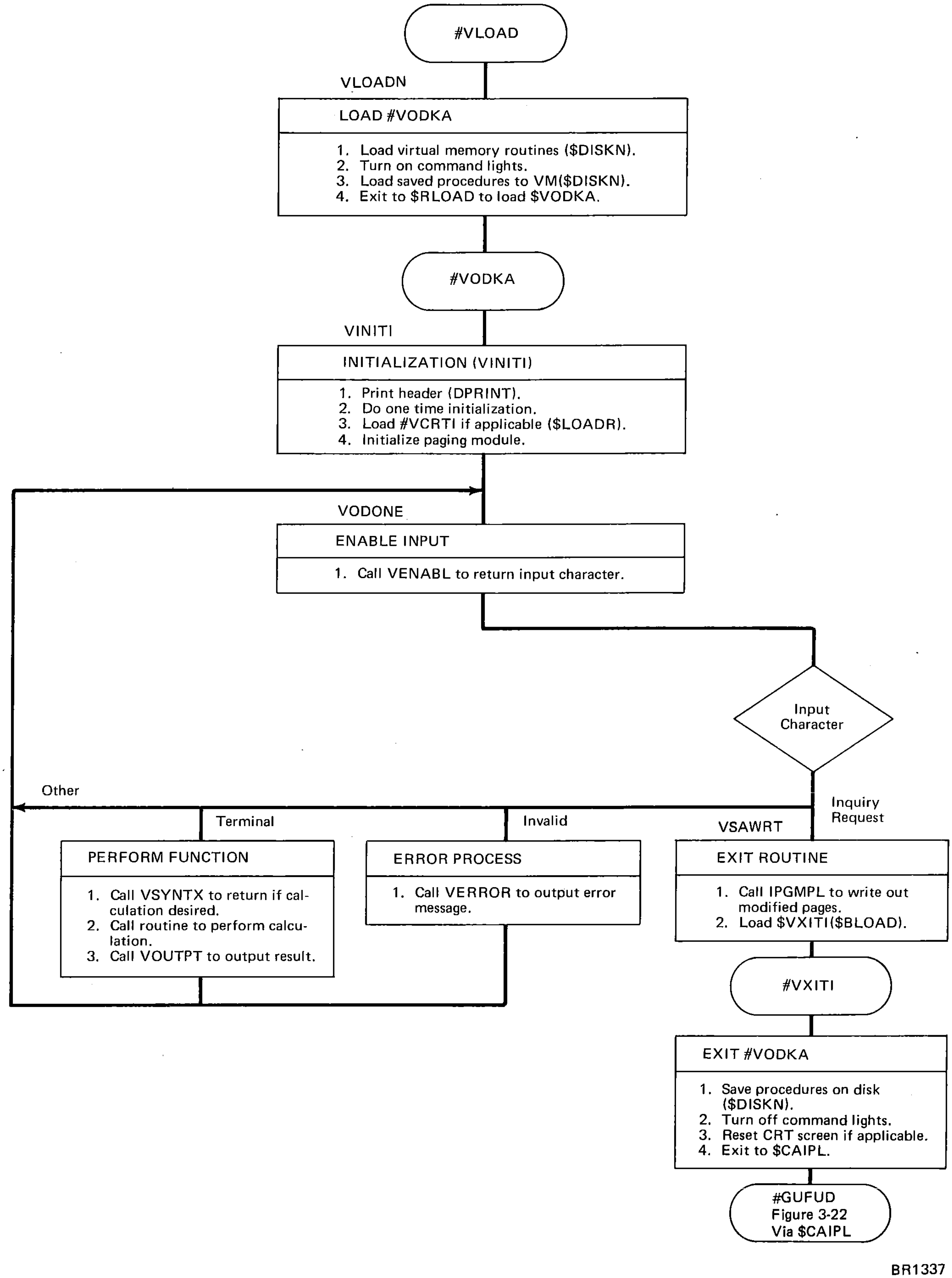


Figure 3-179. Desk Calculator (#VLOAD, #VODKA, #VXITI) Flowchart

Section 4. Directory

This section contains:

- Directory list
- Source module labeling conventions
- System equates

DIRECTORY LIST

The directory is a listing of all components, for quick reference to System/3 BASIC assembly listings on microfiche. The names appear in this directory in the order in which they appear in ##DRTY—system program file directory. This section also contains system equates, and gives the general contents of each equate assembly listing.

The directory list contains the following columns:

- Component Name. The symbolic label used to identify an assembly listing. This name appears on the microfiche and in the heading of each page in the assembly listing.
- Descriptive Component Name. This name identifies the component in Section 3.
- Synopsis. A brief summary of the main functions performed by a program. For components other than programs, the synopsis is a brief summary of the contents of the component.

Component Name	Descriptive Component Name	Synopsis
##0TRK	Cylinder 0, track 0	Contains the IPL bootstrap loader (MLOADS); initial information for track 0.
##1TRK	Cylinder 0, track 1	Contains the system nucleus; initial information for track 1.
##DRTY	System program file directory	Contains a list of all components in System/3 BASIC. Defines the relative disk address and sector count of the component, the core load address, and the program ID number.
#INSTD	Core-resident routines (standard-precision interpreter)	Core-resident routines direct the execution of pseudo machine code (PMC) instructions in standard precision.
#SPSYN	Procedure line checker	Formats procedure lines for insertion into the work file.
#BCOMP	System/3 BASIC language compiler	Compiles the source program into PMC instructions and constants.
#LOADR	Loader	Completes the preparation of virtual memory for the execution of PMC instructions.
#DPRIN	Conversational I/O routines	Provides actual I/O for the matrix printer and keyboard while in conversational mode.

Component Name	Descriptive Component Name	Synopsis
#KGOSL	GO keyword program	Continues or aborts a program from a pause state.
#KEDIT	EDIT keyword program	Edits a saved file to the system work file or prepares the work file for a new file entry.
#KENAB	ENABLE/DISABLE keyword program	Enables or disables statements in the system work file.
#DREAD	Card reader I/O routine	Provides actual I/O for input from the data recorder while in conversational mode.
#KMOUN	MOUNT keyword program	Updates the nucleus communications area when a disk volume is changed.
#KRMOV	REMOVE keyword program	Updates the nucleus communications area when a disk volume is removed.
#KPASW	PASSWORD keyword program	Changes the current password and the password directory to a new password.
#KEXTR	EXTRACT keyword program	Saves specified line numbers on the system work file.
#DPSLY	CRT I/O routine	Provides actual I/O for the CRT while in conversational mode.
#TSYKT	System keyboard tables	Keyboard character tables for various foreign languages.
#KRNUM	RENUMBER keyword program	Renumbers statements in the system work file.
#KROVL	RENUMBER keyword program overlay	See #KRNUM. *Note:* This overlay is assembled with #KOVME.
#KOVME	MERGE keyword program overlay	See #KMERG. *Note:* This overlay is assembled with #KROVL.
#KWRIT	WRITE keyword program	Changes the device assigned as the system printer.
#KREAD	READ keyword program	Changes the system input device.
#KWIDT	WIDTH keyword program	Changes the system printer margin values.
#KRUNI	RUN/STEP/TRACE keyword program	Provides linkage to the System/3 BASIC compiler.
#KDNTE	ENTER keyword program	Enters disk system management programs.
#KMERG	MERGE keyword program	Merges statements from a user file to the system work file.
#TDCKT	Desk calculator keyboard tables	Keyboard tables for various foreign languages.
#KDELE	DELETE keyword program	Deletes statements from the system work file or saved files and passwords from the library.
#KCTLO	LISTCAT keyword program	Displays information from the library directories.
#KLIST	LIST keyword program	Displays the contents of the work file.
#KLOGO	LOGON/OFF keyword program	Defines or cancels a user password.
#KSAVE	SAVE keyword program	Stores the contents of the work file in the two-star library or a user library.

Component Name	Descriptive Component Name	Synopsis
#SPACK	Pack file library subroutine	Condenses the file library to place all null areas at end of the library.
#SPOVL	Second phase of #SPACK	See #SPACK.
#KPOOL	PULL/POOL keyword program	Adds or deletes user files to or from the one-star library.
#KCHAN	CHANGE keyword program	Alters a statement in the system work file or statement containing the last syntax error.
#KSVLA	SAVE keyword program overlay	See #KSAVE.
#KSSPN	SUSPEND keyword program	Saves a program that is currently in an execution pause state.
#KNAME	RENAME keyword program	Changes the filename of the work file or a user file.
#KSYMB	SYMBOLS keyword program	Displays variable names from the system work file.
#KPRTC	PROTECT keyword program	Sets or cancels user file, or one-star protection. Sets two-star protection.
#KSETI	SET keyword program	Assigns a value to a program variable while in a program execution pause state.
#GRAPR	Procedure line fetch processor	Locates sequential procedure statements in the temporary procedure work area.
#KALLO	ALLOCATE keyword program	Reserves space for user data files and defines the data files to be used during program execution.
#KRLAB	RELABEL keyword program	Changes variable names in the system work file.
#KRVLA	RELABEL keyword program overlay	See #KRLAB.
#KDISP	DISPLAY keyword program	Displays the current values of program variables while in a program execution pause state or following a program termination (unless virtual memory is destroyed).
#KDOVR	DISPLAY keyword program overlay	See #KDISP.
#VCRTI	CRT physical IOCS for DCALC	Provides DCALC with an interface to the CRT I/O routine in virtual memory.
#EXMSG	Program interruption processor	Displays a message to identify the type of program interruption.
##CORE	Save area	Disk area used to save core for pause mode.
##ERMS	Error messages	Contains the message number and text for system error messages.
#KHELP	HELP keyword program	Displays help text.
#MIPPE	Nucleus initialization program	Initializes the system nucleus during the IPL procedure.
#KSOVR	SET keyword program overlay	See #KSETI.
#VXITI	DCALC termination	Provides exit linkage to return to BASIC mode of operation.

Component Name	Descriptive Component Name	Synopsis
##VUFA	DCALC subroutines and procedures	Contains user procedure save areas and procedure control subroutines. Resides in virtual memory during DCALC mode of operation.
#VLOAD	DCALC initialization	Initializes and provides linkage to the desk calculator monitor.
#VODKA	DCALC core-resident routines	Controls execution of desk calculator operations.
#TVKBT	Unused	
#VVMRS	DCALC VM-resident subroutines	Contains subroutines that reside in virtual memory during DCALC mode of operation.
#FMSTD	VM-resident execution subroutines	Contains all standard precision execution subroutines that occupy the fixed area of virtual memory during the execution of a BASIC program.
#UEXLI	EXPAND utility program	Changes the disk space allocated to a user library file.
#UALLO	ASSIGN utility program	Allocates disk space for a LIBRARY or system work area.
#KCNDI	CONDITION keyword program	Displays the current status of the system.
##CSAV	Suspended save area	Disk area used to save suspended core.
##SSAV	Status save area	Disk area used to save the status of a suspended program.
##SAVM	Save area	Virtual memory disk save area.
#FISTD	Interpreter execution overlay	Overlays the core-resident interpreter to perform matrix inversion or determinant in standard precision.
#FILNG	Interpreter execution overlay	Overlays the core-resident interpreter to perform matrix inversion or determinant in long precision.
##RSPG	Save area	Start of cylinder 4, R1 area.
#BOVLY	Statement processor overlays	Contains overlays to generate PMC sequences and to terminate the compiler.
#SFSYN	BASIC statement syntax checker	Checks the syntax of all System/3 BASIC language statements entered into the system.
#SFOVR	Syntax checker	See #SFSYN.
#STROV	Third phase of BASIC statement syntax checker	Checks operands of STR function.
##FSPG	Save area	Start of cylinder 4, F1 area.
#GUFUD	Work file update/crusher	Performs maintenance on the system work file and monitors system input while in conversational mode.
#ERRPG	Error message program	Displays error messages.
##BLNB	Bad line buffer	Used to store the input line buffer when the line is invalid. Its purpose is to free the input line buffer for input required to correct the bad line.
#ECMAN	Command analyzer	Analyzes system commands and loads the program required to process that command.

Component Name	Descriptive Component Name	Synopsis
#SFLOA	Interpreter execution overlay	Logical IOCS for disk data files. Transfers data items between saved files in the file library and I/O buffers allocated in virtual memory.
#SDSYN	Data syntax checker	Checks the syntax of all data entered into the system while in conversational mode.
#SFFIN	Interpreter execution overlay	Find disk data file subroutine. Called to find a disk data file in the file library or to get space for a SCRATCH data file.
#UPACK	PACK utility program	Provides interface to the pack user library subroutine. See #SPACK.
#EFKEY	Command key processor	Processes command keys 01 through 11.
#UCNFI	CONFIGURE utility program	Creates or modifies the configuration record on disk.
#UCPLI	COPY file utility overlay	Copies the system program, user library, or help text files.
#UATRC	Alternate track utility program	Tests, assigns, and unassigns individual data tracks.
#UINIT	Initialize disk utility program	Initializes a disk volume to the standard System/3 format.
#UCDIS	COPY volume utility program	Copies the contents of a disk volume to another volume.
#UDELV	VTOC delete utility program	Releases disk space by deleting files defined by labels in the VTOC.
#UDISV	VTOC display utility program	Displays VTOC file label information.
#ZTRAC	Program load trace overlay	When activated, traces all programs loaded by the system nucleus.
#ZDUMP	Virtual memory dump overlay	Displays the pseudo instructions currently in virtual memory.
#ZLOAD	Maintenance utility loader	Loads the maintenance utility monitor.
#ZUTMO	Maintenance utility monitor	Provides maintenance service aid functions.
#INLNG	Long precision interpreter	Core-resident routines. Directs the execution of pseudo machine code (PMC) instructions in long precision.
#KCALL	CALL keyword program	Invokes a procedure file in the user library.
#KRSUM	RESUME keyword program	Restores a suspended program to the execution pause state.
#UPTFI	PTF utility program	Applies program temporary fixes to System/3 BASIC components.
#UPOVL	PTF utility program overlay	See #UPTFI.
#FMLNG	VM-resident execution subroutines	Contain all long-precision execution subroutines that occupy the fixed area of virtual memory during the execution of a BASIC program.
##CNFI	Configurator record	Contains indicators for the hardware components with which BASIC is running.
#KLLAY	LIST keyword program overlay	See #KLIST.
#ZLBMA	Library map overlay	Determines the library map option.

Component Name	Descriptive Component Name	Synopsis
#ZL1MA	Library map option 1 overlay	Processes option 1.
#ZL2MA	Library map option 2 overlay	Processes option 2.
#ZL3MA	Library map option 3 overlay	Processes option 3 (part 1).
#ZLVRL	Library map option 3 overlay	Processes option 3 (part 2).
#KKEYS	KEYS keyword program	Lists, assigns, or restores command key functions.
##CKTB	Command key table	Table of functions assigned to command keys 1 to 11.
##INVD	Save area	The matrix inverse and matrix determinant functions use this disk area.
##PWRK	Procedure work area	This disk work area is used when a procedure file is invoked.
#TEQU1	System equates	Assembled as an aid to resolving symbolic references in microfiche (part 1).
#TEQU2	System equates	Assembled as an aid to resolving symbolic references in microfiche (part 2).

SOURCE MODULE LABELING CONVENTIONS

All labels within the same source module are prefixed by a character that identifies the type of source module. The following list associates each character with the type of source module it identifies:

B–Compiler
C–Conversion subroutines
D–I/O subroutines
E–System control routines
F–Virtual memory routines
G–Work file processing
I–Interpreter
K–Keyword processing
L–Loader
M–Miscellaneous system routines
N–System nucleus
S–Data management or syntax routines
T–Nonexecutable tables
U–Utility processing
V–Desk calculator
Z–Maintenance utilities
@–System equates
$–System equates

SYSTEM EQUATES

The component parts of System/3 BASIC are assembled in the System/3 Basic Assembler Language. Modules composed of equates (EQU) are used for communication between the different component parts (assembly listings). These equate modules are referred to as system equates. These modules are assembled (as needed) with the component parts of the system. To reduce the size of listings and microfiche, the PRINT OFF, PRINT ON feature of the assembler is used. The value and references of equated labels that are not printed can be found in the cross-reference label list in the assembly listing.

All system equate modules have been grouped into two assemblies, #TEQU1 and #TEQU2. All labels not printed at their point of resolution can be located in one of these listings.

#TEQU1

System and Hardware Equates—@SYSEQ

Note: All labels in this module are prefixed by @.

1. CPU equates: registers, instruction lengths and displacements, branch condition codes, miscellaneous constants, and masks.
2. Disk parameter list (DPL) and print parameter list (PPL) equates.
3. System work file equates: segment header displacements and masks, and file index table (FIT) displacements and lengths.

System Hardware I/O Equates—@HDWEQ

Note: All labels in this module are prefixed by @.

1. Disk equates: disk control field (DCF), disk I/O instructions, condition codes, device addresses, track flag byte, nucleus communications area error history log entries, and sense bytes.
2. Matrix printer equates.
3. Keyboard equates: mask values and command keys.
4. CRT equates.
5. Data recorder equates: read and punch I/O instructions, device addresses, error codes, and PPL function code masks.

Fixed Addresses for System Nucleus—@FXDEQ

Note: All labels in this module are prefixed by $.

1. Entries to nucleus interface routines: maintenance utility aids, physical disk I/O, and error logging.
2. Nucleus communications area equates (Figure 5-1).
3. Entries to nucleus resident routines and their work areas.
4. Equates to develop the nucleus end address.

Common Core Locations Outside Nucleus—@CANEQ

Note: All labels in this module are prefixed by $$.

1. Displacements to fields in input line statements (header and text).
2. Entry points, masks, switches, and fields in the keyboard and matrix printer I/O routines (#DPRIN: DEPRES and DPRINT).
3. Entry points to the card reader I/O routine (#DREAD).
4. Entry points and locations in the CRT I/O routine (#DSPLY).
5. Miscellaneous locations.
6. Keyword program load addresses.

Cylinder Zero Equates—@CY0EQ

Note: All labels in this module are prefixed by #.

1. Volume ID equates.
2. SDR/OBR displacements and lengths.
3. Cylinder 0 disk addresses and sector counts.

System Program Area Equates for Relative Disk Addresses and Sector Counts—@SPFEQ

Note: All labels in this module are prefixed by #$, all sector counts are prefixed by #$@, and all core load addresses are prefixed by #$$.

1. Relative disk addresses, sector count, and core load address of all programs, error message modules, keyboard tables, etc., contained in the system program file.

System Work Area Equates for Physical Disk Addresses and Sector Counts—@WKAEQ

Note: All labels in this module are prefixed by #@ and all sector counts are prefixed by #@@.

1. Cylinder 4: selected system programs, bad-line buffer, and I/O record (file directory 1).
2. Cylinders 5 and 6: file index table (FIT) and work file data area.
3. Cylinders 7, 8, and 9: virtual memory.
4. Cylinder 9: temporary work area, core save area, and compiler and interpreter tables on disk.

File Library Addresses and Tables—@DIREQ

Note: All labels in this module are prefixed by ##.

1. Labeling method description.
2. Relative disk addresses, displacements, lengths, and masks for the file library directories.

General Error Message Equates—@ERMEQ, @SEREQ

Note: All labels in this module are prefixed by @@.

1. Equates to system message numbers. ##ERMS contains the message numbers and associated message text.

Volume Label Equates—@VOLEQ

Note: All labels in this module are prefixed by $#T.

1. Displacements to fields in the volume label.
2. Mask values for the files indicator byte.

Volume Table of Contents (VTOC) Equates—@VTCEQ

Note: All labels in this module are prefixed by $@$.

1. Displacements to fields in the VTOC index.
2. Displacements to fields in the VTOC system file labels.

System Configuration Record Equates—@CNFEQ

Note: All labels in this module are prefixed by @#.

1. Component number, displacement factor, and masks for all system I/O devices.
2. Displacement factor and masks for disk size, disk drive configuration, and core size.

Virtual Memory Directory Equates; Directory 1 and Directory 2—@VMDEQ

Note: All labels in this module are prefixed by @$.

1. Labeling method description.
2. Displacements, field lengths, and masks for directory 1 and directory 2.

Halt Indicator Equates—@HLTEQ

Note: All labels in this module are prefixed by @H.

1. All values used in HPL instructions to display halt codes.

Compiler Fixed Equates—BEQU

Note: All labels in this module are prefixed by B$.

1. Addresses of buffers used for disk I/O.
2. PMC generator entry points.
3. Core-resident routine entry points and parameter addresses.
4. Tables, subroutine precision areas, and miscellaneous equates.
5. Common compiler switch locations and masks.

Compiler System Equates—$B@EQU

Note: All labels in this module are prefixed by B@.

1. B@C—Pseudo instruction op codes.
2. B@L—Pseudo instruction lengths.
3. B@B—Condition code values for the BRC pseudo instruction.
4. B@P—Execution control code values for PRS and PRU pseudo instructions.
5. B@T—BASIC statement type codes.
6. B@L—BASIC statement keyword lengths.
7. B@D—Disk addresses of PMC generators, system work file, virtual memory, statement address table, and branch address table.
8. Special Characters.
9. B@LET—Alphabetic characters.
10. B@DEC—Numeric characters.
11. Miscellaneous equates for constants, masks, lengths, function and array table elements, etc.
12. Equates for virtual-memory allocation.
13. Length and displacements in the loader parameter area.

Interpreter Fixed Equates—IEQU

Note: All labels in this module are prefixed by I$.

1. Fixed core region addresses.
2. Core-resident routine entry points and parameter addresses.
3. Indicator masks.

Fixed Addresses in Virtual Memory—VEQU

Note: All labels in this module are prefixed by V$.

1. V$F—Intrinsic functions.
2. V$A—Arithmetic functions.
3. V$M—Matrix assignment functions.
4. V$X—I/O interfaces.
5. V$S—System I/O routines.
6. V$C—Conversion routines.
7. V$D—Execution-time diagnostic routines.
8. V$V—Interpreter utility routines.
9. V$K—Keyboard IOCS character tables.
10. Virtual memory subroutine directory containing virtual addresses, disk addresses, symbolic labels, and functional descriptions of all entry points in virtual memory execution subroutines.

Desk Calculator Equates—@V@EQU

Note: All labels in this module are prefixed by V@.

1. Miscellaneous equates.
2. Mode indicators.
3. Displacements for the procedure table and input table, and from the first byte of the register.
4. Masks and lengths.
5. Output indicator masks.
6. Error codes.
7. Keyboard keys.
8. Keyboard data byte masks for selected keys.

Long Precision Execution Equates—$I@LEQ

Note: All labels in this module are prefixed by I@.

1. Data element equates.
2. Arithmetic function reference equates.
3. Pseudo instruction and stack element displacements.
4. Core pages and miscellaneous equates.

System Level Equates—@LVLEQ

Note: All labels in this module are prefixed by @.

System level number.

#TEQU2

Standard Precision Execution Equates—$I@SEQ

Note: All labels in this module are prefixed by I@.

1. Data element equates.
2. Arithmetic function reference equates.
3. Pseudo instruction and stack element displacements.
4. Core pages and miscellaneous equates.

Section 5. Data Area Formats

This section contains detailed information concerning these areas of System/3 BASIC:

- System communication area (NUCLES) (Figure 5-1)
- Disk volume format (Figure 5-2)
- Configuration record (Figure 5-3)
- Error history log (Figure 5-4)
- Individual volume statistics and master SIO table (Figure 5-5)
- Disk statistical data recording (Figure 5-6)
- Nondisk statistical data recording (Figure 5-7)
- Outboard recording (Figure 5-8)
- Volume label (Figure 5-9)
- Volume table of contents (Figure 5-10)
- Directories to system library file (Figure 5-11)
- Null directory (Figure 5-12)
- Password directory (Figure 5-13)
- Filename directory block (Figure 5-14)
- BASIC program file structure (Figure 5-15)
- File index table (Figure 5-16)
- File directory 1 (Figure 5-17)
- Segment descriptor field (Figure 5-18)
- End of file record (Figure 5-19)
- File directory 2 (Figure 5-20)
- System help text file (Figure 5-21)
- Help text record (Figure 5-22)
- Print parameter list (Figure 5-23)
- Disk parameter list (Figure 5-24)
- Disk control field (Figure 5-25)
- Delete parameter list (Figure 5-26)
- Command key table (Figure 5-27)
- IBM-assigned command key functions (Figure 5-28)
- System program file directory – ##DRTY (Figure 5-29)

System Equate	Hex Disp from Label $NUCBS	Dec Disp	Field Length	Mask	Description
$RMRGN	00	0	1		Right margin value for printer.
$LMRGN	01	1	1		Left margin value for printer.
$PRPOS	02	2	1		Current position of printer head.
$KEYCD	03	3	1		Keyboard indicators:
$TRUNK				X'80'	Last line truncated (keyboard input).
$DTNMB				X'40'	Automatic line numbering (card NUM).
$INRPT				X'20'	Program interrupted and aborted.
$KYBSY				X'10'	Keyboard busy (line not yet complete).
$GUFIR				X'08'	#GUFUD interrupted but not aborted.
$NOLST				X'04'	No listing of card input required.
$IOYES				X'02'	I/O routines are in core.
$CARDI				X'01'	Input from data recorder (bit off indicates keyboard input).
$BRSAV	04	4	2		Base register save area.
$XRSAV	06	6	2		Index register save area.
$TABLN	08	8	4		Automatic line number value (inserted if tab key is first key depressed).
	0C	12	1	X'40'	Blank must follow $TABLN.
$CAERR	0D	13	1		Error code for interface to #ERRPG.
$ERRPG	0E	14	1		Indicators for special functions of #ERRPG:
$ERKEY				X'80'	Standard error (set by command analyzer #ECMAN).
$ER1N2				X'50'	Level 1 and 2 messages required.
$ERFIL				X'40'	File line error has occurred.
$ERSFL				X'35'	File line error occurred in syntax checkers.
$ERSTK				X'30'	Process stacked error codes.
$ERRCT	0F	15	1		Count of stacked error codes.
$XIND1	10	16	1		Primary execution indicators:
$VMDEF				X'80'	Virtual Memory not empty.
$XPREC				X'40'	Execute in long precision (bit off means short precision).
$TRVAR				X'20'	Trace selected variables. } When X'04' is on, at least one of these must be on.
$TRALL				X'10'	Trace all.
$TFLOW				X'08'	Trace flow.
$TRACE				X'04'	Execute in trace mode. } Mutually exclusive.
$STEPT				X'02'	Execute in step mode.
$RUNIT				X'01'	Execute in run mode.
$XIND2	11	17	1		Secondary execution indicators:
				X'E0'	Unused bits.
$ABORT				X'10'	Abort execution.
$PSTMT				X'08'	Pause caused by PAUSE statement.
$PSTEP				X'04'	Pause caused by step mode.
$PAUSE				X'02'	Program in pause state.
$EXCMD				X'01'	Program in execution.

BR1342.1

Figure 5-1. System Communication Area (NUCLES) (Part 1of 4)

System Equate	Hex Disp from Label $NUCBS	Dec Disp	Field Length	Mask	Description
$IOIND	12	18	1		I/O status indicators:
$LNPTR				X'80'	Bidirectional printer option available.
$DTRDR				X'40'	Data recorder present.
$HRDER				X'20'	Hard error.
$PGMST				X'10'	Program start key not used for automatic line number.
$CMDKY				X'08'	Command keys only (bit off for full keyboard input).
$CRTNO				X'04'	CRT can be used for system printer.
$CRTAV				X'02'	CRT present.
$MPDWN				X'01'	Matrix printer is not operational.
$CRTIN	13	19	1		CRT command indicators:
				X'F0'	Unused bits.
$CRTSP				X'08'	Roll stop requested.
$CRTPU				X'04'	Pop requested.
$CRTDN				X'02'	CRT in rolldown mode.
$CRTUP				X'01'	CRT in rollup mode.
$INDR1	14	20	1		System work file status indicators:
$BASIC				X'80'	Basic program in work file.
$KEYDT				X'40'	Keyboard- or card-generated data file in work area.
$PGMDT				X'20'	Program-generated data file in work area.
$FITIN				X'10'	FIT sectors are in core.
$WFLOK				X'08'	File protected (only ALLOCATE can modify file).
$WSIND				X'04'	System work file contains an active file.
$PRESN				X'02'	Long precision in use (bit off means short precision).
$PROCI				X'01'	Work file procedure indicator.
$INDR2	15	21	1		System indicators:
$READY				X'80'	READY will not be printed.
$FDIND				X'40'	Line number list is deleted.
$FUIND				X'20'	Line passed.
$FCIND				X'10'	Single line number deletion, through the command analyzer (#ECMAN).
$DKERR				X'08'	Disk error has occurred (an entry must be made in the individual volume statistics).
$ERPND				X'04'	Error is pending for history log.
$CMODE				X'02'	Conversational mode (bit off means utility mode).
$TMPUT				X'01'	In temporary utility mode.
$INDR3	16	22	1		System indicators:
$NWRKF				X'80'	No work area on F1.
$NWRKR				X'40'	No work area on R1.
$MOUNT				X'20'	Only MOUNT or INITIALIZE command is valid after REMOVE command.
$CLBFR				X'10'	Clear input line buffer.
$NOENB				X'08'	Keyboard already enabled.
$ERHRD				X'04'	Hard halt from #ERRPG.
$LIST				X'02'	Accept rolldown key.
$DBLOK				X'01'	File may be saved to ** library.

BR1342.2A

Figure 5-1. System Communication Area (NUCLES) (Part 2 of 4)

System Equate	Hex Disp from Label $NUCBS	Dec Disp	Field Length	Mask	Description
$DKSIZ	17	23	1		Total disk cylinders on system:
				X'E0'	Unused bits.
$DK800				X'10'	800 cylinders.
$DK600				X'08'	600 cylinders.
$DK400				X'04'	400 cylinders.
$DK200				X'02'	200 cylinders.
$DK100				X'01'	Reserved.
$XIND3	18	24	1		Previous contents of $XIND1 (displacement X'10') used by loader to determine such things as the precision of VM routines.
$FILIB	19	25	2		Current file library disk address.
$USRDR	1B	27	2		Displacement to the first user directory block for the LOGON password.
$CONFG	1D	29	1		Configuration indicators:
$16CKY				X'08'	16 command keys present.
$12K				X'04'	Storage size is 12k.
$16K				X'02'	Storage size is 16k.
$22IMP				X'01'	22-inch matrix printer (bit off means 13-inch).
$BIGCD				X'80'	129 Card Data Recorder configured.
				X'70'	Unused bits.
$LEVEL	1E	30	2		System level number.
$DBGUF	20	32	1		#GUFUD indicators:
$CRUSH				X'80'	Crush the work file if bit is off.
$REORD				X'40'	Reorder the work file if bit is off.
$IRKEY				X'20'	Force return to keyboard mode.
$IOPGS				X'10'	File directory 1 occupies 2 sectors. If only one sector is used, this bit is off.
$CALLI				X'08'	Procedure call indicator.
				X'07'	Unused bits.
$KEYBD	21	33	1		Number associated with the keyboard table being used.
$CRPOS	22	34	1		Current position of the CRT cursor.
$BUFPT	23	35	1		Line printer buffer pointer.
$LPRP3	24	36	1		Line printer indicators.
$LPROS	25	37	1		Line printer print position.
$NEXTB	26	38	1		Relative sector address of next line in procedure call.
$NEXTL	27	39	1		Displacement within relative sector for next procedure line.
$DFDET	28	40	1		Internal procedure line fetch indicator.
$LPRIO	2A	42	2		Save area for line printer.
$PTCH1	2B	43	11		Patch area.

BR1342.3B

Figure 5-1 System Communication Area (NUCLES) (Part 3 of 4).

System Equate	Hex Disp from Label $NUCBS	Dec Disp	Field Length	Mask	Description
$VOLID	36	54	32		Volume-ID table. If volume is not mounted, its entire entry is binary 0's. If no file library is present on the volume, the first byte of the disk address is X'00'.
$VOLR1	36	54	6		Volume-ID for R1.
	3C	60	2		File library disk address on R1.
$VOLF1	3E	62	6		Volume-ID for F1.
	44	68	2		File library disk address on F1.
$VOLR2	46	70	6		Volume-ID for R2.
	4C	76	2		File library disk address on R2.
$VOLF2	4E	78	6		Volume-ID for F2.
	54	84	2		File library disk address on F2.
$PKERT	56	86	16		Disk volume error rate table.
	56	86	2		Total write errors on R1.
	58	88	2		Total read errors on R1.
	5A	90	2		Total write errors on F1.
	5C	92	2		Total read errors on F1.
	5E	94	2		Total write errors on R2.
	60	96	2		Total read errors on R2.
	62	98	2		Total write errors on F2.
	64	100	2		Total read errors on F2.
$PASWD	66	102	8		Current password.
$HISTE	6E	110	10		Error history log entry.
	6E	110	1		SIO instruction Q code.
	6F	111	1		SIO instruction R code.
	70	112	4		Sense bytes.
	74	116	1		Count.
$HIST1	75	117	3		Last 3 bytes of DCF (Figure 5-25).
$DATE	7A	120	3		IPL date.
$EXFTR	7B	123	1		Core expansion factor for over 8k.
$WFNME	7C	124	8		Work file name.
$WFDEF	7C	124	1	X'40'	Indicates the work file is defined.
$DPLSV	84	132	6		DPL save area for keyword programs.
$PRDEV	8A	138	2		Core address of the system printer IOCR.
$CRTAD	8C	140	2		Core address of entry to relocate CRT.
$PLST1	8E	142	7		Last I/O parameter list started.
$PLST2	95	149	7		Second to last parameter list started.
$PLST3	9C	156	7		Third to last parameter list started.
$C0001	A3	163	2		Constant of X'0001'.

BR1342.4

Figure 5-1. System Communication Area (NUCLES) (Part 4 of 4)

Fixed Areas and System Files	CC HH SS	N	Notes
IPL bootstrap loader (#MLOAD)	00 00 00	1	Present on all volumes.
System configuration record	00 00 01	1	Present on all volumes.*
Volume label	00 00 02	1	Required on all volumes.
Error history log	00 00 03	6	Required on all volumes.
VTOC index	00 00 09	2	Required on all volumes.
VTOC file labels	00 00 11	13	Required on all volumes.
System nucleus	00 01 00	12	Present on all volumes.*
IBM program product protection	00 01 12	3	Present on all volumes.*
Disk system management program IPL	00 01 15	8	Present on all volumes.*
PTF Log	00 01 23	1	Present on all volumes.
Alternate data tracks	01 00 00 through 03 01 23		Six tracks present on all volumes.
System work file	04 00 00 through 09 01 23		Twelve tracks required on both R1 and F1 (24 tracks total).
System program file	nn 00 00 through nn 01 23	x	Location defined by user. Must be defined on drive 1.
System library file	nn 00 00 through nn 01 23	n	Location and size defined by user.
System help text file	nn 00 00 through nn 01 23	n	Location defined by user.
System PTF file	xx 00 00 through xx 01 23	x	

Notes:

x—Predefined values.

n—Values that can be defined by the user. These values are defined in the volume label and by labels in the VTOC.

CC HH SS N—Cylinder, head, sector, and number of sectors.

*—Space reserved but not necessarily used.

Figure 5-2. Disk Volume Format

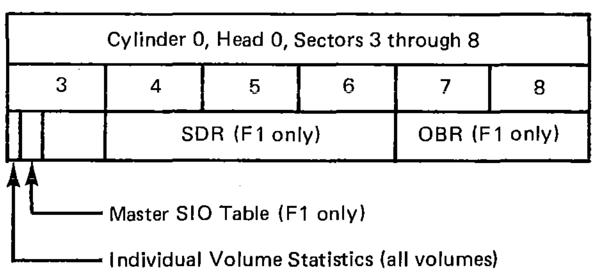
Hex Disp	Dec Disp	Dec Length	Mask (bits on)	Description
00	0	16		Reserved.
10	16	1		Disk:
			X'80'	Supported.
			X'40'	Supported for System/3 BASIC.
			X'3F'	Unused.
11	17	2		Unused.
13	19	1		Disk size and configuration:
			X'04'	Model 1; 2 volumes of 100 cylinders each.
			X'08'	Model 2; 2 volumes of 200 cylinders each.
			X'18'	Model 2 and 3; 3 volumes of 200 cylinders each (F1, R1, and F2).
			X'09'	Two Model 2's; 4 volumes of 200 cylinders each (maximum configuration).
			X'E2'	Unused.
14	20	1		Printer:
			X'80'	Supported.
			X'40'	Supported for System/3 BASIC.
			X'3F'	Unused.
15	21	1		Unused.
16	22	1		Model indicators:
			X'09'	5213 Model 1 or 2; 132 print positions.
			X'05'	5213 Model 3; 132 print positions, bidirectional.
			X'0A'	2222 Model 1; 220 print positions.
			X'06'	2222 Model 2; 220 print positions, bidirectional.
17	23	1		Unused.
18	24	1		Keyboard:
			X'80'	Supported.
			X'40'	Supported for System/3 BASIC.
			X'3F'	Unused.
19	25	1		Keyboard options:
			X'80'	16 command keys.
			X'40'	8 command keys.
			X'3F'	Unused.
1A	26	1		Keyboard character table. Contains a number that corresponds to the keyboard table selected.
1B	27	1		Unused.
1C	28	4		Reserved.
20	32	1		Data Recorder
			X'80'	Supported
			X'40'	5496-supported for System/3 BASIC
			X'48'	129-supported for System/3 BASIC
			X'37'	Unused.
21	33	3		Unused.

Figure 5-3. Configuration Record (Part 1 of 2)

Hex Disp	Dec Disp	Dec Length	Mask (bits on)	Description
24	36	4		Reserved.
28	40	1		2265 Display Station (CRT):
			X'80'	Supported.
			X'40'	Supported for System/3 BASIC.
			X'3F'	Unused.
29	41	3		Unused.
2C	44	16		Reserved.
3C	60	1		Unused.
3D	61	1		Core size:
			X'01'	8k.
			X'02'	12k.
			X'04'	16k.
			X'F8'	Unused.
3E	62	2		Unused.
40	64	192		Reserved.
FF	255			Last byte of configuration record.

Note: Four-byte entries are reserved for each possible System/3 component. This figure illustrates only those areas used by System/3 BASIC.

Figure 5-3. Configuration Record (Part 2 of 2)



Notes:

Refer to Figure 5-5 for individual volume statistics and master SIO table.

Refer to Figure 5-6 for disk statistical data recording (SDR).

Refer to Figure 5-7 for nondisk statistical data recording (SDR).

Refer to Figure 5-8 for outboard recording (OBR).

Figure 5-4. Error History Log

Individual Volume Statistics (present on all volumes)			
Hex Disp	Dec Disp	Dec Length	Description
00	0	4	Count of total temporary errors (includes missing address markers and data checks).
04	4	4	Total write SIO's issued to this volume (includes verifies).
08	8	4	Total read and scan SIO's issued to this volume.
Master SIO Table (F1 only)			
0C	12	4	Total write SIO's issued to R1.
10	16	4	Total read and scan SIO's issued to R1.
14	20	4	Total write SIO's issued to F1.
18	24	4	Total read and scan SIO's issued to F1.
1C	28	4	Total write SIO's issued to R2.
20	32	4	Total read and scan SIO's issued to R2.
24	36	4	Total write SIO's issued to F2.
28	40	4	Total read and scan SIO's issued to F2.
2C	44	212	Unused to end of sector 3.

BR1346

Figure 5-5. Individual Volume Statistics and Master SIO Table (Cylinder 0, Head 0, Sector 3)

Disk Error Counters (2 bytes each)								
Hexadecimal Displacement*								Error Condition
R1		F1		R2		F2		
T	P	T	P	T	P	T	P	
00	10	20	30	40	50	60	70	Overrun
02	12	22	32	42	52	62	72	Data check in ID
04	14	24	34	44	54	64	74	Data check on write
06	16	26	36	46	56	66	76	Data check on read
08	18	28	38	48	58	68	78	No record found
0A	1A	2A	3A	4A	5A	6A	7A	Equipment check
0C	1C	2C	3C	4C	5C	6C	7C	Missing address marker
0E	1E	2E	3E	4E	5E	6E	7E	Seek check

* T—temporary; P—permanent

Notes:

1. Statistical data recording (SDR) is present on F1 only.
2. The remainder of sector 4 and all of sector 5 are not used for error recording.

BR1347

Figure 5-6. Disk Statistical Data Recording (Cylinder 0, Head 0, Sector 4)

Nondisk Error Counters (2 bytes each)			
Hex Disp	Dec Disp	Dec Length	Error Condition
00	0	2	Keyboard parity check
02	2	6	Unused
08	8	2	CRT parity check
0A	10	6	Unused
10	16	2	Printer horizontal cycle check (temporary)
12	18	2	Printer data check (temporary)
14	20	2	Printer margin check
16	22	2	Printer sync check (temporary)
18	24	2	Printer ROS check (temporary)
1A	26	2	Printer vertical cycle check
1C	28	2	Printer horizontal cycle check (permanent)
1E	30	2	Printer data check (permanent)
20	32	2	Unused
22	34	2	Printer sync check (permanent)
24	36	2	Printer ROS check (permanent)
26	38	26	Unused
40	64	2	Data recorder not ready
42	66	2	Unused
44	68	2	Data recorder compare error
46	70	186	Unused to end of sector 6
Note: Statistical data recording (SDR) is present on F1 only.			

BR1348

Figure 5-7. Nondisk Statistical Data Recording (Cylinder 0, Head 0, Sector 6)

Outboard Recording (2 sectors on F1 only)			
Hex Disp	Dec Disp	Dec Length	Description
00	0	2	Displacement to the last byte of the previous OBR entry.
02	2	2	Displacement to the last byte of the OBR table (always X'01FF').
04	4	4	Unused.
08	8	504	OBR entries (either 8 or 16 bytes in length).

Disk OBR Entry (16 bytes)			
Hex Disp	Dec Disp	Dec Length	Description
00	0	2	Q and R bytes from SIO instruction.
02	2	4	Sense bytes.
06	6	1	Retry count for temporary errors (X'00' indicates a permanent error).
07	7	2	Disk address from disk control field (DCF).
09	9	1	Number of sectors from DCF.
0A	10	6	Volume-ID from volume label.

Nondisk OBR Entry (8 bytes)			
Hex Disp	Dec Disp	Dec Length	Description
00	0	2	Q and R bytes from SIO instruction.
02	2	1	Second sense byte.
03	3	1	First sense byte.
04	4	2	Device dependent information.
06	6	2	Unused.

Device Dependent Information (displacement X'04' in nondisk OBR entries)	
Device	Content
CRT	CRT address register.
Printer	One-byte command code followed by one-byte count.
Keyboard	Undefined.
Data Recorder	Undefined.

BR1349A

Figure 5-8. Outboard Recording (Cylinder 0, Head 0, Sectors 7 and 8)

Hex Disp	Dec Disp	Dec Length	Field Name	Field Description
00	0	3	Label identifier	Must contain VOL.
03	3	6	Volume-ID	Six alphanumeric characters that provide unique identification for the volume.
09	9	2	VTOC pointer	Disk address of the first sector in the volume table of contents.
0B	11	71	Reserved	
52	82	10	Owner ID	Ten alphanumeric characters optionally set by the user when the volume is initialized.
5C	92	1	Volume size	Number of cylinders initialized.
5D	93	1		Number of tracks per cylinder.
5E	94	1		Number of sectors per track.
5F	95	2		Number of bytes per sector.
61	97	8	Reserved	
69	105	1	CE cylinder status	X'F0'—Track 0 on the CE cylinder is defective. X'0F'—Track 1 on the CE cylinder is defective. X'FF'—Both tracks on the CE cylinder are defective. X'00'—Both tracks on the CE cylinder are operative.
6A	106	12	Alternate track assignments	One 2-byte entry per alternate track containing the disk address of the defective track. The entry contains X'0000' if the alternate is unassigned.
76	118	51	Track usage mask	Contains a mask of bits in a one-to-one correspondence with each track on the volume. If the bit is on, the track is assigned to a system file. If the bit is off, the track is available. Cylinders 0 through 3 correspond to displacement A8.
A9	169	46	Reserved	
D7	215	1	Work area release level	The system release level of the work area.
D8	216	24	Suspected defective tracks	Twelve 2-byte entries for disk addresses of primary data tracks suspected of being defective. Unused entries contain X'FFFF'.

BR1350.1B

Figure 5-9. Volume Label (Part 1 of 2)

Hex Disp	Dec Disp	Dec Length	Field Name	Field Description
F0	240	1	Help VTOC tag	Entry in the VTOC index for the system help text file.
F1	241	2	Help disk address	Disk address of the first sector allocated to the system help text file.
F3	243	1	PTF VTOC tag	Entry in the VTOC index for the system PTF file.
F4	244	1	PTF file size	Number of cylinders allocated to the system PTF file.
F5	245	2	PTF disk address	Disk address of the first sector allocated to the system PTF file.
F7	247	1	Library file size	Number of cylinders allocated to the system library file.
F8	248	1	Library VTOC tag	Entry in the VTOC index for the system library file.
F9	249	1	Work file VTOC tag	Entry in the VTOC index for the system work file.
FA	250	1	Program VTOC tag	Entry in the VTOC index for the system program file.
FB	251	2	Program disk address	Disk address of the first sector allocated to the system program file.
FD	253	2	Library disk address	Disk address of the first sector allocated to the system library file.
FF	255	1	System files indicator	If the bit is on, the corresponding system file is allocated on this volume: X'80'—System program file. X'40'—System work file (R1). X'20'—System work file (F1). X'10'—System library file. X'08'—System PTF file. X'04'—System help text file. X'03'—Unused bits.

Note: System file disk address in the volume label are all in the form X'nn00', where nn is the first allocated cylinder number. Sector and head are always zero and the drive is always set for R1.

BR1350A.2A

Figure 5-9. Volume Label (Part 2 of 2)

Sector	Content
9	Index
10	
11–23	50 File Labels (4 per sector)

Note: The last 128 bytes of sector 23 are unused.

VTOC Index (512 bytes)		
Hex Disp	Length	Content
00	6	Unused
06	500	50 index tags
1FA	5	Reserved
1FF	1	Total unused index tags

Index Tag (10 bytes)		
Hex Disp	Length	Content
00	8	System file name
08	1	File label sector address
09	1	Relative displacement within sector to last byte of file label.

File Label (64 bytes)		
Hex Disp	Length	Content
00	1	File label ID number
01	2	Reserved
03	8	System file name
0B	7	Reserved
12	1	File type; X'00' for all System/3 BASIC files
13	12	Reserved
1F	2	Starting disk address
21	2	Ending disk address
23	29	Reserved

BR1351

Figure 5-10. Volume Table of Contents (VTOC)

Directories to System Library File (first 7 sectors)						
Null Directory	Password Directory				Filename (**) Directory Block	
0	1	2	3	4	5	6

Notes:

1. Refer to Figure 5-12 for null directory format.
2. Refer to Figure 5-13 for password directory format.
3. Refer to Figure 5-14 for filename directory block format.
4. The first two filename directory blocks, immediately following the password directory, are always the first **directory block followed by the first * directory block.

BR1352

Figure 5-11. Directories to the System Library File

Null Directory (1 sector)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	1	Entry count	Count of active entries in this directory.
01	1	1	Library size	Number of cylinders in the file library.
02	2	2	Unused	
04	4	252	Null entries	Up to 42 six-byte entries. Each entry is associated with null space in the file library.

Null Directory Entry (six bytes)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	2	Null space address	Relative disk address of the null space associated with this entry.
02	2	2	Size	Number of contiguous null sectors.
04	4	2	Unused	

BR1353

Figure 5-12. Null Directory

Password Directory (4 sectors)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	1	Entry count	Count of active entries in this directory.
01	1	3	Unused	
04	4	1020	Password entries	Up to 85 twelve-byte entries.

Password Entry (12 bytes)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	8	Password	Contains *bbbbbbb, **bbbbbb, or a user defined password.
08	8	2	Filename directory address	Relative disk address of the first filename directory block associated with this password.
0A	10	2	Unused	

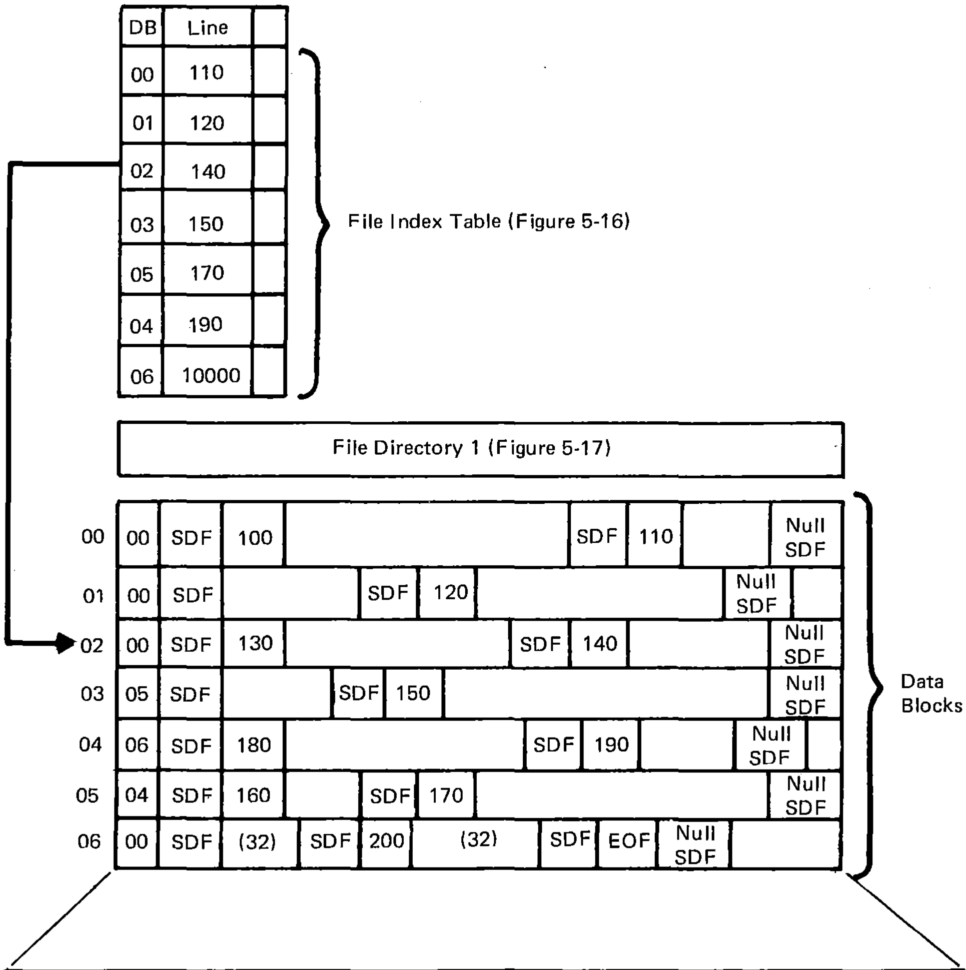
BR1354

Figure 5-13. Password Directory

Filename Directory Block (2 sectors)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	2	Block address	Relative disk address of this block.
02	2	2	Forward link	Relative disk address of the next block in this directory. X'0000' indicates the last block.
04	4	1	Entry count	Count of active entries in this block.
05	5	7	Unused	
0C	12	500	Filename entries	Up to ten 50-byte entries. Each entry is associated with a user file saved under a password.

Filename Entry (50 bytes)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	8	Filename	The name of a file as defined by the user.
08	8	2	File address	Relative disk address of the file.
0A	10	2	File length	Number of sectors allocated to the file. Includes FIT.
0C	12	1	FIT length	Number of sectors allocated to the file index table (FIT).
0D	13	1	Status indicators	If the bit is on, the file is: X'80'—A BASIC program file. X'40'—A data file generated from keyboard or cards. X'20'—A program-generated data file. X'10'—A pooled file. X'08'—A protected file. X'04'—An open file. X'02'—A data file in long precision. X'01'—A procedure file.
0E	14	2	Number of lines	Number of statement lines in the file + 1 (for the system generated EOF record).
10	16	3	Date	MDY in packed decimal.
13	19	25	File header	File identification information specified by the user.
2C	44	6	Unused	

Figure 5-14. Filename Directory Block



Sample Data Block Containing EOF (1 sector)

Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	1	Linkage indicator	Sector displacement of the next logical data block (ascending line number order), relative to the first data sector in the file. Contains X'00' when linkage is to next physical block.
01	1	4	Segment descriptor field (SDF)	Refer to Figure 5-18 for format. This SDF is associated with the second segment of statement 190 in the preceding example.
05	5	32	Data	Second segment of statement 190. Data length in the example is 32 (X'20').
25	37	4	Segment descriptor field (SDF)	Refer to Figure 5-18 for format. This SDF is associated with statement 200 (not segmented) in the example.
29	41	2	Line number	Binary line number. The line number in the example is 200.

BR1356.1

Figure 5-15. BASIC Program File Structure, Example (Part 1 of 2)

Hex Disp	Dec Disp	Dec Length	Field Name	Description
2B	43	1	Statement type code	The statement type code is used by the compiler to classify BASIC program statements. Bit 0 (X'80') on in this byte causes program statement to be bypassed on compilation (disable code). Data statements to be bypassed during input operations to a BASIC program during execution.
2C	44	32	Data	First and only segment of statement 200. Only the first segment is prefixed by the line number and statement type code.
4C	76	4	Segment descriptor field (SDF)	Refer to Figure 5-18 for format. This SDF is associated with the end-of-file (EOF) record in the example.
50	80	4	EOF record	Refer to Figure 5-19 for format. The hexadecimal value of an EOF record is always X'2710751C'. The EOF record need not be followed by a null SDF.
54	84	4	Null segment descriptor field (SDF)	Refer to Figure 5-18 for format. A null SDF can be one to four bytes in length. X'80' in the first byte identifies a null SDF.
58	88	168	Null segment (free space)	All space in a data block that follows a null SDF is referred to as a null segment.

Note: The data portion of all BASIC program statements is packed by replacing repetitions (more than two characters long) with a single character and a repetition count. The repetition count value cannot exceed X'1B'. It must be a lower value than the lowest valid functional character (X'1C'), EOF code. Refer to "Pack BASIC Program Statements (GCPACK)" in Section 3.

BR1356.2

Figure 5-15. BASIC Program File Structure, Example (Part 2 of 2)

File Index Table (up to 3 sectors)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	1	Total data blocks	Number of disk blocks (sectors) in the file that contain statements or data.
01	1	2	Total lines	Number of program or data statements in the file.
03	3	5	Unused	
08	8	2	Save area	Used only by the work file update/ crusher program (#GUFUD).
0A	10	2	FIT pointer	Core address of the first inactive FIT entry.
0C	12	752	FIT entries	Up to 189 four-byte FIT entries. Each active entry is associated with a block of data in the file.

FIT Entry (4 bytes)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	1	Disk address	Sector displacement of the data block associated with this entry, relative to the first data sector in the file.
01	1	2	Line number	Highest statement line number, in binary, associated with the referenced data block.
03	3	1	Free space	Number of unused bytes in the referenced data block (length of the null segment).

BR1357

Figure 5-16. File Index Table (FIT)

File Directory 1 (first sector)							
Up to Eight 32-Byte Data File Entries							
0	1	2	3	4	5	6	7

File Directory 1 (second sector)			
Up to Four 32-Byte Data File Entries			
8	9	10	11

File Directory 1 Data File Entry (32 bytes)

Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	1	Device code	If the bit is on, the device is: X'80'—Permanent disk X'40'—Scratch disk X'20'—Card X'10'—Printer X'08'—CRT X'07'—Unused bits. X'00' in this byte indicates the entry is not active.
01	1	8	GET/PUT filename	From an ALLOCATE system command.
09	9	6	Volume-ID	From the ALLOCATE (permanent disk only) system command.
09	9	2	Scratch file size	Number of sectors allocated to a scratch disk file. From the ALLOCATE (scratch disk only) system command.
0F	15	8	Password	From the ALLOCATE (disk) system command.
17	23	8	Filename	Name assigned to the file in the file library. From the ALLOCATE (disk) system command.
1F	31	1	Second sector indicator	In the first data file entry, this byte contains the page number of the second sector of file directory 1 in virtual memory. When there is no second sector, this byte contains X'00'. This field is used only in the first entry.

Note: Only the device code and GET/PUT filename are present for nondisk files.

BR1358A

Figure 5-17. File Directory 1 Format

Segment Descriptor Field (4 bytes)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	1	Null segment indicator	X'80' in this byte indicates the remainder of the data block is unused. The remainder of this null SDF, if present, contains binary 0's. A null SDF is used to delimit the active data segments in each data block.
00	0	2	Segment length	Binary byte count of the data segment that follows this SDF. This count includes the four bytes of the SDF.
02	2	1	Multisegment indicator	If the bit is on, the segment is: X'00'—A complete statement. X'01'—The first of a multisegment statement. X'02'—The last of a multisegment statement. X'03'—Part of a multisegment statement, but not the first or last.
03	3	1	Unused	

BR1359

Figure 5-18. Segment Descriptor Field (SDF)

End of File Record (4 bytes)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	2	Line number	The binary line number generated in an EOF record is always X'2710'. This value is equal to 10,000 which forces this record to always be the last record in the file. This value exceeds the maximum legal line number that the user can enter.
02	2	1	Statement type code	Contains a value of X'75'. This is the statement type code for an EOF record.
03	3	1	EOF code	Contains a value of X'1C'. This identifies this record as EOF. The total contents of an EOF record is always X'2710751C'.
Note: Only BASIC program files and keyboard-generated data files have a four-byte EOF record. Program-generated files have only the one-byte EOF code (X'1C') following the last data element.				

BR1360

Figure 5-19. End of File Record (EOF)

File Directory 2 (second page in virtual memory)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	2	Current file displacement	During execution of a BASIC program, this field contains the displacement from the start of the directory to the entry associated with an activated data file. Only one file can be activated (ADF) at any time. X'00' indicates that none of the files is activated.
02	2	1	Scratch file status	If this byte is not X'00', scratch disk files have been or are currently being used by the program in execution.
03	3	8	Program name	Filename of the program currently in virtual menory.
0B	11	1	Null directory indicator	X'FF' in this byte indicates the null directory is saved. X'00' indicates it is not saved.
0C	12	52	Unused	
40	64	192	Data file entries	Up to twelve 16-byte entries. Each entry is associated with a GET or PUT filename referenced in the BASIC program.

File Directory 2 Data File Entry (16 bytes)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	1	Device code	If the bit is on, the device is: X'80'—Permanent disk. X'40'—Scratch disk. X'20'—Card. X'10'—Printer. X'08'—CRT. X'07'—Unused bits. X'00' in this byte indicates the entry is not active. This byte is copied from the file directory 1 entry associated with the same GET/PUT filename.
01	1	1	I/O status	If the bit is on, the file is: X'80'—Defined on an input only file. X'40'—Defined on an output only file. X'C0'—Defined on an input/output file. X'20'—Long precision; bit is off for standard precision. X'10'—Program-generated data file; bit is off for keyboard or card-generated data files. X'08'—Currently activated for input. X'04'—Currently activated for output. X'08' and X'04' cannot both be on. X'02'—EOF indicator (output files only). X'01'—Unused bit.
02	2	1	I/O area address	Virtual memory page number of the first I/O buffer allocated to this file. Multiple buffers allocated to the same file are contiguous.

BR1361.1A

Figure 5-20. File Directory 2 Format (Part 1 of 2)

Hex Disp	Dec Disp	Dec Length	Field Name	Description
03	3	1	I/O area size	Number of contiguous buffers (pages) allocated to this file.
04	4	2	Current I/O area pointer	The virtual address, within the buffers, used to GET or PUT the next data element.
06	6	2	Disk address	Physical disk address of the first sector of data in the file. This address references the file in the system library file.
08	8	2	Current disk pointer	Relative sector displacement of the next block of data to be read into or written from the I/O area. This displacement is relative to the preceding physical disk address.
0A	10	2	File size	Total number of sectors in the file containing data.
0C	12	2	SDF count	Current count of the number of data bytes remaining in the current segment (keyboard-generated files) or bytes remaining in the I/O area (program-generated files).
0E	14	2	Unused	

BR1361.2A

Figure 5-20. File Directory 2 Format (Part 2 of 2)

System Help Text File	
Index	Text (Figure 5-22)

Sector Boundary

Help Text Index (starts on first cylinder boundary)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	2	Level number	This value is compared to a level number constant in the HELP keyword program. They must be equal to access the help text.
02	2	4	Unused	
06	6		Keyword entries	One entry for each word recognized by the HELP function. The last entry is followed by a single byte containing X'FF' to identify the end of the index.

			Keyword Entry (variable length)	
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	1	Keyword length	Length (n), in bytes, of the word in this entry.
01	1	n	Keyword	EBCDIC character string (no blanks).
01+n	1+n	3	Text displacement	Displacement from the start of the system help text file (start of index) to the first byte of first help text record to be printed for this keyword. (The first 2 bytes are sector displacement; the last byte is byte displacement.)

BR1362

Figure 5-21. System Help Text File

Help Text Record				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	4	Segment descriptor field (SDF)	Refer to Figure 5-18 for format. Records are fragmented across sector boundaries. Each segment of the record is preceded by a SDF. The system help text file is organized in a manner similar to that of the system library file (Figure 5-15).
04	4	2	Print line length	Number of characters in the line to be printed. X'0000' causes a blank line to be printed.
06	6	1	End-of-text record indicator	X'FF' indicates this record is an end-of-text record. This byte contains X'00' for all print lines.
07	7	n	Text	Length (n) is determined by the SDF. If the print line length is zero, one dummy byte is present (X'00').
Note: The print line length and end-of-text record indicator fields are present only in the first segment of a multisegment record.				

End-of-Text Record				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	4	SDF	Refer to Figure 5-18 for the format of the SDF.
04	4	2	Number of operator options	Number of three-byte operator option entries in this record. X'0000' in this field indicates no operator options are present and causes the HELP function to terminate.
06	6	1	End-of-text record indicator	This byte contains X'FF' for all end-of-text records.
07	7	n	Operator option entries	Consecutive three-byte entries, containing the byte displacements from the start of the system help text file (start of index) to the first byte of the first help text record to be printed after the multiple-choice response. The first displacement corresponds to option A, second to B, etc. n is equal to the number-of-operator-options field times 3.
Note: An end-of-text record can also be a multisegment record.				

BR1363

Figure 5-22. Help Text Records

Print Parameter List (four bytes)			
0	1	2	3
Function	Data Length	Data Area Address	

Data Length:

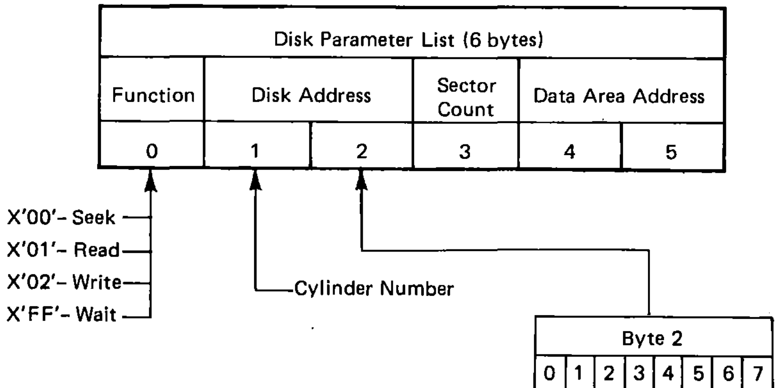
- Character Count when Printing or Punching
- X'80'–Printer Carriage Return
- X'00'–Printer Backspace

Function:

Hex Value	Device Type		
	Printer	Punch	CRT
40	Print	Insert	Print
80	CR	Punch	Return cursor
C0	Print and CR	Insert and punch	Print and return cursor
10	Backspace	Wait	Backspace cursor
11	Backspace and index	Wait	Backspace cursor
FF	Wait	Wait	Wait
4F	Not used	Wait	Roll down and print

BR1364

Figure 5-23. Print Parameter List (PPL)



The table below shows the head, sector, drive, and volume that are selected for each value that can be contained in byte 2.

Sector	Head 0				Head 1			
	R1	F1	R2	F2	R1	F1	R2	F2
0	00	01	02	03	80	81	82	83
1	04	05	06	07	84	85	86	87
2	08	09	0A	0B	88	89	8A	8B
3	0C	0D	0E	0F	8C	8D	8E	8F
4	10	11	12	13	90	91	92	93
5	14	15	16	17	94	95	96	97
6	18	19	1A	1B	98	99	9A	9B
7	1C	1D	1E	1F	9C	9D	9E	9F
8	20	21	22	23	A0	A1	A2	A3
9	24	25	26	27	A4	A5	A6	A7
10	28	29	2A	2B	A8	A9	AA	AB
11	2C	2D	2E	2F	AC	AD	AE	AF
12	30	31	32	33	B0	B1	B2	B3
13	34	35	36	37	B4	B5	B6	B7
14	38	39	3A	3B	B8	B9	BA	BB
15	3C	3D	3E	3F	BC	BD	BE	BF
16	40	41	42	43	C0	C1	C2	C3
17	44	45	46	47	C4	C5	C6	C7
18	48	49	4A	4B	C8	C9	CA	CB
19	4C	4D	4E	4F	CC	CD	CE	CF
20	50	51	52	53	D0	D1	D2	D3
21	54	55	56	57	D4	D5	D6	D7
22	58	59	5A	5B	D8	D9	DA	DB
23	5C	5D	5E	5F	DC	DD	DE	DF

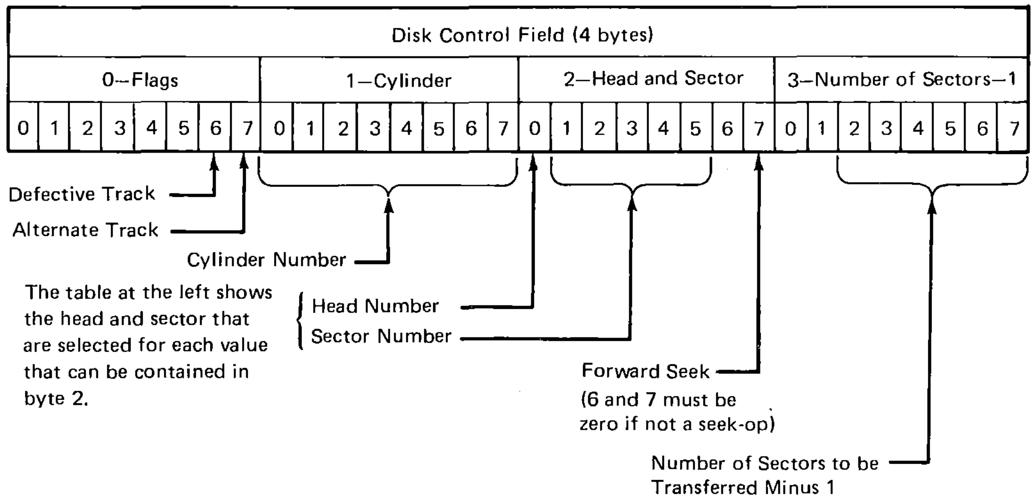
Notes:

1. Bytes 3-5 are not used for a seek function.
2. Bytes 1-5 are not used for a wait function.

BR1365

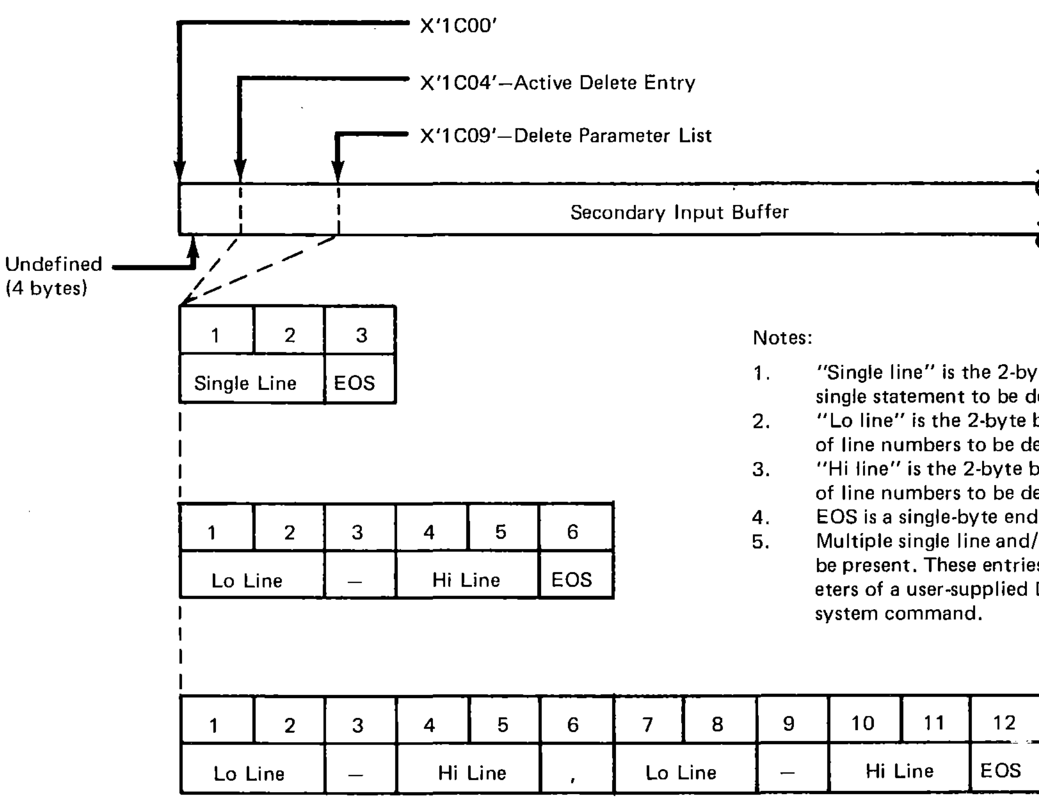
Figure 5-24. Disk Parameter List (DPL)

Sector	Head 0 (byte 3 value)	Head 1 (byte 2 value)
0	00	80
1	04	84
2	08	88
3	0C	8C
4	10	90
5	14	94
6	18	98
7	1C	9C
8	20	A0
9	24	A4
10	28	A8
11	2C	AC
12	30	B0
13	34	B4
14	38	B8
15	3C	BC
16	40	C0
17	44	C4
18	48	C8
19	4C	CC
20	50	D0
21	54	D4
22	58	D8
23	5C	DC



BR1366

Figure 5-25. Disk Control Field (DCF)



Notes:

1. "Single line" is the 2-byte binary line number of a single statement to be deleted.
2. "Lo line" is the 2-byte binary lower limit of a range of line numbers to be deleted.
3. "Hi line" is the 2-byte binary upper limit of a range of line numbers to be deleted.
4. EOS is a single-byte end-of-statement character.
5. Multiple single line and/or range delete entries may be present. These entries correspond to the parameters of a user-supplied DELETE or EXTRACT system command.

BR1367

Figure 5-26. Delete Parameter List

Command Key Table (1024 bytes)				
Hex Disp	Dec Disp	Dec Length	Field Name	Description
00	0	11	Command lengths	Length of each command assigned to command keys 1 through 11 (one byte of this field is assigned to each command key). For the IBM-assigned functions of command keys 1, 4, and 7, the length of the command is set to X'00'. The range of legal values is X'00' through X'5A'.
0B	11	23	Unused	
22	34	990	Command text	This area is divided into eleven 90-byte entries, one for each of the command keys 1 through 11. Each entry contains the command text assigned to one of the command keys 1 through 11.

BR2675

Figure 5-27. Command Key Table (##CKTB)

Command Key	IBM-assigned Function
1	Enters desk calculator operations
2	RENUMBER
3	SAVE
4	Used in editing the last line entered
5	LIST
6	CONDITION
7	EDIT generated file name
8	RUN
9	WRITE CRT
10	WRITE CRT, PRINTER
11	WRITE PRINTER

BR2676

Figure 5-28. IBM-assigned Command Key Functions

System Program File Directory Entry (16 bytes)				
Hex Disp	**Dec Disp**	**Dec Length**	**Field Name**	**Description**
00	0	6	Component name	The name of the system component (program, table, etc.) associated with this directory entry.
06	6	2	Disk address	Relative disk address of the component in the System Program File. This relative disk address points to the first sector occupied by the component.
08	8	2	Load address	Starting storage load address for the component.
0A	10	1	Sector count	Total number of contiguous sectors in the System Program File occupied by the component.
0B	11	1	Program number	This is a number assigned sequentially to the entries in this directory.
0C	12	4	Unused	

BR2677

Figure 5-29. System Program File Directory—##DRTY

MAINTENANCE UTILITY AID PROGRAM—#ZUTMO

Eleven options are provided by the maintenance utility aid program for diagnosing and correcting problems in System/3 BASIC. Certain operating procedures must be followed to initiate and successfully complete the specific option to be performed.

Operating Procedure

Performing a system stop, system reset, and system start activates the maintenance utility aid program (press SYSTEM STOP, set SYSTEM RESET to ON, and then set SYSTEM START to ON). Before activating this program, consideration should be given to manually recording certain information that is lost when this program is executed (for example, recording the IAR and ARR, or recording the last three I/O parameter lists from the system communication area).

This utility program should not be activated twice in succession except to perform diagnosis or problem correction on the maintenance utility aid program itself. A processor check may result if the R option is selected during the second successive entry to this program if the work file is defined. To terminate a selected option before it is completed, press the inquiry request switch; the maintenance utility aid program is then reinitialized, and one of the ten options can be selected. Refer to Section 3 for an internal description of the maintenance utility programs.

CAUTION

It is possible to destroy the work file if the maintenance utility aid program (#ZUTMO) is activated during crushing or reordering of the work file by the work file update/crusher program (#GUFUD).

Upon activating the maintenance utility aid program, the following message is printed:

CD, DD, VM, CP, DP, DC, DW, H, R, T, M . . . .

and the keyboard is enabled. The operator should type in the letters representing the desired function and press the carriage return key. The functions available are:

CD—Core dump
DD—Disk dump
VM—Virtual memory dump
CP—Core patch
DP—Disk patch
DC—Disk compare
DW—Disk write
H—Halt
R—Return to operating system
T—Trace
M—Library map and test

Upon completion of any of the options except H, R, or T, the option list is printed again.

CD—Core Dump Option

Entering CD invokes the core dump option. #ZUTMO then requests the dump limits:

ENTER START ADDRESS . . . .
ENTER END ADDRESS . . . .

The reply to each request should be a four-character hexadecimal address followed by a carriage return. The reply to the first message must have been entered correctly before the second message is printed. If these four characters indicating the addresses to be included in the dump are not valid, the message(s) are printed again. Following the entry of the address, the operator may wish to type in more information that will in some way identify or describe the dump he is taking. These additional characters may be added without interfering with the dump and might be instructive in reviewing it.

Once the dump limits have been entered correctly, core is dumped, 32 bytes per line with an EBCDIC interpretation for each line, beginning at the specified start address and terminating at the specified end address or when the end of core is reached.

Two headings are printed at the top of the dump. The first indicates the contents of the two index registers (BR and XR) and the PSR. The second indicates the columns for the beginning address of each line, the position of the data in the line, and the interpretation field of the line. Any characters in the line that are not printable are represented in the interpretation as EBCDIC periods.

DD—Disk Dump Option

Entering DD invokes the disk dump option. #ZUTMO then requests the read disk address and the sector count:

ENTER RD DISK ADDRESS. . . . .
ENTER SECTOR COUNT . . . .

The user should reply to the read disk request with a four-character hexadecimal address, indicating where he wishes the dump to begin, followed by a carriage return. As in the core dump, he may add additional comments if he wishes. The message is repeated if the entry is incorrect. The reply to the sector count message should be the decimal number of sectors to be dumped followed by a carriage return. When both messages have been entered successfully, the specified number of sectors, beginning at the read address indicated, are dumped, 32 bytes per line with an EBCDIC interpretation for each line. A special first header indicating the sector address is printed preceding the dump of each sector. The other header is the same as that in the core dump, while the addresses printed are displacements from the sector address. The dump terminates after all of the requested sectors have been dumped or after the last physical sector is dumped.

VM—Virtual Memory Dump Option

Entering VM invokes the virtual-memory dump option. The program then requests the first and last line numbers to be included in the dump:

ENTER FIRST LINE #. . . . .
ENTER LAST LINE #. . . . .

The user should reply to the request with a one- to four-digit decimal line number, from the BASIC program he wishes for the beginning of the dump, and a one- to four-digit decimal line number, indicating where he wishes the dump to end. A carriage return must follow each reply. If the entered data is incorrect, the message(s) is (are) reprinted. The following line represents an example of information that is included in the dump:

0190   4F09   STH   00BE   6400BE

where 0190 is the line number being interpreted, 4F09 is the virtual address of the pseudo code being interpreted (see Figure 7-1), STH is the pseudo code being interpreted, 00BE is the operand of the pseudo code, and 6400BE is the instruction that results when an operation code is substituted for the pseudo code to which it corresponds. This kind of interpretation is listed for every pseudo code that is necessary to execute the instruction at a particular line number, beginning with the specified line number and ending with the last line number requested.

CP—Core Patch Option

Entering CP invokes the core patch option. #ZUTMO then requests the beginning core address and the patch data to be put there:

ENTER START ADDRESS . . . .
ENTER PATCH DATA, USE SPACE FOR NO CHANGE

The reply to the first request should be a four-character hexadecimal address specifying where the patch should begin. If the data is not valid, the request is made again. The reply to the second request should be contiguous, hexadecimal patch data which will be terminated by a carriage return. If any errors exist in the patch data, a question mark is printed and all of the data must be reentered. If the data is valid, it replaces the data previously at the specified address. If no change is desired at the indicated address, a space should be entered followed by a carriage return. To be certain the patch is correct, the operator may wish to take a core dump of the area he wishes to patch before entering the CP option to verify the patch address, and possibly again after the function is completed to verify that the entered data is at that address.

DP—Disk Patch Option

Entering DP invokes the disk patch option. #ZUTMO then requests a displacement from a specified disk address where the patch should begin and the patch data that should be placed there:

ENTER WR DISK ADDR . . . .
ENTER DISPLACEMENT . . . .
ENTER PATCH DATA, USE SPACE FOR NO CHANGE

The reply to the first request should be a four-character hexadecimal disk address followed by a carriage return. If the data is invalid, the message is reprinted. The reply to the second message should be the two-digit hexadecimal displacement from the disk address entered where the patch should begin. Again, if the data is invalid, the request is repeated. The reply to the third message should be contiguous, hexadecimal patch data which is terminated by a carriage return. If an error occurs anywhere in the entered patch data, a question mark is printed and all of the data should be reentered. Upon successful entry of all replies, the data is placed at the proper displacement from the specified address. If a space is entered in response to the patch data request, no change is effected. The operator may wish to verify address and data by using the DD option.

DC—Disk Compare Option

Entering DC invokes the disk compare option. The program then requests the two disk addresses that are compared and the number of sectors for which the comparison should continue:

ENTER RD DISK ADDR. . . . .
ENTER CHK DISK ADDR . . . .
ENTER SECTOR COUNT . . . .

The reply to the first request should be a four-character hexadecimal disk address of the first sector to be compared. The message is reprinted if the data is invalid. The second reply should be a four-character hexadecimal disk address of the sector to be compared against the first. This message is also reprinted if the entered data is invalid. The entered sector count should be the decimal number of sectors to be compared. Again, the message is repeated if the response is invalid. If the data at the specified addresses does not correspond, a message is printed indicating the two disk addresses, the displacement from them where the difference occurred, and the data found at each disk address. Only the first nonequal byte of data is documented for each pair of sectors compared. If no difference is indicated, it is assumed that the sectors compared equally. The comparison is continued until all sectors have been compared. An example of the use of this option would be to determine if a system library has changed. The library in question could be compared against the one whose contents are known.

DW—Disk Write Option

CAUTION

The operator should be certain that the address he selects for writing is the sector he intends to change. If there is a possibility of having to restore the sector that will serve as the write address, a disk dump should be taken so that the data can be recreated.

Entering DW invokes the disk write option. #ZUTMO then requests the read and write disk addresses:

ENTER RD DISK ADDR. . . . .
ENTER WR DISK ADDR . . . .

The reply to the first request should be a four-character hexadecimal disk address that is to be read. If the data is invalid, the message is reprinted. The reply to the second message should be a four-character hexadecimal disk address where the sector indicated by the first address should be written. The READ sector is then copied to the sector indicated by the WRITE address.

H—Return to System with Halt Option

Entering H invokes the halt option. All of core that was saved other than the system nucleus area, including any patches made in saved core, is restored and a system hard halt (halt code = D5) results. At this point, the operator may choose to reenter the maintenance utility aid program, re-IPL, or manually intervene using the CE console to set the IAR to cause program execution other than the halt.

R—Return to System Option

Entering R invokes the return to system option. All saved core, including any patches made to the saved core, is restored and control is returned to the system. The system nucleus area is not restored to its previous state; therefore, it may be necessary to re-IPL the system. The system does not resume the RUN command, and is no longer in a pause state if it was in such a state upon entering the maintenance utility aid program.

Entering R on a successive activation (repeated entry) of the maintenance utility aid program may cause unpredictable results (e.g., program check) if the work file is defined. In this event, the system should be re-IPLed.

T—Trace Option

Entering T reverses (activates/deactivates) the maintenance trace option. When the maintenance trace option is activated, the names of all programs loaded by the system nucleus are displayed on the system printer. These names correspond to the names listed in Section 4.

Note: The programs that are not loaded to core by the nucleus (system nucleus, compiler overlays, etc.) are not indicated by the trace option.

Upon reversing the trace, core is restored and control is returned to the system (refer to "R—Return to System Option").

M—Library Mapping Option

Entering M invokes the library mapping option. The following message is displayed:

ENTER OPTION 1, 2, OR 3 . . . . .

The reply to this request can be any one of the three numbers, 1, 2, or 3, followed by a carriage return. The numbers have these meanings:

1—Map null and password directories
2—Map a specified password
3—Map the entire library

If the reply is invalid, the message is reprinted.

This message is displayed to request the starting disk address of the library:

ENTER LIBRARY ADDRESS . . . .

The reply to this request is the four-character hexadecimal physical disk address of the File Library area (first sector), followed by a carriage return. The physical track address (requires conversion) of the File Library is obtained by a VTOC display. If the reply to the preceding message is invalid, the message is reprinted. The following paragraphs describe the processing for each of the three options.

Option 1: This option maps the null and password directories. These items are displayed:

1. Disk address of the null directory
2. Total number of active entries in the null directory
3. Physical disk address of each null area in the library
4. Relative disk address of each null area in the library
5. Length of each null area expressed in sectors
6. Disk address of the password directory
7. Total number of active entries in the password directory
8. Each password with the physical disk address of the first user directory block associated with it

This option tests the following items:

1. Valid entry count fields in both directories
2. All disk addresses are within the library boundaries
3. Library is within the configured disk size

Option 2: This option traces the user directory blocks and files for a specified password. The following message is displayed:

ENTER PASSWORD . . . . .

The reply to this request can be any of the passwords in the password directory. If the reply is invalid or the password is not in the directory, the message is reprinted. The file-name, disk address, size, status, and header is printed for each file linked to the specified password. The status of the file is defined by these numbers:

1—Procedure file
2—Data file in long precision
3—Open file
4—Protected file
5—Pooled file
6—Program-generated data file
7—Data file generated from keyboard or card input
8—BASIC program file

The defaults for the preceding status indicators are short precision, closed, unprotected, etc.

Except for the procedure file, this option tests the following items:

1. Valid FIT
2. All lines of the file are contiguous
3. File extends to the end of the allocated space
4. An EOF is present on program-generated files
5. All elements in a program-generated file are contiguous
6. User directory blocks have no more than 10 entries
7. Any user directory block having a forward link contains 10 entries
8. First user directory block for each password has a zero entry count. All other user directory blocks have a non-zero entry count.
9. All disk addresses are valid and within the library boundaries
10. Library is within the configured disk size

If an error is detected by option 2, a message is printed and processing continues whenever possible. Certain linkages must be valid in the library directories in order to continue processing; if not a program check may occur. The error is in the last file printed in this case.

Option 3: This option maps the entire File Library. Entries for each null space, user directory block, and file are sorted, by ascending disk addresses, and printed. Each entry contains the filename, password (if applicable), physical and relative disk addresses, size, and file status (if applicable). File status is defined by these numbers:

1–Procedure file
2–Data file in long precision
3–Open file
4–Protected file
5–Pooled file
6–Program-generated data file
7–Data file generated from keyboard or card input
8–BASIC program file

The defaults for the preceding status indicators are short precision, closed, unprotected, etc.

This option tests the following items:

1. User directory blocks have no more than 10 entries
2. Any user directory block having a forward link contains 10 entries
3. The first user directory block for each password has a zero entry count. All other user directory blocks have a non-zero entry count.
4. Valid entry count fields in the null and password directories.
5. All disk addresses are within the library boundaries
6. The library is within the configured disk size
7. There are no gaps or overlaps in the library
8. Directory entries for pooled files have the same file disk address and length.

PTF COMMAND

The PTF command initiates the program temporary fix (PTF) function. This function is used to apply PTF patches to a System/3 BASIC system program file or the system help text file. For PTF purposes, any component residing on cylinder 0 is considered part of the system program file. (PTF's to the disk system management program in a co-resident system must be applied using the PTF function in the disk system management program.) This command may be entered from the keyboard or the data recorder if in read card mode.

Following the PTF command, a PTF is entered using four types of secondary commands called PTF statements. If these statements are entered from the keyboard, they are typed as if they were system commands, with the only exception being that rejection of the statement returns control to PTF mode rather than to the system. Thus, if an invalid statement is typed, the statement may be reentered.

Note: If a DATA statement is reentered, tabbing across the input line generally does not reproduce the checksum value originally entered, but instead leaves four blanks in its place.

The only way to abort PTF mode following the PTF command entered through the keyboard is to use inquiry request. This aborts the PTF function being performed and returns control to the system with any partially entered PTF information being lost. Inquiry request does not abort the PTF function after a valid PTF END statement has been entered. A return is made to keyboard mode upon complete processing of the PTF END statement.

If the PTF command is entered from the data recorder, all subsequent PTF statements are read from cards automatically, similar to normal system input from the data recorder. Columns 88-96 (73-80 if configured for a 129) of each card containing a PTF statement after the PTF command are ignored; thus each card can contain a sequence number. Each card is listed as it is read. Any error detected in the PTF function, while in card mode, causes the entire PTF function to be aborted and a return to be made to the system. If inquiry request is used while reading cards, this aborts the PTF function being performed, as long as a valid PTF END statement has not been read yet.

Any line entered in PTF mode, other than the HDR, PTF, DATA, and END statements described, is rejected and a question mark is printed.

PTF's applied to the system can be listed by dumping the PTF Log (e.g., disk dump). Each entry in the log is six bytes in length.

Error Conditions

The command is rejected and PTF mode is not entered if any of the following errors occurs:

1. The system work area has not been allocated on the disk containing the current system program file. This work area is used as a temporary storage for the PTF data.
2. Any character, other than blank(s) or a carrier return, is entered following PTF.

HDR Statement

Syntax

HDR ȼ ptfid ȼ cksum ȼ disk-spec [ȼ disk-spec]

This statement defines the start of a PTF and must be the first statement entered following the PTF command. The parameters of this command are:

1. ptfid—Six-character PTF identification. For programs in the system program file, the first three characters are .BS and the last three characters are the PTF sequence number (in range 000-255). For the help text file the PTF ID is .BH followed by the three-character PTF sequence number.
2. cksum—The four-digit checksum for this statement. (It includes the HDR and ptfid characters but not the checksum and the disk specifications.)
3. disk-spec—The unit containing the system program file or system help text file to which the PTF is to be applied.
4. [disk-spec] —If this parameter is present, it specifies that the PTF will come from disk rather than being entered from the keyboard. The specified unit must contain a System/3 BASIC PTF file (VTOC name is PTF). This file can contain several PTF's in the form of card images containing HDR, PTF, DATA, and END PTF statements.

The file is searched to find the first PTF whose HDR statement matches this statement entered from the keyboard. This PTF is then automatically applied and a return is made to a normal utility mode input condition. The PTF statements applied from disk are not normally printed. If any error is detected, the card image containing the error is printed, followed by the error message. A return is then made to the system.

Error Conditions

This statement is rejected if any of the following errors occurs:

1. The optional disk unit parameter is specified and no PTF file is on that disk.
2. The optional disk is specified and a HDR matching the entered one does not exist in the PTF file.
3. The checksum is incorrect.
4. The first specified disk does not contain a system program file or help text file, depending on what is being patched.
5. A valid HDR statement has already been accepted.

Examples of correct syntax (the use of commas is optional):

```
HDR    .BS000   2244,  R1
HDR,   .BH029   A63F,  F1   R2
```

PTF Statement

Syntax

PTF ¢ prog-name ¢ level ¢ cksum

The PTF statement identifies the component in the system or help text to which the patch data in the following DATA statements is to be applied. It must be the first statement following the HDR statement. The parameters of this command are:

1. prog-name—This is the System/3 BASIC six-character program name prefixed with a period. This identifies the component to be patched.

   Note: The help text file consists of more than one program.
2. level—This is a two-digit number specifying the release level of the system program file or help text file to which the PTF should be applied. The release level can be located in ##DRTY (first component in the system program file).
3. cksum—The four-digit accumulative checksum for this statement and the preceding HDR statement.

Error Conditions

The statement is rejected if any of the following errors occurs:

1. The release level of the system program file or help text file on the unit specified in the HDR statement is not the same as the release level specified in this PTF statement.
2. The specified program name is not a valid System/3 BASIC component.
3. The checksum is incorrect.
4. A HDR statement has not been entered.
5. Two PTF statements are entered without intervening DATA statements.
6. The help text is specified and it is not found on the specified disk.
7. The program name and the PTF identification are incompatible. (For programs in the system program file, .BS must have been specified in the HDR statement. If the help text file is specified, .BH must have been entered.)

Examples of correct syntax (the use of commas is optional):

```
PTF    .#KREAD,  00,  57DC
PTF,   .#T3HEL,  01   A996
```

DATA Statement

Syntax

DATA ₵ cksum ₵ hex-addr
₵ hex-byte [hex-byte] . . .

This statement specifies the patch data. Any number of DATA statements (subject to total patch size defined) may be entered. The end of the DATA statements for this PTF is delimited by another PTF statement, or by the END statement. The parameters of this command are:

1. cksum—The four-digit cumulative checksum including this statement and all previous statements in the PTF.
2. hex-addr—This is the absolute core starting location within the specified program for the data bytes on this statement. This address is not relative to the start of the program. It is the relative byte displacement plus the starting core address of the program to be patched. For example, a patch of the third byte of a program that starts at X'0C00' would specify X'0C02' as the starting patch address. Thus, this address corresponds to the addresses shown on the assembler listings for the program.
3. hex-byte—The hexadecimal bytes (each one represented by two hexadecimal digits) define the information to be placed in the component. The first data byte of the DATA statement replaces the contents of the byte located at the starting address specified for this statement. The second byte is placed at the starting address plus one, etc.

This command saves the specified code change and its location in the system work area section of the disk containing the current system. There is no restriction on the length of an individual DATA statement other than the line width of the input device. However, for any single component PTF, the total number of DATA statements times 10, plus the number of hexadecimal bytes of code changes, must be less than approximately 36k. A file in the work file section of the work area will be destroyed by this function. The disk copy of the specified component is not updated until the END statement is entered.

Error Conditions

This statement is rejected if any of the following errors occurs:

1. The specified cumulative checksum does not match the accumulated checksum.
2. A HDR or PTF statement has not been previously accepted for this PTF.
3. More data than that which can be contained in 36k is entered.

Examples of correct syntax (the use of commas is optional):

```
DATA    0158,    0C00,    F1F2F3F4F5F6
DATA    59BF     0EFE     C08704651092
```

END Statement

Syntax

END ₵ cksum

This command is used to signify the end of a PTF. If the HDR, PTF, and DATA statements were accepted and the specified checksum matches the accumulated checksum for this statement and all preceding PTF statements, the copy of the specified component in the system program file or help text file is updated. The record of installed PTF's on sector 23, track 1, cylinder 0, of the disk containing the patched component, is updated after the successful application of the PTF. If the number of PTF's exceeds that which can be contained in one sector, the record of the oldest installed PTF is lost.

When a PTF is applied to a disk (volume), the system work area on that disk (volume) is unassigned. If the updated component is one of the system components that has a copy in the system work area, any existing work areas are not updated with the PTF.

The ASSIGN-WORKAREA command updates the working copies of the components when the work area is recreated. The work areas on both R1 and F1 contain copies of system programs and both should be updated with the ASSIGN-WORKAREA utility command.

When the PTF END statement is completed, PTF mode is switched to system mode.

Error Conditions

This statement is rejected if any of the following errors occurs:

1. HDR and PTF commands, and at least one DATA command, have not been previously accepted for this PTF.
2. The accumulated checksum for the PTF does not match the specified checksum.

Examples of correct syntax (the use of commas is optional):

```
END     2019
END,    5548
```

The following example might be used as a PTF to the command analyzer system program (#ECMAN):

```
PTF
HDR     .BS001    2A44     F1
PTF     .#ECMAN   00,      33E6
DATA,   3DFS,     0EEC,    6F
END     EE4D
```

Assuming R2 contains a PTF file, the following PTF for the help text component might be entered through the keyboard:

```
PTF
HDR     .BH000    22BD     F1,   R2
```

I/O PARAMETER LIST SAVE AREA

Contained within the nucleus of the system is a pushdown stack that contains the last three I/O parameter lists that have been handled by the system. This area is near the upper end of the nucleus and starts at label $PLST1. On the sample listing shown in Figure 6-1, the label is NPLST1 and is at address X'044E'. The area has three labels—NPLST1, NPLST2, and NPLST3. Each label refers to a seven-byte entry in the stack. NPLST1 is the last I/O parameter list to be handled by the system and NPLST2 is the next to the last, etc.

Interpretation of I/O Parameter List Area

All information about these parameter lists is contained in Figures 5-23 and 5-24 with one exception: the first byte of each seven-byte entry determines the device referenced by the parameter list:

Hex 00, 01, 02, or 03	DPL for disk
Hex D7	PPL for printer
Hex C3	PPL for CRT

```
##1TRK NPAUSE - EXMSGS SAVE/RESTORE CORE INTERFACE
ERR LOC OBJECT CODE      ADDR STMT SOURCE STATEMENT
                              2139 ***********************************************************
                              2140 *  ROUTINE TO SAVE/RESTORE CORE AND EXEC EXECUTION MESS
                              2141 ***********************************************************
                              2142 *
    04B6                      2143        ORG   $PAUSD                      SET LOCATION CO
                         0489 2144        USING $UNMSK,@BR
                         04B6 2145 NPAUSE EQU   *                           ENTRY TO SAVE C
    04B6 34 08 0494           2146        ST    CI0100+@OP1,@ARR            SAVE RETURN ADD
                         04BA 2147 NPAUS1 EQU   *                           ENTRY FOR EXECU
    04BA 34 01 04F6           2148 CS0010 ST    CS0060+@OP1,@BR             SAVE BASE REGIS
    04BE C2 01 0489           2149        LA    $UNMSK,@BR                  LOAD BASE REGIS
```

Replace coding at statement 2148 with this branch: C0 87 05A4. This causes a branch to the patch at address 05A4.

Starting at location 05A4, put in the following 14-byte patch:

Address	Data	
05A4	0C 14 05C6 0462	This instruction moves pushdown stack to save area.
05AA	34 01 04F6	This instruction replaces overlaid instruction at 04BA.
05AE	C0 87 04BE	This instruction is for branch back to statement 2149.

Data from pushdown stack for parameter lists is now saved in 21-byte area from address 05B9 through 05C6.

The following information was required to obtain the second address (X'0462') that is in the MOVE instruction at address 05A4.

Refer to the listing shown below; the address is the high-order byte of NPLST3.

```
##1TRK NUCLES - PERMANENT STORAGE AND CONSTANT AREAS

    044E                 0454 1893 NPLST1 DS    CL(@DPLNG+1)            LAST I/O PARAM L
    0455                 045B 1894 NPLST2 DS    CL(@DPLNG+1)            2ND TO LAST PARM
    045C                 0462 1895 NPLST3 DS    CL(@DPLNG+1)            3RD TO LAST PARM
```

BR1368

Figure 6-1. Procedure to Save I/O Parameter Lists

Recovery of Parameter List Information

It is important to note that one of two methods of retrieval must be used to display this information:

1. Display the parameter list area of the nucleus with the CE console starting at about address X'044E'.
2. Modify the nucleus with a patch to save the list information prior to calling in the maintenance utilities. (The fetch of the maintenance utilities ordinarily updates the list and overlays the information to be displayed.) This patch is useful if several dumps of the parameter lists are required.

Modification of Nucleus to Save Parameter Lists

1. Find the core save routine (label NPAUSE on listing for ##1TRK at about address X'04B6').
2. Overlay patch the four-byte instruction that saves the base register (34 01 XRXR) with an unconditional branch (C0 87 XXXX, where XXXX is the address of a patch area).

3. Locate a patch area of 35 bytes; or locate two areas, one with at least 14 bytes and one with at least 21 bytes. (Try this around address X'05A4' where an area of more than 35 bytes is available.)
4. In the 14-byte area (or first part of the 35-byte area), enter the following patch: 0C 14 YYYY ZZZZ 34 01 XRXR C0 87 RTRT, where

   YYYY = Address of the last byte of the patch area where the parameter lists will be saved.

   ZZZZ = Address of the last byte of the I/O parameter pushdown stack (around X'0462').

   XRXR = Address in last two bytes of instruction that was overlaid in step 2 of this procedure.

   RTRT = Address of next instruction in NPAUSE.
5. Refer to Figures 6-1 and 6-2 for detailed examples of this procedure.

```
THIS PROCEDURE WILL PATCH THE NUCLEUS TO SAVE THE LAST THREE I/O PARAMETERS
CD,DD,VM,CP,DP,DC,DW,H,R,T,L.......CP   THIS IS THE OVERLAY IN THE CORE SAVE ROUTINE
ENTER START ADDRESS..........04BA
ENTER PATCH DATA, USE SPACE FOR NO CHANGE
C08705A4

CD,DD,VM,CP,DP,DC,DW,H,R,T,L.......CP   THIS IS THE PATCH  TO SAVE THE I/O PARAMETERS
ENTER START ADDRESS..........05A4
ENTER PATCH DATA, USE SPACE FOR NO CHANGE
0C1405C60462340104F6C08704BE

CD,DD,VM,CP,DP,DC,DW,H,R,T,L.......R   THE NUCLEUS IS PATCHED AND WE WILL RETURN TO BASIC

READY
EDIT LINE
WORK FILE HAS BEEN CLEARED AND NAMED LINE

READY

CD,DD,VM,CP,DP,DC,DW,H,R,T,L.......CD   THE FOLLOWING IS A DUMP OF THE PARAMETER SAVE AREA AND THE PATCH
ENTER START ADDRESS..........05A0
ENTER END ADDRESS............05D0
     BR=0C00    XR=1D0B   PSR=0101
ADDR   +00 1 2 3  4 5 6 7   8 9 A B  C D E F +10 1 2 3  4 5 6 7  8 9 A B  C D E F  ::::::::::INTERPRETATION::::::::::
                           14-Byte Patch Area            NPLST1          NPLST2
05A0    0C000007 0C1405C6  04623401 04F6C087  04BEC3FF 0104012F 81D7C005 17840180  :.......F.....6....C......P......:
05C0    C3C00517 842F8100  00000000 00000000  00000000 00000000 00000000 00000000  :C..............................:
           NPLST3
CD,DD,VM,CP,DP,DC,DW,H,R,T,L.......
```

BR1369

Figure 6-2. System Printer Output, Example

STAND-ALONE DUMP

The possibility exists that the maintenance utilities cannot be loaded. An example would be if a problem (hardware or software) changed the coding within the nucleus, in an area that is required for saving core and loading the utilities, making the maintenance utilities unavailable. To provide information in such a situation, the following program can be keyed into the system.

This stand-alone dump starts dumping at address 0000 and continues until stopped or until core is exceeded (Figure 6-3).

Stand-Alone Dump Procedure

1. Prior to entering the program, record (from the CE panel and/or operator panel) any hardware data that may aid in diagnosing the problem:

   IAR
   ARR
   Status

2. Enter the following program by using manual entry via the keyboard, starting at address 1F00. Refer to *IBM System/3 Model 6 Components Reference Manual,* GA34-0001, "Altering Storage."

Hexadecimal Address	Data in Hexadecimal Notation							
1F00	3C	7F	1F	5E	C2	01	1F	70
1F08	C2	02	00	00	68	02	00	00
1F10	68	03	01	00	7A	F0	00	7A
1F18	F0	01	7D	FA	00	F2	82	05
1F20	4E	00	00	1F	68	7D	FA	01
1F28	F2	82	05	4E	00	01	1F	68
1F30	D2	01	02	E2	02	01	0F	00
1F38	1F	64	1F	65	C0	01	1F	0C
1F40	31	E4	1F	61	31	E6	1F	63
1F48	F3	E0	00	0E	01	1F	0B	1F
1F50	67	C1	E2	1F	51	3C	40	1F
1F58	64	C0	87	1F	00	C0	7F	05
1F60	1F	70	1F	5D	40	01	00	40
1F68	C7							

Note: X'1F68' is the last byte.

BR1370

3. Alter IAR to address 1F00 and start CPU.
4. Stop CPU when necessary information has been dumped.
5. Write information on the dump to define the core locations (see sample dump in Figure 6-3).

```
ADDR.                   +08              +10              +18
0000  D08704B6E7E7E7E7 E7E7E7E740C3D6D7 E8D9C9C7C8E340C9 C2D4
  40  F802FD4C00010476 3C800476D087417B FF1F7D0219F2010C 5C0
  80  1E5F001C35580218 197BF0193C23009D 78021BF290043C24 009
  C0  19F281045E006237 5E0026265E00263B D0877ED087533C87 047
0100  000003010B4C0407 0000000100020 0F5 00A00000FF010974 0852
  40  1C5C0078265E0078 195C007918F3A100 C08700D65C01211B 7408
  80  2520C08700D35800 B9265800BC267002 2870032A380C03D5 F21
  C0  4027C01002FE7804 28C0100304792E27 C09001FD780127F2 10
0200  219C03DC229C0001 26BA0101F3000198 000B26F1A200E100 A89
  40  E435AD01DADEF201 11AD01DFE1F2810A 6C011EE57C011FF2 871E
  80  011BE77D011FF281 0678801BE010737C 001FF287789F00EA 35F2
  C0  5C041E325C011937 5F0062379F00EB35 F20131C08701E09F 00E
0300  25F287123A0C03D5 1E00043435D0877E D08753D08741C087 00
  40  00F1E200384003D2 F29003F1F200C087 00250394C0870025 039
  80  0000000000000000 00F01844C0870389 0389033002001005 070
  C0  8400001A0000051E F0F1F0F040238007 0000C40110020004 0000
0400  D3C6F14000010000 0000000000020000 0000000000030001 0012
  40  0000000000000000 0000070700000001 0B4C050700000104 011
  80  100488C087047500 00340804943C8704 76C0870DB03A2003 C33
  C0  04895C01710B7402 657C87607C0281F2 870AC20104897C01 817
0500  04F7C0 7051A0510   4001 2001A0600 0107D4030C003C80 0565
```

BR1371

Figure 6-3. Stand-Alone Dump, Example

PATCHING A DISK RESIDENT SYSTEM PROGRAM

Considerations: Two types of patches should be considered:

- Overlay
- Additional coding

Overlay Patch: To make an overlay patch, locate the section of coding to be replaced, and change that specific area.

Additional Coding: To patch with additional coding, several things must be considered:

- How to exit from original coding?
- Will the base register range be exceeded?
- Where is space available for additional code?
- How can this coding be restored if it is not effective?
- How to return to original coding?

Solutions to Considerations: The considerations can be resolved as follows:

- To exit from the original code, overlay patch at the logical point in original coding with a branch to the additional code (example: C0 87 XXXX, where XXXX is address of the patch).

  Note: This may require the overlay of more than one instruction.

- If the patch that is branched to is beyond the range of the base register, use long instructions in the patch, to prevent the need for changing registers (when possible).

- The space for patching with additional coding can be found in four optional areas:

  1. Patch area of the module.
  2. Overlay a message constant.
  3. Overlay a section of the module not being used.
  4. Patch area of the nucleus.

  Note: You must consider that option 2 will cause incorrect messages and options 3 and 4 will work for only a temporary situation.

- To restore coding with a minimum of effort, the original coding should be placed on an unused area of disk (use DW option of maintenance utilities).

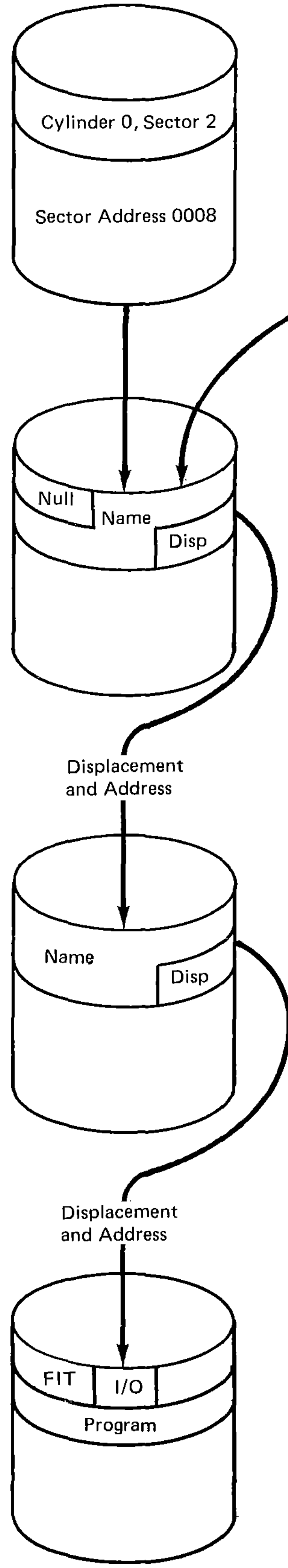
- To return to original coding requires a branch as follows: C0 87 YYYY, where YYYY is the return address.

  Note: Be sure to include, as the last part of the patch, that portion of the program that was overlaid by the branch to this patch.

Procedure to Patch a Disk Resident Program: Patch a disk resident program as follows:

1. Determine the disk address of segment(s) of program to be patched. Locate the directory entry for the program in ##DRTY (first seven sectors of the system program file). This entry contains the starting disk address and number of sectors occupied by the program. The relative location of a patch can be determined by matching machine code between the program's assembly listing (microfiche) and a dump of the sectors the program occupies.
2. Write sector(s) to be patched to some unused area of disk. This step should be taken to provide a simple method of restoring the program to normal.
3. Modify program with disk patch facility.
4. If the patch does not work and/or the program must be returned to normal, copy back the information saved in step 2 to restore the program.

FINDING A LIBRARY FILE ON DISK



1. Dump the volume label at disk address 0008. (Note: The appropriate disk and drive bits must be added for files not on R1; refer to "Disk Address Specifications for Utility Dump" in Section 7.) The cylinder address of the password directory is located in 00FD and 00FE.

2. Dump the password directory (5 sectors) using the disk address obtained in step 1. (Note: Disk and drive bits must be added for files not on R1; refer to "Disk Address Specifications for Utility Dump" in Section 7.) The first sector is the null directory. In the next 4 sectors, a 2-byte displacement follows each 8-byte name field. Add the appropriate displacement to the address of the password directory. Convert this address to a physical disk address by referring to "Disk Address Specifications for Utility Dump" in Section 7 to find the user file directory. Example:

Cylinder address of password directory	2D00
Two-byte displacement	0C1C
Added	391C
Converted	3990

3. Dump the user file directory using the disk address calculated in step 2. The 8-byte name field is followed by a 2-byte displacement. Add this displacement to the disk address found in step 1 and convert to a physical disk address. Dump the number of sectors specified in the file length bytes. The file length is located in the 2 bytes following the file displacement (Figure 5-14). A keyword-generated file has a 1- to 3-sector FIT, a 1-sector I/O record if it is a program file, and data blocks.

   Note: If the user file directory is larger than 2 sectors, a forward link displacement resides in the third and fourth bytes of the block header. If there are no additional blocks, this field will contain binary 0's.

4. Program files have been character-packed and set up as "segments." Data items in data files are in internal format. Keyboard data files also are "segmented."

BR1372

HOW TO USE THE FE MAP

The nucleus contains a six-byte area that identifies the last six program modules that have been loaded by the system. This area can be found by referring to the listing for ##1TRK around address X'0584'. The label for the area is NFEMAP and there is also a system equate of $FEMAP. Each byte in this area identifies a program module that has been loaded. The byte at the lowest address is the last module loaded. You must take into consideration that calling in the maintenance utilities makes three entries in the FE map. If you need to know more than the last three modules loaded, you must display this area using the CE console.

Identification of Programs in the FE Map

Assume that the following hexadecimal data is in the FE map: X'5A 04 2E 01 03 02'. This indicates that the last module loaded was #ZUTMO and that it was preceded by #DPRIN, #EXMSG, #INSTD, #LOADR, and #BCOMP.

Figure 6-4 shows how program numbers can be found in the FE map for #INSTD, #BCOMP, #DPRIN, and #LOADR. To obtain this information, take a disk dump of the system program file directory (first seven sectors of the system program file). This provides a list of all program modules and the hexadecimal number for that particular module.

```
CD,DD,VM,CP,DP,DC,DW,H,R,T,L .......DD
ENTER RD DISK ADDR...........0A00
ENTER SECTOR COUNT...........6
SECTOR ADDR= 0A00
                                                                                      RELATIVE SECTOR NUMBER=0000

ADDR  +00 1 2 3  4 5 6 7  8 9 A B  C D E F +10 1 2 3  4 5 6 7  8 9 A B  C D E F  ************INTERPRETATION************
0000   0000C6D6 E5D947F0 FFFFA7F2 01F9C087  7B7BC4D9 E3E80000 00000700 85C08711  *..FOVR.0....2.9..##DRTY.........*
0020   7BC9D5E2 E3C4001C 06001601 C20115F3  7BC2C3D6 D4D70080 06001802 61F28175  *#INSTD......B..3#BCOMP....../2..*
0040   7BD3D6C1 C4D90100 06001303 61F2818A  7BC4D7D9 C9D5014C 07000504 4D020312  *#LOADR....../2.#DPRIN.(.....(...*
0060   7BD2C7D6 E2D30180 0C000205 4D011012  7BD2C5C4 C9E30188 0C000E06 01963402  *#KGOSL.....(...#KEDIT.........*
0080   7BD2C5D5 C1C201C4 0C000607 C08711BC  7BC4D9C5 C1C40200 08890108 65F20167  *#KENAB.D........#DREAD.......2.
00A0   7BD2D4D6 E4D50204 0C000409 87713C63  7BD2D9D4 D6E50214 0C00030A F201483C  *#KMOUN..........#KRMOV......2..
00C0   7BD2D7C1 E2E60220                    11DEBD6B           E3D90234 0C00030C       CC087  *#KPASW.......#K
```

Note: This is a disk dump of the system program file directory.

BR1373

Figure 6-4. Identification of Program Numbers

ADDRESS STOP PROCEDURE FOR PROGRAM LOADING

This procedure can be used to stop the system before or after execution of a specific program module. This method enables you to obtain a core or disk dump at a specific point, or allow a check of system indicators at a specific time.

Stop Address Selection

The following three stopping points can be used:

Label of Address in Nucleus	*Current Address	Condition at Time of Stop
1. NBLOAD	X'051E'	Last module name printed by trace has executed.
2. NBL067	X'056E'	Last module name printed by trace has loaded but not executed.
3. NLOADR	X'0516'	Will occur when a module is called to load only. Name not yet printed by trace; e.g., this happens when I/O modules are called into the low end of core storage.

*The core addresses used are subject to change and are shown here only as an instructional aid.

BR1374

Method to Activate Address Stop

1. Turn on module trace by using the "T" option of the maintenance utilities.
2. Choose the address that you require and set it up with the address switches on the CE console.
3. Turn on the address compare stop switch. Make sure the roller switch is set for SAR. Run the program until the proper module name prints. Make observations or take dumps that you require.

HALT 2345

This halt occurs during IPL if the volume on disk drive R1 is initialized (contains formatted tracks), but does not have a standard System/3 volume label on cylinder 0, head 0, sector 2. The following procedure is used to bypass halt 2345:

1. Make sure the user wishes to destroy the data content of the volume mounted on R1. The volume label, or any area, can be displayed from R1 using the disk dump (DD) option of the maintenance utility aid program (#ZUTMO). This program can be invoked when the halt occurs.
2. If the first 3 bytes of the volume label are not the characters VOL, the volume does not have a standard System/3 volume label.
3. Invoke the disk patch (DP) option of the maintenance utility aid program.
4. Store X'ABCDEF' as the first 3 bytes of sector 2 on cylinder 0, head 0. The IPL program accepts this hexadecimal value (R1 only) and does not issue halt 2345.
5. Perform a system IPL after completing the patch. Volumes that are patched in this manner are assumed to require initialization.

This section details a method of laying out the contents of an execution-time disk dump of virtual memory. (Note: Taking a *complete* dump of virtual memory is not a realistic approach to troubleshooting user-program execution problems.) This section also details a method for determining the contents of an execution-time core dump. Refer to Figure 7-1 to convert virtual addresses to disk addresses.

HOW TO TAKE A SEQUENTIAL DISK DUMP OF VIRTUAL MEMORY

The disk area occupied by virtual memory is cylinders 7 and 8, and over half of cylinder 9 in the system work area. As this area is a logical four-track file, it is necessary to individually dump the following six disk areas to get a sequential listing of virtual memory:

1. Starting disk address–0700; sector count–48 (cylinder 7; R1).
2. Starting disk address–0701; sector count–48 (cylinder 7; F1).
3. Starting disk address–0800; sector count–48 (cylinder 8; R1).
4. Starting disk address–0801; sector count–48 (cylinder 8; F1).
5. Starting disk address–0900; sector count–48 (cylinder 9; R1).
6. Starting disk address–0901; sector count–16 (cylinder 9; F1).

Total 256 pages (64k)

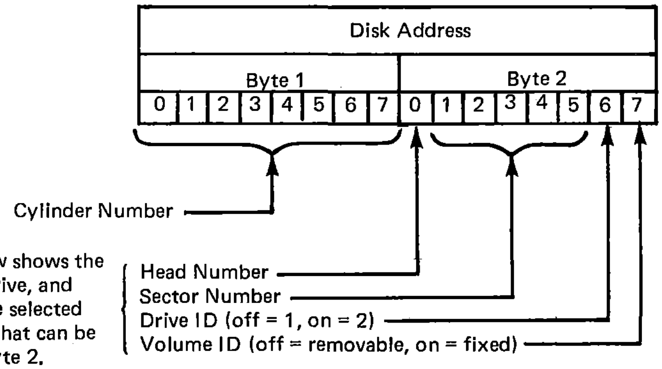
Virtual Memory	Disk Addr	Virtual Memory	Disk Addr	Virtual Memory	Disk Addr	Virtual Memory	Disk Addr
00xx	0700	40xx	0741	80xx	08A0	C0xx	0900
01xx	0704	41xx	0745	81xx	08A4	C1xx	0904
02xx	0708	42xx	0749	82xx	08A8	C2xx	0908
03xx	070C	43xx	074D	83xx	08AC	C3xx	090C
04xx	0710	44xx	0751	84xx	08B0	C4xx	0910
05xx	0714	45xx	0755	85xx	08B4	C5xx	0914
06xx	0718	46xx	0759	86xx	08B8	C6xx	0918
07xx	071C	47xx	075D	87xx	08BC	C7xx	091C
08xx	0720	48xx	0781	88xx	08C0	C8xx	0920
09xx	0724	49xx	0785	89xx	08C4	C9xx	0924
0Axx	0728	4Axx	0789	8Axx	08C8	CAxx	0928
0Bxx	072C	4Bxx	078D	8Bxx	08CC	CBxx	092C
0Cxx	0730	4Cxx	0791	8Cxx	08D0	CCxx	0930
0Dxx	0734	4Dxx	0795	8Dxx	08D4	CDxx	0934
0Exx	0738	4Exx	0799	8Exx	08D8	CExx	0938
0Fxx	073C	4Fxx	079D	8Fxx	08DC	CFxx	093C
10xx	0740	50xx	07A1	90xx	0801	D0xx	0940
11xx	0744	51xx	07A5	91xx	0805	D1xx	0944
12xx	0748	52xx	07A9	92xx	0809	D2xx	0948
13xx	074C	53xx	07AD	93xx	080D	D3xx	094C
14xx	0750	54xx	07B1	94xx	0811	D4xx	0950
15xx	0754	55xx	07B5	95xx	0815	D5xx	0954
16xx	0758	56xx	07B9	96xx	0819	D6xx	0958
17xx	075C	57xx	07BD	97xx	081D	D7xx	095C
18xx	0780	58xx	07C1	98xx	0821	D8xx	0980
19xx	0784	59xx	07C5	99xx	0825	D9xx	0984
1Axx	0788	5Axx	07C9	9Axx	0829	DAxx	0988
1Bxx	078C	5Bxx	07CD	9Bxx	082D	DBxx	098C
1Cxx	0790	5Cxx	07D1	9Cxx	0831	DCxx	0990
1Dxx	0794	5Dxx	07D5	9Dxx	0835	DDxx	0994
1Exx	0798	5Exx	07D9	9Exx	0839	DExx	0998
1Fxx	079C	5Fxx	07DD	9Fxx	083D	DFxx	099C
20xx	07A0	60xx	0800	A0xx	0841	E0xx	09A0
21xx	07A4	61xx	0804	A1xx	0845	E1xx	09A4
22xx	07A8	62xx	0808	A2xx	0849	E2xx	09A8
23xx	07AC	63xx	080C	A3xx	084D	E3xx	09AC
24xx	07B0	64xx	0810	A4xx	0851	E4xx	09B0
25xx	07B4	65xx	0814	A5xx	0855	E5xx	09B4
26xx	07B8	66xx	0818	A6xx	0859	E6xx	09B8
27xx	07BC	67xx	081C	A7xx	085D	E7xx	09BC
28xx	07C0	68xx	0820	A8xx	0881	E8xx	09C0
29xx	07C4	69xx	0824	A9xx	0885	E9xx	09C4
2Axx	07C8	6Axx	0828	AAxx	0889	EAxx	09C8
2Bxx	07CC	6Bxx	082C	ABxx	088D	EBxx	09CC
2Cxx	07D0	6Cxx	0830	ACxx	0891	ECxx	09D0
2Dxx	07D4	6Dxx	0834	ADxx	0895	EDxx	09D4
2Exx	07D8	6Exx	0838	AExx	0899	EExx	09D8
2Fxx	07DC	6Fxx	083C	AFxx	089D	EFxx	09DC
30xx	0701	70xx	0840	B0xx	08A1	F0xx	0901
31xx	0705	71xx	0844	B1xx	08A5	F1xx	0905
32xx	0709	72xx	0848	B2xx	08A9	F2xx	0909
33xx	070D	73xx	084C	B3xx	08AD	F3xx	090D
34xx	0711	74xx	0850	B4xx	08B1	F4xx	0911
35xx	0715	75xx	0854	B5xx	08B5	F5xx	0915
36xx	0719	76xx	0858	B6xx	08B9	F6xx	0919
37xx	071D	77xx	085C	B7xx	08BD	F7xx	091D
38xx	0721	78xx	0881	B8xx	08C1	F8xx	0921
39xx	0725	79xx	0884	B9xx	08C5	F9xx	0925
3Axx	0729	7Axx	0888	BAxx	08C9	FAxx	0929
3Bxx	072D	7Bxx	088C	BBxx	08CD	FBxx	092D
3Cxx	0731	7Cxx	0890	BCxx	08D1	FCxx	0931
3Dxx	0735	7Dxx	0894	BDxx	08D5	FDxx	0935
3Exx	0739	7Exx	0898	BExx	08D9	FExx	0939
3Fxx	073D	7Fxx	089C	BFxx	08DD	FFxx	093D

BR1378

Figure 7-1. Conversion of Virtual Addresses to Disk Addresses

Disk Address Specifications for Utility Dump

The following chart provides a means of converting disk address (cylinder, head, sector ID, and spindle ID) into a two-byte address format that the programming system requires. For example, cylinder 5, head 0, sector 2 for R1 (spindle-drive) is disk address X'0508'.



The table below shows the head, sector, drive, and volume that are selected for each value that can be contained in byte 2.

Hexadecimal Sector	Decimal Sector	Head 0			
		R1	F1	R2	F2
00	0	00	01	02	03
01	1	04	05	06	07
02	2	08	09	0A	0B
03	3	0C	0D	0E	0F
04	4	10	11	12	13
05	5	14	15	16	17
06	6	18	19	1A	1B
07	7	1C	1D	1E	1F
08	8	20	21	22	23
09	9	24	25	26	27
0A	10	28	29	2A	2B
0B	11	2C	2D	2E	2F
0C	12	30	31	32	33
0D	13	34	35	36	37
0E	14	38	39	3A	3B
0F	15	3C	3D	3E	3F
10	16	40	41	42	43
11	17	44	45	46	47
12	18	48	49	4A	4B
13	19	4C	4D	4E	4F
14	20	50	51	52	53
15	21	54	55	56	57
16	22	58	59	5A	5B
17	23	5C	5D	5E	5F

Hexadecimal Sector	Decimal Sector	Head 1			
		R1	F1	R2	F2
18	24	80	81	82	83
19	25	84	85	86	87
1A	26	88	89	8A	8B
1B	27	8C	8D	8E	8F
1C	28	90	91	92	93
1D	29	94	95	96	97
1E	30	98	99	9A	9B
1F	31	9C	9D	9E	9F
20	32	A0	A1	A2	A3
21	33	A4	A5	A6	A7
22	34	A8	A9	AA	AB
23	35	AC	AD	AE	AF
24	36	B0	B1	B2	B3
25	37	B4	B5	B6	B7
26	38	B8	B9	BA	BB
27	39	BC	BD	BE	BF
28	40	C0	C1	C2	C3
29	41	C4	C5	C6	C7
2A	42	C8	C9	CA	CB
2B	43	CC	CD	CE	CF
2C	44	D0	D1	D2	D3
2D	45	D4	D5	D6	D7
2E	46	D8	D9	DA	DB
2F	47	DC	DD	DE	DF

How to Lay Out Virtual Memory (Standard Precision)

Documentation required to lay out an execution-time disk dump of virtual memory is:

1. Execution-time disk dump of virtual memory.

   Note: Modifications to pages in core may not be reflected in the disk dump.

2. Maintenance utility core dump (all of core).
3. Maintenance utility dump of virtual memory (pseudo instructions).
4. Listing of the user's System/3 BASIC language program. (A copy of this can be obtained using the LIST system command.)
5. Assembly listing of #FMSTD.
6. Assembly listings of #TEQU1 and #TEQU2 (system equates).

The first step in laying out virtual memory is to block out the disk dump into the major areas illustrated in Figure 7-2. Lay out the following fixed areas first (these can be individually formatted by referring to the indicated figure or section in this manual):

1. Disk address 0700 (1 sector; starts at virtual address 0000; contains file directory 1); refer to Figure 5-17.
2. Disk address 0704 (1 sector; starts at virtual address 0100; contains file directory 2); refer to Figure 5-20.
3. Disk address 0708 (82 sectors; starts at virtual address 0200; contains fixed execution subroutines); refer to "Virtual Memory Resident Execution Subroutines—#FMSTD and #FMLNG" (Section 3).
4. Disk address 07B1 (2 sectors; starts at virtual address 5400, contains general purpose buffers).
5. Disk address 07B9 (starts at virtual address 5600, contains pseudo machine instructions); refer to "VM—Virtual Memory Dump Option" (Section 6). The last pseudo instruction will always be EOF.
6. Disk address 0911 (starts at virtual address F4FF; contains constants); refer to "Floating-Point Arithmetic" (Section 3) for arithmetic constants, and Figure 3-110 for character constants. Constants are generated at descending virtual addresses as they are encountered in the user's program.
7. Disk address 0915 (X'36' bytes; starts at virtual address F500 to F535; contains internal constants and internal work area—&CWRK and &WRK); refer to "Floating-Point Arithmetic" (Section 3) for internal constants. This area is generated by the loader (#LOADR); refer to "Loader—Second Phase of Compilation—#LOADR" (Section 3).
8. Disk address 0915 (starts at virtual address F536; contains variables); refer to "Floating-Point Arithmetic" (Section 3) for arithmetic variables, and Figure 3-110. for character variables. Variables are allocated at ascending virtual addresses as they are encountered in the user's program.
9. Disk address 093D (starts at virtual address FFFF; contains array dope vectors and virtual addresses of user function definitions); refer to Figure 3-156 for arithmetic array dope vectors, and Figure 3-157 for character array dope vectors. This area is allocated at descending virtual addresses as array references, user function references, and user function definitions are encountered in the user's program. The virtual address operands of the following pseudo instructions reference (Figure 7-3) this area:

   SA1

   SA2

   SB1

   SC1

   SF1

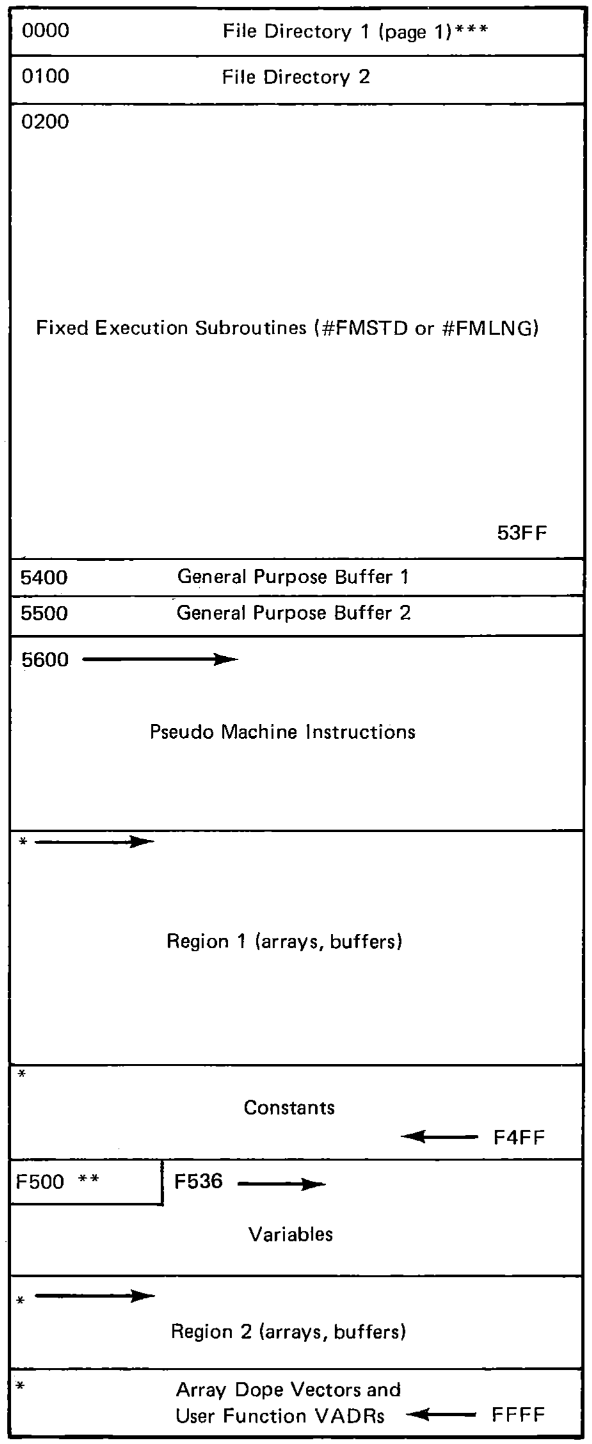
   SF2

   SD0

   SD1

   SD2

   FCI



Notes:

*The virtual addresses that define the limits of region 1 and region 2 are variable.

**F500 and all virtual addresses (constants, internal constants, variables, etc.) developed from it are precision dependent.

***Page 2 of file directory 1 is allocated in either region 1 or region 2.

Figure 7-2. Virtual Memory Map

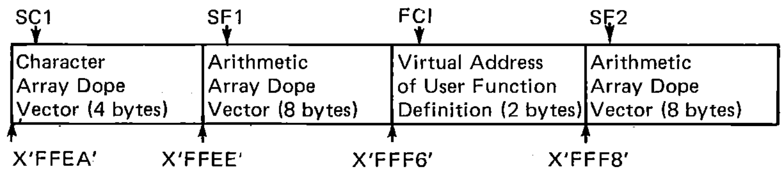


Figure 7-3. Pseudo Instruction Reference to Virtual Memory, Example

The next step in laying out virtual memory is to determine the virtual address limits of region 1 and region 2 (refer to Figure 7-2). Both regions start and end on page boundaries. The limits can be determined by inspecting virtual address operands in the generated pseudo instructions.

1. The starting virtual address of region 1 is the next ascending page following the last page of pseudo instructions. (Example: If the last pseudo instruction is generated at virtual address 5B4E, region 1 starts at 5C00.)
2. The ending virtual address of region 1 is the next descending page preceding the page containing the last generated constant. The virtual address of the last generated constant is determined by inspecting STF and STC pseudo instructions. The virtual addresses in the operands of constant stacking instructions descend from F500. All constants can be formatted by tracing this descending chain of virtual addresses in the generated pseudo instructions. (Example: If the virtual address of the last generated constant is F3F8, region 1 ends at F2FF.)
3. The starting virtual address of region 2 is the next ascending page following the page containing the last allocated variable. The virtual address of the last allocated variable is determined by inspecting STA, STF, and STC pseudo instructions. The virtual addresses in the operands of variable stacking instructions ascend from F536. All variable elements can be formatted by tracing this ascending chain of virtual addresses in the generated pseudo instructions. (Example: If the virtual address of the last allocated variable is F620, region 2 starts at F700.)
4. The ending virtual address of region 2 is the next descending page preceding the page containing the last allocated array dope vector or user function virtual address. These fields normally occupy only one page; therefore, region 2 normally ends at virtual address FEFF.

Alternate Method to Lay Out Virtual Memory (Standard Precision): The preceding virtual addresses are resolved by the compiler and passed to the loader in a common parameter area (Figure 3-155). This area can be inspected in a core dump taken between the execution of these two programs (refer to "CD–Core Dump Option" in Section 6).

The arrays can be formatted by inspecting the contents of the array dope vectors. Allocated buffers can be located by inspecting file directory 2 (page 01).

The virtual address operands of all FCI pseudo instructions should point to a location in virtual memory containing the virtual address of the corresponding user function definition in the generated pseudo instruction (virtual address of a BRA generated for a DEF statement). It is now possible to resolve that the virtual address operands of all generated pseudo instructions reference the correct data element or subroutine entry point.

How to Lay Out Virtual Memory (Long Precision)

Use the preceding method of laying out virtual memory for standard precision, keeping the following considerations in mind:

1. #FMLNG occupies the area starting at virtual address 0200 instead of #FMSTD.
2. All arithmetic data elements are allocated nine bytes instead of five. This includes constants, variables, array elements, and internal constants.
3. The virtual address that divides constants from variables is F000 instead of F500. All virtual addresses affected by this location must be adjusted. Also, the size of the area containing internal constants is increased to accommodate elements of greater length. The area containing variables starts immediately after internal constants and internal work area (virtual address F03F when running in long precision). (Page F0 is located at disk address 0901 in virtual memory.)

HOW TO LAY OUT AN EXECUTION-TIME CORE DUMP

Documentation required to determine the contents of an execution-time core dump is:

1. Maintenance utility core dump taken while the interpreter program is in execution. (A core dump taken while the interpreter is in an execution pause state does *not* contain the complete core-resident interpreter.)
2. Maintenance utility dump of virtual memory (pseudo instructions).
3. Listing of the user's System/3 BASIC language program. (A copy of this can be obtained by using the LIST system command.)
4. Assembly listings of #INSTD and #FMSTD, or listings of #INLNG and #FMLNG, depending upon the precision.
5. Assembly listings of #TEQU1 and #TEQU2 (system equates).

If the following conditions do not exist in the dump, it *is not* a valid execution-time core dump:

1. X'0600' *must* contain #INSTD or #INLNG.
2. X'0700' *must not* contain any valid program name (example: #DPRIN).
3. X'0C00' *must not* contain any valid program name (example: #GUFUD).
4. X'0E00' *may* contain the name of an interpreter execution overlay. The program name at this address, if present, *must be* #FISTD, #FILNG, or #SFFIN.
5. X'0F00' *may* contain the name of an interpreter execution overlay. The program name at this address, if present, *must be* #SFLOA.

The core dump can be divided into the major areas as illustrated in Figure 3-163 (interpreter core map). The location and size of certain areas in the dump are dependent upon the core size of the system. Figure 7-4 lists the fixed core addresses to be used in laying out the core dump (all addresses are in hexadecimal).

Address	Length	Description
0600	7 bytes	Program name and ID number
0607	50 bytes	Arithmetic function work area
0639	240 bytes	Run-time stack (contains variable length entries)
14CA	256 bytes	Core page table (nonzero entry indicates page in core)
1600		Start of core paging area (8k system only)
1FFF	10 pages	End of core paging area (8k system only)
1700		Start of core paging area (12k or 16k system)
28FF	18 pages	End of core paging area (12k system with CRT)
2FFF	25 pages	End of core paging area (12k system without CRT)
38FF	34 pages	End of core paging area (16k system with CRT)
3FFF	41 pages	End of core paging area (16k system without CRT)

BR1381

Figure 7-4. Fixed Core Addresses in Execution-Time Core Dump

MACHINE INSTRUCTION REFERENCE TABLE

Standard Mnemonics		
Instruction	Mnemonic Operation Code	
Zero and add zoned decimal	ZAZ	Two-address format*
Add zoned decimal	AZ	
Subtract zoned decimal	SZ	
Move hex character	MVX	
Move characters	MVC	
Compare logical characters	CLC	
Add logical characters	ALC	
Subtract logical characters	SLC	
Insert and test characters	ITC	
Edit	ED	
Move logical immediate	MVI	One-address format*
Compare logical immediate	CLI	
Set bits on masked	SBN	
Set bits off masked	SBF	
Test bits on masked	TBN	
Test bits off masked	TBF	
Store register	ST	
Load register	L	
Add to register	A	
Branch on condition	BC	
Test I/O and branch	TIO	
Sense I/O	SNS	
Load I/O	LIO	
Load address	LA	
Advance program level	APL	Command format*
Halt program level	HPL	
Start I/O	SIO	
Jump on condition	JC	

*See "Machine Instruction Formats" in this appendix.

BR1382

Extended Mnemonics	
Instruction	Mnemonic Operation Code
Move hex character (MVX):	
Move to zone from zone	MZZ
Move to numeric from zone	MNZ
Move to zone from numeric	MZN
Move to numeric from numeric	MNN
Branch on condition (BC):	
Branch	B
Branch high	BH
Branch low	BL
Branch equal	BE
Branch not high	BNH
Branch not low	BNL
Branch not equal	BNE
Branch overflow zoned	BOZ
Branch overflow logical	BOL
Branch no overflow zoned	BNOZ
Branch no overflow logical	BNOL
Branch true	BT
Branch false	BF
Branch plus	BP
Branch minus	BM
Branch zero	BZ
Branch not plus	BNP
Branch not minus	BNM
Branch not zero	BNZ
Jump on condition (JC):	
Jump	J
Jump high	JH
Jump low	JL
Jump equal	JE
Jump not high	JNH
Jump not low	JNL
Jump not equal	JNE
Jump overflow zoned	JOZ
Jump overflow logical	JOL
Jump no overflow zoned	JNOZ
Jump no overflow logical	JNOL
Jump true	JT
Jump false	JF
Jump plus	JP
Jump minus	JM
Jump zero	JZ
Jump not plus	JNP
Jump not minus	JNM
Jump not zero	JNZ

BR1383

MACHINE INSTRUCTION FORMATS

(A) Two-Address Formats

4 Bytes

OP Code	Length Count	Destination Address Displacement	Source Address Displacement
0 – 7	8 – 15	16 – 23	24 – 31

5 Bytes

OP Code	Length Count	Direct Destination Address	Source Address Displacement
0 – 7	8 – 15	16 – 31	32 – 39

OP Code	Length Count	Destination Address Displacement	Direct Source Address
0 – 7	8 – 15	16 – 23	24 – 39

6 Bytes

OP Code	Length Count	Direct Destination Address	Direct Source Address
0 – 7	8 – 15	16 – 31	32 – 47

(B) One-Address Formats

3 Bytes

OP Code	Q Code	Address Displacement
0 – 7	8 – 15	16 – 23

Q Code:
- Immediate Data
- Bit Mask
- Register Address
- Branch or Skip Condition
- Data Selection

Address:
- Destination Address
- Source Address
- Branch Address

4 Bytes

OP Code	Q Code	Direct Address
0 – 7	8 – 15	16 – 31

(C) Command Format

3 Bytes

OP Code	Q Code	Control Code
0 – 7	8 – 15	16 – 23

Q Code:
- Device Address and Functional Specifications
- Skip Condition
- Halt Identifier

BR1384

Operation Code

The first byte of each instruction, the operation code, specifies the addressing modes to be employed by the instruction in bits 0 through 3, and the operation to be performed in bits 4 through 7.

Q Code

The second byte of each instruction is the Q code. In two-address formats, the Q code is always a length count. In other formats, depending upon the operation specified, the Q code can be:

Length count
Immediate data
Bit mask
Register address
Data selection
Branch or skip condition
Device address and functional specifications

Control Code

The third byte of an instruction in the command format contains additional data pertaining to the command to be executed.

Storage Addresses

For instructions in the one-address and two-address formats, the third byte of the instruction and all bytes following are storage address information.

ASSEMBLER INSTRUCTION REFERENCE TABLE

Operation Entry	Name Entry	Operand Entry
DC	Any symbol or blank	One operand entry containing: duplication factor, type, length, constant.
DROP	Blank	Specified register (1 or 2).
DS	Any symbol or blank	One operand entry containing: duplication factor, type, length.
EJECT	Blank	Blank.
END	Blank	A relocatable expression.
EQU	Any symbol	An expression.
ICTL	Blank	Two decimals in the form of b, e.
ISEQ	Blank	Blank, or two decimal vaules in the form L, R.
ORG	Blank	Blank, or an expression (A) optionally followed by two absolute expressions in the form A, b, c.
PRINT	Blank	One or two entries from: DATA, NODATA; ON, OFF.
SPACE	Blank	Blank, or a decimal value.
START	Name	A self-defining value, or blank.
TITLE	Name or blank	A sequence of characters enclosed in apostrophes.
USING	Blank	A relocatable expression (v) and an index register (r) in the form v, r.

BR1385

Index

Where more than one page reference is given, the major reference is first.

LY34-0001-1

IBM System/3 BASIC (S3-65) Printed in U.S.A. LY34-0001-1

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

READER'S COMMENT FORM

IBM System/3 Model 6
System/3 BASIC Logic Manual

LY34-0001-1

- Your comments, accompanied by answers to the following questions, help us produce better publications for your use. If your answer to a question is "No" or requires qualification, please explain in the space provided below. Comments and suggestions become the property of IBM.

	Yes	No
• Does this publication meet your needs?	☐	☐
• Did you find the material:		
Easy to read and understand?	☐	☐
Organized for convenient use?	☐	☐
Complete?	☐	☐
Well illustrated?	☐	☐
Written for your technical level?	☐	☐

- What is your occupation? ____________________.
- How do you use this publication?

As an introduction to the subject?	☐	As an instructor in a class?	☐
For advanced knowledge of the subject?	☐	As a student in a class?	☐
For information about operating procedures?	☐	As a reference manual?	☐

Other ____________________.

- Please give specific page and line references with your comments when appropriate. If you wish a reply, be sure to include your name and address.

COMMENTS:

- **Thank you for your cooperation. No postage necessary if mailed in the U.S.A.**

YOUR COMMENTS, PLEASE. . .

Your answers to the questions on the back of this form, together with your comments, will help us produce better publications for your use. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Note: Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Fold Fold

— Cut Along Line —

FIRST CLASS
PERMIT NO. 110
BOCA RATON, FLA
33432

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
General Systems Division
Boca Raton, Florida 33432

Attention: Systems Publications, Department 707

Fold Fold

IBM System/3 BASIC (S3-65) Printed in U.S.A. LY34-0001-1

IBM®

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

READER'S COMMENT FORM

IBM System/3 Model 6
System/3 BASIC Logic Manual

LY34-0001-1

- Your comments, accompanied by answers to the following questions, help us produce better publications for your use. If your answer to a question is "No" or requires qualification, please explain in the space provided below. Comments and suggestions become the property of IBM.

	Yes	No
• Does this publication meet your needs?	☐	☐
• Did you find the material:		
Easy to read and understand?	☐	☐
Organized for convenient use?	☐	☐
Complete?	☐	☐
Well illustrated?	☐	☐
Written for your technical level?	☐	☐

- What is your occupation? ______________________________
- How do you use this publication?

As an introduction to the subject?	☐	As an instructor in a class?	☐
For advanced knowledge of the subject?	☐	As a student in a class?	☐
For information about operating procedures?	☐	As a reference manual?	☐

Other ______________________________

- Please give specific page and line references with your comments when appropriate. If you wish a reply, be sure to include your name and address.

COMMENTS:

- **Thank you for your cooperation. No postage necessary if mailed in the U.S.A.**

Cut Along Line

YOUR COMMENTS, PLEASE...

Your answers to the questions on the back of this form, together with your comments, will help us produce better publications for your use. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Note: Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Fold Fold

FIRST CLASS
PERMIT NO. 110
BOCA RATON, FLA
33432

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
General Systems Division
Boca Raton, Florida 33432

Attention: Systems Publications, Department 707

Fold Fold

IBM System/3 BASIC (S3-65) Printed in U.S.A. LY34-0001-1

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)