# Xerox Control Program for Real-Time (CP-R)

## Xerox 550 and Sigma 9 Computers

## System

## Technical Manual

XQ43, Rev. 0
90 30 88C

February 1977

© Xerox Corporation, 1974,1975,1976,1977

File No.: 1X13
Printed in U.S.A.

## REVISION

This publication is a major revision of the Xerox Control Program for Real-Time (CP-R)/System Technical Manual, Publication Number 90 30 888 (dated February 1975). This edition documents changes that reflect both the D00 and E00 versions of CP-R. A change in text from that of the previous manual is indicated by a vertical line in the margin of the page.

## RELATED PUBLICATIONS

| Title | Publication No. |
|---|---|
| Xerox 550 Computer/Reference Manual | 90 30 77 |
| Xerox Sigma 9 Computer/Reference Manual | 90 17 33 |
| Xerox Control Program for Real-Time (CP-R)/RT,BP Reference Manual | 90 30 85 |
| Xerox Control Program for Real-Time (CP-R)/OPS Reference Manual | 90 30 86 |
| Xerox Control Program for Real-Time (CP-R)/RT,BP User's Guide | 90 30 87 |
| Xerox CP-R Availability Features Reference Manual | 90 31 10 |
| Xerox Sigma Character-Oriented Communications Equipment/Reference Manual (Models 7611-7616/7620-7623) | 90 09 81 |
| Xerox Sigma Multipurpose Keyboard Display/Reference Manual (Models 7550/7555) | 90 09 82 |
| Xerox Mathematical Routines/Technical Manual | 90 09 06 |
| Xerox Assembly Program (AP)/LN,OPS Reference Manual | 90 30 00 |
| Xerox SL-1/Reference Manual | 90 16 76 |
| Xerox Extended FORTRAN IV/LN Reference Manual | 90 09 56 |
| Xerox Extended FORTRAN IV/OPS Reference Manual | 90 11 43 |
| Xerox Extended FORTRAN/Library Technical Manual | 90 15 24 |

Manual Content Codes: BP - batch processing, LN - language, OPS - operations, RP - remote processing, RT - real-time, SM - system management, TS - time-sharing, UT - utilities.

The specifications of the software system described in this publication are subject to change without notice. The availability or performance of some features may depend on a specific configuration of equipment such as additional tape units or larger memory. Customers should consult their Xerox sales representative for details.

# CONTENTS


PREFACE x

1. CP-R INITIALIZATION ROUTINE 1

2. CP-R CONTROL TASK 3

Structure 3
Function and Implementation 3
Resident Control Task 3
Key-In Processor 4
Load Module Control 4
Background Sequencing 5
Control Task Dump 6
Periodic Scheduler 6

3. I/O HANDLING METHODS 7

Channel Concept 7
Handling Devices 7
Single Interrupt Mode 7
Interrupt-to-Interrupt Mode 7
System Tables 7
IOQ 7
DCT 7
CIT 8
Handler Tables 8
DOT 8
CLST 8
I/O Control System Overview 9
Interfaces 9
Register Conventions 31
QUEUE 31
CALLSD 31
SERDEV 31
RIPOFF 31
STARTIO 32
CLEANUP/IOSCU 32
REQCOM 33
I/O Error Logging 33
I/O Statistics 33
Side Buffering 34
Output Side Buffering 34
Input Side Buffering 34
Virtual I/O Buffering 34
IOEX 37
Queued IOEX 37
Dedicated IOEX 37
Disk Pack Track-by-Track Logic 37
Disk Pack Seek Separation 37
Disk Pack Arm-Position Queue Optimization 37
Disk Angular-Position Queue Optimization 38
Deferred SIO 38
Logical Devices 38
User I/O Services 39
OPEN 39
CLOSE 39
READ/WRITE 39
PRINT 41
TYPE 41
DFM 41
DVF 41
DRC 41
DEVICE (Set Device) 41
DEVICE (Get Device) 41
CORRES 41
REWIND 41
WEOF 42
PREC 42
PFILE 42
ALLOT 43
DELETE 43
TRUNCATE 43

4. ERROR LOGGING 45

Error Log Record Formats 45

5. JOB CONTROL PROCESSOR 54

Overview 54
ASSIGN Command Processing 54
JCP Loader 70
Job Accounting 72
Background TEMP Area Allocation 73

6. FOREGROUND SERVICES 75

Implementation 75
RUN 75
RLS 75
MASTER/SLAVE 75
STOPIO/STARTIO 75
DEACTIVATE/ACTIVATE 76
IOEX 76
TRIGGER, DISABLE, ENABLE, ARM, DISARM, CONNECT, DISCONNECT 76
Task Control Block 76

7. MONITOR INTERNAL SERVICES 79

CP-R Overlays 79
Entry and Exit Point Inventory 81
Overlay Inventory 81
Event Control Block and Event Control Services 82
Overview of ECB Usage 82
ECB and Data-Area Formats 83
Dynamic Space (SPACE) 88
Dynamic-Space Service Calls 88
GETTEMP 88
RELTEMP 88
SYSGEN Considerations 88

# CONTENTS


PREFACE x

1. CP-R INITIALIZATION ROUTINE 1

2. CP-R CONTROL TASK 3

- Structure 3
- Function and Implementation 3
  - Resident Control Task 3
  - Key-In Processor 4
  - Load Module Control 4
  - Background Sequencing 5
  - Control Task Dump 6
  - Periodic Scheduler 6

3. I/O HANDLING METHODS 7

- Channel Concept 7
- Handling Devices 7
  - Single Interrupt Mode 7
  - Interrupt-to-Interrupt Mode 7
- System Tables 7
  - IOQ 7
  - DCT 7
  - CIT 8
- Handler Tables 8
  - DOT 8
  - CLST 8
- I/O Control System Overview 9
  - Interfaces 9
  - Register Conventions 31
    - QUEUE 31
    - CALLSD 31
    - SERDEV 31
    - RIPOFF 31
    - STARTIO 32
    - CLEANUP/IOSCU 32
    - REQCOM 33
- I/O Error Logging 33
- I/O Statistics 33
- Side Buffering 34
  - Output Side Buffering 34
  - Input Side Buffering 34
- Virtual I/O Buffering 34
- IOEX 37
  - Queued IOEX 37
  - Dedicated IOEX 37
- Disk Pack Track-by-Track Logic 37
- Disk Pack Seek Separation 37
- Disk Pack Arm-Position Queue Optimization 37
- Disk Angular-Position Queue Optimization 38
- Deferred SIO 38
- Logical Devices 38
- User I/O Services 39
  - OPEN 39
  - CLOSE 39
  - READ/WRITE 39
  - PRINT 41
  - TYPE 41
  - DFM 41
  - DVF 41
  - DRC 41
  - DEVICE (Set Device) 41
  - DEVICE (Get Device) 41
  - CORRES 41
  - REWIND 41
  - WEOF 42
  - PREC 42
  - PFILE 42
  - ALLOT 43
  - DELETE 43
  - TRUNCATE 43

4. ERROR LOGGING 45

- Error Log Record Formats 45

5. JOB CONTROL PROCESSOR 54

- Overview 54
- ASSIGN Command Processing 54
- JCP Loader 70
- Job Accounting 72
- Background TEMP Area Allocation 73

6. FOREGROUND SERVICES 75

- Implementation 75
  - RUN 75
  - RLS 75
  - MASTER/SLAVE 75
  - STOPIO/STARTIO 75
  - DEACTIVATE/ACTIVATE 76
  - IOEX 76
  - TRIGGER, DISABLE, ENABLE, ARM, DISARM, CONNECT, DISCONNECT 76
  - Task Control Block 76

7. MONITOR INTERNAL SERVICES 79

- CP-R Overlays 79
  - Entry and Exit Point Inventory 81
  - Overlay Inventory 81
- Event Control Block and Event Control Services 82
  - Overview of ECB Usage 82
  - ECB and Data-Area Formats 83
- Dynamic Space (SPACE) 88
  - Dynamic-Space Service Calls 88
    - GETTEMP 88
    - RELTEMP 88
  - SYSGEN Considerations 88

GAN 195
SCAN 195
GETIOID 195
GETFID, GETDEV, GETOPLB, GETANY 195
GETFSTSD 195
GETNXTSD 195
GETAX 196
GET1SFIL and GETNXFIL 196
UNPKDIRE 196
PACKDIRE 196
Control Commands 196
:ALLOT 196
:DELETE 196
:TRUNCATE 197
:SQUEEZE 197
Library File Maintenance 199
Library File Formats 200
Command Execution 202
:ALLOT 202
:COPY 202
:DELETE 202
:SQUEEZE 202
Bad Sector Handling 203
Command Execution 203
:BDSECTOR 203
:GDSECTOR 203
Utility Functions 203
:MAP 204
:LMAP 206
:SMAP 207
:CATALOG 207
:CLEAR 208
:COPY 208
:DPCOPY 208
:DUMP 208
:XDMP 209
Access Control Image 210
:SAVE 211
:RESTORE 212

12. TERMINAL JOB ENTRY 231

TJE COC Tables 231
TJE Commands 233
TJE Structure 234
Account Maintenance 234
TEX Operation 235
TEL Operation 235
Time Slicing 236

13. MEDIA 238

14. EDIT SUBSYSTEM 246

Functional Overview 246
Operational Overview 246
Module Analysis 250
BEGINEDITOR 250
MASTERPARSER1 251
MASTEREXECUTIVE Routine 265
Indexed Scratch File Management 317
Indexed File Structure 318

15. SYSTEM GENERATION 362

Overview 362
SYSGEN/SYSLOAD Flow 363
Loading Simulation Routines, CP-R and CP-R Overlays 363
Rebootable Deck Format 372
Stand-Alone SYSGEN Loader 373
SYSGEN LOADER LOADER 373

## APPENDIXES

A. CP-R SYSTEM FLAGS AND POINTERS 375

B. XEROX STANDARD OBJECT LANGUAGE 380

Introduction 380
General 380
Source Code Translation 380
Object Language Format 381
Record Control Information 381
Load Items 382
Declarations 382
Definitions 384
Expression Evaluation 385
Formation of Internal Symbol Tables 388
Loading 389
Miscellaneous Load Items 390
Object Module Example 390

C. XEROX STANDARD COMPRESSED LANGUAGE 396

D. SYSTEM OVERLAY ENTRY POINTS 397

## FIGURES

1. Initialize Routine Core Layout 1
2. CP-R Initialize Routine Overall Flow 2
3. Overall IOCS Organization 10
4. IOCS: QUEUE Routine 12
5. IOCS: SERDEV Routine 14
6. IOCS: CLOCKIO Routine 16
7. IOCS: RIPOFF Subroutine 17
8. IOCS: STARTIO Routine 18
9. IOCS: IOINT Routine 20
10. IOCS: IOALT Routine 22
11. IOCS: CLEANUP Routine 23
12. IOCS: REQCOM Routine 25
13. IOCS: ENDAC Subroutine 27
14. IOCS: IOERROR Subroutine 28
15. IOCS: IOLOG Subroutine 29
16. IOCS: PUSHLOG Subroutine 30
17. Logical Flow of ALLOT 44
18. Initialize JCP 55

19. Read and Process JCP Commands 56
20. Wait for JCP Command 57
21. Process JCP Command Errors 58
22. JOB Command Flow 59
23. FIN Command Flow 60
24. ASSIGN Command Flow 61
25. DAL Command Flow 62
26. ATTEND Command Flow 62
27. MESSAGE Command Flow 62
28. PAUSE Command Flow 63
29. CC Command Flow 63
30. LIMIT Command Flow 63
31. STDLB Command Flow 64
32. NAME Command Flow 65
33. RUN Command Flow 67
34. ROV Command Flow 67
35. INIT, SJOB, or BATCH Command Flow 67
36. ALLOBT Command Flow 68
37. PMD Command Flow 69
38. PFIL, PREC, SFIL, REWIND and UNLOAD Command Flows 69
39. WEOF Command Flow 69
40. Pre-PASS1 Core Layout 71
41. ARM, DISARM and CONNECT Function Flow 77
42. Arrangement of SYSLOAD Input ROMs 80
43. ECB Format and Chained Data Areas 84
44. Relationship of Task Controlled Data 120
45. Relationship between a Primary Task Control Block and Other Control Blocks 131
46. Relationship between Secondary Task Control Block and Other System Control Data 133
47. Relationship of AST to Other System Tables 137
48. Relationship of Job-Associated Control Tables 144
49. Enqueue/Dequeue Table Relationship 149
50. Overlay Structure of the Overlay Loader 155
51. Overlay Loader Core Layout 156
52. LIB Reorganization of Dynamic Table Area 169
53. PASSTWO Reorganization of Dynamic Table Area 172
54. MAP Table Reference 175
55. Program File Format 180
56. Overlay Loader Flow, OLOAD 184
57. Overlay Loader Flow, CCI 184
58. Overlay Loader Flow, PASSONE 185
59. Overlay Loader Flow, PASSTWO 188
60. Overlay Loader Flow, MAP 190
61. Overlay Loader Flow, RDIAG 191
62. Overlay Loader Flow, RDIAGX 191
63. Overlay Loader Flow, DIAG 192
64. RADEDIT Functional Flow 194
65. RADEDIT Flow, ALLOT 214
66. RADEDIT Flow, TRUNCATE 215
67. RADEDIT Flow, COPY 216
68. RADEDIT Flow, SQUEEZE 221
69. RADEDIT Flow, SAVE 226
70. Field and Indicator Definitions 244
71. Memory Allocation EDIT 246
72. Command Description Table (CDT) 247
73. Overall Flow Diagram of EDIT 248
74. Flow Diagram of MASTERPARSER 252
75. Flow Diagram of PARSE:I:CMNDSSTRG and PARSE:I:CMNDSINTG 259
76. Flow Diagram of MASTEREXECUTIVE 266
77. Flow Diagram of F:EDIT 270
78. Flow Diagram of F:END 272
79. Flow Diagram of SAVE Command 274
80. Flow Diagram of R:FINDSSEQUENCE, R:FINDSDELETE, and R:FINDSTYPE 276
81. Flow Diagram of R:MOVESDELETE and R:MOVESKEEP 280
82. Flow Diagram of GETNEXTNAME 298
83. Flow Diagram of GETNEXTPARAM 301
84. Flow Diagram of SHIFTRIGHT 312
85. Flow Diagram of OPENSCR 321
86. Flow Diagram of CLOSESCR 323
87. Flow Diagram of WRGRANS 325
88. Flow Diagram of OPENSCRI 327
89. Flow Diagram of READD 328
90. Flow Diagram of READX 330
91. Flow Diagram of FINDX 333
92. Flow Diagram of FINDNXX 335
93. Flow Diagram of GETX 336
94. Flow Diagram of GETREC 340
95. Flow Diagram of PUTREC 342
96. Flow Diagram of GETRBYTE/PUTRBYTE 345
97. Flow Diagram of DELETERECORD 346
98. Flow Diagram of WRITERANDOM 348
99. Flow Diagram of WRITENEWRANDOM 350
100. Flow Diagram of READRANDOM 351
101. Flow Diagram of READSEQUEN 353
102. Flow Diagram of BUILDSCR 355
103. Flow Diagram of SAVESCR 358
104. SYSGEN and SYSLOAD Layout before Execution 362
105. SYSGEN and SYSLOAD Layout after Execution 363
106. SYSGEN/SYSLOAD Flow 364

## TABLES

1. ASSIGN Table Description 70
2. RAD File Table Allocation for a Disk File 101
3. RAD File Table Allocation for a Block Tape 102
4. DCT Subtable Formats 103
5. IOQ Allocation and Initialization 108
6. Overlay Loader Segment Functions 155
7. T:DCBF Entries 176
8. Background Scratch Files 178
9. Command Number Table 249
10. Standard System Modules 369
A-1. CP-R System Flags and Printers 375
D-1. System Overlay Entry Points 397

# PREFACE

The primary purpose of this manual is to provide a guide for better comprehension of the program listings supplied with the Xerox Control Program for Real-Time (CP-R) operating system. The programs and processors included are the System Generation program and the Monitor and all its associated tasks and subprocessors.

The manual is intended for CP-R users who require an in-depth knowledge of the structure and internal functions of the operating system for system maintenance purposes. Since the CP-R System Technical Manual and program listings are complementary, it is recommended that the listings be readily available when referencing this manual. Manuals offering other levels of information regarding CP-R features are outlined below.

- Control Program for Real-Time/RT,BP Reference Manual, 90 30 85, is the principal source of reference information for the real-time and batch processing features of CP-R; (i.e., job control commands, system procedures, I/O procedures, program loading and execution, hardware interrupt and software interface, and service processors). The purpose of the manual is to define the rules for using background processing and real-time features.

- Control Program for Real-Time/OPS Reference Manual, 90 30 86, is the principal source of reference information for CP-R computer operators. It defines the rules for operator communication with the system (i.e., key-ins and messages), system start-up and initialization, job and system control, peripheral device handling, and recovery procedures.

- The Control Program for Real-Time/RT,BP User's Guide, 90 30 87, describes how to use the various batch and real-time features that are basic to most installations. It presents the information in a semitutorial format that offers the user a job-oriented approach toward learning the features of the operating system.

- CP-R Availability Features Reference Manual, 90 31 10, describes the available techniques to rapidly identify a system problem as either a hardware or software malfunction that has already occurred, or to anticipate a potential system alarm. It also describes the techniques to further define the problem via software diagnostic criteria, including the Error Log Lister (ELLA), ANALYZE processor, On-Line Exercisers, and system alarm procedures. The manual is primarily addressed to computer operators, local system programmers and analysts, and Xerox Customer Service personnel.

- Information for the language and applications processors that operate under CP-R is also described in separate manuals. These manuals are listed in the Related Publications page of this manual.

# 1. CP-R INITIALIZATION ROUTINE

The CP-R Initialize routine sets up core prior to the execution of CP-R. It is entered from the CP-R Bootstrap every time the system is booted from the disk. It also modifies the resident CP-R system (including all system tables), the CP-R overlays, and the Job Control Processor. Modifications may be made from the C or OC device that is selected by a corresponding sense switch setting SSW1-3. If sense switch 4 is reset, the Initialize routine loads all programs on the FP area of the disk designated as resident foreground. The Initialize routine may extend into the background and can be overwritten by background programs, since it executes only once. In Figure 1 below, the background first word address is the first page boundary after RBMEND (the end of resident CP-R). The Initialize routine terminates by entering the CP-R Control Task.

The general flow of the Initialize routine from CP-R Bootstrap entry to triggering the Control Task interrupt is illustrated in Figure 2.

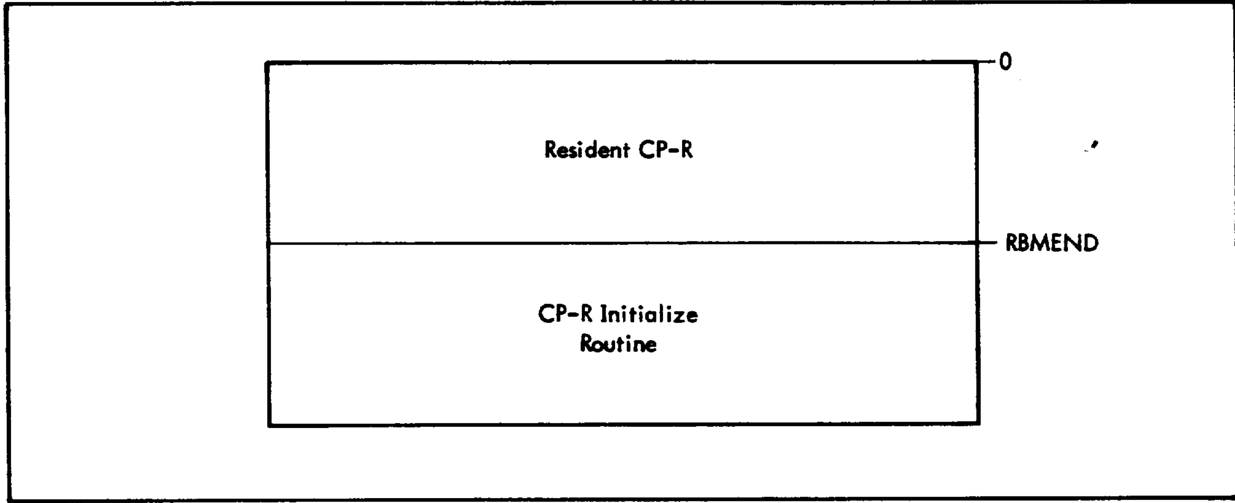



Figure 1. Initialize Routine Core Layout

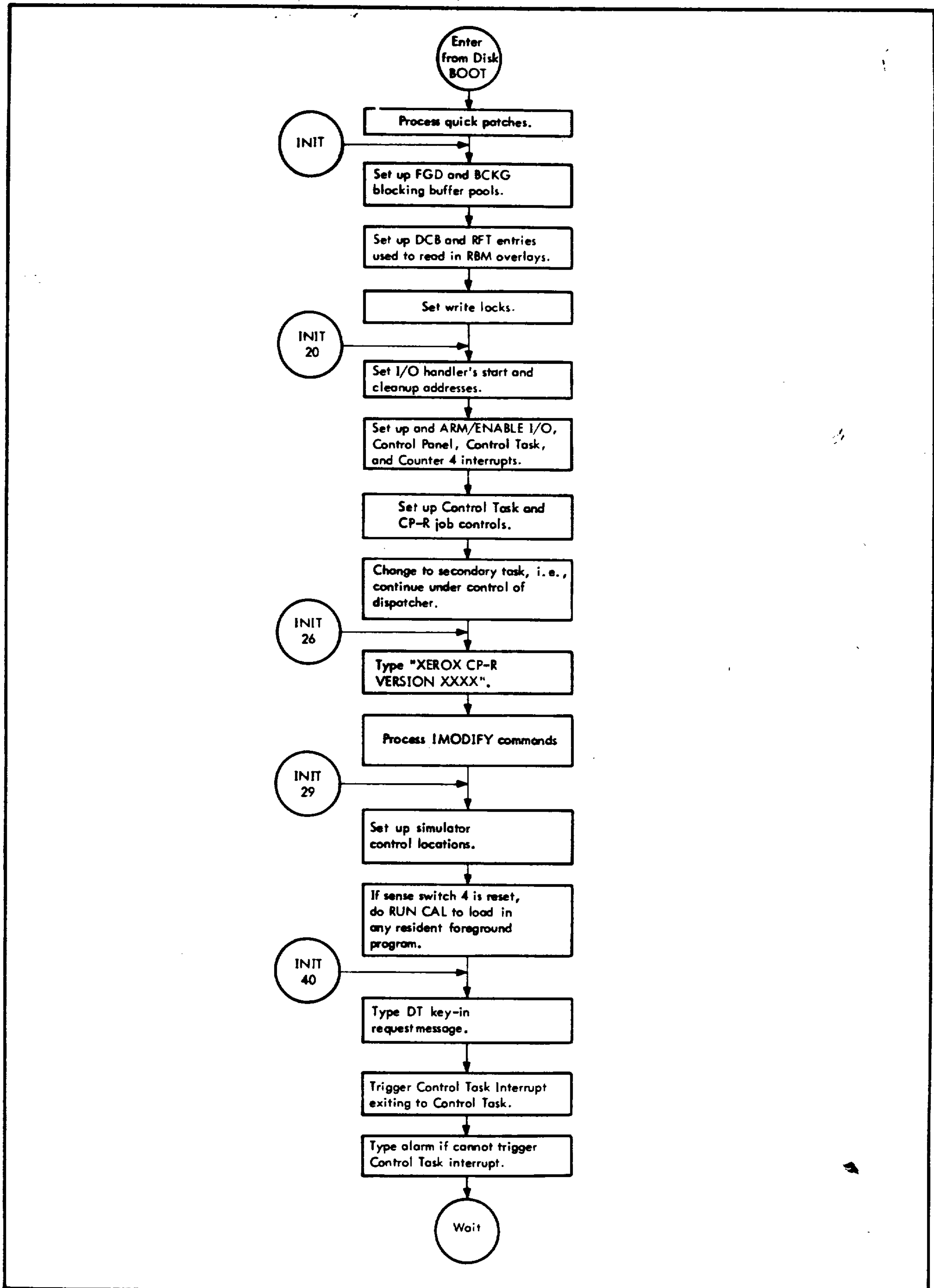



Figure 2. CP-R Initialize Routine Overall Flow

# 2. CP-R CONTROL TASK

The CP-R Control Task is connected to the lowest priority system interrupt. Among the functions performed by the Control Task are

- Key-in processing
- Primary program run
- Primary program release
- Background program load
- Background sequencing
- Background dump initiation
- Real memory dump
- Deferred I/O processing
- Periodic service of all devices
- Crash data handling
- I/O error log handling
- Periodic scheduling

In facilities where there are no system interrupts, the Control Task is connected to the Control Panel interrupt (see "Key-In Processor" later in this chapter).

## Structure

The Control Task consists of a resident portion and a number of monitor overlays. The overlays are

- Load module release (FGL1)
- Load module run (FGL2)
- Load module loader (FGL3)
- Background program initiation (BKL1)
- Background Abort/Exit/Dump (ABEX)
- Key-in Processor (KEY1-KEY7)
- Error logger (LOG)
- Error summary (ESUM)
- Crash saver (CRS, CRS2)
- Crash-save dumper (CRD)
- Direct crash dumper (CKD1, CKD2)
- Periodic scheduler (SCHED, SCMSG)

## Function and Implementation

### Resident Control Task

The resident portion of the Control Task functions as a scheduler for the various subtasks. The priority of the subtasks is determined by the order in which the resident Control Task tests the signal bits.

## Key-in Processor

When the Control Panel interrupt is triggered, its handler sets the flag in K:CTST to run KEY1 and triggers the interrupt for the Control Task dispatcher.

When KEY1 is entered, it determines whether an operator key-in must be read or has just been read. If the key-in has not yet been read, KEY1 prompts the OC device with a "-" and queues a read request to the same device. It then sets a flag indicating that key-in input is in process, and exits to the Control Task without clearing its run flag in K:CTST.

The combination of the flags mentioned forces the Control Task to skip KEY1 but to continue cycling through its scan until the key-in input is complete. It then enters KEY1.

When KEY1 is entered after a key-in has been read, it analyzes the input and branches to the appropriate processor in one of the key-in overlays. If the key-in is unrecognized, KEY1 outputs the message

!!KEY ERR

and repeats the attempt to read a key-in.

## Load Module Control (Formerly "Foreground Loader")

Load Module Control consists of three monitor overlays: FGL1 (Load Module Release), FGL2 (Load Module Run), and FGL3 (Load Module Loader). Monitor services that require a primary load module to be initiated or released set the appropriate status indicators in the LMI entry, set the flag for Load Module Control in K:CTST, and trigger the Control Task dispatcher interrupt.

Load Module Control is entered in the FGL1 overlay, which first searches the Load Module Inventory (LMI) for primary load modules to be released. If a releasable load module is found, FGL1 releases it. The System Task Inventory (STI) is searched for entries identifying tasks in the load module. If any are found, they are released, and the associated interrupts are disarmed and set to MTW,0 0. For clock-connected tasks, both the clock pulse and the corresponding count-equals-zero interrupts are treated. If the load module used PUBLIBs, their use counts are decremented, and the PUBLIB LMI entry is released if the use count becomes zero.

While searching for releasable primary load modules, FGL1 also finds all primary load modules that are waiting on memory in order to run ("run queued") and sets flags indicating that their loading is to be attempted again.

When all load module releases have been performed, FGL1 calls FGL2.

FGL2 searches the LMI for a primary load module entry flagged for loading. If the "run-queueing" option is not specified, the first loadable entry is selected. Otherwise, the loadable entry with the highest priority is chosen. (If there is none, FGL2 returns to the Control Task, clearing the Load Module Control flag from K:CTST.)

When an entry is found, the Job Program Table (JPT) for the job in which the load module will run is searched. If the task name from the LMI entry matches a task name in a JPT entry, the load module file name is provided by the JPT entry. If no such match is found, the task name is used as the file name. FGL2 calls FGL3 to load the load module. If FGL3 is successful, FGL2 sets up certain LMI entry values which are obtained from the load module header and allots Associative Enqueue Table (AET) space from the monitor's dynamic memory pool. The load module initialization sequence is executed. Normal completion posting is effected for the originating RUN or INIT request.

If FGL3 is unsuccessful at loading because the required memory was in use, FGL2 leaves the LMI entry for a later attempt at loading. If the load failed for another reason, the tables are deleted, and the originating request is posted as abnormally completed.

FGL3 acquires dynamic memory for load module header input. The header is read, and it is determined whether the memory between the program bounds is free of foreground programs. If it is not, the load terminates unsuccessfully. The root segments of the load module are read into their execution locations. If any PUBLIB is required, the LMI is

searched. If a PUBLIB is already loaded, its use count is incremented. If the required PUBLIB has no LMI entry, its header is read and its space requirement is determined. If the PUBLIB does not overlap an existing program or PUBLIB, the PUBLIB is loaded and given an LMI entry. If memory space is not available, the loading of the original program load module is terminated unsuccessfully.

When the root segments and PUBLIBs for a load module are all loaded, FGL3 returns successfully to FGL2.

### Background Sequencing

Background sequencing is provided by two monitor overlays: Background Program Initiation (BKL1, formerly "Background Loader part 1") and Background Abort/Exit (ABEX).

Background sequencing is begun by a "C" keyin received while the background is inactive. The key-in causes flags to be set in K:CTST indicating that BKL1 must run and the Job Control Processor (JCP) is to be loaded.

There are three main paths through BKL1: one for initiating JCP, one for initiating a processor or user program, and one for completing the initiation process after Load Module Control has loaded the background. BKL1 may also exit without doing anything, if it is entered without the indicator set for any of its three functions. In this case, the flag in K:CTST for BKL1 execution is cleared. At this point, background sequencing has terminated.

When BKL1 is called to initiate either JCP or another background program, the general process is to associate the task name "BKG" with the load module file name using a SETNAME CAL, and request task initiation with a no-wait INIT CAL. BKL1 then exits to the Control Task to allow the task initiation to proceed in the background context.

ASSIGNs are done as indicated in the ASSIGN table during task initiation.

The final path through BKL1 is taken after completion of the INIT service requested in either of the first two paths. Task initiation, on completing a background INIT request, sets the flag in K:CTST for BKL1 execution. When BKL1 is entered, it performs a CHECK on the INIT request. If an abnormal completion code is returned, flags are set to run ABEX to abort the background. BKL1 notifies the operator and exits. If the completion is normal, BKL1 then clears flags that block background execution and exits. Background can then run.

When a service requests that the background task be terminated (e.g., EXIT or ABORT CALs, trap processing abort), task termination is deferred. Instead, a flag is set in K:CTST indicating that ABEX must run, and another flag is set in K:JCP indicating whether the termination is an exit or an abort. The Control Task dispatcher is then triggered.

ABEX first determines what is to be run next in the background sequence on the basis of what was just run, and how it terminated. If a normal termination occurred, there are three alternatives: If a program other than JCP was running, ABEX indicates that JCP will run next. If JCP was running, and a !FIN command was received, nothing is to follow. If JCP was running and exited without !FIN, it was the result of some variety of !RUN command, and the next program to run is indicated by a file area and name in K:BAREA and K:BFILE, respectively. ABEX indicates that a user program is to be loaded next. If the previous background program aborted, ABEX indicates that JCP will run next. Additionally, ABEX sends an abort notification to the OC and LL devices, and sets a flag which forces JCP to skip control cards until a !JOB or !FIN is encountered.

If a postmortem dump is required, ABEX dispatches the background to the background dump routine, resets its own flag, and exits. When the dump is complete, an EXIT is executed, causing ABEX to be reentered. If there is no dump, or upon reentry after a dump, ABEX calls the TMLM monitor routine, which forces the background to execute termination. ABEX then exits, clearing its execution flag.

The background task then executes Task Termination, which closes files, waits out or stops I/O (the former in an EXIT, the latter in an ABORT), and releases table space. Termination ends by setting the K:CTST flag to run BKL1, and triggering the Control Task.

When BKL1 runs, as described earlier, it initiates the next load module, or, if there is none, terminates background sequencing.

**Control Task Dump**

A Control Task dump is requested by a DF or DM key-in. The required dump control block is obtained from the system dynamic space pool, and initialized for the required address range to be dumped to the DO oplabel. Also specified is a post-print routine address, to be executed after the Dump routine prints each line. The Dump routine is then called. It returns after performing the required dump. The Control Task dump flag is then reset, and the control task scheduling loop is reentered.

The post-print routine specified in the dump control block accomplishes two functions.

1. It causes a delay, followed by a retry if a device-manual error is returned from the print request.
2. It enters the Control Task scheduling loop in order to allow other Control Task functions to procede during the dump. (However, a DF or DM key-in will be rejected while a dump is executing.)

**Periodic Scheduler**

The Periodic Scheduler is divided into two major monitor overlay sections and two small sections of monitor root-resident code. The overlay sections are the CAL processor and the Scheduling Processor. The resident sections are in the clock 5-second timing code and an end-action receiver for an INIT call.

The CAL processor builds a chain of linked tspace entries with a monitor root word, SC:LIST, pointing to the oldest entry. These entries contain information required to create an appropriate INIT FPT as derived from the SCHED FPT. Upon entry, the CAL processor will create new TSPACE entries, delete entries if appropriate, set a bit in K:CTST, call CTRIG, and exit.

The Scheduling Processor, when called by the Control Task, resets the scheduler bit in K:CTST, opens the file SCHED in area SP, and examines the chained TSPACE entries pointed to by SC:LIST. If such entries exist, the SCHEDFIL entries will be created or deleted as appropriate. As the data in each TSPACE entry is entered in a SCHEDFIL record, that TSPACE is released and the chain pointer repositioned.

A monitor root word, SC:INIT, may point to a previously scheduled INIT FPT in TSPACE. If so, a CHECK call is issued. If the INIT is complete, its TSPACE is released and the completion status is analyzed. Any error code is added to the scheduled task's SCHED record and appropriate messages are output if this is the first time such status has been encountered. Some error codes cause automatic deletion of the entry.

If no INIT's are necessary, the Scheduling Processor examines the SCHEDFIL entries to see if the current time of day has exceeded the "next-time-to-run" value for a particular task. If a task is due to run, an INIT FPT is built in TSPACE and its address placed in SC:INIT. The task's next-time-to-run value in SCHEDFIL is incremented by its interval value and the INIT call is issued, with a 30-second timeout value to prevent blockage.

The Scheduling Processor then searches the SCHEDFIL for the lowest value of "next-time-to-run", which is posted in a root block of three words headed by SC:YEAR. The Scheduling Processor exits after resetting the K:CTST bit.

Logic in the 5-second count code of the CLOCK routine examines the three-word block SC:YEAR. If the current date/time has exceeded this value, the scheduler bit in K:CTST is set and CTRIG is called.

# 3. I/O HANDLING METHODS

## Channel Concept

A "channel" is defined as a data path, connecting one or more devices to memory, only one of which may be transmitting data (to or from memory) at any given time.

Thus, a magnetic tape controller connected to an MIOP is a channel but one connected to an SIOP is not, since in this case, the SIOP itself fits the definition. Other examples of channels are a card reader on an MIOP, a keyboard/printer on an MIOP, or a disk controller on an MIOP.

Input/output requests made on the system will be queued by channel to facilitate starting a new request on the channel when the previous one has completed. The single exception to this rule is the "off-line" type of operation, such as the rewinding of magnetic tape or the arm movement of certain moving arm devices. For this type of operation, an attempt is always made to also start a data transfer operation to keep the channel busy if possible.

## Handling Devices

The CP-R system offers the capability of multiple-step operations by providing an interrupt-to-interrupt mode in addition to the standard single interrupt mode.

### Single Interrupt Mode

On the lowest level the I/O handler is supplied a function code and device type. These coordinates are used to access information from tables used by the handler to construct the list of command doublewords necessary to perform the indicated operation. Included will be a dummy (nonexecuted) command containing information pertinent to device identification, recovery procedure, and follow-on operations (see below).

### Interrupt-to-Interrupt Mode

A function code for a follow-on operation may be included in the dummy command. This causes the request to be reactivated and resume its normal position in the channel queue, but with a different operation to be performed. It will be started by the scheduler in the normal manner as if it were any other request in the queue. The process may be cascaded indefinitely.

Error recovery may be specified at any point within a series of follow-on operations and will be itself treated by the system as a type of follow-on operation. It should be noted that follow-ons may be intermixed with other operations on the same channel or even on the same device if the situation warrants. Thus, a series of recovery tries on a disk may be interrupted to honor higher priority requests, or on a tape for higher priority requests on other drives (but not on the same drive).

## System Tables

Information pertaining to requests, devices, and channels is maintained in a series of parallel tables produced at System Generation time. A definition of these tables is presented here as reference for the remainder of this manual. The first entry (index=0) in each table is reserved for special use by the system. See Chapter 10 for a more complete description of these tables.

### IOQ (Request Information)

These tables contain all information necessary to perform an input/output operation on a device. When a request is made on the system, a queue entry is built that completely describes the request. The entry is then linked into the channel queue below other requests of either higher or the same priority.

### DCT (Device Control)

The Device Control Tables contain fixed information about each system device (unit level) and variable information about the operation currently being performed on the device.

### CIT (Channel Information)

These tables are used primarily to define the "head" and "tail" of entries that represent the queue for given channel at any time. A channel queue may have more than one entry active at any time (e.g., several tapes rewinding while another entry reads or writes).

## Handler Tables

Associated with each handler are two tables: the Device Offset Table (DOT), and the Command List Pointer Table (CLST).

### DOT (Device Offset Table)

The DOT table is a word table that begins on a doubleword boundary and contains:

Byte 0 — A byte offset from the beginning of the DOT table to the corresponding CLST entry.

Byte 1 — The time-out value, which is an integer that represents the number of five-second intervals that are allowed to pass between the SIO and the I/O interrupt before the interrupt is considered lost. The value X'FF' indicates the operation should not be timed out.

Byte 2 — The retry function code. This is the function code to be used for automatic error recovery.

Byte 3 — The continuation function code. This is the function code to be used for multiple interrupt requests. For example, a forward space record on magnetic tape can be performed n times by the basic I/O using the same queued request. Zero is used for no continuation.

The function code is used as the index to reference this table.

### CLST (Command List Pointer Table)

The CLST table is a byte table containing the doubleword displacement from the beginning of the corresponding DOT table to the appropriate skeletal command doubleword.

The general method for constructing the command doublewords for an I/O request is to access the DOT table using the function code as index, and then find the skeletal command doubleword offset by using the contents of byte 0 of the DOT entry as index to the CLST table. The skeletal command doubleword has the form

| Order | X | | |
|---|---|---|---|
| Flags | 0 | Y | Z |
| 0 7 | 8 | | 31 |

where

Y = 0 if the command is complete and to be used as is. This implies X is the address and Z is the byte count.

Y = 1 if a seek address contained in IOQ12 is to be placed in the first word. In this case, the value of X is irrelevant.

Y = 2 if a regular data transfer is to be performed. In this case, the buffer address is taken from IOQ8 and placed in the first word, and the byte count is taken from IOQ9 and placed in the second word (byte 1).

Y = 3 if the request represents an I/O error message. This will cause the proper N/LIIyyndd to be chained to the pointed message.

Y = 4 if a special handler function is to be performed. In this case, X is the address of the entry to the function.

When the building of the command doubleword is completed, a test is performed for command-chaining (command doubleword flag field bits 0 or 2 are on). If another command doubleword is to be chained, it is accomplished by accessing the next successive entry in the CLST table to find the offset of the skeletal command doubleword that is to be used to create the next command doubleword. This command doubleword is constructed in the same fashion as the first, and the process may continue to the limits imposed by the size of the command list area allocated at SYSGEN.

## I/O Control System Overview

The I/O Control System (IOCS) is based around three major concepts. They are device dependent variables, channel dependent variables, and request dependent variables. The device dependent variables include the device address, device state flags, pointers to channel and request variables, pointers to pre- and post-handlers and storage for hardware I/O status. The channels are software logical channels defined by the SYSGEN process. Only one data transmission can occur on a channel at any given time (two in the case of device pooling hardware). Channel variables include the state of the channel (busy, held, etc.) and queue head and tail pointers for the request queues. Request variables contain the information supplied by the IOCS user (file management, overlay manager, utility routines, etc.), indicating which I/O operation is to be performed and how completion is to be signaled. Request variables include buffer address, byte count, function code, maximum error retry count, end-action information, device pointer, priority, and others. There are also entries for forwards and backwards pointers in the channel queues.

All device-dependent code is in device pre- and post-handlers that are called before the I/O is started and after the I/O interrupt is received, respectively. They are dependent not only on the gross device type (i.e., card reader or magnetic tape unit), but also on the exact model of device and controller.

Figue 3 shows the overall organization of the IOCS.

### Interfaces

There are only two program interfaces into the IOCS. The first is QUEUE which is called with the request parameters in order to add a request to the proper queue. It identifies the proper channel and adds the entry in priority position. The second is SERDEV (Service Device) which, while called with a device pointer, identifies the associated channel and checks it for a possible state change.

The only interface out of the IOCS is IOSCU. When any I/O is finally terminated, IOSCU calls REQCOM which signals the requestor based on the clean-up code and/or end-action control word supplied with the original request.

The IOCS interfaces are described in further detail below, together with an I/O control sequence example for a simple case.

Figures 4 through 16 show the detailed control flow for the individual IOCS routines and subroutines.

#### Interfaces into the IOCS

QUEUE. This subroutine is called by the monitor to enter an I/O request into the IOCS. It must be supplied with many parameters such as:

- Byte address of the buffer
- Byte count
- Logical function code (read, write, rewind, etc.)
- Priority
- Device ID
- End-action control data
- Maximum number of recovery attempts

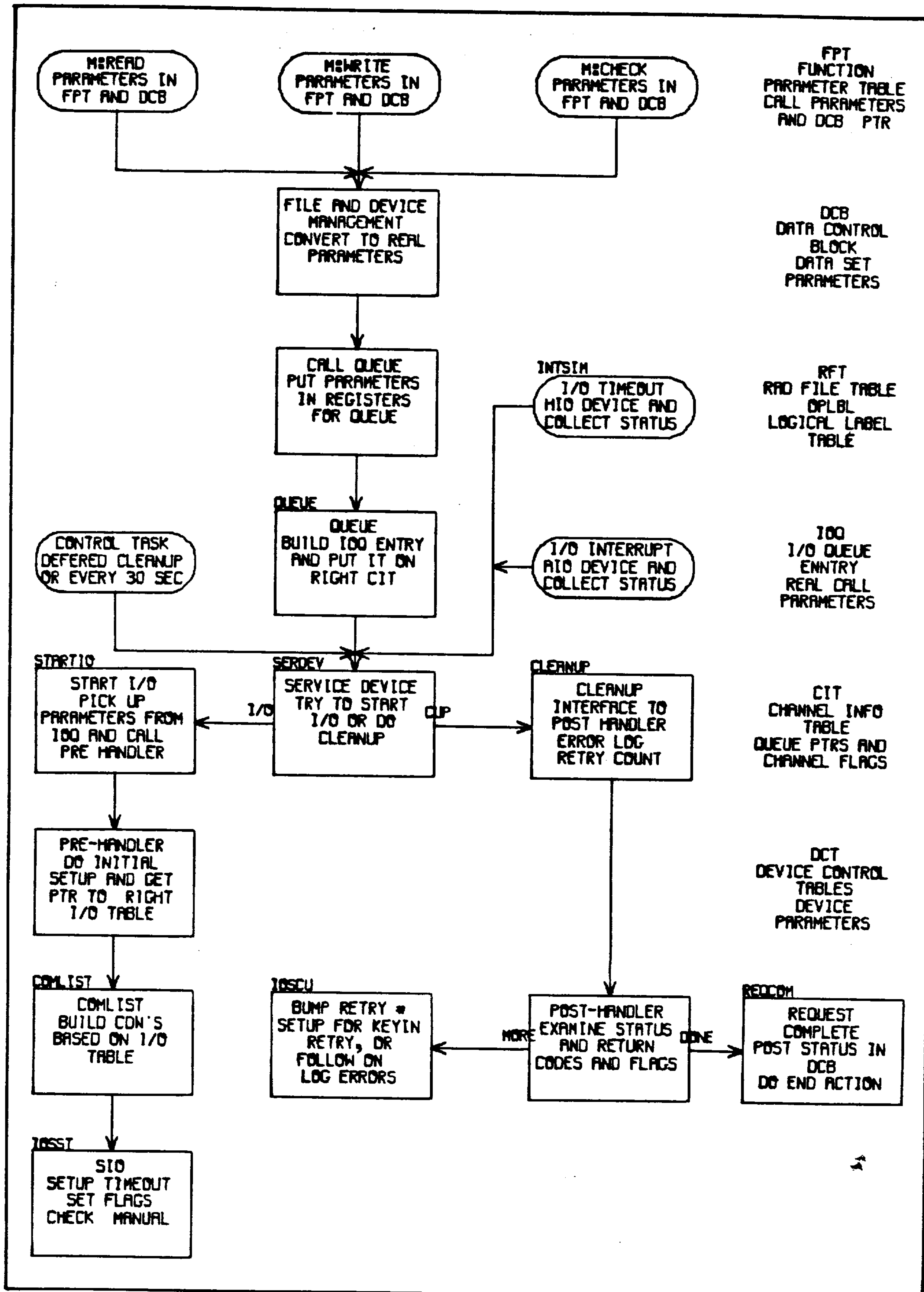



Figure 3. Overall IOCS Organization

Interface out of the IOCS

IOSCU. This routine, when final completion of an I/O request occurs, can signal that completion in two ways:

- A post word may be posted with the Actual Record Size (ARS) and type-of-completion (TYC) code.
- A monitor subroutine may be entered with the ARS, type-of-completion code and user end-action information in registers.

IOCS Control Sequence/Example

The sequence followed when a single I/O request is made to IOCS for an idle channel is as follows:

1. The monitor makes a call on QUEUE with the request parameters. QUEUE places the request on the proper channel queue in the proper priority order.
2. The monitor calls SERDEV to start the channel.
3. SERDEV finds the channel idle and a startable entry in the queue. It calls STARTIO for that queue entry.
4. STARTIO calls a device dependent pre-handler which builds the proper channel program based on the queue entries. The I/O is started on the device and STARTIO returns through SERDEV to the monitor.
5. While the I/O is proceeding, the task for which the I/O is being done may get blocked and be waiting for the I/O to complete. The monitor then makes successive calls on SERDEV while it is waiting for the task. If SERDEV finds the device busy, it checks the elapsed time for the I/O in progress to see if it is taking too long.

   (SERDEV is also called every 30 seconds for all devices. This makes sure that the system does not hang up.)
6. When the I/O operation completes, or errors, an I/O interrupt is generated. IOINT is entered.
7. IOINT collects all the status about the I/O operation and marks the device as needing clean-up. IOINT then either calls SERDEV itself or stacks the device ID and triggers another interrupt level which will call SERDEV for all the device IDs in the stack.
8. SERDEV finds the channel blocked by a device requiring clean-up and thus calls IOSCU.
9. IOSCU calls a device-dependent post-handler which analyzes the status saved by IOINT. The post-handler returns to IOSCU with parameters indicating what action to take. The possibilities are:

   Output an operator message.

   Request an operator key-in.

   Follow-on to a new function.

   Decrement the retry count.

   Post some type of completion code.
10. IOSCU then re-enters SERDEV in order to get the channel started again (step 3).
11. This sequence goes on, round and round, until some type of I/O completion is posted.

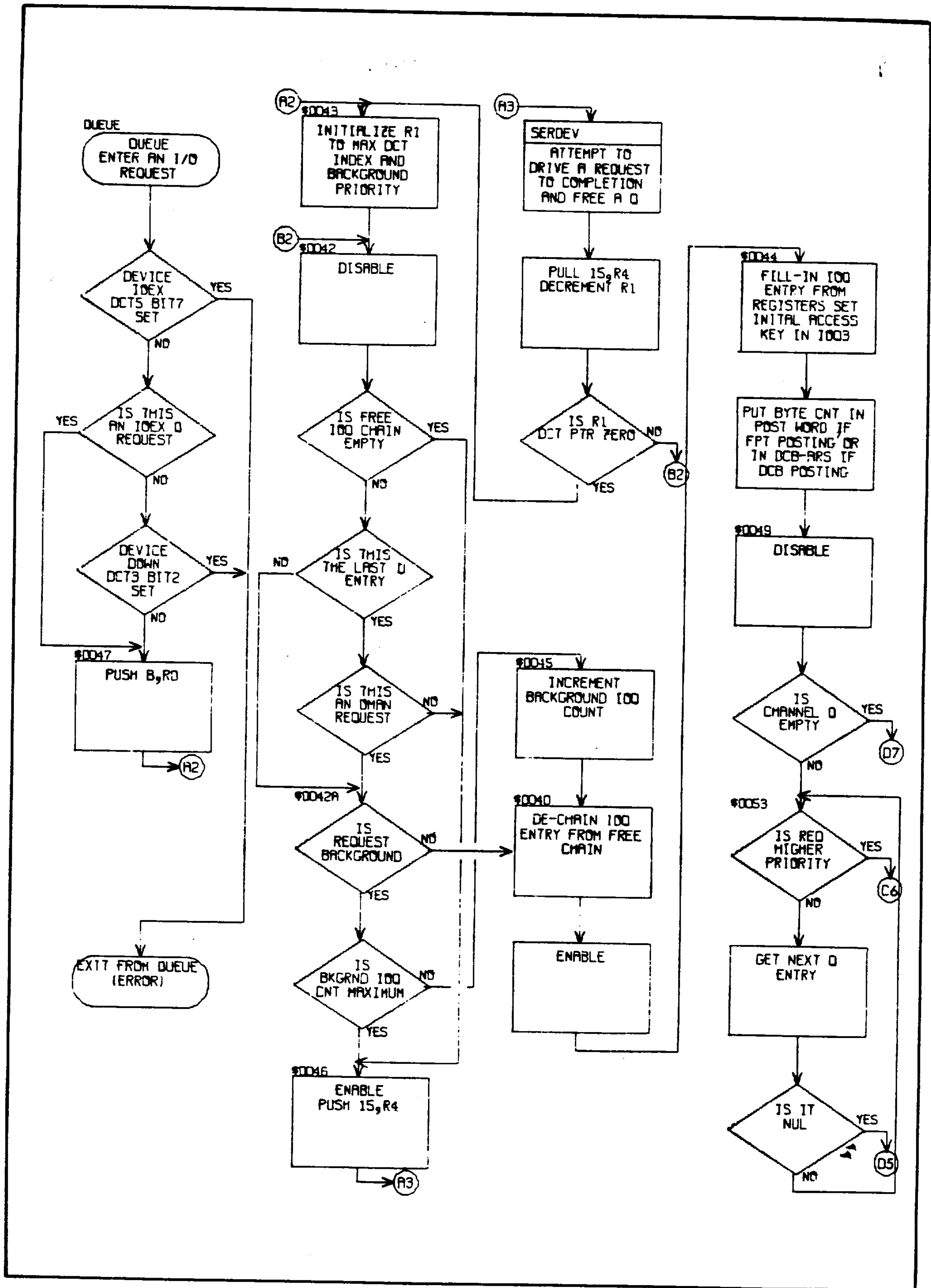



Figure 4. IOCS: QUEUE Routine

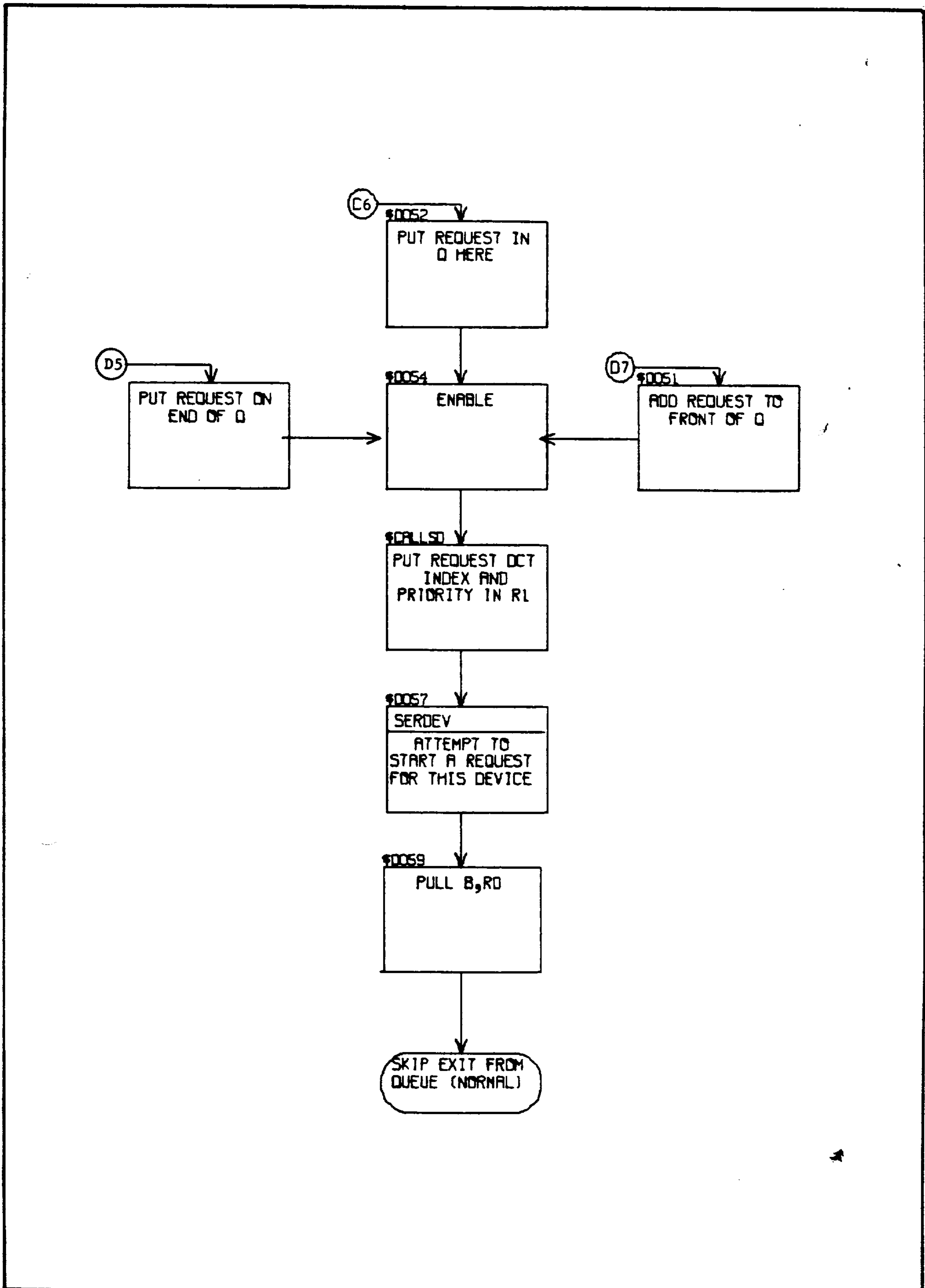



Figure 4. IOCS: QUEUE Routine (cont.)

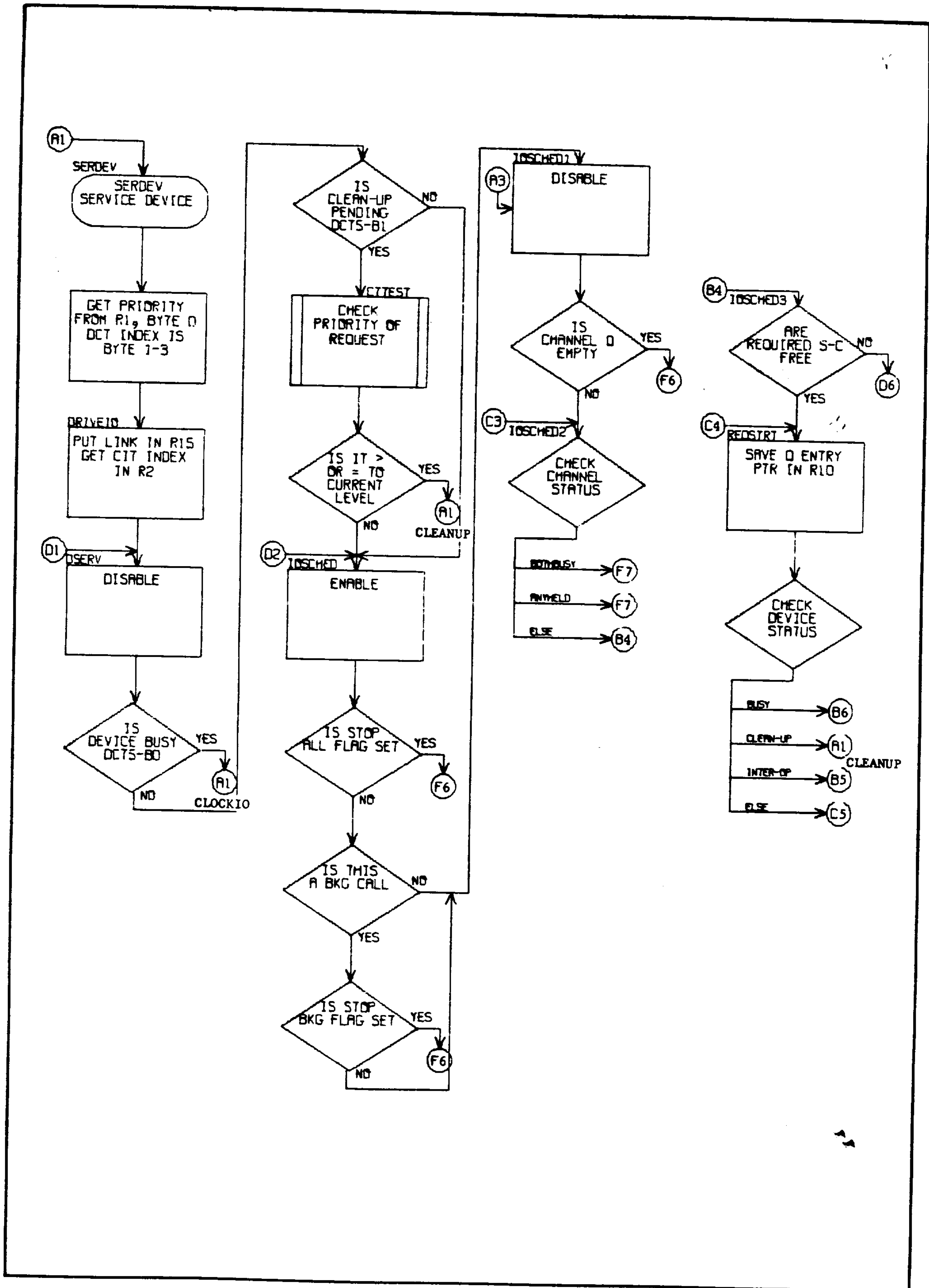



Figure 5. IOCS: SERDEV Routine

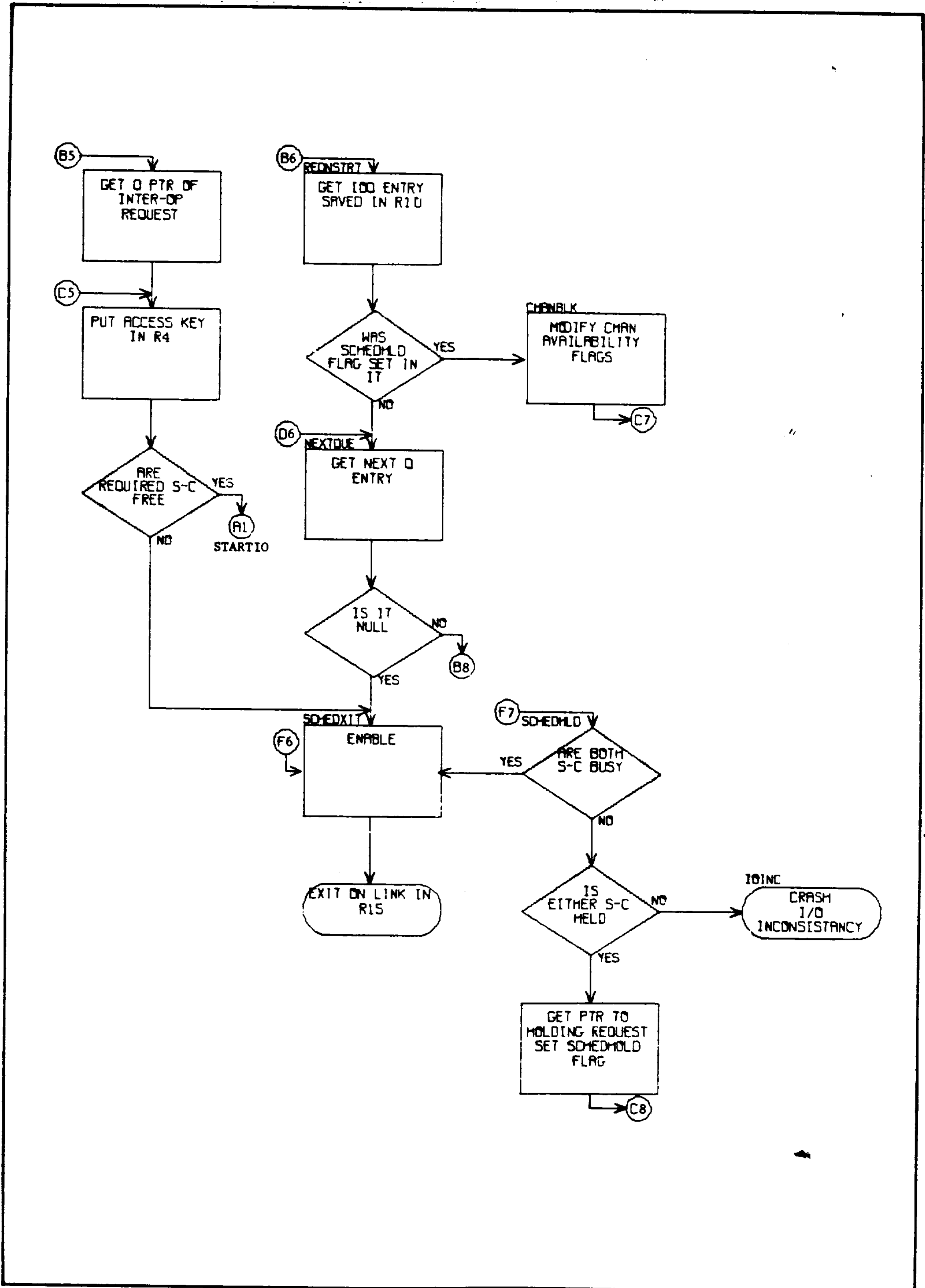



Figure 5. IOCS: SERDEV Routine (cont.)

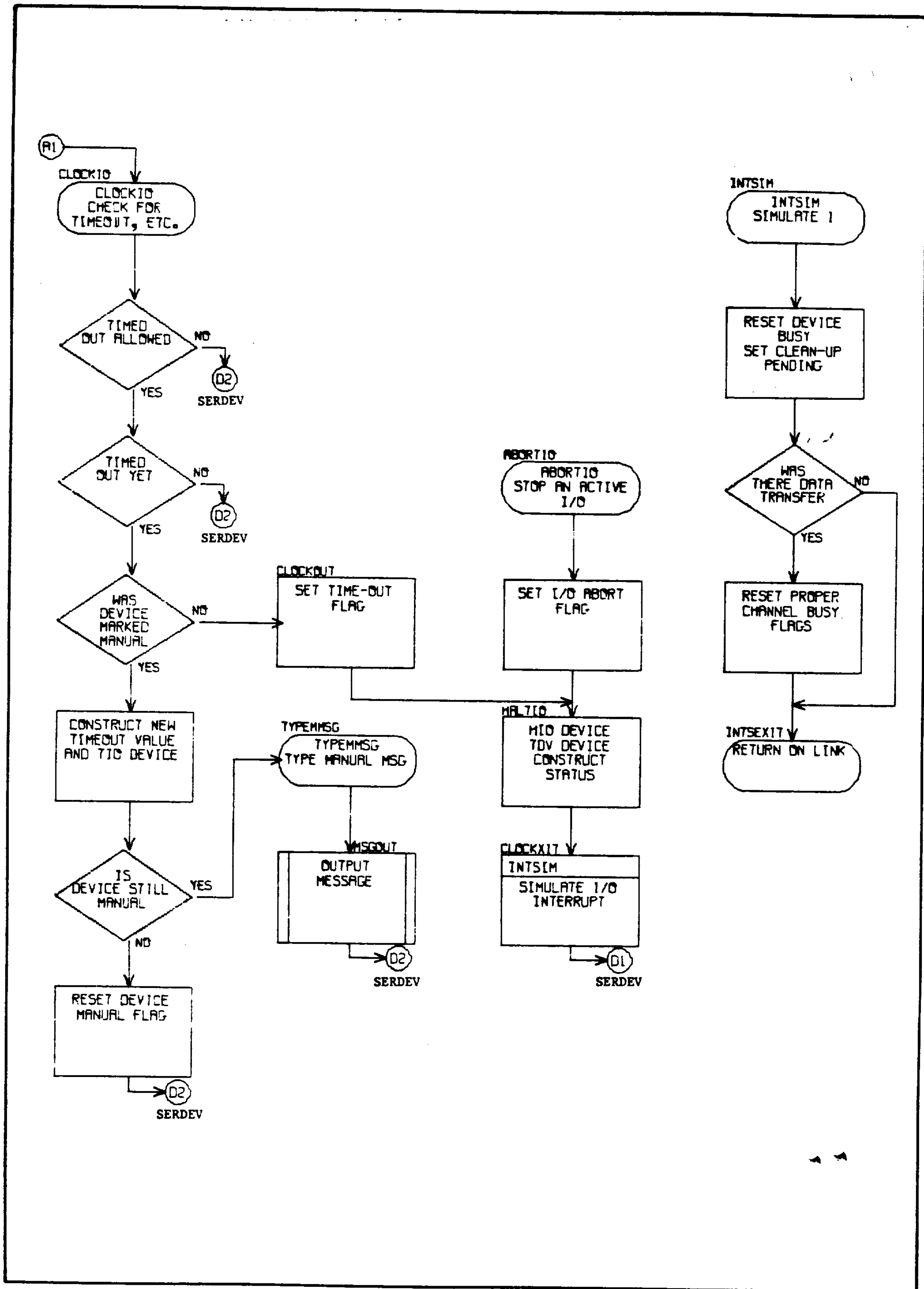



Figure 6. IOCS: CLOCKIO Routine

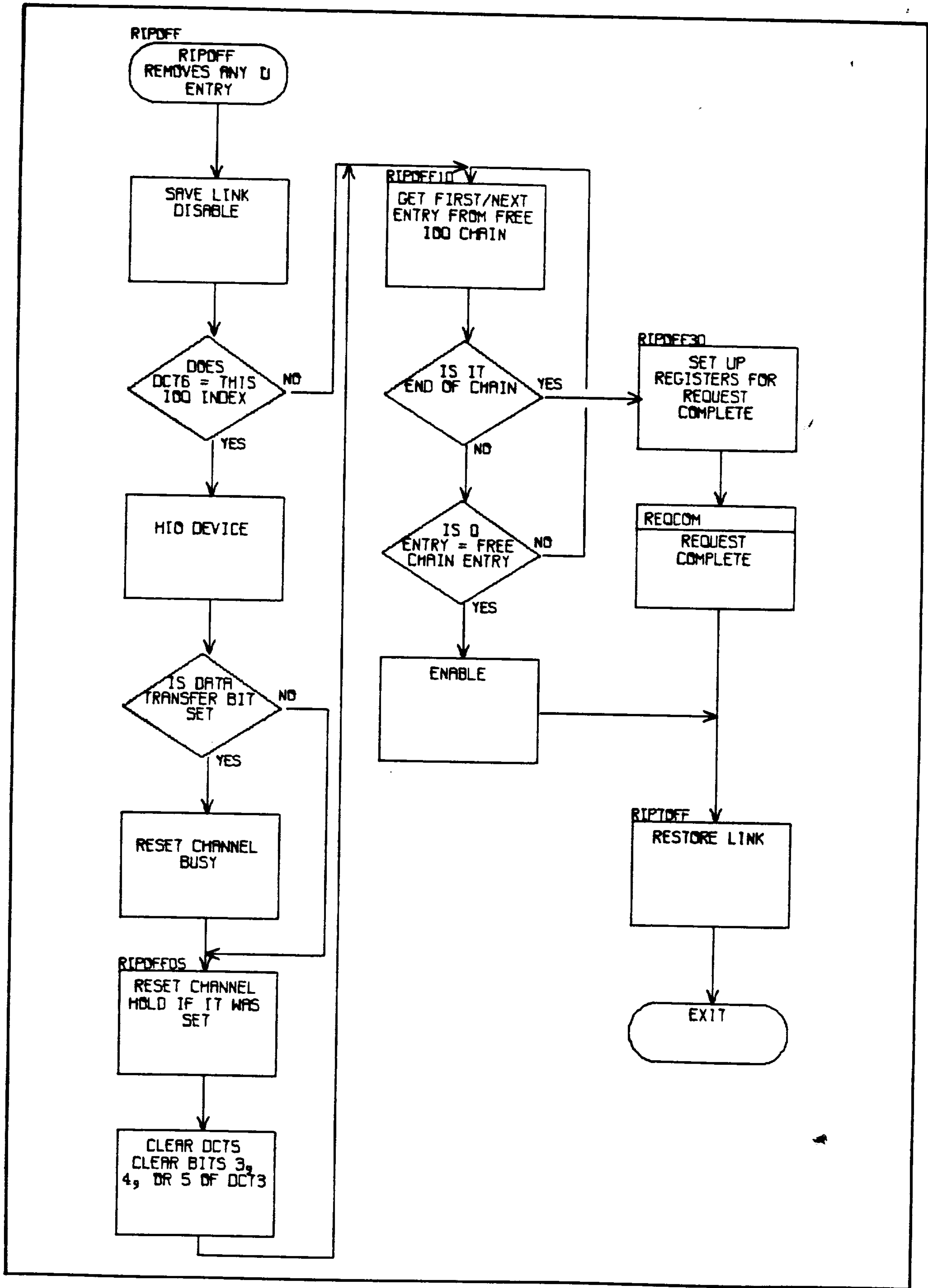



Figure 7. IOCS: RIPOFF Subroutine

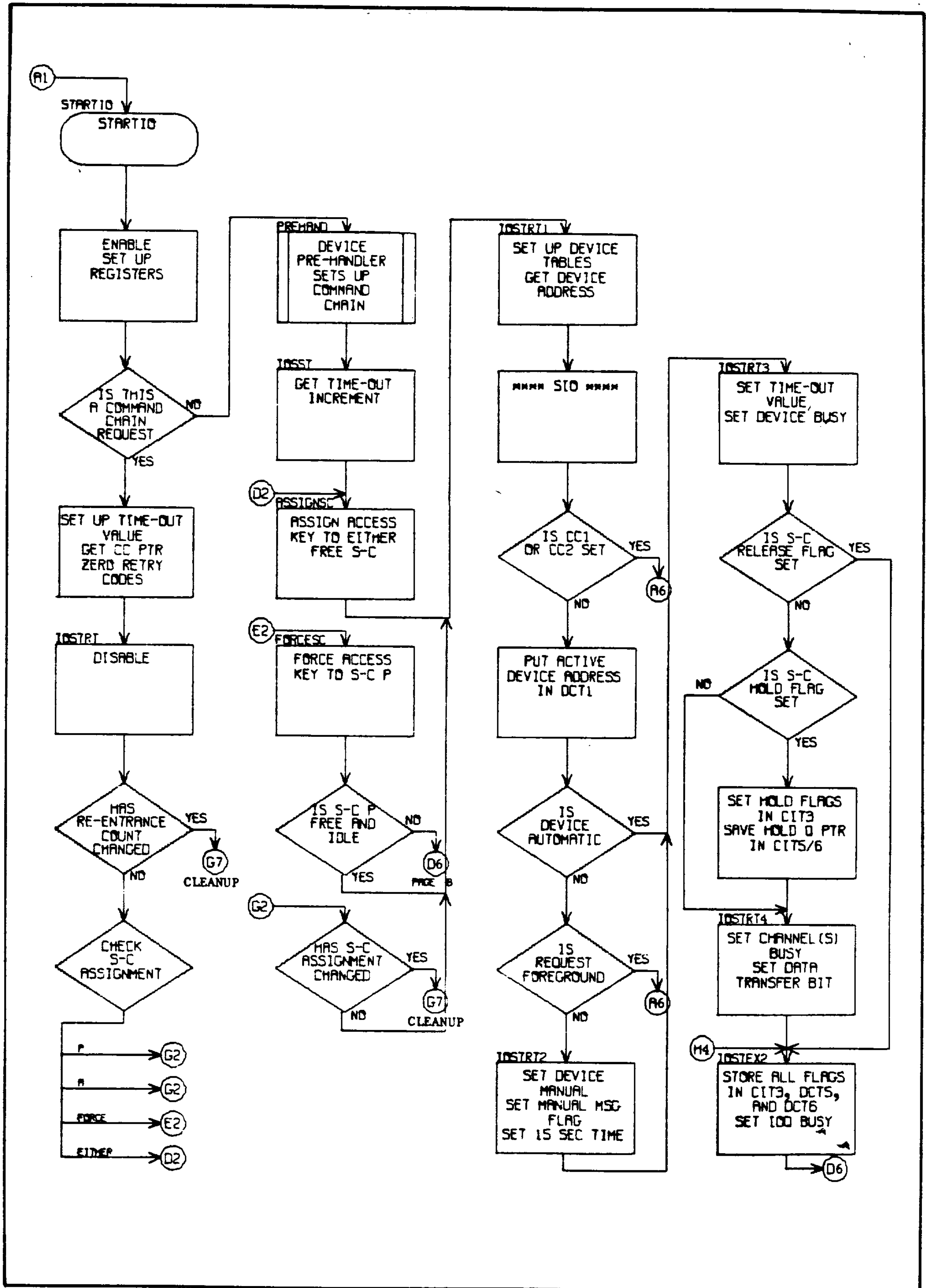



**Figure 8. IOCS: STARTIO Routine**

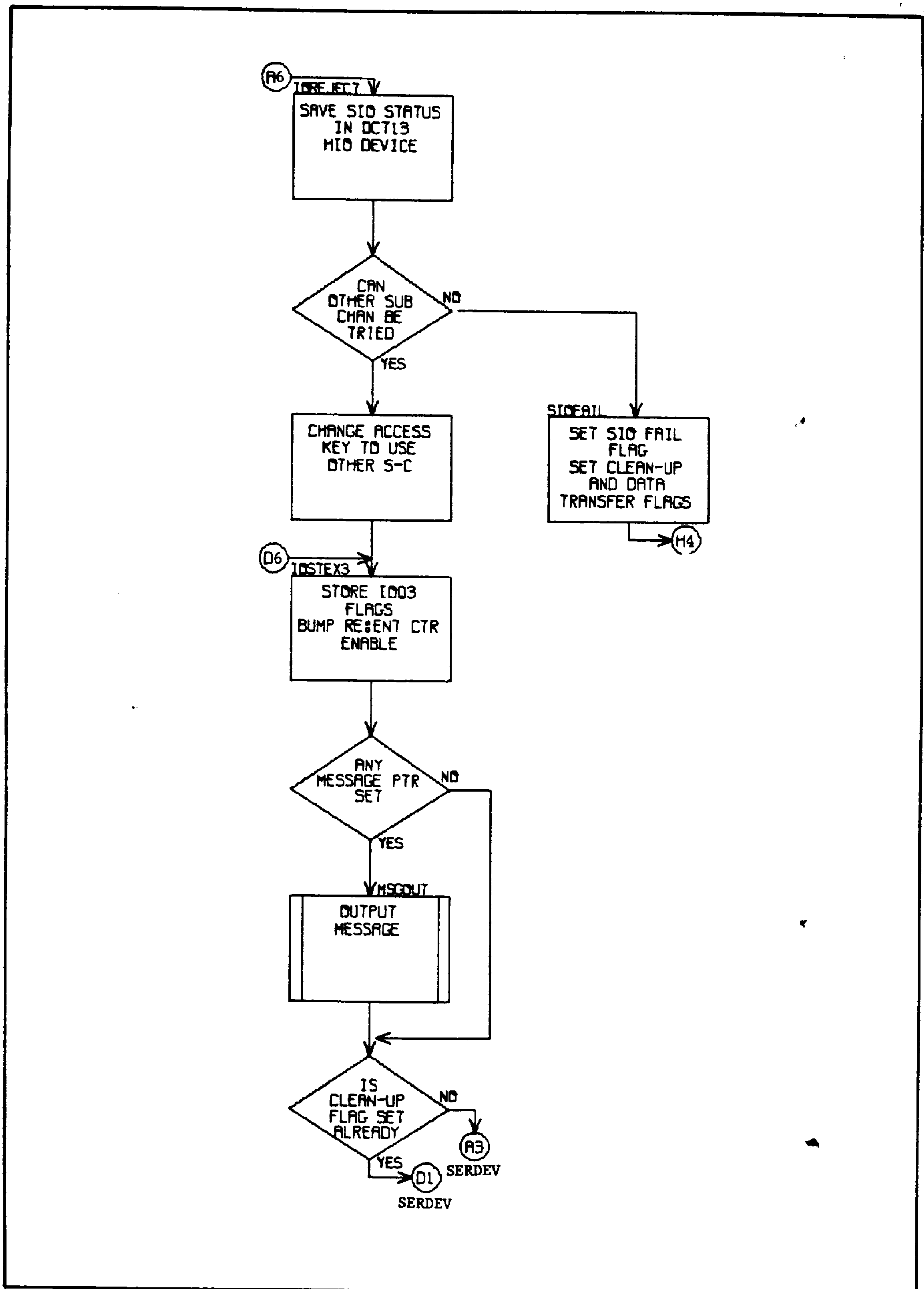



Figure 8. IOCS: STARTIO Routine (cont.)

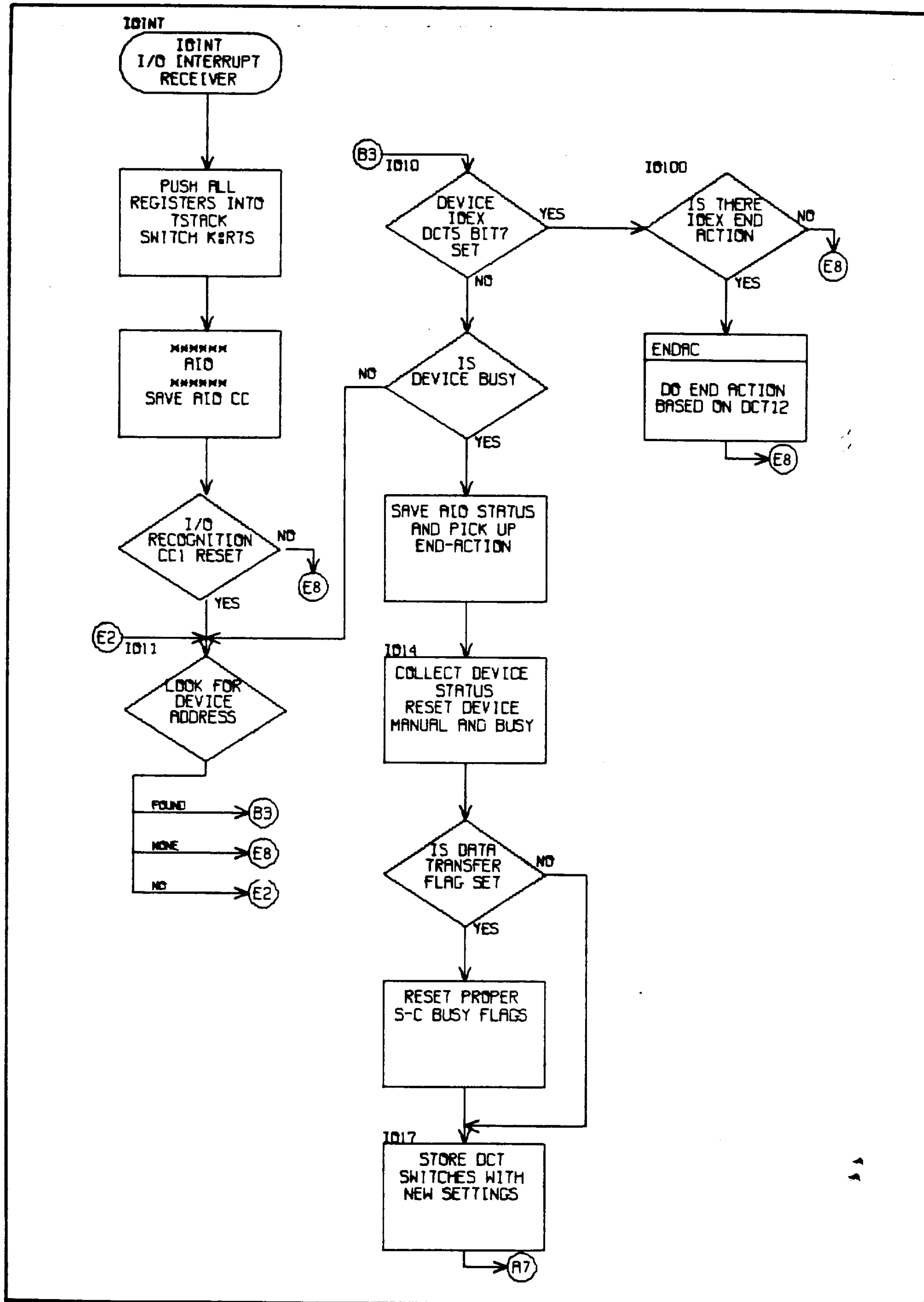



Figure 9. IOCS: IOINT Routine

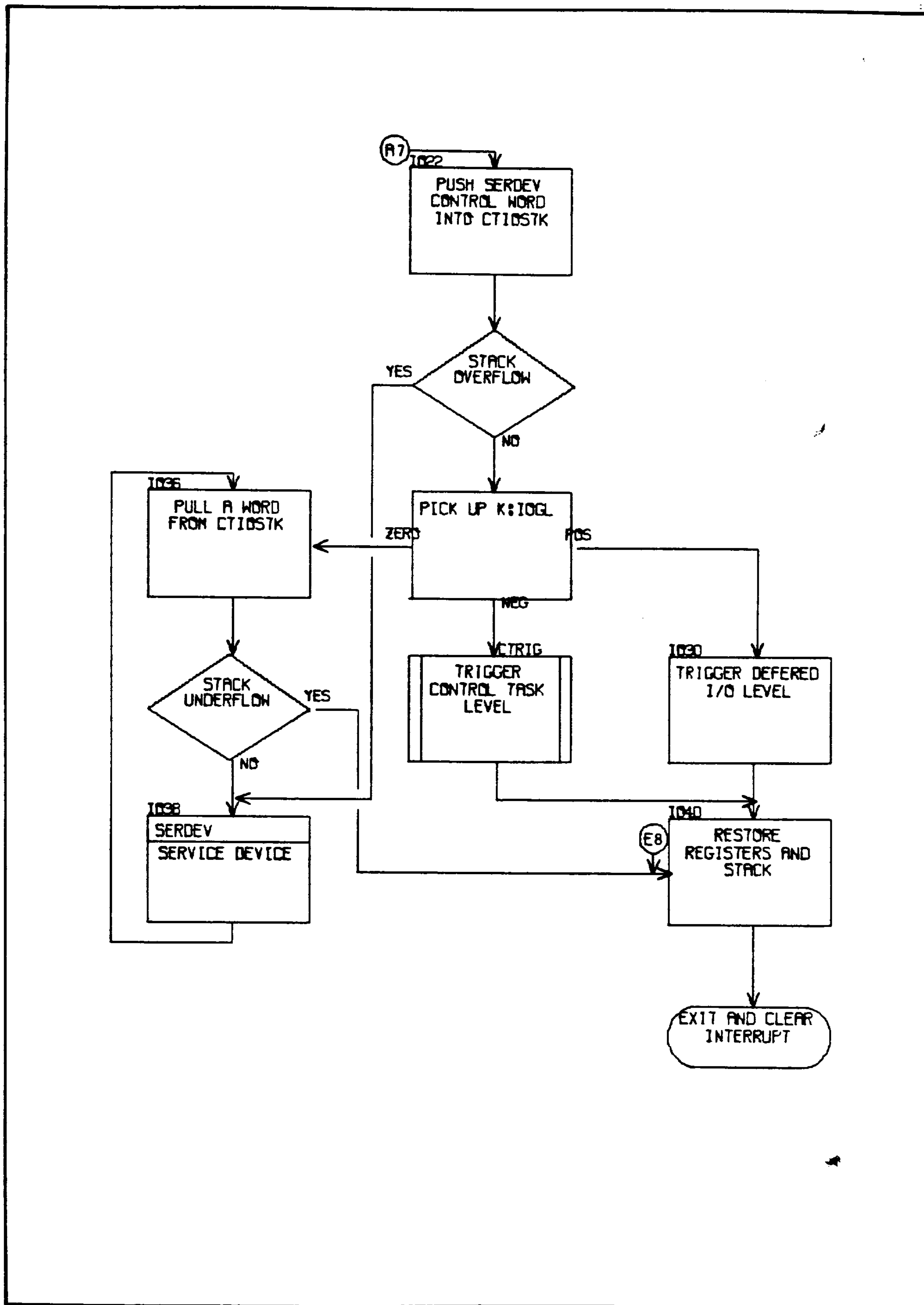



Figure 9. IOCS: IOINT Routine (cont.)

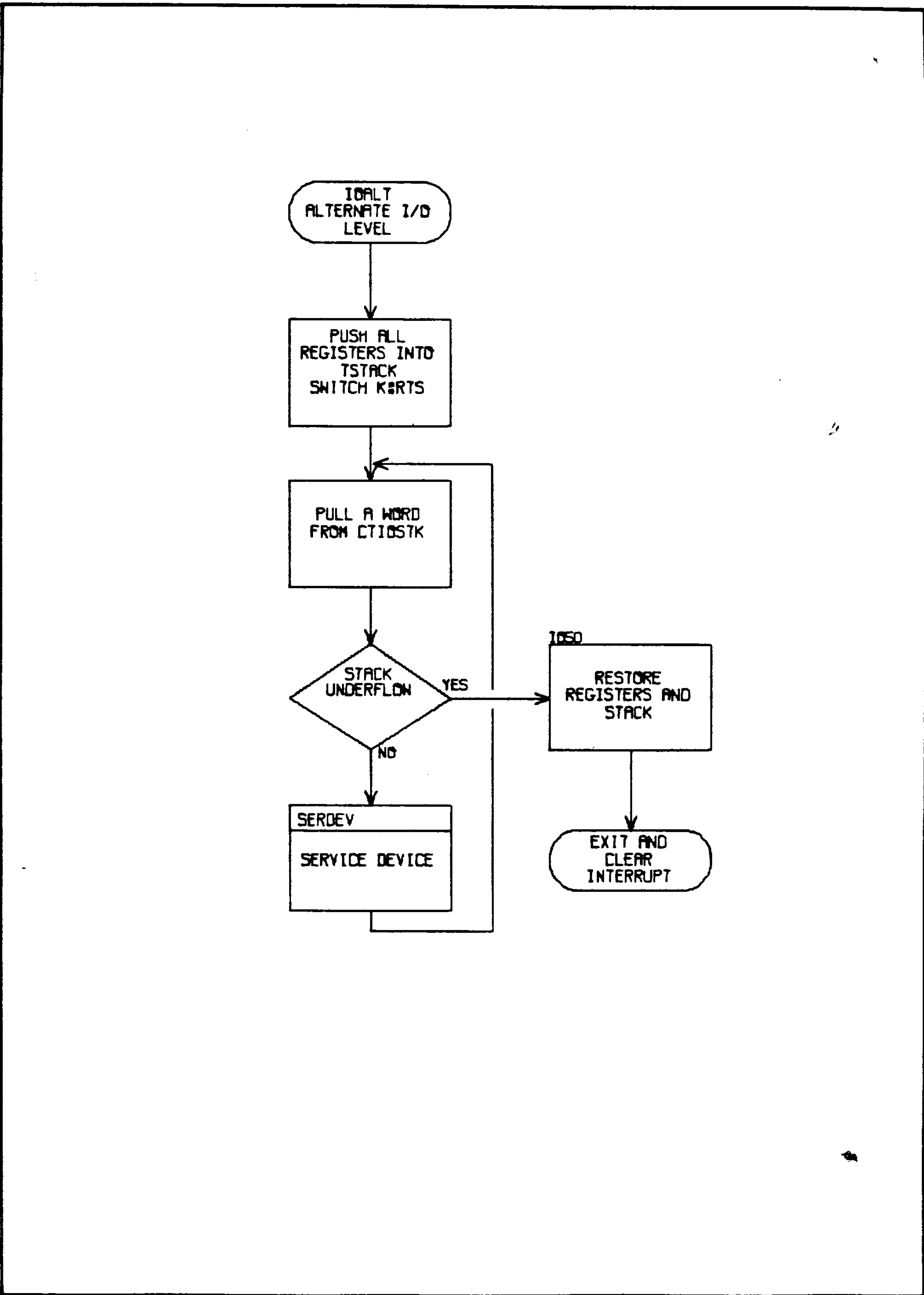



Figure 10. IOCS: IOALT Routine

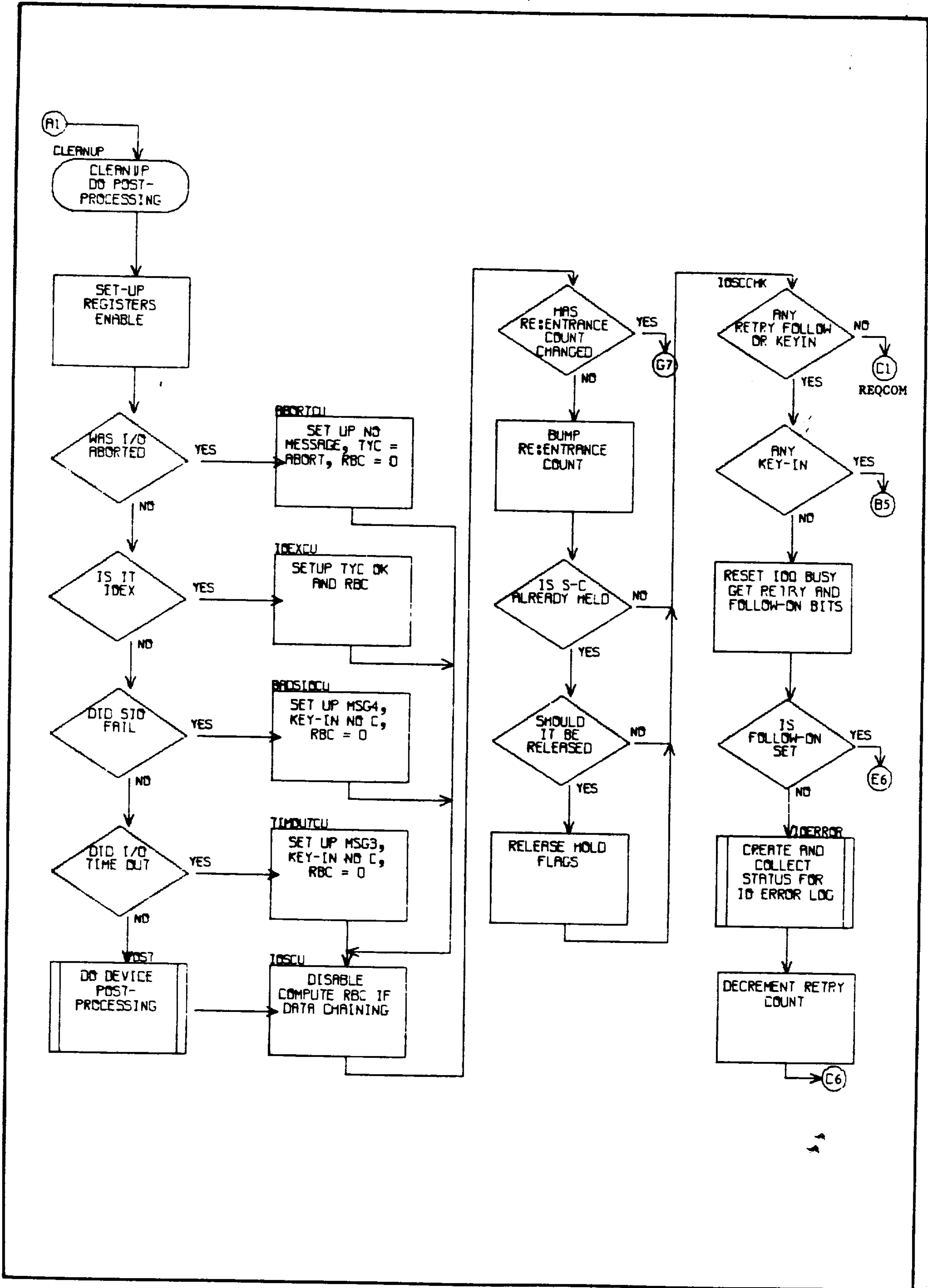



Figure 11. IOCS: CLEANUP Routine

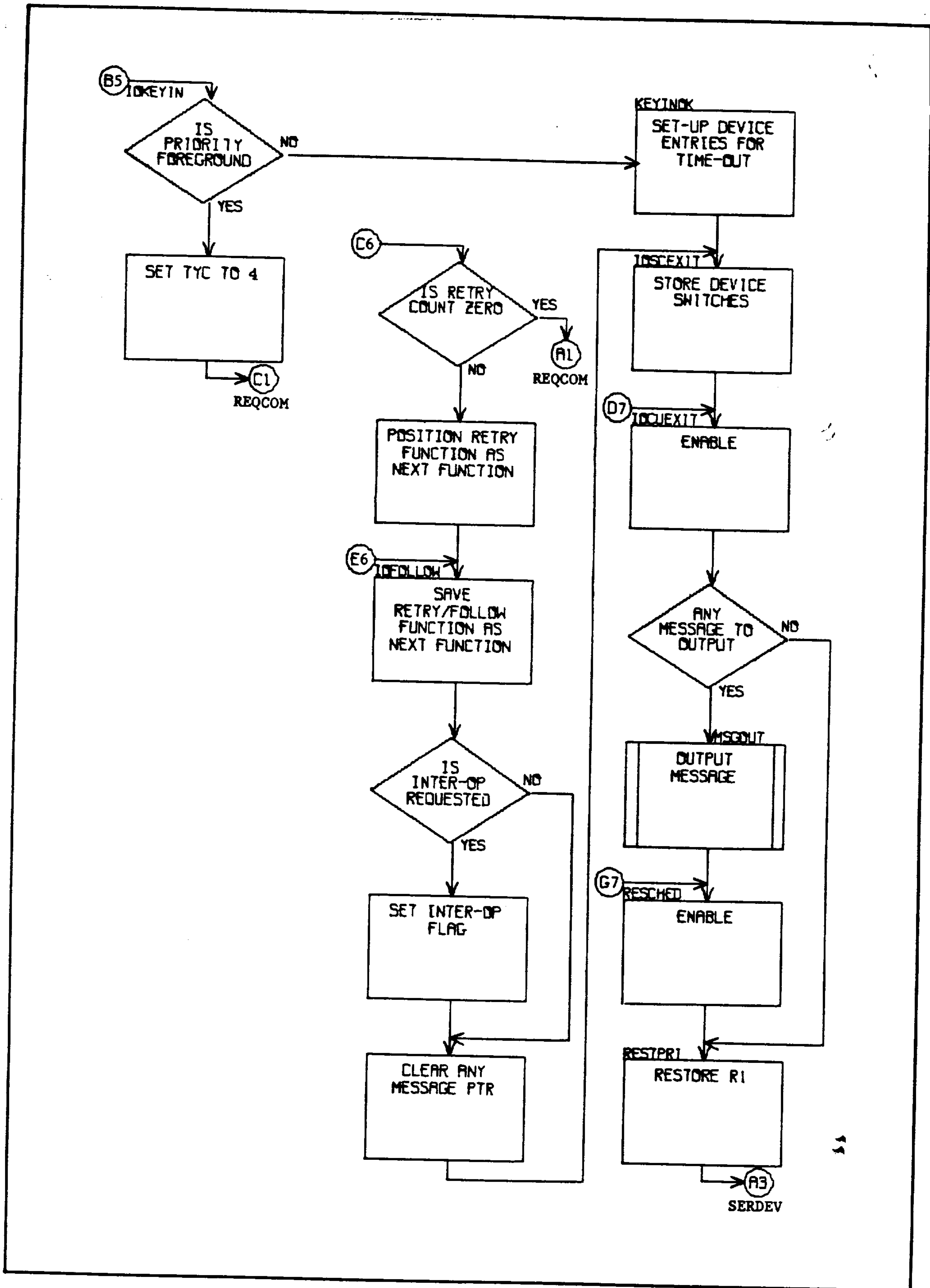


Figure 11. IOCS: CLEANUP Routine (cont.)

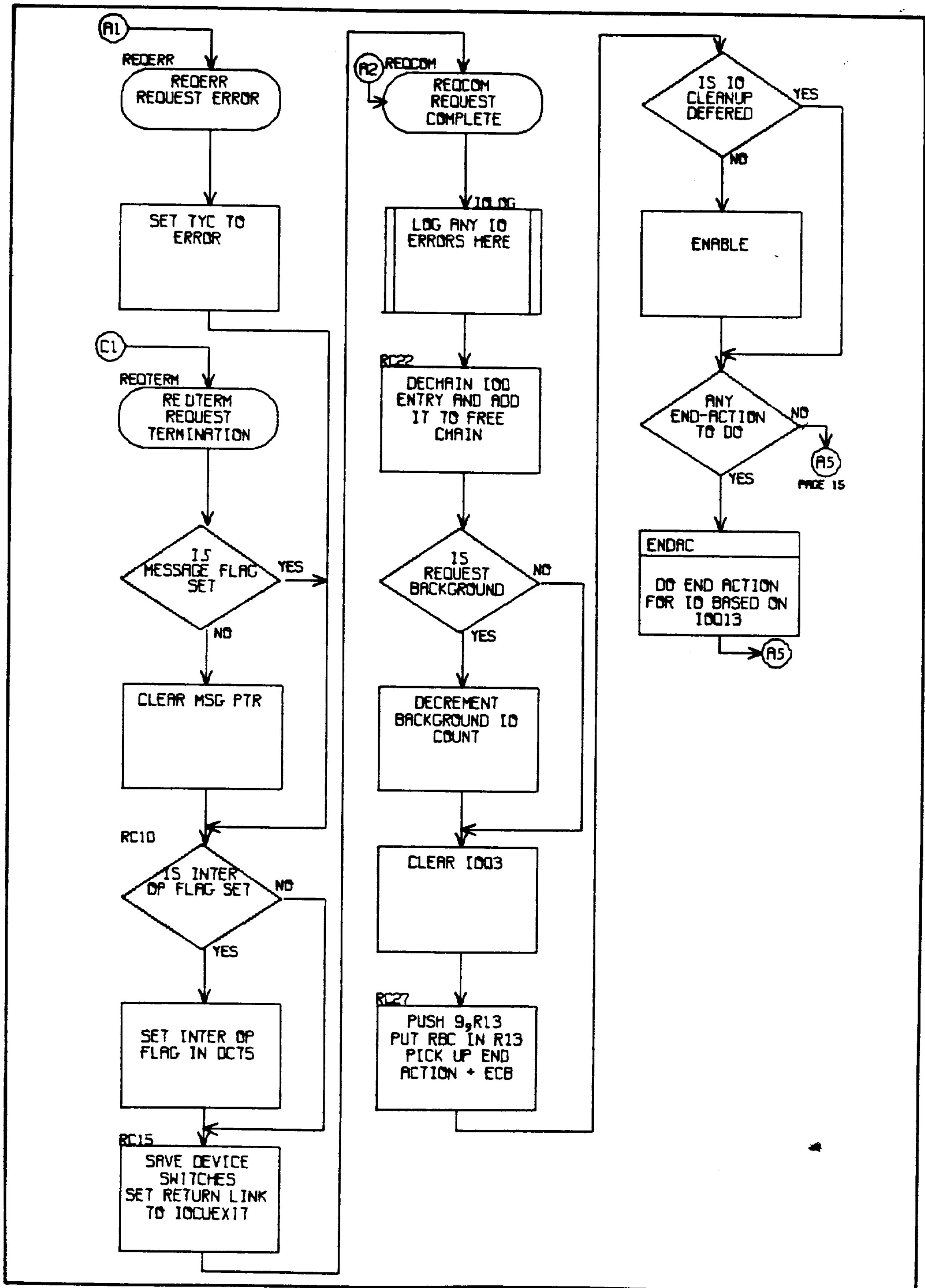



Figure 12. IOCS: REQCOM Routine

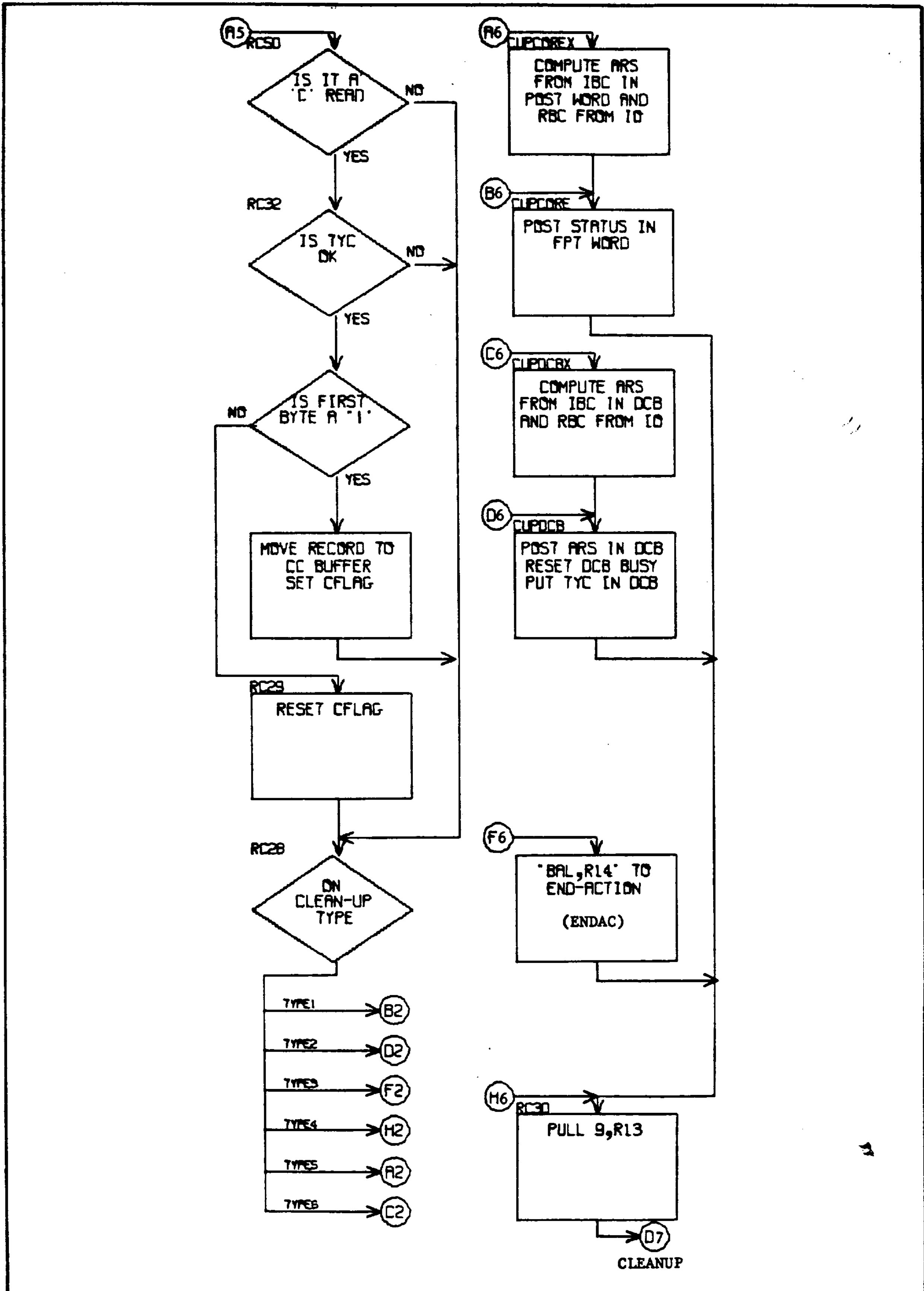



Figure 12. IOCS: REQCOM Routine (cont.)

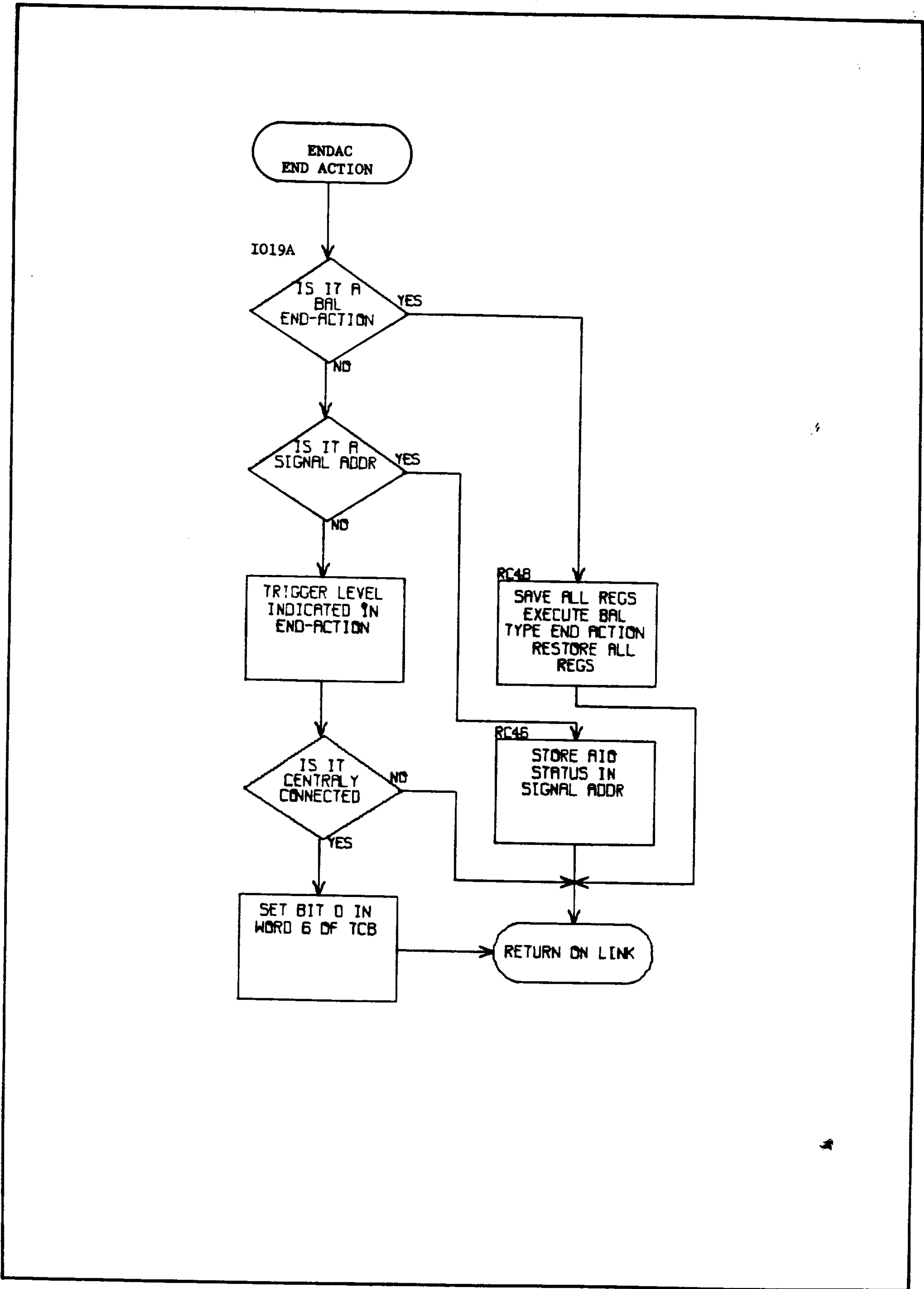



Figure 13. IOCS: ENDAC Subroutine

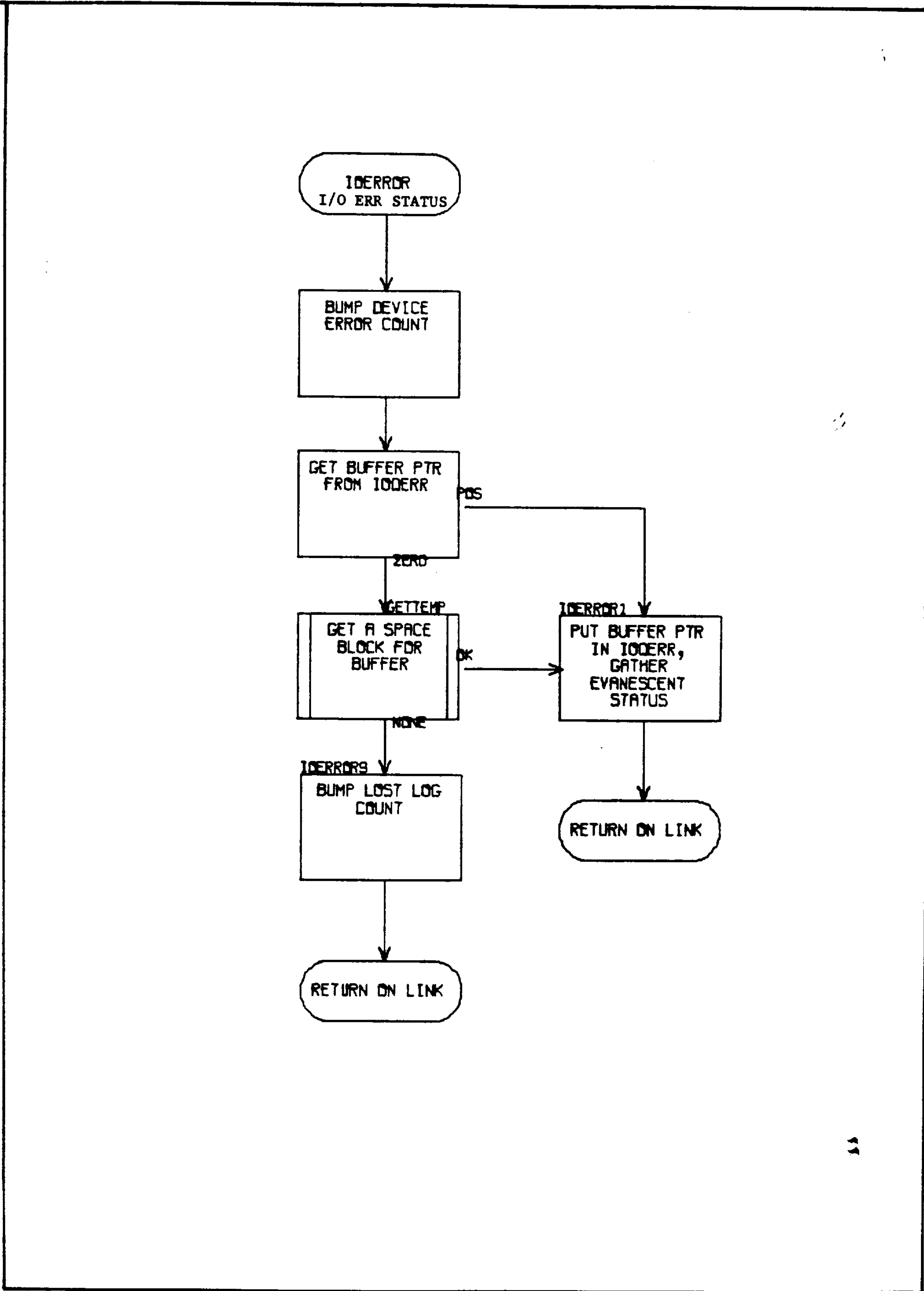



Figure 14. IOCS: IOERROR Subroutine

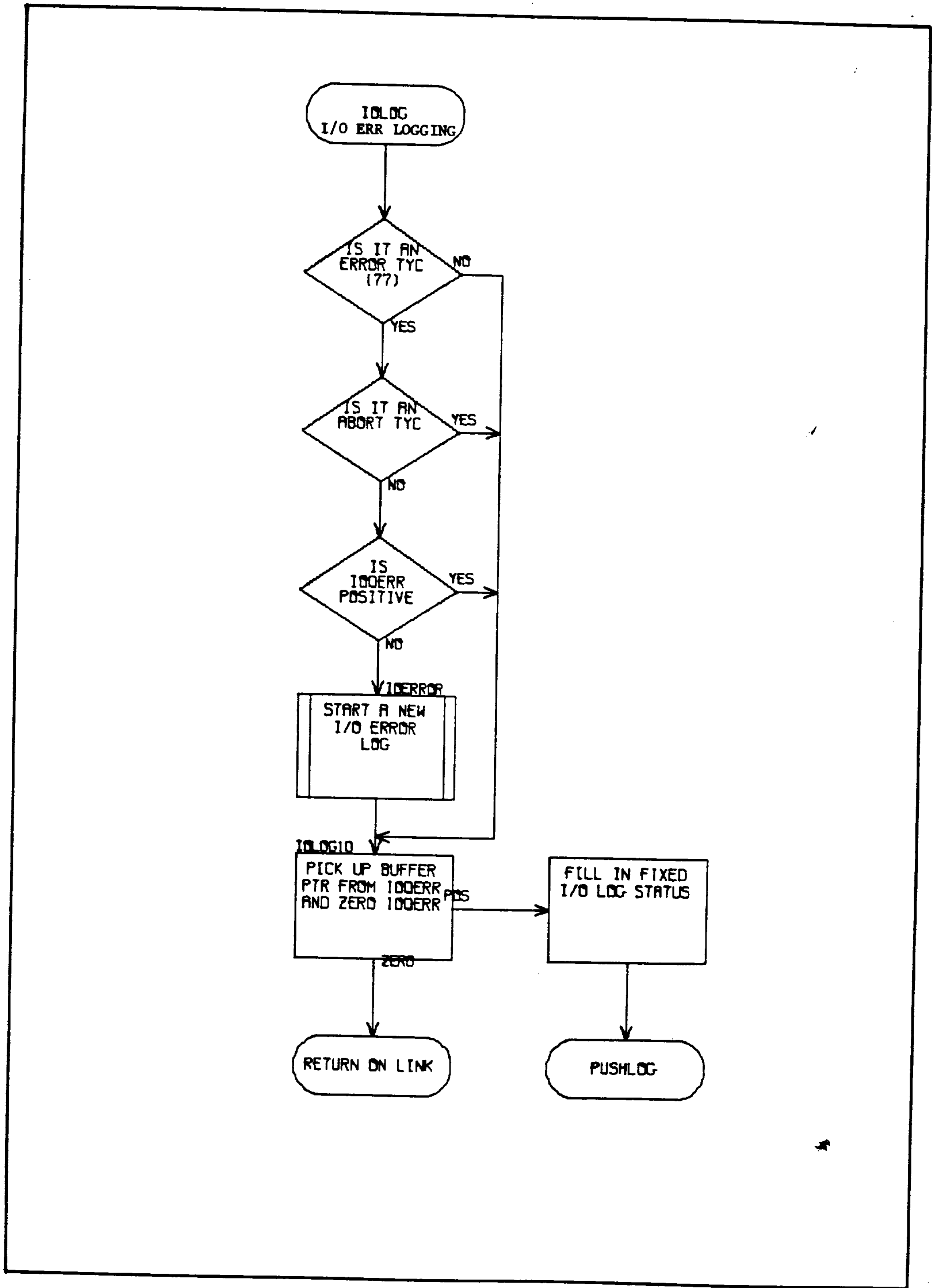



Figure 15. IOCS: IOLOG Subroutine

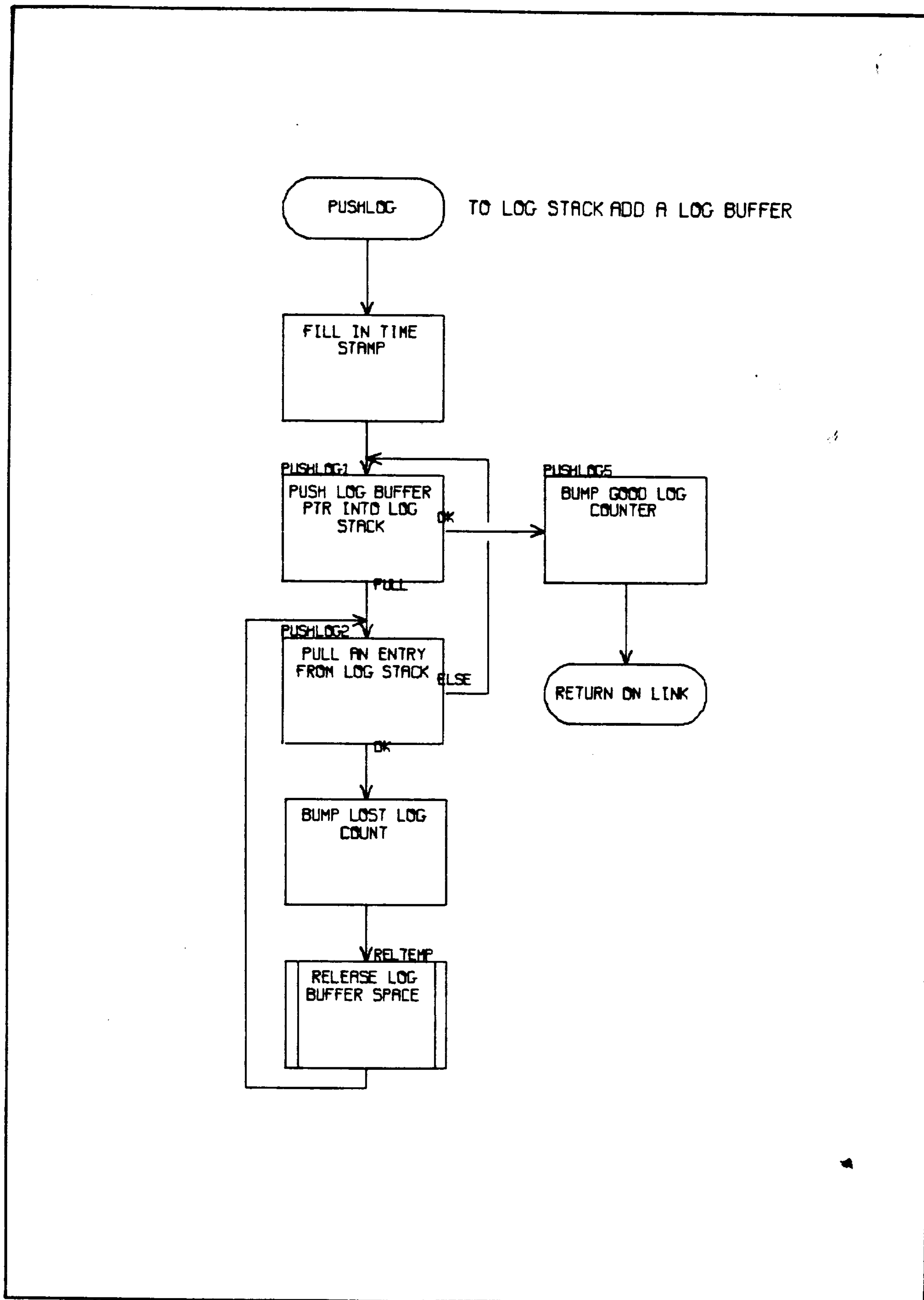



Figure 16. IOCS: PUSHLOG Subroutine

## Register Conventions

### QUEUE

Routine returns +1 if device is IOEX or down; +2 otherwise

At entry:

| | |
|---|---|
| R2 | ECB ID or zero |
| R4 | I/O Function code |
| R5 | Link |
| R6 | Number of retries |
| R7 | DCT index |
| R8 | CLEANUP Information Word 1 |
| R9 | CLEANUP Information Word 2 |
| R10 | I/O buffer address (byte address) |
| R11 | I/O length (in bytes) |
| R12 | Disk seek address or number of records to pass (magnetic tape) |
| R13 | Priority |

Registers R0 – R7 preserved; contents of R8 – R15 are lost

### CALLSD

At entry:

| | |
|---|---|
| R1 | FPT code |
| R2 | DCB address |
| R3 | FPT address |
| R5 | Link |

R1 – R7 preserved; contents of R0, R8 – R15 are lost

### SERDEV

At entry:

| | |
|---|---|
| R1 | DCT Index |
| R2 | Link |

Contents of all registers are lost

### RIPOFF

At entry:

| | |
|---|---|
| R2 | Task priority |
| R3 | IOQ pointer for Q entry to be removed |
| R5 | Link |

Contents of all registers are lost.

**STARTIO**

At entry: There is a startable request in R3. The device activity counter is set in R14 and interrupts are enabled. The I/O handler preprocessor is called unless user command list is specified. Handler return is to 'IOSST'.

Registers, after pre-handler return:

| | |
|---|---|
| R0 | Doubleword address of command list |
| R1 | Priority, CIT check mask, DCT index (8, 4, 20) |
| R2 | Flags, SERDEV exit, CIT index (3, 10, 19) |
| R3 | Request IOQ index |
| R4 | Handler flags, subchannel allocation code (8, 24) |
| R10 | Device operation table ('DOT') for 'IOSST' |
| R14 | Device activity count for re-entrancy check |
| R15 | Link for service device |

**CLEANUP/IOSCU**

Normal register usage:

| | |
|---|---|
| R1 | Priority, DCT index (8, 24) |
| R2 | Flags, SERDEV exit, CIT index (3, 10, 19) |
| R3 | Scratch, IOQ index (8, 24) |
| R11 | Remaining byte count (RBC) from post-handler |
| R12 | Flags returned from post-handler: |

| | |
|---|---|
| Bit 16 | Retry sequence |
| Bit 17 | Follow-on sequence |
| Bit 18 | Inter-operative request |
| Bit 19 | Key-in pending (normal) |
| Bit 20 | Key-in pending (special) |
| Bit 21 | Continue channel hold |
| Bit 22 | Force message print |
| Byte 3 | Type of completion |

| | |
|---|---|
| R13 | Message to be typed (0 if none) |
| R14 | Device activity count |
| R15 | Not used - reserved for future systems. |

**REQCOM**

At entry

| | |
|---|---|
| R1 | DCT and priority |
| R3 | IOQ pointer |
| R5 | Link |
| R11 | RBC |
| R12 | TYC |

R13 - R15, R0 - R4 preserved; contents of R5 - R12 are lost

## I/O Error Logging

Optionally, an I/O error-logging capability is provided. Whenever an I/O error is indicated by the device post-handler (by requesting a retry), IOSCU gets space for an error-log record, saves all evanescent I/O status, and puts the space pointer in IOQERR. Subsequent retries use the same space again.

In REQCOM, when the I/O completion is done, IOQERR is checked. If a log was started, the error-log record is completed and the pointer is stacked for later filing. Also, if an error completion code is indicated and no error-log record had been started, i.e., no retries were done, one is started and treated as above.

This assures that for any I/O request, no more than one error log will be generated. The error log will always indicate the status of the last error in a retry sequence.

The error log records relating to I/O errors are as follows:

- SIO failure
- Device timeout
- Unexpected interrupt
- Device error
- Secondary record for device sense data

The formats for these error logging records are given in Chapter 4, "Error Logging".

## I/O Statistics

Optionally, with error logging, I/O statistics are maintained. These may be displayed using the ESUM key-in.

The total number of SIOs issued for each device since system boot is kept in DCT#IO (word). The total number of I/O errors, counted when I/O error-log status is collected, for each device since system boot is kept in DCT#ERR (word).

The number of Log records successfully filed since system boot is kept in GOODLOGS (word). The number of Log records lost, because of space or time overruns, since system boot is kept in LOSTLOGS.

## Side Buffering

Both input and output side-buffering are optionally available for certain unit record devices. These allow effective double-buffered I/O for processors which do not themselves do double buffering.

DCTSDBUF is a word entry for all cevices which points to a post word followed by a buffer space for each side buffered device.

### Output Side Buffering

Output side buffering is done for all line printer, card punch and teletype output except for PRINT and TYPE CALs. The WRITE CAL waits for previous I/O to complete and the side buffer to be free. It then copies the users data into the side buffer. A request is made to output the side buffer. The caller is posted with the completion code of the previous output and all appropriate posting and end-action done.

### Input Side Buffering

Input side buffering is done only for the card reader. If the side buffer is free and a 'wait' READ CAL is issued, a side buffer read is started. Then this or any other READ CAL will wait for the side buffer read to complete. The input data will be copied into the user's buffer and posting/end-action will be done. If the record read is not a '!' or ':' card and the read was 'automatic', not binary, another side buffer read will be started before returning to the user.

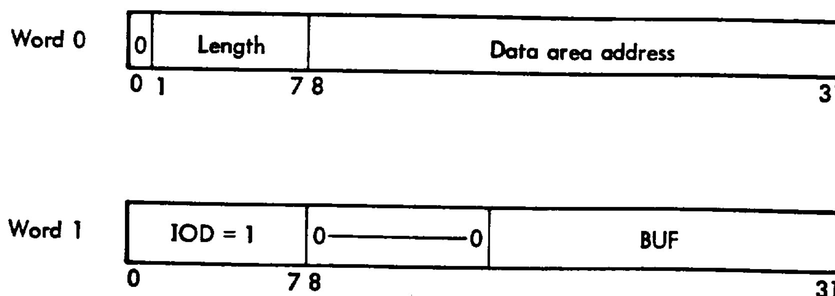
## Virtual I/O Buffering

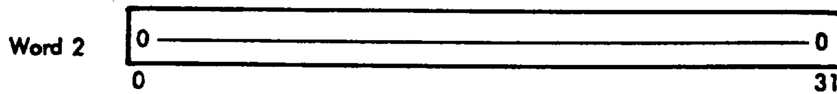
When service calls are initiated in mapped tasks that involve I/O, the monitor subroutines, FMGETRAD and FMLOCK convert the virtual buffer address to a real buffer address by performing one of the following functions as defined by the data area attached to the I/O ECB:

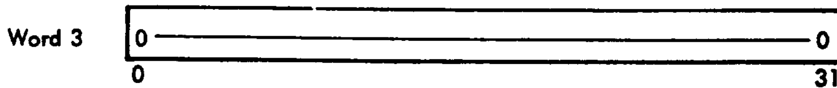
- Obtain a side buffer in CP-R TSPACE for unit record devices (i.e., CR, CP, LP, TY). If the request is a write operation, the user's data record is moved to the side buffer in TSPACE and is output from there. If the request is an input operation, the record is read into the buffer in TSPACE and moved to the user's buffer at the conclusion of the I/O operation.
- Set page locks if the user's buffer is contained within a single page or within contiguous real pages.
- Build a skeleton IOCD list of real buffer locations and sizes in CP-R TSPACE if the user's buffer is contained in two or more noncontiguous real pages, and set the page locks. The size and location of the IOCD list are input to QUEUE in place of buffer size and location.

The FMGETRAD routine obtains and initializes the data area which has one of the following formats:

Format 1






where

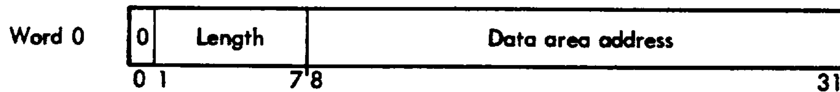
Word 0 contains the size and word address of the CP-R TSPACE obtained for the I/O request.

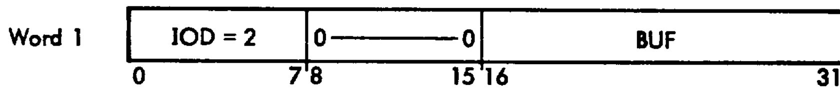
Word 1 IOD = 1 means that the I/O buffer is located in the TSPACE pointed to from word 0 and the I/O request is a WRITE. The user's data record is moved to the buffer in TSPACE before the QUEUE subroutine is called.

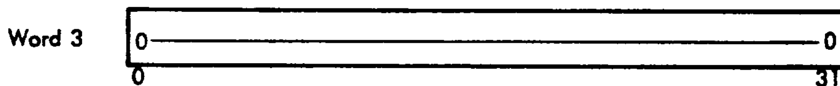
BUF is the byte address of the I/O buffer in TSPACE.

Words 2 and 3 are not used.

Format 2






where

Word 0 contains the size and word address of the CP-R TSPACE obtained for the I/O request.

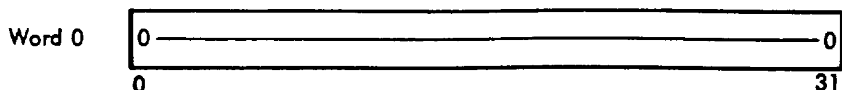
Word 1 IOD = 2 means that the I/O is a READ request and the data record is to be read into a buffer in the TSPACE pointed to from word 0. The data record is moved to the user's virtual buffer address at CHECK time.

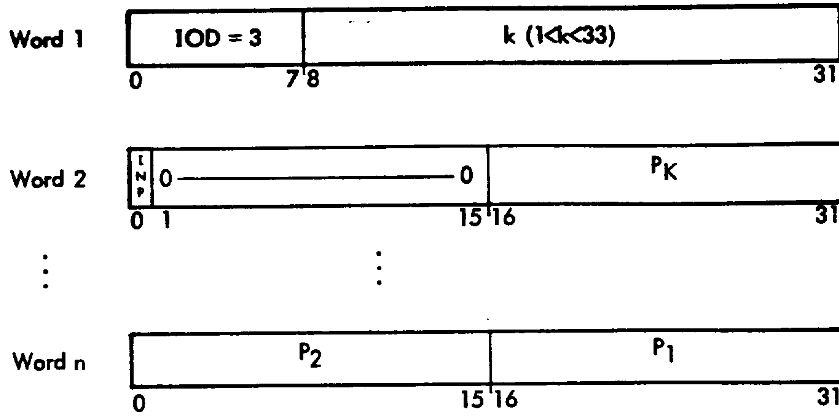
BUF is the real byte address of the buffer in TSPACE that will be used for the READ request.

Word 2 UBUF is the virtual byte address of the I/O buffer supplied by the user in the FPT or DCB.

Word 3 is not used.

Format 3





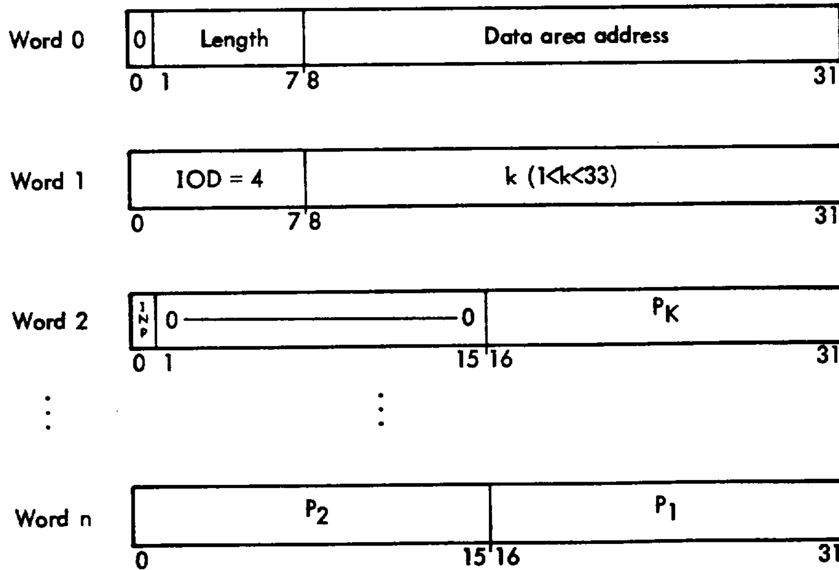

where

Word 0 is not used.

Word 1 IOD = 3 means that the I/O request will use the specified number (K) of real pages that contain the user's buffer.

Words 2 through n INP = 0 if the real Pages ($P_i$) have been locked; 1 if the pages have not been locked. $P_i$ are the real pages that contain the user's I/O buffer.

Format 4




where

Word 0 is a pointer to the data area in TSPACE that contains the IOCD list.

Word 1 IOD = 4 means that the I/O request will use the specified number (K) of real pages that contain the user's buffer and Word 0 contains a pointer to the data area in TSPACE that contains the IOCD list. Note that IOD = 4 means that the buffer is in noncontiguous real pages.

Words 2 through n are as shown in Format 3.

## IOEX

Two forms of IOEX are supported by the IOCS.

### Queued IOEX

Queued IOEX allows IOEX requests to be added to the queues just as any other request. They will be performed like any other request, but will not invoke either the pre- or post-device handler. Both queued IOEX requests and normal requests may be made on a device at the same time.

### Dedicated IOEX

Dedicated IOEX requires that all I/O management for the channel must be done by the user himself. The device must be dedicated either at SYSGEN or by a STOPIO call to IOEX, and no normal (queued) requests will be honored while it is dedicated.

## Disk Pack Track-by-Track Logic

All disk pack requests are initially attempted unchanged. If returning status indicates that either a cylinder crossing or a flawed track was encountered, the operation is retried with the data transfer broken into operations not larger than one track. Flawed tracks then encountered will be processed using the alternate track address in the header. This requires that packs be properly initialized with valid alternates in the headers.

The track-by-track parsing is wholly contained in the disk handler and uses only the original IOQ entry. No I/O time or execution time overhead is used if flawed tracks and cylinder crossings are avoided.

## Disk Pack Seek Separation

For all disk-pack operations, a separate seek order is issued without a data transfer. This takes advantage of two hardware features available on all disk packs. First, such seek operations do not tie up the channel and all disk packs may be seeking and therefore arm-moving at the same time. Second, the disk pack interrupts only when its arm motion is complete and when it is rotationally positioned in the sector previous to the indicated seek address.

This allows both arm-motion time as well as rotational-latency time to be overlapped with data transfers when disk-pack I/O traffic gets high.

## Disk Pack Arm-Position Queue Optimization

Optionally, an arm-positioning optimizer is used to minimize arm positioning time on all disk packs. No rotational-position optimization is intended or performed except that achieved on a multipack controller by virtue of multiseek operations which interrupt at a minimum rotational latency time.

The optimizing algorithm is intended to minimize disk arm-movement time by ordering disk-I/O-queue requests by arm position. No account is taken of request priority or order of time of request. The only guarantee is that two or more requests with the same seek address will be run in FIFO order.

The algorithm is as follows: At the end of any disk I/O operation, the current seek address is noted. The disk I/O queue is searched, in priority order, for the request which has the closest seek address in a forward direction. Requests which have seek addresses before the current position have their seek address biased so as to be forward, beyond any normal forward position. A queue entry with the same seek address is considered to be the farthest-away seek address. This guarantees that all requests will be eventually reached.

The result of this algorithm is to guarantee service to all requests. The arm motion tends to sweep from low to high arm position and then snap back to a low position. The algorithm offers up to 25% improvement in the time required to service short, random seek requests.

This snap-back or cyclic sweeping was chosen over an 'elevator' algorithm; i.e., two-way sweep, to minimize wait-time dispersion.

The code required for implementation of this algorithm is wholly contained in one piece at the logical end of the disk post-hardler. It is 38 words long and is conditional on the assembly switch #DISQING.

## Disk Angular-Position Queue Optimization

Optionally, an angular-position queue optimizer is used to select the "best" disk-I/O-queue entry to run. This is done to minimize rotational latency time without precluding priority queuing considerations.

At the end of any disk I/O option, the current rotational position is computed from the I/O start seek address and the byte count transferred. A tolerance is allowed for I/O-interrupt processing time, on the order of 1 ms.

The disk I/O queue is searched, in priority order, to determine if any lower priority request can be run entirely (including interrupt processing time) ahead of the normally selected high-priority request. As each one is found, it becomes the selected high-priority request. When the end of the queue is reached or when a request is elected which starts in the next available rotational position, I/O system flags are set to cause that request to be the next one started. This algorithm offers up to a 50% improvement in the time required to service many short, random seek, requests.

The code required for implementation of this algorithm is wholly contained in one piece at the logical end of the disk post-handler. It is 73 words long and is conditional on the assembly switch #RADQING.

## Deferred SIO

In a dual-redundant multi-processor system, where a pool of devices (i.e., disk packs or magnetic tape units), are shared, the I/O system allows limited concurrent use of these devices. If both processors try to use the same device, one will receive busy status from the device and the other will obtain use of the device. The I/O system, upon receiving the busy status, defers the SIO attempt for 5 seconds and then tries again.

In certain cases, such as an interruption between a seek and a read or write on a disk pack, recoverable errors may be reported by the hardware. The user of the deferred SIO capability should allow a reasonable number of retry attempts on his I/O requests.

## Logical Devices

Provision is made in SYSGEN to include logical devices. These are pseudo-devices which form a logical connection between Read and Write or Write File Mark I/O requests. They are SYSGENed as if they were real devices (including fictitious device addresses), and may be used like any other I/O device.

Read and Write requests are entered into the I/O queue normally. When the logical device finds a match between a Read and a Write request, the data transfer is made directly from the write buffer to the read buffer. Requests are handled on a first in, first out basis within priority and otherwise in order by priority. Actual record sizes are posted as usual. Write File Mark requests result in an EOF TYC and abnormal code being posted for the Read request.

Logical devices supply the capability of communicating between tasks via normal Read/Write services. It also provides the capability of intercepting or monitoring a data stream.

The user should be aware that I/O buffers are locked in main memory during any I/O operation, and that where very large buffers or very many outstanding I/O requests are used, this may result in a deadlock. This is particularly true of logical device requests which must be satisfied by another I/O request and not by independent action by a peripheral. Similarly, I/O queue entries may be tied up and result in a deadlock condition.

Logical device requests are not subject to I/O timeouts. The user must supply a time interval parameter on the service request (P13). This will cancel the request after the specified time period and post an FPT error code of X'67'.

## User I/O Services

**OPEN** This function opens a DCB that results in opening a disk file when the DCB is assigned to a disk file. If the Error and/or Abnormal address is given in the function call, the addresses are set in the DCB.

Opening a disk file involves constructing an RFT (RAD File Table) entry for the file. If the file is a permanent file, the area file directory is searched to locate the parameters that describe the file. These parameters are formatted and entered into the RFT. If the "file" is an entire area, the parameters used to construct the RFT entry are taken from the Master Dictionary. If the file is a background temporary file, the RFT entry must already have been constructed by the JCP. If the file is on a disk pack and a DED DPndd, R key-in is in effect, an abnormal code (X'2F') is posted in the DCB.

Blocking buffers or user-provided buffers are used for the directory search. Background requests use background buffers; foreground requests use foreground buffers.

**CLOSE** This function closes a DCB that may result in the closing of a disk file. Closing a permanent disk file involves updating the file directory if any of the directory parameters have been changed by accessing the file. Among such parameters that may change are file size (adding records to the file), record size (by Device File Mode call), etc.

If the file is extensible, all extents numerically higher than the one currently positioned in are deleted.

Disk files are closed only when (1) the DCB being closed is the last open DCB assigned to the file and (2) no operational labels are assigned to the file. Blocking buffers or user-provided buffers are used for the directory update as in the case of OPEN. If the file being closed is on a disk pack, a DED DPndd, R key-in is in effect, and this is the last open file on device ndd, the message !!DPndd IDLE will be output.

**READ/WRITE** A READ or WRITE function call will cause the addressed DCB to be opened if it is closed. READ/WRITE checks for legitimacy of the request by determining whether or not the following conditions are present:

1. For type 1 requests, the DCB is not busy with another type 1 request.
2. The assigned device or op label exists.
3. The user buffer lies in a legitimate region of core memory.
4. The type of operation (input or output) is legitimate on the device (e.g., output to the card reader is not allowed.

For device I/O, READ/WRITE builds a partial QUEUE calling sequence and calls a device routine that performs device-dependent testing such as:

1. Mode flag in DCB (BIN, AUTO) for devices that recognize it.
2. Testing byte count against physical record size for fixed-record-length devices.
3. Testing for PACK bit in DCB for 7T magnetic tape.
4. Testing for VFC for line printer or keyboard/printer.

The device routines set up the proper function code in the QUEUE calling sequence and transfer control to a routine called GETNRT. GETNRT completes the QUEUE calling sequence by obtaining the number of retries, setting up the user's end-action and building an ECB. GETNRT then calls QUEUE. When the request has been queued, control is transferred to the TESTWAIT routine which checks the wait indicator for the request. No-wait requests transfer to CALEXIT. Otherwise, requests transfer control to the CHECK logic at FMCK1 which waits for the I/O to complete.

For disk file I/O, READ/WRITE calls the routine labeled RWFILE. RWFILE tests for write protection violation on write requests, end-of-file on sequential read requests, and end-of-tape on all requests. The different types of requests are handled as follows.

Direct Access. The disk seek address is computed from the granule number provided in the FPT, and a QUEUE calling sequence is constructed that will queue up the request. Control then transfers to the CHECK logic.

Device Access. When the DCB associated with the READ/WRITE call is assigned directly to a disk, the disk device routine is entered. The disk device routine computes the disk seek address from the sector number provided in the FPT (Key parameter), obtains the proper function code and completes the queue calling sequence by branching to GETNRT.

Sequential Access (Unblocked). The disk seek address is computed from the file position parameters and a QUEUE call is made. Control then transfers to the CHECK logic.

Sequential Access (Blocked). The next record is moved from/to the blocking buffer and blocks are read/written as required to allow the record transfer. For example, the first read request results in the first block being read and the first record in the block being deblocked into the user buffer. Successive read requests will not require actual input from the disk until all records in the blocking buffer have been read. The blocks are always 256 words long and contain an integral number of fixed length records; that is, no record crosses a block boundary.

Sequential Access (Compressed Files). Compressed files are treated in a manner similar to blocked files with the following exceptions:

1. The records are compressed/decompressed on the way to/from the blocking buffer.

2. The buffer does not contain a fixed number of records since the records are no longer of fixed length after compression. However, no compressed record crosses a block boundary.

To compress a record, the following EBCDIC codes are used:

X'FA' End-of-Block code

X'FB' End-of-Record code

X'FC' Blank Flag code

X'FD' Control character code

All occurrences of two or more successive blank codes (X'40') are replaced by a Blank Flag code (X'FC') followed by a byte containing the length of the blank string. An End-of-Record code follows each record, and an End-of-Block code appears after the last record in a block.

The control character code (X'FD') allows a record to be compressed and decompressed without restrictions on the individual characters within the record. In the blocking routine, when a data character is detected which is a control character (i.e., X'FA', X'FB', X'FC', X'FD'), the data character is preceded by an X'FD' character. True control characters are not preceded by an X'FD' character. The deblocking routine removes X'FD' characters from the data record.

When compressing records into the blocking buffer, a length of the compressed record is first computed and a test performed to determine whether the record will fit in the block. If so, it is placed in the buffer. If not, an End-of-Block code is written in the buffer and the buffer is written to the file.

If the disk file is extensible, special handling is done as follows:

Sequential Access Write. The write routines for unblocked, blocked and compressed format files automatically allocate an extension file when an end-of-file condition is detected. The appropriate RFT entries are updated to reflect any characteristics which may be different from those of the previous extent and then the write continues using the new file extent.

Due to repositioning a file, it is possible to detect an end-of-file on an extent and find that the next extent already exists. When this happens, the procedure is the same as outlined above except that the previously allocated extent is used.

Sequential Access Read. When an EOD is detected in the READ routines for unblocked, blocked and compressed format files, the file directory is searched for the next extent in sequence. When it is found the RFT is updated to reflect any characteristics which may be different from those of the previous extent and the READ continues using the new extent.

Direct Access Write. Direct access files automatically extend as described for sequential access files. In addition, writing records which are larger than one granule is permitted. In order to accomplish this, the direct access write routine can

- Allocate one extent large enough for the record if the file was originally allotted without the "fix" option.
- Allocate two or more extents if the record will not fit in one extent and the file was originally allotted with the "fix" option specified. The record will be written in two or more sections if it is too large to be contained within a single extent. If the request is with no-wait, only the last portion of the record will be written without wait. Extents between the first and last extent will appear to have been written even if they have not been in order to simulate the characteristics of nonextensible files.

Direct Access Read. Automatic switching to the next extent will occur for direct access files as described for sequential access files. In addition, the direct access read routine can read more than one granule even if the record crosses two or more extents. This is accomplished by breaking up the read into sections where each section is equal to or smaller than the extent size. If the READ request is with no-wait, only the last section is read without wait.

At the conclusion of the file access, the status is posted in the user DCB or FPT and control is transferred to the CHECK logic.

**PRINT** This function builds the QUEUE calling sequence to perform the output on LL. After calling QUEUE, the routine either waits for completion, if wait was requested in the system call, or returns control to the user.

**TYPE** This function builds the QUEUE calling sequence by using code contained in the PRINT function. As in PRINT, a wait or return is performed as requested by the user.

**DFM** This function sets the MOD and PACK indicator in the addressed DCB to values given in the system call. If the DCB is assigned to a disk file, the record size (RFT5), the organization (RFT7), and/or the granule size (RFT4) are set if requested by the user. The corresponding parameters on the file directory are updated when the file is closed.

**DVF** This function sets the DVF bit in the addressed DCB to the value (0 or 1) specified by the user.

**DRC** This function sets the DRC bit in the addressed DCB to the value (0 or 1) specified by the user.

**DEVICE** (Set Device/File/Oplb Index.) This function assigns a DCB to the specified device or file. The assignment is accomplished by setting one or more of the following parameters in the addressed DCB: ASN, DEVF, TYPE, DEV/OPLB/RFILE, or disk file name.

**DEVICE** (Get Device/File/Oplb Name.) This function returns requested information regarding the assignment of a DCB. The information is in EBCDIC form. The request is fulfilled when it is consistent with the actual assignment of the DCB. Otherwise, a word, or words, of zero will be substituted for the EBCDIC information.

**CORRES** This function determines if the two specified DCBs have corresponding assignments. If the assignments are the same, upon return to the user, register 8 will contain a value of 1. Otherwise, register 8 will contain a value of 0.

**REWIND** This function rewinds magnetic tapes and disk files. No action is taken if the addressed DCB is assigned to any other type of device.

Magnetic tapes are rewound by building a QUEUE calling sequence with the Rewind function code and calling QUEUE.

Disk files are rewound by zeroing the file position (RFT11), current record number (RFT12), blocking buffer position (RFT10), and blocking buffer control word address (RFT17) and using job (RFT14) parameters. Extensible files are positioned at the first record of extent 0.

**WEOF** This function writes an "end-of-file" on paper tape punch, card punch, magnetic tape, and disk files. A request addressing a DCB assigned to some other type of device results in no action.

An "end-of-file" is written on paper tape by calling QUEUE with a request to write an EBCDIC '!EOD' record.

An "end-of-file" is written on a card by calling QUEUE with a request to write an EBCDIC '!EOD' record.

An "end-of-file" is written on magnetic tape by calling QUEUE with a request to write a tape mark.

An "end-of-file" on a disk file is "written" by copying the current record number minus 1 (RFT12) to the file size (RFT6) and setting an indicator so that the file directory will be updated when the file is closed.

**PREC** This function positions magnetic tapes and disk files by moving some specified number of records either backward or forward. It does not affect other devices. Positioning is performed as follows:

1. A magnetic tape QUEUE call is constructed that specifies through the function code the direction of the motion, and through the "seek-address" parameter the number of records to move. The basic I/O system then moves the tape.
2. The new position within the file of an unblocked disk file is computed as a function of the record size and the sector size. File position (RFT11) and current record number (RFT12) parameters are set to indicate the new position.
3. The new position of a blocked disk file is computed as a function of the current record number, record size, block size, current blocking buffer position, current file position, and disk sector size. The blocking buffer position (RFT10), file position (RFT11), and current record number (RFT12) are set to indicate the new position.
4. The new current record number of a compressed disk file is computed and subroutine PCFIL is called. This subroutine positions a compressed disk file at the specified record by counting records from the beginning of the file until the desired position is found. PCFIL sets the blocking buffer position (RFT10), file position (RFT11), and current record number (RFT12) parameters to indicate the new position.

**PFILE** This function positions magnetic tape and disk files at the beginning or end of files. It does not affect other devices. Positioning is performed as follows:

Magnetic Tape. A QUEUE call is constructed with function code to "space file" either backwards or forwards. This results in the tape being positioned past the tape mark in the specified direction. If a skip was not requested, the tape is positioned on the other side (near side) of the tape mark through a QUEUE call for a position one record opposite in direction to the space file.

Disk File Backward. A rewind is done (see description for REWIND).

Disk File Forward.

Unblocked Disk File. Current file position is computed as a function of the file size, the record size, and the disk sector size. The current file position (RFT11) and the current record number (RFT12) are set to indicate the new position.

Blocked Disk File. Current file position (RFT11) and the Blocking Buffer Position (RFT10) are computed as a function of the file size, record size, block size, and disk sector size. These parameters and the current record number (RFT12) are set to indicate the new position.

Compressed Disk File. Subroutine PCFIL is called with file size plus one as the record number. This subroutine positions the file at the start of the specified record.

Extensible disk files are positioned as described above within the last extent.

**ALLOT** This function defines a file in a permanent disk area. The input parameters are used to form a new file directory entry.

The directory of the specified area is searched to insure that the file is not a duplicate. If it is not a duplicate, it is allotted as a new file. The logical flow of the allot algorithm is shown in Figure 17. In general, new files are allocated the next free space in the area if there is room for the entry in the last directory sector. When the last directory sector is filled, deleted file space is reused, if possible, before a new directory sector is created.

When a deleted entry is reused, the entry having the smallest size large enough for the new file is used. Disk space is lost if the deleted file containd more space than the new entry requires. This space and the space held by other deleted files can be reclaimed by executing a RADEDIT :SQUEEZE command.

The number of sectors to allocate for a file is calculated using the formulas

$$C = \left(\frac{FSIZE}{25} + r\right) * \left(\frac{256}{s} + r\right)$$

$$B = \left(\left(FSIZE/\frac{256}{RSIZE}\right) + r\right) * \frac{256}{s}$$

$$U = ((RSIZE/s)+r)*FSIZE$$

where

r = 1 if remainder $\neq$ 0, and 0 if remainder = 0.

s equal disk sector size in words.

**DELETE** This function deletes a file in the specified permanent disk area. The input file name is used to search the file directory for the entry to be deleted. When the entry has been located, the first two words of the file directory entry are zeroed out. The BOT and EOT remain unaltered. If the file is extensible each extent is deleted as described above starting with the last extent and proceeding to the first extent (extent 0). The space formerly allocated by the entry becomes unused until either a RADEDIT :SQUEEZE command is executed, or an ALLOT command or call is executed with insufficient space at the end of the specified area. Space is then allocated by using a deleted entry.

**TRUNCATE** This function uses the specified area and file name to search the file directory for the entry to be truncated. The actual size of the file is calculated and the EOT of the file directory entry is updated accordingly.

The actual file size for blocked and unblocked files is determined by using the FSIZE and RSIZE of an entry; for compressed files, an RFT entry (RFT11) containing the current record number is used. The space formerly allocated between the EOT of an entry and the BOT of the next entry becomes lost and is not available until a RADEDIT :SQUEEZE command is executed.

If the file is extensible, the last extent is determined by a directory search and when located, it is truncated as described for nonextensible files.

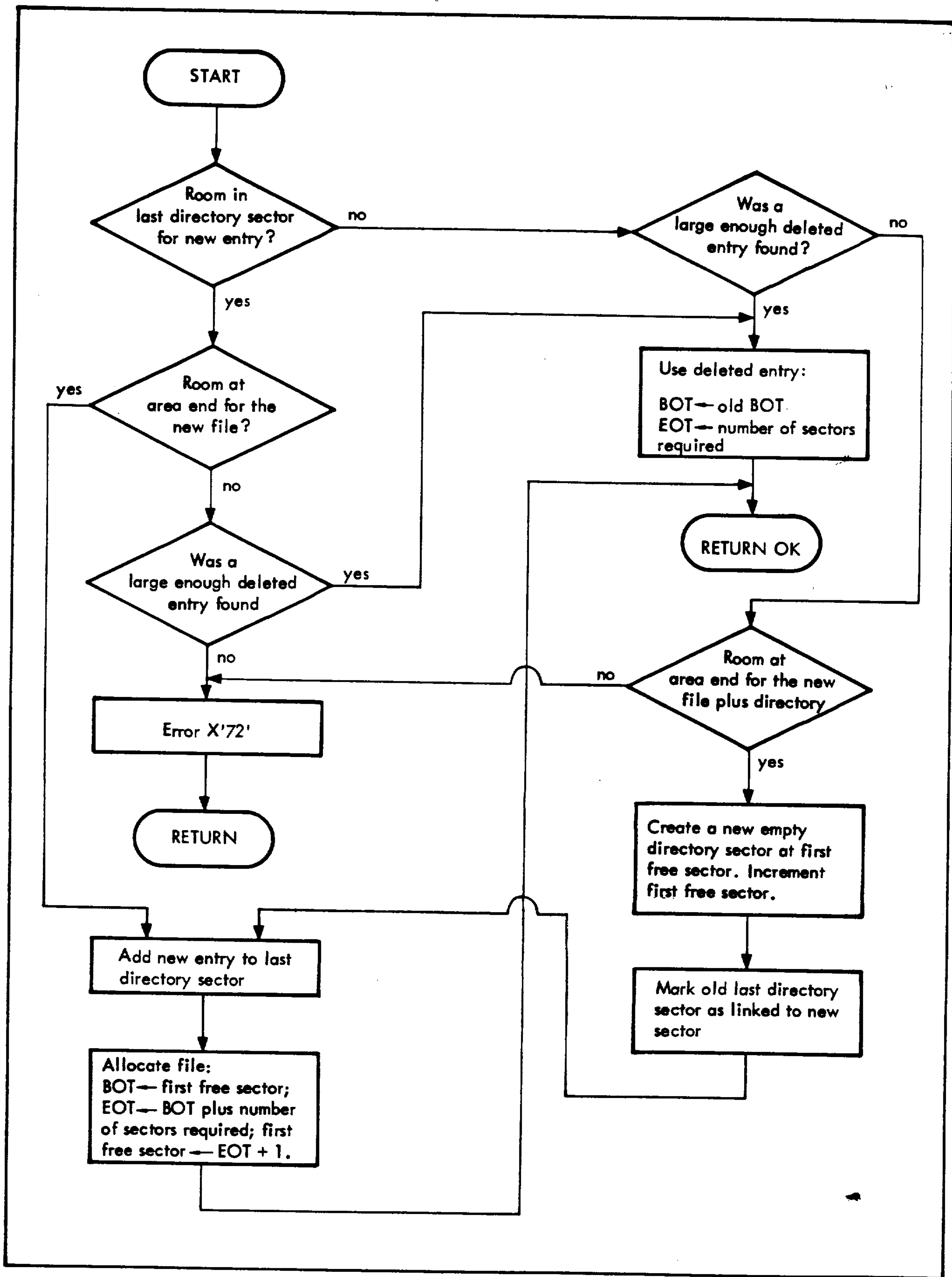



Figure 17. Logical Flow of ALLOT

# 4. ERROR LOGGING

The detection of a system, device, or software error will cause CP-R to acquire information about the error, generate a log record, post the log record, and perform some form of recovery. Upon finding a stacked error-log record pointer, the Control Task will call the LOG overlay to file the log.

The LOG overlay unstacks the log record and writes it to the ER oplabel in 16-word records. Normally, the ER oplabel should be directed to a file in the SP area named ERRFILE with a record size of 16 words and blocked format. However, the ER oplabel can also be directed to a card or tape device.

It should be noted that if ERRFILE does exist in the SP area, the ER oplabel will be connected to it by default at system boot time.

## Error Log Record Formats

The following error logs can be generated by CP-R:

| Code | | Code | |
|---|---|---|---|
| 11 | SIO Failure | 22 | System Identification |
| 12 | Device Timeout | 23 | Time Stamp |
| 13 | Unexpected Interrupt | 27 | Operator Message |
| 15 | Device Error | 28 | I/O Activity Count |
| 16 | Secondary Record for Device Sense Data | 30 | PFI Primary Record |
| 17 | Hardware Error | 31 | MFI Primary Record |
| 18 | System Startup | 32 | Processor Poll Record |
| 19 | Watchdog Timer | 41 | 550 Processor Configuration |
| 1D | Instruction Exception | 42 | 550 Memory Parity Secondary Record |
| 21 | Configuration Record | 43 | Memory Poll Record |

The formats for these error log records are given below consecutively:

SIO FAILURE

<table>
<tr><td>X'11'</td><td colspan="2">Count=6</td><td colspan="2">Model Number</td><td>DCTMODX 010x<br>100x<br>110x</td></tr>
<tr><td colspan="5">Milliseconds Since Midnight</td><td></td></tr>
<tr><td colspan="3">SIO Status</td><td colspan="2">I/O Address</td><td>DCT21, DCT1</td></tr>
<tr><td>MFI if Σ6 or Σ7</td><td colspan="2">SIO CC</td><td>TDV CC</td><td></td><td>-, DCT19, DCT20</td></tr>
<tr><td>Subchannel Status</td><td></td><td colspan="3">TDV Current Command DA</td><td rowspan="2">DCT13</td></tr>
<tr><td colspan="3">TDV Status</td><td colspan="2">Bytes Remaining</td></tr>
</table>

The SIO failure is emitted when the following SIO CC are returned:

The I/O sequence is SIO, TDV.

DEVICE TIMEOUT

| | | | | Label |
|---|---|---|---|---|
| X'12' | Count = D | Model Number | | DCTMODX |
| Milliseconds Since Midnight | | | | |
| HIO Status | | I/O Address | | DCT12 |
| | HIO CC | TDV CC | TIO CC | -, DCT19, DCT20, DCT20A |
| Subchannel Status | | TDV Current Command DA | | DCT13 |
| TDV Status | | Bytes Remaining | | DCT13 |
| Current Command Doubleword | | | | |
| TIO Status | | Retry Request | Retries Remaining | DCT21, IOQ10, IOQ11 |
| I/O Count | | | | DCT25 |
| | | | | |
| Seek Address | | | | IOQ12 |

UNEXPECTED INTERRUPT

| | | | | Label |
|---|---|---|---|---|
| X'13' | Count = 4 | Model Number (0 if unknown) | | DCTMODX |
| Milliseconds Since Midnight | | | | |
| AIO Status | | I/O Address | | DCT12 |
| | AIO CC | | | -, DCT19, -, - |

DEVICE ERROR

| | | | | |
|---|---|---|---|---|
| X'15' | Count=D | Model Number | | DCTMODX |
| Milliseconds Since Midnight | | | | |
| AIO Status | | I/O Address | | DCT12 |
| ////// | AIO CC | TDV CC | TIO CC | –, DCT19, DCT20, DCT20A |
| Subchannel Status | ////// | TDV Current Command DA | | DCT13 |
| TDV Status | | Bytes Remaining | | DCT13 |
| Current Command Doubleword | | | | |
| TIO Status | | Retry Request | Retries Remaining | DCT21, IOQ10, IOQ11 |
| I/O Count | | | | DCT25 |
| ////// | | | | |
| Seek Address | | | | IOQ12 |

SECONDARY RECORD FOR DEVICE SENSE DATA

| | | |
|---|---|---|
| X'16' | Count as Needed | I/O Address |
| Milliseconds Since Midnight | | |
| Sense (Up to 16 bytes) | | |

Note: The I/O address links the secondary record to the corresponding device error entry.

SYSTEM STARTUP

| 0–7 | 8–15 | 16–23 | 24–31 |
|---|---|---|---|
| X'18' | Count = 4 | Startup Type = 3 | Recovery Count = 0 |
| Milliseconds Since Midnight | | | |
| Year - 1900 | | Julian Day | |
| (hatched) | | | |

HARDWARE ERROR

| 0–7 | 8–15 | 16–23 | 24–31 |
|---|---|---|---|
| Code | Count = 10 | 0———0 | Trap CC |
| Milliseconds Since Midnight | | | |
| PSD Word 1 | | | |
| PSD Word 2 | | | |
| 0 (reserved) | | | |
| 0 (reserved) | | | |
| Real Address of Trapped Instruction | | | |
| Trapped Instruction | | | |
| (hatched) | | | |

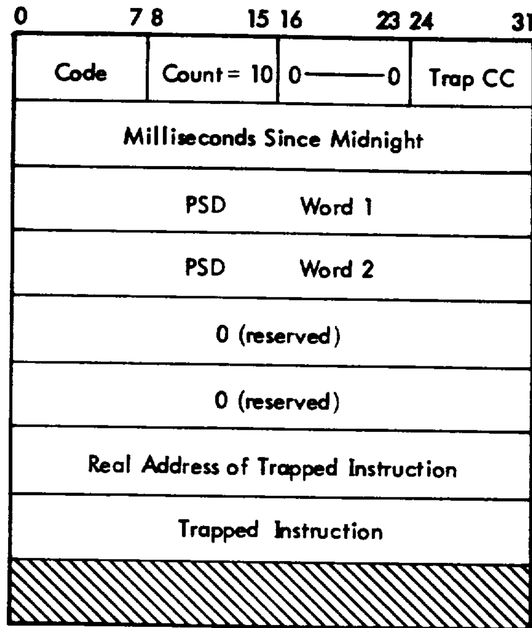
Generated by trap 4C.

WATCHDOG TIMER

| 0–7 | 8–15 | 16–23 | 24–31 |
|---|---|---|---|
| Code | Count = 10 | 0———0 | Trap CC |
| Milliseconds Since Midnight | | | |
| PSD Word 1 | | | |
| PSD Word 2 | | | |
| 0 (reserved) | | | |
| 0 (reserved) | | | |
| Real Address of Trapped Instruction | | | |
| Trapped Instruction | | | |
| (hatched) | | | |

Generated by Trap 46.

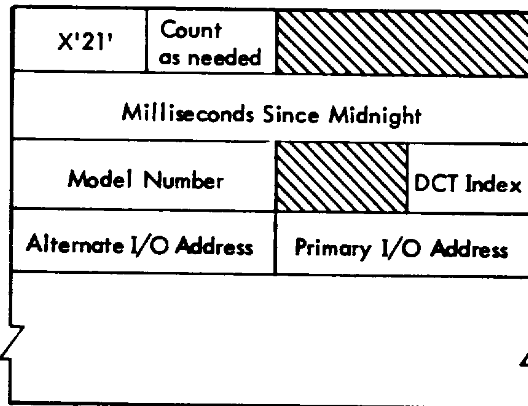
INSTRUCTION EXCEPTION




Generated by Trap 4D

CONFIGURATION RECORD

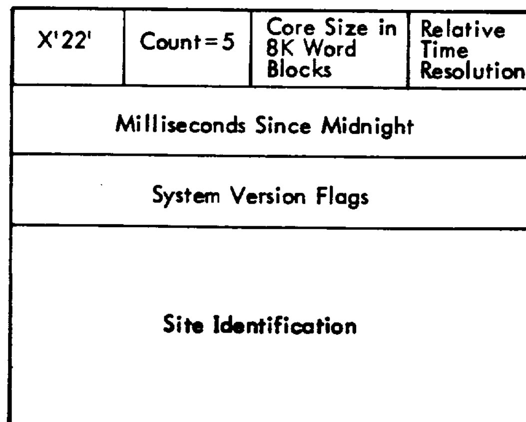
Entered at system STARTUP




One pair of words per device in DCT order; multiple records may occur (maximum five devices per record).

SYSTEM IDENTIFICATION

Recorded at system STARTUP




Relative Time Resolution is expressed as a value of n such that actual relative time resolution = $2^n$ msec. The value of n for the most likely resolutions are

n = 0 when the timing space is supplied by a frequency ≥ 1 KHZ

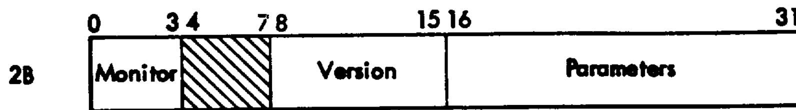
n = 1 500 HZ

n = 4 60 HZ

For CP-R, n = 1.

System, Version, Flags

The format of system, version, flags and site identification is operating system specific. For the CP-R system, version and flags are formatted at location X'2B'.




Location 2B contains three items:

1. Monitor - This field contains the code number of the monitor. The codes are as follows:

| Code | Monitor |
|---|---|
| 0 | None or indeterminate |
| 1 | BCM |
| 2 | RBM |
| 3 | RBM-2 |
| 4 | BPM |
| 5 | BTM/BPM |
| 6 | UTS |
| 7 | CP-V |
| 8 | CP-R |
| 9-F | Reserved for future use |

2. Version - This is the version code of the monitor and is coded to correspond to the common designation for versions. The alphabetic count of the version designation is the high-order part of the code and the version number is the low-order part. For example, A00 is coded X'10' and D02 is coded X'42'.

3. Parameters - The bits in this field are used to indicate suboptions of the monitor.

| Bit | Meaning if Set |
|---|---|
| 31 | Symbiont routines included. |
| 29 | Real-time routines included. |
| 28 | Unused. |
| 27 | Reserved. |
| 26 | Reserved. |
| 24-25 | Field defining CPU. |

| Bit 24 | Bit 25 | Meaning |
|---|---|---|
| 0 | 1 | Sigma 5-7 |
| 1 | 0 | Sigma 9 |
| 1 | 1 | Xerox 550 |

TIME STAMP

| X'23' | Count = 3 | |
|---|---|---|
| Milliseconds Since Midnight | | |
| Year - 1900 | | Julian Day |

This record entered once each hour on the hour.

Binary integers

OPERATOR MESSAGE

| X'27' | Count as needed | |
|---|---|---|
| Milliseconds Since Midnight | | |
| TEXTC Count | | |
| TEXTC Message<br>Max Size = 56 characters (CP-R) | | |

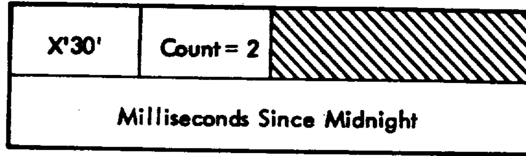
A facility is provided to inject messages from the computer operator (or diagnostic program) into the error log. The operator may enter these messages from the operator console via the ERRSEND key-in.

I/O ACTIVITY COUNT

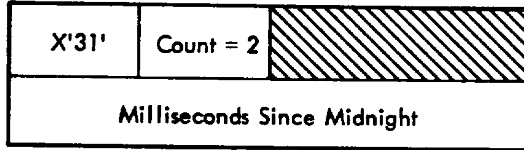
| 28 | Count as needed | | DCT Index of First Device |
|---|---|---|---|
| Relative Time | | | |
| I/O $Address_1$ | | | DCT $Index_1$ |
| I/O $Count_1$ | | | |
| I/O $Address_2$ | | | DCT $Index_2$ |
| I/O $Count_2$ | | | |
| ⋮ | | | |

Recorded once per hour and at recovery. Maximum of 5 entries per record. Counts are reset to zero at Boot.

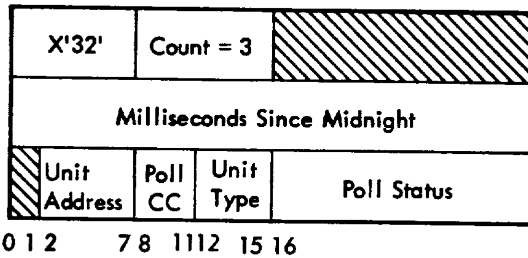
PFI PRIMARY RECORD




MFI PRIMARY RECORD

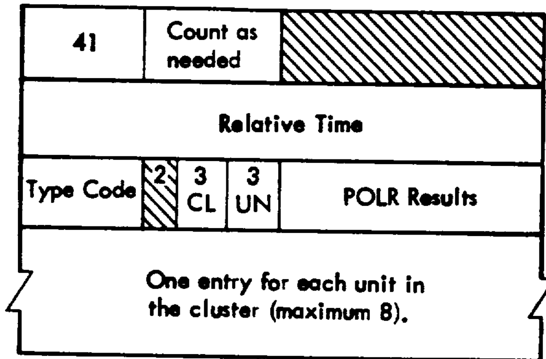



PROCESSOR POLL RECORD




One record produced per nonzero poll status received.

550 PROCESSOR CONFIGURATION

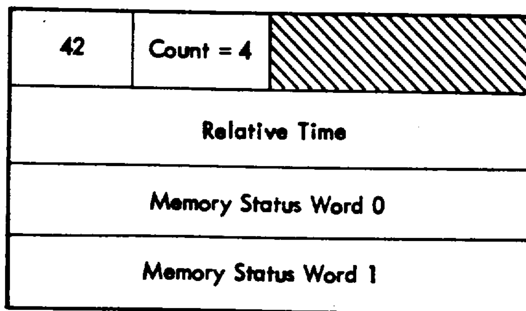



One record per cluster defined in SYSGEN.

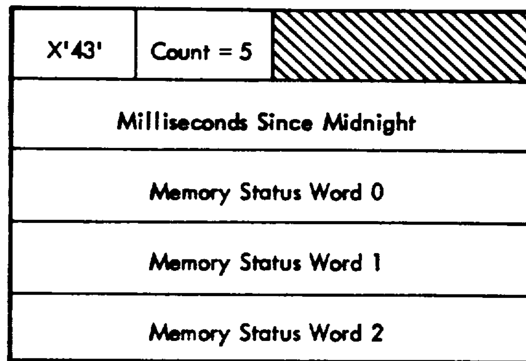
CL = cluster #
UN = unit #
TYPE = unit type

| Type Code | Unit Name |
|---|---|
| 1 | CPU |
| 2 | MI |
| 3 | PI |
| 4 | MIOP |
| 7 | SU |

550 MEMORY PARITY SECONDARY RECORD



MEMORY POLL RECORD



# 5. JOB CONTROL PROCESSOR

## Overview

The Job Control Processor (JCP) is assembled as a Relocatable Object Module (ROM) and is loaded at SYSGEN time by the SYSLOAD phase of SYSGEN. The JCP is absolutized to execute at the start of background and is loaded into the JCP file on the disk. The JCP is loaded from disk for execution by the Background Loader upon the initial "C" key-in; and thereafter, is loaded following the termination of execution of each processor or user program in background memory.

The JCP executes with special privileges since it runs in Master Mode with a skeleton key. Master Mode rather than Slave Mode is essential to the JCP since, at appropriate times, it executes a Write Direct instruction to trigger the CP-R Control Task. A skeleton key instead of the background key is also essential to the JCP since it sets flags for itself and the Monitor in the resident Monitor portion of memory. Bit zero of system cell K:JCP1 is set to 1 to inform the Monitor that the JCP is executing.

The JCP controls the execution of background jobs by reading and interpreting control commands. All cards read from the "C" device that contain an exclamation mark in column one (except for an !EOD command), are defined as JCP control commands. The I/O portion of the Monitor will not allow any background program except the JCP to read a JCP control command. The JCP runs until a command is read that requires the execution of a processor or user program, or until a !FIN command is encountered.

The JCP presently requires a minimum of about 4K of core to execute, which means that the smallest possible core space allocated to the background must be at least 4K.

The flowchart illustrated in Figures 18-21 depict the overall flow of the JCP, and Figures 22 through 39 illustrate the JCP commands. The labels used in the flowcharts correspond to the labels in the program listing.

## ASSIGN Command Processing

The !ASSIGN commands are read from the "C" device by the JCP, and are primarily used to define or change the I/O devices used by a program. The !ASSIGN command can also be used to change parameters in a DCB. Since all !ASSIGN commands must be input prior to the RUN or Name command (where Name is the name of a processor or user program file in the SP area) to which they apply, the information from each !ASSIGN command is saved in core by the JCP. The JCP builds an ASSIGN table containing the information from each !ASSIGN command. This table consists of ten words for each !ASSIGN, plus one word specifying the number of ten-word entries. The table remains in a job-reserved page and is passed to the Background Loader. After the Background Loader initiates the program, it makes the appropriate changes to the program's DCBs from the information in the ASSIGN table. The ASSIGN table can then be destroyed as the program executes; therefore, !ASSIGN commands take effect only for a job step and not an entire job. The ASSIGN table has the format shown in Table 1.

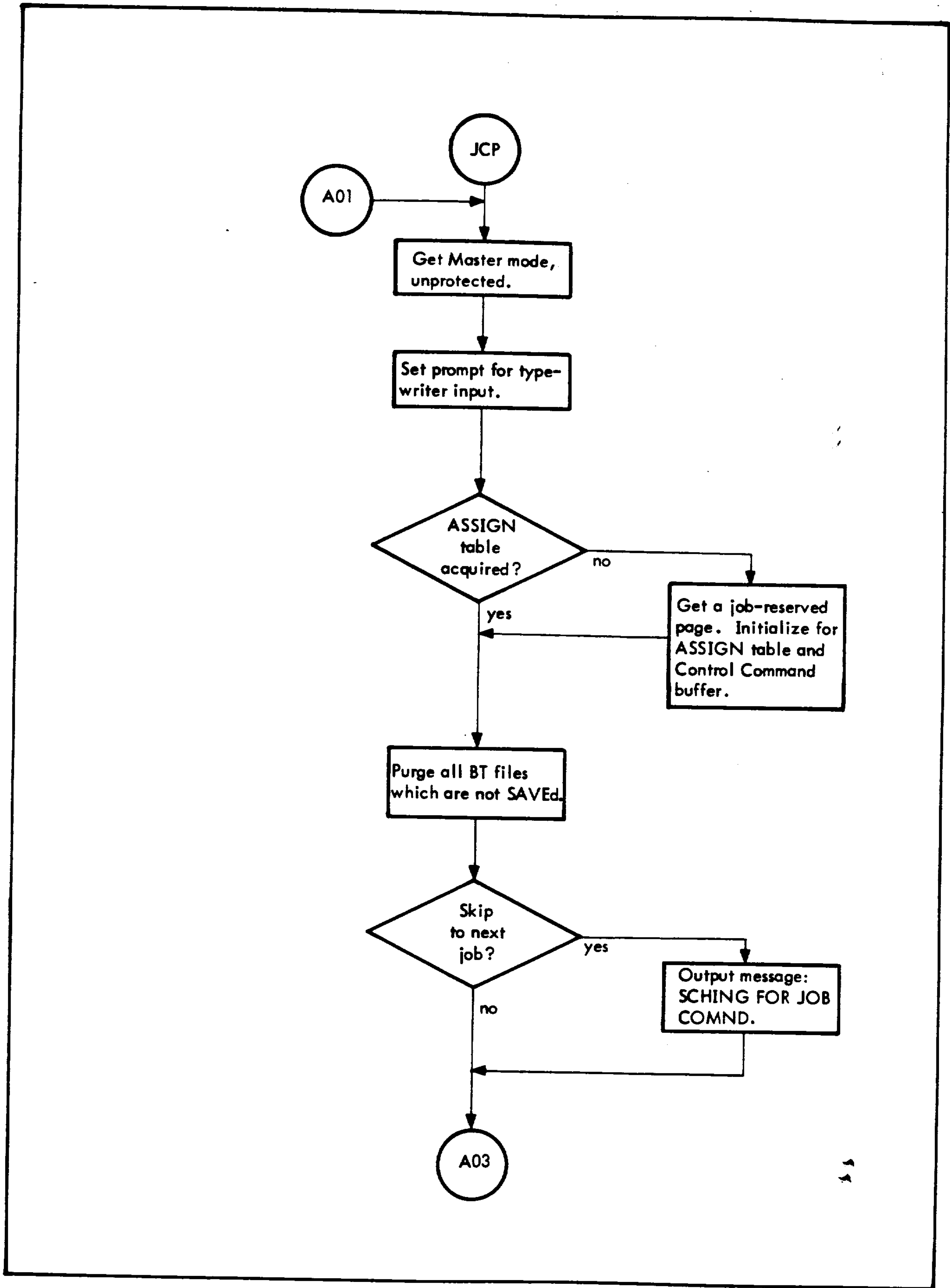



Figure 18. Initialize JCP

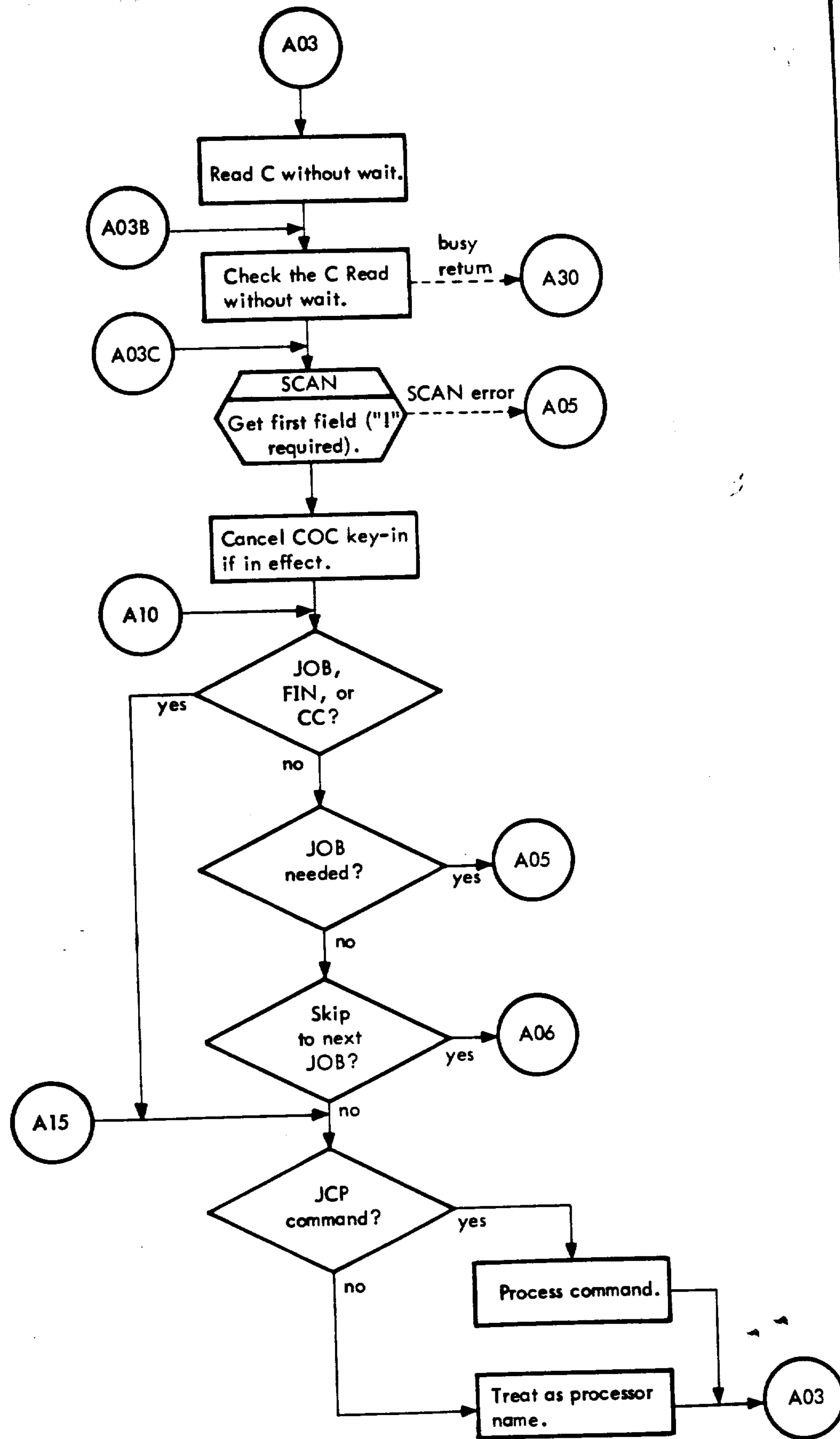



Figure 19. Read and Process JCP Commands

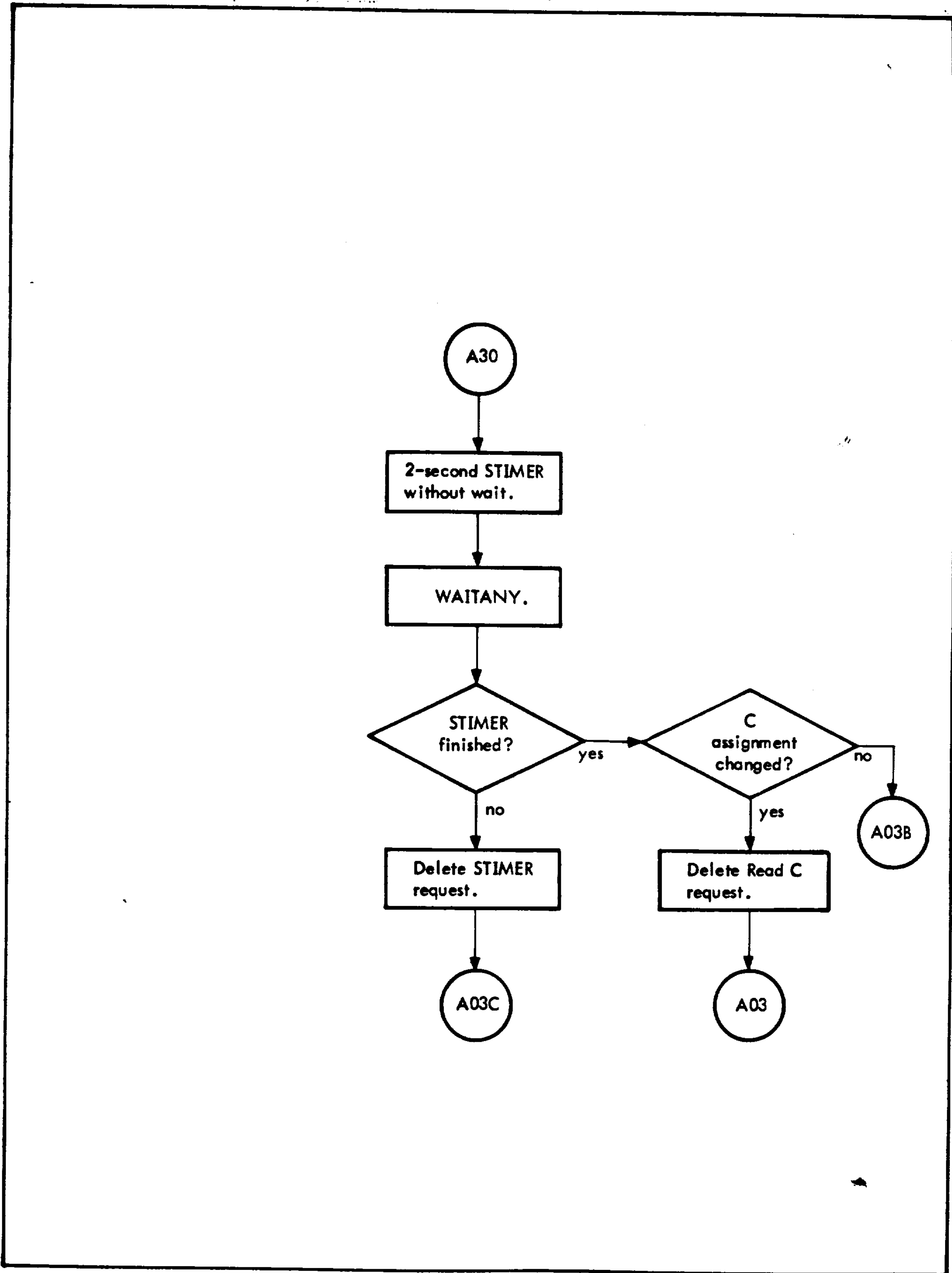



Figure 20. Wait for JCP Command

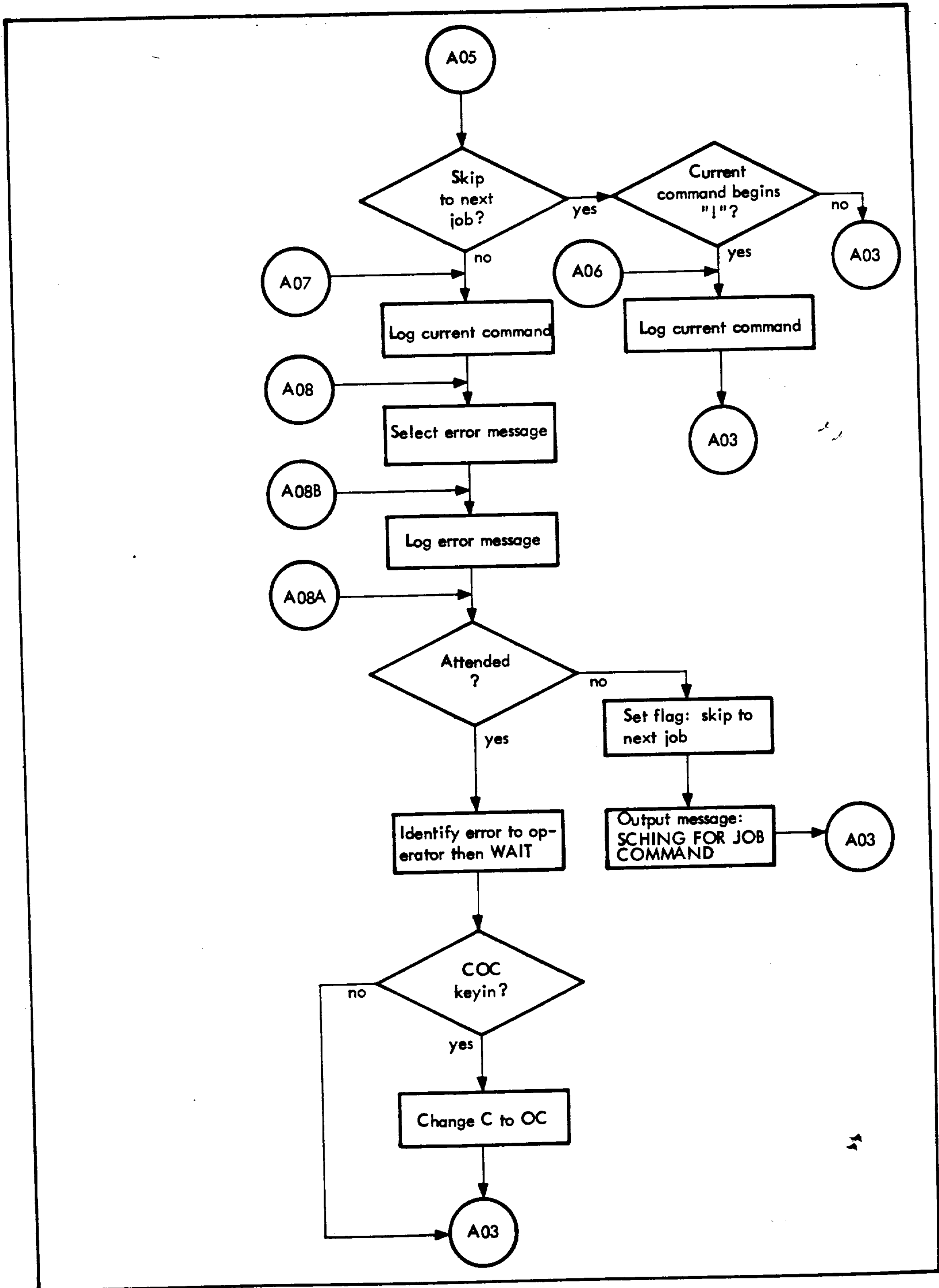



Figure 21. Process JCP Command Errors

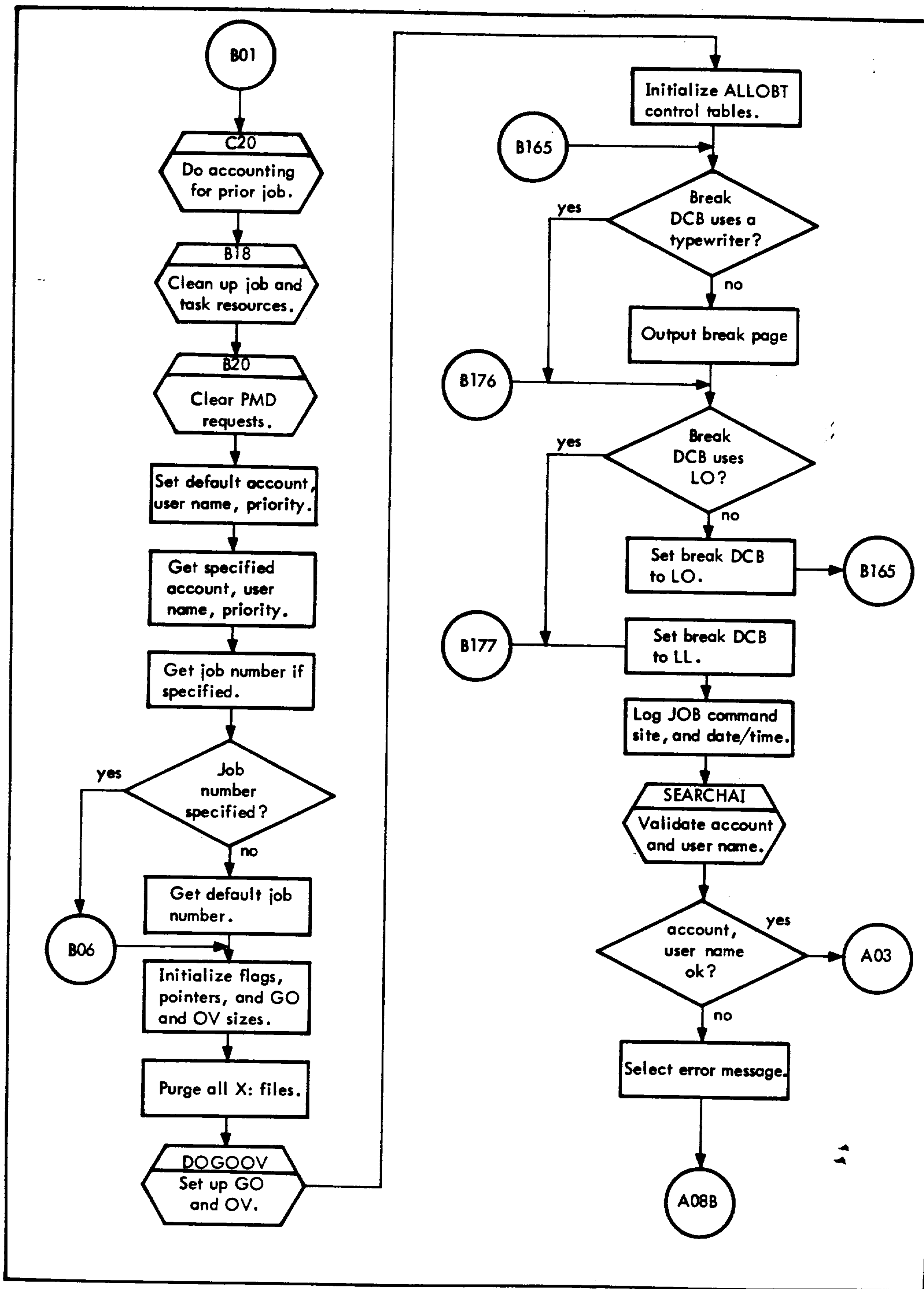



Figure 22. JOB Command Flow

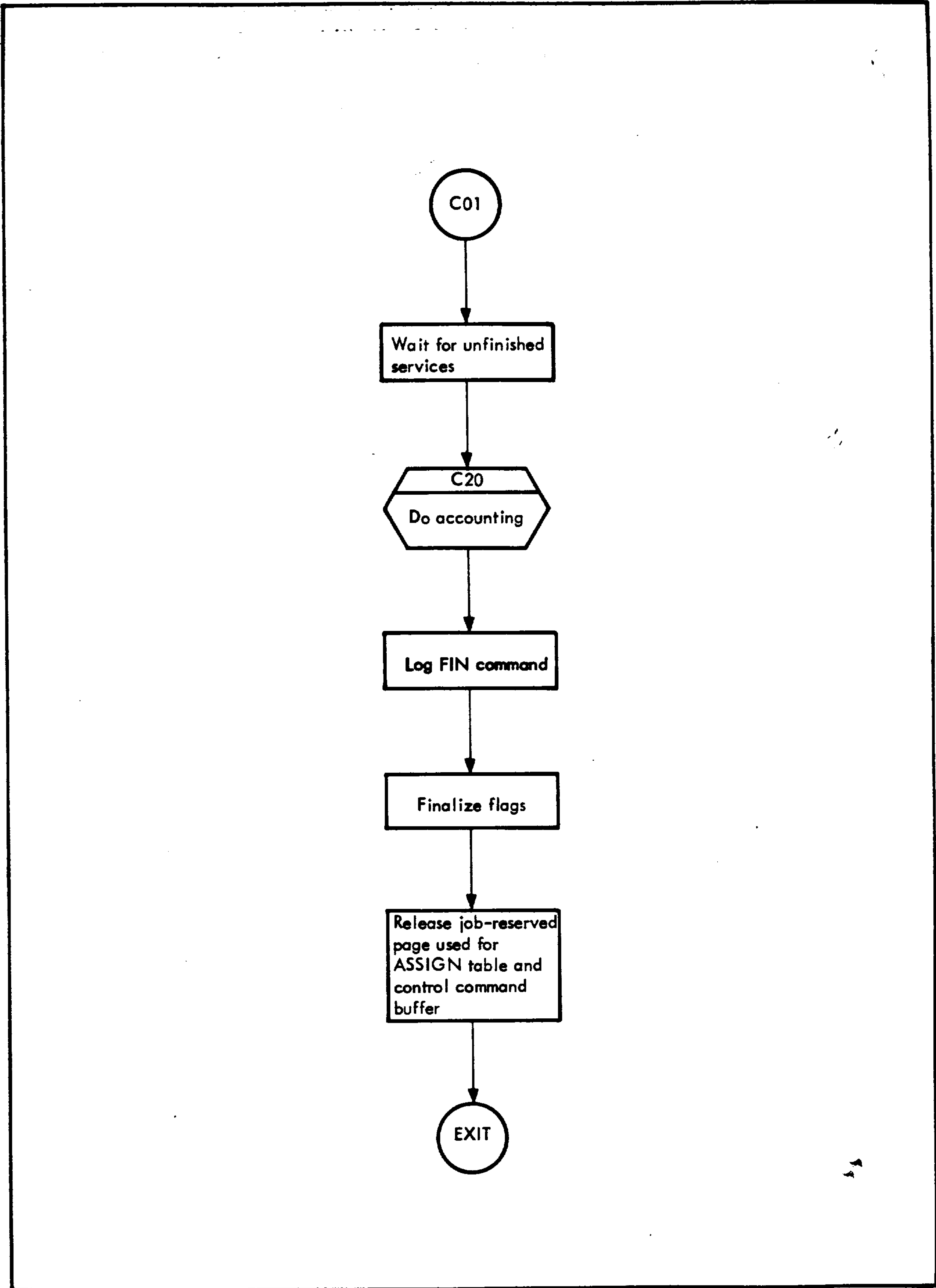



Figure 23. FIN Command Flow

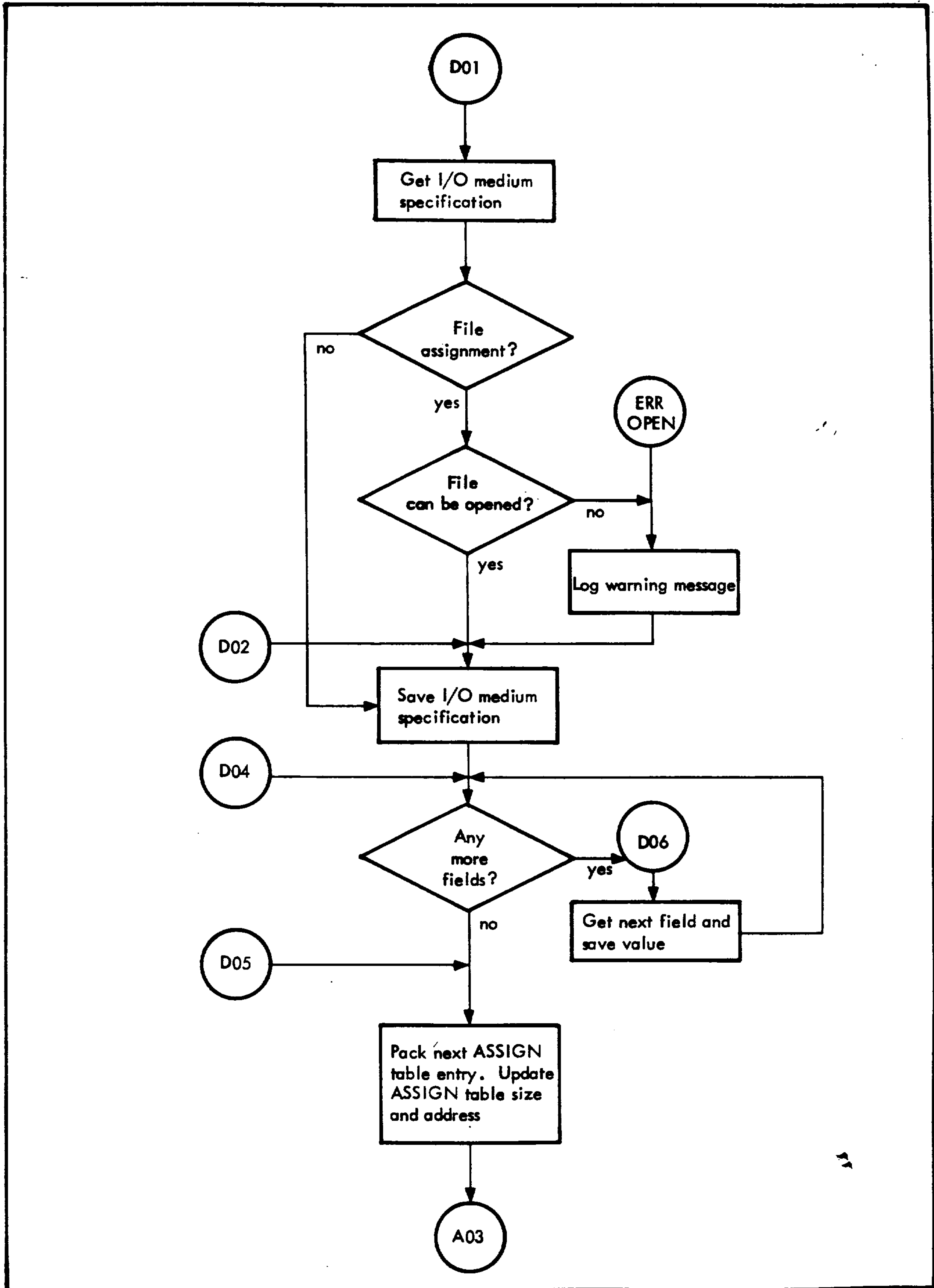



Figure 24. ASSIGN Command Flow

E01

Format and print accounting log on LO device.

E10 → Enter here when EOF returned from Accounting Log

If purge option, purge AL file by rewinding AL and write an EOF.

Exit from DAL command

A03

Figure 25. DAL Command Flow

F01

Set attend mode flag.

Exit from ATTEND command

A03

Figure 26. ATTEND Command Flow

G01

Set flag not to wait after message is output.

G02

Output message on "OC" device.

Exit if MESSAGE command

A03

PAUSE command

Set idle bit. Execute STOP CAL.

Figure 27. MESSAGE Command Flow

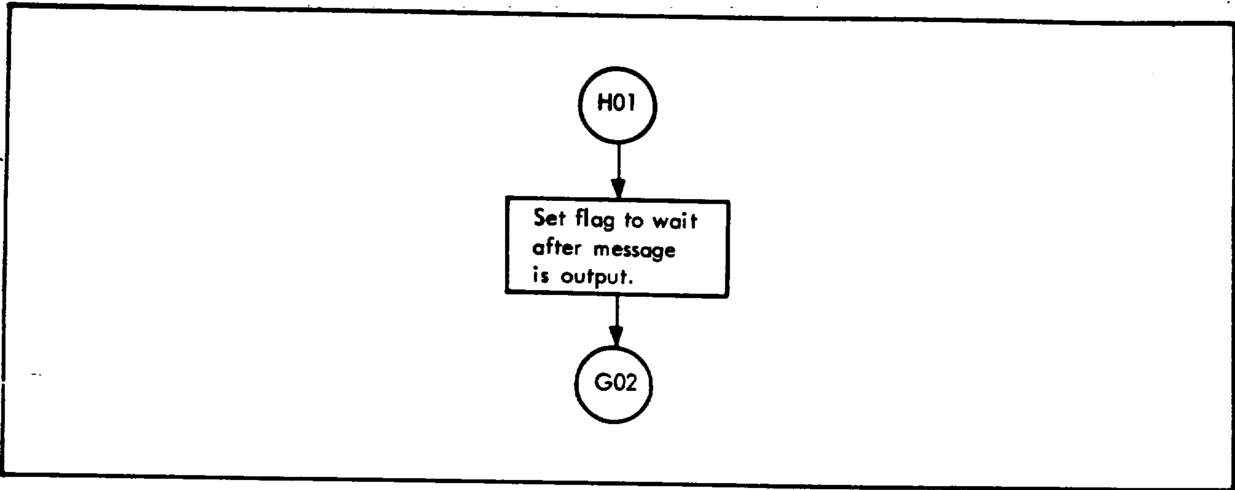



Figure 28. PAUSE Command Flow

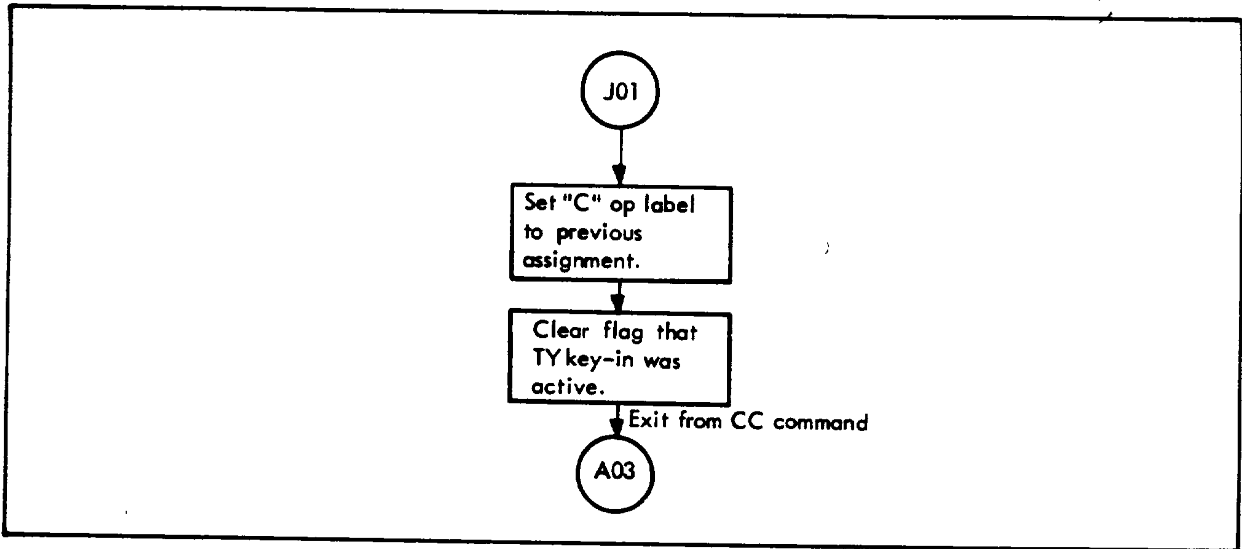



Figure 29. CC Command Flow

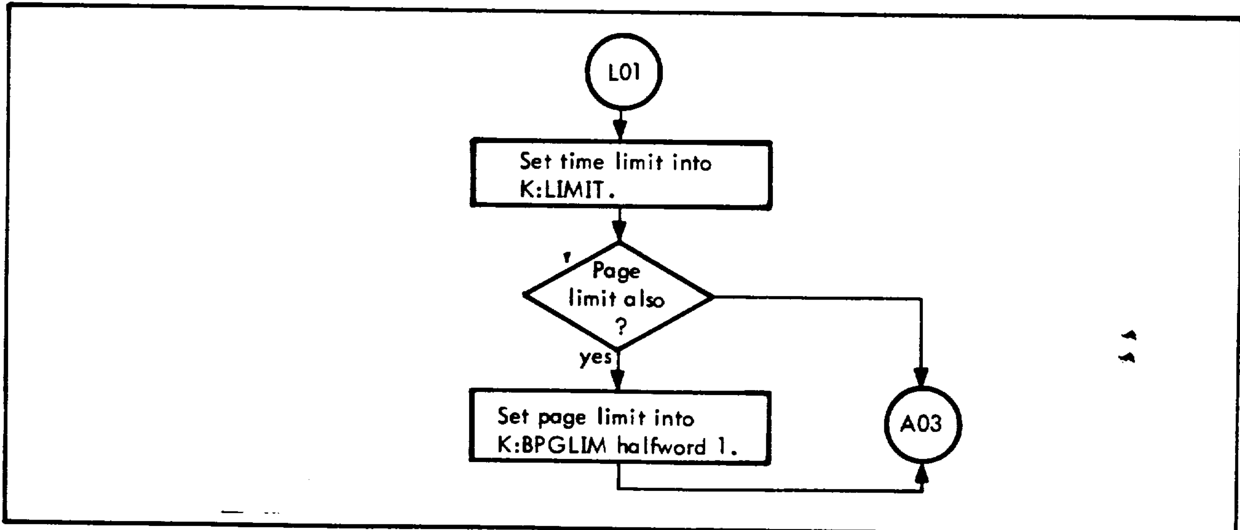



Figure 30. LIMIT Command Flow

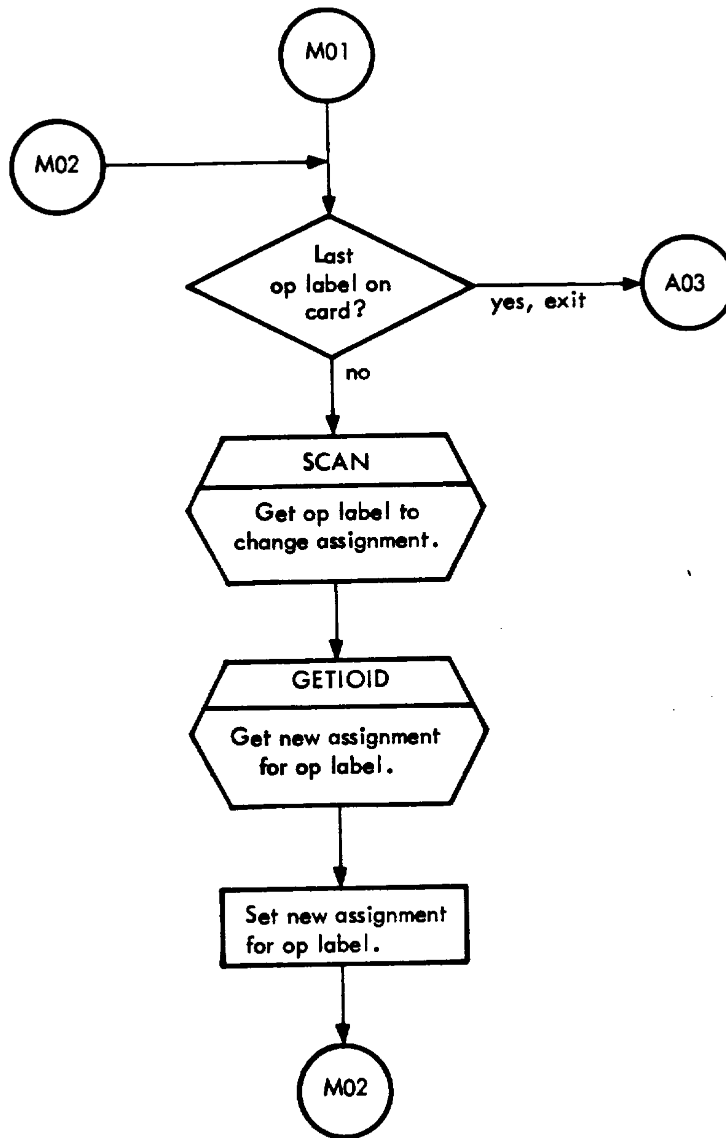



Figure 31. STDLB Command Flow

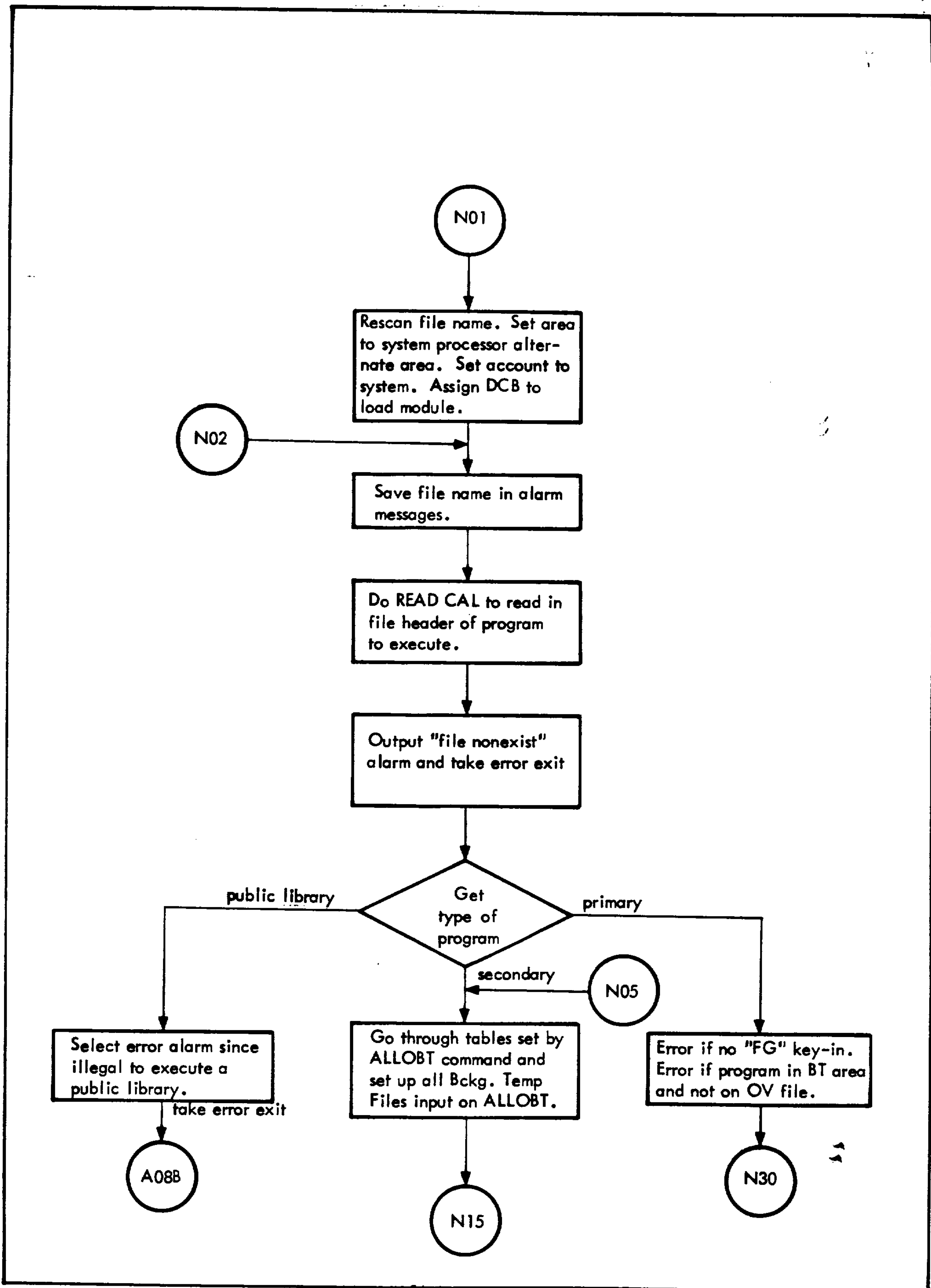



Figure 32. NAME Command Flow

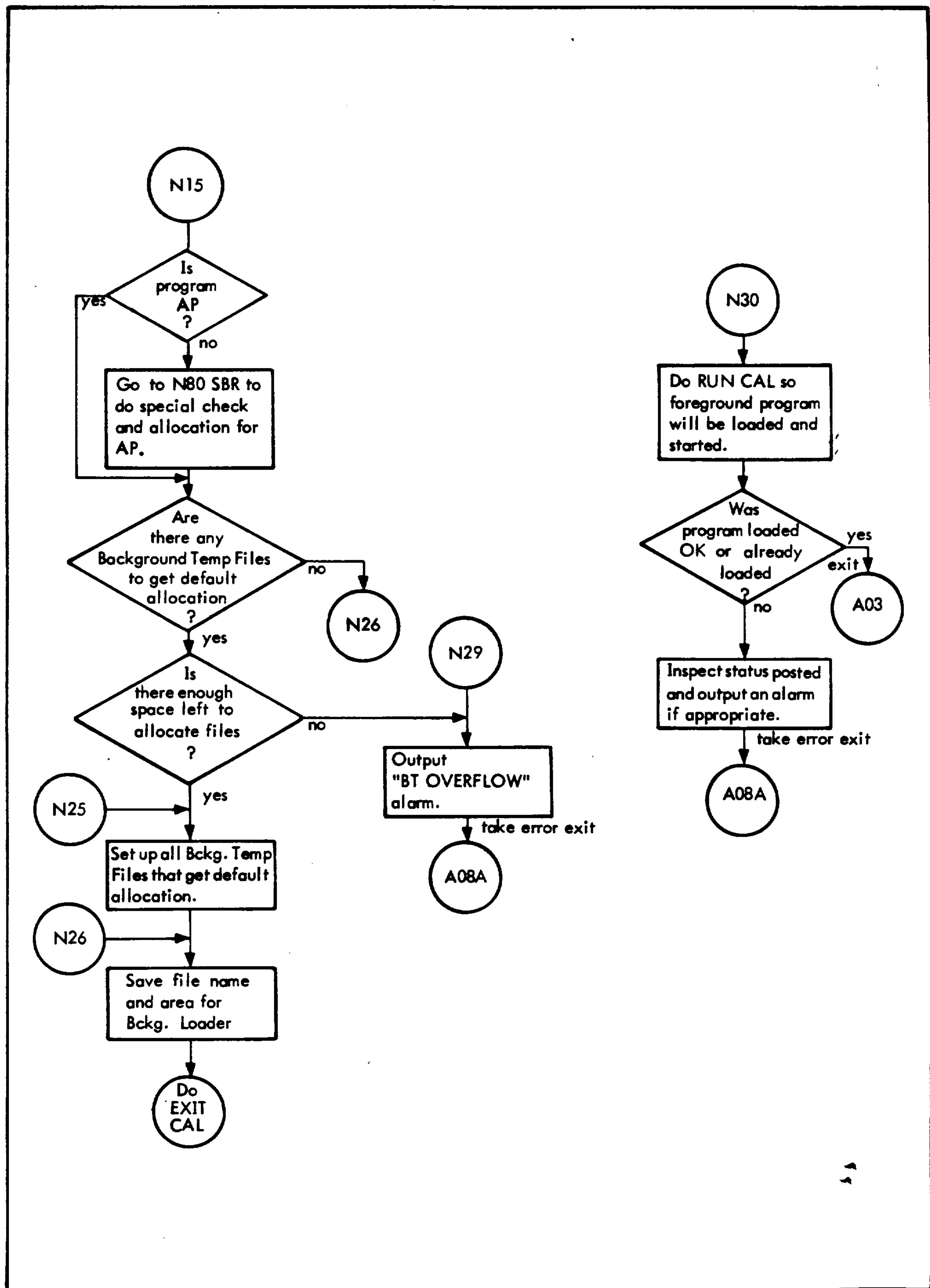



Figure 32. NAME Command Flow (cont.)

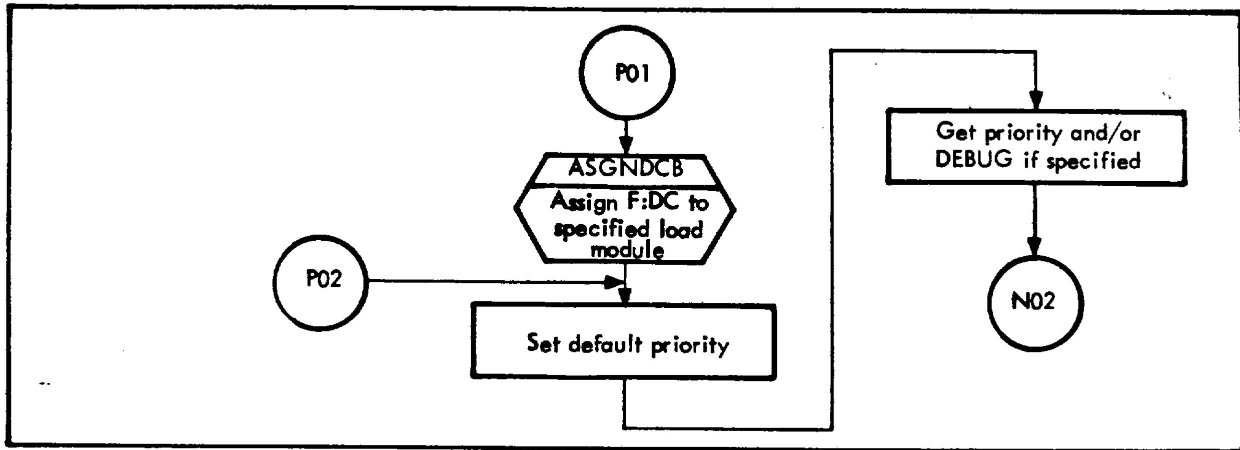



Figure 33. RUN Command Flow

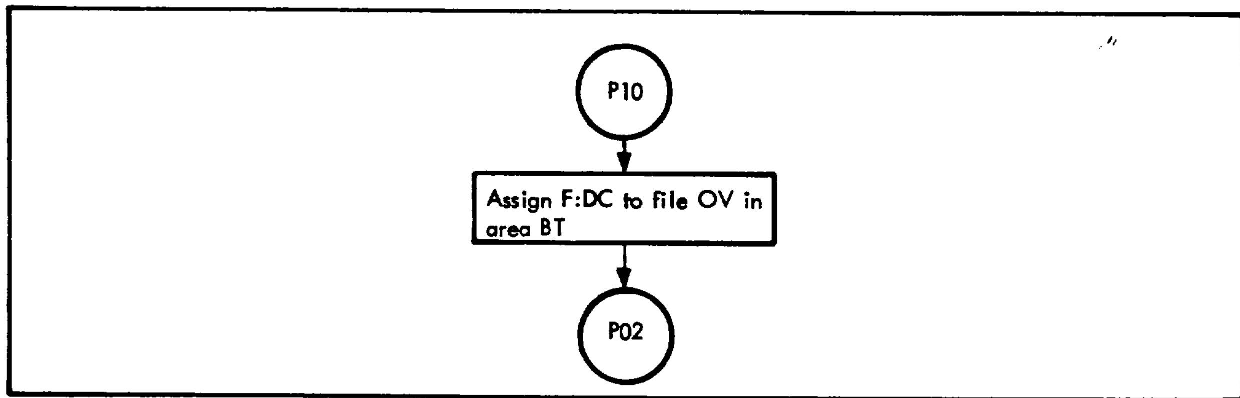



Figure 34. ROV Command Flow

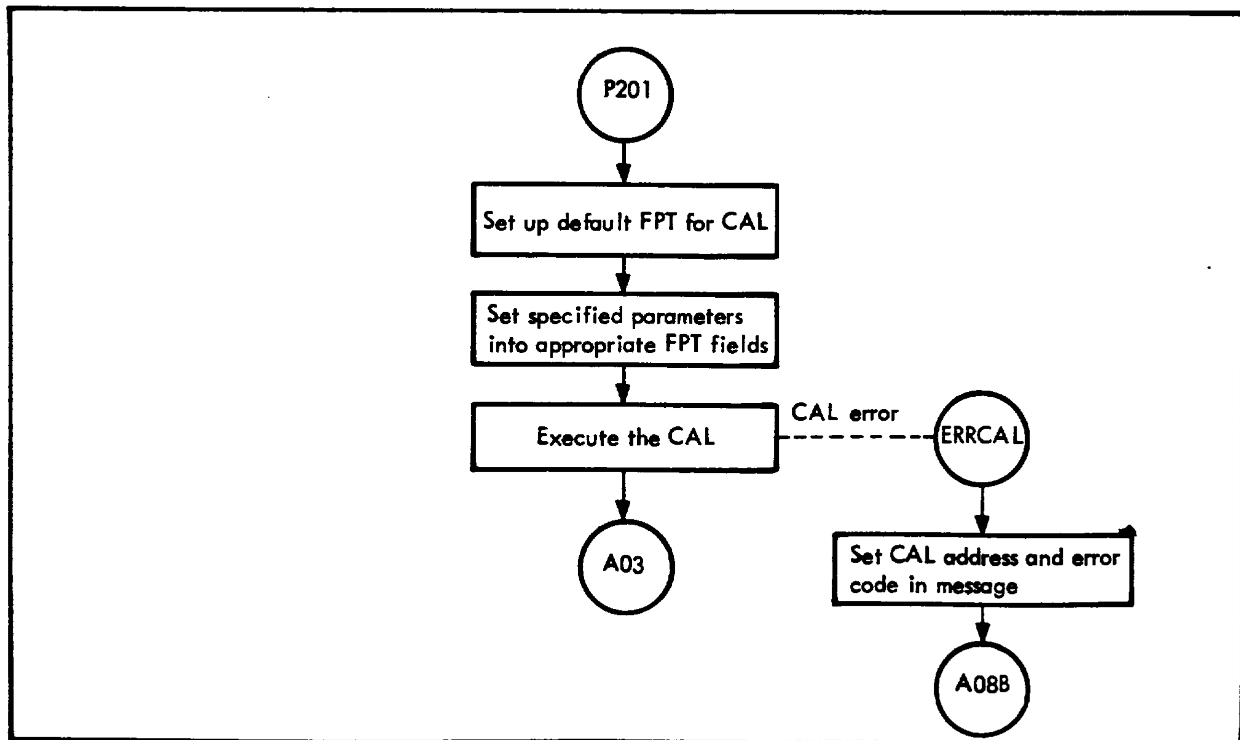



Figure 35. INIT, SJOB, or BATCH Command Flow

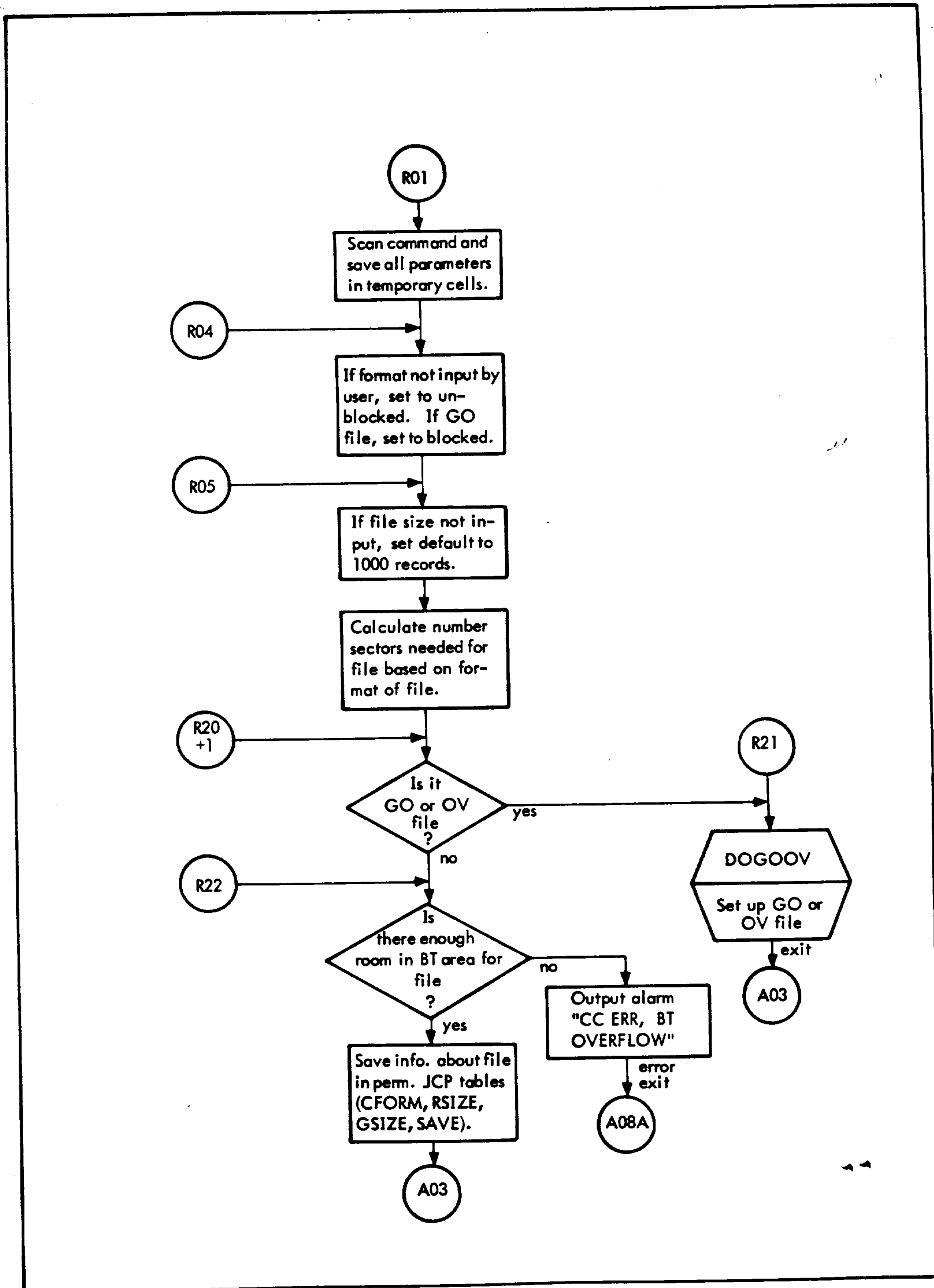



Figure 36. ALLOBT Command Flow

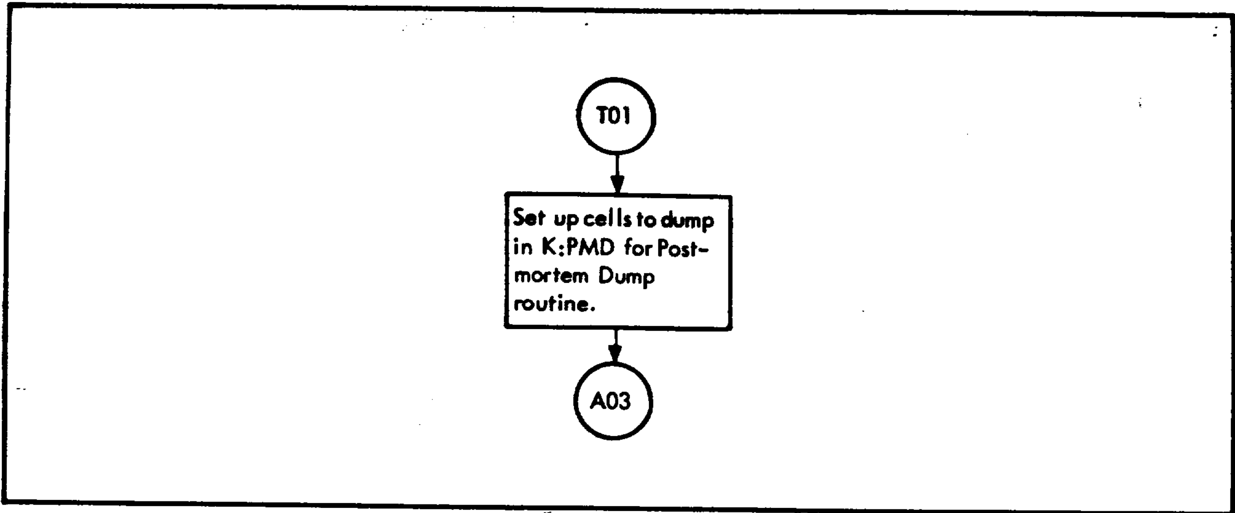



Figure 37. PMD Command Flow

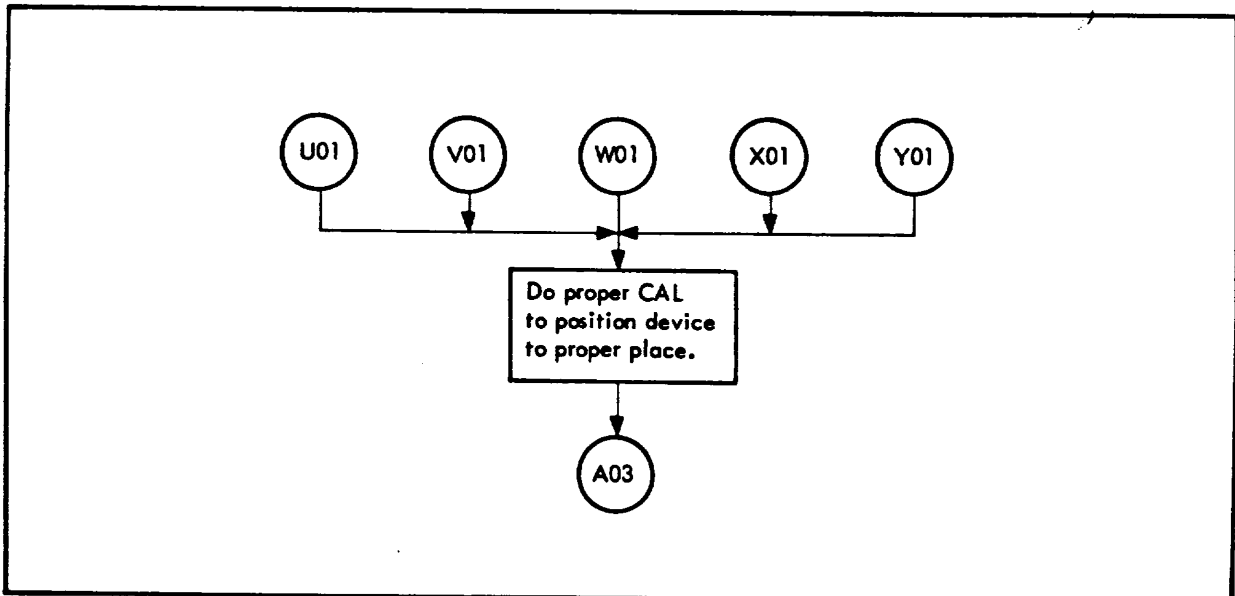



Figure 38. PFIL, PREC, SFIL, REWIND, and UNLOAD Command Flows

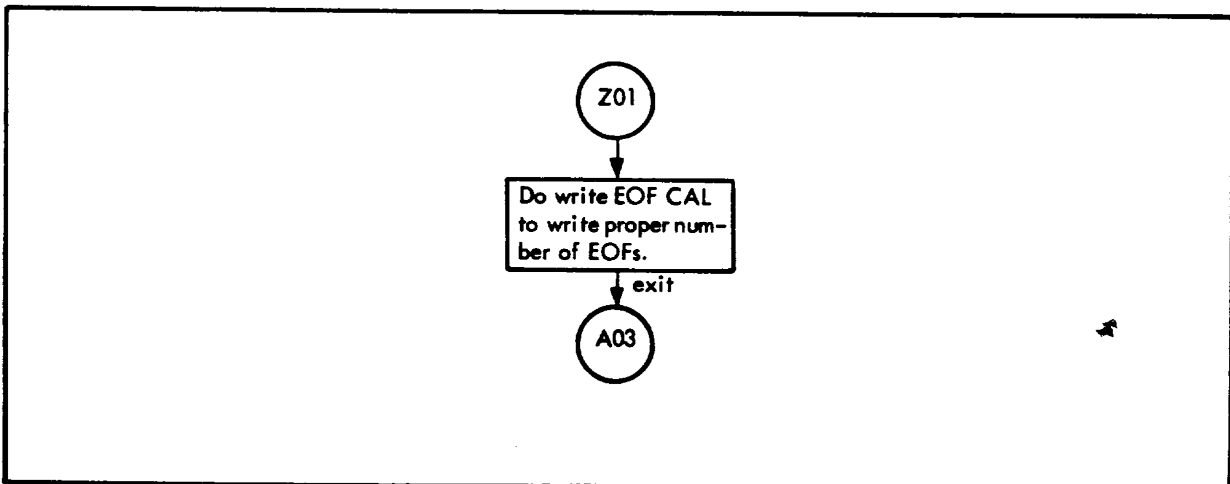



Figure 39. WEOF Command Flow

Table 1. ASSIGN Table Description

| Word | Description |
|---|---|
| 0 | Contains number of entries in table. Must fall on an odd virtual address. Is pointed at by K:ASSIGN. |
| 10n+1 thru 10n+10 | Entry number n, described in more detail below. |
| 10n+1, 10n+2 | EBCDIC name of DCB associated with entry n. |
| 10n+3 | Flags controlling I/O medium name representation. Bit 1 is set for an oplabel name, right-aligned in word 10n+4. Bit 2 is set for a device name left-aligned in words 10n+4, 10n+5. Bit 3 is set for a disk area name right-aligned in word 10n+4, and a file name left-aligned in words 10n+5, 10n+6. If a file name is all blank or all zero, a whole disk area is indicated. Bit 13 is set for a disk file account name in words 10n+7, 10n+8. |
| 10n+4 thru 10n+8 | The EBCDIC name of an I/O medium, formatted as indicated by flags in word 10n+3. |
| 10n+9, 10n+10 | Indicator flags and values for changes to DCB fields other than those identifying the I/O medium. |
| 10n+9 | 1V \| 1C \| 2V \| 2C \| 3V \| 3C \| 4V \| 4C \| 5V \| 5C \| 6V \| 6C \| 7V \| 7C \| (unused) |
| 10n+10 | 8V \| 8C \| (unused) |
| | kC: if reset kV is unused; if set, kV is to be inserted.<br>1V: value for MOD field<br>2V: value for ASC field<br>3V: value for DRC field<br>4V: value for D/P field<br>5V: value for VFC field<br>6V: value for BTD field<br>7V: value for NRT field<br>8V: value for RSZ field |

## JCP Loader

The JCP Loader loads Relocatable Object Modules (ROMs) or groups of object modules that use a subset of the Xerox Sigma 5/7 Object Language. Initially, the Loader processes all parameters on the ILOAD command and sets up the appropriate DCBs and flags. If the program being loaded has overlays, space is reserved for the program's OVLOAD table at the end of the JCP Loader. The OVLOAD table contains 11 words for each overlay; the first word of OVLOAD contains the number of entries in the table. The exact format of the OVLOAD table is given in the "CP-R Table Formats" chapter. Note that words 2 through 10 of the OVLOAD table have the same format as the Read FPT that is needed to read an overlay into core. Next, the first word addresses of the Symbol table (SYMT1 and SYMT2) are set up. The diagram in Figure 40 depicts the core layout before PASS1 of the JCP Loader.

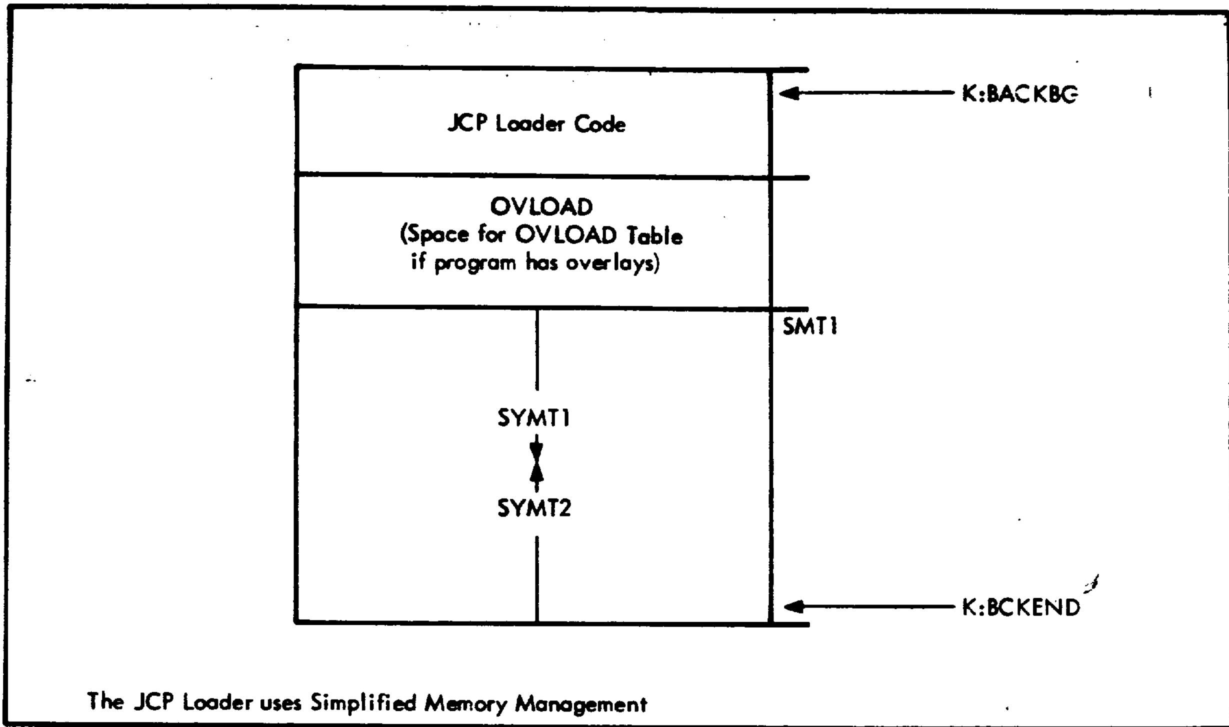
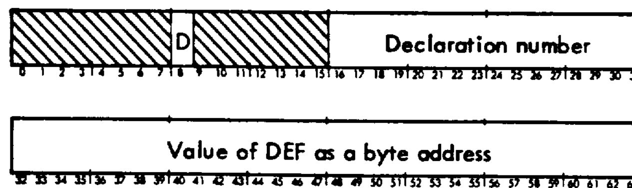



Figure 40. Pre-PASS1 Core Layout

The JCP Loader is a two-pass loader. In PASS1, the ROMs are input from the BI op label and copied onto the X1 file on the disk. The X1 file is set up to use all of the Background Temp area of the disk that is available for scratch storage. The main function of PASS1 is to build the symbol table (SYMT1 and SYMT2) containing all DEF items, and to assign a value to each DEF. The symbol table has the following format:

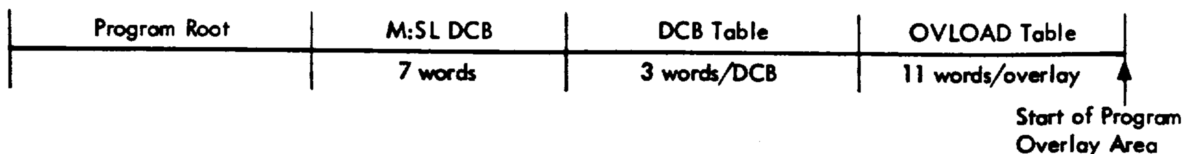
SYMT1 a doubleword-entry table containing the names, in EBCDIC, of each DEF item in the program being loaded. The first entry is not used.

SYMT2 a doubleword-entry table. The first word of the table contains the total number of DEFs in the table. The subsequent entries have the following format:



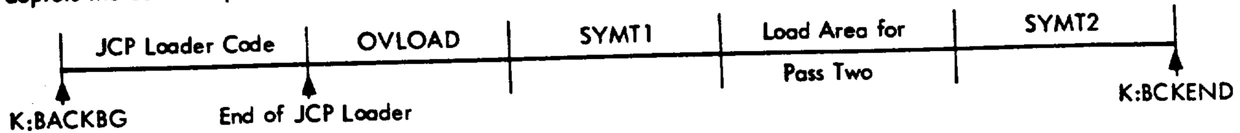

where bit 8 = 1 if this is a duplicate DEF.

At the end of PASS1, the size of the symbol table is fixed so the remainder of core can be used as a load area in PASS2. After loading the program root in PASS1, space is allocated for the M:SL DCB (if the program has overlays), the DCB table, and the OVLOAD table (if the program has overlays). These items are allocated in the following order:



The DCB table is built in an internal table in the JCP Loader in PASS1 after loading the program root. The DCB table is made up of all M: and F: DEFs in the root, including the value of each DEF. The complete OVLOAD table is also built during PASS1; each overlay's entry being made after the overlay is loaded. Hence, PASS1 completely allocates all space for the program.

After the last ROM is loaded at the end of PASS1, the file header is written to the appropriate disk file. The remainder of core not used by the Symbol table is then rounded down to an even multiple of disk granules and set up as the load area for PASS2. There must be enough room to hold at least one disk granule, plus 12 extra words, or the load will be aborted at this point. The X1 file is then rewound and PASS2 commences. The following diagram depicts the core setup at the start of PASS2:




PASS2 inputs the ROMs from the X1 file, satisfies all external REFs by finding the value of the corresponding DEF in the Symbol table, and then writes the program in core image format to the proper disk file in a multiple of granules at a time. Between 8 and 12 extra words are loaded each time at the end of the load area in case a define field load item requires that the load location be backed up a maximum of 8 words. This prevents having to read a granule back into core after it has been written in the event a word has to be changed because of a define field item.

These 12 words are copied from the bottom of the load area to the top of the load area after the granules are written on the disk. The previous 8 words are therefore always available in core to satisfy a define field item.

After the root has been loaded in PASS2, the M:SL DCB (if appropriate), the DCB table, and the OVLOAD tables are attached in that order to the end of the root and written on the disk. After all ROMs have been loaded, the JCP Loader outputs the map if requested, closes all files, and exits to JCP.

## Job Accounting

Job accounting is an option selected at SYSGEN time. An accounting file will be kept on the disk by the JCP if the accounting option was chosen. The accounting file is named AL, and resides in area D1. It is automatically alloted by INIT.

Whenever a !JOB or !FIN command is read by the JCP, the JCP will update the AL file for the previous job. The format and record size of the AL file is automatically set by the JCP via a File Mode CAL. The JCP defines the AL file as a blocked file with a record size of 32 bytes. The AL file on the disk consists of a series of eight-word records, where a new eight-word record is added for each job. The format of each record in the AL file is as follows:

| Word | Description |
|---|---|
| 1,2 | Account number in EBCDIC |
| 3,4,5 | Name in EBCDIC |
| 6 | Left halfword = (year - 1900) in binary, Right halfword = date as day of year (1 - 365) |
| 7 | Start time of job in seconds (0 - 86399) |
| 8 | Elapsed time of job in seconds |

Whenever an entry is added to the AL file, the file is opened and a file skip performed so that the new entry can be made at the end of the existing entries. No attempt is made to combine entries in any way. The contents of the AL file can be listed via the !DAL command, (Dump Accounting Log), and the option exists for the user to purge the file after the dump is completed. The AL file is purged by rewinding it and writing an EOF.

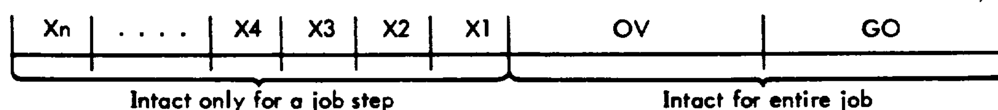
## Background TEMP Area Allocation

The JCP allocates and sets up the files in the Background Temp (BT) area (X1-X9, GO, OV) before exiting to the Background Loader to load a processor or user program. The BT files needed by the user are defined either via !ALLOBT commands or through default by the JCP from inspection of the user's DCBs. The GO and OV files are set up at the start of each job and remain intact for an entire job; the required files X1 through X9 are normally set up for each job step only.

Information for files X1-X9 read in from !ALLOBT commands is stored in tables (GSIZE, FSIZE, FORM, SAVE, RSIZE) that are internal to the JCP. If the GO or OV file is changed via an !ALLOBT command, the file is re-defined at the time the command is processed.

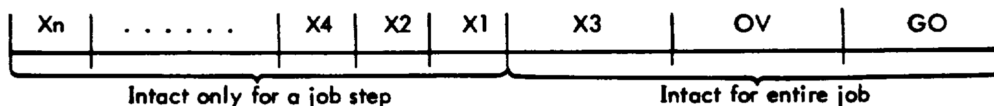
The files in the BT area are allocated so that files remaining intact only for that job step are allocated at the front of the BT area. Files that remain intact for the entire job are allocated at the back of the BT area. Normally, this means that X1 through X9 are allocated at the front of the BT area, and GO and OV at the opposite end. If the SAVE option is used on an !ALLOBT command for an Xi file, the Xi file will be allocated at the opposite end of the BT area, as will GO and OV. The following diagrams illustrate the BT allocation:

BT allocation without !ALLOBT Commands:




The proper Xi file is allocated for each M:Xi DCB in the user program. The remainder of the BT area after GO and OV have been allocated is evenly divided among the Xi files.

BT allocation with !ALLOBT Command:




The above diagram illustrates how BT would be allocated if an !ALLOBT command was input to save the X3 file. Note that X3 is allocated at the opposite end of the area with OV and GO.

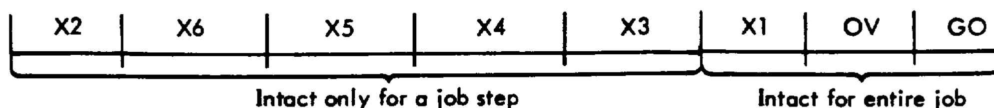
Allocation of the Xi ($1 \le i \le 9$) files is performed in the following sequence: First, any files input on an ALLOBT command are allocated at the proper end of the BT area. Next any Xi files that were not input on an ALLOBT command are allocated by default in the remaining area. Note that if the "ALL" option is used for file size in the ALLOBT command, there will be no room remaining for default allocations.

The following example depicts the allocation of BT as previously described:

Example 1:

1. An !ALLOBT command for X1 file with SAVE option.
2. An !ALLOBT command for X2 file.
3. The system was SYSGENed with (BT, 6) on the RESERVE command.

In this case, the BT area would be allocated as



In this example, the X1 and X2 files would receive the sizes input on the !ALLOBT command, while the X3, X4, X5, and X6 files would be evently distributed over the remaining area.

The JCP does special allocation of the BT area for the AP and MACRSYM processors, since the scratch space requirements of these processors depend on the parameters of their calls and the space is unevenly divided among files involved. This special allocation is done by the use of nonstandard allocation-control tables when JCP is invoked to run either the AP or MACRSYM processor in the background. Other special allocation tables could be added for other processors requiring nonstandard allocations.

# 6. FOREGROUND SERVICES

Foreground services are those service functions restricted to foreground utilization. In general, they are associated with the control of system interrupts, the handling of foreground tasks, and direct I/O (IOEX). The following service functions fall in this category:

RUN/INIT

RLS/EXTM

MASTER/SLAVE

STOPIO/STARTIO

IOEX

TRIGGER

ENABLE/DISABLE

ARM/DISARM

CONNECT/DISCONNECT

In terms of the functions as part of the resident CPR, the resident function sets indicators for RUN and RLS, and the Control Task actually performs the function.

## Implementation

**RUN** If an entry for the specified program does not already exist in the LMI table, an entry is built. The LMI subtables are set as follows:

| | |
|---|---|
| LMI1 | Program name |
| LMI2 | Group code for interrupt to be triggered at conclusion of initialization by Control Task |
| LMI3 | Group level for said interrupt |
| LMI4 | Signal address and (optionally) priority |
| LMI5 | Switches |

K:FGLD is set nonzero, the Control Task is triggered and control is returned to the user program.

If an entry does exist in the table for the program, a code is placed in the signal address. The codes used are

| | |
|---|---|
| 3 | Program already loaded |
| 4 | Program waiting to be loaded |

If no entry exists for the program and there are no free entries in the LMI table, a code of 5 is placed in the signal address. Sufficient reentrance testing is performed (for details, see the program listing).

**RLS** If an LMI entry does not exist for the specified program, control is returned to the user.

If an entry exists and the program is not loaded, LMI1 and LMI5 are zeroed, and control is returned to the user.

If an entry exists and the program is loaded, a flag in LMI5 is set, K:FGLD is set nonzero, the Control Task is triggered, and control is returned to the user (for details of reentrance testing, see the program listing).

**MASTER/SLAVE** The mode bit in the PSD saved in the user Temp Stack is set to the proper state and control is returned to the user. When returning control, CALEXIT executes an LPSD that establishes the proper mode for the user.

**STOPIO/STARTIO** The specified device is determined and all other devices associated with it (all other devices on a multidevice controller or all devices on the IOP if the call so requests) have their proper STOPIO counts incremented or decremented. The count is either in DCT14 or DCT15 as specified by the call.

An HIO is performed on these devices if requested by the call.

If a DCT15 count goes to zero as a result of a decrement, the IOEX busy bit in DCT5 (bit 7) is reset for the device.

**DEACTIVATE/ACTIVATE** The specified device is determined, and it and all other devices associated with it (all other devices on a multidevice controller, or all devices on the IOP if the call so requests) are marked "down" (Deactivate) or marked operational (Activate). An HIO is always performed on these devices for a Deactivate request.

**IOEX** For TIO and TDV instructions, the instruction is executed and the status is placed in the copies of R8 and R9. The condition code field of the saved PSD is placed in the Temp Stack. Then at CALEXIT, these copies are placed in R8, R9, and the PSD, and returned to the user.

For SIO, the IOEX bit (DCT5, bit 7) is tested. If the IOEX bit is set the SIO is executed and status and condition codes are returned to the user. If the IOEX bit is not set, the request is queued and status is returned to the user indicating that the SIO was accepted. The user obtains actual status by specifying end-action. Various registers contain pertinent status at that time.

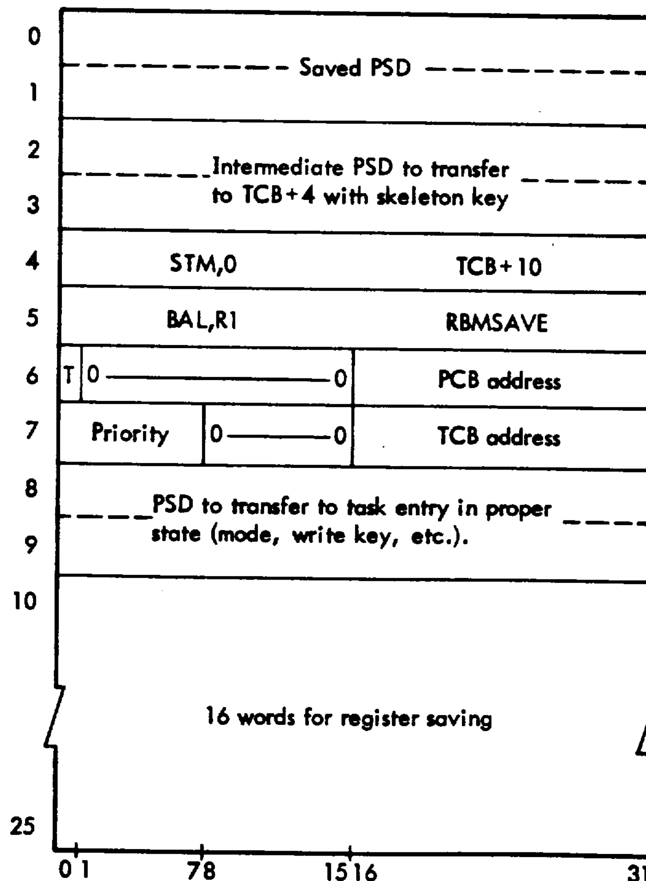
For HIO, the IOEX bit (DCT5, bit 7) is tested. If the bit is set, the HIO is executed and status and condition codes are returned to the user. If the IOEX bit is not set, the monitor routine RIPOFF is called which will eliminate any ongoing or queued requests for the device. The user receives status and condition code settings which indicate the HIO request was accepted.

**TRIGGER, DISABLE, ENABLE, ARM, DISARM, CONNECT, DISCONNECT** These functions are similar in that they involve the execution of a Write Direct after determining the group code and group level of the specified interrupt.

In addition, a task connection is performed if requested by ARM, DISARM, and CONNECT requests. Note that the CONNECT call is a special case of the ARM call. The logic for ARM, DISARM, and for CONNECT functions is illustrated in Figure 41.

### Task Control Block (TCB)

The CONNECT function initializes words 2-9 of the user-allocated TCB for interrupts and CALs that are to be centrally connected. The format of the TCB is shown below:



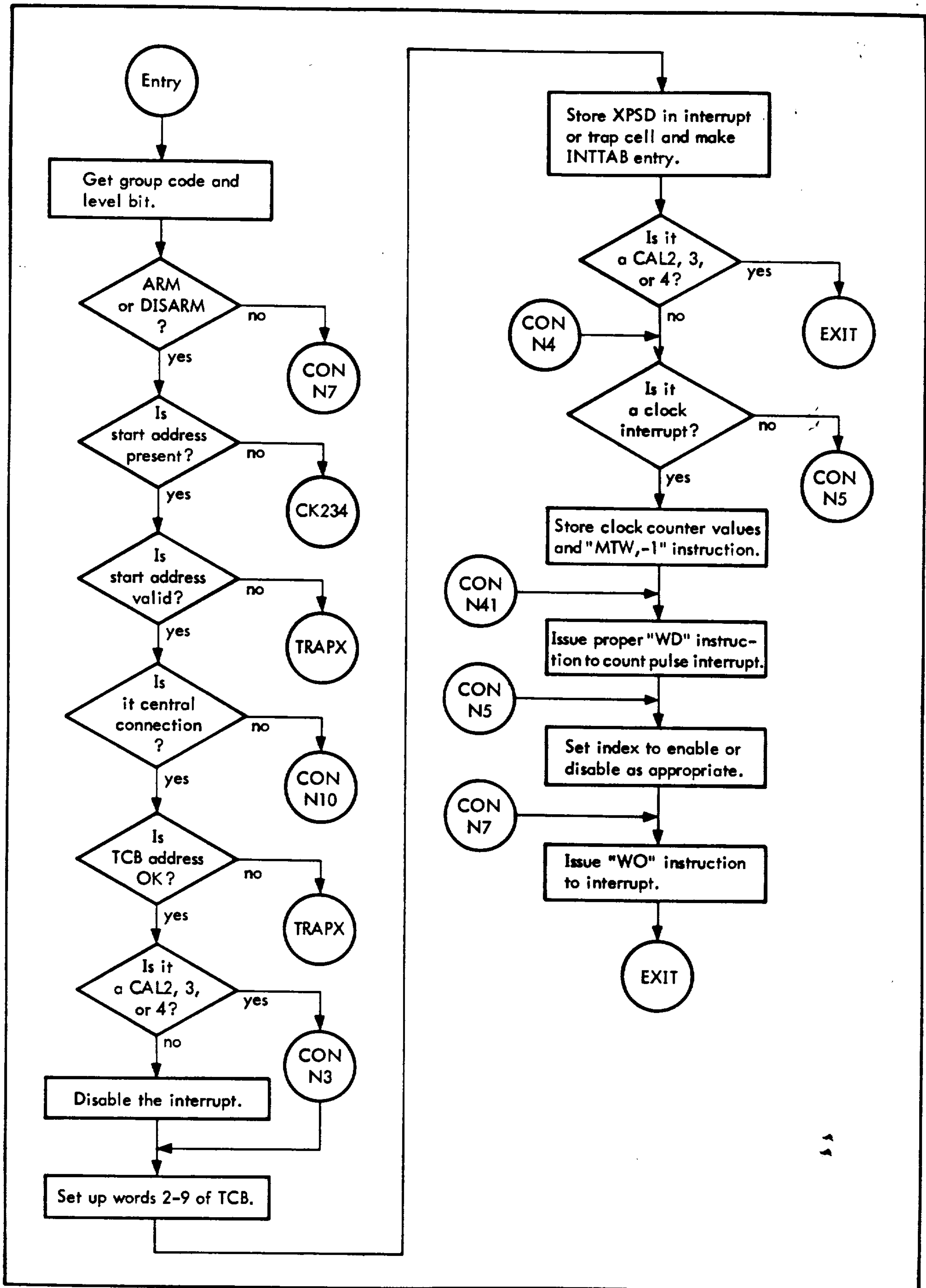



Figure 41. ARM, DISARM, and CONNECT Function Flow

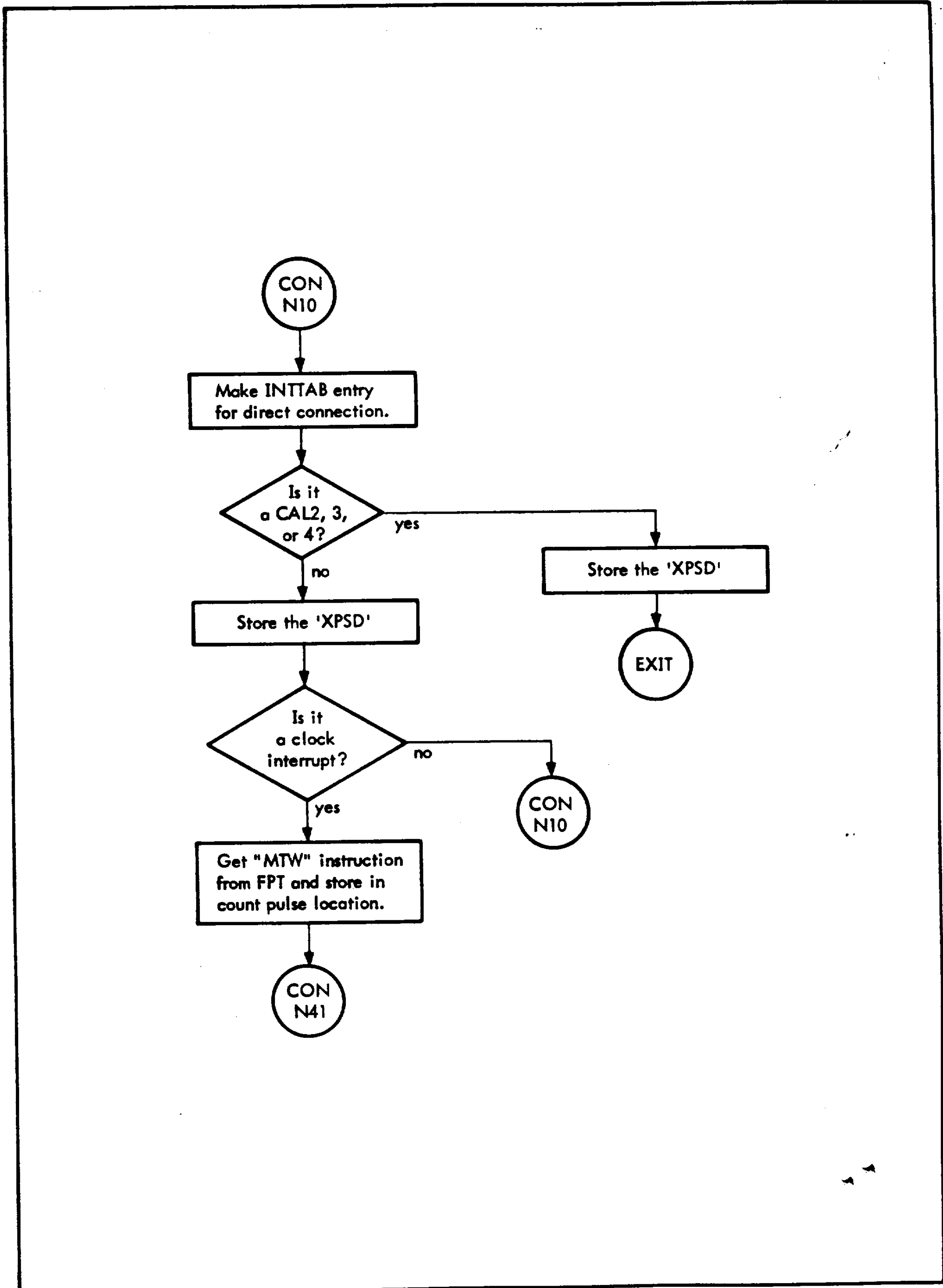



Figure 41. ARM, DISARM, and CONNECT Function Flow (cont.)

# 7. MONITOR INTERNAL SERVICES

## CP-R Overlays

All CP-R overlays may be declared to be resident or nonresident at SYSGEN time, in order to increase performance of a particular function or to reduce monitor size, respectively. This is done by means of the :MONITOR control command.

The overlay technique allows a user call for such functions as OPEN and REWIND to bring in an overlay to perform the function. The structure is reentrant (allows multiple users at different priorities to use the overlay area), recursive (allows an overlay to call an overlay), and usable for any monitor function (allows overlays at the control-task level to use the same area as those for user services). The overlay technique employed requires no explicit calls for overlays. When an overlay is needed all that is necessary is a branch to a REF:

```
REF     OEP (overlay ENTRYpoint)

B       OEP
```

SYSLOAD will fulfill these references by having them branch to the Overlay Manager (OMAN) which will load the overlay.

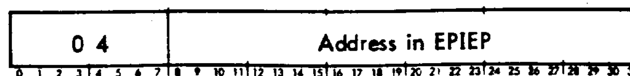
In order to create an overlay the programmer must include DEF's in the overlay ROM for all possible ENTRY points and all possible EXIT points. An ENTRY point is defined as a point at which one would enter the overlay via any type of branching instruction (BAL, BCR, BCS, LPSD, etc.). An EXIT point is defined as a point at which one would exit the overlay with no intention of returning to this overlay without first going through an ENTRY point. For instance, a BAL to a resident subroutine from the overlay would not be considered an EXIT point since a return to the overlay will take place. All EXIT point instructions must be unconditional branch instructions, either B*Rx or B address. This is due to the fact that the EXIT point instructions will be replaced by unconditional branches to the Overlay Manager which may replace the overlay with a previously active overlay and then execute the EXIT point instruction.

An overlay will be named by the first DEF in the module, which must be the first BO-generative statement. As the CP-R ROM and the overlay ROMs are read by SYSLOAD all unsatisfied REFs are assumed to be overlay-load requests and thus are satisfied by creating an entry in the Entry Point Inventory (EPI), described below, and using that address to satisfy the REF.

As the overlays are read, all DEFs are checked for possible ENTRY points or EXIT points. A DEF will be considered an ENTRY point if a previous REF for that name has been located. If a previous REF has not been encountered the DEF will be considered an EXIT point. This algorithm implies that the order of the overlay ROMs as read by SYSLOAD is significant. All overlays which call overlays should do so with forward references.

As each overlay is encountered, its name (the first DEF) is compared against the list of resident or nonresident overlays as defined by the user on the :MONITOR SYSGEN command. If found to be nonresident, the overlay is linked to run in the overlay area and written out to the SP area. If found to be resident, it is linked at the end of the present monitor end and, or course, is written out with the monitor. The last ROMs on the SYSLOAD medium must be the subsystem overlays (currently TEL, LOAD, and JCP) preceded by INIT. Figure 42 shows the general arrangement of the SYSLOAD-input ROMs.

OMAN uses the EPI and OVI tables to make sure the proper overlay is in core at all times. OMAN is activated by a reference to the EPIEP as set up by SYSLOAD. EPIEP contains a CAL1 instruction. OMAN is entered from the CAL1 processor with inhibits set, and examines the address of the CAL1 to calculate the index for EPI if it is an OMAN call. If the address is in the EPIEP table this is a request for an overlay load. If it is in the overlay area and of the form




then it is an EXIT.

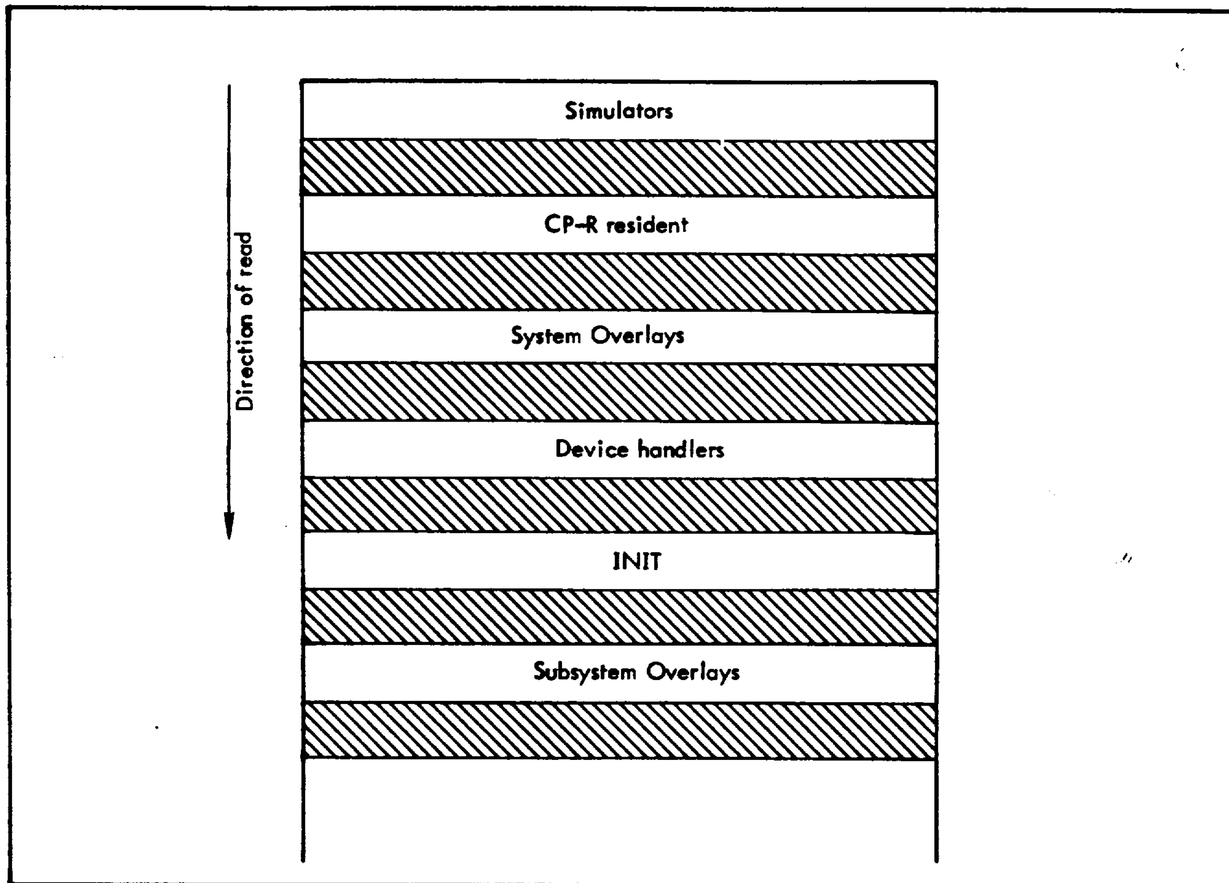



Figure 42. Arrangement of SYSLOAD Input ROMs

For entries, the previously overlay information is stacked, the new overlay is loaded, and control is transferred to the ENTRY address. For an EXIT, previous overlay information is unstacked, the last overlay is reloaded if necessary, and the instruction in EPIEP is executed.

After every activation the active overlay ID (OVI index) is placed in the STIOV field. When an exit takes place the STIOV field is cleared. EXIT checks STIOV to see if the task to which it is exiting has an active overlay. If it does and the presently active overlay for the system is not the same, EXIT forces an entry to OMAN to reload the active overlay for the task. (This is done at the level of the task which is being exited to.)

This overlay technique has several unique aspects which should be noted:

- Any reentrant piece of code which is entered via a branching type instruction and exited via an unconditional branch may be converted to an overlay simply by

  - Assembling it as a separate ROM.
  - Placing a REF where a branch to it takes place.
  - Placing a DEF for the ENTRY point in the ROM (first DEF also used as overlay name).
  - Placing a DEF for the EXIT points in the ROM.

  The system overhead incurred by this conversion is only one instruction when the resultant overlay is declared resident.

- No registers are destroyed in loading and transferring control.

- Many such pieces of code may be placed into one overlay.

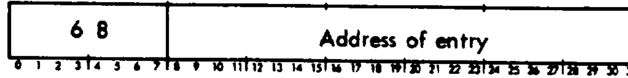
## Entry and Exit Point Inventory (EPI)

Purpose: The EPI is used to intercept all entries to overlays and to save all exit instructions from overlays in order that the Overlay Manager (OMAN) can load the proper overlay.

Type: Parallel in CP-R table space with a fixed number of entries. Generated by SYSLOAD.

Logical Access: The EPI index is, in essence, generated by SYSLOAD. When SYSLOAD encounters a reference to an entry point, the address is replaced by the address of an EPI entry (EPIEP). When an exit point is encountered the entire instruction is replaced by a CAL1 instruction.

EPIEP: An EPI table entry can have one of three forms. If the entry is an ENTRY point to a resident overlay:




If the entry is an ENTRY point to a nonresident overlay:

If the entry is an exit point:

(This is the actual instruction that was in the overlay and has been replaced by a CAL1 with an effective address of the replaced instruction.)

## Overlay Inventory (OVI)

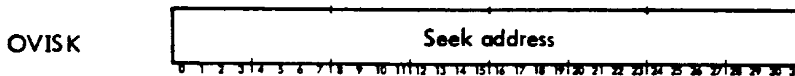
Purpose: The OVI replaces the table previously defined as OVLOAD. It is used by OMAN to load overlays for both primary and secondary tasks. For each overlay it contains the sector address, length, and name.

Type: Parallel in CP-R table space with a fixed number of entries. Generated by SYSLOAD.

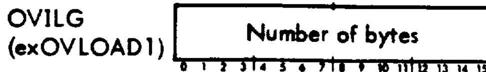
Logical Access: The EPI (Entry and Exit Point Inventory) has a subfield of EPIEP which indexes the proper overlay for that Entry Point.

Entries:

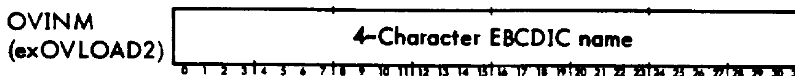
OVISK

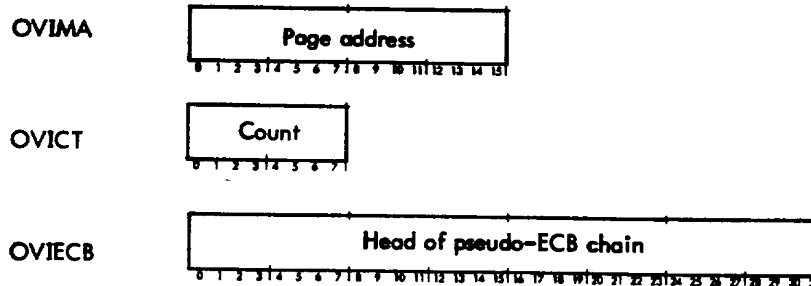



OVILG (exOVLOAD1)




OVINM (exOVLOAD2)






where

OVISK is the seek address of the overlay on the device containing the SP area.

OVILG is the length of this overlay in bytes (≤2048).

OVINM is a 4-character EBCDIC name representing the first DEF in the overlay. This is the name used in the SYSLOAD map and the name to be used for all communications about the overlay.

OVIMA is the page address of the overlay if it is in core. The value is zero if it is not in core.

OVICT is the use-count field used to determine which overlays should remain in core in roll-out circumstances.

OVIECB chain head of a list of two word pseudo ECBs. The first word contains the forward link. The second word contains the ID of the task which is waiting for receipt of the overlay.

## Event Control Block and Event Control Services

Purpose: Event Control Blocks (ECBs) provide task management and CAL processors with the mechanism for controlling system services explicitly requested by tasks or invoked by CP-R.

Type and Location: ECBs are eight-word serial control blocks in TSPACE, with chained data areas also in TSPACE.

Logical Access: ECBs are members of two chains and can be located only via one or the other of these chains. The chains are as follows:

Solicited ECB chain – A chain headed in the LMI entry corresponding to the task for which the event is being performed. The chain head is in LMISECB.

Request ECB chain – A chain generally headed in the LMI entry corresponding to the task performing the service. If no one specific task is responsible for posting, the R- chain is either not used or is headed elsewhere.

### Overview of ECB Usage

Asynchronous or synchronous (vs. immediate) service requests must create ECBs to control the event processing. Asynchronous or synchronous service calls are those performing functions which require waits for some other logic within the processor or external event to complete prior to completing the original request. They are as follows:

| | | | | |
|---|---|---|---|---|
| RUN | SEGLOAD | UNLOAD | TYPE | DFM |
| INIT | OPEN | WEOF | ALLOT | DEVN |
| ENQ | CLOSE | PFIL | TRUNCATE | ACTIVATE (MM) |
| SIGNAL | READ | PREC | DELETE | |
| STIMER | WRITE | DEVICE | STDLB | |
| POLL | REW | PRINT | GETPAGE | |

In addition to the above CAL processors, CP-R tasks may create and use ECBs to control their own scheduling and communicate with other modules. These tasks are as follows:

Task Initiation

Task Termination

Key-in Processors

Memory Management executive including ROLL-IN and ROLL-OUT

CAL Processor Usage

The CAL processor will create and initialize the ECB. If the service is requested with wait, the CAL processor will loop waiting for the ECB to be posted if the caller is primary, or set the ECB and dispatcher controls for secondary tasks and return to the dispatcher. A posting phase is executed when the ECB is posted. A checking phase is performed following the post. The completion data is returned to the user and the ECB deleted. The CAL processor then exists.

If services are requested without wait by the user, the CAL processor creates and initializes the ECB and starts the service to the extent possible until a wait would occur. The CAL then returns to the caller. Some time later a posting phase is executed. The caller must eventually issue a CHECK on the service. Failure to do so would cause the ECB to remain 'active' until task termination. When the CHECK call is performed, the service is processed until a roadblocked condition occurs or the service is done. If the service completes, the cleanup is done as above and control returned to the caller. If the service is still not complete, the busy exit will be taken if it was provided. If no busy exit was provided, the system waits for the service to complete as described above, then does the cleanup and exits.

Note that the order of posting and checking is variable. A post may precede the execution of a check.

Task-Termination Usage

Task termination keys on the ECBs during its initial phases. Each ECB must be posted before the task is allowed to terminate and release its core resources. The termination routines drive the ECBs to completion as rapidly as possible by calling special subroutines for each ECB type. It then does a WAITALL on the ECBs.

## ECB and Data-Area Formats

Figure 43 shows the detailed format of an ECB and gives an example of chained data areas.

Description of the individual data elements follow.

ECBDATA (Word 0)

Length: The length of the first data area in the chain, in words.

Data area address: The address of the first data area. Initially, this word is set to zero. If a data area is added to the ECB, the length and address (as returned from the GETTEMP) are stored here, and the first word of the data area is zeroed. Subsequent data area additions continue to store this word into the first word of the newest data area and put the new control in the first word of the ECB. Data area deletions do the inverse, namely, move the first word of the data area being deleted (always the first in the chain) into this word.

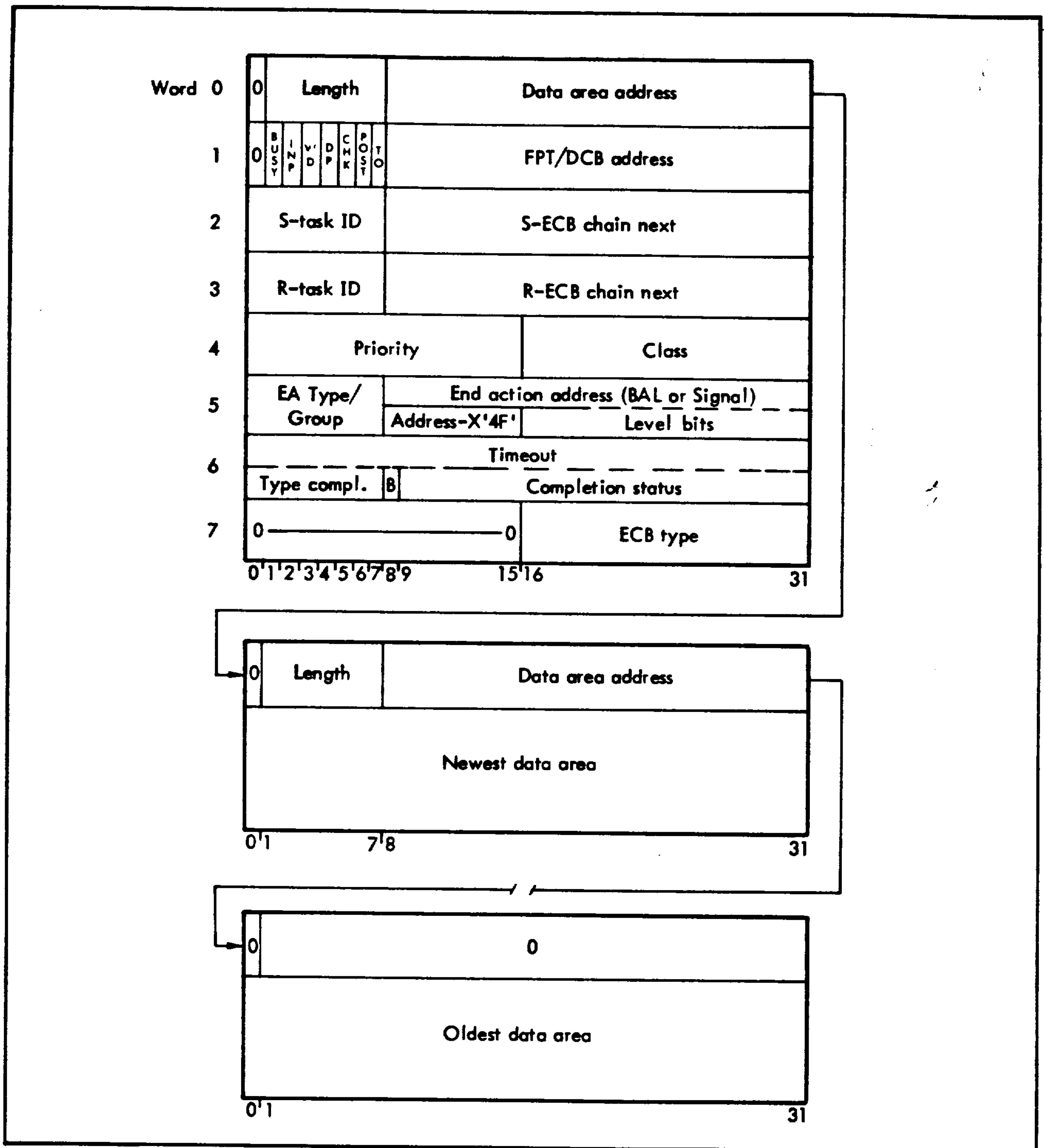



Figure 43. ECB Format and Chained Data Areas

ECBFPT (Word 1)

Flag bits as follows:

| | | |
|---|---|---|
| Bit 0 | | Reserved |
| BUSY (bit 1) | = 1 | if the ECB has not been posted. This means that word 6 contains the timeout threshold, if any. |
| | = 0 | if the ECB has been posted. This means that the type of completion and completion status have been stored over the timeout threshold in word 6. |

INP (bit 2) = 1 if the ECB is 'in-process'. This bit is set during a POLL, check phase, to avoid subsequent polls from acquiring the same ECB.

=0 if the ECB activity has not been initiated.

In-process may be set by internal CP-R tasks which do not use a POLL to indicate that the ECB is being operated upon.

WD (bit 3) = 1 The wait count in the STI entry of the S-task is to be decremented by one (if it is not already zero) when the ECB is posted. If the count becomes zero due to the post, the dispatcher should be triggered and the task entered if the S-task is a higher priority than the posting task. If it is lower, the dispatching is deferred.

= 0 Do not alter any dispatch controls at posting. The task is not waiting for the ECB.

WD is set by the EMWAIT subroutine and WAITANY, and WAITALL calls. It is reset by posting. It is also reset by WAITANY after gaining control on a multivalued wait.

DP (bit 4) = 1 Delete the ECB as soon as the posting logic is complete. The user does not expect to check the FPT nor does he require feedback of the type of completion.

= 0 Do not delete the ECB until after the checking/cleanup phase is complete.

DP is set on service calls with Delete-on-Post set (F8 = 1), and on service calls that generate ECBs but are not CHECKable. On all other ECBs, it is reset.

CHK (bit 5) = 1 Checking is in process on this ECB by some task, and other checking phases are not to be allowed. This bit is set by service call processors when requested with wait. It is set by CHECK CAL entry before going to the ECB-type-dependent checking routine. It is set by TEST, WAITANY and WAITALL when processing the ECB through checking phases. It is reset by EMWAIT when taking a busy exit. CHECK tests the bit prior to setting it. If nonzero, the CHECK is rejected as invalid and the busy exit is taken if provided. If not provided, the calling task will be trapped. TEST, WAITANY and WAITALL ignore ECBs in the S-chain with the CHK bit set.

POST (bit 6) = 1 Posting is in process on this ECB. Other posting operations are not allowed. This bit is set by the posting subroutine entry prior to entering the ECB type-dependent logic. If POST is already set, an error exit is given to the caller. POST is reset by checking phases if the ECB is 'unposted' to allow additional processing phases.

Note that if POST = 1 when an ECB is created, no posting operation will be allowed. If CHK = 1 when an ECB is created, no checking operations will be allowed.

TO (bit 7) = 1 Timeout of the ECB is in process and other timeout operations are not allowed. The proper ECB posting routine will be called.

FPT/DCB address: This is the address of the caller's original FPT (or DCB in the case of Type-I I/O). On all CHECK or DELFPT service calls, this serves as the control field to locate the ECB which represents the service being checked. It also allows the WAITANY, WAITALL and TEST calls to know the location of the original FPT or DCB in order to build an internal check FPT. An FPT/DCB address must be stored in all ECBs at creation. If the FPT was in registers, the register address (0-F) is stored.

ECBSECB (Word 2)

S-Task ID: The task-ID of the task that solicited the service or that is checking the service.

S-ECB Chain Next: The address of the next ECB in the solicited-ECB chain of the S-task.

As a task requests asynchronous services, the ECBs created are added to the end of a chain which is headed in the LMI entry corresponding to the task. This provides the system with knowledge of all the outstanding service requests for a load module. On checks or deletes, this chain is used to search the S-ECBs. It is also used by Task Termination, WAITANY, WAITALL and TEST to define all the services in process. The S-chain is maintained as ECBs are created and deleted. The S-task ID tells the chaining logic, indirectly, in which LMI S-chain to place the ECB. More importantly, at posting time, it tells the EMPOSTYC subroutine, whose task controls, to update if wait decrement is set.

ECBRECB (Word 3)

R-Task ID: The task ID of the task that is to provide the requested service and that will post the ECB, if any.

R-ECB Chain Next: The address of the next ECB in the request-ECB chain of the R-task.

Some events are directed to one CP-R task or user load module that is to provide the service and post the ECB. This task is called the responsible task and has a chain (R-chain) through all ECBs currently directed to him, which is headed in the LMI entry corresponding to the task. CP-R tasks will have a load-module-inventory entry to head these chains. The chain is in priority order, with the newest requests at the beginning of their priority group. The chain is used by POLL to locate requests and give them to the task for processing. It is also used by POST to validate the ECB identification in the FPT. Internal CP-R tasks may use the R-chain directly to locate and operate on request ECBs. The R-chain is maintained as ECBs are created and posted. The R-task ID tells the standard R-chain maintenance routine, indirectly, in which R-chain the ECB is to be placed, or removed.

In the following cases, an R-task can be identified:

- INIT requests – Task Initiation on behalf of the initiated task.
- SIGNAL requests – The task signalled.
- ACTIVATE/GETPAGE – Memory Management Executive.

In some cases, the service is provided in such a way that a specific task cannot be identified which provides the service. In these cases, the R-chain is either not used, or is headed in some other control table, not an LMI. The following ECBs are this type:

- ENQ requests – Service provided by the DEQ CAL processor. The R-chain is headed in an EDT.
- STIMER requests – Service provided by the clock-4 interrupt processing. No R-chain is used.
- POLL requests – Service provided by the SIGNAL CAL processor. The R-chain is not used.
- I/O requests – Service provided by the I/O – interrupt processing. Instead of containing R-task information, bits 0-7 contain the service-call FPT code and bits 15-31 contain the byte count.

ECBPC (Word 4)

Priority: The priority of the ECB as requested by the caller. Generally it will default to the caller's priority. Priority is used to determine the order of the R-chain. It also will become the execution priority of tasks which poll for the R-ECBs according to the description in the POLL specification. Priority is set when the ECB is created.

Class: The class mask that is set when the ECB is created. Generally the class will be the default value of X'FFFF'. On polls, this field is logically ANDed with the class specified in the POLL (default is also X'FFFF'). If the result is nonzero, the ECB qualifies for the poll.

Note that for I/O requests, word 4 instead contains clean-up information (see IOQ13, word 1).

Memory Management ECB's contain control information in bits 16-31 of word 4.

ECBENDAC (Word 5)

The end action for posting, as follows:

| | |
|---|---|
| Word = 0 | No-end action for service. |
| Byte 0 = 00-0F | End-action contains interrupt-trigger data. The interrupt group is the value in byte 0. |
| Byte 0 = 7F | End-action contains a completion signal address. |
| Byte 0 = FF | End-action contains an address to be BALed to at post time. |

End-Action Address: The entry location for BAL-type end action or signal address.

End-Action Address and Level: The address of the interrupt – X'4F' – and level bits for a write direct on trigger-type end action.

ECBTIME/ECBCOMPL (Word 6)

Timeout: The timeout threshold for busy ECBs. When the value (K:UTIME – timeout) is greater than or equal to zero, the ECB has 'timed out' and CP-R will do a post with the timeout code (X'67'). The posting logic which is a function of ECB type will be entered. If timeouts require special logic, the posting routines must test for the X'67' type of completion and take the appropriate action.

Type Compl.: The type-of-completion code set by the caller posting.

B(Busy): This bit will always be zero after posting.

Completion Status: Actual record size (ARS) for READ/WRITE requests.

ECBCTLS (Word 7)

ECB Type: An integer which represents the type of service which is being provided. This value is set symbolically (for flexibility) by the creator of the ECB and can be altered by the processing logic during the life of the ECB. The system uses the ECB type to control the service-dependent logic as follows:

- When an ECB is to be posted, the routine that wishes to do the post will BAL,R8 EMPOST with the ECB identification in R2. EMPOST will use the ECB type as an index into the byte-table EMPOSTX which provides an index into the word table EMPOSTB. The EMPOSTB entry thus located is a branch to the posting logic for that ECB type, and will be executed. EMPOST uses R7 for the indexing.
- When a CHECK call or DELFPT call is issued, the check service call branches to the check processing for the service type. This entry is derived as above, with EMCHKX + ECB type providing an index to the EMCHKB branch table to the entry point. The ECB identification is in R2. R8 is the return register.
- When a wait occurs for a primary task on an event control block, the ECB type is used as an index to the bit-table EMWAITF. If the bit thus located is 1, the primary-task wait is illegal on the ECB, and the task will be aborted. A zero indicates that the wait is valid and the waiting routine will loop, calling SERDEV and waiting for the Busy bit in the ECB to be reset.
- When DELFPT or termination occurs, the ECB type will again be used as an index into the byte-table EMABNX which will provide an index into the word-table EMABNB. The word thus located contains a branch to the logic to handle abnormal conditions for the ECB type.

Values for the ECB type are

| | | | |
|---|---|---|---|
| 1 | I/O service calls | 5 | INIT |
| 2 | SIGNAL | 6 | ENQ |
| 3 | STIMER | 7 | Memory Management activities |
| 4 | POLL | 8 | STDLB for an exclusive device |

## Dynamic Space (TSPACE)

Such routines as error logging and monitor crash analysis as well as the reentrant overlays require temporary space, which they may obtain, hold for a period of time,and then release.

The space is managed by use of an algorithm that requires space to be parcelled out in powers of two (2, 4, 8, 16, 32, 64, 128, 256) only. Thus if a routine asks for 19 words it will be given 32. The reason for chosing this method is its minimal processing time for obtaining and releasing space.

The algorithm is as follows:

1. When obtaining space, if the smallest power of two needed is not available the next higher power of two will be examined. If space is available at that level the block is split into two blocks of the size needed. This is a recursive technique which may be repeated until the maximum power (8) is reached.

2. When releasing space, an attempt is made to find the released block's complement (the other half of the original split block) and if found they are joined and the procedure repeated for the next higher power of 2 until 8 is reached.

### Dynamic-Space Service Calls

**GETTEMP** Get Space

Inputs:

R7 = number of words (1 through 255)
R8 = link

Output if space available:

R7 = byte 1/number of words
byte 2, 3, 4/address of space
R8 = link
Return to link + 1.

Output if no space:

R7 = number of words
R8 = link
R15 = X'66' (no-space TYC)
Return to link.

**RELTEMP** Release Space

Input:

R7 = byte 1/number of words
byte 2, 3, 4/address of space
R8 = link

Output:

R7 = number of words
R8 = link
Return to link.

### SYSGEN Considerations

The number of words needed may be specified at SYSGEN by use of the TSPACE option on the :RESERVE card:

:RESERVE (option), (TSPACE, n), . . .

where n is number of words for temporary space (a default is provided by SYSGEN).

## Dispatcher

Each dispatcher in CP-R possesses a queue whose head may be found in RDLISTI. (RDLI is a parallel table with one set of entries per dispatcher.) The queue pointers chain secondary STI entries (through STIDNXT) for the dispatcher in order of priority.

To enter a dispatcher level, the higher of the two interrupt levels associated with the dispatcher is triggered. Upon being entered, the dispatcher searches its queue from the head down for the highest priority task that is ready to run.

A task is ready to run when

- It is not waiting (STICOUNT = 0).
- It is not suspended.
- It is not stopped.
- It is not rolled out.

If such a task is not located, the next lower dispatcher level is triggered with the final dispatcher waiting in an idle loop.

If a task is found, the lower dispatcher level is entered. At the lower level, the map for the secondary task is loaded and control is given to RBMEXIT. This causes control to be given to the secondary task, or to the Overlay Manager if an overlay reload is necessary.

It should be noted that the lowest dispatcher level requires only one interrupt level since the null level is used as its second level.

See the Terminal Job Entry chapter for description of time-slicing and swapping.

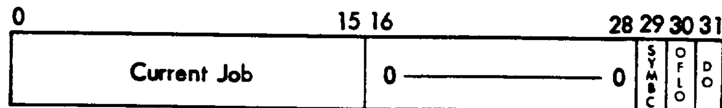
## Symbionts

The monitor cells shown below contain information about the symbionts that is not related to a particular device. Device related information may be found in the DCTRBM, DCTSYM1, DCTSYM2 and DCTSYM3 tables.

SYMB

SYMB is a word in the resident portion of the symbiont task that contains general information about the symbionts.

The format of SYMB is




where

Current Job is a number that will be used in naming the next job file in the IS area. The number is maintained and used by the M:JOB service call.

SYMBC = 0 means do not start background automatically; SYMBC = 1 means start background when the first file for a new job has been closed by the input symbiont.

This indicator is set by the "C S" keyin and reset by the "C O" keyin.

OFLO is an indicator set by the output cooperative when the OS disk area is full and background can no longer execute. The output symbiont automatically switches to DO mode when this indicator is set.

DO=0 the symbiont task will delete a job's files in the OS area when all of the files associated with this job have been output. This is the default mode.

DO=1 the symbiont task will delete a job's files in the OS area as they are output. This mode prevents overflow of the OS area by a job which has a large amount of output. If this mode is not in effect and a single job overflows the OS area, a switch to DO mode will automatically occur. This mode does not allow backspacing of files in the OS area since prior data records may have been in files that were deleted.

The DO bit is set by the DO key-in and reset by the RDO key-in.

JOBPRI

JOBPRI is a word in the resident portion of the monitor that contains the priority of the running task. This cell is used only in a symbiont system and is maintained by the input cooperative.

JOB#

JOB# is a word in the root of the monitor that contains the number of the running background job.

In a symbiont system, the value is used in the naming of files in the OS area and is maintained by the Job Control Processor.

# 8. MISCELLANEOUS SERVICES

Miscellaneous services are functions available to both foreground and background programs but which do not directly involve I/O services.

## SEGLOAD

This function loads explicitly requested overlay segments of a program into memory for execution. The user's M:SL DCB (allocated by the Overlay Loader) is used to perform the input operation.

For an FPT for READWRIT, the system uses the entry in the program OVLOAD table that corresponds to the segment. The OVLOAD table is constructed by the Overlay Loader.

The function locates the proper entry in the OVLOAD table and places the user-provided error address in both the OVLOAD entry (FPT) and in the M:SL DCB. If end-action was requested, the FPT is set to cause end-action at conclusion of the segment input.

If the calling program has requested that the segment be entered (at its entry point), the PSD at the top of the user Temp Stack is altered so that upon CALEXIT, control goes to the segment entry address.

The function then sets R3 to point at the FPT in the OVLOAD table and transfers to READWRIT. The segment input is then treated as a READ request with possible end-action, and at the user's option, control is returned either following the SEGLOAD CAL1, or to the segment entry address.

## Trap Handling

### Trap CAL and JTrap CAL

The Trap function sets up the trap control field and TRAPADD field in a user's PCB and sets the Decimal Mask (DM) and Arithmetic Mask (AM) bits in the user PSD to mask out occurrences of these traps. PSD bits are modified by changing them in the user PSD at the top of the Temp Stack and in the PSD contained in the user's TCB.

The JTRAP function has the same effect on the DM and AM bits, but stores the trap controls and trap address in the Job Control Block.

If the user-provided trap address is invalid or if the user specifies that he is to receive occurrences of some trap and no trap address is provided, control is transferred to TRAPX. This results in the message

```
ERR xx ON CAL    @yyyyy    ID = task name
```

where

xx is the Error Code in hexadecimal (00 if none).

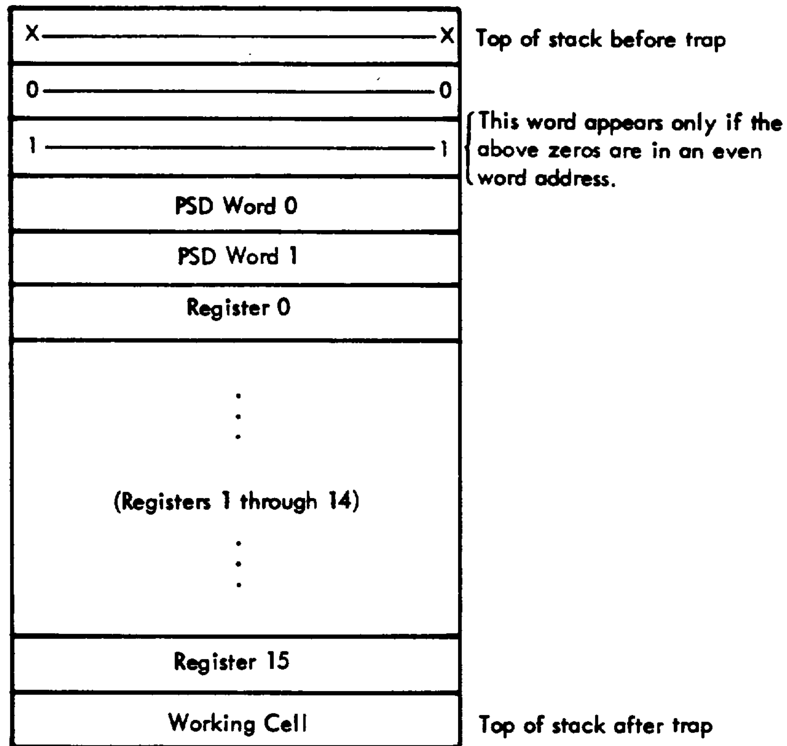
yyyyy is the address of the CAL.

being output on OC and LL.

### Trap Processing

Traps are either handled by the user, cause simulation of the instruction where possible, or result in an abort condition. If the user is to handle traps, task-level trap handling takes precedence over job-level trap handling.

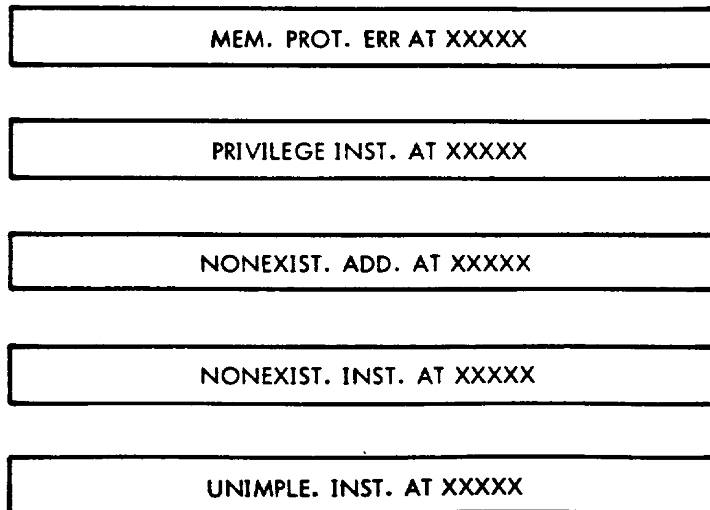
The registers and PSD are saved in the user Temp Stack in the following format:




If the trap is either a nonexistent instruction or unimplemented instruction, the instruction causing the trap is analyzed to determine whether the proper simulation package (if any) is in the system. If so, the simulation is called; if not, it is treated like any other trap.

A test is performed to determine whether the user is to process this particular trap. If so, the trap address (X'40', X'41', etc.) is placed in the top word of the stack and the user's trap handling routine is entered by LPSD, eight of the user PSD, with the trap handler substituted for the address where the trap occurred.

Traps not handled by instruction simulation or by the user result in one of the following messages being output to OC and LL:



| STACK OVERFLOW AT XXXXX |
|---|

| ARITH. FAULT AT XXXXX |
|---|

| WDOG TIMER RNOUT AT XXXXX |
|---|

| MEM. PARITY ERR AT XXXXX |
|---|

| BREAK ERROR AT XXXXX |
|---|

| ERRxx ON CAL @yyyyy ID = task name |
|---|

Note that the last message results from the simulation of a trap (called Trap X'50'). This is done by the system when a system call cannot be processed because of incorrect parameters being input or an error having occurred in the processing of a system call with no error address provided in the caller's FPT. After the message is output, the task will be aborted unless the user has provided a handler for this trap. If the user has provided a handler for this trap, the message will not be output and the trap handler will be entered.

### TRTN (Trap Return)

This function returns control following the instruction which caused a trap and is employed by the user to return control after processing a trap.

At the time of the TRTN call, the user Temp Stack is set as described previously under "Trap Processing". The TRTN function strips the stack of the context placed there by the CAL processing (from the TRTN CAL). It then clears the stack by the Trap processor and returns control to the instruction that follows the one causing the trap.

### TRTY (Trap Retry)

This function is similar to TRTN, but returns to the instruction causing the trap.

### TEXIT (Trap Exit)

This function removes the trap information from the user Temp Stack and exits the trapped task. Note that an EXIT CAL if executed from a user trap handler would leave this data in the user Temp Stack.

# 9. CP-R TABLE FORMATS

## General System Tables

The tables shown in the subsection are either not job or task controlled, or relate equally to both jobs and tasks. The index 0 entries of the tables are not used as true entries.

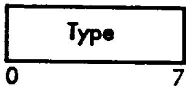
### 550 Processor Configuration Tables

These are parallel tables that contain data pertaining to Processor Polling.

**CNFGADDR**

This contains the address of the processor.

**CNFGTYPE**




This contains the type code for the processor,

where

1 = CPU.

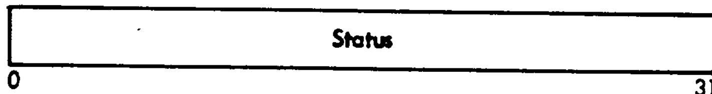
2 = Memory Interface (MI).

3 = Processor Interface (PI).

4 = MIOP.

7 = System Unit (SU).

**CNFGSTAT**




This is used for temporary storage of processor status and condition codes during the logging process.

CNFGADDR and CNFGTYPE are initialized by SYSGEN based on :PROC cards.

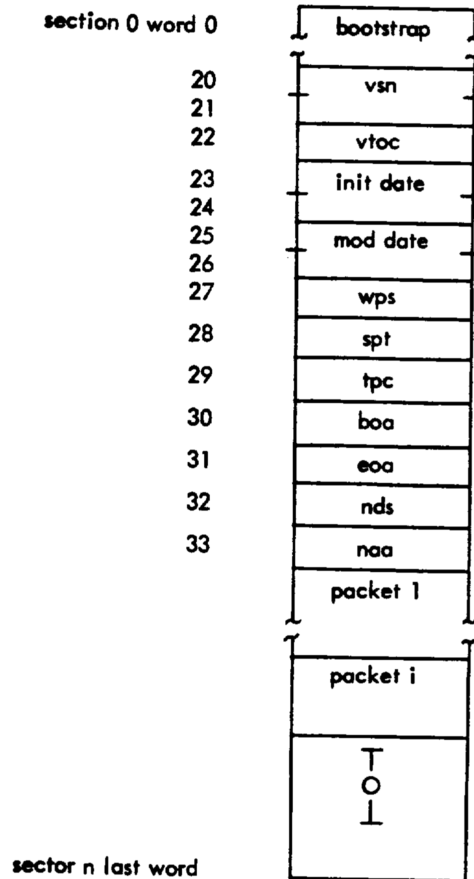
These tables are used to control Processor Polling and are primarily used to provide information for the error log.

### Volume Table of Contents (VTOC)

Information describing the allocation of areas on a private disk pack is maintained in this data structure, which begins in sector 0 of the device. Its length depends on the number of areas defined, and is specified in the structure. The VTOC may be extended by RADETIT : ADD commands after the original initialization. Such extension may proceed to the end of the last VTOC sector. Additional VTOC sectors can be obtained from the first area on the pack only if it is a skipped area.

Devices with Sysgened areas are not private devices. They have no VTOC; and their structural information permanently resides in the Master Dictionary.

The Format of the VTOC follows:




where

| | |
|---|---|
| bootstrap | is a default boostrap program that will type a message indicating this is not a system device and then go into an idle state. |
| VTOC | is the character string 'VTOC' that identifies the disk as have been initialized and allocated by the :INIT command. |
| vsn | is the 8 EBCDIC character identification of the disk. It may be any 1 to 8 character string composed of letters and/or digits, left justified, space filled. |
| init date | is the date on which the disk pack was originally initialized, in the form mmmdd yy. |
| mod date | is the date on which the allocation of the pack was most recently modified, in the form mmmdd yy. |

wps — is the words per sector of the disk.

spt — is the number of sectors per track of the disk.

tpc — is the number of tracks per cylinder of the disk.

boa — is the sector number of the first sector on the disk available for data. It normally is initialized to 1 (see nds below).

eoa — is the sector number of the last sector on the disk.

nds — is the number of directory sectors required to hold the directory. Normally this will be a 1, meaning all area allocation information is contained in this, the first sector on the disk, and data area may begin on the next sector (sector 1). However, if a large number of areas and skipped areas are specified, there may be insufficient space in one sector for the directory. For this case, the directory will be continued on subsequent sectors, nds set to indicate this number, and boa set to begin on the next sector.

naa — is the number of allocated areas. It is the total number of areas currently active on the disk. It does not include any "SKIP" ped areas or unallocated space at the end of the disk.

packet 1 ⋮ packet i — is a group of data describing a contiguous block of sectors of like use. Such use may be one of three types: allocated for a file area, skipped, or unallocated remainder. There is always one packet describing the unallocated remainder, even if its size is zero. There is one packet in addition for each file area and each explicitly skipped area on the pack. Packets are grouped by type, first file area packets, then skipped area packets, then the unallocated remainder packet. Within groups, packets are ordered by ascending start address. The format of a packet follows.

| | byte 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| word 0 | name | | wp | |
| 1 | 0 | | it | |
| 2 | ssec | | | |
| 3 | esec | | | |

where

name — is a two-character EBCDIC area name for a file area, x'FFFF' for a skipped area, or numeric 0 for the unallocated remainder.

wp — is the write protection:

numeric 0 for public;
1 for background;
2 for foreground;
3 for system;
4 for IOEX.

it — is the initialization type code:

numeric 0 for OVR;
1 for FAST;
2 for ALL.

## RAD File Table (RFT)

Parameters describing the file are taken from the directory entry for the file. These parameters include:

File name, area index, and account name

Beginning sector address (relative to beginning of the area)

Ending sector address (relative to beginning of the area)

Granule size

Record size

File size (number of records)

Organization (blocked, unblocked, compressed)

The parameters specifying the physical characteristics of the disk, the boundaries of the disk area, and the Write Protection key are in the Master Dictionary. To enable access to these, the RFT contains a Master Dictionary Index (specifying the area).

For manipulation of the file, the RFT contains the following items:

Blocking buffer control word address

Blocking buffer position

Position within the file (sector last accessed – used for blocked and unblocked)

Current record number

Number of DCBs open to the file.

These parameters are entered in the RFT by the OPEN function. The parallel table concept is used for the RFT, and the tables are allocated and initialized as given in Table 2.

In Table 2:

| | |
|---|---|
| File name all 0 | Signifies entry not in use. |
| RFT4 index 0 | Entry contains the total number of RFT entries. |
| RFT13 index 0 | Entry contains the maximum number of RFT entries allowed for background use. |
| RFT14 index 0 | Entry contains the current number of background file entries. |
| RFT15 index 0 | Entry contains the number of temp files allocated. |
| Other index 0 | Entries are not used. |

The Job Control Processor builds the RFT entries for the Background Temp Files. These entries are the first n + 2 in the table (n is the number of Xi files), where entry 1 is for the OV file, entry 2 is for the GO file, entry 3 is for the X1 file, etc.

## FILE DIRECTORY

The information concerning allocated files is contained in a file directory entry. The file directory entries for files in each area are kept in directory sectors within that area.

The first sector (sector zero) of an area containing files is always the first directory sector. There may be more than one directory sector if there are more files than can be held in the first directory sector. The subsequent sectors are linked together starting from the first sector.

The format of all directory sectors is identical. It is

| Index | | Word # |
|---|---|---|
| 0 | FLAG \| Index | 1 |
| 1 | Sector | 2 |
| 2 | File directory ID code word 1 | 3 |
| 3 | File directory ID code word 2 | 4 |
| 4 | File directory entry 1 | 5 |
| | File directory entry 2 | |
| | ⋮ | |
| | File directory entry n | |
| 255 | Unused directory entries or unusable space at end of sector | |

where

flag  is end of directory indicator.

- 0  means this is the last directory sector.
- 1  means directory continued on another sector.

index  is the index to the next unused word in the sector.

sector  depends on the value of FLAG.

- FLAG=0  means this is the next free, unassigned sector in the area.
- FLAG=1  means this is the sector number of the next directory sector.

**file directory code words** are two words containing identifying codes used to verify that the sector is actually a directory sector. These words are:

X'AAAAAAAA'

X'55555555'

There are two possible formats of the first directory sector which both mean the area contains no files. The first four words may be all zeroes, which is its condition if it has been cleared by SYSGEN (FAST or ALL options), or it has been cleared by the :CLEAR command in RADEDIT.

The first four words may also be

| Index | | | Word # |
|---|---|---|---|
| 0 | 0 | 4 | 1 |
| 1 | 1 | | 2 |
| 2 | Code word 1 | | 3 |
| 3 | Code word 2 | | 4 |

The all zero format will be converted to the normal format when the first file is allocated in the area.

### Permanent File Directory Entry

A file directory defines and describes a file. It contains the information needed by the system to access and use the file. There is one entry for each file or file extension in an area. The fixed portion of an entry is identical in format for all files. It contains the file's name, size, organization and position in the area.

If the File Account option is not used, entries are 9 words long; if the option is used, the entries are 11 words long. An entry will not cross a directory sector boundary.

The format of an entry is:

| Index | | | | | Word # |
|---|---|---|---|---|---|
| 0 | F | I | L | E | 1 |
| 1 | N | A | M | E | 2 |
| 2 | FLAG1 | FLAG2 | RESERVED | LEN | 3 |
| 3 | GSIZE | | RSIZE | | 4 |
| 4 | FSIZE | | | | 5 |
| 5 | BOT | | | | 6 |
| 6 | EOT | | | | 7 |
| 7 | XTNT | | | | 8 |
| 8 | ESIZE | | | | 9 |
| 9 | A | C | C | O | 10 |
| 10 | U | N | T | # | 11 |

| FLAG1 = | S | D | R | $F_{IX}$ | R | R | ORG |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 7 |

| FLAG2 = | PRIORITY | 0 | 0 | 0 | RF |
|---|---|---|---|---|---|

where

FILENAME is the 8 character EBCDIC name, left justified and space filled, of an active file. A name of all binary zeroes indicates a deleted file, and a name of all binary ones indicates a bad sectors entry (space that is not to be allocated to an active file).

FLAG1

R reserved.

S file was last written sequentially. } Maintained by the monitor; initially set to 0.

D file was last written directly. }

FIX if the flag is set, extended files will not be combined into one large file during a RADEDIT :SQUEEZE operation.

ORG file organization (same as in ALLOT CAL).

00 = unblocked

01 = blocked

10 = compressed

FLAG2

PRIORITY for files in IS and OS areas, the job's priority; for other files, zeroes.

RF RF = 1 means resident foreground program if in FP area.

RF = 0 for all other files and areas.

LEN number of words in this directory entry.

GSIZE the granule size, in bytes; used for direct access.

RSIZE the number of bytes per logical record for UNBLOCKED and BLOCKED files.

FSIZE the number of records in the file if the file is not extended; the number of records in this extension if it is extended.

BOT }
EOT } the area relative first and last sector of the file or extension.

XTNT the extent number indicating the position of this extent in the file. This word is zero if this is the first or only extent of a file.

ESIZE the number of sectors to allocate to the next extension if file extension occurs. This word is zero if the file is not to be extended.

ACCOUNT# the 8 character EBCDIC name, left justified and space filled, under which this file was allocated.

| Address | Contents | Initial Value | Length |
|---|---|---|---|
| RFT1 | File Name | 0 | Doubleword |
| RFT2 | Beginning Sector Address (Relative to area) | X | Word |
| RFT3 | Ending Sector Address (Relative to area) | X | Word |
| RFT4 | Granule size (in bytes) | X | Halfword |
| RFT5 | Record size (in bytes) | X | Halfword |
| RFT6 | File Size (in records for sequential access files; in granules for direct access files) | X | Word |
| RFT7 | Switches<br>where<br>Bit 0 = 1 means sequentially written<br>Bit 1 = 1 means directly written<br>Bit 3 = 1 means extents are fixed in size<br>Bit 6 = 1 means compressed<br>Bit 7 = 1 means blocked | X | Byte |
| RFT8 | Master Dictionary Index | X | Byte |
| RFT9 | Job Identification | X | Byte |
| RFT10 | Blocking Buffer Position (in bytes) | X | Halfword |
| RFT11 | File Position (in sectors) | X | Word |
| RFT12 | Current Record Number | X | Word |
| RFT13 | Number of Open DCBs (total) | X | Byte |
| RFT14 | If RFT17 is nonzero, this entry identifies the job that obtained the blocking buffer | X | Byte |
| RFT15 | Number of OPEN background DCBs | X | Byte |
| RFT16 | Status (bit 0 on for sequential write, bit 1 on for direct access write) | X | Byte |
| RFT17 | Blocking Buffer Control Word Address | X | Word |
| RFTE# | Number of current extent (extensible files only) | 0 | Halfword |
| RFTESZ | Number of sectors per extent. Derived from extent size given when file was allotted. | 0 | Word |
| RFTACNT | EBCDIC disk file account name | X | Doubleword |

The RFT is also used to maintain data controlling access to a tape drive open in blocked mode. The data in an RFT entry used for blocked tape is described in Table 2A. It is deliberately similar to that maintained for a blocked file, so that common processing is more frequently possible.

Table 3. RAD File Table Allocation for a Block Tape

| Address | Contontrol |
|---|---|
| RFT1 | The device name as in DCT16:<br>' NL yyndd' |
| RFT2 | (not used) |
| RFT3 | (not used) |
| RFT4 | Size in bytes of current block on input |
| RFT5 | Logical record size in bytes |
| RFT6 | (not used) |
| RFT7 | Switches, where<br>Bits 0,1 = 10 always (indicates sequential use)<br>Bit 6 = 1 means compressed<br>Bit 7 = 1 means blocked |
| RFT8 | DCT index for drive |
| RFT9 | Job ID (maintained but not used) |
| RFT10 | Blocking buffer position in bytes |
| RFT11 | (not used) |
| RFT12 | Current record number (maintained but not used) |
| RFT13 | Number of open DCBs (total) |
| RFT14 | Job ID of job which obtained the current blocking buffer |
| RFT15 | Number of open background DCBs (maintained but not used) |
| RFT16 | Bit 0 = 1 indicates the tape has been written since it was opened. |
| RFT17 | Blocking buffer control word address |
| RFT# | (not used) |
| RFTSZ | (not used) |
| RFTACNT | (not used) |

## Device Control Table (DCT)

### DCT Format

The Device Control Table (DCT) is composed of several parallel subtables (see Table 4). The various entries associated with a given device are accessed using the DCT index of the device and addressing the tables DCT1 through DCT19. For example DCT1 would be accessed by

```
LHR,R    DCT1,X
```

DCT2 would be accessed by

```
LB,R     DCT2,X
```

where Register X contains the DCT index value for the device.

Table 4. DCT Subtable Formats

| Subtable Address | Contents | Length |
|---|---|---|
| DCT1 | Active I/O address for device | Halfword |
| DCT1P | Primary (P) device address | Halfword |
| DCT1A | Alternate (A) device address | Halfword |
| | (DCT1, DCT1P, DCT1A format: bits 0–2 = 0—0; bits 3–7 = IOP; bit 8 = 0; bits 9–15 = Device) | |
| DCT2 | Channel Information Table Index - A pointer to the CIT entry for the channel associated with the device. | Byte |
| DCT3 | Bit 0 = 1 means output is legal for this device.<br>Bit 1 = 1 means input is legal for this device.<br>Bit 2 = 1 means device has been marked down and is inoperative.<br>Bit 3 = 1 means device timed out.<br>Bit 4 = 1 means SIO has failed.<br>Bit 5 = 1 means the I/O has aborted.<br>Bits 6/7 = 00 - "Busy" both subchannels.<br>= 01 - Use the P subchannel only.<br>= 10 - Use the A subchannel only.<br>= 11 - Use either subchannel. | Byte |
| DCT4 | Device Type<br>0 = NO (IOEX)<br>1 = TY<br>2 = PR<br>3 = PP<br>4 = CR<br>5 = CP<br>6 = LP<br>7 = DC<br>8 = 9T<br>9 = 7T<br>10 = CP (Low Cost)<br>11 = LP (Low Cost)<br>12 = DP (7242/46, 7270)<br>13 = PL<br>14 = DP<br>15 = Reserved<br>16 = 9T (550)<br>17 = Reserved<br>18 = Special user devices<br>19 = LD | Byte |
| DCT5 | Status Switches<br>Bit 0 = device busy.<br>Bit 1 = waiting for cleanup.<br>Bit 2 = between inseparable operations.<br>Bit 3 = data being transferred. | Byte |

Table 4. DCT Subtable Formats (cont.)

| Subtable Address | Contents | Length |
|---|---|---|
| DCT5 (cont.) | Bit 4 = error message given (key-in pending).<br>Bit 5 = deferred SIO pending<br>Bit 6 = SIO was given while device was in manual mode.<br>Bit 7 = Unqueued IOEX on this device. | |
| DCT6 | Pointer to queue entry representing current request. | Byte |
| DCT7 | Command list doubleword address. | Halfword |
| DCT8 | Handler start address. | Word |
| DCT9 | Handler cleanup address. | Word |
| DCT10 | Device activity count (used for I/O Service reentrance testing). | Word |
| DCT11 | Timeout value (used to abort request when no interrupt occurs). | Word |
| DCT12 | AIO status (or end action control word for unqueued IOEX). | Word |
| DCT13 | TDV status. | Doubleword |
| DCT14 | STOPIO (background only) count. | Byte |
| DCT15 | STOPIO (all system I/O) count. | Byte |
| DCT16 | The five-character device name (e.g., CRA03) preceded by the three characters "(NL)!!". | Doubleword |
| DCT17 | Retry function code (for error recovery) and continuation code. | Halfword |
| DCT19 | AIO condition codes. | Byte |
| DCT20 | TDV condition codes. | Byte |
| DCT20A | TIO condition codes. | Byte |
| DCT21 | TIO status. | Halfword |
| DCTSDBUF | Side-buffer address. | Word |
| DCTMOD | Device model number, EBCDIC. | Word |
| DCTMODX | Device model number, decimal. | Halfword |
| DCT#ERR | Number of I/O errors. | Word |
| DCT#IO | Number of I/O starts. | Word |
| DCTJID | Job ID for reserved devices. | Byte |

Table 4. DCT Subtable Formats (cont.)

| Subtable Address | Contents | Length |
|---|---|---|
| DCTTJE | TJE flags (see TJE Chapter) | Byte |
| DCTSYM1 | Bit 0 = 1 means the device is on-line. Symbiont activity is to start when possible, is active or is in a suspended state. The bit is set by the Syyndd,I key-in or the output cooperative. The bit is reset by the symbiont task when a !FIN card is read. The bit may also by reset by a !JOB card if the Syyndd,L or Syyndd,T key-in is in effect.<br><br>Bit 1 = 1 means the device is locked out. Symbiont I/O will cease when the current job has completed. If exclusive use of the device has been obtained by the symbiont task, the device will be released.<br><br>The Syyndd,L key-in sets this bit. The Syyndd,I key-in will reset the bit and start symbiont I/O on the device.<br><br>Bit 2 = 1 means the same as bit 1 except that the device is removed from use by the symbionts when the current job has completed if it is not dedicated to symbionts.<br><br>The "T" bit is set by the Syyndd,T key-in. The Syyndd,I key-in will reset the bit, acquire the device (if necessary), and start symbiont I/O on the device.<br><br>Bit 3 = 1 means symbiont activity has been suspended on this device. The bit is set by the Syyndd,S key-in. The C,B or R options of the Syyndd key-in will reset the bit and restart symbiont I/O on the device.<br><br>Bit 4 = 1 means the device was dedicated to symbionts at SYSGEN. This bit is set by SYSGEN and is never reset.<br><br>Bit 5 = 1 means the device is in use by symbionts. If the device is dedicated to symbionts, the bit is set by SYSGEN and will not be reset. Otherwise the bit is set by the Syyndd,I key-in and reset by the symbiont task if the "T" bit is set when the current symbiont file has completed.<br><br>Bit 6 = 1 means the current job is to be released. Symbiont activity for the specified device will be terminated and associated symbiont files will be deleted.<br><br>Bit 7 = 1 means save the current output file and terminate. What remains of the file is returned to the output queue and the symbiont is locked immediately. The entire file is saved if the symbiont device is not in DO mode. | Byte |
| DCTSYM2 | The address of a TSPACE block which contains the address of the context block for a symbiont device.<br>This entry is zero if the device is not in use by symbionts. | Word |
| DCTSYM3 | Bit 7 = 1 means the operator has requested that an output symbiont file be rewound (R) or backspaced (B). | Byte |
| DCTRBM | Bit 0 = 1 means the device is being requested by the SYMBIONT task.<br>Bit 1 = 1 means the device is in use by the MEDIA task. When the device is no longer needed by MEDIA, this bit will be reset if bit 0 is set. This will allow the symbiont task to obtain the device. | Byte |

Table 4. DCT Subtable Formats (cont.)

| Subtable Address | Contents | Length |
|---|---|---|
| DCTRBM | Bit 2 = 1 means the device is in use by the symbiont task. When the device is no longer needed by SYMBIONTS, this bit will be reset if bit 3 is set. This will allow the MEDIA task to obtain the device.<br>Bit 3 = 1 means the device is being requested by the MEDIA task.<br>Bit 6 = 1 means DED DPndd,R key-in is in effect. | Byte |
| DCTCD | The DCT index of a device which cannot be operated concurrently. Used only for 3243 devices that share arm position mechanism. | Byte |
| DCTDISCI | The disk type index. Used only for disk devices. This points to disk characteristics in the DISC tables. | Byte |
| DCTDCB | Number of DCBs OPEN to this device. | Byte |
| DCTRFT | Nonzero only for a tape drive open in blocked mode. If DCTRFT = x '80' an RFT entry is not currently assigned. If DCTRF + x '80', it is the index of the RFT entry containing the blocking controls. | Byte |

SYSGEN DCT Consideration

System Generation allocates the space for the DCT subtables. Initial values are defined for the following entries (all other entries are initially zero):

| | |
|---|---|
| DCT1 | As specified by :DEVICE command |
| DCT1P | As specified by :DEVICE and :CHAN commands. |
| DCT1A | As specified by :DEVICE and :CHAN commands. |
| DCT2 | As specified by :DEVICE and :CHAN commands. |
| DCT3 | As specified by :DEVICE command. |
| DCT4 | As specified by :DEVICE command. |
| DCT7 | Pointer to SYSGEN allocated space for command list. |
| DCT14 | 1 if (DEDICATE, F); otherwise, zero. |
| DCT15 | 1 if (DEDICATE, X); otherwise, zero. |
| DCT16 | "Ⓝ I lyyndd" where yyndd comes from the :DEVICE command. |
| DCTDEBUG | External interrupt location minum X'4F'. (Used for BREAK logic.) |
| DCTSDBUG | Pointer to side buffer. |
| DCTMOD | EBCDIC model number. |
| DCTMODX | Decimal model number. |
| DCTJID | X'FF' if reserved device; otherwise, zero. |
| DCTCD | Initialized for adjacent 3243 devices. |

The index 0 entry of each subtable is not used as a true table entry because of the nature of the BDR instruction.

DCT1, index 0, contains the number of non-TJE devices.

DCT7, index 0, contains the total number of devices including TJE devices.

DCT7 contains the DW address of space allocated by SYSGEN for the command list for each device. These areas are on a doubleword boundary.

## DISC Tables

The DISC tables are a series of parallel subtables with an entry for each different disk type. They are built by SYSGEN based on :DEVICE commands. The index value used with these tables is obtained from DCTDISCI.

| Address | Usage | Size |
|---|---|---|
| DISCNSPT | Number of sectors per track | Byte |
| DISCNWPS | Number of words per sector | Halfword |
| DISCMAXS | Last relative sector number | Word |
| DISCMINS | First relative sector number | Word |
| DISCSSFT | Sector number shift to build seek address | Byte |
| DISCTSFT | Track number shift to build seek address | Byte |
| DISCCSFT | Cylinder number shift to build seek address | Byte |
| DISCNTPC | Number of tracks per cylinder | Halfword |
| DISCNCYL | Number of cylinders total | Halfword |

## Channel Information Table (CIT)

The Channel Information Table consists of parallel subtables, each with an entry per channel. There is one channel per controller connected to a MIOP, and one channel per SIOP. The "channel" concept is used since there cannot be more than one data transfer operation in process per channel. I/O device requests are queued on a per-channel basis. System Generation allocates these subtables as shown below:

| Address | Usage | Size |
|---|---|---|
| CIT1 | Queue head | Byte |
| CIT2 | Queue tail | Byte |
| CIT3 | Switches: | Byte |
| | Bit 0 - Subchannel P busy | |
| | Bit 1 - Subchannel A busy | |
| | Bit 2 - Subchannel P held | |
| | Bit 3 - Subchannel A held | |
| | Bit 4 - Dual-access channel | |
| | Bit 5 - Preferred channel (0 = P; 1 = A) | |
| CIT5 | Holding Request Q pointer for subchannel P | Byte |
| CIT6 | Holding Request Q pointer for subchannel A | Byte |

The CIT subtable entries are accessed by using

```
LB,R        CITN,X
```

where Register X contains the index (1-N).

The index 0 entry is not used because of the nature of the BDR instruction.

## I/O Queue Table (IOQ)

The I/O Queue Table consists of parallel subtables each with an entry per queue entry. These tables are accessed in the same manner as DCT and CIT by using an index. As is true for DCT and CIT, the index 0 entry of each subtable is not used as a true queue entry.

System Generation allocates and initializes the IOQ tables as given in Table 5.

Notice that IOQ2 index 0 is initialized by SYSGEN. This byte is used and maintained by the I/O system as the "free entry pool" pointer. By initializing IOQ2 as shown, SYSGEN links all entries into this pool.

IOQ1 index 0 is initialized by SYSGEN to the maximum number of queue entries allowed to the background.

IOQ3 index 0 is initialized to 0, since this byte is used and maintained by the I/O system as the current number of queue entries in use by background. IOQ4 (index 0) is the total number of IOQ entries.

Table 5. IOQ Allocation and Initialization

| Address | Contents | Initial Value | Length |
|---|---|---|---|
| IOQ1 | Backward Link | 0 | Byte |
| IOQ2 | Forward Link | Entry M contains M + 1 for N > M ≥ 0. Entry N contains 0. N is the number of queue entries. | Byte |
| IOQ3 | Switches<br>Bit 0 = 1 – request busy.<br>Bit = 5-7:<br>= 000 – Both subchannels required.<br>= 001 – Subchannel P only.<br>= 010 – Subchannel A only.<br>= 100 – Use either subchannel. | 0 | Byte |
| IOQ4 | Function Code (:DOT table index) | 0 | Byte |
| IOQ5 | Current Function Step | 0 | Byte |
| IOQ7 | Device Index | 0 | Byte |
| IOQ8 | Bits 0, 1 = 0 - byte address of buffer.<br>Bit 0 = 1 – DW address of a data chain.<br>Bit 1 = 1 – DW address of command chain (Queued IOEX). | 0 | Word |
| IOQ9 | IOQ8 bits 0, 1 = 0 - Byte count of buffer.<br>IOQ8 bit 0 = 1 – number of DWs in data chain.<br>IOQ8 bit 1 = 1 – timeout value for command chain. | 0 | Halfword |

Table 5. IOQ Allocation and Initialization (cont.)

| Address | Contents | Initial Value | Length |
|---|---|---|---|
| IOQ10 | Maximum retry Count | 0 | Byte |
| IOQ11 | Retry Count | 0 | Byte |
| IOQ12 | Seek Address | 0 | Word |
| IOQ13 | End-Action data<br><br>Word 1<br><br>Byte 0 is cleanup code where value:<br><br>1 = Post status in FPT.<br><br>2 = Post status in DCB.<br><br>3 = Transfer to address specified in bits 15 - 31.<br><br>4 = No end action (only available to the monitor).<br><br>Bit 8 = control device read.<br><br>Bit 9 = end action data in word 2.<br><br>Bits 15-31 = FPT completion-status word address for cleanup-code 1; DCB address for cleanup-code 2.<br><br>Word 2<br><br>If word 2 = 0, parameter not present.<br><br>If byte 0 = X'7F', bits 15-31 are user's signal address.<br><br>If byte 0 = X'FF', bits 15-31 are user's endaction address.<br><br>If word 2 ≠ 0, and byte 0 ≠ X'FF' or X'7F', byte 0 = end-action interrupt group code, byte 1 = interrupt address X'4F', bits 15-31 contain level bit for interrupt. | 0 | Doubleword |
| IOQ14 | Priority | 0 | Byte |
| IOQECB | ECB Pointer | 0 | Word |
| IOQERROR | Error-log buffer pointer | 0 | Word |

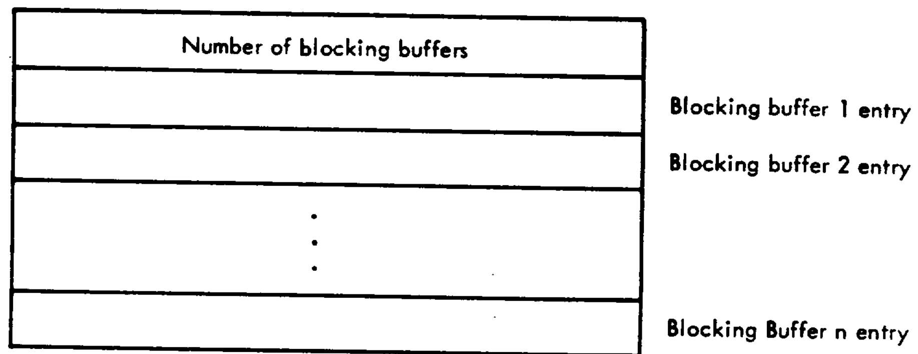
Since the 0th entry is never used in subtables whose entires are words or doublewords, it is not necessary to allocate space for this entry. If the 2N words for IOQ13 are allocated beginning at location ALPHA, IOQ13 is given value ALPHA-2. Thus, IOQ13 may actually point into another table but presents no problem because IOQ13 will never be accessed with index 0.

It should be noted that none of the subtables need be positioned in any particular relationship to each other. They may be allocated anywhere in core with the restriction that Doubleword Tabels being on doubleword boundaries.

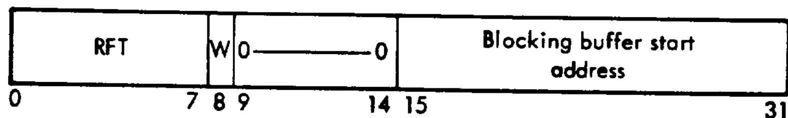
**Blocking Buffers**

Blocking buffers are 256-word buffers that are directly accessible only by the monitor. They are primarily used for blocked and compressed file I/O and for accessing file directories in OPEN/CLOSE service calls.

Each blocking buffer pool is controlled by means of a Blocking Buffer Control Word Table (BBCWT) that contains a one-word entry for each blocking buffer. The BBCWT has the format shown below.




Each entry is of the form



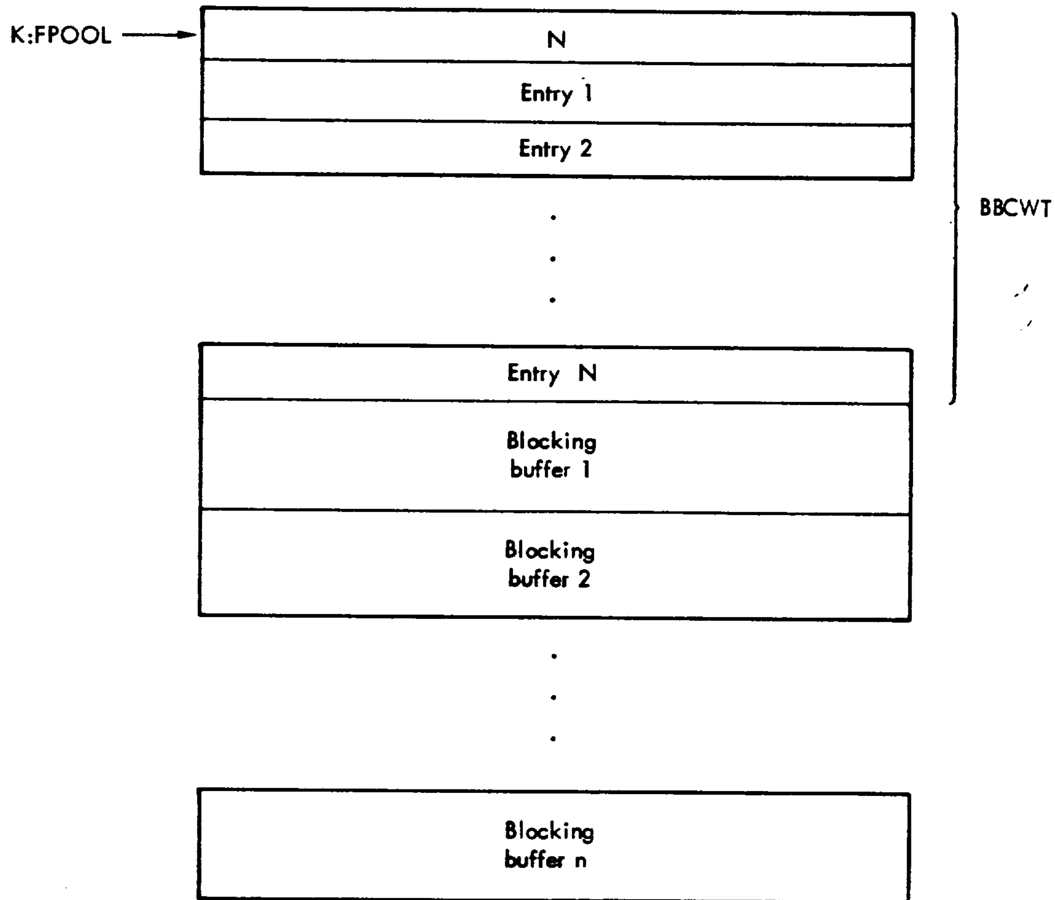

where

RFT is the index of the RFT entry for the file currently using this buffer. 0 signifies that the buffer is not in use. X'FF' means the buffer is in use, but not by any particular file.

W is set if the blocking buffer has been written in.

Primary and secondary tasks are kept in different blocking buffer pools and, therefore, have different BBCW tables. K:FPOOL contains a real address pointer to the BBCW table used by all primary tasks in the system. The number and

location of blocking buffers available to primary tasks is determined at SYSGEN by the FFPOOL parameter and cannot be changed except by SYSGEN. The primary-task blocking buffer structure is shown below:




In addition, a maximum of twelve pages will be made available to each job for blocking buffers from the job's Reserved Pages. Secondary tasks will be allocated blocking buffers from these pages as they are needed. The BBCW table is kept in the job's JCB and is constructed and maintained as blocking buffers are required and released.

### Master Dictionary

The Master Dictionary contains all the information needed to define an area for use by the system. It consists of six parallel subtables which are allocated and initialized by SYSGEN from information given by the :RESERVE and :DEVICE commands.

The entries for an area are accessed by the area index. This index corresponds to the position of the area's name in the MDNAME table. The tables are:

| Subtable Name | Contents | Length |
|---|---|---|
| MDNAME | The two EBCDIC character name of the area. | Halfword |
| MDFLAG | Control flags [A, (shaded), R, R, R, wp; bits 0 1 2 3 4 5 6 7]<br>where:<br>A 1 means the area is defined and allocated.<br>R Reserved.<br>wp The write protection assigned to the area.<br>0 P Public (no protection)<br>1 B Background<br>2 F Foreground<br>3 S System only without "SY" keyin<br>5 X IOEX | Byte |
| MDDCTI | The index to the DCT table for the device on which the area resides. | Byte |
| MDBOA | The start sector address of the area relative to sector zero of the disk. | Word |
| MDEOA | The end sector address of the area relative to sector zero of the disk. | Word |
| MDDISCI | The index to the DISC table for the device on which the area resides. | Byte |
| MDVSN | The Volume Serial Number of the dispack if the area is on a private pack. | Double-word |

The Master Dictionary is accessable to user programs through the following K: cells:

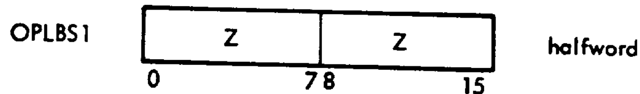
| Name | Location | Contents |
|---|---|---|
| K:MDNAME | X'212' | Address of MDNAME table. Byte 0 contains the number of entries allocated in the tables. |
| K:MASTD | X'14A' | Address of MDFLAG table. |
| K:MDBOA | X'218' | Address of MDBOA table. |
| K:MDEOA | X'219' | Address of MDEOA table. |
| K:MDDCTI | X'21A' | Address of MDDCTI table. |
| K:NUMDA | X'14B' | The highest valid index for the Dictionary. |

A sample of a Master Dictionary created by a SYSGEN in which the standard areas SP, FP, ..., D1, plus data areas D2 and D3 symbiont areas IS and OS, and three user defined areas were specified as:

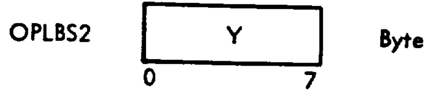
| Index | Name | MDFLAG | | Comments |
|---|---|---|---|---|
| 0 | SP | A | wp = 3 (System) | Fixed areas |
| 1 | FP | A | wp = 3 | |
| 2 | BP | A | wp = 3 | |
| 3 | BT | A | wp = 1 (Background) | |
| 4 | XA | A | wp = 5 (IOEX) | |
| 5 | CK | A | wp = 3 | |
| 6 | D1 | A | wp = 1 | |
| 7 | D2 | A | wp = 2 (Foreground) | |
| 8 | D3 | A | wp = 0 (Public) | |
| 9 | IS | A | wp = 1 | IS and OS symbiont areas |
| 10 | OS | A | wp = 1 | |
| 11 | U1 | A | wp = 0 | User defined area names specified during SYSGEN |
| 12 | U2 | A | wp = 0 | |
| 13 | U3 | A | wp = 1 | |

### Operational Label Table (OPLBS)

The Operational Label table is a parallel table with the format



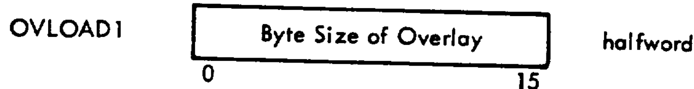

where ZZ is the operational label in EBCDIC

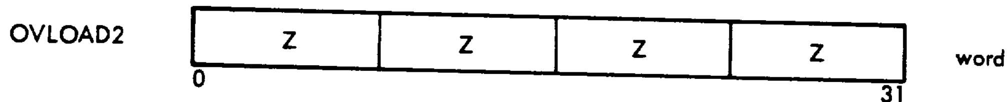



where Y is the DCT or RFT index of the permanent assignment (bit 0 = 0 if DCT index; bit 0 = 1 if RFT index). There is an OPLBS2 table for each active job, which is accessed by an address pointer in the associated job's JCB. The OPLBS2 table for the CP-R job contains the permanent assignment of each operational label. When a new job is activated, the OPLBS2 table it receives is a copy of the OPLBS2 table for the CP-R job at that time. The number of entries in OPLBS is in the first halfword of OPLBS1.

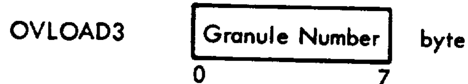
### OVLOAD Table (for CP-R Overlays Only)

The OVLOAD Table is a parallel table with the format




where ZZZZ = first four characters of name of overlay in EBCDIC

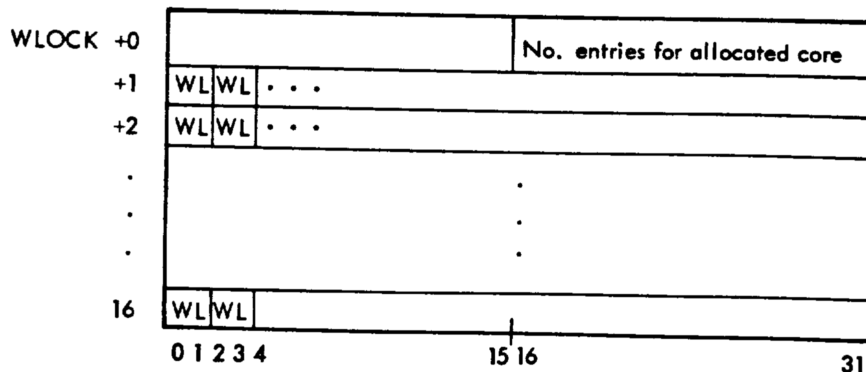



where the specified Granule Number is in the file CP-R.

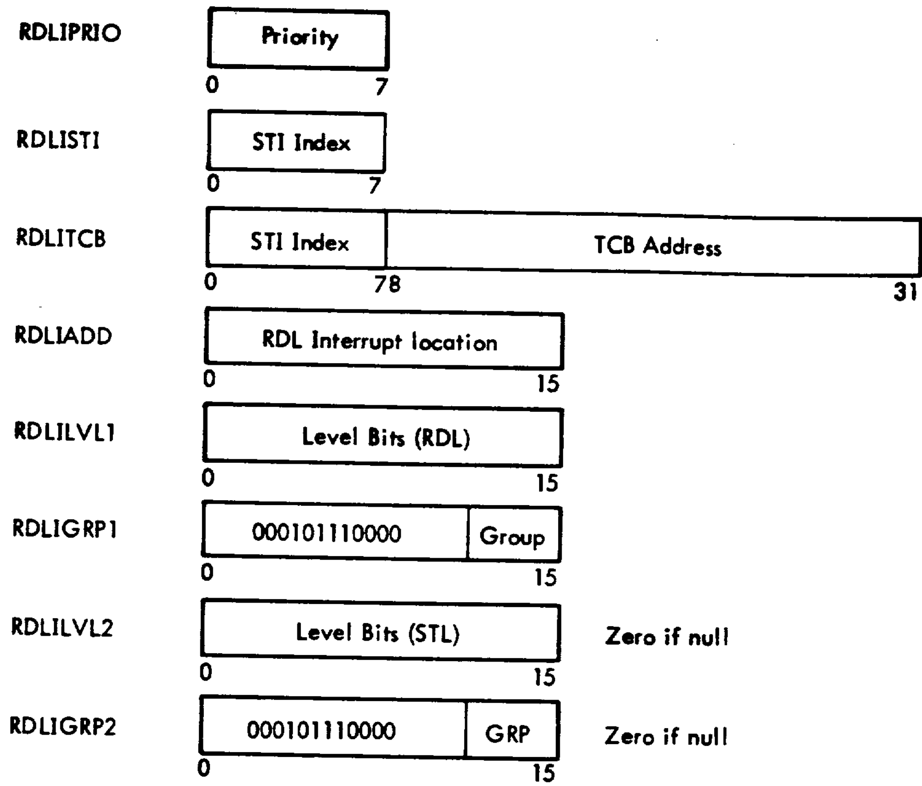
The number of entries in OVLOAD is in first halfword of OVLOAD1.

### Write Lock Table (WLOCK)

WLOCK contains write locks for the current core allocation. The table contains one entry for each real page of memory.



## CP-R Dispatcher Level Inventory (RDLI)




where

RDLIPRIO is the priority, in internal byte format, to which RDL is connected. This is the RDL interrupt location X'4F'. Entry 0 of RDLIPRIO is 0. Priority is set by SYSGEN and is not altered during execution.

RDLISTI is the task ID of the highest priority task operating within the level. Entry zero contains the overall STI head of the dispatcher chain. Each subsequent entry contains the subchain head that enters the dispatcher chain at the first task within the level. All entries are set to the first permanent CP-R task STI by SYSGEN.

RDLITCB is the STI index and TCB address for the dispatcher level.

RDLIADD is the core address of the RDL interrupt location in which to store the XPSD. It is set by SYSGEN and is not altered during execution. RDLIADD entry 0 contains the number of RDLI entries.

RDLILVL1 are the level bits for the RDL to be used on Write Direct commands.

RDLIGRP1 is the address field for a Write Direct interrupt control to trigger the RDL, including the trigger and group codes.

RDLILVL2, RDLIGRP2 are the level and group codes to trigger STL in the same format as RDLILVL1 and RDLIGRP1. All level and group codes are set by SYSGEN and are not altered during execution.

## Associative Enqueue Table (AET)

### Purpose

The AET provides a record of the enqueues done for controlled items by system services. It is used in conjunction with the Enqueue Definition table to control access to enqueued items.

### Type

Serial in the JCB or linked from the JCB depending on space requirements for job level ENQs. For task level ENQs linked from the LMI. Low-memory cell K:JAET contains the number of nonsharable devices in the Device Control table. This is used as the default number of AET entries allowed.

### Logical Access

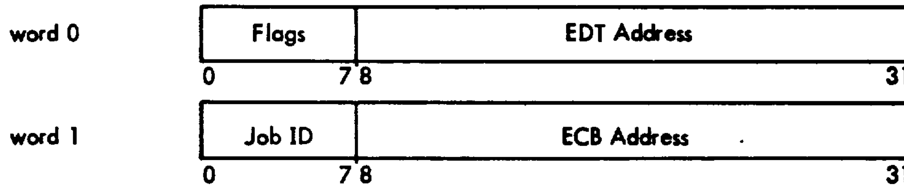
The AET is located via a pointer in a fixed position in the JCB or through a pointer in the LMI. Byte zero of the pointer word contains the number of words in AET.

### Overview of Usage

The job level AET table may be included in the JCB fixed portion or may be acquired separately from TSPACE and linked from the JCB depending on space requirements at the time the JCB is created. The task level AET is acquired from TSPACE at task initiation and is linked from the LMI. Byte zero of the pointer word contains the number of words in the AET and bytes 1-3 contain the real address of the start of the table.

At task or job termination, a flag in the JCB will indicate which usage applies and will release space appropriately.

### Associative Enqueue Table (AET) Format




where

| | | |
|---|---|---|
| Flags: | bit 0 = 1 | Job level AET |
| | = 0 | Task level AET |
| | bit 1 = 1 | System level EDT |
| | = 0 | Job level EDT |
| | bit 2 = 1 | ECB is for immediate enqueue |
| | = 0 | ECB is for an asynchronous enqueue |
| | bit 3 = 1 | Sharable enqueue |
| | = 0 | Exclusive enqueue |
| | bit 4 = 1 | Enqueue granted |
| | = 0 | Enqueue pending |
| | bit 5 = 1 | AET entry in use |
| | = 0 | AET entry free |
| | bit 6 = 1 | Dequeue CAL in progress |
| | bit 7 = 1 | Enqueue CAL in progress |

ECB Address    The location of the ECB created to wait for an ENQ. At check time, this address is set to zero. ENQ is set to 1 if the post is normal. The AET is freed if the post is not normal.

EDT Address    The location of the EDT of the controlled item which was enqueued.

Job ID    The identification of the job in which the item was enqueued.

## Real Memory Partition Table (RMPT)

### Purpose

The RMPT is used to describe and control all real memory resources. It contains one entry for each defined memory partition.

### Type

Serial consecutive doubleword entries in CP-R table space.

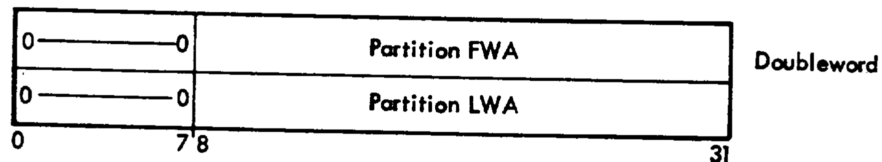
### Logical Access

The RMPT is pointed to by the system pointer K:RMPT. The RMPT starts on a doubleword boundary. The number of entries in the RMPT is contained in byte 0 of K:RMPT. The index 0 entry is not used as a table entry.

### Overview of Usage

The RMPT space is allocated, and partition entries are initially set by SYSGEN.

### Real Memory Partition Table (RMPT) Format




where Partition FWA and LWA define the real address limits of the partition.

## Partition Pointer Table (PPT)

### Purpose

Describes Partition attributes and points to Partition Control Tables.

### Type

Serial consecutive word entries in CP-R table space. Parallel to RMPT.

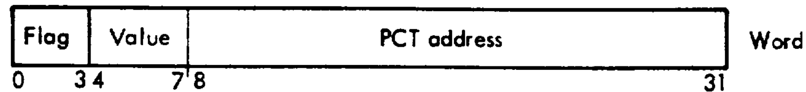
### Logical Access

The PPT is pointed to from K:PPT. Entries are located by index correspondence to an entry in the RMPT. Alternatively, a search can be made on the type field of the flag byte to identify a particular partition type. The number of table entries is contained in byte 0 of K:RMPT. The index 0 entry is not used as a true entry.

Overview of Usage

The PPT is accessed each time a request for memory in a preferred partition is made. Additionally this table in conjunction with the RMPT is used to verify that primary load modules will be loaded into Foreground Private Partitions.

Partition Pointer Table (PPT) Format




where

Flag contains control information as follows:

Bits

0 1 2 3

x x x 0 = perferred partition is no STM mode.
x x x 1 = preferred partition is STM mode.

Value indicates the type of partition according to values as follows:

0 – System (not allocatable)
1 – Private
2 – Foreground mailbox
3 – Foreground blocking buffers
4 – Preferred
5 – Secondary Task Memory
6 – 15 – Not used

PCT address is the address of the partition's control table. It has a 0 value if the partition does not have a PCT.

**Partition Control Table (PCT)**

Purpose

The PCT serves as a repository of information used in controlling allocation and access into Foreground Preferred and STM Memory Partition. One PCT exists for each of these defined memory partitions. The PCT also contains chain headers for controlling free pages in Secondary Task Memory (STM).

Type

Serial table located in CP-R table space.

Logical Access

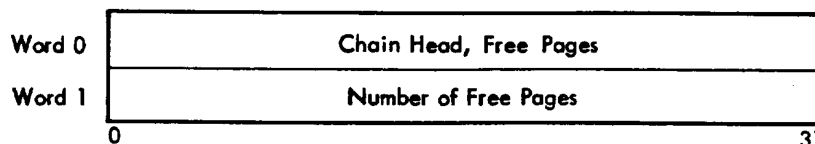
Each PCT table is pointed to from the Partition Pointer Table (PPT). Information in the PCT identifies the pages in the partition that have been allocated and those that are free, or points to chain headers that control STM.

Overview of Usage

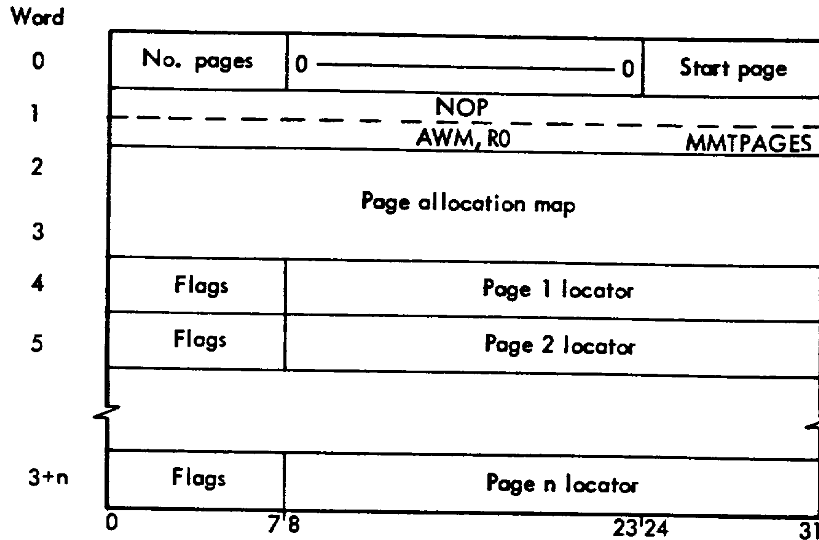
The PCT is allocated by SYSGEN based on the "size" parameter on the PMEM option of the :RESERVE Command. The STM chain headers are filled in at system boot time when the free pages are established.

Partition Control Table (PCT) Format

The PCT has one of two formats. If the PCT contains an STM chain header, the format is



otherwise, the PCT has the format




where

Word 0

No. pages is a count of the number of pages in the partition. Set by SYSGEN.

Start page is the page address of the start of this partition. Set by SYSGEN.

Word 1 is an instruction used to maintain the total number of free pages.

Words 2 and 3

Page allocation map is a doubleword bit table used to indicate the availability of a given page in this partition. A value of 0 indicates that the page is available, a value of 1, that the page has been allocated. Bit 0 represents the first page in the partition, bit 1 the second page, etc.

This entry is initialized to all zeros by SYSGEN.

Words 4 through (3 + n)

Page locator flags:

Bit 0 – always 0.

Bit 1 – (P/S) set to 1 if this page acquired by primary task; set to 0 if secondary task.

Bit 2 – (S) set to 1 if memory management is swapping this page for some other real page; set to 0 if swapping is not in progress.

Bit 3 – (STM) set to 1 if the page has been released to secondary Task Management; 0 if not.

Bits 4-7 – Unused.

Page locator if P/S (bit 1) is 0, this is the real word address of the segment descriptor where this page is allocated. If P/S is 1, this is the LMI index of the primary load module that acquired this page. These entries are used by memory management in swapping real pages and by task termination when freeing real memory.

## I/O Lock Table (IOLOCK)

Purpose

The table maintains a count of the number of ongoing I/O activities in each page of real memory. It is used by Memory Management to prevent roll-out of I/O active pages.

Type

Serial consecutive entries in CP-R table space.

Logical Access

The location of IOLOCK is established by SYSGEN via DEF. The Real Memory Page Number is used as an index to find the corresponding lock entry.

Overview of Usage

The IOLOCK table, created by SYSGEN, reserves sufficient space to accommodate n one-byte entries, where n is the number of real memory pages specified on the CORE option of the :MONITOR Command.

Prior to calling IOCS, File Management increments by 1 the IOLOCK entry that corresponds to the real page(s) that will be effected by the I/O operation. At POST time the entry will be decremented by 1.

Prior to rolling out a segment Memory Management inspects the IOLOCK table so as not to roll out those pages with ongoing I/O activity.

I/O Lock Table (IOLOCK) Format

| | | | |
|---|---|---|---|
| Page 0 | Page 1 | Page 2 | Page 3 |
| Page 4 | Page 5 | Page 6 | Page 7 |
| Page 8 | Page 9 | Page 10 | Page 11 |
| Page 12 | Page 13 | Page 14 | Page 15 |
| | | | |
| Page n-11 | Page n-10 | Page n-9 | Page n-8 |
| Page n-7 | Page n-6 | Page n-5 | Page n-4 |
| Page n-3 | Page n-2 | Page n-1 | Page n |

where n is the number of real memory pages.

## Task-Controlled Tables

The tables shown in the subsection are task controlled, i.e., contain task related data. Figure 44 shows the overall relationship of the task-controlled tables and data.

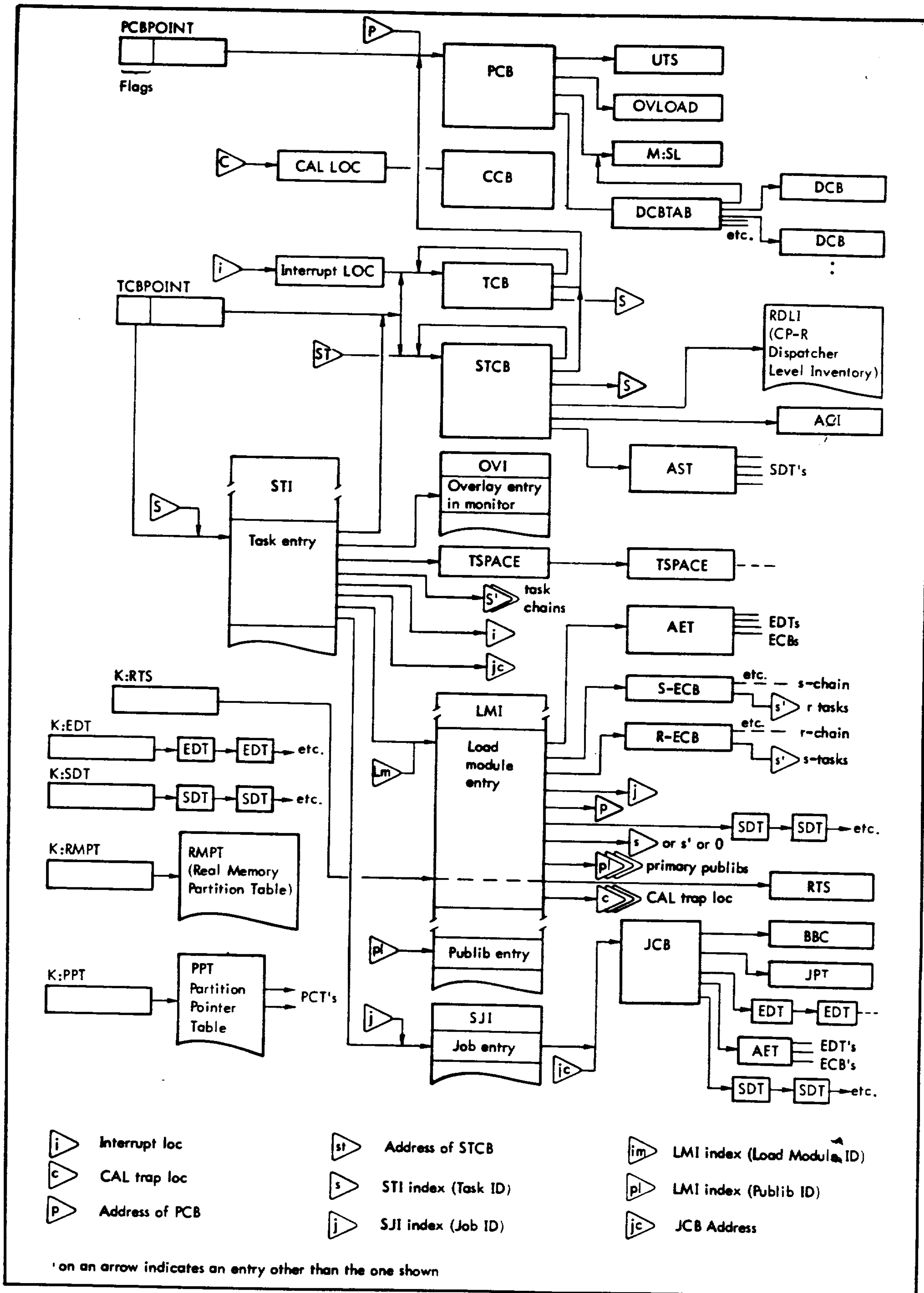


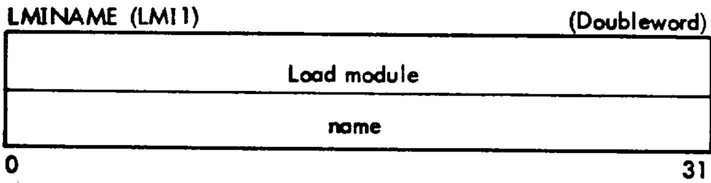

Figure 44. Relationship of Task Controlled Data

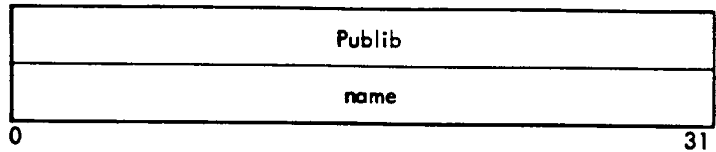
## Load Module Inventory (LMI)

LMIMAXR (LMI9) entry 0 contains the number of entries in the LMI tables.

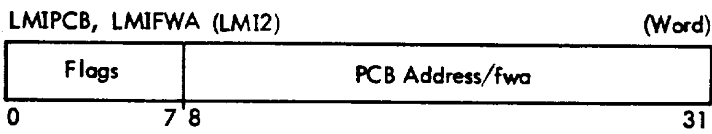
Usage for a Program | Usage for a PUBLIB

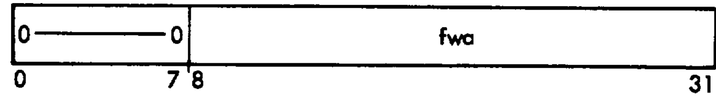
LMINAME (LMI1) (Doubleword)



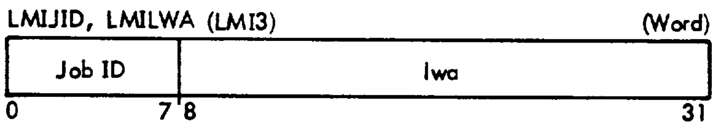



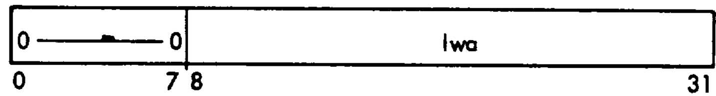

LMIPCB, LMIFWA (LMI2) (Word)



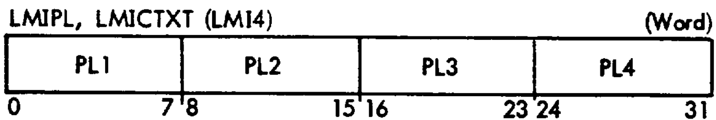



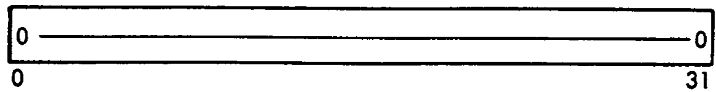

LMIJID, LMILWA (LMI3) (Word)



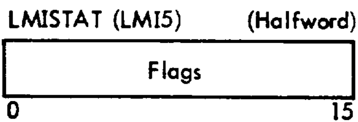



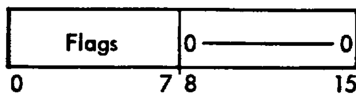

LMIPL, LMICTXT (LMI4) (Word)



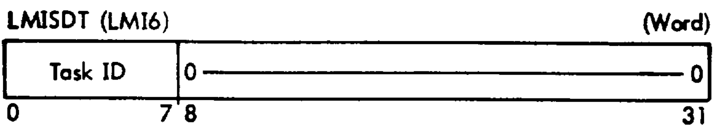



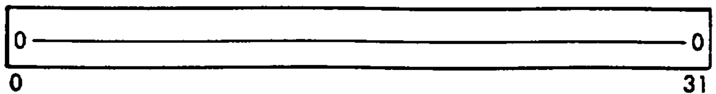

LMISTAT (LMI5) (Halfword)




LMISDT (LMI6) (Word)

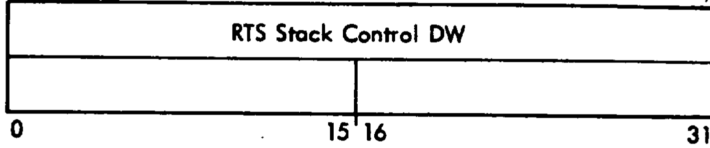


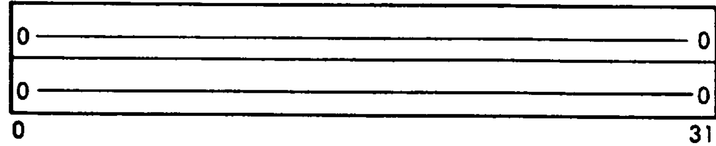



Usage for a Program

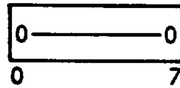
Usage for a PUBLIB

LMIRTS (LMI7) (Doubleword)

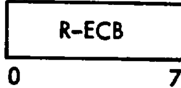


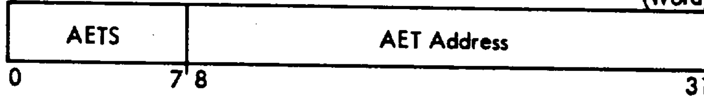

LMIMAXS (LMI8) (Byte)

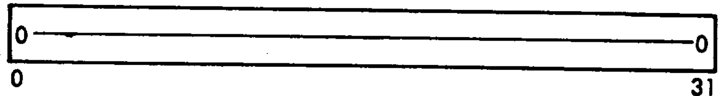



LMIMAXR, LMIUSE (LMI9) (Byte)

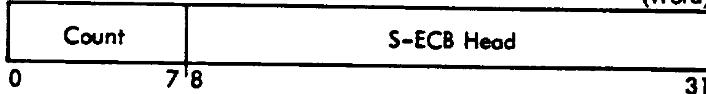


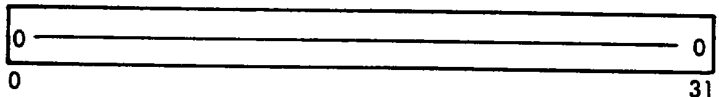
LMIAET (Word)



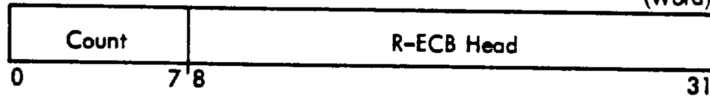



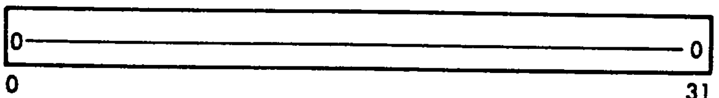

LMISECB (Word)



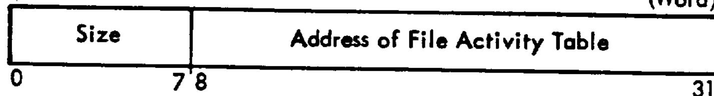



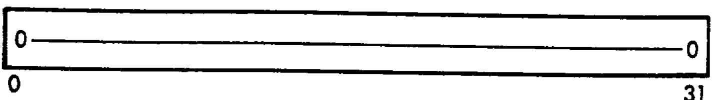

LMIRECB (Word)




LMIRFT (Word)



LMINAME (LMI1)

For user load modules – Task Name: User load module name, as received on the INIT or RUN call.

For Publibs – Publib Name: The file name of the Publib load module. The task or Publib name is stored by task ini.iation and remains unaltered during task execution.

LMIPCB, LMIFWA (LMI2)

Flags:

| Bit | Meaning if Set (1) |
|---|---|
| 0 | Task has CP-R priveleges |
| 1 | Task is background |
| 2 | Task is secondary |
| 3 | Task is mapped |
| 4-6 | Not used |
| 7 | Task is running with DEBUG |

PCB Address/fwa:

For user load modules – PCB Address: The location of the load module's PCB. This is also the first word address of the load module. The PCB address is stored by task initialization and remains unaltered during the task's execution. When central CONNECTs are requested to a primary load module, the PCB address and flags in the LMI entry are used for the TCB. The fwa is used for memory management during later task loads.

For Publibs – fwa: The first word address of the Publib load module. Fwa is set by task initiation when the Publib is loaded and remains unaltered during the Publib life.

LMIJID, LMILWA (LMI3)

For primary load modules – Job ID: The identification of the job to which the load module belongs; also the index of the job's entry in SJI. Load modules can only exist once within a job. This value is set by task initiation and remains unaltered during task execution.

For both User and Publib Load Modules – lwa: The last location used. The lwa is set by task initiation and remains unaltered during task execution. It is used to manage memory during later task loads.

LMIPL, LMICTXT (LMI4)

For primary load modules – PL1, PL2, PL3, and PL4: These bytes each contain a load module ID (index into LMI) of the Publibs being used by the load module. A zero indicates that the byte is not used. They are set by task initiation, remain unaltered during task execution, and are used by task termination to decrement Publib use counts and eventually release Publibs.

LMI STAT (LMI5)

Status Flags:

| Bit | Meaning if Set (1) |
|---|---|
| 0 | Termination has begun (TTFINAL entered). |
| 1 | Connected to CAL2. |
| 2 | Connected to CAL3. |
| 3 | Connected to CAL4. |

| Bit | Meaning if Set (1) |
|---|---|
| 4 | Background load module. |
| 5 | Secondary (dispatcher scheduled) load module. |
| 6 | Abnormal termination requested. |
| 7 | For a module being loaded, load was requested by INIT, not RUN. |
| 8 | Load module is being loaded. |
| 9 | PUBLIB that may be used by primary tasks. |
| 10 | PUBLIB that may be used by secondary tasks. |
| 11 | Termination (normal or not) requested. |
| 12 | Control "Y" sequence to be performed. |
| 13 | Load module that is running. |
| 14 | Primary load module that is waiting for memory to load (Run queued). |
| 15 | Break has occurred. |

LMISDT (LMI6)

For user load modules – Task ID: STI index for the secondary tasks; otherwise, zero.

LMIRTS (LMI7)

For user load modules – RTS Stack Control DW: The stack control doubleword for the load module's CP-R temp stack. Set up during loading, from information in the load module header. Used as a stack control doubleword by monitor services executing in the task's context. Accessed indirectly through K:RTS for dispatched and centrally connected tasks.

LMIMAXS (LMI8)

For user load modules – S-ECB: The maximum number of solicited ECBs to allow any single task running in the load module to have simultaneously. This is set at task initiation from the program header. As new S-ECBs are created, and the current S-ECB count incremented, it is compared to this limit and the load module aborted if the maximum is exceeded.

LMIMAXR (LMI9)

For user load modules – R-ECB: The maximum number of request ECBs to allow any single task running in the load module to queue. Used as S-ECB maximum above.

LMIAET

AETS (byte 0): The length of the Associative Enqueue Table, in entries.

AET Address: The first word address of the Associative Enqueue Table for job level controlled items. The AET space is reserved as each load module is initialized. Enough space is acquired to hold the maximum number of ENQs as specified in the task's load module header. This control word does not change during task executions. At task termination, the AET space is released.

LMISECB

Count (byte 0): Current count of the number of ECBs in the solicited ECB chain.

S-ECB chain head: Address of the oldest solicited ECB in the S-chain. When a load module is initially loaded, the solicited ECB chain is empty. As service requests are made which create S-ECBs, they are added to the S-chain, and the count is incremented. If the current count exceeds the maximum allowed as specified in LMIMAXS, execution of all the tasks in the load module is immediately suspended (primary tasks are disconnected), and the load module is abnormally terminated. As services are checked, the S-ECB is de-linked from the chain and the count is decremented.

LMIRECB

Count (byte 0): Current count of the number of ECBs in the request ECB chain.

R-ECB chain head: Address of the highest-priority request ECB in the R-chain. When a load module is initially loaded, the request ECB chain is empty. As service requests are made of the load module (signals if user load module), they are added to the request chain in priority sequence, with the last request being placed at the end of its priority group. The current R-ECB count is incremented and compared to the maximum allowed in LMIMAXR. If it is greater, all member tasks are suspended and the load module is abnormally terminated. As the R-ECBs are posted by the R-task, they are delinked from the R-chain and the current count is decremented.

LMIRFT

Size: Size of the File Activity Table (in words)

Address of File Activity Table: Address of a byte table in TSPACE which is parallel to the RFT and contains the same number of entries as the RFT. The table is used to maintain the number of DCBs open to disk files by a particular load module.

## System Task Inventory (STI)

### Purpose

The System Task Inventory is the key to all controls for tasks. It contains an entry for all primary and secondary, user and CP-R tasks currently defined. For each task, it contains the identification of the task's job and load module, priority, and linkage to other control blocks.

### Type

Parallel in CP-R Table Space

### Logical Access

An STI entry is addressed using the task ID as an STI index into each of the parallel subtables.

If a task is in execution, the task ID is in byte 0 of TCB POINT.

If a task is not in execution and the task ID is not known, then:

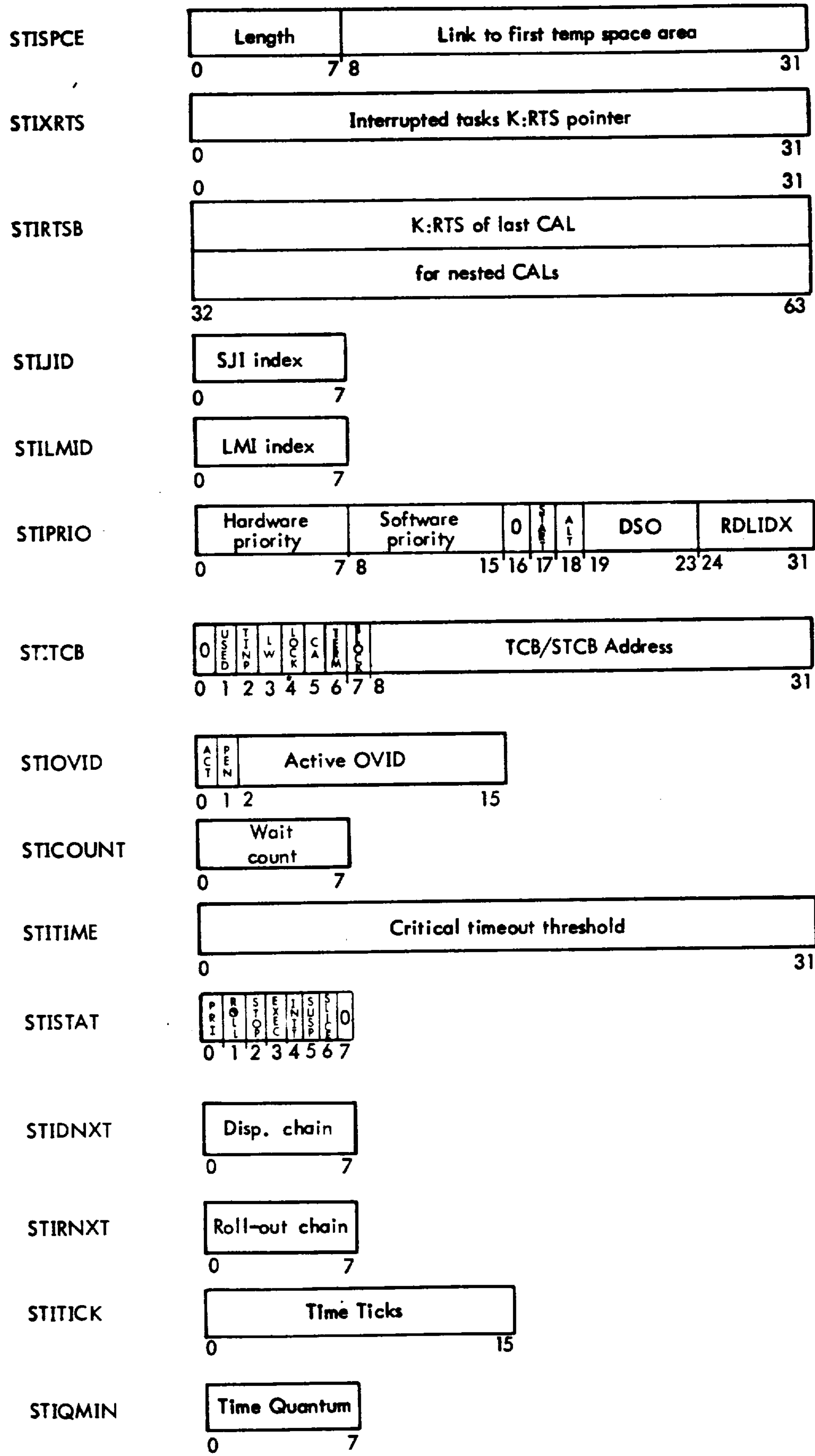
Primary tasks can be uniquely identified by a search for equality on the interrupt priority to which they are connected.

Secondary tasks must be located by searching the LMI for a task name/job ID match. The LMISDT contains the secondary task ID.

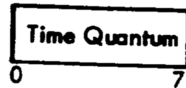
### Overview of Usage

The STI table space is allocated by SYSGEN, reserving enough entries in each subtable to satisfy the TASKS option on the :RESERVE command, plus a fixed number for internal CP-R tasks. The CP-R task entries are initally set by SYSGEN/IPL. The user entries are all zero.

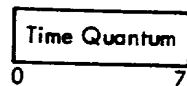
**STILMID entry 0 contains the number of entries in the STI tables.**



STIQMAX




STIQSWAP




**STISPCE**

Head of the TSPACE chain. The chain represents all of the temp space that has been obtained by the task.

**STIXRTS**

Location in which the task's RTS pointer is saved when interrupted by a higher-priority centrally connected task.

**STIRTSB**

RTS Control Doubleword at the last entry to a CAL1 processor. This address is the STIRTS value after CAL1 entry has stored the caller's R0-R15, PSD and context. It is used by the monitor to quickly locate the register values for effective address resolution or error value setting, and by CAL1EXIT to ignore residual data in RTS. STIRTSB is set to 0 at task initiation and should always be 0 except when the task is within CAL1 processing.

**STIJID**

Identification of the job to which the task belongs, and index into the SJI. This is set when the task is defined, and is not altered during execution.

**STILMID**

Identification of the load module to which the task belongs, and index into the LMI. This is set when the task is defined, and is not altered during execution.

**STIPRIO**

Priorities (bits 0 - 15):

If the task is primary:

Byte 0 is the address corresponding to the interrupt level, -X'4F'; byte 1 = X'00'.

If the task is secondary:

Byte 0 is the address -X'4F' of the CP-R dispatcher level at which the task is dispatched and executed. Byte 1 is the software priority within the dispatcher level (X'01' through X'FF', where X'FE' = control task and X'FF' = background).

This value is set when the task is defined. If the task is secondary, it will be altered as the task's priority is altered by MODIFY calls or internal CP-R priority-changing logic.

START (start pending on task):

The secondary task has been STARTed, and the start has not been honored. This bit is set by the START CAL processor and reset by the dispatcher when it causes the reversing of the STOP bit in STISTAT.

ALT (Dispatch using the alternate PSD):

The secondary task will be dispatched using an alternate PSD the next time. The current PSD will be found in the Alternate PSD after dispatch and Alt will be reset.

DSO (Dispatcher Skip Override):

The task is in a CAL processor modifying a shared segment marked DS (Dispatcher Skip). This value, if nonzero, prevents rollout of the task so that the CAL processor can complete the service while holding up other users of the shared segment.

RDLIDX (RDLI index): Dispatcher ID.

### *STITCB

Used bit (bit 1) = 1 if entry is being used.
= 0 if entry is free.

TINP bit (bit 2) = 0 if task is not waiting for terminal input.
= 1 if task is waiting for terminal input.

LW bit (bit 3) = 1 if task is in a long-wait.
= 0 if task is not waiting or is in a normal wait.

Lock bit (bit 4) = 1 if task has locked itself in memory.

CA (bit 5) = 1 if the task is to be chained after its priority group when returning to the dispatcher.
= 0 if the task is to be placed at the beginning of its priority group. CA is reset to 0 on every requeue.

Term (bit 6) = 1 if the task is executing in the Task Termination phase.

Block (bit 7) = 1 if the task has executed a non-I/O service with wait (including STOP).

TCB/STCB address is TCB real address if task is primary STCB 1-1 address if task is secondary.

Task initiation and CONNECT acquire STI entries and store the TCB address or STCB address. These fields are constant throughout the task's life. The remaining indicator bits are initialized to zero and are modified during execution by service calls. Task termination resets STITCB to zero, releasing all task control information.

### STIOVID

ACT bit = 1 if the overlay is active.

PEN bit = 1 if the overlay is being read from the disk.

Active OVID is the index into the Monitor Overlay Inventory.

### STICOUNT

Wait count: The number of ECBs in the S-ECB chain, which must be posted prior to the task leaving the wait state. Only ECBs with the WD (Wait Decrement) bit set will decrement the wait count at posting time. If the wait count is nonzero, the task is roadblocked. STICOUNT is zeroed at task initialization, is set nonzero by the CALs that cause waits and task termination, and is decremented by the ECB posting logic.

### STITIME

Critical Timeout Threshold: When placing any task into a roadblocked or wait state, the ECBs being checked (WD = 1) are scanned and the most critical time threshold is extracted and placed in STITIME. On subsequent

timeout passes, the threshold is compared to the value of K:UTIME to detect timeouts. If a timeout has occurred, the ECB chain is scanned again to locate any or all timed-out events, and the posting is done with a completion code of X'67'. If wait count is still not zero, the setting of the critical time is repeated.

### STISTAT

Status flags that inhibit dispatching of the task, as described below.

The dispatcher examines the status of all tasks in the dispatch chain. If the content is nonzero, the task is considered ineligible for dispatching.

Primary tasks always have a status of X'80', as set by CONNECT. Secondary tasks will have an initial status of X'00' or X'20'. The secondary task status bits are altered during execution as described below:

| Status | Bit | Set by | Reset by |
|---|---|---|---|
| Primary | 0 | Connect CAL | Task Termination |
| Rolled Out | 1 | Roll-Out | Roll-In |
| Stopped | 2 | STOP, EXIT task initiation without execution | START, task initiation with execution |
| In execution | 3 | Dispatcher when dispatching, loading PSD and registers | Dispatcher when returning PSD and registers |
| In initialization | 4 | Task initiation | Task initiation |
| Suspended | 5 | WAIT CAL | Any event affecting task status |
| Slice | 6 | Task initiation | Task termination |
| Swapped | 7 | Memory Management | Dispatcher |

### STIDNXT

Dispatcher Chain – the STI index of the next task in the dispatch chain. Entry 0 contains the chain head to the highest priority task in the system, primary or secondary.

This chain continues through all tasks in the system. It is used by the dispatcher to locate the next secondary task to execute and the timeout routines to locate those primary services that need timeout.

As each task is created, it is added to the dispatcher chain and remains as a member of the chain until termination. Its position within the chain is changed as it changes priority or enters a wait state. A value of X'00' is the end of the chain.

### STIRNXT

Roll-out Chain – the STI index of the next task in the roll-out chain. Entry 0 is the head of the roll-out chain and contains the first task to be rolled-out. The chain continues through all tasks until the highest priority task, which is the last task to be rolled out. A value of X'00' is the end of the chain.

The chain is strictly in order of priority. As each task is created, it is added to the roll-out chain and remains as a member of the chain until termination. Its position within the chain is changed as it changes priority or enters a wait state. The roll-out chain is the exact inverse of the dispatcher chain, and each serves as a backward link for the other.

### STITICK

Task Execution Time – contains a count of the number of clock ticks of execution time this task has received. For secondary tasks, the CP-R dispatcher converts this value to milliseconds and accumulates the total execution time in a one-word entry in the tasks Job Control Block.

### STIQMIN

Execution time-quantum remaining – contains a count of the number of clock ticks remaining before this task must give up control of the CPU. The count is decremented by the CP-R clock routines and refreshed by the CP-R dispatcher.

### STIQMAX

In-core execution time used – contains a count of the number of clock ticks of execution time this task has used since the task was last rolled-out. The count is incremented by the CP-R clock routines and is used by the CP-R dispatcher and memory management executive to control memory use between time-sliced tasks.

### STIQSWAP

Swap time threshold – contains a count of the number of clock ticks that a swapped out task must remain swapped out. The count is decremented by the CP-R clock routines and is referenced by the CP-R dispatcher.

## Task Control Block (TCB)

Purpose

The TCB provides the context save area, system pointers, partial entry linkage and entry PSD for centrally connected primary tasks. Each primary task has its own TCB.

Type and Location

A TCB is a serial table in the users memory at a location provided by the user in the connect call.

Logical Access

The TCB for a primary task is pointed to by:

- The XPSD in the interrupt location.
- TCBPOINT during the task's execution.
- The STI entry corresponding to the primary task.

Figure 45 illustrates the logical links between the TCB and other system control data.

Overview of Usage

The TCB content is initialized by the CONNECT service routine. When the primary task is entered, the context of the interrupted task is saved in the TCB, including the interrupted-tasks TCB and PCB pointers which are swapped with those of the primary task that is being entered. When exiting the level, the central exit logic swaps the TCB and PCB pointers which restores the TCB to the original values. The registers and PSD are also restored.

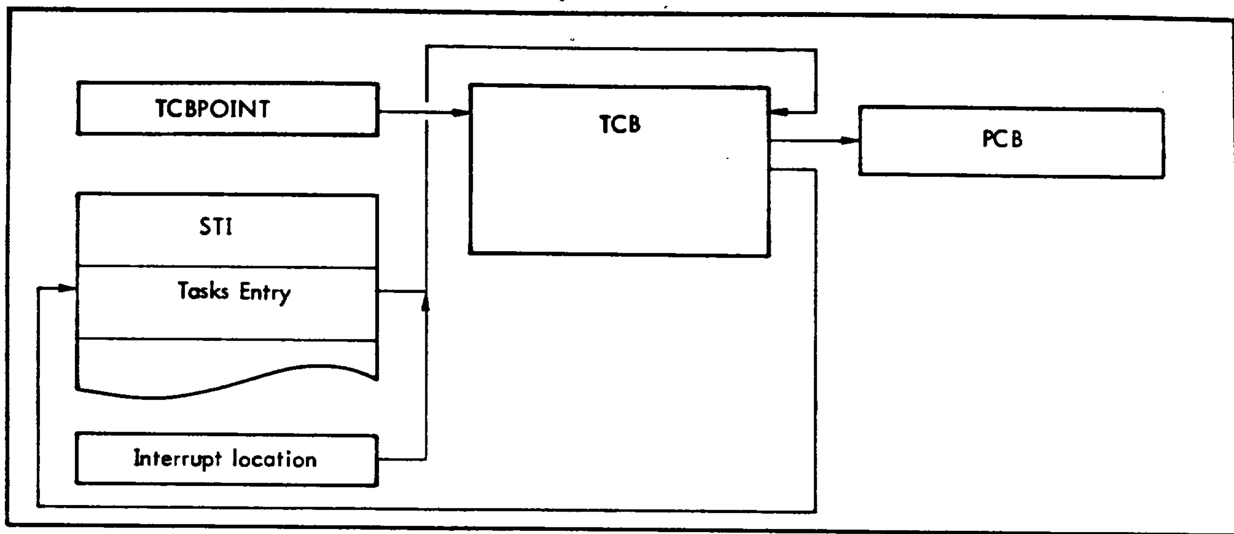
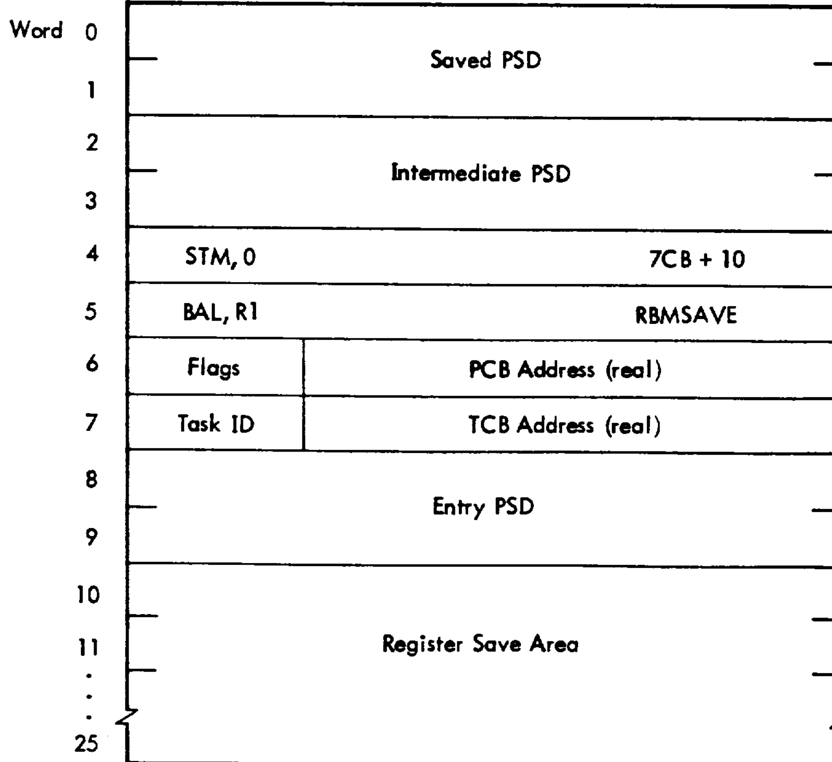



Figure 45. Relationship between a Primary Task Control Block and Other Control Blocks

**Task Control Block (TCB) Format**



where

Saved PSD (words 0, 1) is the PSD of the task the primary task interrupted at its last entry.

Intermediate PSD (words 2, 3) is the PSD loaded by the XPSD command at entry. The contents of this PSD are set by CONNECT to all zeros with these exceptions:

Instruction address – TCB + 4

Condition Code = the number of registers to be saved with the STM command in TCB + 4. CC = 0 if the CONNECT command specified that all 16 registers be saved via the central connection.

Since the XPSD does not alter the register block value in the PSD but leaves that of the interrupted task (LP = 0), the Register Block Pointer = 0.

STM, BAL commands (words 4, 5) are commands executed as part of the central connection entry logic. STM causes the number of registers requested to be saved, and BAL enters the remainder of the central connection logic (RBMSAVE).

Flags (word 6) have the following meaning:

Bit 0 = 0 for user task
= 1 for CP-R task

1 = 0 for foreground task
= 1 for background task

2 = 0 for primary task
= 1 for secondary task

3 = 0 for real addressing
= 1 for virtual addressing

4 = 0 if reserved

5 = 1 if the task is to be reentered instead of exited at EXIT. This bit is transient. It is set when end-action triggers are performed, and reset during RBMSAVE and when reentry occurs. It can exist only in TCB + 6.

PCB Address (word 6) is the real address of the PCB in the load module to which the task belongs.

Task ID (word 7) is the index into STI of the task's entry.

TCB Address is the real address of the first word of the TCB.

Note: When a task is active, flags, PCB address, task ID and TCB address contain the values for the interrupted task versus the primary task corresponding to the TCB.

Entry PSD (words 8, 9) is the PSD to be loaded when entering the primary task. All bits are zero except those specified otherwise on the CONNECT call as follows:

Master/Slave – as specified

Decimal and Arithmetic Masks – as specified

Instruction Address – callers start address

Write Key – 10 (foreground)

CI, II, EI – inhibits as specified

Register Save Area (words 10 through 25) are the save area for the registers of the interrupted task.

## Secondary Task Control Block (STCB)

### Purpose

The STCB contains all controls for software scheduled secondary tasks which reflect the execution status and memory usage of the task.

### Location and Type

The STCB is a serial control block in TSPACE.

### Logical Access

The STCB is pointed to by the following:

TCBPOINT (during task's execution only)

STI entry corresponding to the secondary task

The XPSD in the interrupt location corresponding to the CP-R Dispatcher Level (RDL) immediately above the Task Level (STL) (during execution only).

Figure 46 illustrates the logical links between the STCB and other system control data.

### Overview of Usage

A user STCB is created by task initialization if the load module requested is secondary. CP-R task STCBs are included in the resident portion of the task's code, as are all control blocks "lower than" the STCB. The initial STCB content set by task initiation is described for each data element, as is the element usage. The STCB is used by the CP-R control functions, dispatcher, memory management services and roll-in/roll-out during the life of the task. STCB space is released by task termination.

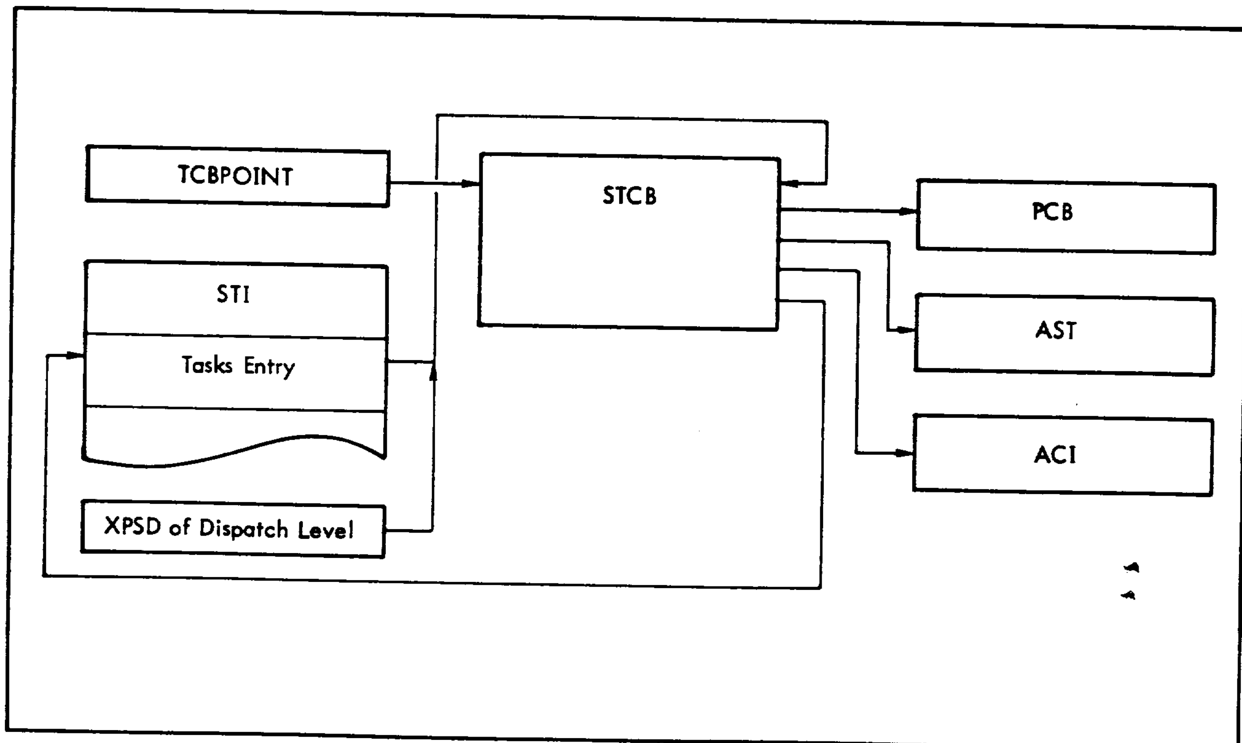
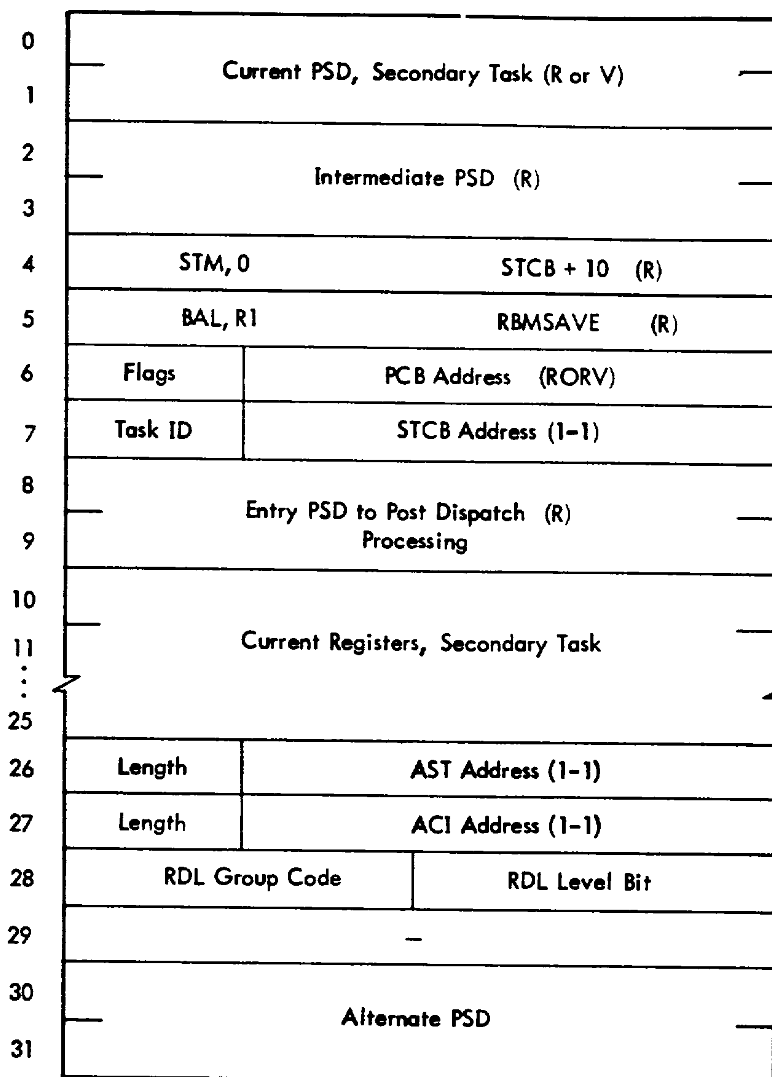



Figure 46. Relationship between Secondary Task Control Block and Other System Control Data

Secondary Task Control Block (STCB) Format




where

Current PSD of the secondary task (words 0, 1) either the PSD to be loaded on the next dispatch (if not in execution), or that loaded on the last dispatch (if in execution).

Task initiation resets the initial PSD to all zeros except:

MS = 0 if master mode.
= 1 if slave mode.

MM = 0 if unmapped.
= 1 if mapped.

IA load module entry address unless run with Debug, in which case IA is the Debug entry address. Address is real or virtual as per mode of execution.

Write Key = 01 if foreground secondary task.

Entries to RDL subsequent to dispatching the task save the current PSD.

Intermediate PSD (words 2, 3) a PSD to transfer control to real address STCB + 4. All other intermediate PSD bits are zero. Task initiation sets the intermediate PSD address which remains unaltered.

STM and BAL commands (words 4, 5) stored by task initiation to cause context saving and swapping via RBMSAVE after a task has been executing. These commands are set by task initiation, real addresses, and are not altered.

Flags (word 6) the task flags set by task initiation as follows:

Bit 0 = 0 for user task
= 1 for CP-R task

1 = 0 for foreground task
= 1 for background task

2 = 1 for secondary task

3 = 0 for real memory, unmapped
= 1 for virtual memory, mapped

4 Reserved

5 Reserved

The flags are not altered during the task's life.

PCB Address the address of the task's Program Control Block, which is set by task initiation and not altered. Mapped secondary tasks will have the PCB Address in virtual. Unmapped secondary tasks (CP-R or primary initiation tasks) will have the PCB Address in real.

Task ID (word 7) the identification of the secondary task and index into the task's STI entry. This ID is set by task initiation and not altered.

STCB Address the 1-1 address of the STCB, set by task initiation and not altered.

Note: Words 6 and 7 are swapped with PCBPOINT and TCBPOINT when a task is executing, as is done with primary tasks. Therefore, between the time a task is dispatched (in execution) and its status is returned to the STCB by an RDL entry, words 6 and 7 contain the dispatchers PCBPOINT and TCBPOINT values. When a task is not dispatched, its own values appear. One secondary task can be "in execution" for each CT pair in the system. "In-execution" is equivalent to a hardware level being active. The task is either executing, or waiting for higher task to drop its interrupt level and return to the lower priority task.

Entry PSD (words 8, 9) a PSD to transfer control to clean-up processing for tasks returning from an "in-execution" state. After RDL is triggered and has saved context via RBMSAVE this PSD is loaded. It is all zeros except for IA which is the real address of RDLRTRN, is set by task initiation, and remains unaltered.

Current Registers (words 10-25) the registers to be loaded on the next dispatch (if not in execution), or those loaded on the last dispatch (if in execution). They are set randomly by task initiation and saved on all entries to RDL subsequent to the task being dispatched.

Length (word 26) the number of words in the Associative Segment Table (AST).

AST Address the first word address (1-1) of the AST. AST space is allocated by task initiation and the address and length are set in the STCB. This word is not altered.

ACI Address (word 27) The first word address (1-1) of the Access Control Image. The ACI space is allocated by task initiation and the address is stored in the STCB. This word is not altered.

RDL Group and Level (word 28) The group and level bits of the RDL Level under which the secondary task is currently queued. Set by the dispatcher queue maintenance routines.

Word 29 Spare.

Words 30, 31 Alternate Program Status Doubleword or alternate PSD to be used the next time the task is dispatched if ALT is in the STI=1. When ALT is honored by the dispatcher, this PSD and the current PSD in words 0 and 1 are swapped.

## Associative Segment Table (AST)

### Purpose

The AST provides the Task Dispatcher with a list of all segments whose map image must be loaded into the hardware map before the task can be dispatched. It is used by roll-out to record that an active segment was rolled out and task execution suspended; by roll-in to reactivate deactivated roll-out segments.

### Type

Serial consecutive entries in CP-R TSPACE.

### Logical Access

The AST is pointed to from the STCB, containing an entry for every segment defined in a Secondary Task Load Module including Root Part two. In addition CP-R adds a segment for Job Reserved Pages and Task Reserved Pages. The AST entries are ordered with respect to the segment's virtual starting address (i.e., the next higher adjacent AST entry represents a segment with a starting virtual address that is equal to or larger than the preceding AST entry.) AST is typically accessed by scanning for active entires that point to Segment Descriptors, which contain all information necessary to describe the segment.

Figure 47 shows the AST and its relationship to other system tables.

### Overview of Usage

The AST is established by task initiation from information in the task's load module header, and deleted by Task Termination.

When it dispatches a task, Task Management will scan the AST for entries that are active and will use the pointer to the SD to access the information necessary to load the task's map.

Memory Management uses the AST when performing segment operations for the user. The information contained in the AST and the SD is used to maintain the status of the task's segments for ACTIVATE, DEACTIVATE, and page operations (GETPAGE, RELPAGE).

Memory Management also uses the AST and SD information when performing roll-out and roll-in functions.

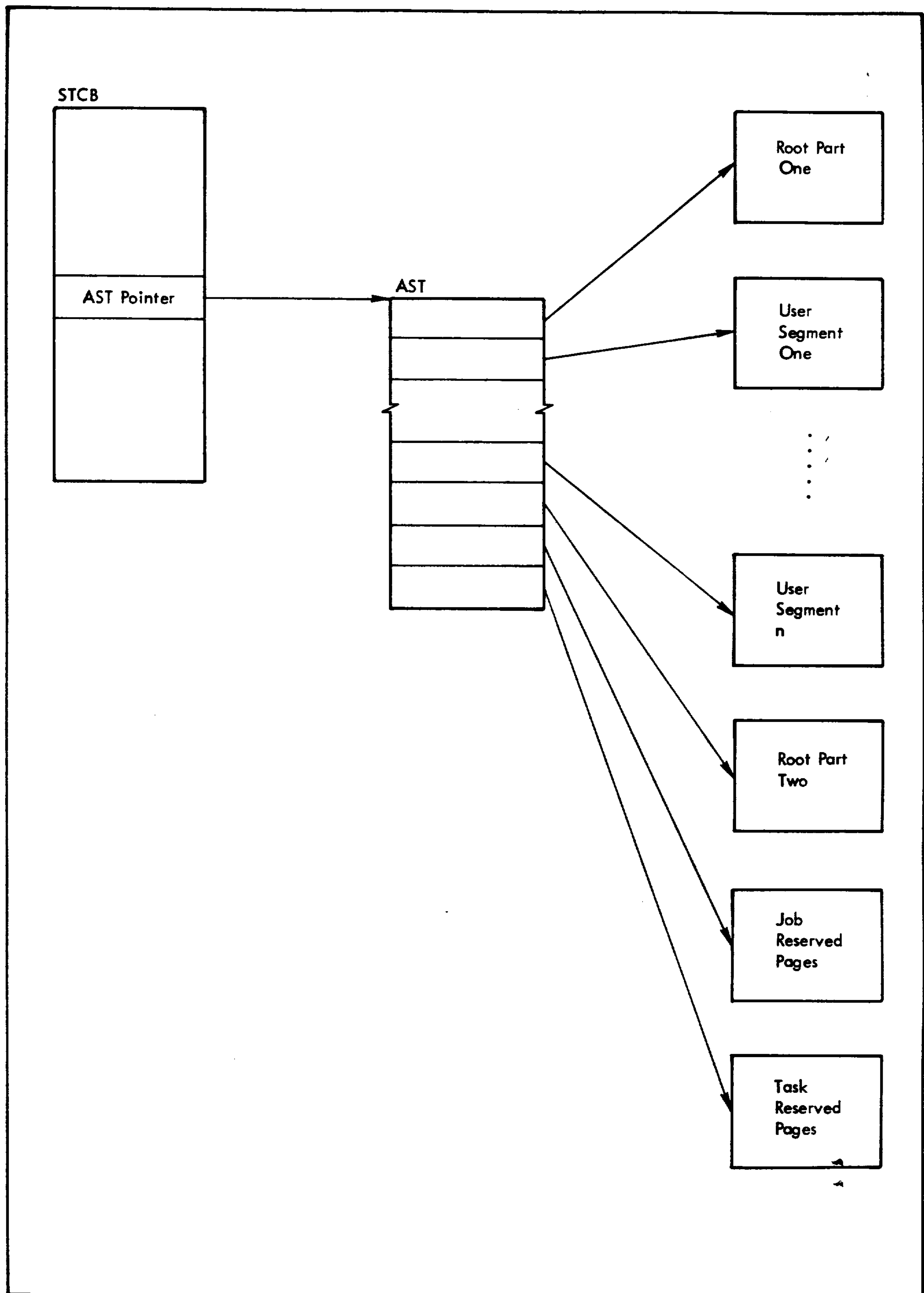



Figure 47. Relationship of AST to Other System Tables

Associative Segment Table (AST) Format

| Flags | SD address |
|---|---|
| | |
| Flags | SD address (Root part one) |
| Flags | SD address (first user segment) |
| | |
| Flags | SD address (last user segment) |
| Flags | SD address (Root part two) |
| Flags | SD address (JOB reserved pages) |
| Flags | SD address (TASK reserved pages) |

where

Flags

bit 0 — is always zero.

bit 1(A) — = 0 if this segment is not currently active to this task; = 1 if it is active. Set by ACTIVATE, GETPAGE. May also be set by roll-in routines. Reset by DEACTIVATE, ERASE or RELPAGE routines.

bit 2(FA) — is first activate flag. Set by Memory Management when the segment is first activated to prevent decrementing segment erase count before an ACTIVATE. Reset by ERASE.

bit 3(RD) — is Roll-Out Deactivated flag. Set by roll-out task when rolling out an active segment. Reset by roll-in task.

bit 4 (DS) — is Dispatcher Skip flag. If set, the Dispatcher, Skip (DS) flag in the Segment Descriptor (SD) addressed in bits 8-31 was set by the task owning this AST. See the description of the DS flag in a SD for dispatcher implications.

bit 5-7 — are unused.

SD address

bits 8-31 — is the real word address of the corresponding Segment Descriptor established by task initiation.

## Segment Descriptor (SD)

Purpose

The SD contains all segment-associated controls. It contains the access protection image, map image, the roll-out file position information and information relative to the status of each real page allocated to the segment. The SD is used by Task Management and Memory Management.

Type

Serial with four parallel subtables in TSPACE.

## Logical Access

The SD is pointed to from the AST. Alternatively, SDs may be accessed through the appropriate SD chain (task level (LMISDT), job level (JCBSDT), system level (S:SD).

## Overview of Usage

The SD is created by task initiation from information contained in the task's load module. Task initiation obtains sufficient space for the access image, map image, file image and page flags based on the size of the segment. When the segment is activated (by task initiation or later by the task itself), Memory Management obtains real memory resources for the segment filling in the access and map image and loading the images into the hardware map.

The file image is initialized when the segment is being rolled-out. The initial file assignment remains with the segment until the segment is terminated.

Memory Management controls the acquisition and release of real memory pages in response to segment or page operation calls. Additionally, memory pages may be freed by the roll-out process.

The SD is deleted at task termination time by Task Management.

## Segment Descriptor (SD) Format

| | Word | Bits 0–7 | Bits 8–15 | Bits 16–23 | Bits 24–31 |
|---|---|---|---|---|---|
| | 0 | DS (0), SS (1–2), [unused] (3–5), APM (6–7) | Map Image Address | | |
| | 1 | Count | [unused] | Control Start (S) | [unused] |
| | 2 | Segment Flags | | Segment Number | |
| | 3 | LMI | SDT FLINK Address | | |
| | 4 | Tasks Active | Tasks Erased | Tasks Using | Tasks Locked |
| | 5 | Real Page Count | Virtual Page Count (n) | ACI Count | No. Pages Rolled-Out |
| Access Image | | AC AC AC AC | AC AC AC AC | AC AC AC AC | AC AC AC AC |
| | | ⋮ | | | |
| | | AC AC AC AC | AC AC AC AC | AC AC AC AC | AC AC AC AC |
| Page Flags | | Page Flags (S) | Page Flags (S+1) | Page Flags (S+2) | Page Flags (S+3) |
| | | ⋮ | | | |
| | | Page Flags (S+n-4) | Page Flags (S+n-3) | Page Flags (S+n-2) | Page Flags (S+n-1) |
| File Image | | File Address (S) | | File Address (S+1) | |
| | | ⋮ | | | |
| | | File Address (S+n-2) | | File Address (S+n-1) | |
| Map Image | | Page(S) Real Address | | Page (S+1) Real Address | |
| | | ⋮ | | | |
| | | Page (S+n-2) Real Address | | Page (S+n-1) Real Address | |

where

Word 0

Bit 0 is the dispatcher skip (DS) flag. The DS flag indicates that a change of state of the segment is in progress. A task will not be dispatched if any of its segment descriptors have DS set, unless DS is set in the task's AST entry for the segment, also, indicating that the task is responsible for the state change.

Bits 1,2 indicate segment status:

00 – normal
01 – abort all using tasks
10, 11 – not assigned

Set by Memory Management when major errors are discovered in the composition of the Segment Descriptor.

Bits 3-5 Always zero.

Bits 6,7 is a 2-bit index (APM) into a mask table used by Task Management to load the hardware Access Protection registers. Set by task initiation.

Bits 8-31 is the real word address of the first word of the map image subtable. Initialized by task initiation.

Word 1

Bits 0-7 is a count of the number of words in the map image subtable, equivalent to the integer portion of the expression.

$$\frac{\text{Virtual Page Count} + 1}{2}$$

(Count is set by task initiation.)

Bits 8-14 always zero.

Bits 15-22 control start – the virtual page address of the first page of this segment. Set by task initiation.

Bits 23-31 always zero.

Note: Words 0 and 1 form the control doubleword used by an MMC instruction for loading the hardware map.

Word 2

Bits 0-15 Segment flags:

bit 0 unused.

1 indicates that real address correspondence (RAC) is required for this segment. Set by task initiator.

2 indicates that only secondary task memory (STM) is used for this segment. Set by task initiator.

Bits 3,4 is the access protection code (APC) for this segment:

00 – all access
01 – read and execute
10 – read only
11 – no access

Set by initiator.

Word 2 (cont.)

Bit 5 not assigned.

Bits 6-8 indicate segment type:

000 – task level
001 – job level
010 – system level
011 – Public Library
100 – not assigned
101 – task-reserved pages
110 – job-reserved pages
111 – system overlay

Set by task initiator.

Bits 9-11 indicate segment state – a code representing the current state and remaining dynamic throughout the life of the segment:

000 – being initiated or erased
001 – erased
010 – inactive
011 – active
100 – being rolled out
101 – rolled out
110 – being rolled in
111 – being loaded

Initially set by task initiator and manipulated by Memory Management.

Bits 12-15 not assigned.

Bits 16-31 represent segment number.

bit 16 = 0 if this is a user-numbered segment.
= 1 if this is a special system-numbered segment as follows:

bits 17-19:

000 – system overlay
001 – special system use
010 – PUBLIB
011
: not assigned
111

bits 20-31 – system segment number.

This field is established by task initiator.

Word 3

Bits 0-7 are the Load Module Inventory (LMI) entry index for a Public Library segment (type = 010). Set by task initiator.

Bits 8-31 are the real word address of the next SD in this level SD chain. Set by task initiator.

Word 4

Bits 0-7 contain a count of the number of tasks that have this segment active. Initially set to zero, count is incremented by 1 for each ACTIVATE directed to this segment. Decremented by 1 for each DEACTIVATE. When 0, the segment is a candidate for roll-out.

Word 4 (cont.)

Bits 8–15 contain a count of the number of tasks that have erased this segment. Initially zero, count is incremented by 1 for every first ACTIVATE and decremented by 1 for each ERASE. When this count goes to 0, the real memory that has been allocated to this segment will be released.

Bits 16–23 contain a count of the number of tasks using this segment. Initially set to 1 by task initiation when the segment is defined; thereafter incremented by 1 for each task sharing the segment. Decremented by 1 when using tasks terminate. When the count goes to 0, the segment descriptor is deleted.

Bits 24–31 contain a count of the number of tasks that have locked this segment. Initially 0, count is incremented by 1 for LOCK and decremented by 1 for UNLOCK. The segment will not be rolled out if the count is greater than zero.

Word 5

Bits 0–7 contain a count of the number of real memory pages currently in use by this segment. Maintained by Memory Management.

Bits 8–15 contain a count of the number of virtual pages required for this segment. Set by task initiation.

Bits 16–23 contain a count of the number of words in the Access Image Table. Used by Task Dispatcher when constructing the ACI table, the count is set by Task Initiation.

Bits 24–31 contains a count of the number of pages that have been rolled out.

Access Image Bits 0–31 is a parallel word table with two-bits per entry that contains an image of the access protection for this segment. Each word controls 16 pages of access protection. However, all 2-bit entries need not be used for any given segment depending on its starting virtual address and length; unused entries are set to no-access.

Page Flags Bits 0–31 is a parallel byte table that contains the status of each page in this segment:

bit 0 FFP, Foreground Preferred Page. A real page from a Foreground Preferred Parition. Set by Memory Management.

bit 1 FSA, File Space Allocated. Indicates that roll-out file space has been allocated to this virtual page. Set by Memory Management.

bit 2 RPP, Real Page Present. Indicates that a real memory page has been allocated to this virtual page. Set by Memory Management.

bit 3 PRO, Page Request Outstanding. Indicates that a request for this virtual page has been made but could not be satisfied immediately.

bit 4 ROL. Indicates that this page has been rolled out.

bit 5 not assigned.

bits 6,7 APC, Access Protection Code. Used to control access to reserved pages and therefore, present only if segment type is 101 or 110 (reserved pages). Set by Memory Management.

File Image Bits 0–31 is a parallel halfword table that contains the roll-out file granule displacement that corresponds to the particular virtual page address controlled by this segment.

Map Image 0–31 is a parallel halfword table that contains the real page address that corresponds to the particular virtual page addresses controlled by this segment. The last entry of this table may or may not be used depending on control start.

## Job-Controlled Tables

The tables shown in this subsection are job controlled, i.e., contain data associated with the job level of control. Figure 48 shows the overall relationship of the job-associated tables and data. (Note that the OPLBS and AET tables were described in the "General System Tables" subsection, being both job and task related.)

### System Job Inventory (SJI) Table

Purpose

All jobs are known to the system by means of the SJI. It contains one permanent entry for the CP-R job, one permanent entry for the background and one temporary entry for each foreground job active at a given time. For each job, it contains the EBCDIC job name, the JCB address, a bit indicating whether the SJI entry is in the process of being created, and length of the Job Control Block (fixed portion) in words.

Type

Parallel; in CP-R system table space with a fixed number of entries.

Logical Access

The SJI table location is known via a DEF on the subtable names. The job ID is the SJI index into each of the parallel subtables. If the job ID is known, job name and JCB location are obtained by using the job ID as an index into the appropriate subtable. If job name is known, table lookup will produce the job ID and JCB location. The SJI entry for CP-R is the first entry. The SJI entry for the background is the second entry (i.e., the CP-R SJI index is 1; the background SJI index is 2).

Overview of Usage

The SJI space is allocated by SYSGEN from CP-R system table space. Space is reserved for the maximum length specified by a SYSGEN parameter that limits the total number of jobs that can exist at any one time. This limit is some number less than 31, where one of the number is for background. In addition, one entry is made for the CP-R system job (not one of the number specified). The background entry is also always made and is the default (1 entry plus the CP-R entry) if no limit is specified.

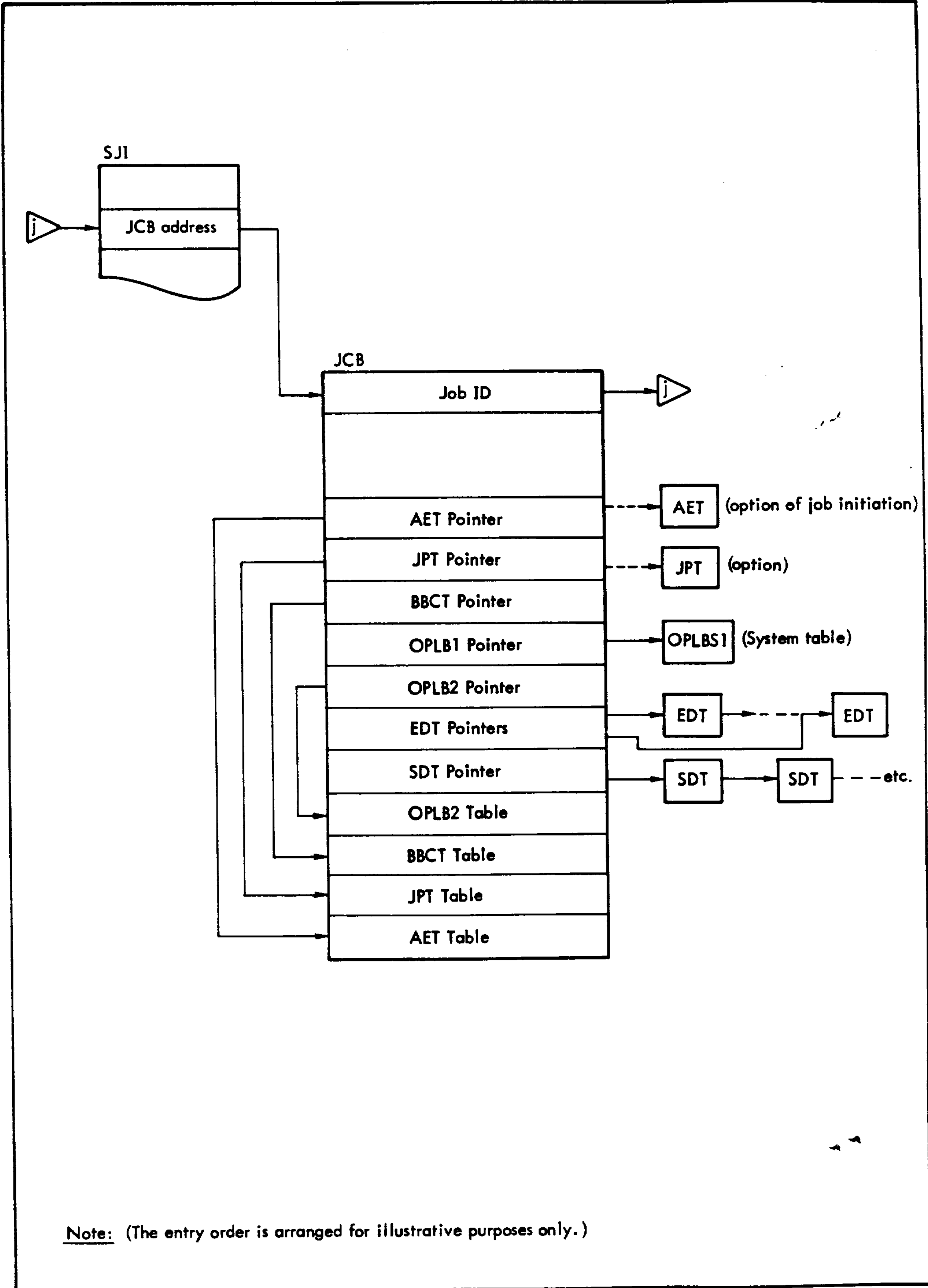



Figure 48. Relationship of Job-Associated Control Tables

The CP-R and background entries are initialized by CP-R INIT. All other entries are initialized to zero. SJOB requests cause job management to make new entries for foreground jobs. KJOB requests and requests from task management cause job management to delete entries. The JOBS option of the SYSGEN :RESERVE command specifies the number of user (background plus foreground) SJI entries.

System Job Inventory (SJI) Table Format

| Name | Content |
|---|---|
| SJI1 | Bit 0: 0; bits 1–7: No. of words in JCB; bits 8–31: JCB Address |
| SJI2 | Bits 0–63: EBCDIC job name |
| SJI3 | Bit 0: 0; bit 1: L; bits 2–7: 0———0 |

where L = 1 indicates job-initiation is in progress.

(SJI3, index 0 contains the maximum number of jobs allowed to be active at a given time, i.e., length of SJI.)

## Job Control Block (JCB)

### Purpose

The JCB contains information sharable or common to all tasks in the job. Each job has one JCB pointed to from the SJI. It contains job ID, trap controls, pointers to JCB tables, chain headers for job-related chained tables, and JCB tables. The JCB is comprised of a fixed length portion and two variable length subtables: The JPT and the AET. The JPT length is a SYSGEN parameter and may be long, and the AET length is dynamic. Therefore, at job creation, the job initiation routines may elect to exclude one or both of these two tables (which are themselves serial tables) from the fixed portion of the JCB. Two JCB flags are provided to indicate their presence in the fixed portion or linking from the JCB. If present in the fixed portion of the JCB, the respective flag is zero and the table pointer contains the number of words in the table in byte zero and the address in the JCB in bytes 1-3. If linked from the JCB, the respective flag is set to one and the table pointer contains the number of words of TSPACE in byte zero and the address of the table in bytes 1-3.

### Type

Serial; in CP-R TSPACE with consecutive entries and linked entries.

### Logical Access

JCBs are pointed to from the SJI. Job ID is the index into the SJI. JCB data elements occupy fixed positions in the JCB or are linked from pointers in fixed positions in the JCB. The Job Operational Label Table (OPLB), and the Blocking Buffer Control Table (BBCT) are part of the fixed portion of the JCB and are located by pointers in fixed locations in the JCB. The Enqueue Definition Table (EDT) and the Segment Descriptor Table (SDT) are tables whose entries are acquired as needed by tasks in the job. They are linked from pointers in fixed positions in the JCB. The Job Program Table (JPT) and the Associative Enqueue Table (AET) may be in the fixed portion of the JCB or may be linked from the JCB.

## Overview of Usage

The JCBs are allocated by job management from CP-R TSPACE. Space is acquired when the job is initiated and released when the job is terminated. The JCBs for the CP-R job and the background are established in CP-R INIT and are never released. The EDT and SDT entries are each linked in a chain from the JCB. EDT entries are acquired and released by resource management.

## Job Control Block Format

| Word | Content |
|---|---|
| | Bits: 0 1 2 3 4 5 6 7 8 13 14 15 16 23 24 31 |
| 0 | 0 \| D \| A \| J \| 0 0 \| DIS \| 0——0 \| T \| S \| Job Priority \| Job ID |
| 1 | 0 —————————— 0 |
| 2 | Trap Flags \| JTrap Address (Secondary) |
| 3 | Trap Flags \| JTrap Address (Primary) |
| 4 | 0 \| No. Entries \| OPLB1 Table Pointer → OPLBS1 system table |
| 5 | 0 \| No. Entries \| OPLB2 Table Pointer |
| 6 | 0 \| No. Entries \| BBCWT Pointer |
| 7 | 0 \| Max. length \| JPT Pointer |
| 8 | 0 \| Max. length \| AET Pointer |
| 9 | 0——0 \| EDT Pointer – head |
| 10 | 0——0 \| EDT Pointer – tail |
| 11 | Prompt Char. \| SDT Chain Head Pointer |
| 12 | Debug Control Word 1 |
| 13 | Debug Control Word 2 |
| 14 | Task ID \| Break Receiver Address |
| 15 | TJE BBCW Address |
| 16 | Size \| TJE Tab Settings Address |
| 17 | Size \| Next LM Name Address |
| 18 ⋮ | Account and User Names (5 words) |
| 23 | Job Time Accounting |
| 24 ⋮ | Blocking Buffer Control Word Table (BBCWT) (25 words) |
| | OPLB2 Table |
| | Job Program Table (JPT) (quadruple-word entries, DW bound) |
| ⋮ | Enqueue Table (AET) for job level enqueues (DW entries, DW bound) |
| | Bits: 0 31 |

These tables may not be contiguous to the JPT or to each other, in order that dynamic space may be more efficiently used.

where

Word 0

Bit 1 (D) = 1 for job running under DEBUG.
= 0 for normal run.

Bit 2 (A) = 0 if AET is in fixed portion of JCB.
= 1 if AET is linked from JCB.

Bit 3 (J) = 0 if JPT is in fixed portion of JCB.
= 1 if JPT is linked from JCB.

Bits 6,7 (DIS) are debug initialization status bits:

= 00 No DEBUG.
= 01 Needs initialization startup.
= 10 In initialization.
= 11 Fully initialized.

Bit 14 (T) is job-being-terminated bit.

Bit 15 (S) is job-being-initiated bit.

Words 12, 13

contain Debug Oplabel device assignment before initialization, BBCWT pointer after initialization.

Word 14

Bits 0-7 Id of task that will field break conditions.
Bits 8-31 Address of Break handler.

Note: Word 14 is zero if no Break handler is specified.

Word 15

contains the address of the Blocking Buffer Control Word for the TEL context blocking buffer.

Word 16

contains the TSPACE control word for the TSPACE block containing tab settings for a TJE line.

Word 17

contains the TSPACE control word for a TSPACE block containing the area name, file name, and account name for the next load module to load (for TJE) and background sequencing.

Words 18-22

words 18 and 19 contain the account name field. Words 20-22 contain the user name field.

Word 23

contains the total amount of CPU time that has been used by all secondary tasks in this job. The value is in milliseconds.

## Job Program Table (JPT)

Purpose

The JPT allows the user to specify the name of a load module to be used for execution of a task.

Type

Serial; in the JCB or linked from the JCB (depending on space requirements) with the maximum number of entries fixed at SYSGEN by the JPT option of the :RESERVE command. Default is zero entries. S:JPT contains the maximum number of entries specified. (The maximum that may be specified is 63 entries.)

Logical Access

The JPT is located from a pointer in a fixed position in the JCB. It is composed of doubleword pairs of EBCDIC task-name/load-module-name equivalences. Table lookup on task name is used to determine which load module is to be used for the task. (Byte 0 of the pointer, JCBJPT, contains the total number of words in the JPT table.)

Overview of Usage

Space may be provided in the JCB for the JPT, or the JPT may be linked from the JCB, depending on space requirements at the time the JCB is created. If it is included in the fixed portion of the JCB, it will be on a doubleword boundary pointed to from a fixed location in the JCB. If it is linked from the JCB, it will be on a doubleword boundary and will contain the number of entries specified at SYSGEN (space acquired as a power of 2). In either case, byte zero of the pointer word contains the number of words in the table and bytes 1-3 contain the address of the start of the table. On job termination, a flag (J) in the JCB will indicate which linkage applies and will release space appropriately. S:JPT contains the maximum number of entries allowed in the JPT.

Entries are made by tasks via the SETNAME system function call. SETNAME may be used across jobs. The default JCB is the calling task's job. SETNAME specifies a task-name/load-module-name pair of doublewords which are entered in the JPT. Task initiation uses table lookup on task name to determine if any entry exists for the specified task name. If no entry exists, the task name is assumed to be the desired load module name. If an entry exists, task initiation uses the corresponding load module for task execution. SETNAME is also used to delete JPT entries by providing a task name and blanks in place of the load module name. Duplicate task names are not allowed, so a replacement will occur if a SETNAME call uses a task name which is already represented in the JPT.

JPT Table Format

| Name | Content | Size |
|---|---|---|
| JPT | EBCDIC Task Name 1 | 1st doubleword |
| | EBCDIC Load-Module Name 1 | 2nd doubleword |
| | EBCDIC Task Name 2 | 1st doubleword |
| | EBCDIC Load-Module Name 2 | 2nd doubleword |
| | ⋮ | (etc.) |

where the EBCDIC Task Name characters and EBCDIC Load-Module Name characters are left-justified and blank filled.

## Enqueue Definition Table (EDT)

Purpose

The Enqueue Definition Table defines the current controlled items and resources in the system, and provides a mechanism for queuing outstanding requests for the item. This table is used in conjunction with the Associative Enqueue table.

Type and Location

Each EDT is a serial table in TSPACE.

Logical Access

Each EDT is a member of a chain whose head is either in CP-R location S:EDT (system level ENQs and all device resources) or in the JCB (job level ENQs). Figure 49 shows the overall relationship between system tables that directly or indirectly affect the EDT.

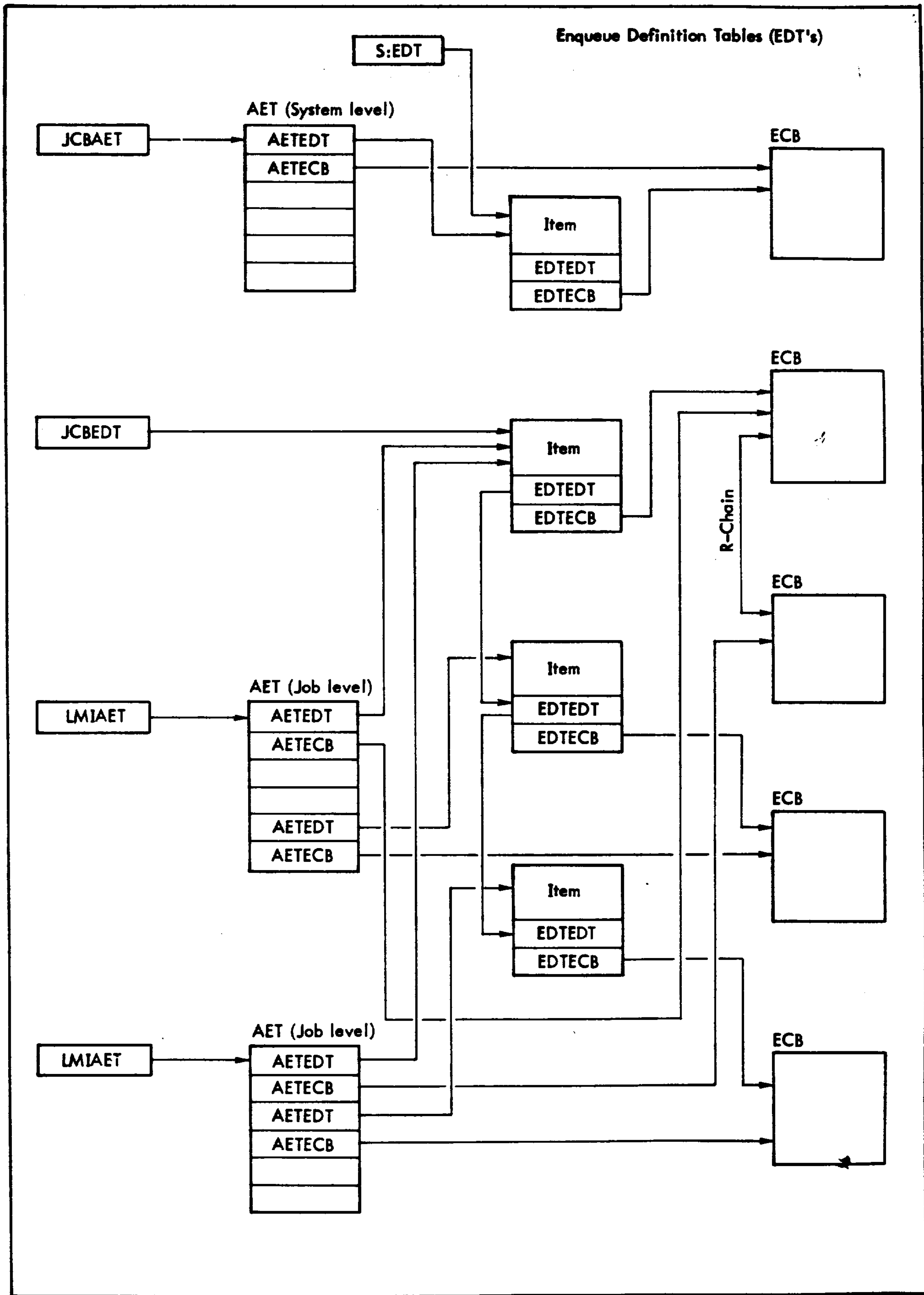
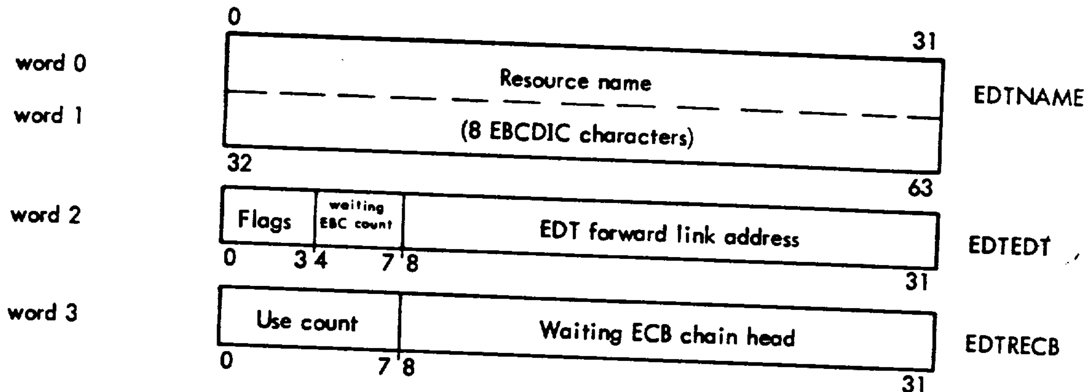



Figure 49. Enqueue/Dequeue Table Relationship

### Overview of Usage

The first acquisition of any resource causes a new EDT to be created and added to the appropriate chain. This allows later ENQs to know that the item is in use and check for conflicts. When conflicts do occur, ECBs are created to provide a waiting mechanism. The R-chain in the ECBs are used to connect the ECBs to the EDT for which they are waiting. This chain is in order of time within priority as are normal R-chains. When DEQ updates the EDT and detects that the item has been freed, it checks for the existence of waiting ECBs. If none exist, the EDT is removed from the EDT chain and deleted. If ECBs do exist, the DEQ assigns access to the item to the highest priority ECB in the chain and all lower priority ECBs which do not conflict, posting the ECBs as it does so.

### Enqueue Definition Table (EDT) Format




EDTEDT

Flags:

bit 0 = 1 This EDT is held by a job-level AET.
= 0 This EDT is held by a task-level AET.

bit 1 = 1 This is a system-level EDT.
= 0 This is a job-level EDT.

bit 2 Unused.

bit 3 = 1 This EDT is held by a sharable enqueue.
= 0 This EDT is held by an exclusive enqueue.

EDT forward link address: A pointer to the next EDT in the system or job level chain. Zero signifies the end of the chain.

EDTRECB

Use Count: The number of tasks which currently have acquired use of the item. If the ENQ is exclusive this count will be 1. If the ENQ is sharable, the count will be ≥ 1.

Waiting ECB Chain Head: The address of the ECB representing the highest priority outstanding ENQ for the item. 'R-ECB' of zero indicates no ENQs are waiting.

EDTNAME

Name: The name of the controlled item from the original ENQ call, or the device index, right-justified in the first word of the doubleword.

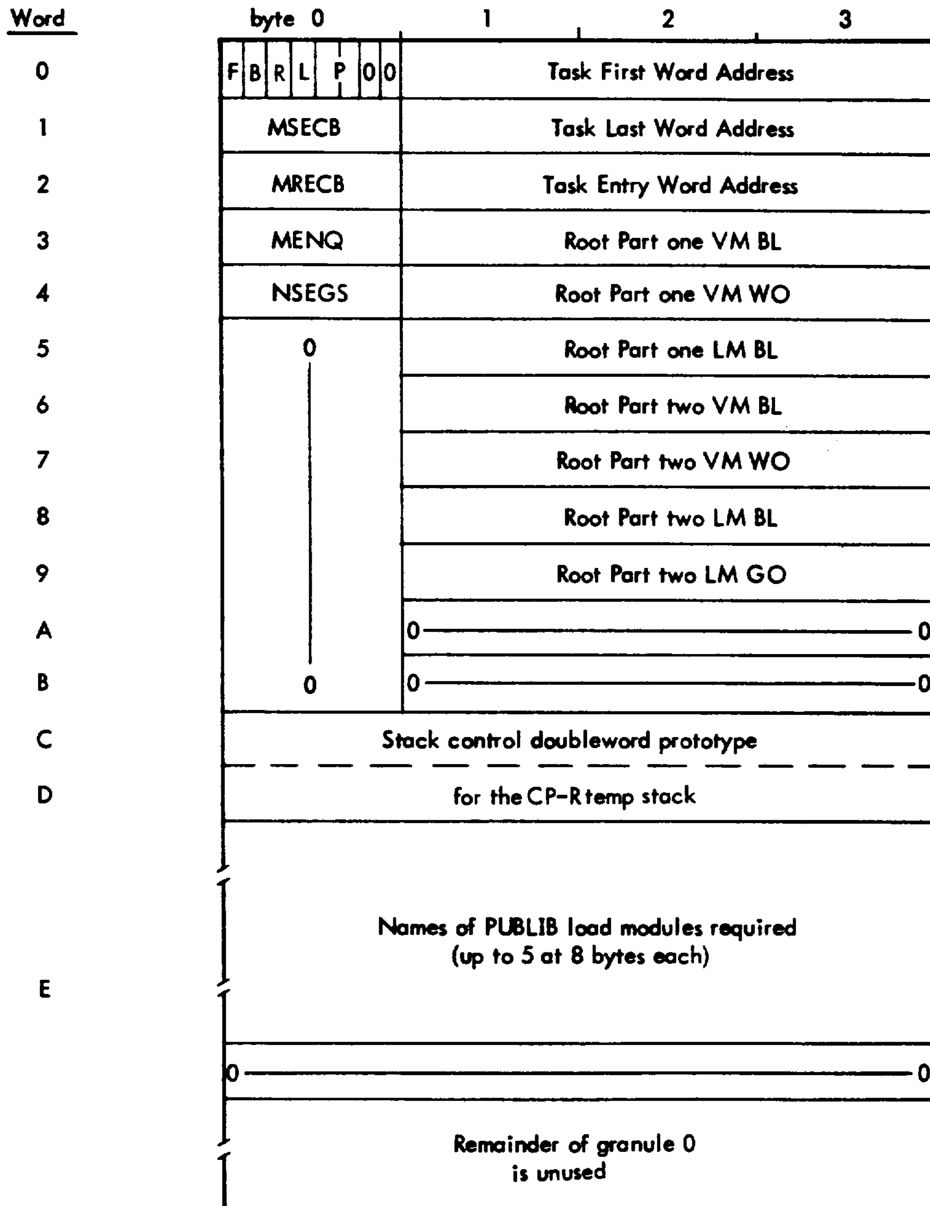
## Load-Module Data Structures

The control blocks and table shown in this subsection relate to load-module files.

**Load Module Headers**

The first sector of a load module file contains a block of information used to control the loading of the module and the allocation of system table space to it. This block is the load module header, and is written by the JCP Loader or Overlay Loader when the load module is created. A similar header is associated with each PUBLIB file.

Task Load Module Header




where

F = 0 for a background task.
 = 1 for a foreground task.

B = 0 for a program not linked for Simplified Memory Management (SMM).
 = 1 for a background program linked for SMM.

R = 0 if a program is not restricted from using foreground Preferred Memory.
 = 1 if it is restricted.

L = 0 for a task module (not a PUBLIB load module).

P = 01 for a secondary task.
= 10 for a primary task.

MSECB = maximum permitted number of solicted ECBs; S'FF' if system default is to be supplied.

MRECB = maximum permitted number of received ECBs; X'FF' if system default is to be supplied.

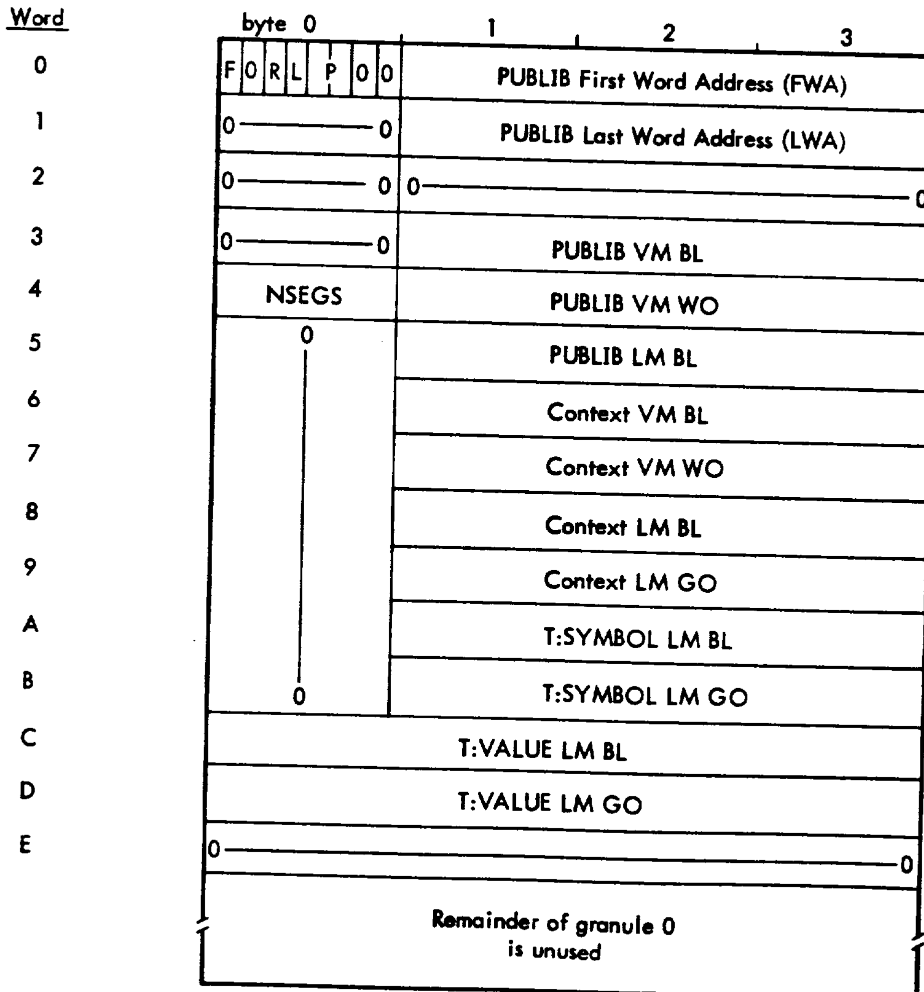
MENQ = maximum permitted number of resource enqueues; X'FF' if system default is to be supplied.

NSEGS = number of segments in task, to include both parts of root, PUBLIBs and DEBUG.

Legend:

BL Byte length
GO Granule origin
LM Load module
VM Virtual memory
WO Word origin

PUBLIB Load Module Header




where

F = 1 for a foreground load module.

R = 0 if the PUBLIB is not restricted from occupying foreground preferred memory.
= 1 if it is restricted.

L = 1 for a PUBLIB load module (not a task load module).

P = 01 for a secondary PUBLIB.
= 10 for a primary PUBLIB.

NSEGS = 1 for PUBLIB only.
= 2 for PUBLIB with context segment.

Legend:

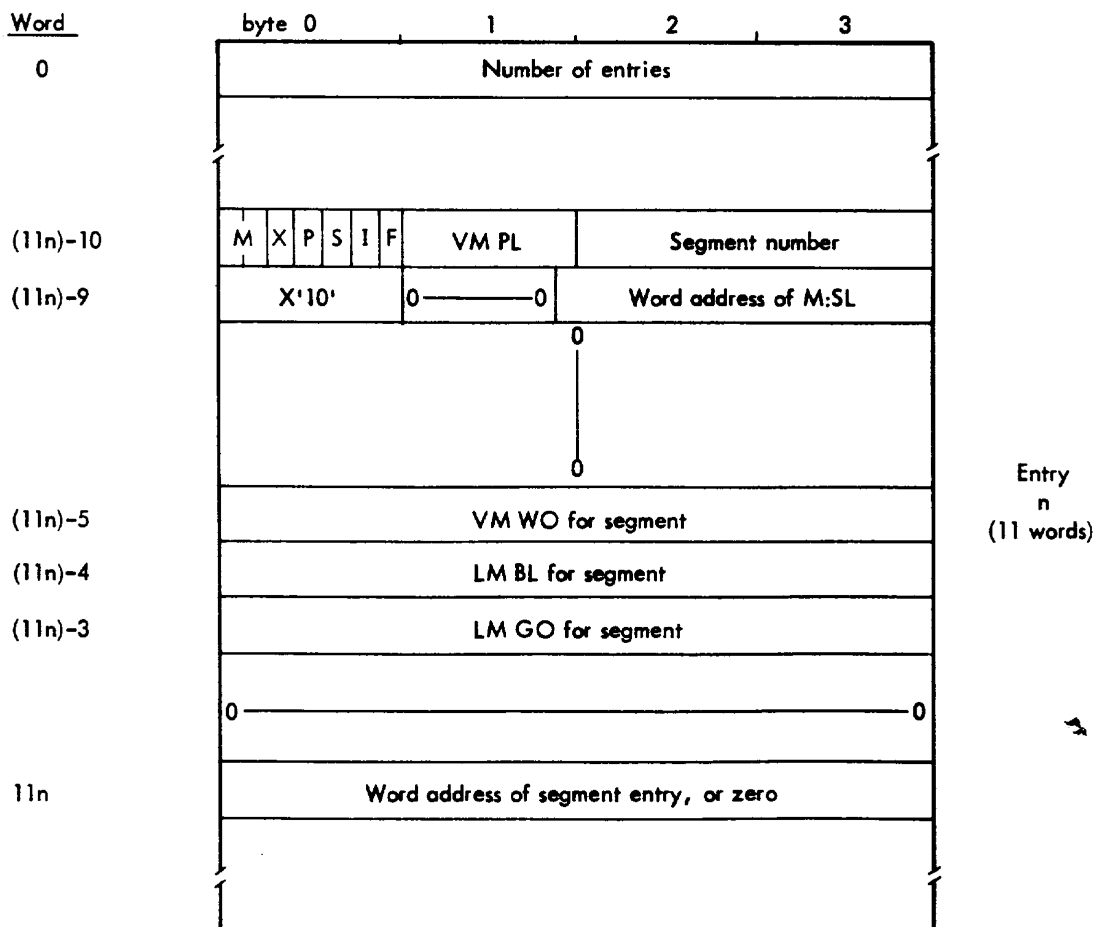
BL Byte length
GO Granule origin
LM Load module
VM Virtual memory
WO Word origin

Notes: FWA-LWA refers only to the PUBLIB segment, not the context. FWA = PUBLIB VM WO.

## OVLOAD Table (for Load Modules)

In the root of every load module (root part 2 if there is one) is the OVLOAD table for that module. This table provides information about the size and nature of each segment, its segment identification number, and the READ FPT to load it.

There is one entry for each segment, except for the root, PUBLIB, and PUBLIB-context segments, which are omitted.



where

| | |
|---|---|
| M = 00 | for any access |
| = 01 | for read/execute |
| = 10 | for read only |
| = 11 | for no access |
| X = 0 | (reserved for Memory Management) |
| P = 0 | segment image is in this load module |
| = 1 | segment is sharable and must be pre-loaded by another task, since its image was omitted from this load module (PRELOAD) |
| S = 00 | for non-sharable |
| = 01 | for job-sharable |
| = 10 | for system-sharable |
| I = 0 | for explicit activation required |
| = 1 | for initial activation with root (ILOAD) |
| F = 0 | if real-virtual address correspondence not required |
| = 1 | if required (FIX) |
| VM = | Virtual Memory |
| BL = | Byte Length |
| WO = | Word Origin |
| LM = | Load Module |
| PL = | Page Length |
| GO = | Granule Origin |

# 10. OVERLAY LOADER

## Overlay Structure

The Overlay Loader is itself an overlayed program with a root and the six segments as illustrated in Figure 50. The functions of the Root and segments is given in Table 6.

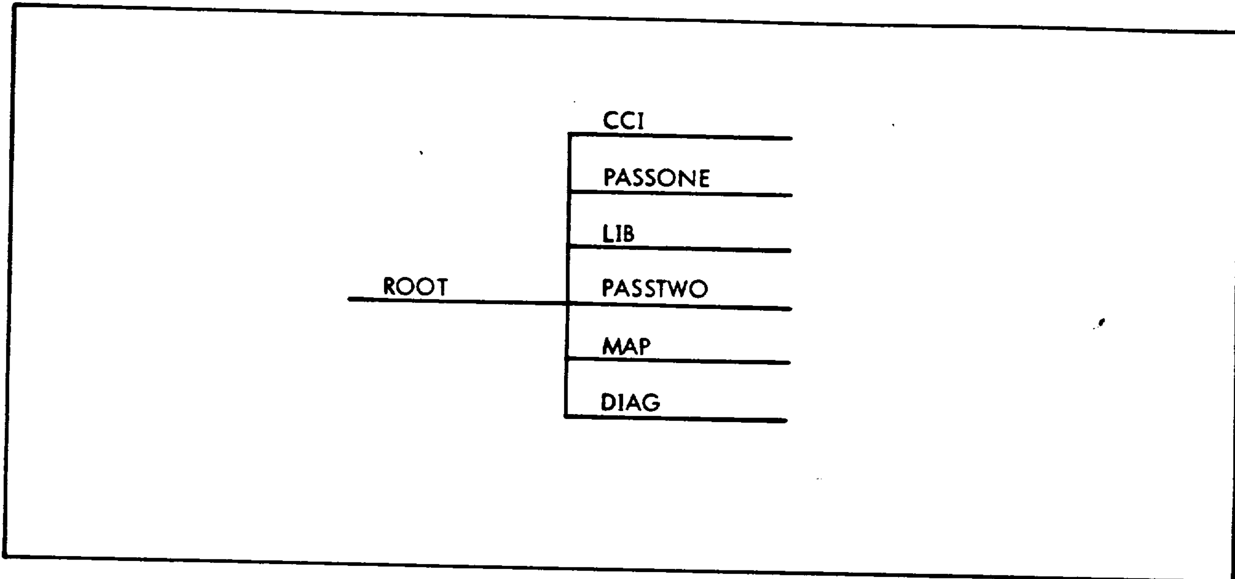



Figure 50. Overlay Structure of the Overlay Loader

Table 6. Overlay Loader Segment Functions

| Segment | Function |
|---|---|
| ROOT | Calls in the first segment (CCI) but thereafter, the segments call in other segments. ROOT is a collection of subroutines, tables, buffers, FPTs, DCBs, flags, pointers, variables, and temp storage cells. Root is resident at all times. |
| CCI | Reads and interprets all Loader control commands. |
| PASSONE | Makes the first pass over the Relocatable Object Modules, satisfies DEF/REF linkages between ROMs in the same path, links references to Public Library routines, and allocates the loaded program's control and dummy sections (e.g., assigns absolute core addresses). |
| LIB | Searches the library tables for routines to satisfy primary references left unsatisfied at segment end. |
| PASSTWO | Makes the second pass over the ROMs, creates absolute core images of segments, provides the necessary CP-R interface (PCB, Temp Stack, REFd DCBs, DCBTAB, INITTAB, and OVLOAD), and writes the absolute load module on the output file. |
| MAP | Outputs the requested information about the loaded program. |
| DIAG | Outputs all Loader diagnostic messages. |

## Overlay Loader Execution

The Root of the Overlay Loader is read into the background when the Job Control Processor (JCP) encounters an !OLOAD control command on the "C" Device. The JCP allocates six scratch files (X1, X2, X3, X4, X5, and X6) in the Background Temp area of the disk unless otherwise specified on a Monitor !ALLOBT command, and three blocking buffers unless otherwise specified on a Monitor !POOL command. The core layout of the Overlay Loader is illustrated in Figure 51.

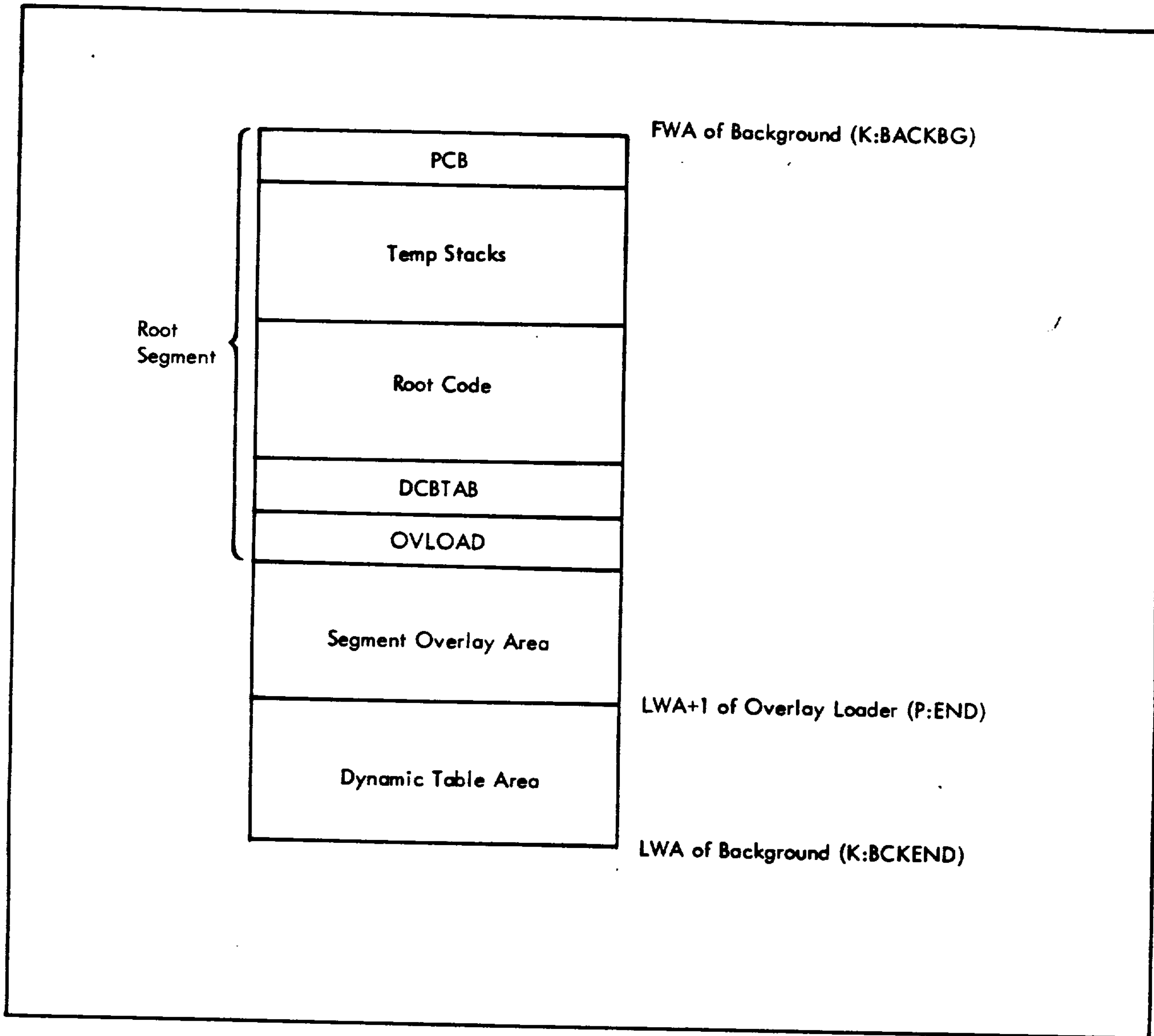



Figure 51. Overlay Loader Core Layout

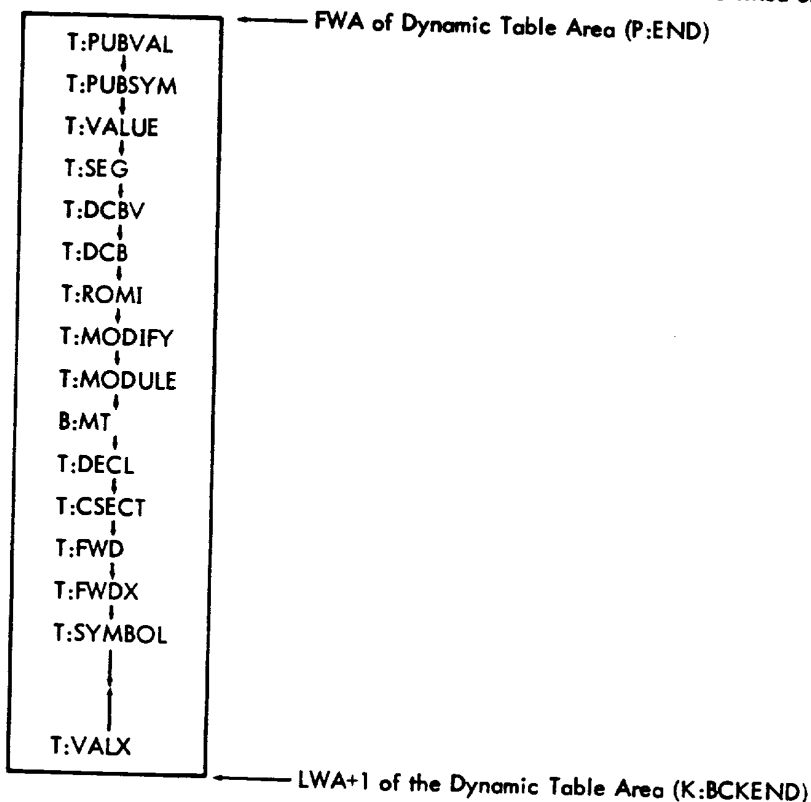
## Dynamic Table Area

The Dynamic Table Area is an area of core beginning at the LWA+1 of the Overlay Loader's code and extending to the beginning of the background blocking buffer pool. That is, the Loader uses the remaining core in background for a work area.

The Dynamic Table Area is divided into 16 table areas with boundaries that can change, subject to the length of the tables. The tables are built by CCI and PASSONE from information on the control commands and ROMs, and are therefore only dynamic until the beginning of PASSTWO, when the table areas are fixed. Since these tables are an essential part of the load process, it is important to understand the function of the tables.

## Dynamic Table Order

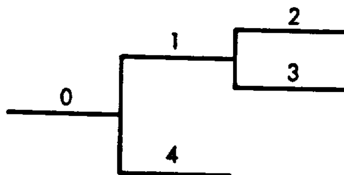
During the first pass over the object modules, the 16 table areas have a fixed order as follows:




For better reader comprehension, the table area descriptions given below are given in a logical order rather than the program listing sequence.

## T:SYMBOL and T:VALUE

The program's external table is a collection of DEFs, PREFs, SREFs, and DSECTs (excluding DCBs). The external table is divided into two parts: one containing the EBCDIC name of the external (T:SYMBOL), and the other containing the value (T:VALUE). Each table is divided into segment subtables that overlay each other in core in the same way that the segments themselves are overlayed. For example, the external tables of a program with the overlay structure




would exist in core (for both PASSONE and PASSTWO) as follows:

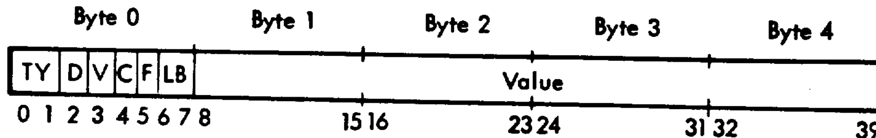
| For Root | For Seg 1 | For Seg 2 | For Seg 3 | For Seg 4 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 1 | 4 |
| | | 2 | 3 | |

Segments in different paths cannot communicate (i.e., the subtables of segments in different paths are never in core at the same time). A segment's T:SYMBOL and T:VALUE subtables are built by CCI and PASSONE and saved on a disk scratch file at path end (i.e., when the next segment starts a new path). However, only tables overlayed by the new segment at path end get written out. For example, at the end of path (0,1,2), segment 2 would be written out; at the end of path 0,1,3), segments 3 and 1 would get written out; and at the end of the program, segments 4 and 0 would get written out.

A segment's subtable consists of all DEFs in the segment, DSECTs not allocated in a previous segment of the path, and any REFs not satisfied by DEFs in a previous segment of the path. Since the DEF/REF links are all satisfied by PASSONE, T:SYMBOL is not used by PASSTWO.

T:VALUE ENTRY FORMATS

T:VALUE entries are numbered from 1 to n and have a fixed size of 5 bytes, with the format




where

TY is the entry type

TY = 00 DEF

TY = 01 DSECT

TY = 10 SREF

TY = 11 PREF

D is a flag specifying whether or not the external is defined/allocated/satisfied.

D = 1 external has been defined/allocated/satisfied.

D = 0 external is undefined/unallocated/unsatisfied.

V is a flag specifying the type of value (meaningful only if D = 1).

V = 1 value is the value of the external.

V = 0 value is the byte address of the expression defining or satisfying the external in T:VALX.

C is a constant (meaningful only if V = 1).

C = 1 value is a 32-bit constant.

C = 0 value is a positive or negative address with byte resolution.

F is a flag specifying whether the external is a duplicate or an original.

F = 1 external is a duplicate.

F = 0 external is an original.

LB specifies source of external.

LB = 00 external from input ROM or CC.

LB = 01 external from System Library.

LB = 10 external from User Library.

Value is initially set to zero; usage is dependent upon D, V, and C flags.

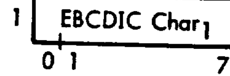
Since the T:VALUE entries are kept as small as possible, unused bit combinations are reserved to define the following two intermediate external types:

1. If TY = PREF, C = 0, and V = 1, the external is an "excluded pref" which means that the PREF will cause neither library loading nor linkage (including the Public Library). Instead, the PREF will be satisfied by a DEF in a segment further up the path.
2. If TY = DSECT, D = 1, and V = 0, the external was input from the :RES control command and is to be allocated at the end of the segment.

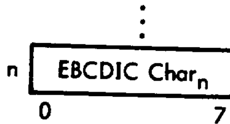
T:SYMBOL ENTRY FORMATS

T:SYMBOL is a byte table with variable sized entries that are numbered from 1 to n. There are three types of entries: EBCDIC, "continuation", and "pseudo". The EBCDIC entry contains the name of the external. The "continuation" entry contains the size of a DSECT and only follows a DSECT entry. The "pseudo" entry is a FWD or CSECT entry that has been added to T:SYMBOL because the entry was referenced in a T:VALX expression that could not be resolved at "module end". The entry formats are as follows:

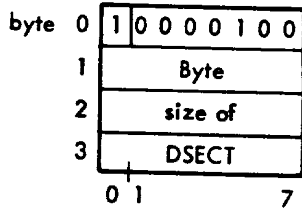
EBCDIC entry: byte




(Range = X'02' to X'40)




"Continuation" entry: byte




= X'84'

A byte size of -1 indicates that the entry is a reference to a DSECT allocated in a later segment.

"Pseudo" entry: byte 0 [0|0 0 0 0 0 0 1] (bits 0 1 ... 7) = X'01'

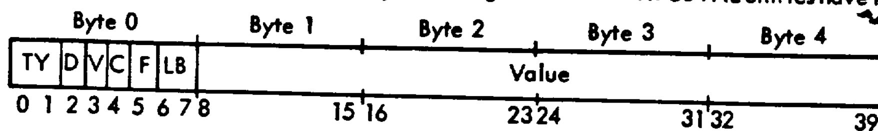
Note that the first byte contains the byte count of the entry (in bits 1-7).

## T:PUBVAL and T:PUBSYM

Each Public Library file has an external table of DEFs (there are no DSECTs or unsatisfied REFs in a Public Library) that is divided into two parts; VALUE and SYMBOL. T:PUBVAL contains the VALUE tables for each public library specified in the PUBLIB option of the IOLOAD control command, and T:PUBSYM contains the corresponding SYMBOL tables. Since the sizes of the table areas are fixed once T:PUBVAL and T:PUBSYM have been input, there are only 14 dynamic table areas.

T:PUBVAL ENTRY FORMATS

T:PUBVAL entries are numbered from 1 to n and have a fixed size of five bytes. Since the size of T:PUBVAL does not change, T:PUBSYM is located at the next doubleword boundary following T:PUBVAL. T:PUBVAL entries have the format




where

TY = 00 = DEF

D = 1 the DEF has been defined.

V = 1 value is the value of the DEF.

C = 1 value is a 32-bit constant.

C = 0 value is a positive or negative address with byte resolution.

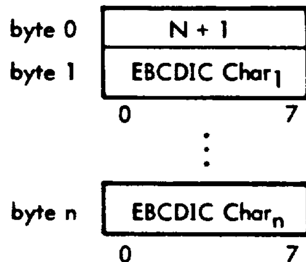
F = 0 not a duplicate DEF.

LB = 11 PUBLIB

Note that the T:VALUE and T:PUBVAL entries have the same formats even though the T:PUBVAL entries are a subset of the T:VALUE format.

T:PUBSYM ENTRY FORMATS

T:PUBSYM is a byte table with variable sized entries that are numbered from 1 to n. Since the size of T:PUBSYM does not change, the table following is located at the next doubleword boundary after T:PUBSYM. T:PUBSYM entries have the format



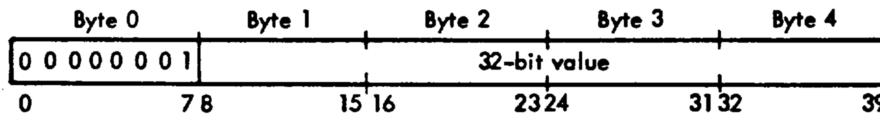

## T:VALX

External definitions are defined with expressions. If the expression can be resolved, its value is stored in the DEFs T:VALUE entry. If the expression cannot be resolved, it is saved in T:VALX and the byte address of the expression is stored in the DEFs T:VALUE entry.

Once an expression is resolved, its entry is zeroed out. The T:VALX entries cannot be packed to regain space, since the T:VALUE entries contain address pointers, however, empty entries are reused where possible.

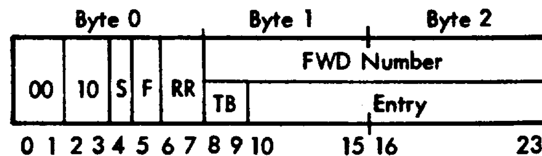
Expressions have a variable size and are made up of expression bytes, combined in any order. The formats for the T:VALX expression bytes (slightly different than the object language) are

Add Constant (X'01')




This item causes the specified four-byte constant to be added to the Loader's expression accumulator. Negative constants are represented in two's complement form:

Add/Subt Value (X'2N')




where

S = 1 subtract value.

S = 0 add value.

F = 1 add/subtract value of T:FWD entry where the FWD number is in bytes 1 and 2.

F = 0 add/subtract value of TABLE entry where

TB = 00 Entry points to T:DCB.

TB = 01 Entry points to T:VALUE/T:SYMBOL.

TB = 10 Entry points to T:CSECT.

TB = 11 Entry points to T:PUBVAL/T:PUBSYM.

RR = 00 byte address resolution.

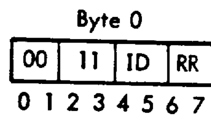
RR = 01 halfword address resolution.

RR = 10 word address resolution.

RR = 11 doubleword address resolution.

This item causes the value of the FWD or TABLE entry to be converted to the specified address resolution (only if the value is an address) and added to the Loader's expression accumulator. Note that expressions involving T:FWD and T:CSECT entries point to the current ROM's FWD and CSECT tables. If these expressions are not resolved at module end, the Loader creates dummy T:SYMBOL and T:VALUE entries from the FWD or CSECT entry and changes the pointer in the expression to point to the dummy entry in T:VALUE. However, unresolved expressions rarely happen.

Address Resolution (X'3N')




where

ID = 00 changes the partially resolved expression (if an address) to the specified resolution.

ID = 01 identifies the expression as a positive absolute address with the specified resolution (add absolute section).

ID = 10 identifies the expression as a negative absolute address with the specified resolution (subtract absolute section).

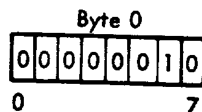
RR = 00 byte address resolution.

RR = 01 halfword address resolution.

RR = 10 word address resolution.

RR = 11 doubleword address resolution.

Expression End (X'02')




This item identifies the end of an expression (the value of which is contained in the Loader's expression accumulator).

**T:DCB**

T:DCB contains the DEFs and REFs that are recognized as either system (M:) or user (F:) DCBs. DCBs declared as external definitions must exist in the Root segment. The Loader allocates space in part two of the Root for DCBs

that are declared external references, and supplies default copies of system DCBs. T:DCB is resident at all times. Entries have a fixed size of three words and have the format

| Word | 0–1 | 2 | 3 | 4 | 5 | 6–7 | 8–12 | 13–15 | 16–23 | 24–31 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | TY | D | V | C | F | LB | (shaded) | Byte Address | | |
| 1 | E1 | | | | | | | E2 | E3 | E4 |
| 2 | E5 | | | | | | | E6 | E7 | E8 |

0 1 2 3 4 5 6 7 8 12 13 15 16 23 24 31

where

Word 0

TY = 00 DEF (coded in the Root by the user).

TY = 11 PREF (allocated in Root part 2 by Loader).

D = 1 defined or allocated.

D = 0 undefined/unallocated.

V = 1 address is the byte value of the DCB, only meaningful if D = 1.

V = 0 address points to an expression in T:VALX, only meaningful if D = 1.

C = 1 the DCB was defined with a value that is either a constant or an illegal address (i.e., negative or mixed resolution), only meaningful if V = 1.

C = 0 the value of the DCB is an address, only meaningful if V = 1.

F = 0 DCB cannot be a duplicate (duplicates are put in T:SYMBOL/T:VALUE).

LB = 00 the DCB was input from a nonlibrary ROM.

LB = 01 the DCB was input from the System Library.

LB = 10 the DCB was input from the User Library.

Word 1,2

E1 - E8 is the EBCDIC name of the DCB, padded with blanks if necessary.

## T:SEG

T:SEG contains information about the program's segments and is resident at all times. One entry is allocated per segment. Entries have a fixed size of ten words and have the format

| Word | Bits 0–15 | Bits 16–31 |
|---|---|---|
| 0 | Segment Ident | Link Ident |
| 1 | Gran no. of T:VALUE (I) on X4 | Gran no. of T:MODIFY/ T:MODULE on X3 |
| 2 | Gran no. of T:SYMBOL (I) on X5 | Gran no. of core image on Program File |
| 3 | BD of T:VALUE (I) in T:VALUE | Byte length of T:VALUE (I) |
| 4 | BD of T:SYMBOL (I) in T:SYMBOL | Byte length of T:SYMBOL (I) |
| 5 | Byte length of T:MODIFY | Byte length of T:MODULE |
| 6 | DW EXLOC of SEG | DW length of SEG |
| 7 | R, L, W, F, I, M, S, P, E A (bits 0–14) | Entry Address |
| 8 | Byte Length of Library Routines in SEG | |
| 9 | Byte length of load-module image of segment | |

0 1 2 3 4 5 6 7 8 9 12 13 14 15 16 31

where

Gran no. the granule number in the disk file where the table begins. If the disk file overflows, Gran No. will equal X'FFFF'. Granules are numbered from 0 to n.

(I) segment's subtable.

BD byte displacement.

EXLOC execution location.

DW doubleword.

R = 1 error severity level set on at least one ROM in the segment.

R = 0 error severity level reset on every ROM in the segment.

L = 1 load error (duplicate DEFs, unsatisfied REFs, etc.).

L = 0 no loading errors in SEG.

W = 1 T:VALUE (I) and T:SYMBOL (I) output on X4, X5.

W = 0 T:VALUE (I) and T:SYMBOL (I) not output on X4, X5.

F = 1 segment is fixed in real memory (FIX option).

F = 0 segment may be mapped onto any available real memory.

I = 1 segment is to be initially loaded with the root (ILOAD option).

I = 0 segment will be loaded only on explicit request.

M = 00 segment is any-access.

M = 01 segment is read-and-execute.

M = 10 segment is read-only.

M = 11 segment is no-access.

S = 00 segment is non-sharable.

S = 01 segment is job-level sharable.

S = 10 segment is system-level sharable.

S = 11 unused.

P = 1 segment must be preloaded sharable (PRELOAD option).

P = 0 segment may be loaded from the load module being built.

EA = 00 value in bits 15-31 (if nonzero) is last entry address (in words) encountered on non-Lib ROM.

EA = 01 unused.

EA = 10 SEG's entry address input from CC and value in bits 15-31 is the entry address (in words).

EA = 11 SEG's entry address input from CC and value in bits 15-31 is the entry number of the T:SYMBOL/T:VALUE DEF specified on the CC.

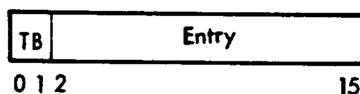
## B:MT

There are four tables associated with each ROM loaded (including library ROMs): T:DECL, T:CSECT, T:FWD, and T:FDX. The size of these tables can be extremely large or small, depending upon which processor produced the ROM and the content of the program. To conserve time and space, these tables are packed into the Module Tables buffer (B:MT) at module end, and output to the X2 Temp File on the disk only when either the buffer is full or at segment end. The size allocated for B:MT is dependent upon the size of the Dynamic Tables area and is made a multiple of the sector size of the X2 disk file.

## T:DECL

DEFs, PREFs, SREFs, DSECTs, and CSECTs are referenced in the object language by declaration number. Therefore, associated with each ROM is a table of declarations whose entries point to DEF, REF, DSECT, and CSECT entries in other tables.

According to the object language convention, entry zero points to the standard control section declaration. Entries are numbered from 0 to n; have a fixed size of two bytes; and have the format




where

TB = 00 Entry points to T:DCB.

TB = 01 Entry points to T:SYMBOL/T:VALUE.

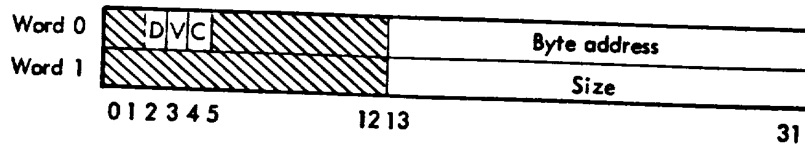
TB = 10 Entry points to T:CSECT (associated with current ROM).

TB = 11 Entry points to T:PUBSYM/T:PUBVAL.

Entry Table entry number. The range is 1 through 16,383.

## T:CSECT

Associated with each ROM is a table of standard and nonstandard control sections. A nonstandard control section is allocated by the Loader when the declaration is encountered. The standard control section is allocated when the first reference to declaration 0 is encountered in an expression defining the origin load item. T:CSECT entries are numbered from 1 to n; have a fixed size of two words; and have the format




where

Word 0

D = 1 allocated.

V = 1 value.

C = 0 address.

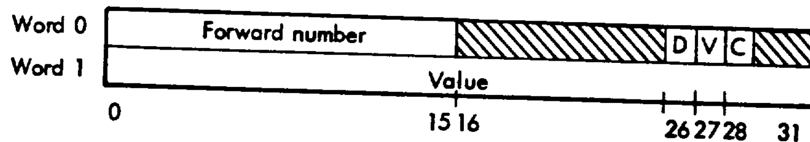
Byte address first byte address of the control section.

Word 1

Size Number of bytes in the control section.

## T:FWD

Associated with each ROM is a table of forward reference definitions (forwards). Each forward is identified by a random two-byte reference number. Thus, when a forward is referenced in an expression, the T:FWD table for that ROM must be searched for a matching number. T:FWD entries have a fixed size of two words with the format




where

D = 1 defined.

V = 1 value is the value of the resolved expression.

V = 0 value is a byte displacement pointer to the expression in T:FWDX.

C = 1 value is a constant (only meaningful if V = 1).

C = 0 value is a positive or negative address with byte resolution (only meaningful if V = 1).

## T:FWDX

Forwards are defined with expressions and are of two types: the first is defined with an expression that can be resolved by module end; the second type is defined with an expression that involves an external DEF, REF, or DSECT (many of these cannot be resolved at module end). Associated with each ROM is a table containing all unresolved expressions defining FWDs. When a T:FWDX expression is resolved, its entry is zeroed out and the space reused, if possible. T:FWDX entries have the same format as T:VALX entries.

## T:MODULE

Each segment has a T:MODULE table. T:MODULE contains information about a segment's Relocatable Object Modules (ROMs). One entry is allocated per ROM. Entries have a fixed size of five words and have the format

| | 0 | 1–12 | 13 | 14–15 | 16 | 17 | 18–31 |
|---|---|---|---|---|---|---|---|
| Word 0 | V | Entry no. | G | LB | (unused) | PACK | Record Displacement |
| 1 | Gran no. of B:MT on X2, or BD of T:DECL (J) in B:MT | | | | Byte length of T:DECL (J) | | |
| 2 | BD of T:CSECT (J) in B:MT | | | | Byte length of T:CSECT (J) | | |
| 3 | BD of T:FWD (J) in B:MT | | | | Byte length of T:FWD (J) | | |
| 4 | BD of T:FWDX (J) in B:MT | | | | Byte length of T:FWDX (J) | | |

where

V = 1 Entry no. in bits 1-12 points to T:DCBV.

V = 0 Entry no. in bits 1-12 points to T:DCBF, or is zero.

Entry no. the entry number of the DCB (in either T:DCBV or T:DCBF) that points to the disk file where the ROM is located, or zero, to load another ROM from the previous source.

G = 1 T:DECL (J) begins at byte zero in B:MT and HW0 (halfword zero) in word 1 contains the granule no. of B:MT on X2. If the Granule no. equals X'FFFF', X2 has overflowed and B:MT did not get saved on the disk.

G = 0 T:DECL (J) is located in B:MT at the byte displacement specified in HW0 of word 1.

LB = 00 not Library ROM.

LB = 01 ROM from System Library (SP area of disk).

LB = 10 ROM from User Library (FP area of disk).

Record displacement in the MODULE file (only meaningful for library ROMs).

Pack = 1 if the PACK input option was associated with this ROM.

## T:ROMI

T:ROMI contains the information necessary for PASSONE to load a segment's ROMs. T:ROMI is built by CCI from the input options specified on the segment's :ROOT, :SEG, or :PUBLIB control command, or by :LIB to point to the library routines required for the segment. At the beginning of PASSTWO, the area size for T:ROMI is set to zero. There are three types of T:ROMI entries, as illustrated below, and entries have a fixed size of one word.

Entry for ROMs input from user files (built by CCI):

| 0–15 | 16 | 17 | 18 | 19 | 20–31 |
|---|---|---|---|---|---|
| NROM | (unused) | PACK | (unused) | V | Entry No. |

where

NROM is the number of ROMs to input or contains -5, which means to input until IEOD is encountered. This halfword is used as a decreasing counter by PASSONE and eventually equals zero.

Bits 16-23 always equal zero to specify entry type.

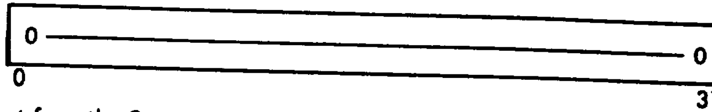
V = 1 Entry no. in bits 20–31 points to T:DCBV.

V = 0 Entry no. in bits 20–31 points to T:DCBF.

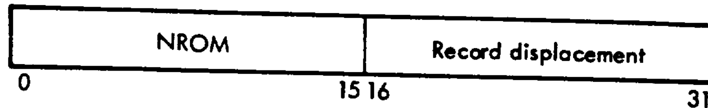
Entry no. is the entry number of the DCB in T:DCBF or assignment in T:DCBV that points to the medium where the ROM is located.

PACK = 1 if the PACK input option was associated with this ROM.

Entry for the NONE option (built by CCI):




Entry for ROMs input from the System or User Library (built by LIB):




where

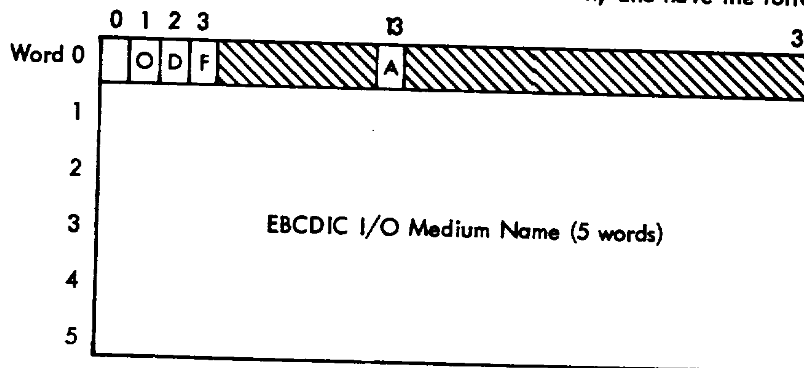
NROM is described above.

Record displacement is the record displacement of the ROM in the MODULE file of the area specified by FL:LBLD.

Library ROM entries are distinguished from the other two entry types by the Loader flag FL:LBLD. The flag is always reset when the other entry types are in T:ROMI.

## T:DCBV

T:DCBV is a table of DCB assignments for the various ROM media specified (other than GO) on the input options of the :ROOT and :SEG, or :PUBLIB control commands. One entry is created for each input option specified. T:DCBV is resident at all times. T:DCBV entries are numbered from 1 to n, and have the following format:




where

O = 1 if word 1 contains a right-aligned operational label name. The remaining flags are ignored.

O = 0, D = 1 if words 1 and 2 contain a left-aligned device name. The remaining flags are ignored.

O = D = 0, F = 1 if word 1 contains a right-aligned disk area name or blanks, and words 2 and 3 contain a left-aligned file name.

O = D = 0, F = 1, A = 1 if words 4 and 5 contain a left-aligned disk file account name.

## T:MODIFY

Each segment's :MODIFY commands are translated into object language load items and stored in the segment's T:MODIFY table, and each :MODIFY command is translated into a T:MODULE entry. Entries begin with an "origin" load item and are terminated by either the next "origin" load item or a "module end" load item. Entries are made up of the load items described below and expressions in the T:VALX/T:FWDX format:

Origin (X'04')

This one-byte item sets the load-location counter to the value designated by the expression (in T:VALX format) immediately following the origin control byte. The value of the expression equals the location specified on the :MODIFY command.

Load Absolute (X'44')

This one-byte item causes the next four bytes to be loaded absolutely and the load-location counter advanced appropriately.

Define Field (X'07')
(X'FF')
(field length)

This three-byte item defines an expression value to be added to the address field of the previously loaded four-byte word. The expression is in T:VALX format and immediately follows the 'field length' byte.

Load Expression (X'60')

This one-byte item causes an expression value to be loaded absolutely and the load-location counter advanced appropriately. The expression to be loaded is in T:VALX format and immediately follows the 'load expression' control byte.

Module End (X'0E')

This one-byte item terminates the load items in T:MODIFY.

## Use of the Dynamic Table Area During LIB

During the library search, LIB temporarily reorganizes the Dynamic Table area by packing the 16 tables together at the top of the area. LIB uses the remaining space for its tables. The core layout of these tables and their formats are illustrated in Figure 52.

## T:LDEF

T:LDEF is located in the Dynamic Table area only when the LIB segment is executing and is used by LIB to satisfy REFs to library routines. Initially, T:LDEF contains the following items:

1. All unsatisfied REFs from the current segment's T:VALUE subtable.
2. All excluded PREFs from the current segment's T:VALUE subtable.

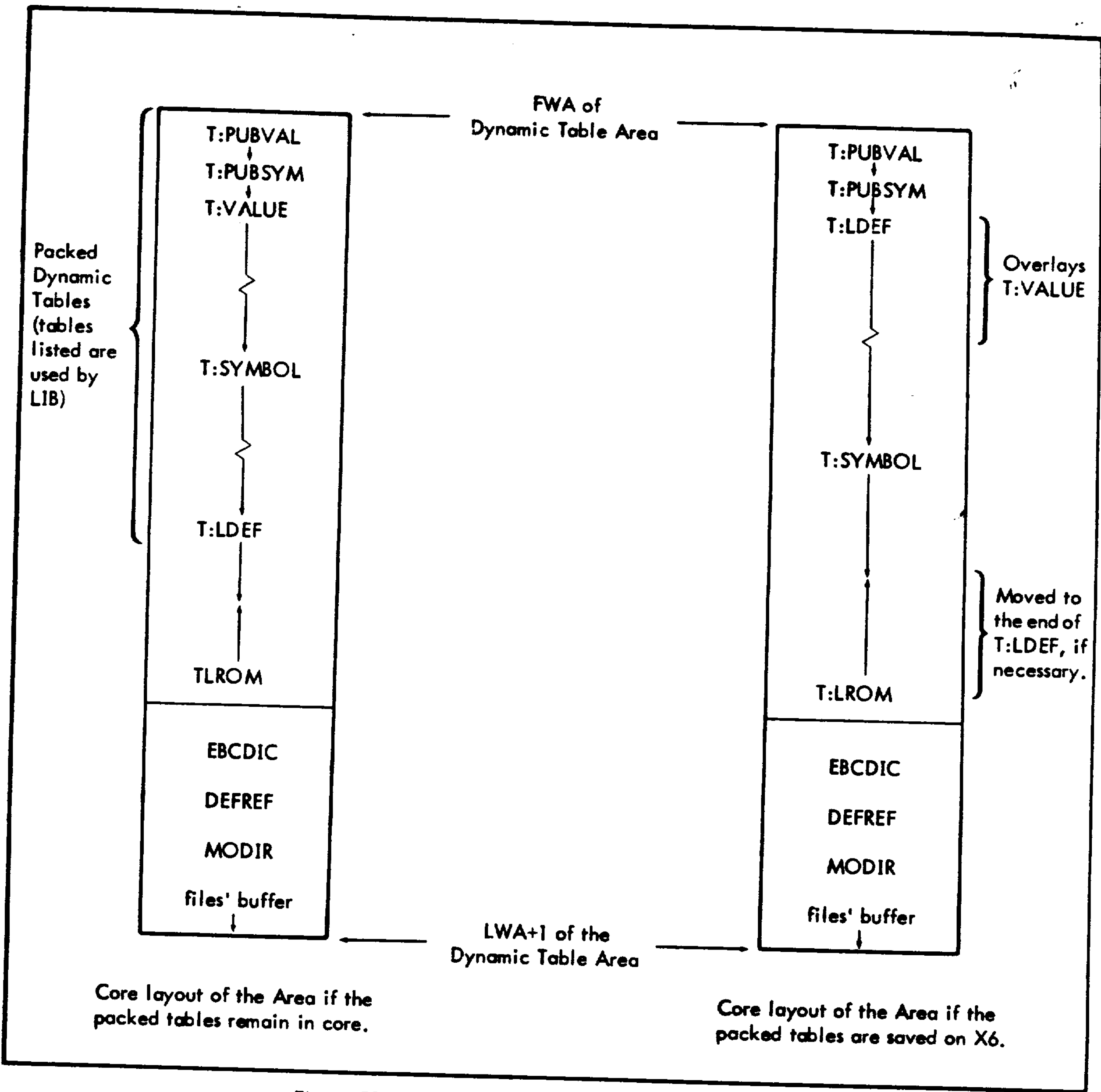


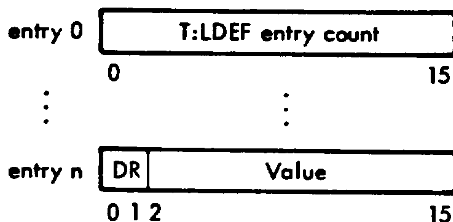

Figure 52. LIB Reorganization of Dynamic Table Area

3. All DEFs and DSECTs in the path T:VALUE table that are from the same library as the one being searched.

4. All Public Library (T:PUBVAL) DEFs.

The Library DEFs are included so that library routines loaded in previous segments of the Public Library will not be duplicated. The excluded PREFs (that inhibit library loading) are treated as DEFs. Since library routines may themselves reference other library routines, the set of DEFs and REFs associated with a library routine are included in T:LDEF if, and only if, at least one of the DEFs satisfies a REF in T:LDEF. When a REF is satisfied it is changed to a DEF. Eventually, T:LDEF contains library DEFs, any REFs that cannot be satisfied in the Library, and the excluded PREFs.

T:LDEF has a variable number of entries with the count kept in entry 0. Entries have a fixed size of two bytes with the format




where

DR = 00 null entry.

DR = 01 DEF or excluded PREF.

DR = 10 unsatisfied PREF.

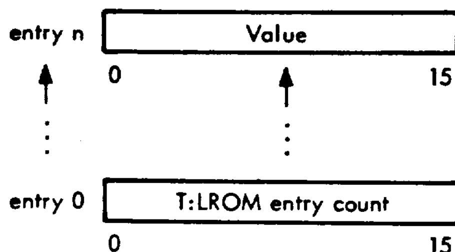
DR = 11 DSECT.

Value entry number in the T:SYMBOL, that is later changed to the corresponding entry's byte offset in the EBCDIC file.

## T:LROM

T:LROM is located in the Dynamic Table area only when the LIB segment is executing and contains pointers to library routines whose DEFs have satisfied REFs in T:LDEF. That is, T:LROM points to the library routines that are to be loaded along with the segment.

T:LROM entries initially point to a library ROM's entry in the MODIR file and then get changed to point to the corresponding ROM's location in the MODULE file. T:LROM has a variable number of entries, with the count kept in entry 0. T:LROM is built backwards but has forward entries. Entries have a fixed size of two bytes with the format




where

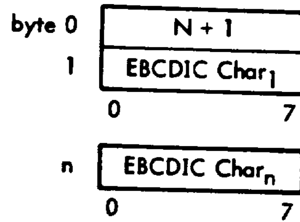
value halfword offset of the library ROM's entry in the MODIR file, which is later changed to the starting record number of the ROM in the MODULE file.

## MODULE File

The MODULE file is a blocked sequential file, with 120 bytes per record, that contains the Library's ROMs.

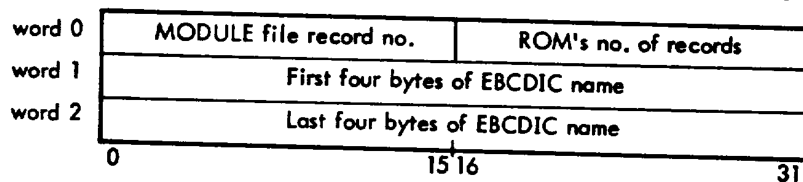
## EBCDIC File

The EBCDIC file is an unblocked sequential file consisting of one variable length record. The EBCDIC file contains the unique EBCDIC names of all DEFs and REFs declared in the ROMs in the MODULE file. Entries have a variable number of bytes with the format



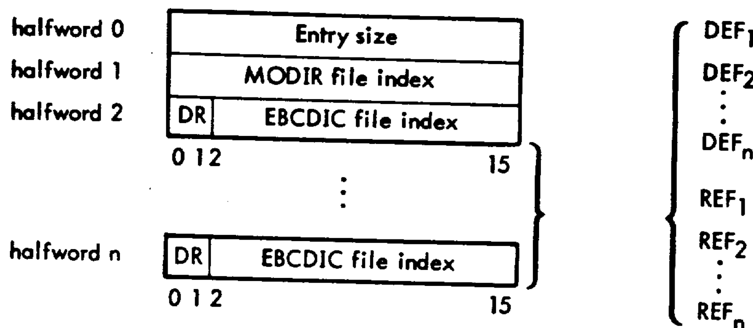

## MODIR File

The MODIR file is an unblocked sequential file consisting of one variable length record. Each MODIR file entry corresponds to a ROM on the MODULE file and contains the name of the ROM, its location on the MODULE file, and the number of records in the ROM. Entries have a fixed size of three words with the format




## DEFREF File

The DEFREF file is an unblocked sequential file consisting of one variable length record. Each entry in the DEFREF file corresponds to a ROM in the MODULE file and contains all the external DEFs and REFs declared in the ROM, plus a pointer to the ROM's entry in the MODIR file. Entries have a variable number of halfwords with the format




where

Entry size  number of halfwords in the entry (including itself).

MODIR file index  relative halfword of the ROM's corresponding entry in the MODIR file. X'FFFF' means that the entry has been deleted.

DR = 00  not used.

DR = 01  DEF.

DR = 10  PREF.

DR = 11  DSECT.

EBCDIC file index  relative byte of the external name entry in the EBCDIC file.

## Use of Dynamic Table Area During PASSTWO

PASSTWO reorganizes the Dynamic Table area by moving the resident tables T:SEG, T:DCBV, and T:DCB to the end of T:PUBVAL. PASSTWO uses the remaining space to read in the necessary tables built during PASSONE to build its own tables and to create the core image of the segment. The core layout of these tables and their format is illustrated in Figure 53.

## T:GRAN

Since the Work area has a finite size that varies according to the size of B:MT, it may not be large enough to contain a segment's total core image at all times. Therefore, before a segment is created, its core image length is divided into granule size partitions, where the granule size equals the sector size of the program file. T:GRAN

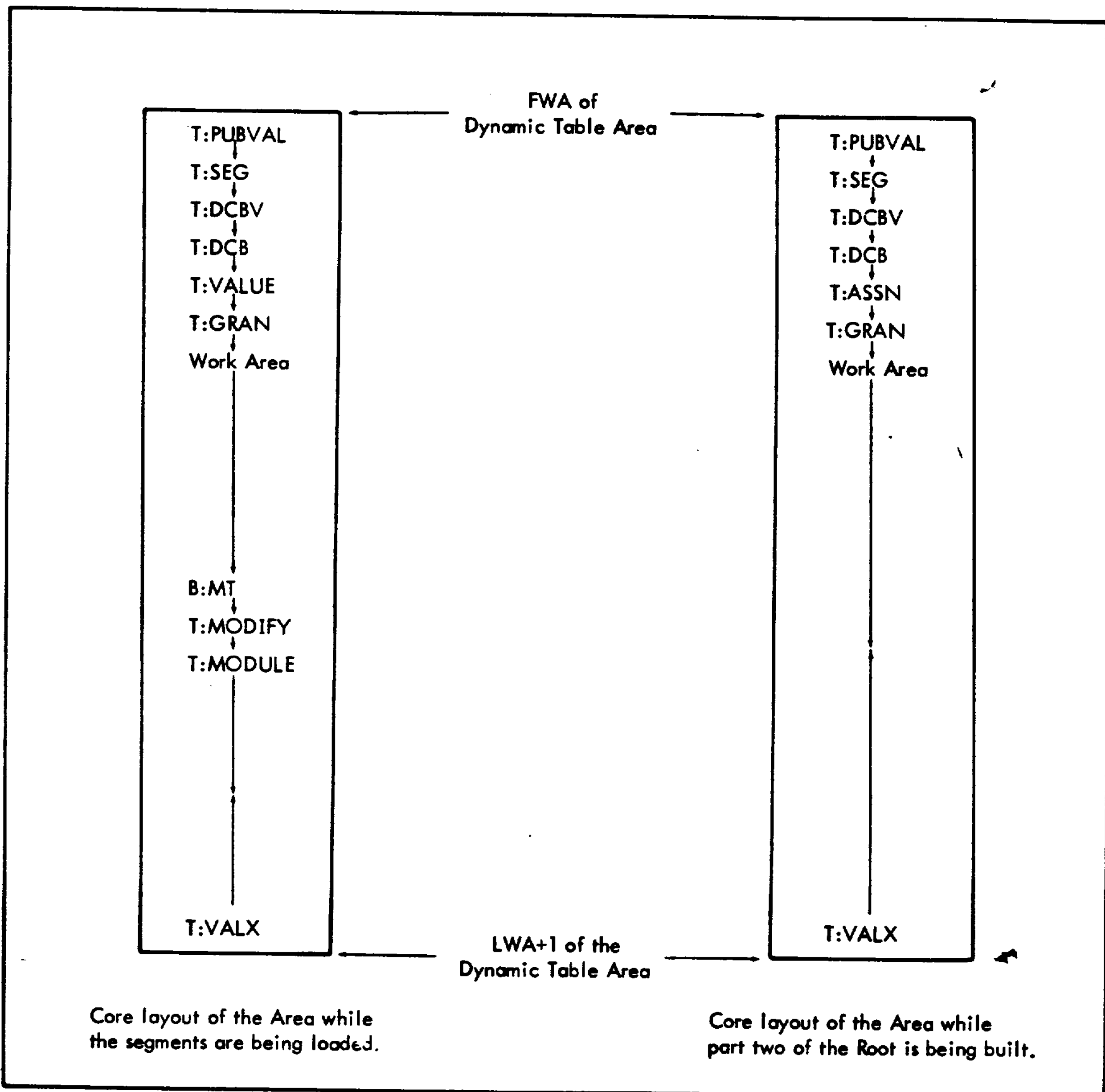
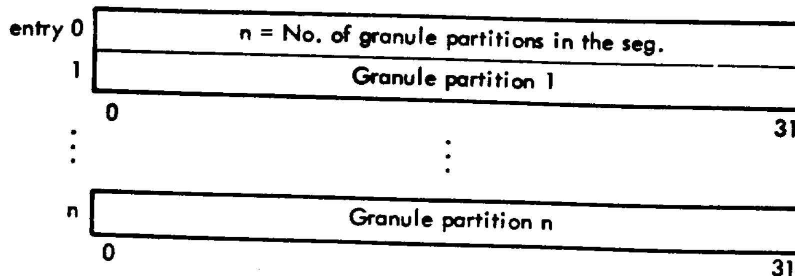


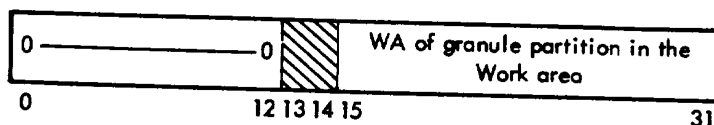

Figure 53. PASSTWO Reorganization of Dynamic Table Area

entries point to the location of a segment's partition (if created) either in core or on the program file. T:GRAN has the following format:

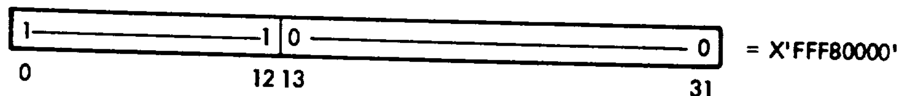



T:GRAN entries have a fixed size of one word with three different formats.

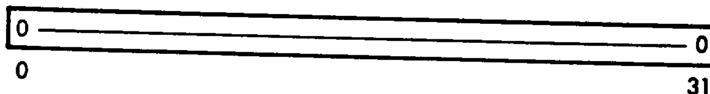
If the granule partition exists in the Work Area:




If the granule partition exists on its corresponding granule in the Program File:

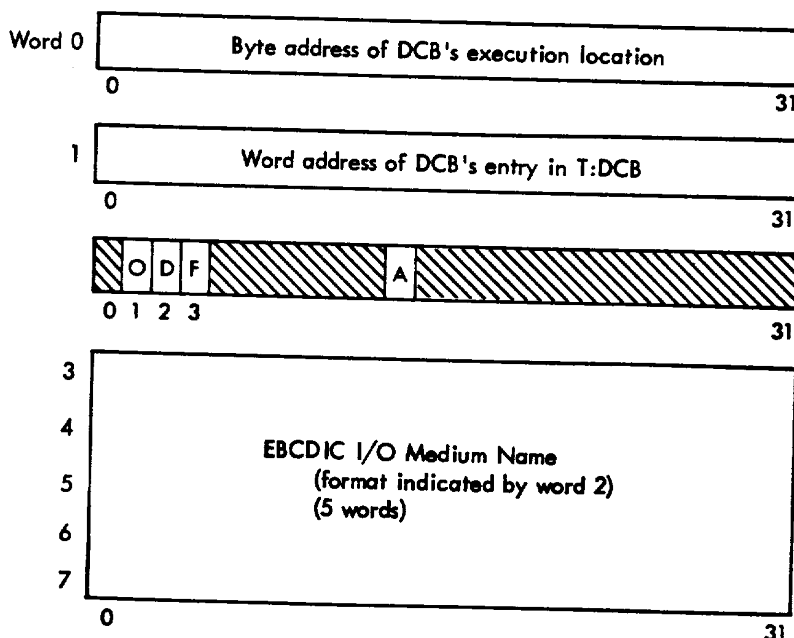


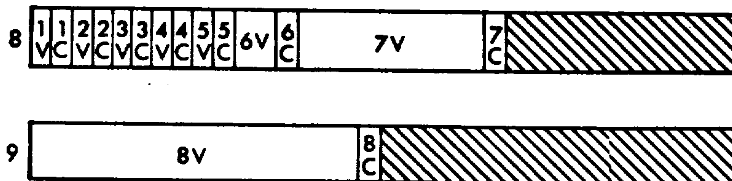

If the granule partition has not been allocated; and data has not yet been loaded into that area of the segment:




## T:ASSN

T:ASSN contains the information necessary to reassign DCBs as specified on :ASSIGN commands. T:ASSN is located in the Dynamic Table area during PASSTWO (after all the segments have been loaded) and is built by CCI. Each :ASSIGN command is translated into a T:ASSN entry. Entries have a fixed size of ten words with the format






where

O = 1 if word 1 contains a right-aligned operational label name. The remaining flags are ignored.

O = 0, D = 1 if words 1 and 2 contain a left-aligned device name. The remaining flags are ignored.

O = D = 0, F = 1 if word 1 contains a right-aligned disk area name or blanks, and words 2 and 3 contain a left-aligned file name.

O = D = 0, F = 1, A = 1 if words 4 and 5 contain a left-aligned disk file account name.

kC: if reset kV is unused; if set, kV is to be inserted.

1V: value for MOD field.

2V: value for ASC field.

3V: value for DRC field.

4V: value for D/P field.

5V: value for VFC field.

6V: value for BTD field.

7V: value for NRT field.

8V: value for RSZ field.

## MAP Use of Dynamic Table Area

MAP moves the resident tables T:SEG and T:DCB to the top of the area, and uses the remaining space to read in and reference the tables necessary for the MAP output. MAP does not build any tables. The core layout of the table referenced by MAP is illustrated in Figure 54.

## DIAG Use of Dynamic Table Area

DIAG only uses the Dynamic Table area to reference TSEG and T:MODULE.

ROOT TABL

Two tables in the Root, T:PL and T:DCBF, have a fixed size and are referenced by other tables. Their format and use is given below. The usage and format of other tables in the Root are well documented in the Overlay Loader's listing and are not detailed in this manual.

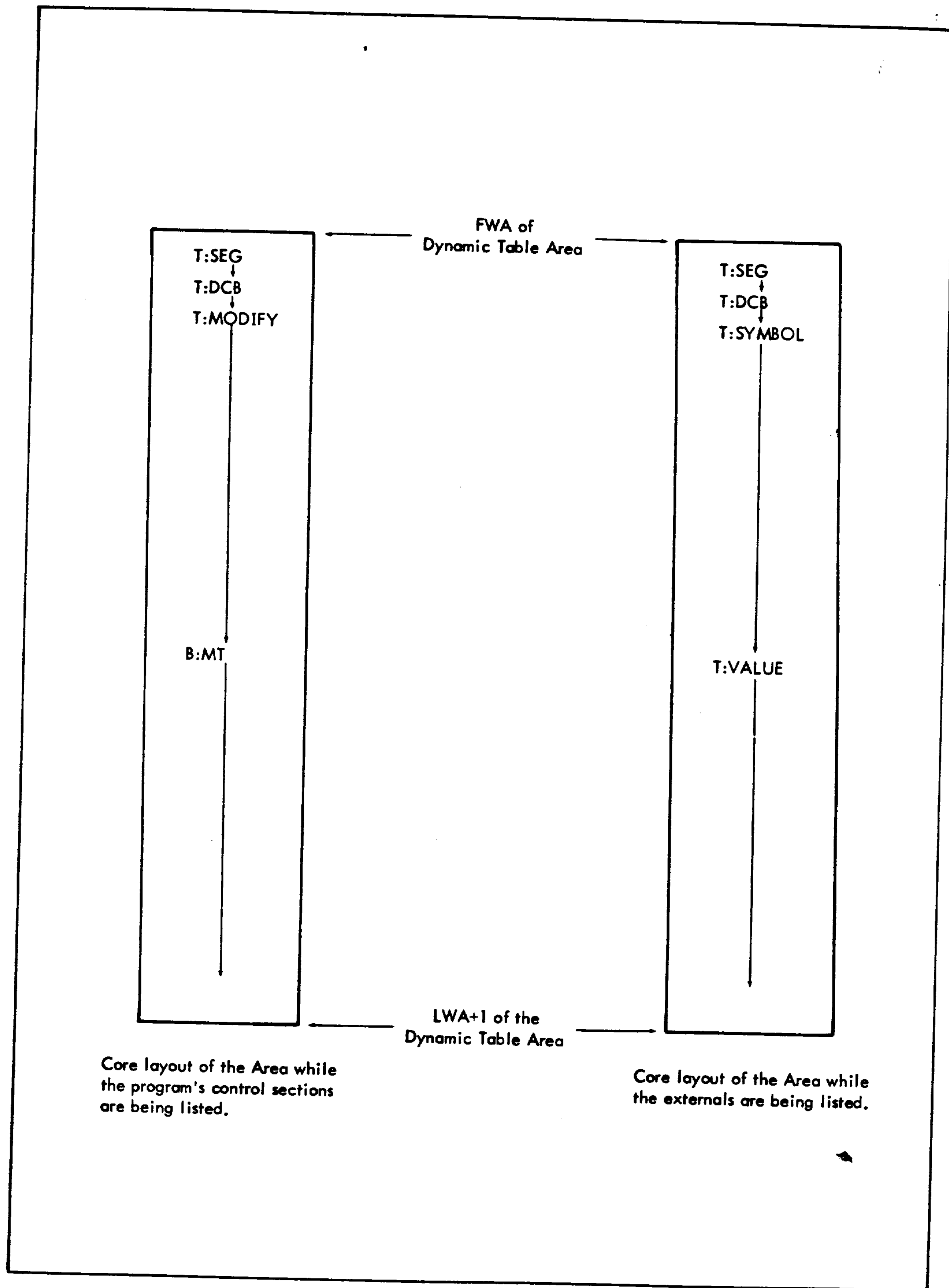
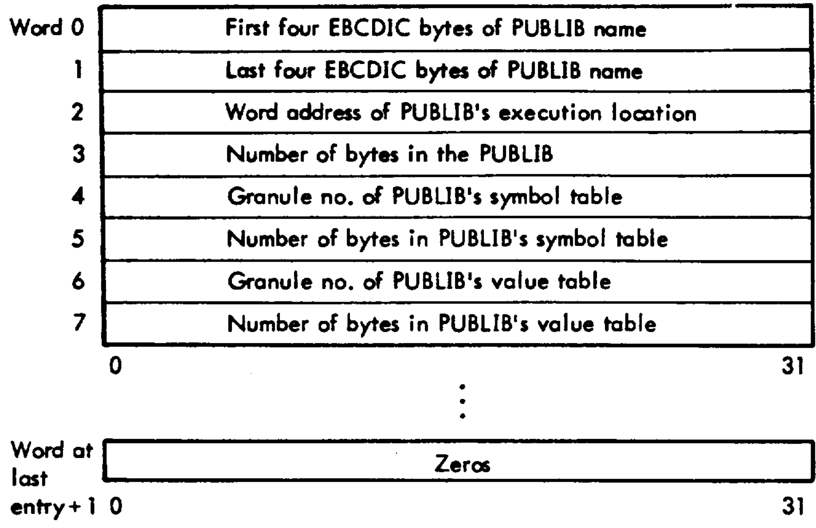



Figure 54. MAP Table Reference

## T:PL

T:PL contains the information necessary to create T:PUBSYM and T:PUBVAL and to load the Public Libraries specified on the !OLOAD control command. T:PL exists in the Root and has a maximum of three entries. Table end is indicated by a word of zeros. Entries have a fixed size of eight words with the format



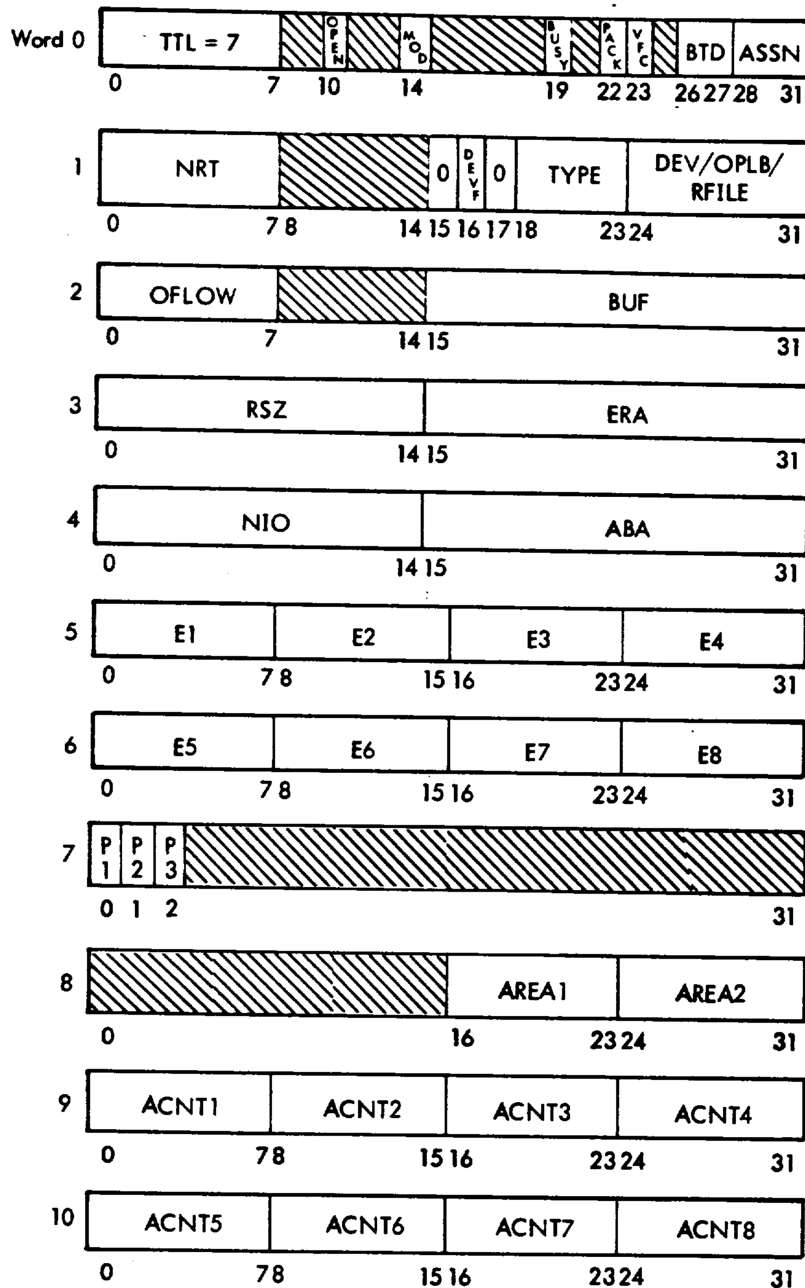

## T:DCBF

T:DCBF contains the set of fixed DCBs that are required by the Loader. Each entry contains one DCB. T:DCBF has a fixed number of entries and exists in the Root. T:DCBF entries are numbered from 1 to 18, and have the fixed order given in Table 7.

Table 7. T:DCBF Entries

| Entry | Mnemonic | Pointer To |
|---|---|---|
| 1 | F:PUBL | Files specified in the PUBLIB option of !OLOAD. |
| 2 | F:DEVICE | Devices specified in the DEVICE and OPLB input options. |
| 3 | M:GO | GO file in the Background Temp area. |
| 4 | M:OV | Either OV or the file specified in the FILE option of !OLOAD. |
| 5 | M:X1 | X1 in the Background Temp area. |
| 6 | M:X2 | X2 in the Background Temp area. |
| 7 | M:X3 | X3 in the Background Temp area. |
| 8 | M:X4 | X4 in the Background Temp area. |
| 9 | M:X5 | X5 in the Background Temp area. |
| 10 | M:X6 | X6 in the Background Temp area. |
| 11 | F:MODIR | MODIR file in either the SP or FP area. |
| 12 | F:EBCDIC | EBCDIC file in either the SP or FP area. |
| 13 | F:DEFREF | DEFREF file in either the SP or FP area. |
| 14 | F:MODULE | MODULE file in either the SP or FP area. |
| 15 | M:C | C operational label. |
| 16 | M:LL | LL operational label. |
| 17 | M:OC | OC operational label. |
| 18 | M:LO | LO operational label. |

All T:DCBF entries have the standard 11-word DCB format, with two exceptions: OFLOW and NIO, that are used only for the M:OV, M:X1, M:X2, M:X3, M:X4, M:X5, and M:X6 DCBs. The 11-word DCB format is




where

OFLOW = 0 EOT not encountered.

OFLOW = 1 EOT encountered.

NIO number of records (for X1) or granules required.

## Scratch Files

The six scratch files in the Background Temp area of the disk are used by the Loader as temporary storage and are written during the first pass over the object modules. The number of granules required by each scratch file is calculated (whether the file overflows or not) and saved in the DCB assigned to the file. If any of these files overflows (e.g., if the EOT is encountered during a Write operation), the Loader continues PASSONE, skips PASSTWO, then calls the MAP to communicate the number of granules required for each scratch file to the user. The Loader's use of these files is defined in Table 8.

Table 8. Background Scratch Files

| File Name | Loader Use |
|---|---|
| X1 | A sequential file with blocked record format. Record size equals 120 bytes; granule size equals 256 words. ROMs input from non-RAD devices are copied onto X1. |
| X2 | A direct access file with the granule size set equal to the sector size. The module's tables (T:DECL, T:CSECT, T:FWD, and T:WDX) are output on X2 when either B:MT is full or at segment end. |
| X3 | A direct access file with the granule size set equal to the sector size. A segment's T:MODIFY and T:MODULE tables are packed together at segment end and output on X3. |
| X4 | A direct access file with the granule size set equal to the sector size. A segment's T:VALUE subtable is output on X4 when the end of a path is encountered and the segment is being overlayed by another segment. |
| X5 | A direct access file with the granule size set equal to the sector size. A segment's T:SYMBOL subtable is output on X5 when the end of a path is encountered and the segment is being overlayed by another segment. |
| X6 | A direct access file with the granule size set equal to the sector size. The LIB overlay packs the 16 Dynamic Tables at the top of the Dynamic Table area and outputs the "pack" on X6 only if the remaining area will not contain the tables required for the library search. |

### Program File Format

The format for the Program File is illustrated in Figure 55.

The foreground/background program-header format is described in the "CP-R Tables Format" chapter. The Public Library (PUBLIB) header format is also described in that chapter.

:ROOT, :SEG, and :PUBLIB Commands. CCI creates an entry in T:SEG; builds T:ROMI and T:DCBV entries from the specified input options; allocates space for the PCB in the Root segment; and for the :SEG command, calls the PATHEND subroutine. PATHEND determines if the segment starts a different path; if so, writes out the T:SYMBOL and T:VALUE subtables for the overlaid part of the prior path on the disk scratch files; and sets and byte displacement pointers for the new segment's T:SYMBOL and T:VALUE subtables.

## Logical Flow of PASSONE

PASSONE branches to process T:MODIFY if CCI has just been previously called by PASSONE to input :MODIFY commands. Otherwise, PASSONE processes T:ROMI which has been built by either CCI or LIB. PASSONE inputs the ROMs from the devices specified in T:ROMI; builds T:MODULE entries for each ROM input; saves ROMs input from non-disk devices onto the X1 scratch file; and scans the ROMs for pass-one type load items. It then builds the following entries:

1. Parallel T:SYMBOL and T:VALUE entries from external DEF, PREF, SREF, and DSECT declarations. Entries in T:VALX are built when expressions defining DEFs cannot be resolved. Except for blank COMMON, a DSECT is allocated when first encountered, and its address is stored in the T:VALUE entry.

2. T:DCB entries from external DEF and REF declarations that begin with either M: or F:. The address of the DCB is either defined with an expression (for DEFs), or allocated by PASSTWO (for REFs) and stored in the T:DCB entry.

3. T:CSECT entries and allocates CSECTs when encountered.

4. T:FWD entries when FWDs are defined. Entries in T:FWDX are built when expressions defining FWDs cannot be resolved.

5. Entries in T:DECL whenever a DEF, REF, SREF, CSECT, or DSECT declaration is encountered.

At module end, the four module tables (T:DECL, T:CSECT, T:FWD, and T:FWDX) are packed together and moved to B:MT. If the buffer is full, the tables are output on X2.

When all the entries in T:ROMI have been processed, PASSONE determines whether the libraries specified have been searched. If not, PASSONE calls LIB to search the library specified. Note that the library is searched and the ROMs from the library are loaded before the next library is searched.

If there are any :MODIFY commands for the segment, PASSONE calls CCI. After CCI recalls PASSONE, control is returned to this point where T:MODIFY and T:MODULE are packed together and output on X3.

If there is a :SEG command in B:C, PASSONE calls CCI. Otherwise, the end of PASSONE is signaled. Blank COMMON is allocated at the end of the longest path (if not allocated previously) and the remaining T:SYMBOL, T:VALUE subtables are output. The resident table areas (T:DCB, T:SEG, T:DCBV, T:VALX) are set equal to the actual lengths of the data in the tables. The T:ROMI area length is set to zero (since it is not used by PASSTWO) and an end-of-file is written on X1. If any of the six scratch files overflowed, MAP is called; otherwise, PASSTWO is called.

## Logical Flow of LIB

The LIB segment first packs the 16 Dynamic Tables together at the top of the Dynamic Table area. The remaining space will be used for the LIB's tables. (Whenever enough room does not exist for the LIB's tables, the "pack" is written on the disk scratch file, X6.) LIB then creates T:LDEF, starting from the end of the "pack".

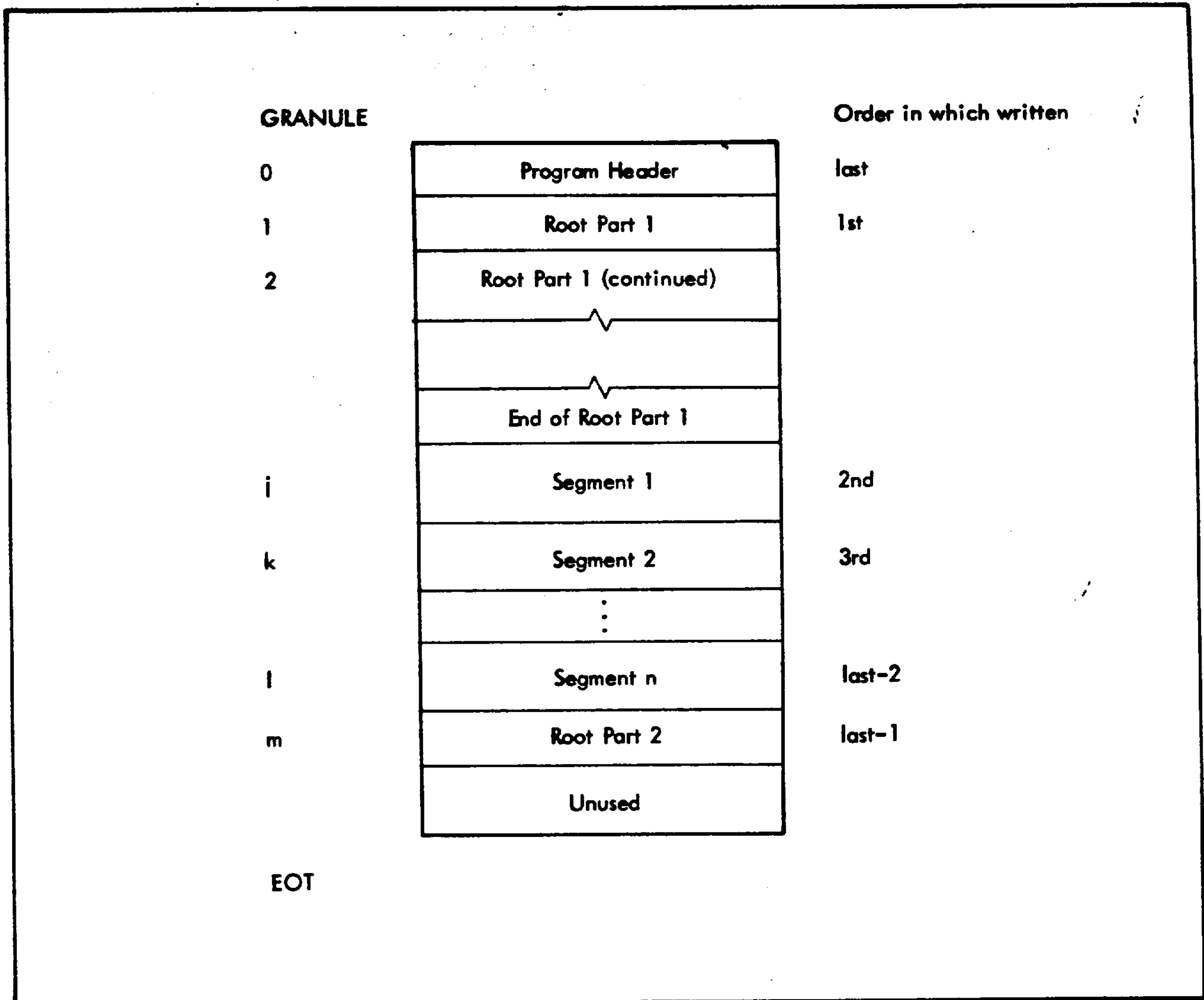



Figure 55. Program File Format

## Logical Flow of the Overlay Loader

After the Root segment has been loaded by the JCP, the Root calls the Monitor SEGLOAD function to read CCI into the overlay area and then transfers control to CCI to process the !OLOAD control command.

### Logical Flow of CCI

When CCI is called, there is usually a control command in the control command buffer (B:C). If not, CCI reads the next command into B:C and logs it onto LO. If the command terminates a :ROOT, :SEG, or :MODIFY substack, PASSONE is called; if it terminates an :ASSIGN substack, PASSTWO is called. If the command does not terminate a substack, CCI scans the options specified and performs the following functions for the different control commands.

!OLOAD Command. CCI sets flags; puts the program file name in M:OV DCB; builds T:PL, T:PUBVAL, and T:PUBSYM from files specified in the PUBLIB option; allocates the 14 remaining Dynamic Table areas; and if the GO option has been specified, builds T:ROMI.

The FWA of the EBCDIC, DEFREF, and MODIR files' buffer is calculated by subtracting the length of the longest file from the end of the Dynamic Table area. The EBCDIC file is read into the buffer and the entries in T:LDEF are converted to point from T:SYMBOL to entries in the EBCDIC file. T:LDEF entries not having corresponding EBCDIC entries are changed to null entries.

The DEFREF file is then read into the buffer. LIB uses the DEFREF file to satisfy PREFs in T:LDEF. All the DEFs and REFs from an entry in the DEFREF file are added to T:LDEF if at least one of the DEFs satisfies a PREF in T:LDEF The pointer to the ROM's MODIR file entry is saved in T:LROM, which is built backwards, beginning from the top of the DEFREF buffer. The DEFREF search is finished when all the PREFs in T:LDEF, that can be, are satisfied. T:LROM now contains pointers to all the library ROMs, and T:LDEF is no longer required.

The MODIR file is read into the buffer and the T:LROM entries are changed to point to the ROM's starting record number in the MODULE file.

The packed tables are read from the disk (if they were saved in X6), and T:LROM is moved to the temporary buffer (TEMPBUF) inside the LIB overlay while the Dynamic Tables are being unpacked. Note that if the DIAG segment were to be called at this point, TEMPBUF would be destroyed. T:LROM entries are converted into T:ROMI format and added to T:ROMI in the Dynamic Table area. PASSONE is then called to input the ROMs specified in T:ROMI.

## Logical Flow of PASSTWO

PASSTWO branches to process T:ASSIGN if CCI has just been previously called by PASSTWO to input :ASSIGN commands. Otherwise, it reorganizes the Dynamic Table area and moves the resident tables T:SEG, T:DCBV, and T:DCB to the end of T:PUBVAL and locates T:VALUE at the end of T:DCB. PASSTWO then allocates part two of the Root either at the end of the longest path or where specified on a :ROOT card.

PASSTWO is now ready to process the segments. It points to the first/next T:SEG entry; reads the segment's T:VALUE subtable into T:VALUE; calculates the number of granules required for the segment on the Program File; creates T:GRAN at the end of T:VALUE; reads the segment's T:MODIFY and T:MODULE tables at the top of T:VALX; and allocates the Work area (which is divided into granule partitions and contains all or part of the segment's partitioned core image) at the end of T:GRAN. The Work area extends to the Module Tables Buffer (B:MT), which varies in size, and is allocated backwards from the top of T:MODIFY. The Work area is dynamic and changes in size either when tables in B:MT are no longer required, or when another set of Module Tables is input.

PASSTWO is now ready to process the segment's ROMs. It points to the first/next T:MODULE entry; reads in the first/next set of Module Tables into B:MT if necessary; points to the current module's T:DECL, T:CSECT, T:FWD, and T:FWDX table; inputs the ROM; scans the load items; creates the absolute core image in the Work area using T:GRAN to locate the granules; and if the Work area gets full, outputs the necessary granules to the Program File.

PASSTWO repeats this cycle until all the modules in the segment have been input and then writes the granules remaining in core onto the program file. It then points to the next T:SEG entry and repeats the outer cycle until all the segments in the program have been created.

If a Public Library is not being created, PASSTWO builds T:GRAN for part two of the Root, located at the end of T:DCB. If there is an :ASSIGN command in B:C, PASSTWO allocates T:ASSN from the end of T:GRAN to the beginning of T:VALX and calls CCI to build T:ASSN. After CCI recalls PASSTWO, control is returned to this point. PASSTWO allocates the Work area at the end of T:ASSN (which may be of zero length); creates OVLOAD, DCBTAB, INTTAB, and the referenced DCBs; reassigns DCBs referenced in T:ASSN; writes part two of the Root on the Program File; creates the program header; and writes it on the Program File. If a Public Library is being created, T:SYMBOL and T:VALUE are output on the Program File. PASSTWO then exits by calling the MAP.

## Logical Flow of MAP

MAP moves T:SEG and T:DCB to the top of the Dynamic Table area, and unless "no MAP" was specified, outputs the program header information.

MAP points to the first/next T:SEG entry, and unless "no MAP" was specified, outputs the segment's header information. If either the PROGRAM or ALL option was specified, MAP reads the segment's T:MODIFY and T:MODULE tables into core at the end of T:DCB; locates B:MT at the end of T:MODULE; uses T:MODULE to read in the Module Tables associated with the segment; maps the segment's control sections (including Library CSECTs if ALL specified); and if this is the Root segment, lists T:DCB.

Regardless of the option specified, MAP reads the segment's T:SYMBOL and T:VALUE subtables into core at the end of T:DCB. If the ALL option was specified, MAP reads T:PUBSYM and T:PUBVAL in as part of the root's external table and lists all the symbols in the external table. If the PROGRAM option was specified, MAP lists all the non-library symbols in the external table. If either the SHORT or "no MAP" option was specified, MAP lists only the duplicate DEFs, undefined DEFs, unsatisfied REFs, and duplicate REFs.

This cycle is repeated until all the entries in T:SEG have been mapped. If a disk file used by the Loader overflowed, the number of granules used or needed for all files is listed. Otherwise, this information is output only if either the PROGRAM or ALL option was specified.

MAP terminates the Overlay Loader by either calling the Monitor EXIT function or ABORT function. MAP aborts and destroys the Program File if either a disk file overflowed or there were loading errors when a Public Library was being created.

### Logical Flow of DIAG

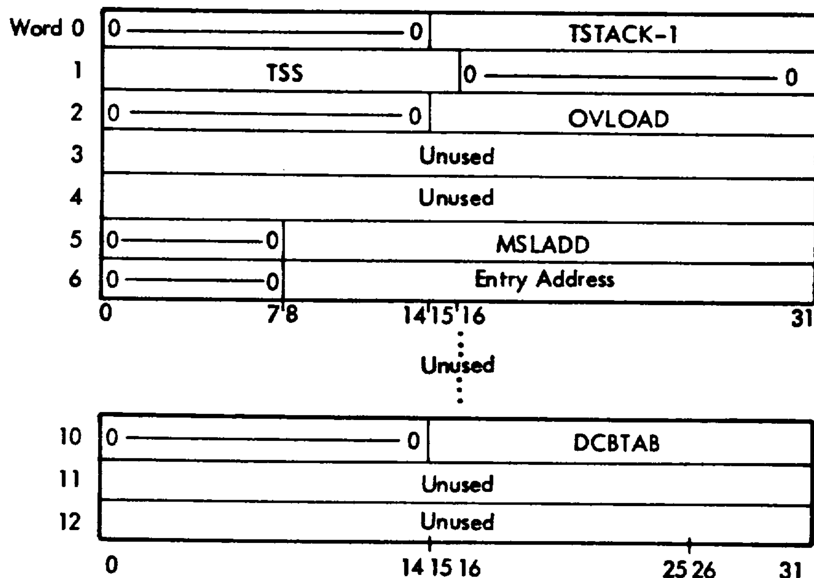
When the DIAG overlay is called, the environment of the calling program is unchanged. Since the DIAG segment overlays the calling segment, all the temporary and permanent storage cells used by the calling segment are located in either the Root or the Dynamic Table area. DIAG is called by the RDIAG subroutine which exists in the Root. When RDIAG is called, it saves the 16 registers and then calls in DIAG via the Monitor SEGLOAD function. DIAG outputs the specified diagnostic and depending upon the exit code associated with the diagnostic, either aborts, returns to RDIAG, or calls the Monitor WAIT function. If control is returned from the WAIT function, DIAG returns to RDIAG. RDIAG then reloads the calling segment via the Monitor SEGLOAD function, restores the 16 registers, and returns to the calling segment at the address following the RDIAG call.

## Loader-Generated Table Formats

The Loader creates the program's Program Control Block (PCB), DCB Table (DCBTAB), and Segment Loading Table (OVLOAD).

### PCB

The PCB exists as part of the Root segment and is initialized as shown below by PASSTWO, when the Root segment is created.




where

TSTACK is the address of the current top of the user's Temp Stack.

TSS indicates the size, in words, of the user's Temp Stack.

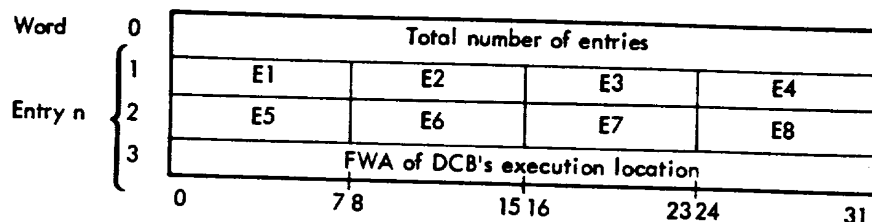
OVLOAD is the address of the table used by the SEGLOAD function to read in overlay segments or zero.

MSLADD is the address of the M:SL DCB used to load overlay segments.

DCBTAB is the address of a table of names and addresses of all of the user's DCBs. This table has the form given below.

## DCBTAB

DCBTAB is built from T:DCB, and is located in part two of the Root. DCBTAB has the format




where

E1-E8 is the EBCDIC name of the DCB (left-justified with trailing blanks).

## OVLOAD

The OVLOAD table contains the information necessary for the Monitor SEGLOAD function to read in overlay segments at execution time. One entry is created for each overlay segment. Thus, a program consisting only of a Root would not have an OVLOAD Table.

OVLOAD is located in part two of the Root. The format of an entry is such that it can be used as an FPT by SEGLOAD to read in the requested segment. OVLOAD is formatted as described in the "CP-R Tables Format" chapter.

## Loading Overlay Loader

Before the Overlay Loader can be loaded, the OLOAD file in the SP area must be previously allocated by RADEDIT. It is loaded by the JCP Loader with the ILOAD command. It is critical that the ROMs of the Overlay Loader's segments be ordered correctly, so that the segment's idents assigned by the JCP Loader coincide with the idents used within the program. The segment idents are listed below:

| SEG | IDENT |
|---|---|
| ROOT | 0 |
| CCI | 1 |
| PASSONE | 2 |
| PASSTWO | 3 |
| MAP | 4 |
| DIAG | 5 |
| LIB | 6 |

The overall flow of the Overlay Loader is illustrated in Figures 56 through 63.

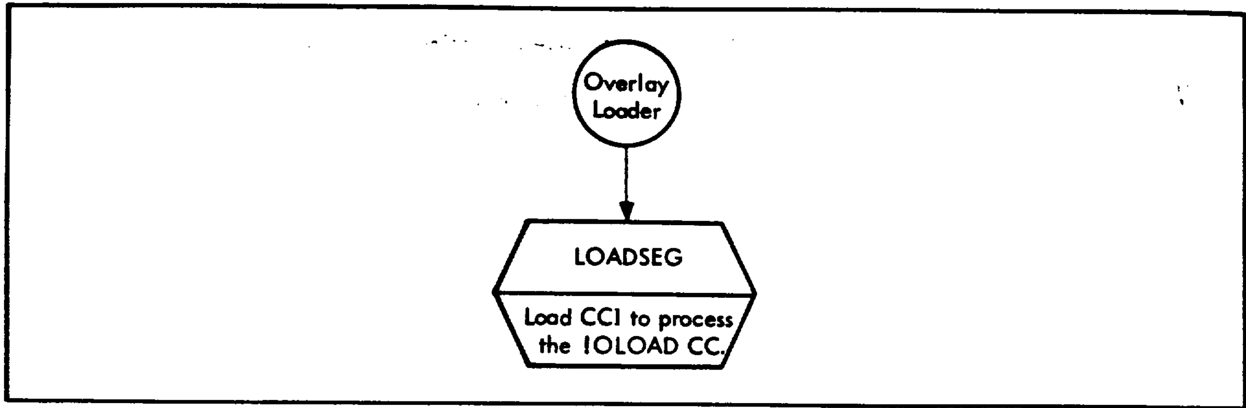



Figure 56. Overlay Loader Flow, IOLOAD

CCI

Is there an unprocessed CC in B:C ?

no

RDCC

Read next CC into B:C.

yes

Does CC terminate a :ROOT, :SEG, or :MODIFY sub-stack?

yes

LOADSEG

Load PASSONE to process T:ROMI.

no

Does CC terminate an :ASSIGN substack ?

yes

LOADSEG

Load PASSTWO to process T:ASSN.

no

Process control command.

Figure 57. Overlay Loader Flow, CCI

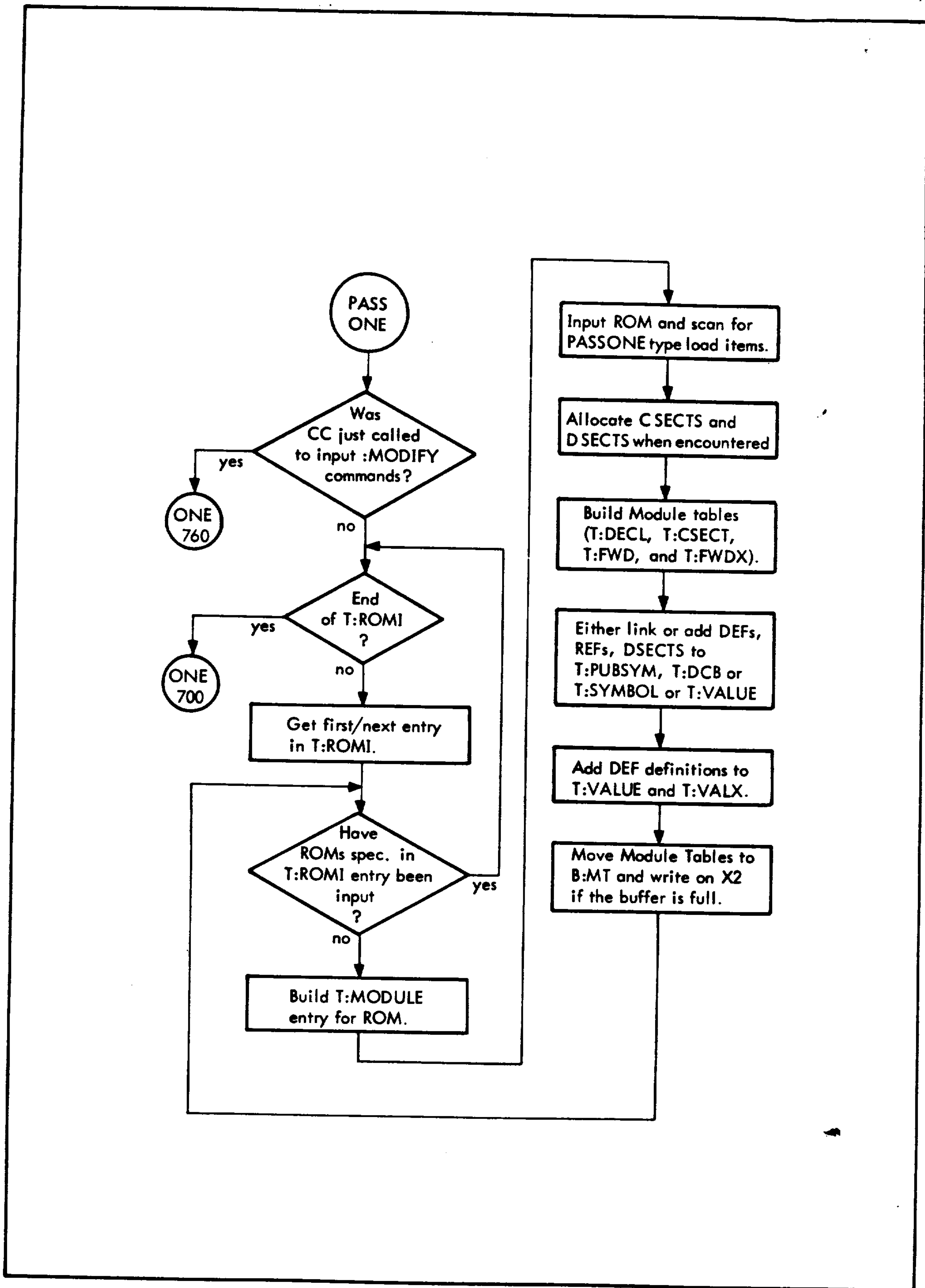



Figure 58. Overlay Loader Flow, PASSONE

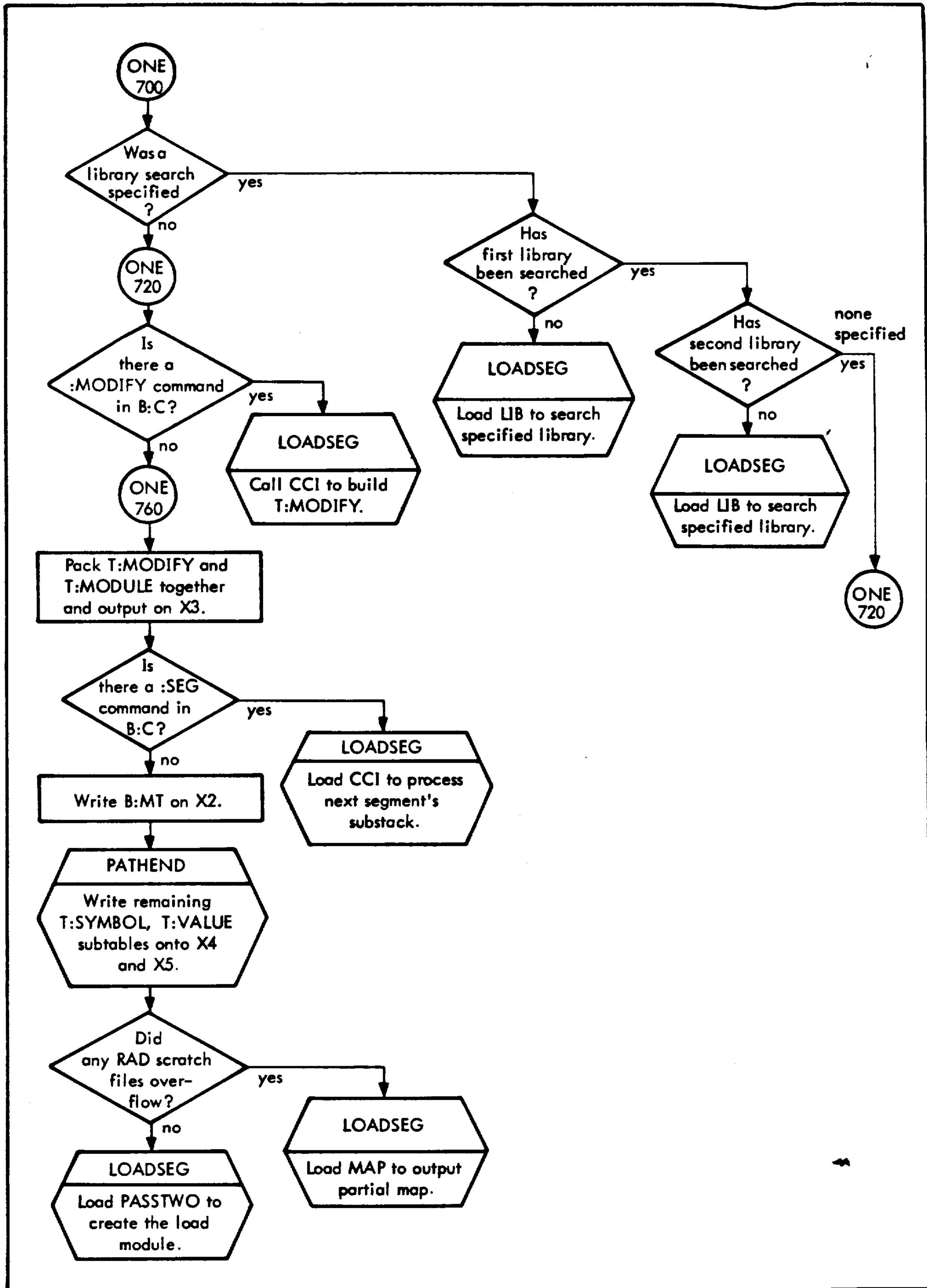



Figure 58. Overlay Loader Flow, PASSONE (cont.)

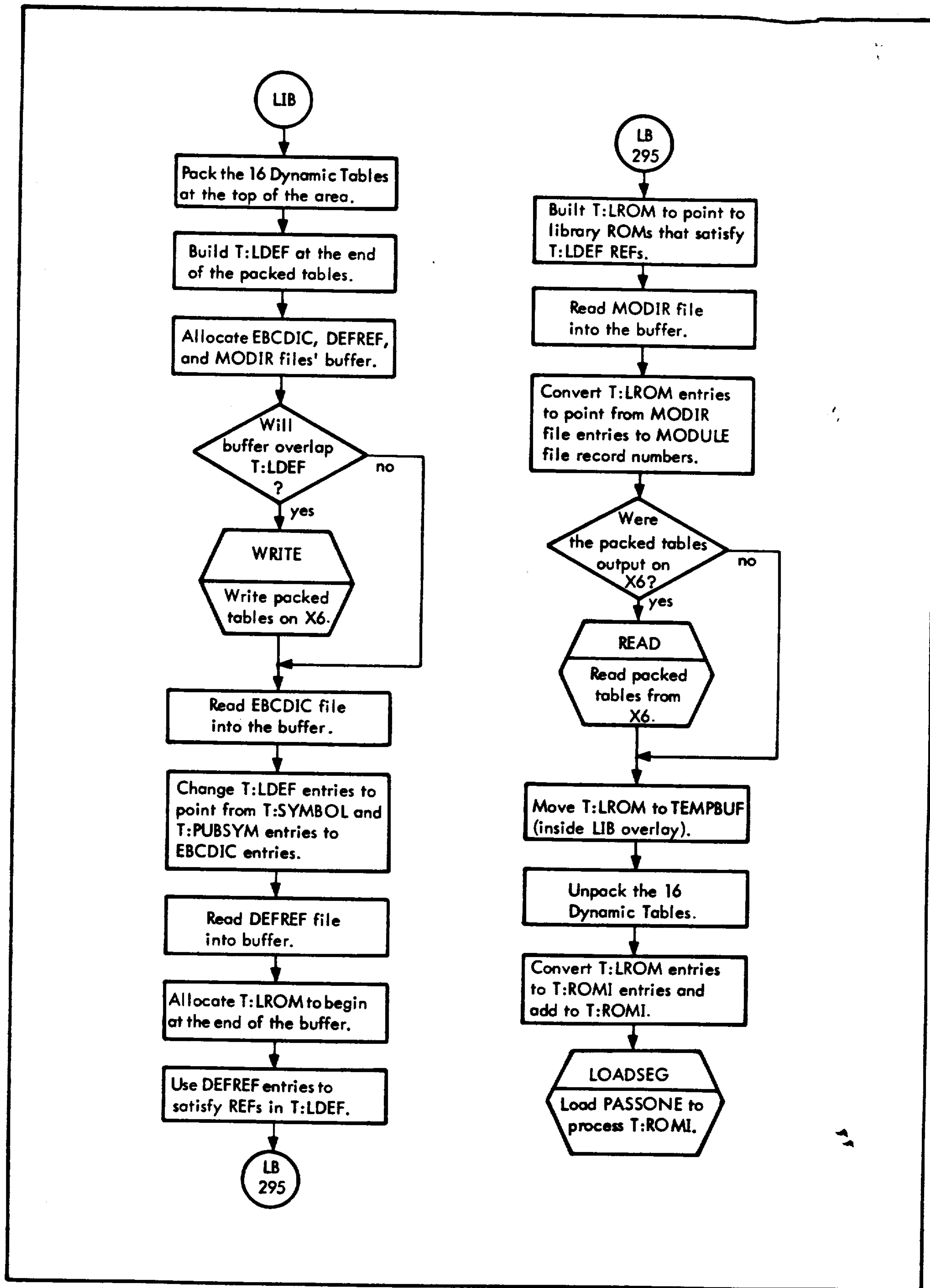



Figure 58. Overlay Loader Flow, PASSONE (cont.)

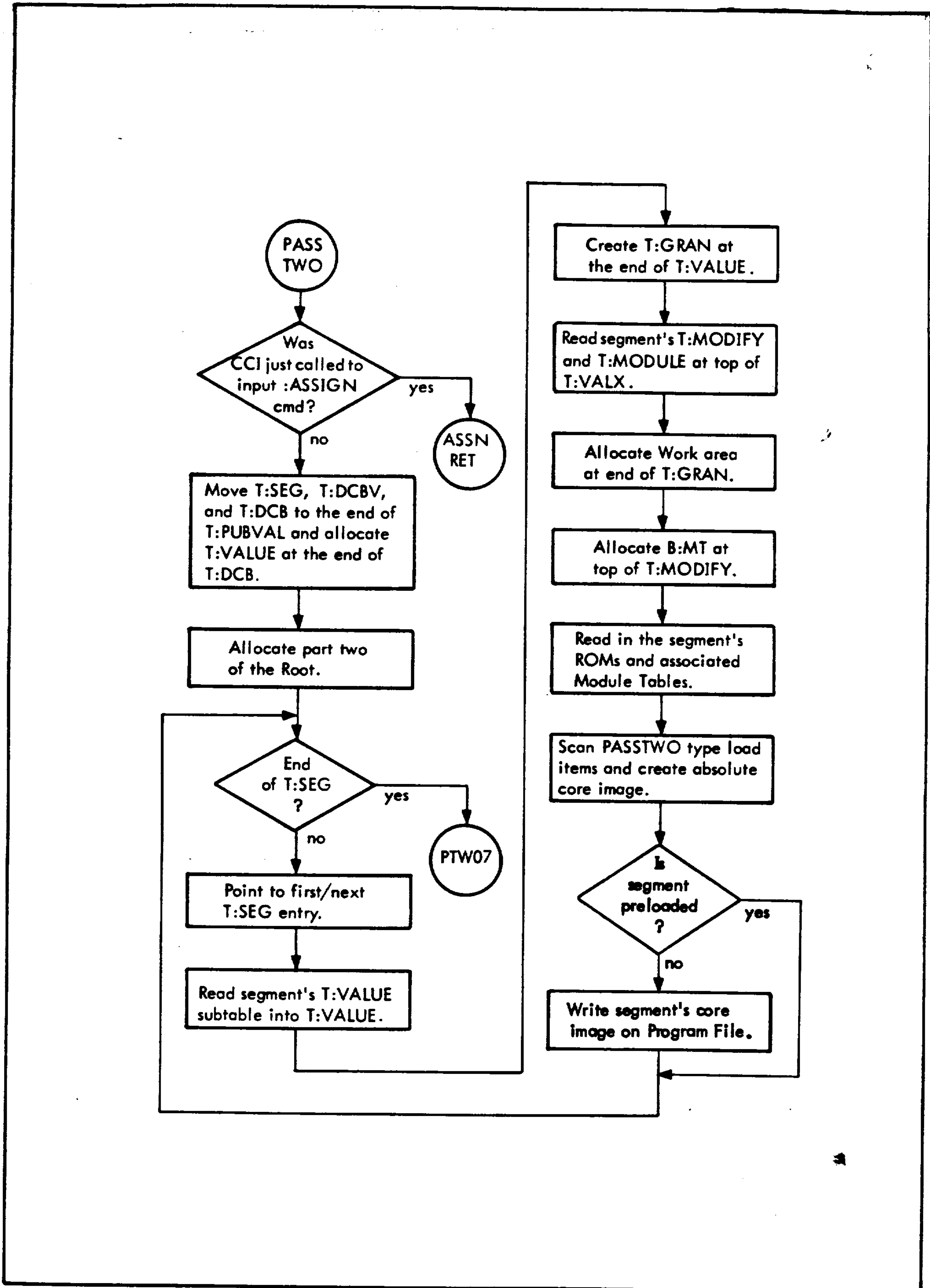



Figure 59. Overlay Loader Flow, PASSTWO

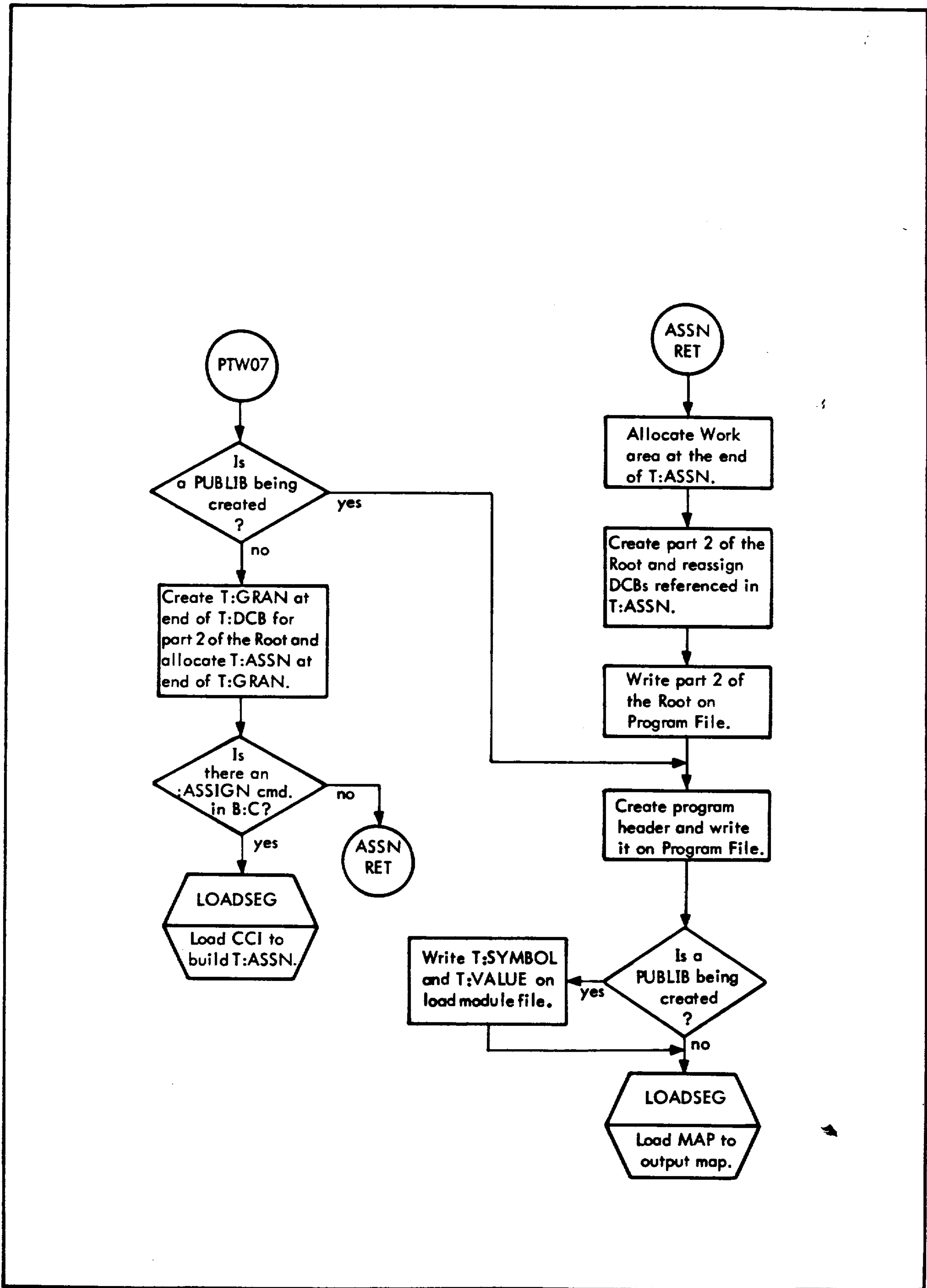



Figure 59. Overlay Loader Flow, PASSTWO (cont.)

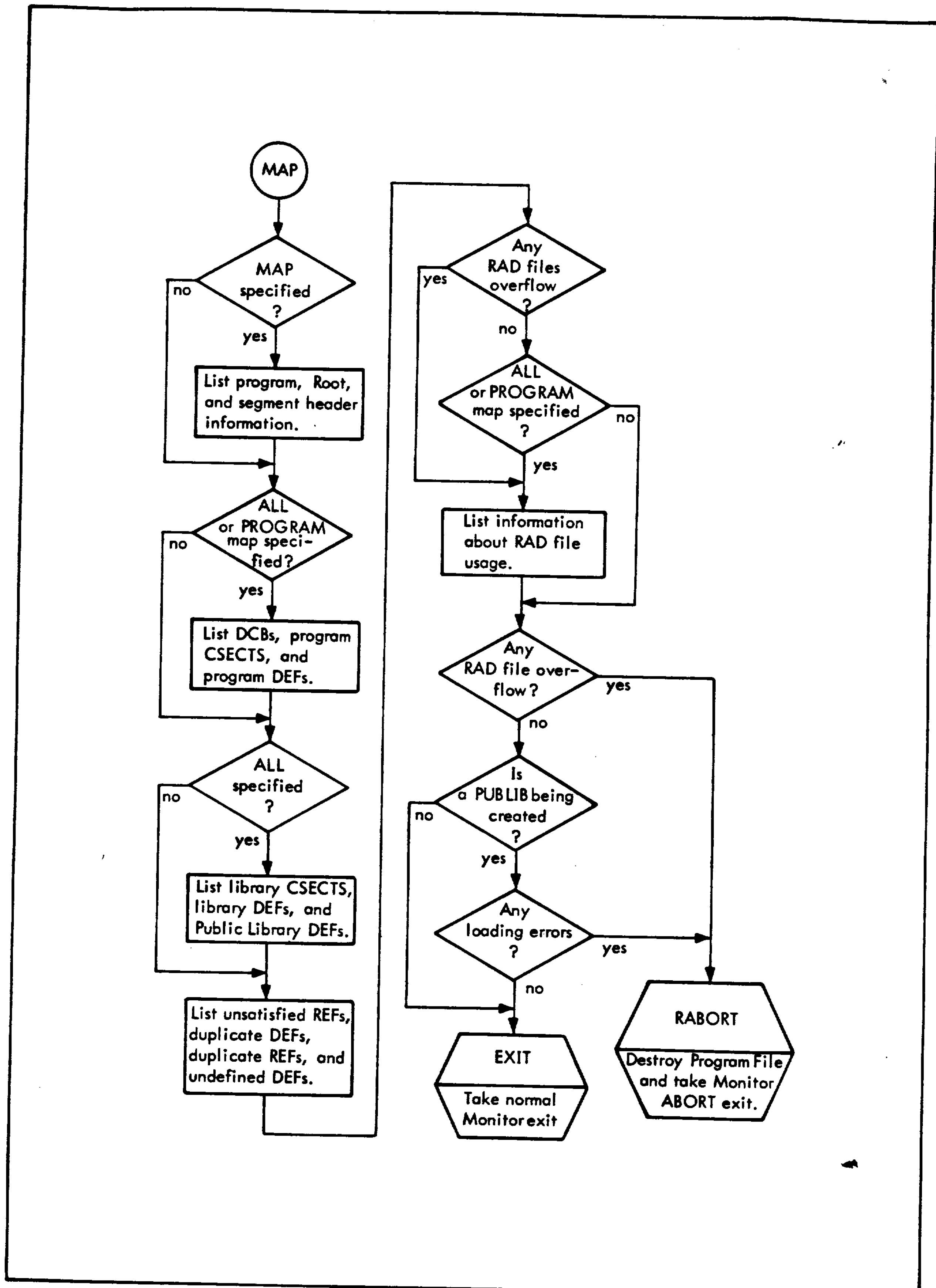



Figure 60. Overlay Loader Flow, MAP

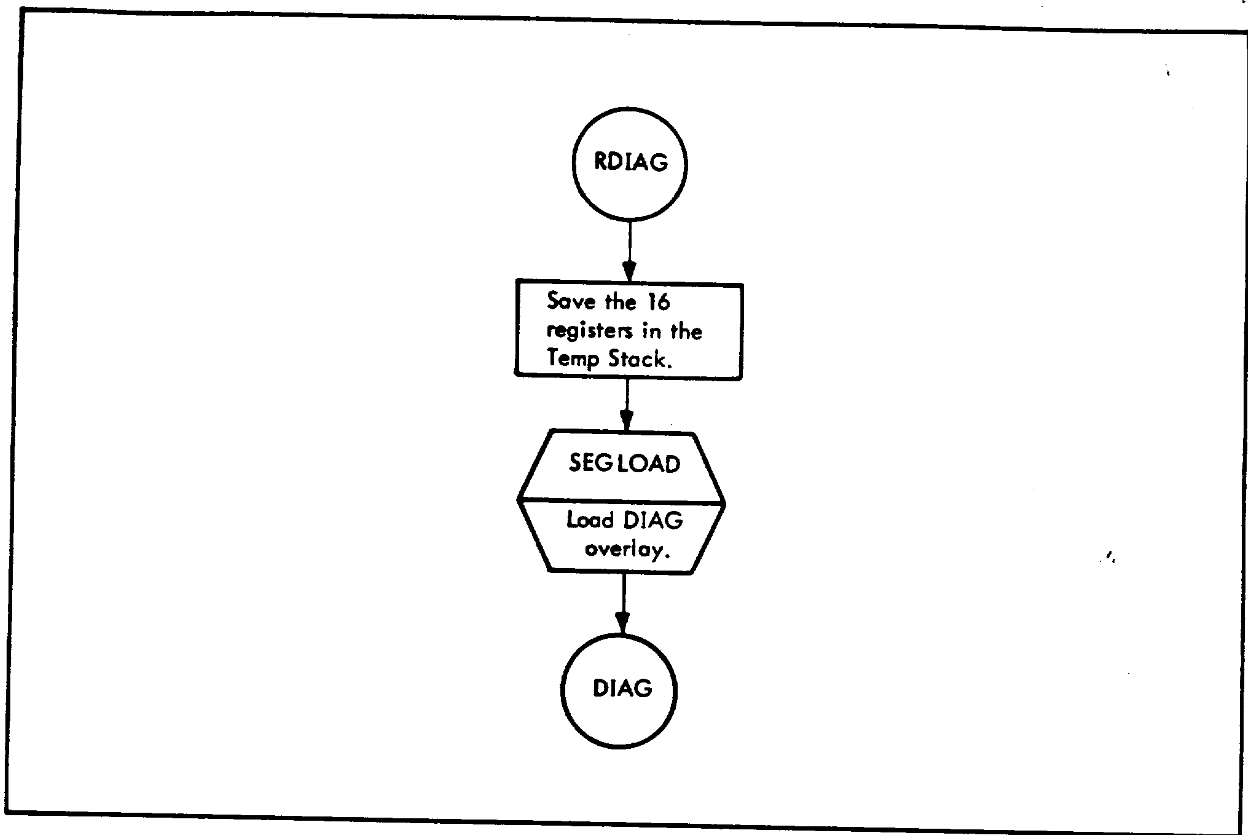



Figure 61. Overlay Loader Flow, RDIAG

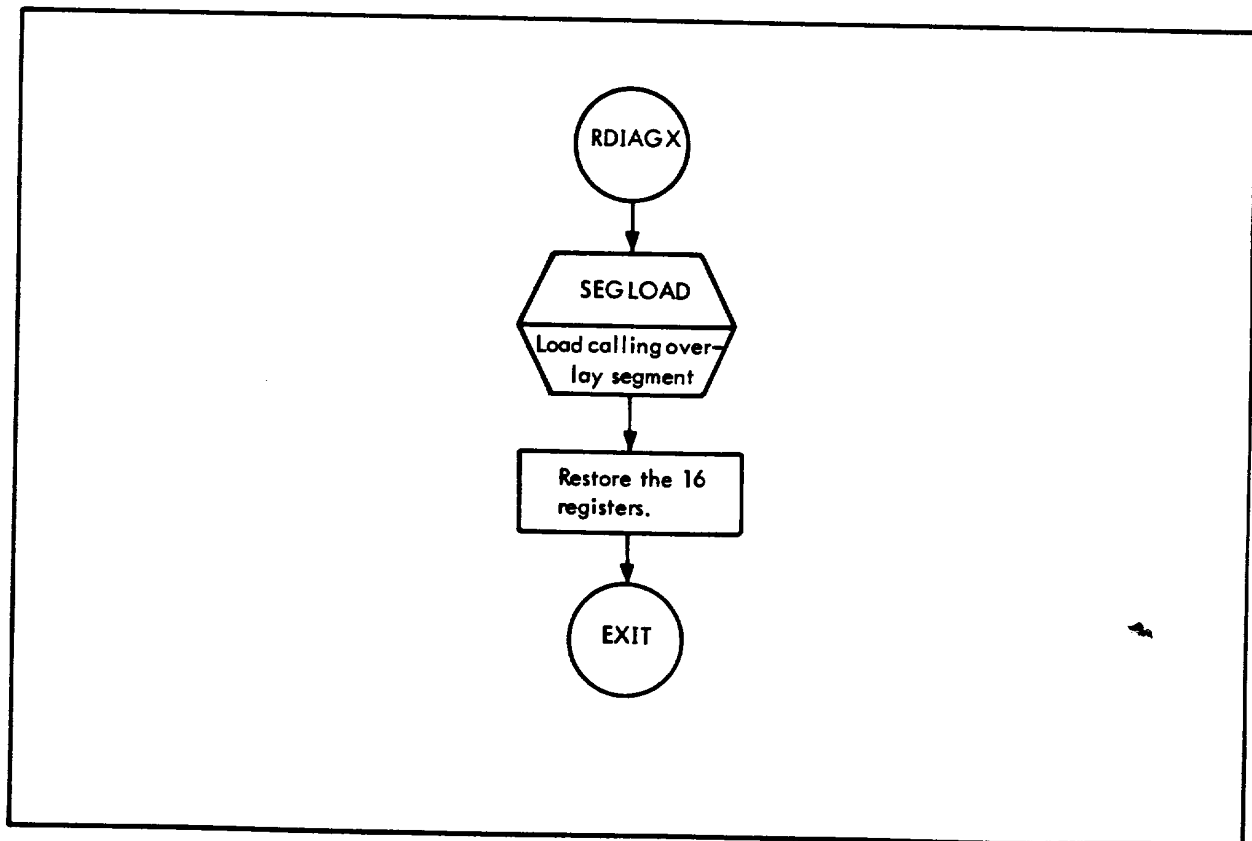



Figure 62. Overlay Loader Flow, RDIAGX

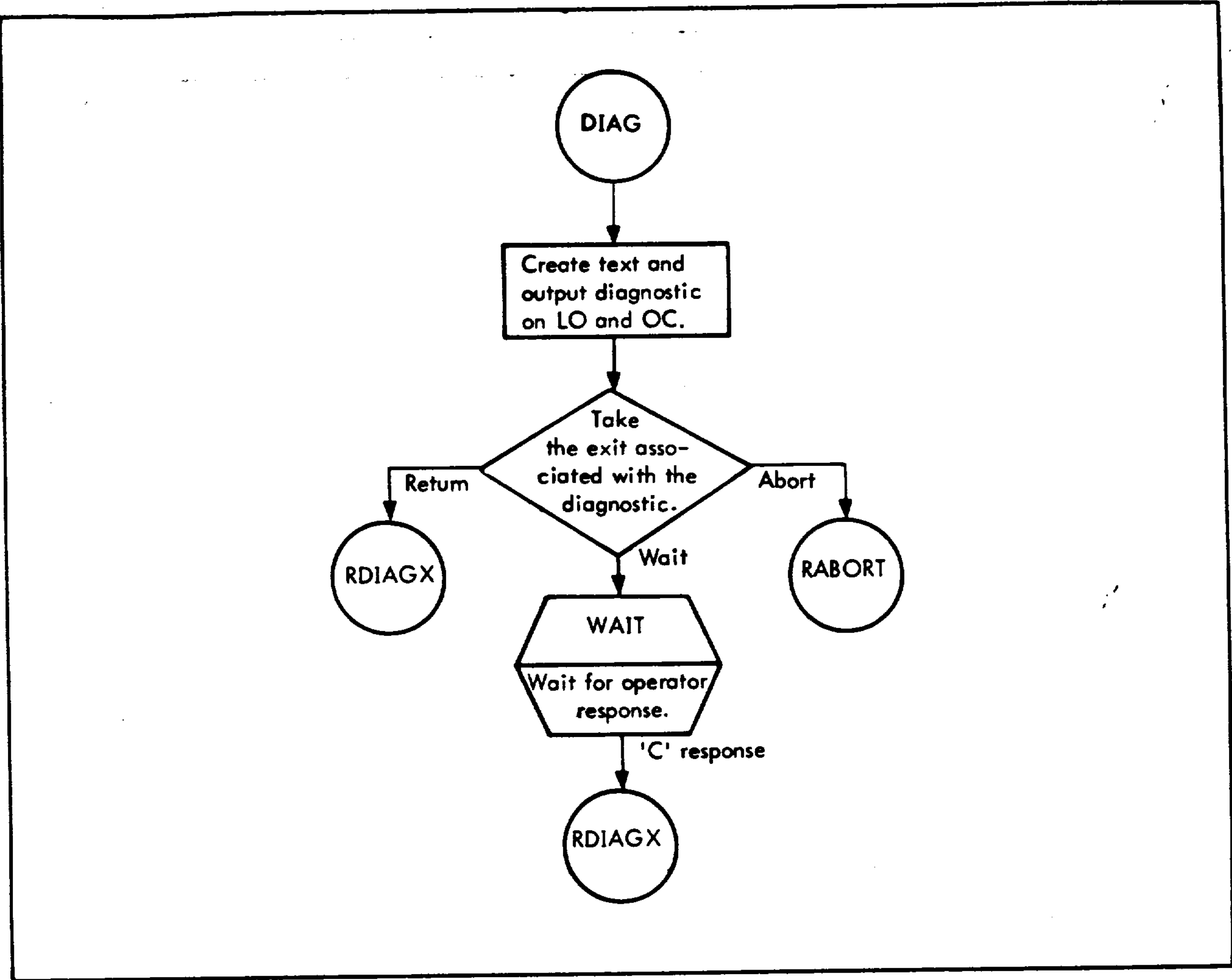



Figure 63. Overlay Loader Flow, DIAG

# 11. RADEDIT

The RADEDIT processor provides a convenient means for the creation and maintenance of files in the various permanent disk areas defined by SYSGEN. In particular, it provides job stream and terminal access to the system calls that create and delete files in these areas, to functions that reclaim space lost to deleted files, and to a means of initializing an area to its unused space.

It also provides utility functions that produce listings of active files, enter data into a file, and save areas on magnetic tape.

RADEDIT executes as a normal program in the background job stream. When called from TEL, using the name MUST, it executes as a foreground secondary task.

## Functional Flow

Upon being loaded, RADEDIT performs one-time initialization to acquire memory for use as buffers and work space. This was created during OLOAD as a defined segment 20,000 words long. How much space is actually acquired, in pages, is determined by

N = max-path-bgn

where

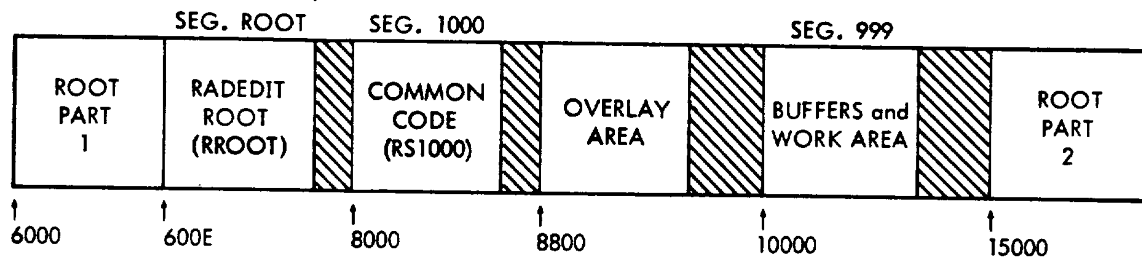
path is the number of pages in the longest segment path of RADEDIT.

bgn is the page address of the start of RADEDIT.

max is the page address of the end of the range permitted to an SMM program (considered as a good estimate at how big a variable-sized program should try to be).

The last value is taken from K:FSMM if RADEDIT is executing in foreground (under TJE), or from K:BCKEND if it is in background.

The layout of RADEDIT in memory is then:




After the one-time initialization is completed, the normal command processing loop is entered. The functional flow of this loop is shown in Figure 64.

## Permanent Disk Area Maintenance

RADEDIT creates and maintains files and file directories on any of the areas defined in the Master Dictionary except the BT, XA, or CK areas. (See Chapter 8 for a description of the Master Dictionary.) It uses the system service calls ALLOT, DELETE and TRUNCATE to perform these functions.

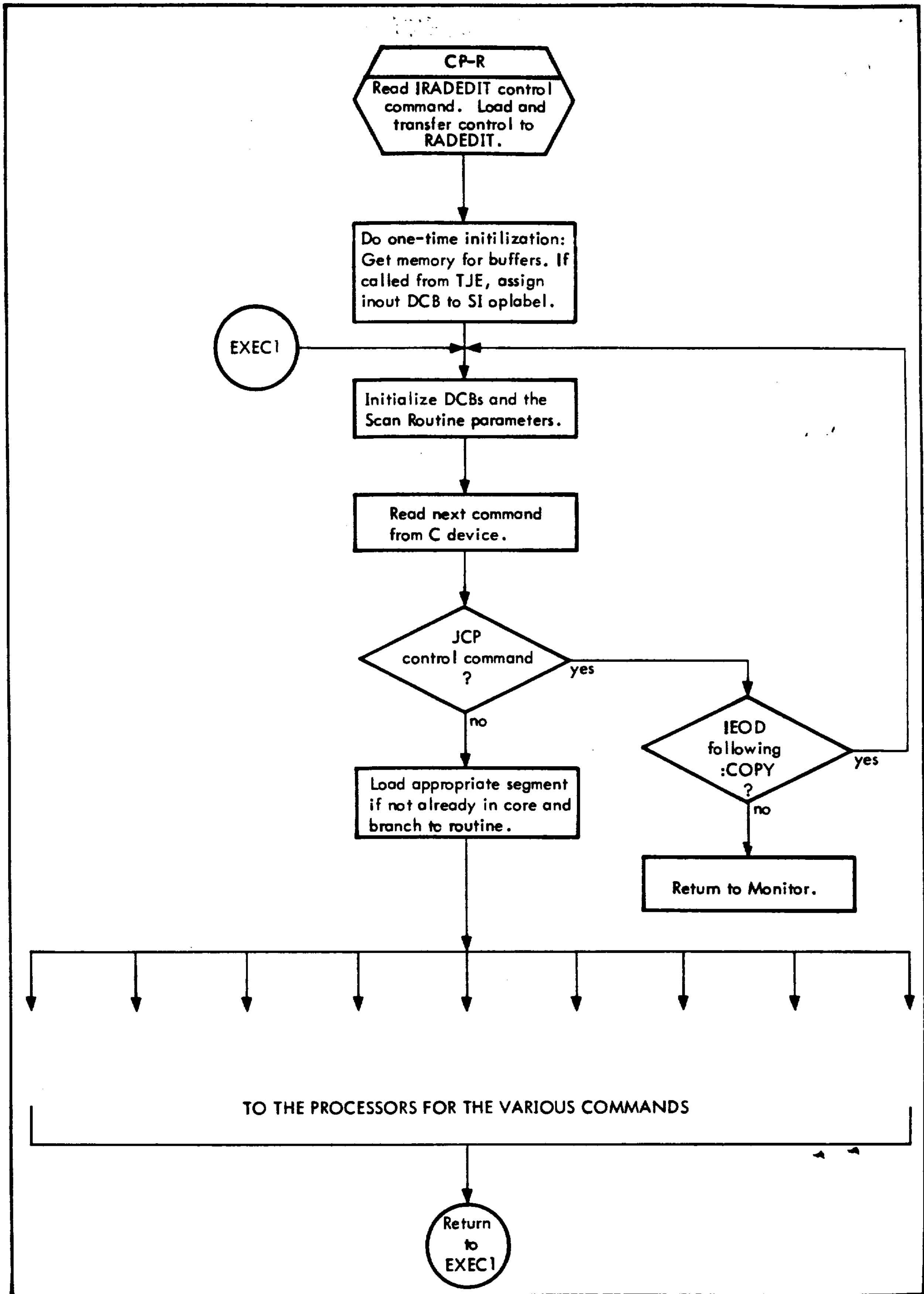



Figure 64. RADEDIT Functional Flow

## Utility Routines

In order to simplify processing and reduce dependencies on system table fomats, a set of utility subroutines have been implemented. These routines are called whenever any of the information they process is required. In the description of the individual commands that follow in the next section, some of these routines are explicitly mentioned as being called. However, almost every command processer will call one or more whether mentioned or not.

**UNPKMASD** This routine gets information on the area whose index is in the cell AREA. If the area is allocated, the F:BI DCB is assigned to the area and a GETASN CAL is issued to get such information as device address, begin of area, end of area, and write protection code. All information is stored in the MASDxxxx table in the context segment. AREA < Ø specifies any public area; this is accepted by setting the area name to blanks and bypassing the GETASN CAL. A valid area is indicated by returning to the link address + 1, unallocated areas cause a return to the link address.

**GAN** This routine processes a list of area names (as for example in a :MAP command) and marks those given in the AREASWS byte table. Each area requested explicitly is marked with an X'FF'; areas requested by the 'ALL' option are marked with X'ØF'. The 'ALL' option may or may not include the BT, CK, IS and OS areas depending upon the setting of the switch passed to GAN in the link register + 1 (RLNK + 1). If no errors are detected, the return is to the link address + 1. If any errors, invalid area names, or no area names given, are found, the return is to the link address and the address of an error message is in R15 (link register + 1).

**SCAN** This is a general routine used to scan all names, numbers, etc. It is identical to the SCAN routines in the CP-R KEYSCN module.

**GETIOID** The general routine to scan an input identifier, which may be any of a file, device, or oplabel. The routine determines which is given, and builds a table in the form required by the ASSIGN CAL, supplying any defaults requested or implied. It also generates the appropriate P-bits for ORing into the P-bit word of an ASSIGN CAL.

**GETFID, GETDEV, GETOPLB, GETANY** These routines are used to call GETIOID to get a file identifier, device name, oplabel name, or any of these three, respectively.

**GETFSTSD** This routine reads the first directory sector (sector Ø) of the area assigned to the F:BI DCB into BUFF1 and does some preliminary determination of the status of the area. Return condition codes indicate:

| | |
|---|---|
| CC = Ø | Directory continued (1st word < Ø) |
| CC = 4 | Directory empty (1st word = Ø) |
| CC = 8 | Directory's last sector (1st word > Ø) |

**GETNXTSD** This routine reads the next sector of the directory whose current sector is in BUFF1 by accessing the directory link address. It reads both old style (pre-DØØ-CP-R) as well as new style (DØØ and on) directories based on the setting of the MASDFRMT cell. (See GET1SFIL below.)

**GETAX** This routine converts a symbolic area name into an area index. The input name is left justified in R8; the area index is returned in AREA and R1. An undefined area name results in an error being indicated by returning to the link address. A valid area returns the link address + 1. An area name of zeros or blanks is treated as the specification of any public area and is indicated by an index of - 1.

**GET1SFIL and GETNXFIL** These routines are used whenever an area's directory is to be searched or processed one file at a time in their order in the directory. GET1SFIL is called initially. It calls GETFSTSD to read directory sector zero and test for a cleared area. It then tests the header field to determine if it has an old (pre-DØØ-CP-R) or a new (DØØ and on) style directory and sets MASDFRMT to 1 or Ø accordingly.

The address of the first or next directory entry (in the BUFF1 buffer) is returned on each call in R5. GETNXTSD is called to read the next directory sector as needed. The routines return relative to the link address according to what is found

LINK + Ø: Error in directory

LINK + 1: Directory empty (GET1SFIL), no more entries (GETNXFIL)

LINK + 2: Next entry address in R5

**UNPKDIRE** This routine will unpack a directory entry into the DIRExxxx table in the context segment. It will process old and new style formats according to the setting of MASDFRMT. It supplies blank account names for entries which do not have them. For old style entries it also supplies zero for extent information and the length of the entry. The address of the entry to process is in R5.

This routine and PACKDIRE are the only two routines in RADEDIT that know, or should know, the actual format of a directory entry.

**PACKDIRE** This routine will form a new style directory entry from the information in the DIRExxxx table and store it in memory starting at the address in R6. The name that is stored in the entry is a function of the cell DIRESTAT, as follows.

DIRESTAT = Ø ⇒ deleted file ==⇒ name of all zeros

DIRESTAT = 1 ⇒ bad sector entry ==⇒ name of all F's

DIRESTAT = 2 ⇒ good file ==⇒ name in DIRENAME

## Control Commands

The creation and maintenance of files in the disk areas is done through the :ALLOT, :DELETE, :TRUNCATE and :SQUEEZE commands. All but :SQUEEZE use the monitor service call to do their function.

**:ALLOT** The utility subroutine GETFID is used to get the file name, and area and account name if given. The other parameters are scanned and validated, and stored in the DIRExxxx table entries. RF is only allowed if the FP area is specified. Specifying a library file in the SP or FP areas forces organization, and RSIZE (if MODULE), to the needed values. See ALLOT under LIBRARY FILE MAINTENANCE.

An ALLOT FPT is then constructed based on the parameters present in the command and the monitor service called to actually perform the allocation.

See also the description of the ALLOT service in Chapter 3.

**:DELETE** The file name and area and/or account name is scanned using the GETFID utility routine. A DELETE FPT is constructed specifying the appropriate area and account if present, and the monitor service called to delete the file. Multiple files can be deleted with one :DELETE command by repeating the FILE, fid parameters.

**:TRUNCATE** This command can truncate either a specific file or all files in an area. If the keyword "FILE" is scanned, an individual file is assumed, and the utility routine GETFID is called to get its name, area, and account. A TRUNCATE FPT is formed from the parameters obtained by GETFID and the monitor service TRUNCATE called.

If the keyword "FILE" is not scanned first, it is assumed what was scanned was an area name. The utility routine GETAX is called to validate it as an area name, and then UNPKMASD to get its Master Dictionary information. The routines GET1SFIL and GETNXFIL are used to get each entry in the directory, UNPKDIRE is used to determine if it is an active file entry and get its name if so. A TRUNCATE FPT is built and the monitor service called for each active file that has a file size greater than zero. After each file or area, :TRUNCATE loops back to accept another file or area.

**:SQUEEZE** The :SQUEEZE command list the areas to be processed. The list is scanned using the GAN routine to set flags in the AREASWS table for each area specified. SQUEEZE serves two purposes: 1) it allows the reclaiming of space that has been lost due to truncations and reallocation of deleted files or that is allocated to deleted files; and 2) it provides a means of collecting the extents of extensible files together into contiguous space on the disk, and, if not inhibited by a FIX when allotted, merging the extents into a single entry in the directory. Figure 65 illustrates the disk areas before and after squeezing to reclaim space.

The :SQUEEZE processor uses UNPKMASD to get the area's BOA and EOA sector addresses and the sector size, and GET1SFIL/GETNXFIL to get each active or Bad Sector entry in the directory. It uses UNPKDIRE/PACKDIRE to get all information from a files entry and to move an entry to and from the directory sectors.

To process each area, the directory is read and each active entry is copied to a linked list in the background buffer. This list is linked, both forward and backwards in the order in which it appeared in the directory. This is also the order in which they are allocated space in the area. Also, extended files are linked by extents in ascending order. Thereafter, the directory on disk is no longer used.

The algorithm used to relocate files and extents in order combine extents of normal extended files and juxtapose extents if files marked by "FIX" is described in greater detail in Figure 69. Basically, however, each file is processed as it occurs in the directory. If it is not an extended file it is moved down to the next sector to be allocated in the squeezed area. When the first occurrence of a file is extent Ø, it is processed initially as a normal file. When the first occurrence of a file is not extent Ø, the lowest numbered extent that has not been squeezed is found and it is moved to the next sector to be allocated in the squeezed area.

After an extent of an extended file has been squeezed, non-FIXed extents are combined with the previous extent, if any, and the next extent chosen as the file entry to be squeezed next.

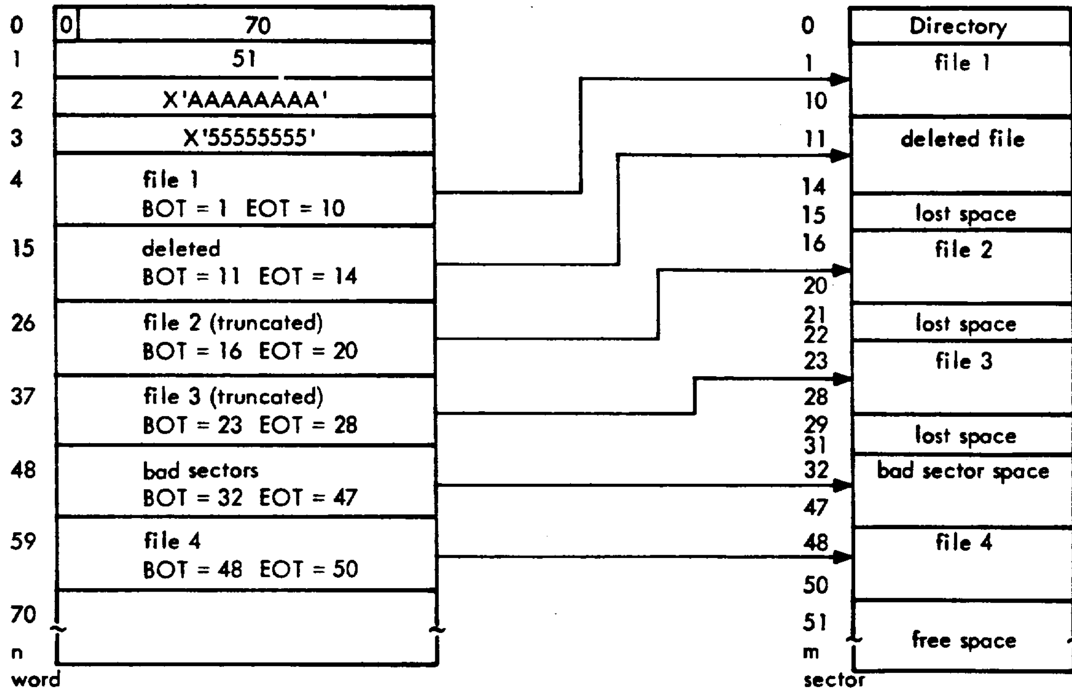
Whenever there is insufficient space between the last squeezed file (extent) and the next nonsqueezed entry to insert the next extent; space is made by trying, in order 1) move the next entry and its file to free space at the end of the area; 2) move the file to the best fit space not used (either a deleted file or truncated space); or 3) rotate the next extent and all intervening entries up on the disk: the entry immediately before the next extent will be moved up over where the next extent was, the entry before it will be moved over it, etc., and the next extent will then follow the previous.

These attempts to create enough space for the next extent will fail if the next entry is a BADSECTORs entry, or if there is such an entry in the range of files to be rotated. In these cases, the extent chain is broken and the two pieces are processed as two separate files.

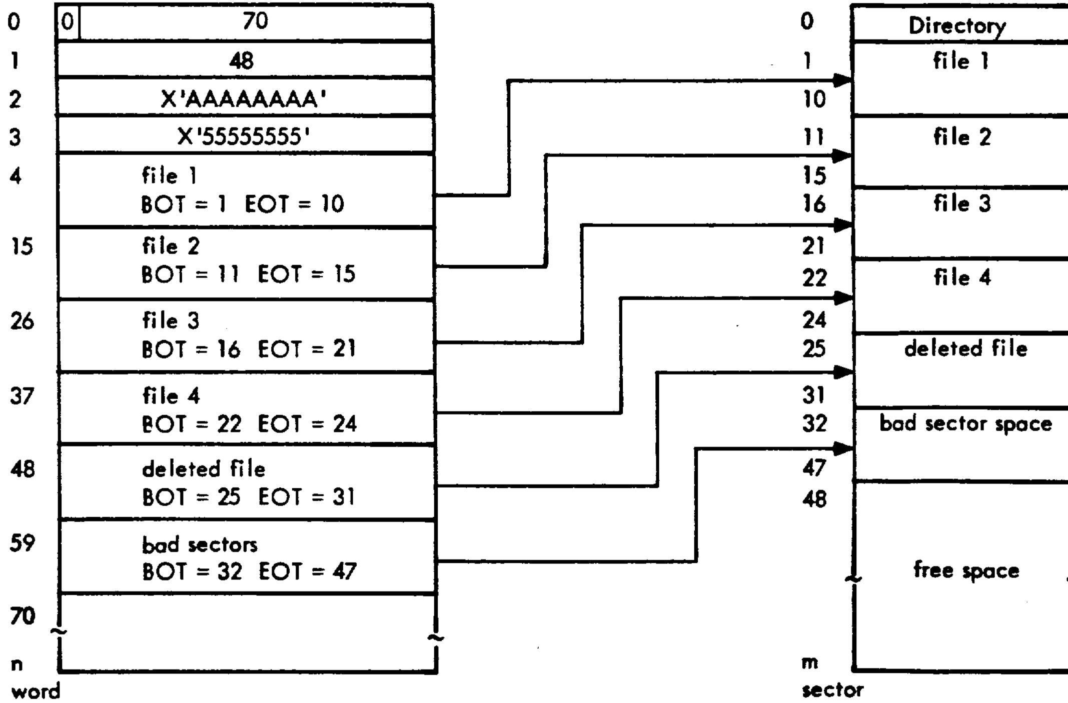
Space before BADSECTOR entries are filled with 1) the largest nonextended file that will fit; then 2) the smallest extent Ø of an extended file – this will be followed by as many of its extensions as will fit; and finally 3) a deleted file entry that exactly fills the space.

Note that for areas containing BADSECTOR entries, it is possible for the SQUEEZED area to require more space than before. This is a function of where the BADSECTOR entry is in the area and whether a large deleted file entry has to be formed.

Disk Area Before SQUEEZE




Disk Area After SQUEEZE



## Library File Maintenance

Both the System Library files residing in the SP area and the User Library files residing in the FP area have the same file structure. Each library consists of one blocked Module File (MODULE) and three unblocked files: the Module Directory File (MODIR), EBCDIC File (EBCDIC), and DEFREF File (DEFREF).

The MODIR File contains general information about each library module, including its name, where in the MODULE File it is located, and its size. The MODULE File contains the object modules. The EBCDIC File contains only the DEFs and REFs of the library modules. The DEFREF File contains indices to the DEFs and REFs in the EBCDIC File for each module. These files must be defined via the :ALLOT command before attempting to generate them via the :COPY command.

### Algorithms for Computing Library File Lengths

The following algorithms may be used to determine the approximate lengths of the four files in a library. It is not crucial that the file lengths be exact, since any unused space can be recovered via the :TRUNCATE command. The approximate number of sectors ($n_{MODIR}$) required in the MODIR File is

$$n_{MODIR} = \frac{3(i)}{s}$$

where

i  is the number of modules to be placed in the library.

s  is the disk sector size in words.

3 words  is the length of a MODIR File entry.

The approximate number of sectors ($n_{EBCDIC}$) $= \frac{2(d)}{s}$

where

d  is the unique number of DEFs in the library.

s  is the disk sector size in words.

2 words  is the average length of an EBCDIC File entry.

The approximate number of records ($n_{MODULE}$) required in the MODULE File is

$$n_{MODULE} = \sum_{i=1}^{n} C_i$$

where

n  is the total number of modules in the library.

$C_i$  is the number of card images in the ith library routine.

The approximate number of sectors ($n_{DEFREF}$) required in the DEFREF File is

$$n_{DEFREF} = \frac{\sum_{i=1}^{n} 1 + \frac{d_i + r_i}{2}}{s}$$

where

- n is the total number of routines in the library.
- d is the number of DEFs in the ith library routine.
- r is the number of REFs in the ith library routine.
- s is the disk sector size in words.

### Library File Formats

The library file formats are described below. These files are generated from object modules read in via the :COPY command.

#### MODIR File

The MODIR File is an unblocked, sequential access file and acts as a directory to the MODULE File. The file always consists of one variable length record that increases in size as object modules are added to the library. There is one entry in the MODIR File for each object module, with each entry consisting of three words.

| Words | Bits 0–15 | Bits 16–31 |
|---|---|---|
| 0 | MODULE File record no. | Records per module |
| 1 | Module name (first DEF) | |
| 2 | Module name | |
| 3 | MODULE File record no. | Records per module |
| 4 | Module name | |
| 5 | Module name | |
| 6 | | |
| 7 | ⋮ | |
| 8 | | |
| 9 | | |
| 10 | ⋮ | |
| 11 | | |
| 12 | | |

where

MODULE File record no. is the relative record within the MODULE File where the object module (corresponding to this entry) begins.

records per module  is the number of records in the object module.

module name  is the name of the object module that is the first DEF in an object module.

A deleted entry contains zeros in all three words.

## MODULE File

The MODULE File is a blocked, sequential access file and contains the object modules. The location of the object module within the file and the size is indicated by the MODIR File entry.

## EBCDIC File

The EBCDIC File is an unblocked, sequential access file. The file always consists of one variable length record that increases in size as object modules are added to the library. The EBCDIC File contains all the unique DEFs and REFs in the library object modules.

| | | | | |
|---|---|---|---|---|
| 0 | n | e | e | e |
| 1 | e | n | e | e |
| 2 | e | e | e | e |
| 3 | e | e | | |

where

n  is the number of bytes in entry (including itself).

e  is an external definition or reference in EBCDIC.

## DEFREF File

The DEFREF File is an unblocked, sequential access file. The file always consists of one variable length record that increases in size as object modules are added to the library. For each module there is one entry that varies in size according to the number of DEFs, DSECTs, and REFs. DEFs and DSECTs always precede the REFs in the entry.

| 0 | 1 | : 15 | 16 | 17 | 18 : 31 |
|---|---|---|---|---|---|
| Entry size (no. 1) | | | MODIR File index | | |
| | d | DEF 1 | | d | DEF 2 |
| r | d | DSECT 1 | r | | REF 1 |
| r | | REF 2 | Entry size (no. 2) | | |
| MODIR File index | | | | d | DEF 1 |
| r | | REF 1 | r | | REF 2 |

where

entry size  is the number of halfword entries (including itself) for the object module. 3 ≤ entry size ≤ 32,767.

MODIR File index  is the relative halfword in the MODIR File that identifies the object module. 0 ≤ MODIR File index ≤ 32,767. −1 means a deleted entry.

| | | |
|---|---|---|
| d | if d = 1, the entry is a DEF | if d and r both = 1, the entry is a DSECT. |
| r | if r = 1, the entry is a REF | |

DEF n  is the byte index of an external definition in the EBCDIC File.

REF n  is the byte index of an external reference in the EBCDIC File.

DSECT n  is the byte index of a DSECT in the EBCDIC file.

A deleted DEFREF entry contains a MODIR File index of -1, with the rest of the entry remaining the same.

## Command Execution

The library files are maintained through the execution of :ALLOT, :COPY, :DELETE, and :SQUEEZE commands. The entries in the MODIR File, MODULE File, and DEFREF File are in the same sequential order. The ith entry in the MODIR File identifies the ith object module in the MODULE File, and corresponds to the ith entry in the DEFREF File. The ordering of these files is always preserved.

**:ALLOT** Library files are allocated in the same general manner as other files described previously, but with certain specific differences. When area SP or FP is specified, a check is made to determine if the file name is MODIR, MODULE, DEFREF, or EBCDIC. If MODULE is specified, RSIZE is set to be 30 words and FORMAT set to be blocked. If MODIR, DEFREF or EBCDIC is specified, FORMAT is set to unblocked. RSIZE can be any value for the unblocked files and is used solely for calculating the amount of space to allocate for the file. The record size for these three files is set to 0 when allocated. GSIZE on all library files is ignored, and is always set equal to disk sector size by RADEDIT.

**:COPY** The permanent disk area specified on the :COPY command determines which library a module(s) is to be added to. For each object module added, the following procedure is followed:

1. An object module is read from the input device specified on the command. The module is added to the end of the MODULE File as it is being scanned for external definitions and references. The MODULE File record number for the MODIR File is obtained from RFT12 (current record no. of file). The MODIR File index is obtained from RFT5 (record length).

2. As DEFs and REFs are encountered, they are added as entries to the end of the EBCDIC File. The first DEF encountered is used as the MODULE File name. However, REFs are added to the EBCDIC File if they are not in duplicate.

3. The indices to the EBCDIC File entries are saved to create the DEF n and REF n words of the entry to the DEFREF File.

4. The addition of the object module to the library is completed by updating the "records per module" in the MODIR File entry; "entry size" in the DEFREF File entry; and writing the MODULE, DEFREF, and EBCDIC Files to the disk.

**:DELETE** The permanent disk area on the :DELETE command is used to determine which area contains the library object module to be deleted. The MODIR File entry containing the same module name as that appearing on the command is zeroed out. The corresponding DEFREF File entry is located and the halfword containing the MODIR File index is set to -1. No other changes are made to the EBCDIC and MODULE Files as a result of the :DELETE command.

All unused space resulting from a module deletion is recovered when a :SQUEEZE command is executed.

**:SQUEEZE** The area containing the library is determined from the subparameter to "LIB" from the SQUEEZE command. Only the library will be squeezed by this form of the command. A search is made of MODIR for any deleted entries. If none are found, there is no space to be reclaimed and squeeze terminates. If there are deleted entries, all remaining modules are copied from the MODULE file to the Temporary File X1.BT. Then, using X1.BT as the source input, the library files are recreated by a normal library build.

## Bad Sector Handling

Bad sectors within permanent file areas on a disk are removed from use by making special entries to the appropriate file directory. All bad sectors can be handled in this manner except these that contain a sector of the file directory. These cannot be removed from use as it would make accessing of certain files impossible.

### Command Execution

Bad sectors are handled through execution of :BDSECTOR and :GDSECTOR commands. The :BDSECTOR command removes the sectors from use by allocating a BADSECTOR entry equal to the limits of the bad sectors. The :GDSECTOR command returns the sectors for use by deleting the entry made by :BDSECTOR.

**:BDSECTOR** The permanent disk area containing the bad sectors is determined from the disk address and sector limit on the command and the BOA and EOA limits of the areas in the Master Dictionary. Specifying sector zero of the area is not allowed, and doing so will cause the command to abort. If any other directory sector is specified, an attempt will be made to move its data to another sector. The directory is searched for all files that fall within the bad sector limits. Files that begin within the limits are deleted and messages are produced to indicate which files they were. Files that terminate in, or completely contain, the bad sectors are truncated at the last good sector, and message(s) produced to warn of this condition. For extended files, all extents beyond the truncated or deleted extent are also deleted.

A BADSECTOR entry is created, either from one of the deleted files or an entirely new entry, whose name is set to -1 (X'FFFFFFFF', X'FFFFFFFF') and BOT and EOT to the bad sector limits.

**:GDSECTOR** The area to process is determined in the same way as for :BDSECTOR. The appropriate area's directory is searched for all BADSECTOR entries included within the sector limits. Entries completely within the limits are converted to deleted file entries; others are adjusted to reflect their new BOT's and EOT's.

## Utility Functions

The following utility functions are performed by RADEDIT.

- Maps permanent disk areas.
- Maps libraries.
- Clears permanent disk areas.
- Enters data onto permanent disk files.
- Appends records to the end of an existing permanent disk file.
- Copies permanent disk files.
- Copies library object modules.
- Copies the contents of a disk pack to another disk pack.
- Dumps the contents of disk files or entire disk areas.
- Saves the contents of disk areas in self-reloadable form.
- Restores disk areas previously saved.

**:MAP** The permanent disk area(s) to be mapped is indicated on the :MAP Command, with the map information being output to the device assigned to the M:LO DCB.

Each map consists of up to three sections: one section when disk areas CK, XA, or BT are mapped; three sections if any other areas are mapped. The three sections of the map are as follows:

1. Information from the Master Directory identifying the permanent disk area, starting and ending disk addresses, write protection, and device number of the disk from the Device Control Tables.

2. Information obtained from the permanent file directories concerning each file in the area; its name, format, granule size, record size, file size, beginning of file, and ending of file.

3. Information about the space remaining in the area.

Section 1 of the map has the format

```
AREA   DEVICE-   WORDS/   SECTORS/   BEGIN    END      WRITE
       ADDRESS   SECTOR   TRACK      SECTOR   SECTOR   PROTECT
zz     yyndd     sssss    ttttt      bbbbb    eeeee    w
```

where

zz identifies the permanent disk area.

yyndd is the disk that contains the permanent disk area.

sssss is the words per sector in decimal.

ttttt is the sectors per track in decimal.

bbbbb is the absolute disk address of the first sector of the area in decimal.

eeeee is the absolute disk address of the last sector of the area in decimal.

w is the write protection for the file.

P is no write protection.

F is write-permitted by foreground only unless SY key-in.

B is write-permitted by background only unless SY key-in.

S is write-permitted only if SY key-in.

X is write-permitted by IOEX only.

Section 2 of the map has the format

```
                             FLGS AREA RELATIVE  GRANULE  RECORD  FILE   APPROX  EXTEND
FILENAME.ACCOUNT XTNT             BEGIN  END     SIZE     SIZE    SIZE   RECORDS SIZE
                       ORG        SECTOR SECTOR  (BYTES)  (BYTES) (RECS) REMAIN  (SECTR)
nnnnnnnn.aaaaaaaa xxx  o fff      sssss  ttttt   ggggg    rrrrr   lllll  uuuuu   vvvvv
    .    .    .    .   . .        .      .       .        .       .      .       .
    .    .    .    .   . .        .      .       .        .       .      .       .
    .    .    .    .   . .        .      .       .        .       .      .       .
```

where

nnnnnnnn is the name of a file in the permanent disk area.

aaaaaaaa is the account number under which the file was allotted. It is not printed if it is all zeros or blanks.

xxx is the number of this extent if the file is extensible.

o is the file organization.

U specifies unblocked.

B specifies blocked.

C specifies compressed.

f are the flags.

F for FIX specified.

R if Resident Foreground.

S if written Sequentially.

D if written Directly.

ggggg is the granule size in bytes in decimal.

rrrrr is the record size in bytes in decimal.

lllll is the number of records in file in decimal.

sssss is the relative disk address of the first sector defined for the file in decimal.

ttttt is the relative disk address of the last sector defined for the file in decimal.

uuuuu is the approximate number of additional records the file can contain.

vvvvv is the number of sectors to be allotted to any extension to this file.

Section 3 of the map gives statistics on the use of the area and has the format

```
NUMBER OF FILES:          nnnnn

REMAINING SECTORS:        xxxxx
SECTORS RECOVERABLE:      yyyyy
```

where

nnnnn is the number of directory entries listed.

xxxxx is the number of unused sectors in the area; those between the end of the last allocated file and the end of the area.

yyyyy is the number of additional sectors that will become available if a SQUEEZE is performed.

The mapping of an area is performed as follows:

1. Information is obtained from the Master Directory for Section 1 of the map and output to the LO device. If an area is not allocated, the mapping of that area is ignored.
2. Information is then obtained from the permanent file directory for Section 2 and output to the LO device. If an area other than CK, XA, or BT does not contain files, a message will be output to that effect. When a bad sectors entry is encountered, " BADSECS" is printed as the name of the file.
3. As the information for each file is printed, sectors contained in deleted files or between the end of one file and the beginning of the next (truncated areas) are counted for reporting in Section 3.

The information on the Master Dictionary is unpacked by the subroutine UNPKMASD into a table. All subsequent references to MASTD information during a MAP operation then use this table. UNPKMASD also computes the number of sectors in the area and initializes values used in accounting for free space, used space, and lost space for Section 3 output.

Each file's entry in the directory is unpacked into a table as it is scanned. This table, rather than the actual entry in the directory, is used to print the information for Section 2.

As each area's map is produced, checks are made for a valid directory. Conditions tested are

1. The "Address" portion of the last directory sector is larger than a sector.

2. The "Next Available Sector" portion of a directory sector points out of the area.

3. The End sector of a file entry is beyond the end of the area.

4. The size of a file (EOF - BOF) < Ø.

Whenever any of these conditions are found, the processing of the area is terminated by the message

```
AREA HAS AN INVALID DIRECTORY
```

**:LMAP** This command functions only on libraries in the SP and FP areas.

The output map has the format

```
MAP OF LIBRARY IN AREA aa

MODULE NAME    LOCATION    DEFS                  REFS

mmmmmmmm       llll        dddddddd  dddddddd    rrrrrrrr  rrrrrrrr
   .             .            .         .           .         .
   .             .            .         .           .         .
   .             .            .         .           .         .
```

where

aa is the permanent disk area that contains the library.

mmmmmmmm is the object module name.

llll is the relative sector address of the first sector of the object module.

dddddddd is the name of an external definition (up to three per line).

rrrrrrrr is the name of an external reference (up to three per line).

If the area contains no library, the message

```
AREA CONTAINS NO LIBRARY
```

is output.

**:SMAP** This command functions similarly to the :MAP function except that the output is greatly abbreviated for output to a terminal.

Section 1 of the map has the format

AREA: zz

Section 2 of the map has the format

```
#RECS   FILENAME.ACCOUNT
IIIII    nnnnnnnn.aaaaaaaa
```

The mapping of the area is performed in the same steps described under :MAP.

**:CATALOG** The :CATALOG command uses the utility routine GETFID to get its input parameter and decide what type of :CATALOG command is given. If a file name is given, it is assumed to be the first of a list of individual files, specifically named, that are to be processed. This is Format "A" catalog. If no file name is given, it is Format "B" wherein files to be processed are selected based on area and/or account.

Format "A" processing.

Immediately on determining that it is a Format "A" command, the header

ORG #RECS NAME

is output.

For each file listed, the total number of records in the file and all its extents is determined and the files organization, number of records and FILENAME.AREA.ACCOUNT output.

Format "B" processing.

This format may select all accounts in a particular area, (TYPE 2) a particular account in all areas, (TYPE 1), or a particular account in a particular area (TYPE 0). Based upon which is requested, a list of all areas to be scanned is built in the AREASWS table.

All areas specified are scanned in order of ascending area index. Area information is gathered by UNPKMASD, each file entry in the directory by GET1SFIL/GETNXFIL, and the status and information in an entry by UNPKDIRE. As each new file name in the proper area, account and file name limits is found, it is added to a linked bit of files created in the background buffer space. This list is created and kept in order alphabetized by file name by account and then by area index. As each extent of an extended file is found, its file size is added to the accumulated total in the list.

A list entry has the format

| Word | | | Index |
|---|---|---|---|
| 1 | File Name | | Ø |
| 2 | | | 1 |
| 3 | Account Name | | 2 |
| 4 | | | 3 |
| 5 | Area Index | ORG | 4 |
| 6 | FSIZE | | 5 |
| 7 | Back Link | | 6 |
| 8 | Forward Link | | 7 |

The output produced varies according to the type of :CATALOG requested, the type is maintained in the cell MAPSW. It is used to branch to the proper code to produce the required header, and again to decide whether the area and/or account is to be displayed.

**:CLEAR** The permanent disk area on the :CLEAR command is used to determine the area to be cleared (set to zero). The area is cleared using the direct access method. The granule size is set equal to the amount of unused background space available, which is zeroed out and written to the disk.

**:COPY** The parameters on the :COPY command are used to set up the F:SI and F:SO DCBs. Files are copied sequentially. When an !EOD, :EOD, or EOT is encountered, the number of files to copy is decremented. If there are no more files to copy, the request is terminated; otherwise, the next file-copy is started. When an object module is copied to an output device, the COPY is terminated when the module end load item is encountered.

**:DPCOPY** The parameters in the :DPCOPY command are used to set up input and output DCBs which are assigned directly to the specified disk packs. The copy is double buffered on input and output using buffers that are as large as the background work space will allow. The copy continues until the specified number of sectors have been copied.

**:DUMP** The permanent disk area or file to be dumped is indicated on the :DUMP command. The information is dumped to the device assigned to the M:LO DCB. The file dump has the format

```
DUMP OF FILE nnnnnnnn IN AREA AA

RECORD rrrr

WD 0000 dddddddd   dddddddd   ...dddddddd

WD 0008    .           .           .

WD 0016    .           .           .
```

where

nnnnnnnn is the name of the file.

AA identifies the permanent disk area (area BT inclusive).

rrrr is the relative record number and begins with 1.

dddddddd is a data word in hexadecimal.

The area dump has the format

```
DUMP OF AREA ZZ

SECTOR  ssss

WD0000  dddddddd   dddddddd   ...dddddddd

WD0008     .           .           .

WD0016     .           .           .
```

where

ZZ identifies the disk area.

ssss is the relative sector number, and begins with 0.

dddddddd is a data word in hexadecimal.

## Access Control Image (ACI)

Purpose

The ACI contains an image of the access protection codes for a given secondary task.

Type

Serial consecutive entries in TSPACE.

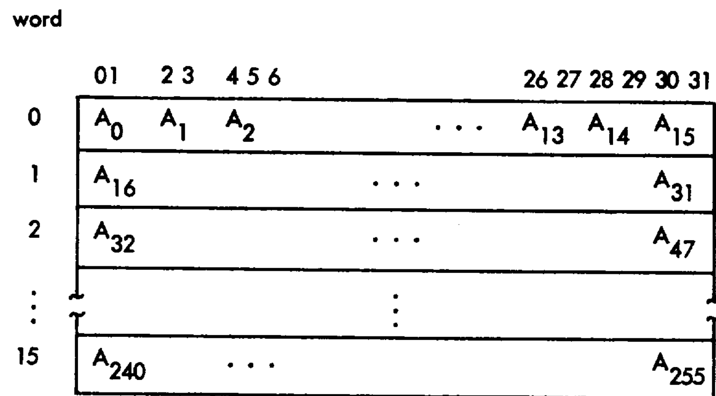
Logical Access

The ACI is pointed to from the STCB. Entries are accessed by index displacement with entry 0 representing virtual page 0.

Overview of Usage

The ACI is created by task initiation and is filled in by the Task Dispatcher before each dispatch operation from information in the tasks segment descriptors. The ACI is also manipulated by Memory Management routines.

Access Control Image (ACI) Format




where

$A_i$ is the 2-bit access protection code for virtual page i.

The dumping of an area or file is performed as follows:

1. The directive is scanned to determine whether an area or file is to be dumped. If a value for SREC is not specified, 0 is assumed. If a value for EREC is not specified, the last record of the file or area is assumed.

2. The record(s) to be dumped is accessed sequentially. Within a record, if a word is duplicated more than sixteen times in order, it is output only once in the message

   | 'WDxxx THRU xxx CONTAIN xxxxxxxx' |
   |---|

   If records are duplicated, the message

   | 'RECORDxxx THRU xxx CONTAIN xxxxxxxx' |
   |---|

   is output.

   If sectors are duplicated, the message

   | 'SECTOR xxx THRU xxx CONTAIN xxxxxxxx' |
   |---|

   is output.

3. The dump is terminated when the specified number of records have been dumped or when a complete file or area has been dumped.

**:XDMP** The specified input is displayed on the device assigned to the M:LO DCB. The input may be a file, a disk, a tape, a card reader, or any other valid input device. Files and disks are read in a sector by sector mode; tapes by physical blocks; and other devices by records. For input coming from a disk, the read is limited to sector size. For all other input, the read is for 65536 bytes (the maximum possible in READ CAL) or the size of the background buffer space, which ever is smaller.

Each XDMP output starts on a new page. Each page of output is headed by a title line that gives the name of the device, file or area being processed, and the oplabel if accessed through one. Each sector or record is processed individually. A sample of the output is given below. The two addresses on the left are the byte (and word) displacement from the beginning of the block to the first byte (word) on that line.

Example: XDMP output

```
 FILE   BNX     .PA                         76 AUG 17


   SECTOR     0   LENGTH =  1024 (   400) BYTES

 BYTE  (WORD)    0 (0/8)   4 (1/9)   8 (2/A)   C (3/B)   10 (4/C)  14 (5/D)  18 (6/E)  1C (7/F)   *0   4   8   C   10  14  18  1C  *
 00000  00000    FFFFFFFF  FFFFFFFF  00060000  000003E8  00000400  0A000000  07000000  040A0A00   *...............Y................*
 00020  00008    00000BB8  00000414  0A000000  0FA00000  041E0A00  000013B8  00000428  0A000000   *................................*
 00040  00010    17700000  04320A00  00000000  00000000  00000000  00000000  00000000  00000000   *................................*
 00060  00018    00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000   *................................*
          BYTES 00080 TO 003DF IDENTICAL TO ABOVE LINE; 00864 (00360) BYTES, 00216 (000D8) WORDS.
 003E0  000F8    00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000   *................................*


       END OF XDMP
```

When there are three or more lines of output having the same contents, only the first line is displayed. The other lines are replaced by the "identical to above line" message seen in the sample output. The first two numbers are byte displacements of the first and last bytes of the duplicate line. The length of the duplicate information is given in decimal and hexadecimal bytes, then in decimal and hexadecimal words.

**:SAVE** . The area(s) to be saved is specified on the :SAVE command. The data is dumped to the device assigned to the M:BO DCB, and consists of the following:

1. A small 88-byte bootstrap that loads the large bootstrap when booted from the console.
2. A large bootstrap that restores the disk from magnetic tape.
3. An 88-byte RBM bootstrap used for booting the disk.
4. Records containing data to be saved.

Each record to be restored is preceded by a six-word header with the format:

| 0 f18 15 | 16 | 17 | 18 | 19 | 20 23 | 24 31 |
|---|---|---|---|---|---|---|
| WPS | LRA | LRT | MRT | DPA | SEQ | Area index |
| SPR | Device address | | | | | |
| TPC / FWA | | | | | | |
| SPT | Area name | | | | | |
| NSZ | | | | | | |
| CKSM | | | | | | |

where

| | |
|---|---|
| WPS | is the number of words per sector. |
| LRA | is a flag to indicate this is the last record of an area. |
| LRT | is a flag to indicate this is the last record of the tape. |
| MRT | is a flag to indicate the save data is continued on another volume. (The LRT flag will also be set, and SPR and NSZ will be zero. It is followed immediately by double tapemarks.) |
| DPA | is a flag to indicate the area is on a disk pack. |
| SEQ | is the volume sequence number. It starts with zero and counts modulo 16. |
| Area index | is the Master Dictionary index of the area to which the record belongs. |
| SPR | is the number of sectors of data in the record that follows this header. If zero, there is no following data record. |
| Device address | is the physical device address from which the data was read. |
| TPC | is the number of tracks per cylinder for the device. It is used for restoring areas to disks by the bootstrap. |
| FWA | is the absolute disk sector number where the data records should begin being restored. |
| SPT | is the number of sectors per track for the device. It is used for restoring areas to disk by the bootstrap. |
| Area Name | is the two character EBCDIC name of the area to which the data record belongs. |
| NSZ | is the number of sectors of zeros to write preceding the data record (if any) that follows. |
| CKSM | is the checksum of this record in 2's complement form. |

The saving of an area for subsequent restoration is performed as follows:

1. A small and large bootstrap are written with their checksums.
2. A header for the CP-R disk bootstrap is written. The FWA and device number for the header is obtained from K:RDBOOT.
3. The image of the CP-R disk bootstrap is read from the file RADBOOT in the SP area, and written.
4. Data records are written with each record being preceded by a header and followed by a checksum. Leading and trailing zeros of a record are not written. Size of the data records depends upon the amount of available background space used as a buffer.
5. After all the specified areas are saved, the tape is verified by using the checksum word of each header and data record. If no checksum errors are found, the message 'SAVE TAPE OK' is printed.

Since :SAVE makes no attempt to interpret the directory in an area, it will attempt to read and save bad sectors even if they have been removed from use by a :BDSECTOR command. Normally :SAVE reads as many sectors at a time as will fit in the background buffer and processes these as a unit and writes the non-zero data to the tape. If the READ service call reports an unrecoverable error, :SAVE enters the error mode for that group of sectors. In this mode, all the background buffer is first filled with the doubleword data string C'LOSTDATA', and then each sector is read individually one behind the other. If no errors are detected, :SAVE leaves the error mode with no message. If errors are detected again, the first and last sector numbers getting errors are saved until the read-by sector is complete. One of the messages below output to M:LO.

DATA IN SECTOR xxxxx MAY BE LOST IN AREA zz

DATA IN SECTOR xxxxx TO xxxxx MAY BE LOST IN AREA zz

At the end of the save tape build, if any such messages have been printed, the message

WARNING: ERRORS WRITING SAVETAPE. CHECK LISTING

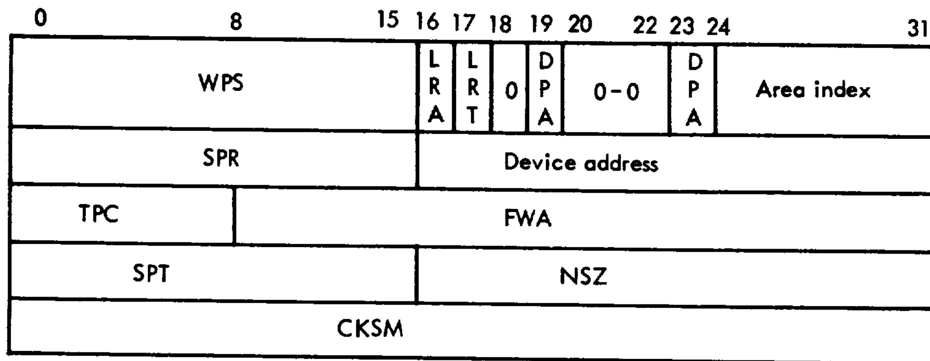
is output to M:LL and to M:OC in an ATTEND symbiont system. These messages do not cause processing to stop.

**:RESTORE** The area(s) to be restored is specified on the :RESTORE command. The data is read using the device assigned to the M:BI DCB. The small bootstrap, large bootstrap, and CP-R disk bootstrap are skipped. Data records are read and restored using the headers that precede them with all leading and trailing zeros of a record also being restored. Restoration has to be made to the same type of disk as that from which the records were saved.

The names of all areas to be restored are stored in the AREASWS table. The input tape is read once. As each area on the tape is found, it is looked up in the AREASWS table and, if requested, restored and marked as such in the table. Areas on tape are identified by name, not index. If the area index on tape does not match the current index for the area name, a warning is generated, but the restore is done for the named area anyway. Whenever all requested areas have been found, RESTORE terminates. When the end of the tape is reached, the table is scanned to ensure all explicitly named areas were restored. If any were not, an error message is given. Areas requested by ALL do not produce an error message if they are not found.

The :RESTORE processor is also able to process pre E00 format :SAVE tapes. These format tapes are distinguished by having a 5 word header record rather than a 6 word header.

This header has the form




where the fields are the same as in the 6 word header. Whenever :RESTORE reads a 5 word header, it will reformat it to the six word form. The area name currently having "Area index" will be inserted as the name; therefore changes in the index value for an area will not be detected.

Flowcharts of various RADEDIT commands are illustrated in Figures 65 through 70.

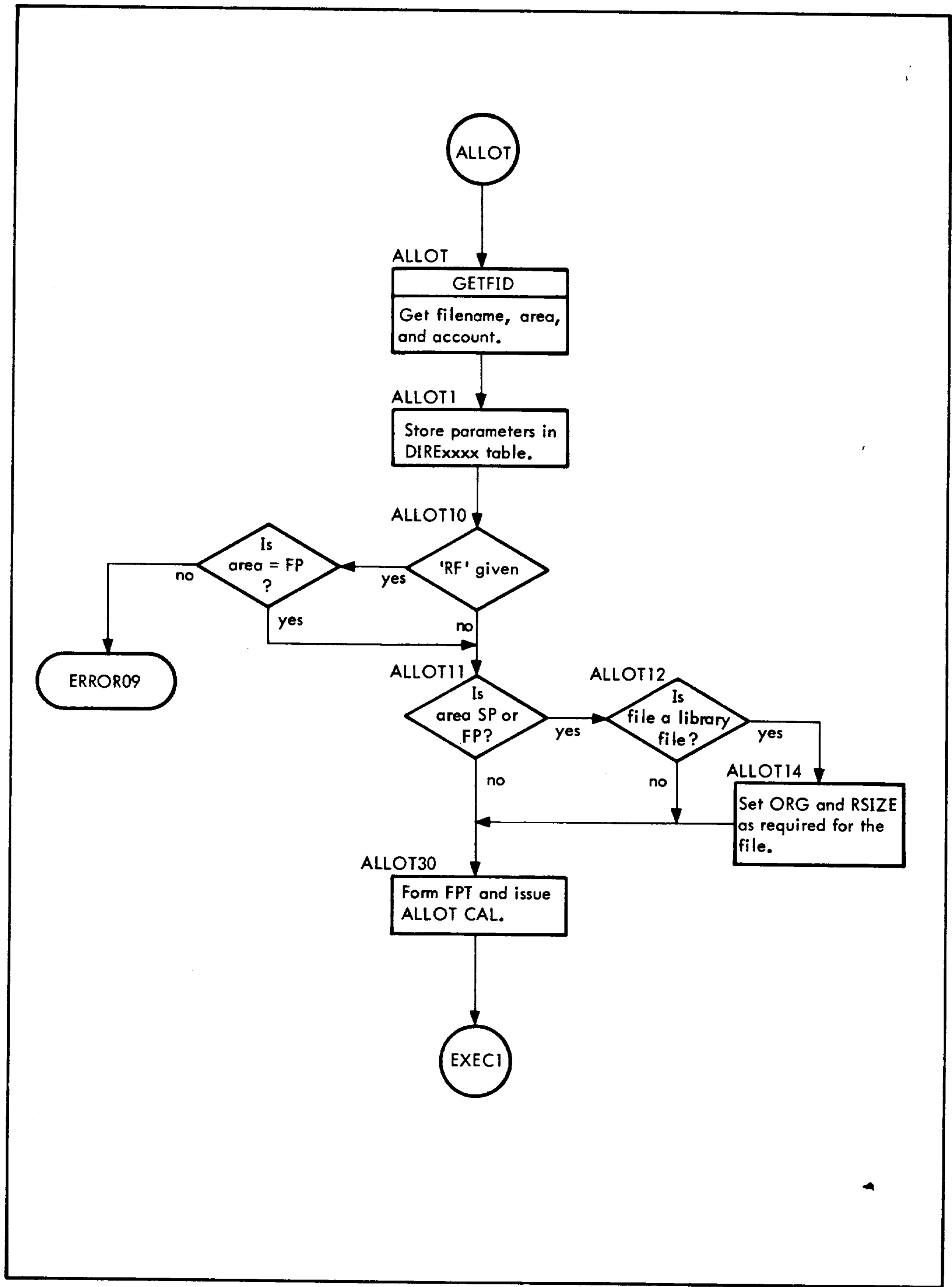



Figure 65. RADEDIT Flow, ALLOT

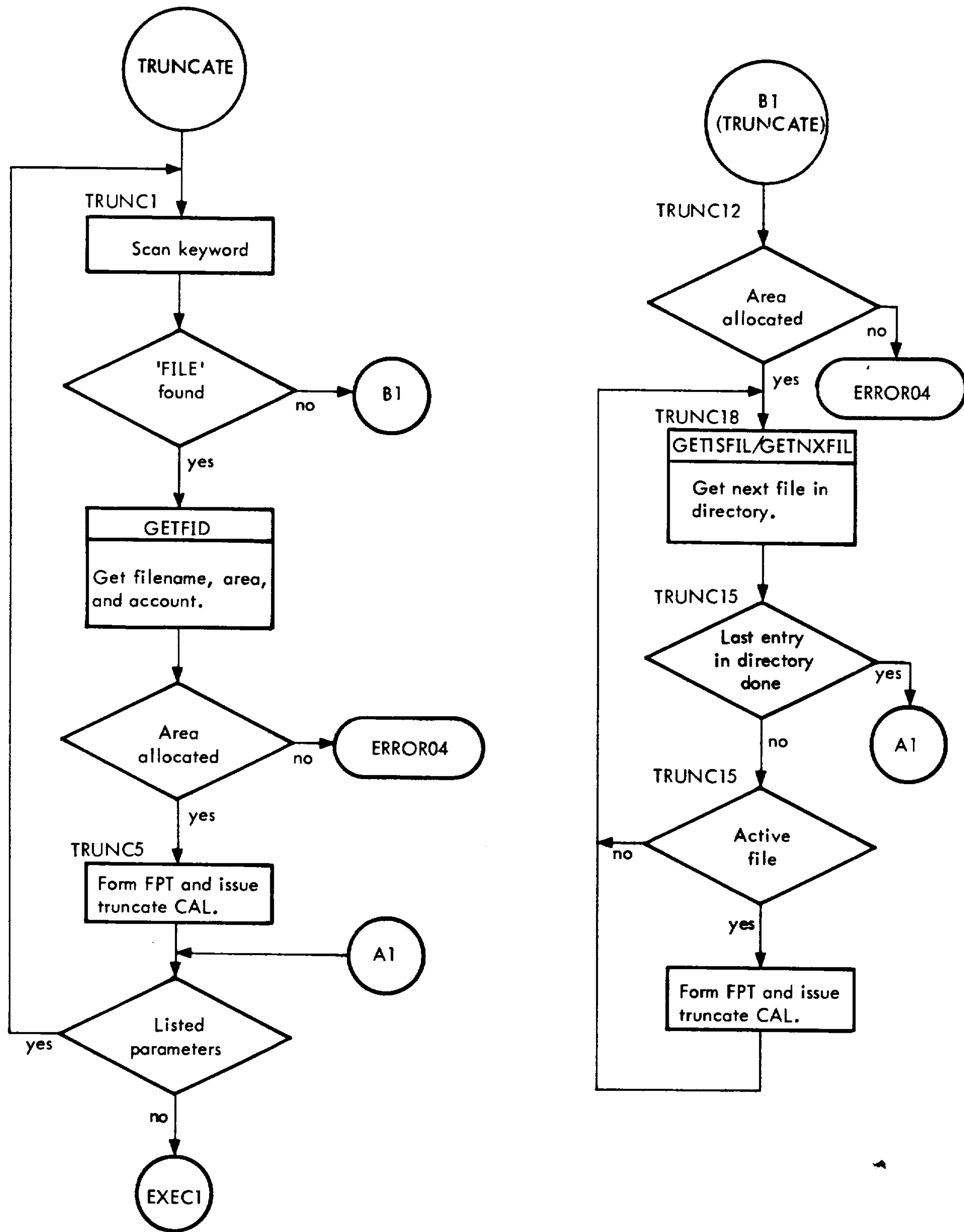



Figure 66. RADEDIT Flow, TRUNCATE

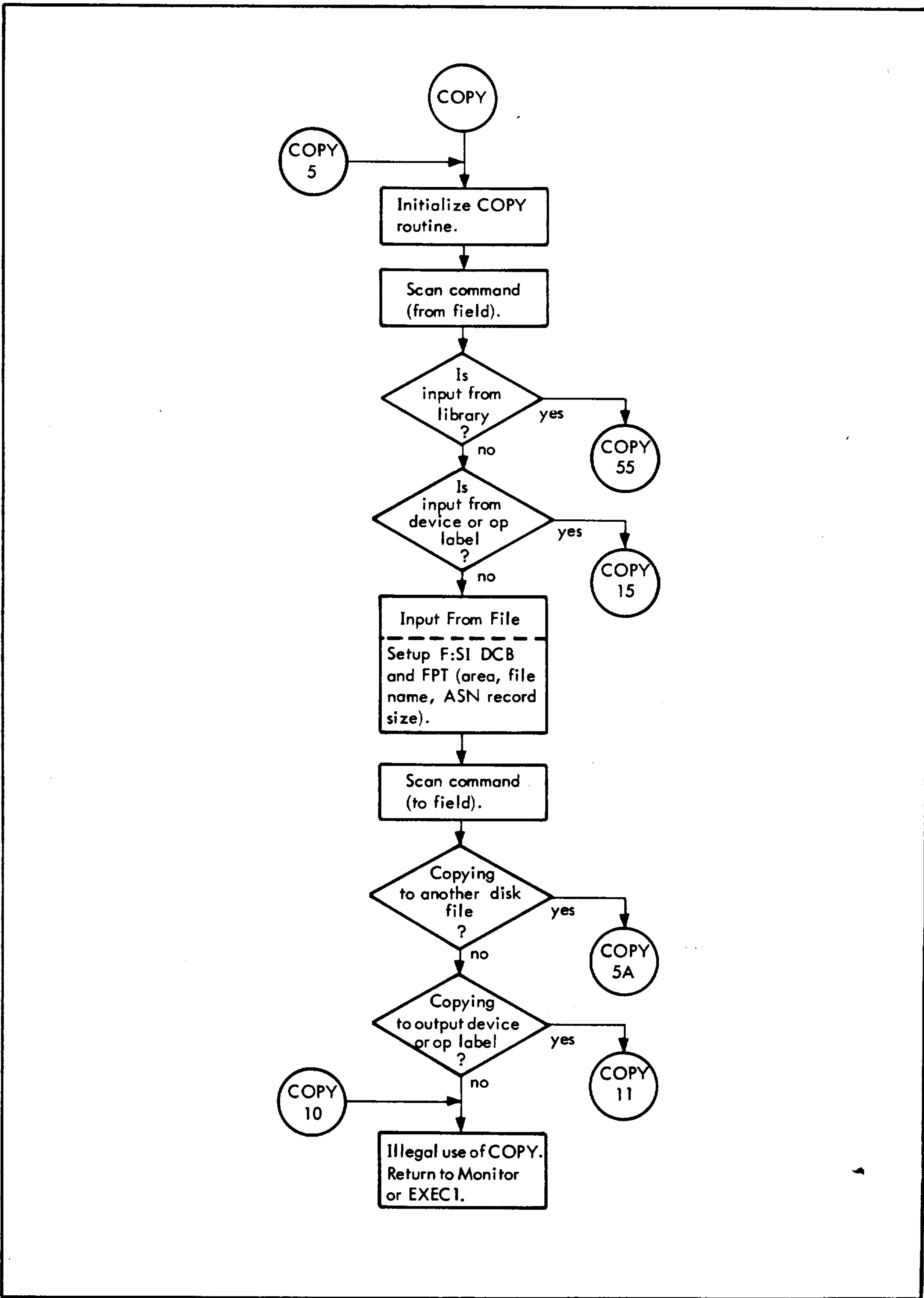



Figure 67. RADEDIT Flow, COPY

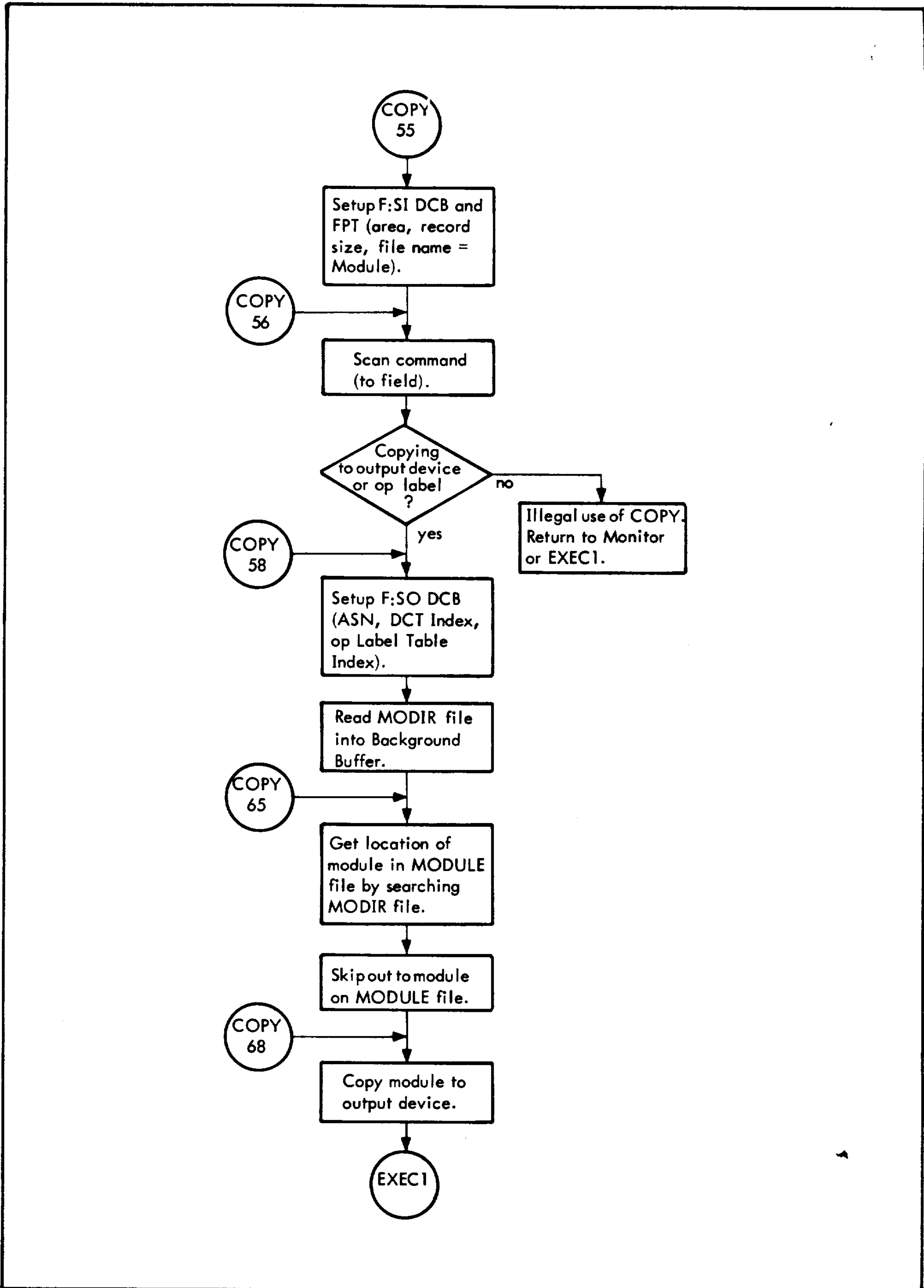



Figure 67. RADEDIT Flow, COPY (cont.)

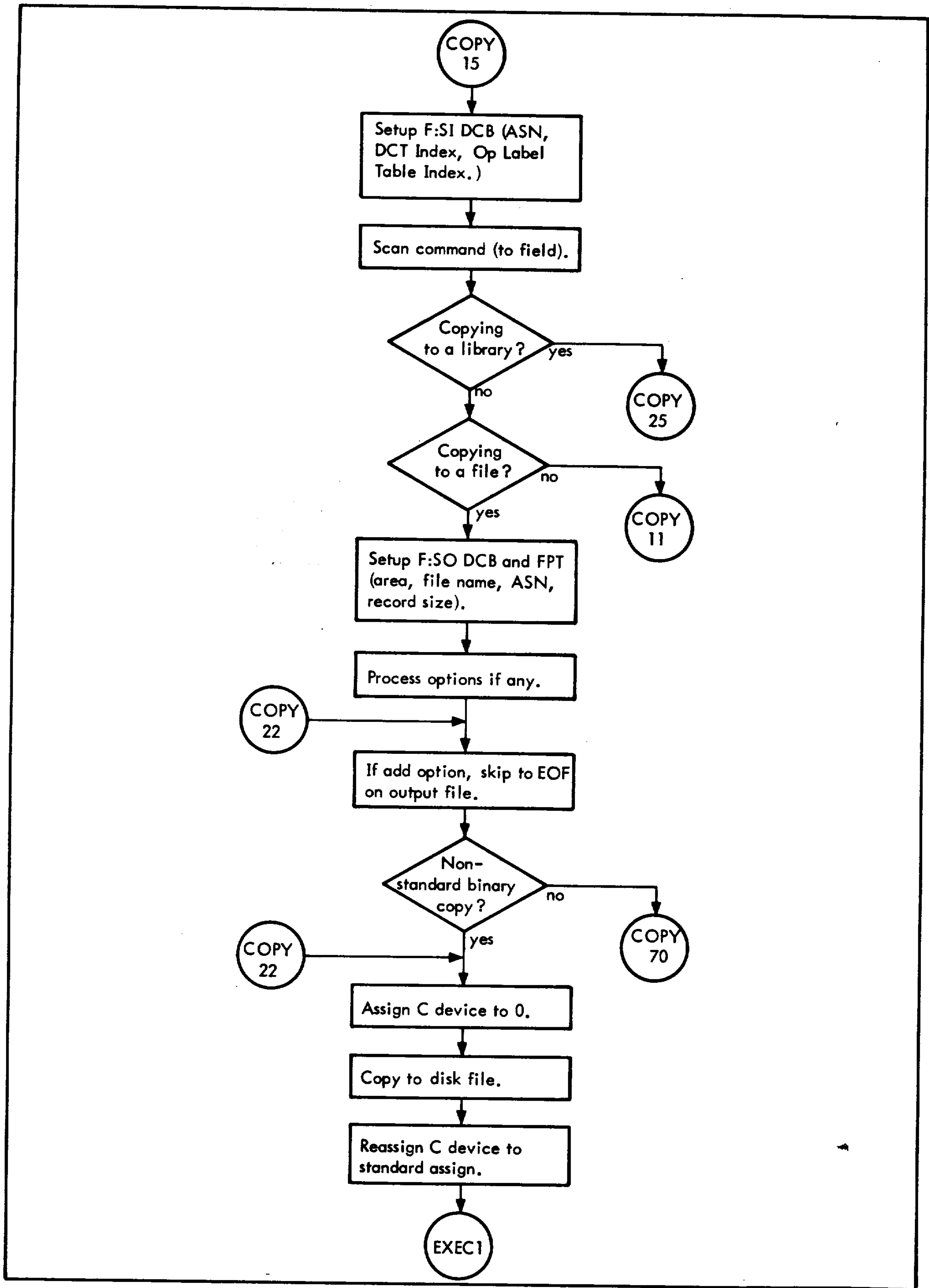



Figure 67. RADEDIT Flow, COPY (cont.)

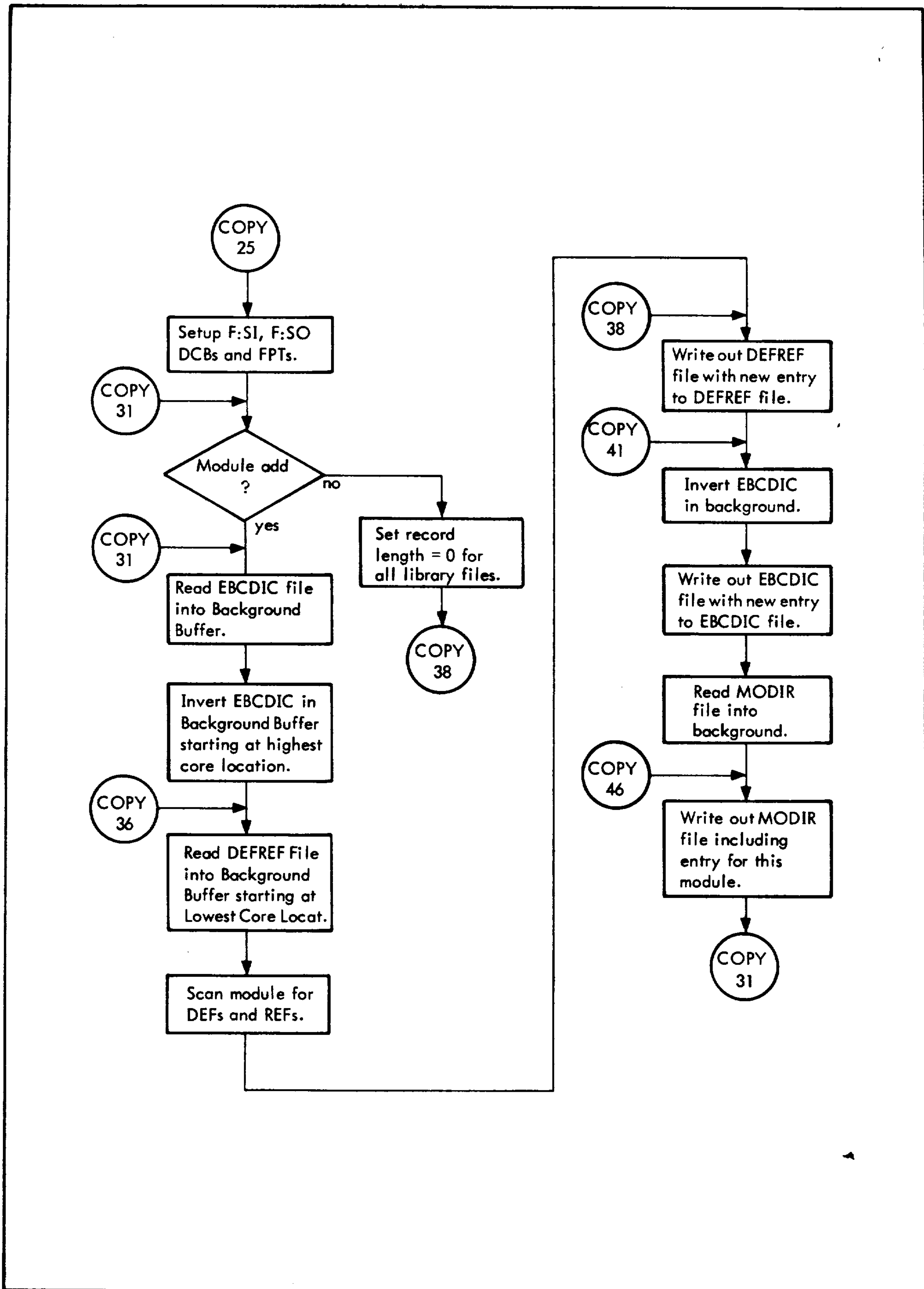



Figure 67. RADEDIT Flow, COPY (cont.)

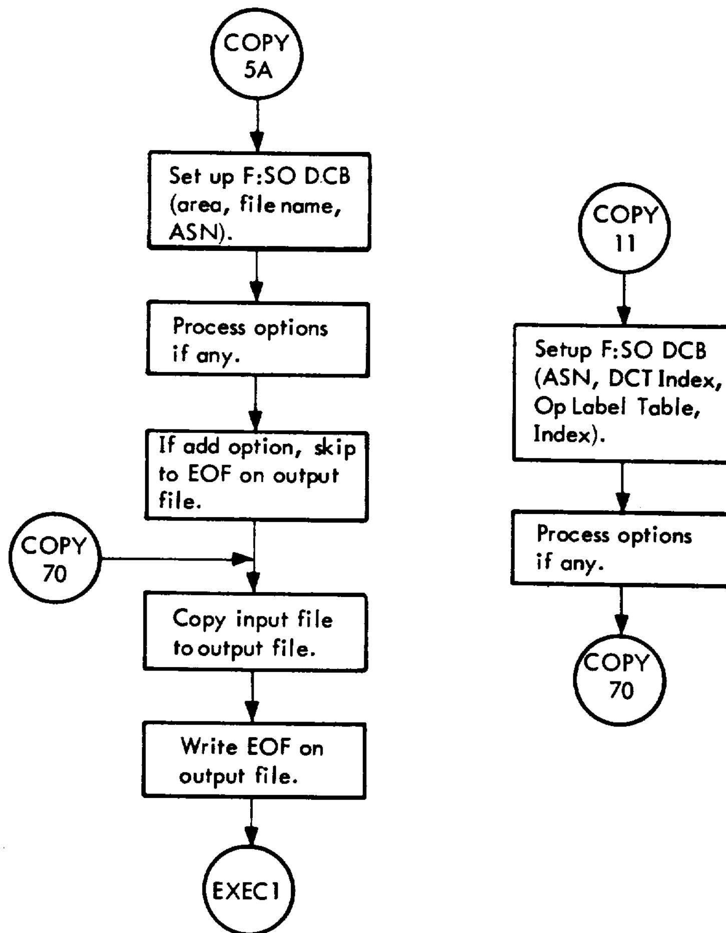



Figure 67. RADEDIT Flow, COPY (cont.)

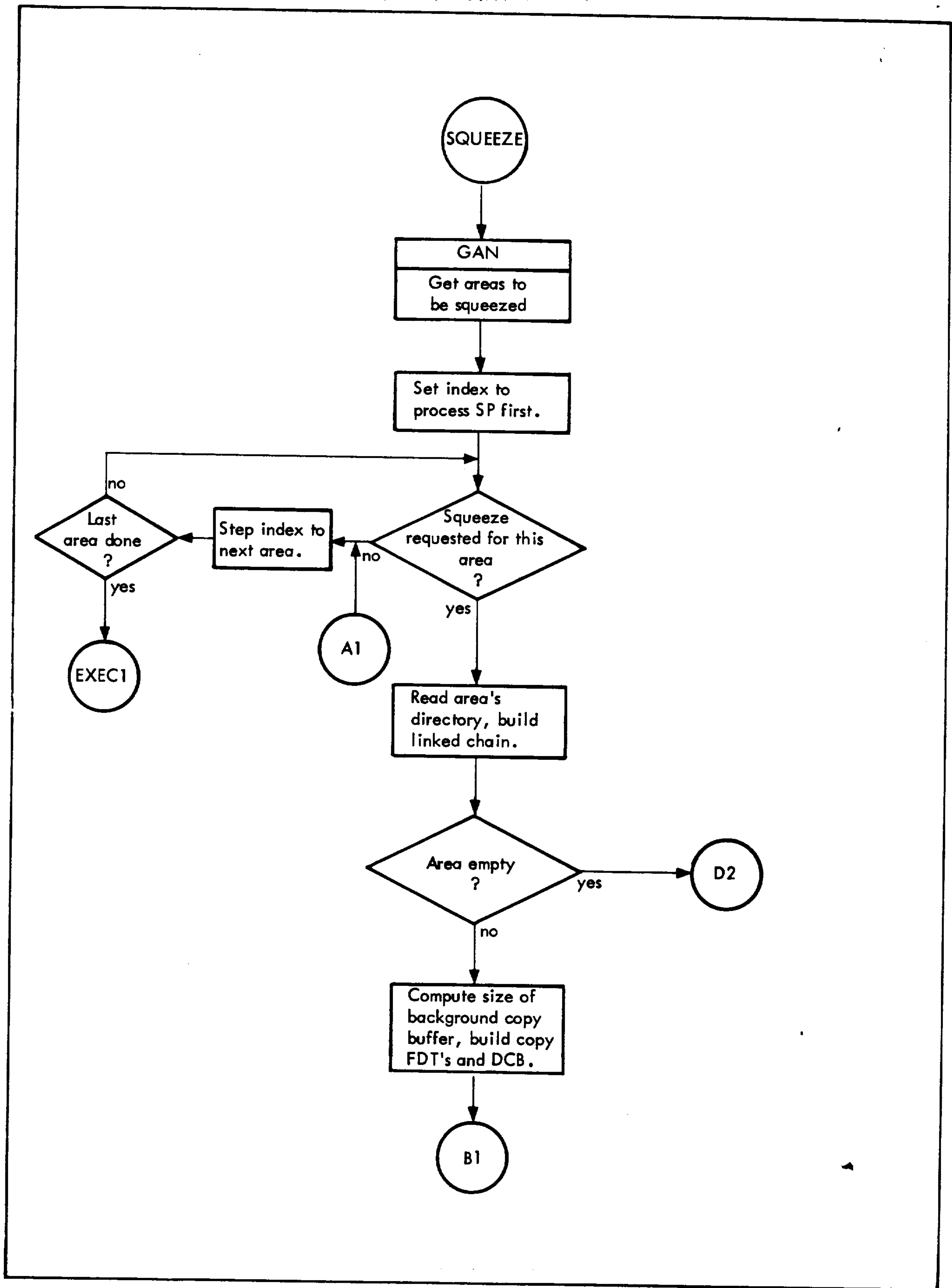



Figure 68. RADEDIT Flow, SQUEEZE

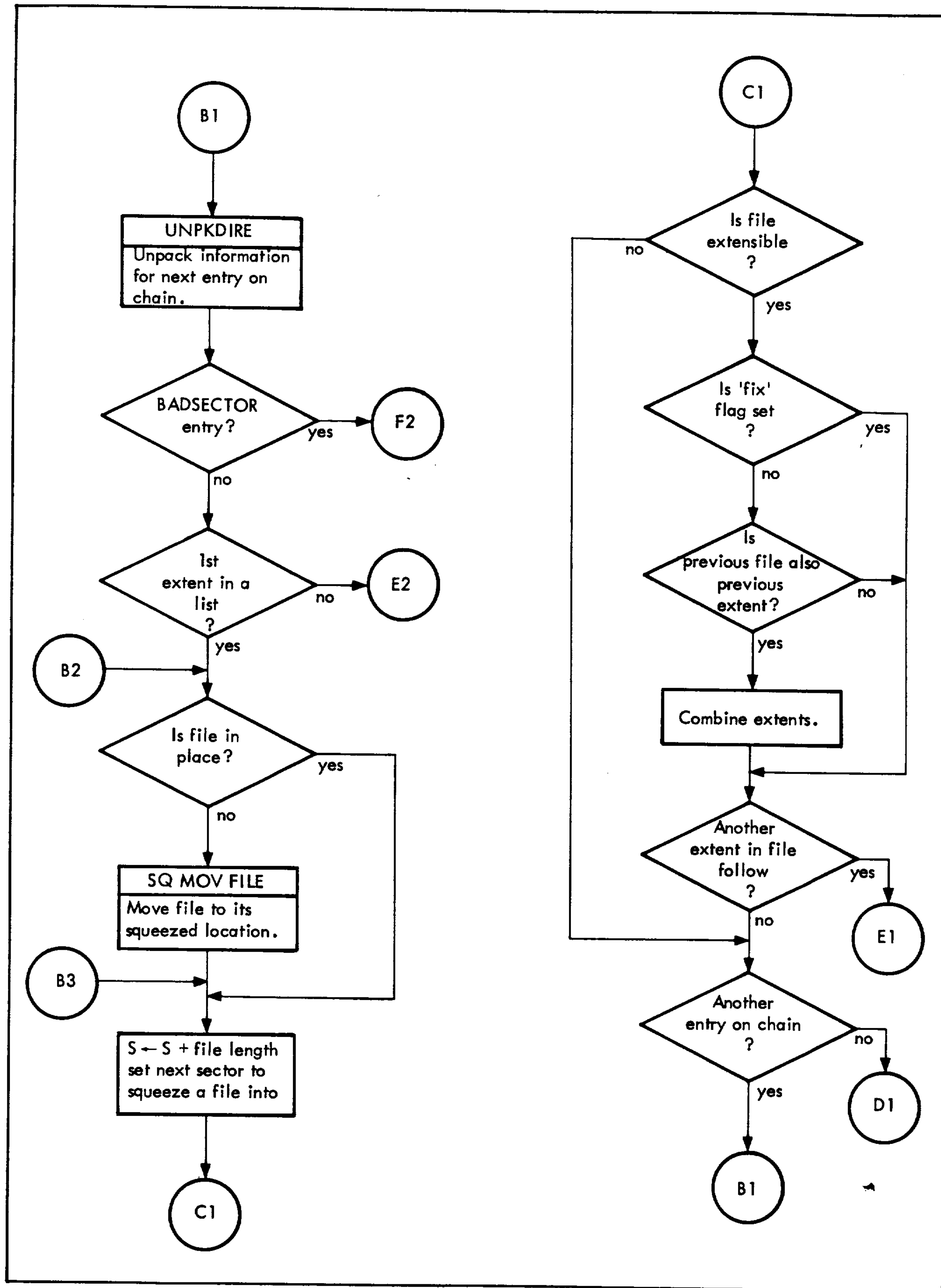



Figure 68. RADEDIT Flow, SQUEEZE (cont.)

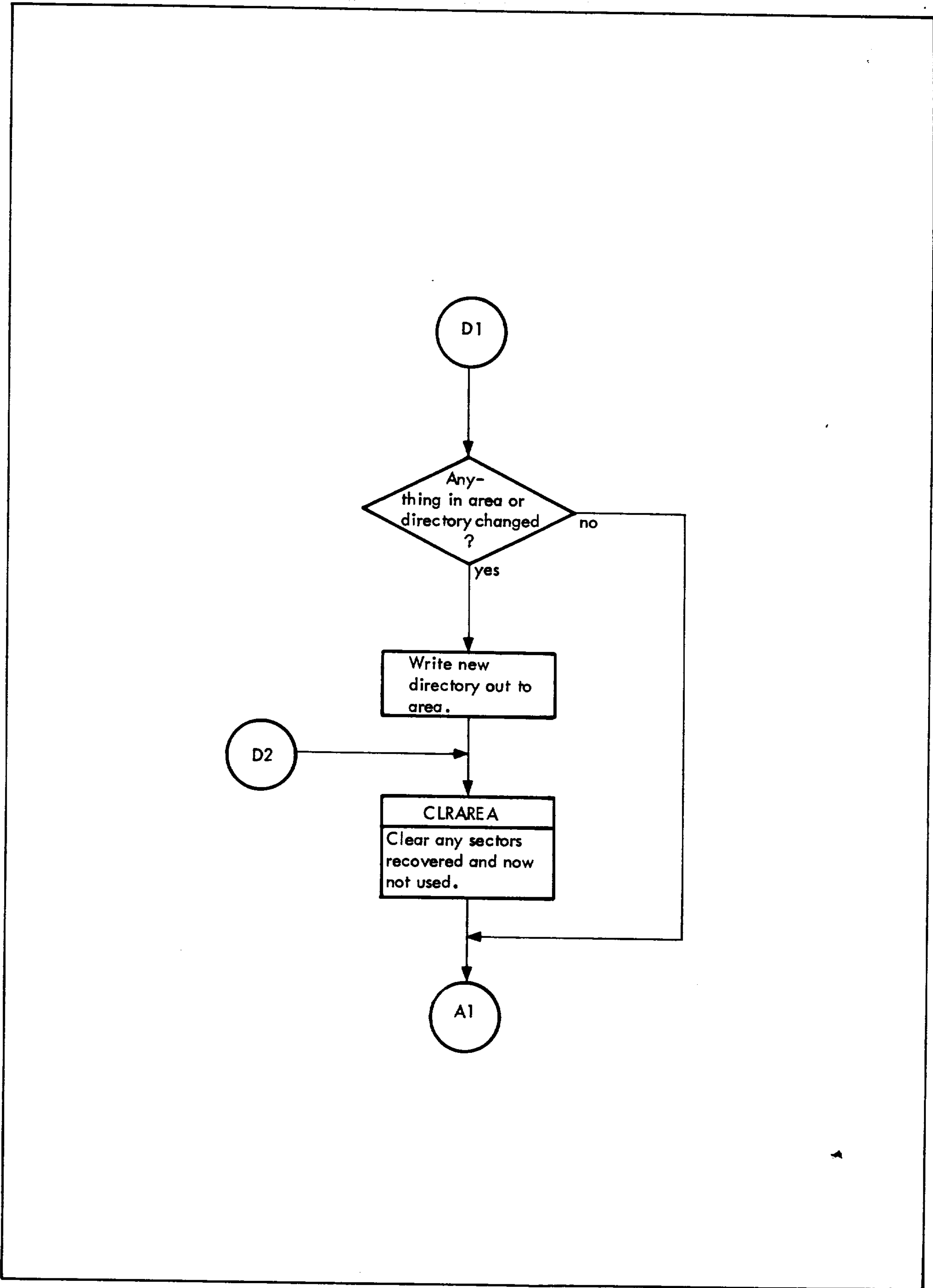



Figure 68. RADEDIT Flow, SQUEEZE (cont.)

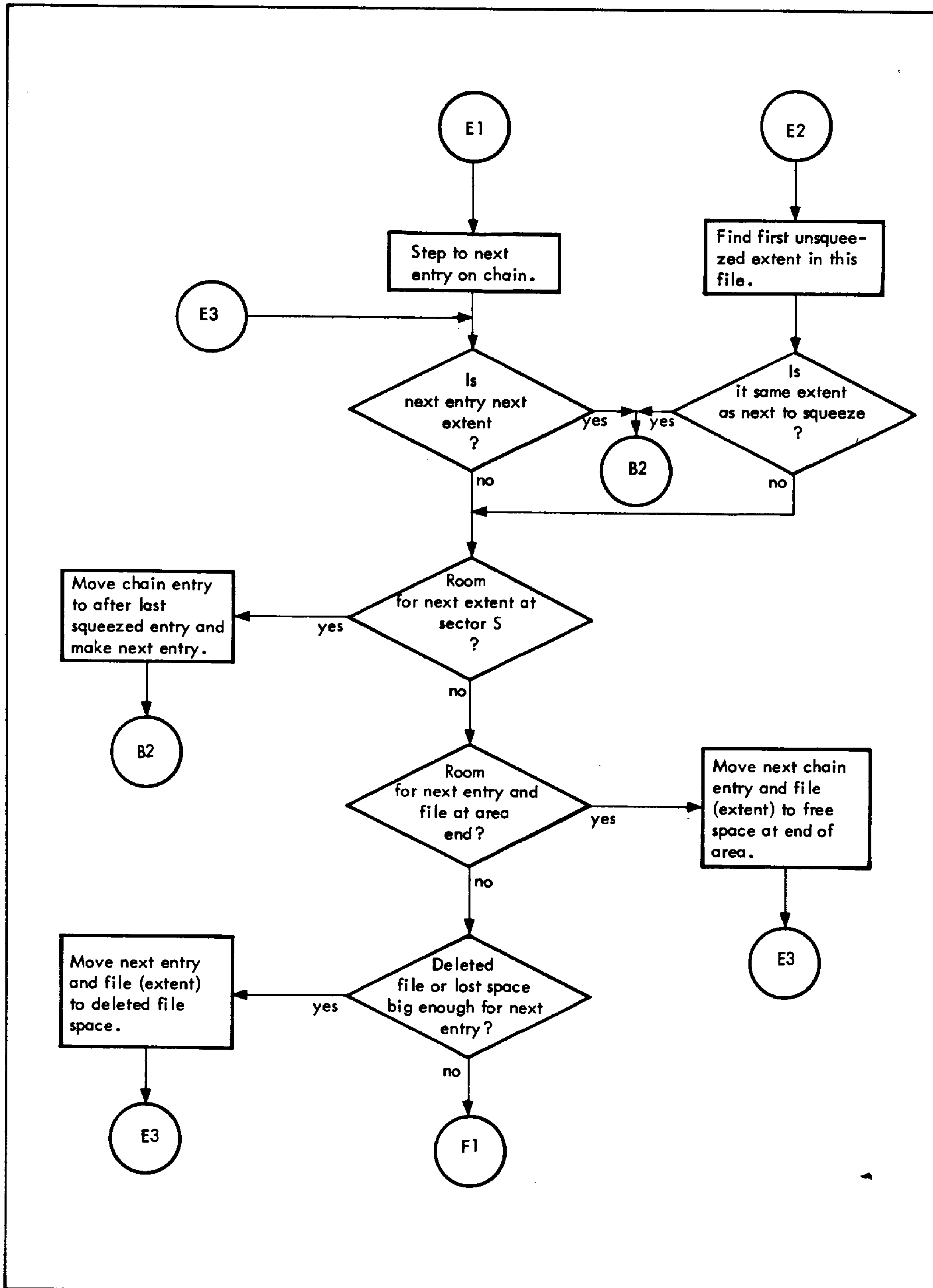



Figure 68. RADEDIT Flow, SQUEEZE (cont.)

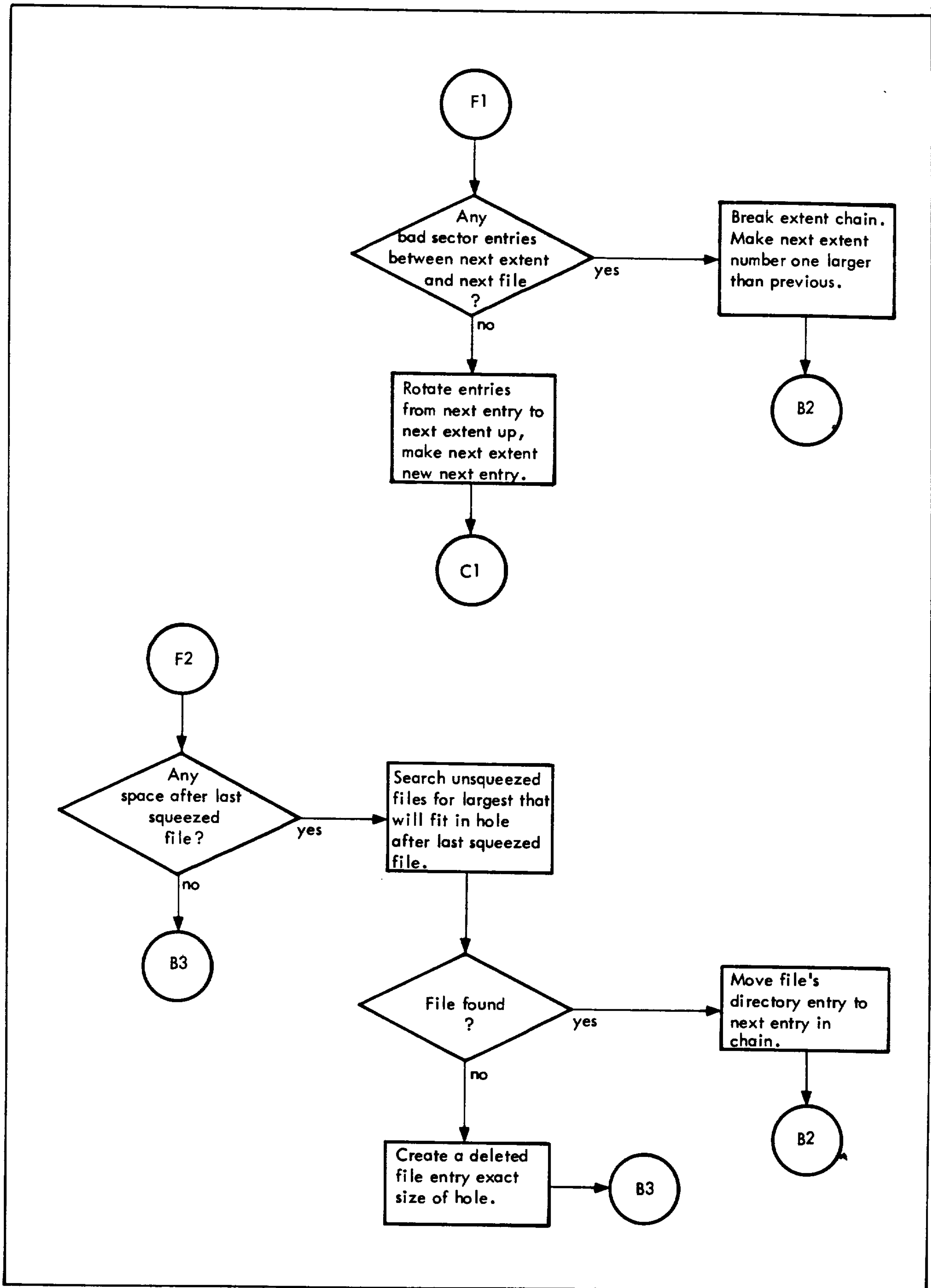



Figure 68. RADEDIT Flow, SQUEEZE (cont.)

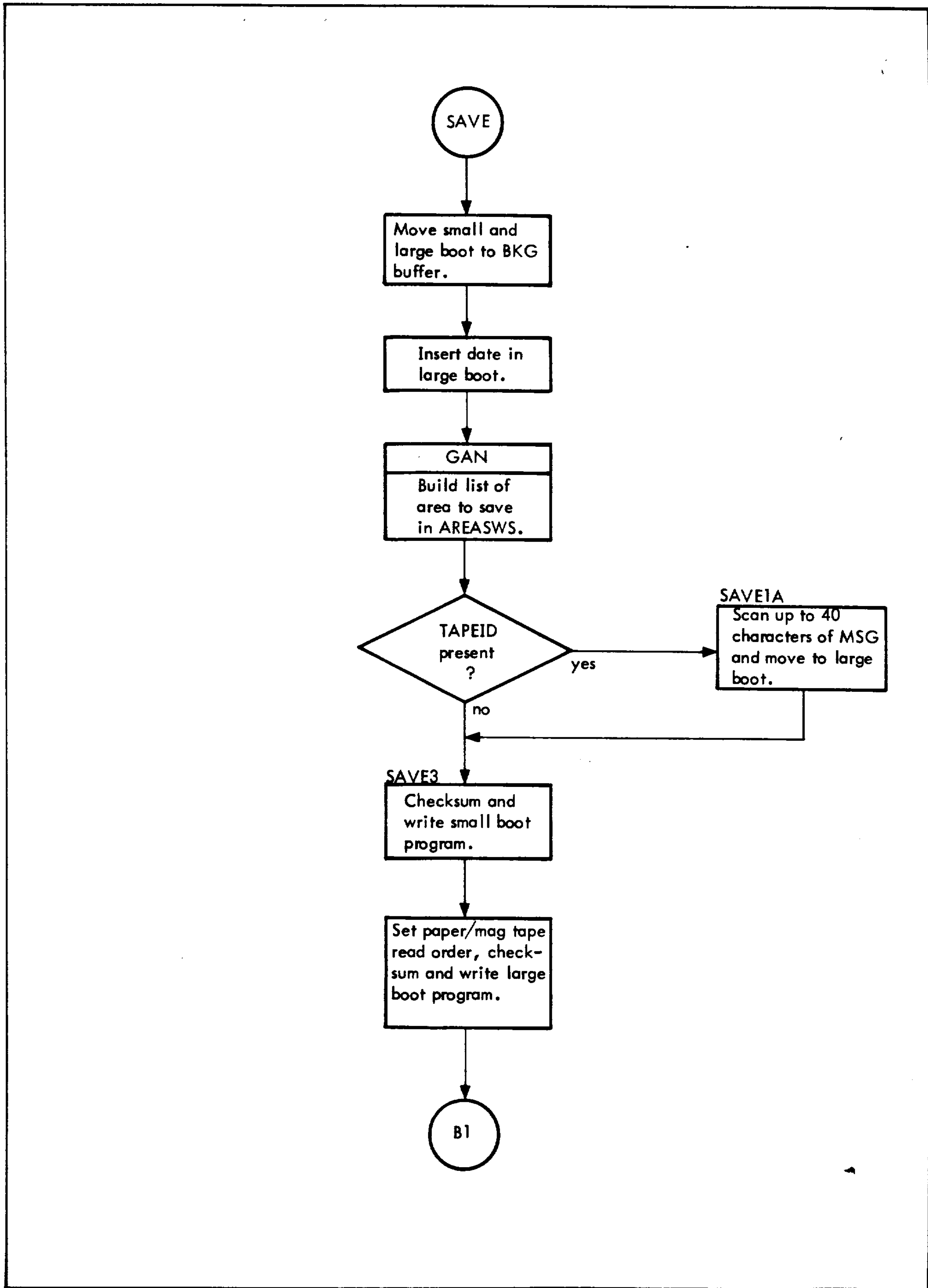



Figure 69. RADEDIT Flow, SAVE

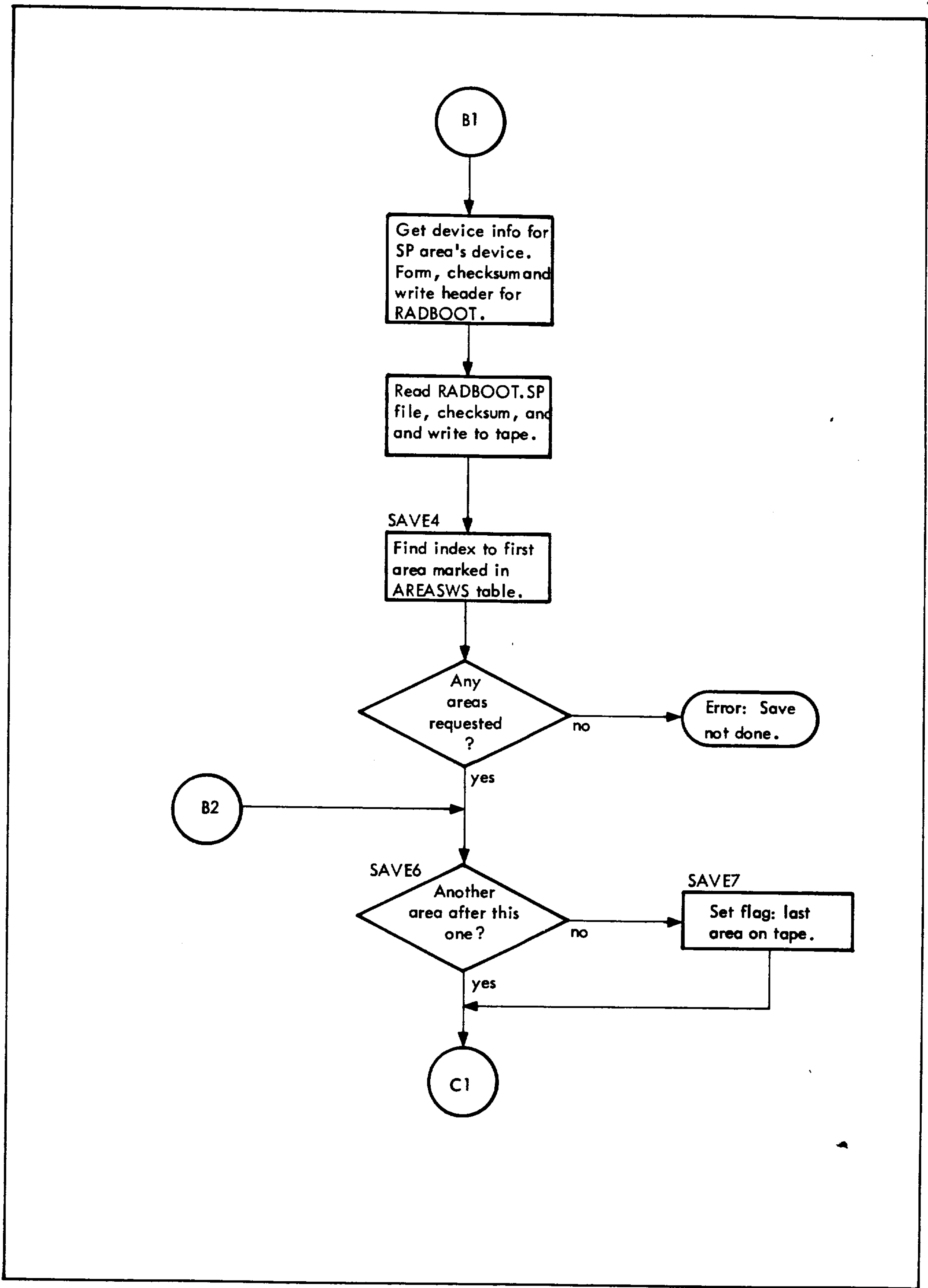



Figure 69. RADEDIT Flow, SAVE (cont.)

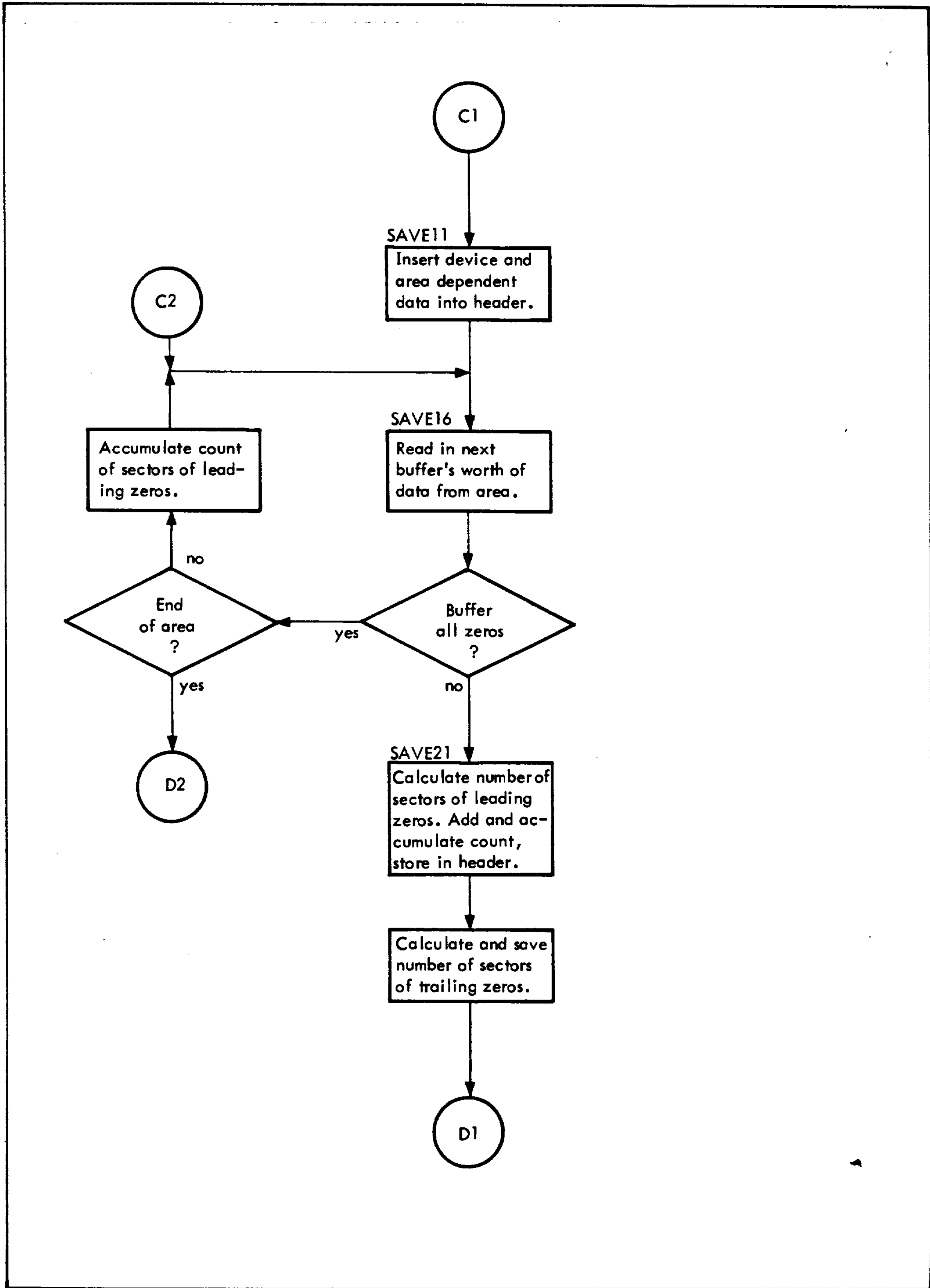



Figure 69. RADEDIT Flow, SAVE (cont.)

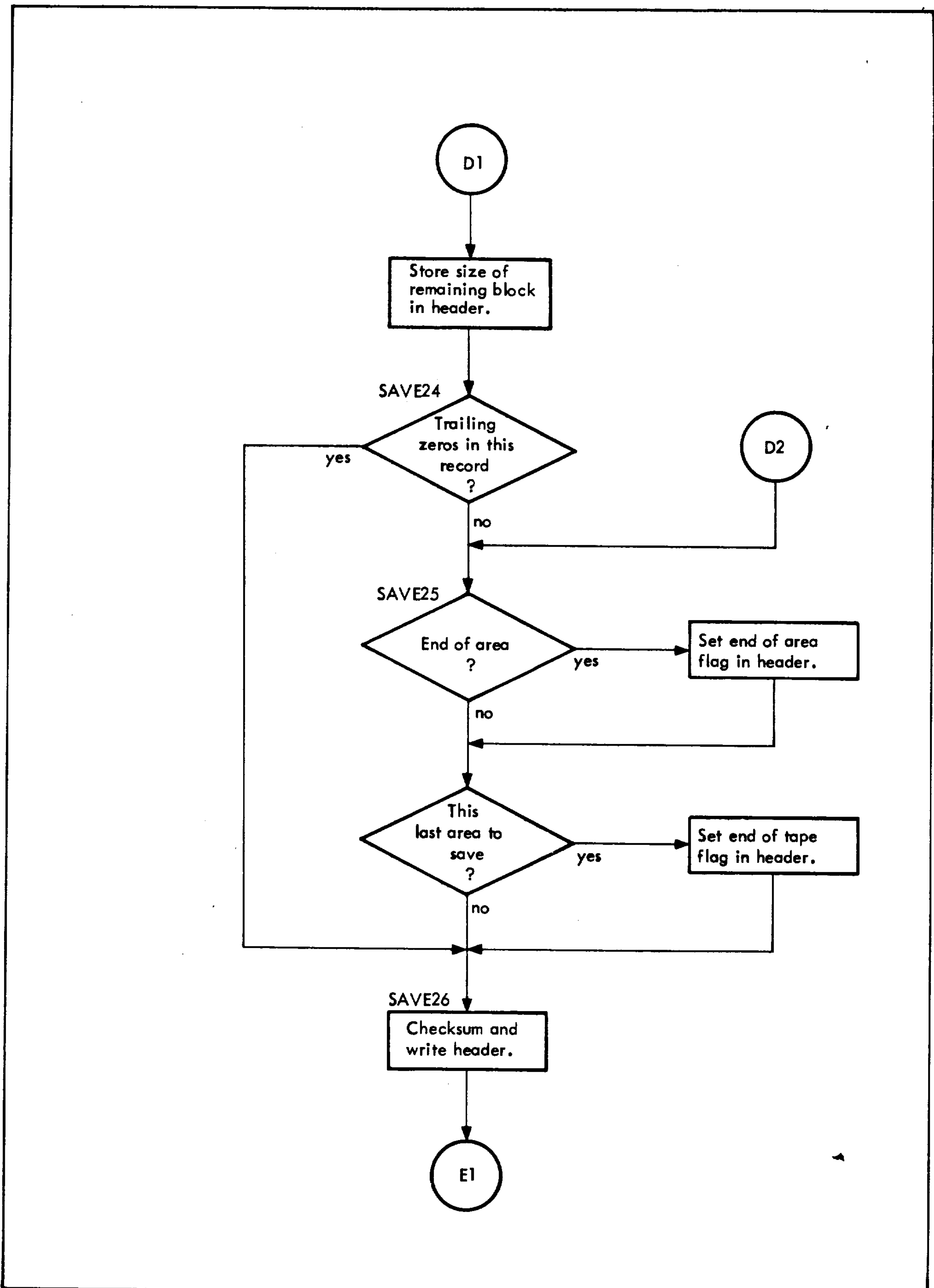



Figure 69. RADEDIT Flow, SAVE (cont.)

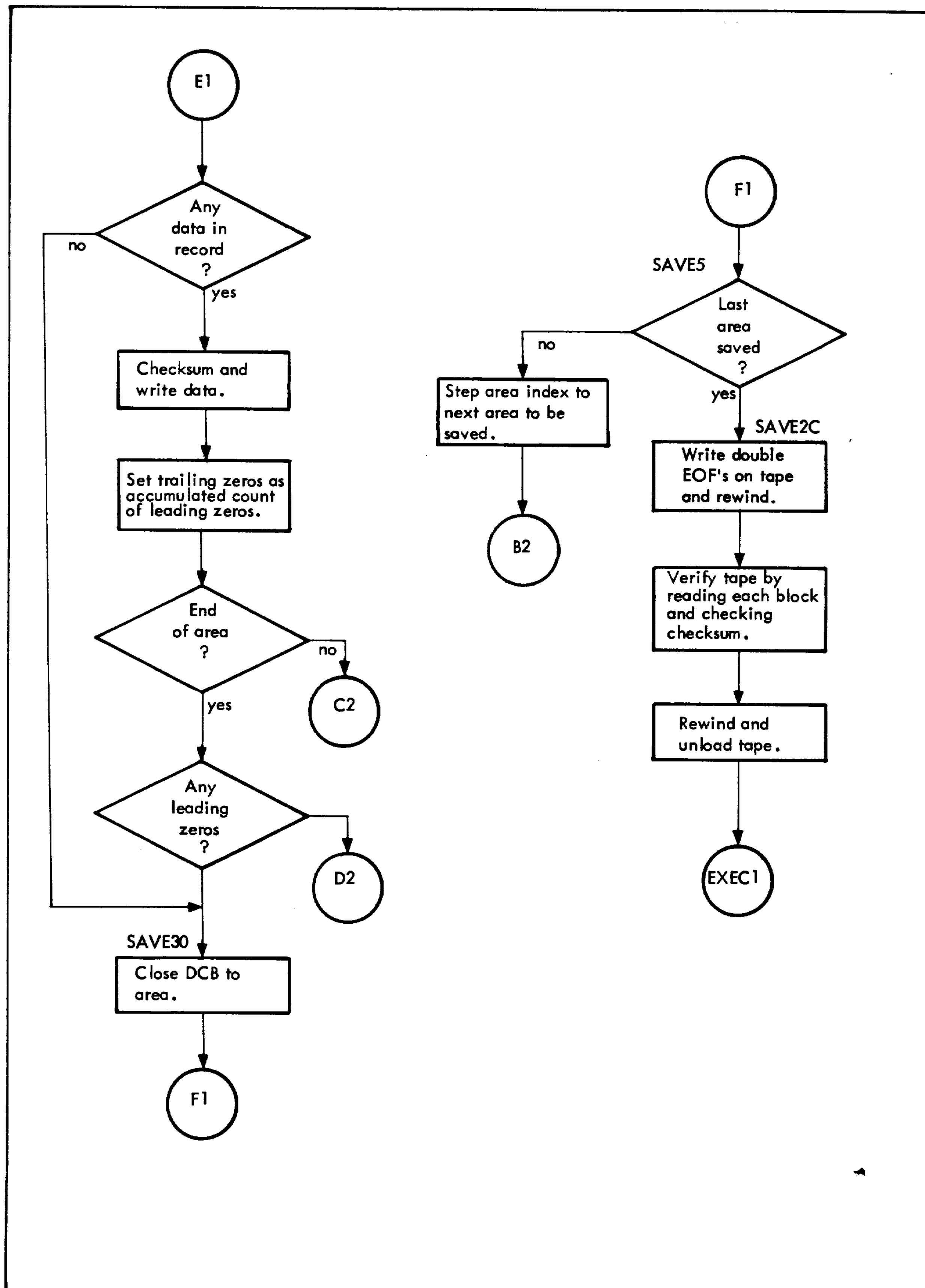



Figure 69. RADEDIT Flow, SAVE (cont.)

# 12. TERMINAL JOB ENTRY

## TJE COC Tables

In order to treat a communication line as a device, the extensions given below exist for the DCT tables. These DCT entries are extended according to the number of communication lines declared on the :COC command.

| Label | Size (Words) | Length | Contents |
|---|---|---|---|
| DCT2 | 1/4 | No. lines | COC index (begins at zero) |
| DCT3 | 1/4 | No. lines | X'C0' for all entries |
| DCT4 | 1/4 | No. lines | X'01' for all entries |
| DCT5 | 1/4 | No. lines | Zero |
| DCT6 | 1/4 | No. lines | Input queues |
| DCT14 | 1/4 | No. lines | Line index (begins at zero) |
| DCT16 | 2 | No. lines | 'nl !!LNxxx', where 'xxx' is the EBCDIC representation of the decimal line number beginning at '000'. |
| DCT18 | 1/4 | No. lines | Output queues |
| DCTMOD | 1 | No. lines | '7611' for all entries |
| DCTJID | 1/4 | No. lines | Zero |
| DCTTJE† | 1/4 | No. lines | X'04' = INIT to be performed<br>X'08' = active line<br>X'01' = logon to be performed<br>X'10' = logoff to be performed<br>X'20' = INIT with debug<br>X'02' = logon being performed |

In addition, a new index is kept in the 0th entry of DCT7. This index represents the total number of DCT entries including communication lines. DCT1 entry zero will continue to represent the total number of noncommunication type equipment.

Upon completion of SYSGEN, K:DCT1 points to DCT7 instead of DCT1.

The following tables and values are generated at SYSGEN from the parameters on the :COC control commands.

| Label | Size (Words) | Number | Contents |
|---|---|---|---|
| LCOC | value | -- | Number of COCs-1 |
| COD:LPC | 2* | COC | Each double word represents the range of logical line numbers for the COC (e.g., COC0 has 7 lines; COC1 has 8 lines; DBLWRD1 0, 6; DBLWRD2 7, 14). |
| COD:HWL | 2* | COC | Each double word is a bit mask representing HARDWIRED lines with a bit set. |

---

†New field.

| Label | Size (Words) | Number | Contents |
|---|---|---|---|
| COH:DN | 1/2 | COC | Device address |
| CO:AIIL | 1 | 1 | All input interrupt levels |
| CO:AOIL | 1 | 1 | All output interrupt levels |
| CO:IIL | 1 | 1 | Input interrupt level-COC0 |
| CO:OIL | 1 | 1 | Output interrupt level-COC0 |
| COA:IIG | value | -- | Input interrupt group number |
| COA:OIG | value | -- | Output interrupt group number |
| CO:STAT | 1 | COC | WD, 10 X'30n0' when n begins at 0 and is incremented by 1 for each successive entry |
| CO:OUTRS | 1 | COC | RD, 7 X'30n0' |
| CO:RCVON | 1 | COC | WD, 7 X'30n1' |
| CO:XDATA | 1 | COC | WD, 6 X'30n5' |
| CO:RCVDO | 1 | COC | WD, 7 X'30n3' |
| CO:TRNDO | 1 | COC | WD, 7 X'30n7' |
| CO:XSTOP | 1 | COC | WD, 7 X'30nE' |
| CO:LST | 1 | COC | Offset to next RING buffer character |
| CO:RINGE | 1 | COC | Pointer to last word of RING buffer + 1 |
| COH:RBS | 1/2 | COC | Size of RING buffer |
| CO:IN0 | 4 | 1 | COC input PSD |
| CO:INN | 7 | (LCOC>0) | COC n input PSD |
| CO:OUT0 | 6 | COC | COC output PSD |
| CO:OUT | 1 | COC | Address of output PSD |
| CO:CMND | 4 | COC | COC command list |
| COH:II | 1/2 | COC | Address of input interrupt |
| COH:IO | 1/2 | COC | Address of output interrupt |
| COCBUF | 4 | total BUFFERS | Used for input and output buffers |
| COCHPB | 1 | 1 | Head pointer for COC buffers |
| HRBA | value | -- | 4x(total BUFFERS-1) |
| LNOL | value | -- | Total lines for all COCs |
| COCOC | 1/4 | LNOL | Output character count |

| Label | Size (Words) | Number | Contents |
|---|---|---|---|
| LB:UN | 1/4 | LNOL | DCT index if in use |
| ARSZ | 1/4 | LNOL | Actual record size |
| BUFCNT | 1/4 | LNOL | Number of buffers in use |
| MODE | 1/4 | LNOL | X'80' = echoplex<br>X'40' = escape sequence<br>X'10' = read pending<br>X'08' = tab simulation |
| MODE2 | 1/4 | LNOL | X'80' = turnoff signal<br>X'40' = paper tape mode (XON = 1, XOFF = 0)<br>X'20' = space insertion (esc 5)<br>X'08' = shift to lower case (esc (, esc))<br>X'04' = check parity |
| MODE3 | 1/4 | LNOL | X'80' = tab relative<br>X'40' = paper tape mode (escP)<br>X'08' = input lost (insufficient buffers) |
| MODE4 | 1/4 | LNOL | if RATE<br>0 - 10 = 18<br>11 - 15 = 19<br>16 - 30 = 1A<br>31 - 60 = 1B<br>60 - = 1C |
| MODECPR | 1/4 | LNOL | X'80' = non-TJE line |
| COCTERM | 1/4 | LNOL | 0 = M33 Teletype<br>1 = M35 Teletype<br>2 = M37 Teletype<br>3 = Xerox Model 7015 |
| RSZ | 1/4 | LNOL | Record size |
| CPI | 1/4 | LNOL | Input carriage position |
| CPOS | 1/4 | LNOL | Present carriage position |
| COCII | 1/2 | LNOL | Input insertion point |
| COCIR | 1/2 | LNOL | Input removal point |
| COCOI | 1/2 | LNOL | Output insertion point |
| COCOR | 1/2 | LNOL | Output removal point |
| TL | 1/2 | LNOL | Tab link |
| EOMTIME | 1/2 | LNOL | Time out value |

## TJE Commands

The following TEL commands translate to service calls:

| Command | Service Call |
|---|---|
| MESSAGE | TYPE with id |
| STDLB | STDLB |

| Command | Service Call |
|---|---|
| MEDIA | MEDIA |
| BATCH | JOB |
| JOB | JOB |
| CANCEL | JOB |
| SETNAME | SETNAME |
| INIT | INIT |
| DEBUG | DEBUG, WAIT |
| EXIT | WAIT |
| EXTM | EXTM |
| STOP | STOP |
| START | START |

Each of the other TJE commands is treated as follows:

TABS obtains a four-word piece of temp space, fills it with the indicated tabs, and attaches it to the JCB through JCBTABS.

OFF sets bit TJEOFF of DCTTJE and executes a TERM service call.

RUN does a SETNAME of TEL to the taskname, sets bit TJEDBG if DEBUG is specified and executes a TERM service call.

QUIT must execute a TERM service call.

CONTINUE or GO must execute a TRTN service call.

## TJE Structure

### Account Maintenance

The structure of the AI file in the SP area is as follows:

- The file is created with EDIT; thus, columns 73-80 will contain the line numbers.
- The file contains one account record per account and multiple subaccount records per account record.
- An account record contains the account number (e.g., K1514201).
- A subaccount record which must immediately follow the account record contains the account and subaccount (e.g., K1514201, CRO107143151).

More parameters will be added to each type of record later.

Since the file is a fixed length record, blocked file, a binary search may be used to locate and verify logon data.

## TEX Operation

When the COC handler recognizes a new line to be logged on, bit TJEON of DCTTJE is set and the terminal executive (TEX) will be started. Upon activation, TEX scans DCTTJE using DCT7 entry zero as the index to determine what is to be done. The following operations are performed if the indicated bit is set:

TJEON causes TEX to output the logon message. If time-out occurs, COC sets TJEOFF and starts TEX. If input is successful within five tries, and the AI file contains the matching account and subaccount, TEX creates a job with jobname equivalent to the controlling device (DCT16, bytes 3-7). TEX executes a SETNAME TEL=TEL, outputs the logon message to the user and the operator, resets TJEON, sets TJEACT and TJEINIT and continues.

TJEOFF causes TEX to KJOB the job, clears DCTTJE, and writes a logoff message to the terminal and the operator and continues.

TJEINIT causes TEX to reset TJEINIT and TJEDBG, and to INIT TEL within the indicated job with DEBUG if TJEDBG is set.

All ECB operations execute with no wait.

TEX uses WAIT whenever it is at an idle state. WAIT will return when an ECB is posted or a START is received.

In all error cases, appropriate messages are produced and if fatal, TEX sets TJEOFF. Therefore, the order of bit checking by TEX is important and is as follows:

TJEOFF, TJEON, TJEINIT

TEX is a mapped secondary task that runs in the CP-R job. It exists as monitor overlays, and calls SCAN to break apart the logon accounts. It is in part a PROLAY that processes an RTS stack and other needed data.

## TEL Operation

TEL exists as both NROLAY(s) and an SSOLAY. The SSOLAY is used to provide context (stacks) in which to run TEL when no user's load module is being run. The actual executable code of TEL is available as NROLAY(s) which run in either the user's or TEL's load module. The toggling between TEL and the user in a synchronous environment (one task) is actually accomplished by TEX toggling between TEL and the user. TEL may therefore be entered in two ways:

1. By a direct branch to TEL from the TEL load module.
2. By an entry to TELCNTL when a CONTROL sequence occurs (identical to BREAK) from a user load module. In this case, TEL will eventually do a TRTN if GO or CONTINUE is input.

Upon initially (job creation) gaining control, TEL obtains a blocking buffer, places the pointer in JCB word JCBTJEBB, assign default operational labels, assign default tabs, and begins operation. At any point in its operation, TEL is prepared to receive a CONTROL sequence. Its action at any time other than that initial CONTROL from a user load module is to execute a TRTN, since this is simply used to activate TEL after a WAIT.

TEL takes its input through the C operational label and directs its output to the LL oplabel. TEL uses the blocking buffer to construct its FPTs, DCBs, and buffers. This information is constructed for each TEL command. All asynchronous operations are executed with wait.

TEL uses read with prompt (!) for command input. When input is complete, it uses SCAN to parse the input. If an error is discovered during parsing a (?) is output beneath the offending field. If no error occurs during parsing, but instead on the service call generated as a result of the input, TEL generates a question mark in column 0 and then prompts with a new cycle.

Job management and task management are altered as follows to accommodate TJE:

- Task termination recognizes the TEL task, upon termination sets TJEINIT in DCTTJE, and starts TEX.
- Job termination recognizes terminal jobs and does not allow job termination until TJEACT is reset.

## Time Slicing

Time slicing in CP-R is available for non-TJE systems since the implementation is not terminal dependent.

The algorithm used for time-slicing in CP-R must be predicated upon the following guidelines:

1. Scheduler thrashing (inefficient context changing) must be avoided.

2. Swapper thrashing (inefficient rolling in and out) must be avoided.

3. Background must run at nearly full speed.

4. Symbionts and media must run at full speed.

5. The algorithm should fit nicely into CP-R's present structure and be easily expandable if the need arises. The algorithm and its implementation follow:

   a. Three variables exist, all of which will be fixed at SYSGEN, depending on the swapping device.

      10 ms ≤ QMIN ≤ 140 ms is the minimum time a time-sliced task will be allowed to run when scheduled before being interrupted to service other time-sliced tasks.

      150 ms ≤ QSWAP ≤ 400 ms is the minimum time an unblocked time-sliced task must remain in core before being considered a candidate for swap out. Unblocked means either compute or I/O bound. (Each I/O operation is equivalent to 10 ms compute time. Terminal input is not considered as I/O bound.)

      500 ms ≤ QMAX is the amount of time an unblocked time-sliced task will run before being considered for exchange with another user who was rolled out in an unblocked state.

   b. Time slicing is specified by bit $F_5$ in the INIT service call and indicated by bit TSLICE of LMISTAT. Default priority is X'FFFF', which is equivalent to the priority of background; thus all TEL and TEL INITed tasks will run at a priority equivalent to background.

   c. The dispatcher searches its queue to find the highest priority candidate to run. If the task is time-sliced, CLOCK 4 will be set up to interrupt after QMIN. The task is then given control and may only be interrupted at its dispatcher level for rescheduling by the occurrence of QMIN or a higher priority nontime-sliced task.

   d. When the dispatcher again gains control, it examines the last dispatched task. If the task is time-sliced, it is requeued to the bottom of its priority.

   e. In order to give background improved response, the posting logic moves background to the top of its priority queue when it comes off a wait condition.

   f. MMEXEC's roll-out search is as follows:

| W/O TJE | With TJE |
|---|---|
| BKGRD INACTIVE | TIME SLICED TERMINAL INPUT |
| BKGRD ACTIVE | TIME SLICED BLOCKED |
| LONG WAIT INACTIVE | |
| LONG WAIT ACTIVE | LONG WAIT INACTIVE |
| : | LONG WAIT ACTIVE |
| | : |
| | TIME SLICED EXCEED QSWAP |

BLOCKED is defined as anything other than UNBLOCKED (I/O or compute bound) or terminal input.

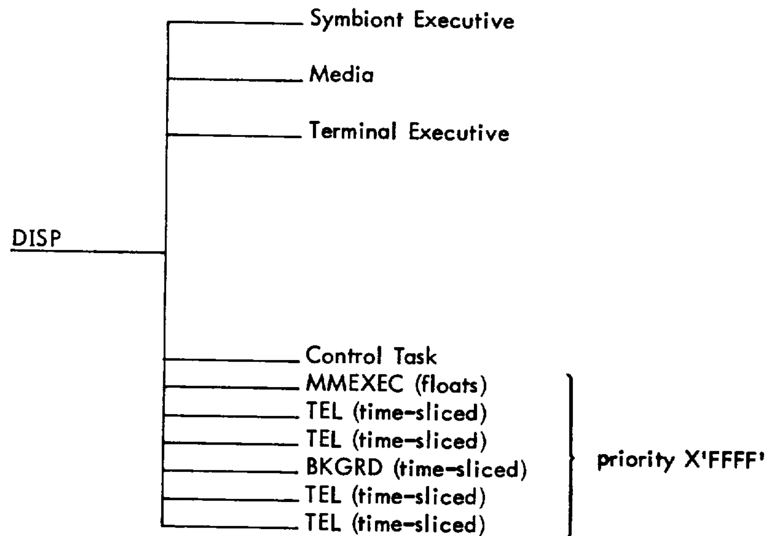
Terminal input and BLOCKED tasks are treated similarly to long wait tasks. Higher priority tasks in these states may be rolled out. The roll ECB will be added to the R-chain when it becomes executable.

Since QSWAP tasks are placed in MMEXEC's R-chain upon roll-out (thus being immediate candidates for roll-in), the search for QSWAP tasks applies only to equal or lower priority.)

g. MMEXEC continues to run until no candidates are found to satisfy R-chain requests or until the R-chain is exhausted and then executes a WAIT. Whenever a time-sliced task is awaiting terminal input, becomes blocked, or exceeds QMAX, the MMEXEC is started.

h. In order to keep MMEXEC above the time-sliced tasks, the following rule applies:

Whenever time-sliced and nontime-sliced tasks are queued at the same priority, the nontime-sliced tasks are queued above the time-sliced tasks. By way of example, a typical CP-R task structure may appear at a given moment as follows:



# 13. MEDIA

The MEDIA processor consists of three parts:

1. Key-in processor
2. CAL service call processor
3. The MEDIA task. This task is subdivided into two parts: an overlay section and a resident section.

The MEDIA key-in processor forms a 13-word table containing the information from the parameters specified. This table is then sent to the MEDIA task as a signal data packet to be processed. The signal to the MEDIA task is sent with a class flag of

X'8000'

to indicate a key-in signal. The format and contents of a key-in packet are as follows:

| 0–7 | 8–15 | 16–23 | 24–31 | |
|---|---|---|---|---|
| IFILE, 0, ALL, DEL, 0, IREW, IUNLOAD | MASN | Area Name Input | | Input Specification |
| | | Sfile Count | 0——0 | |
| Input Name | | | | |
| Input Name | | | | |
| Input Account Name | | | | |
| Input Account Name | | | | |
| Spare | | | | |
| Spare | | | | |
| Spare | | | | |
| OFILE, SPACE, NVFC, ADD, WEOF, OREW, OUNLOAD | Space Count | WEOF Count | Area Name Output | Output Specification |
| | | Sfile Count | 0——0 | |
| Output Name | | | | |
| Output Name | | | | |
| Output Account Name | | | | |
| Output Account Name | | | | |

where

IFILE / OFILE = Ø device name specified.

= 2 area and file name specified.

| | | |
|---|---|---|
| ALL | = Ø | copy only the specified file. |
| | = 1 | on magnetic tape, copy all files up to an end-of-file double tapemark. |
| DEL | = Ø | retain the file after the copy. |
| | = 1 | delete the file after the copy. |
| IREW | = Ø | leave magnetic tape positioned after file copied. |
| | = 1 | rewind the tape to BOT after the copy. |
| IUNLOAD | = Ø | same as IREW = 0. |
| | = 1 | tape is rewound "off-line" after the copy. |
| MASN | | the MEDIA Action Sequence Number to identify the action request. The next number in sequence is assigned to each MEDIA request. |
| Area Name, Input (Output) Name, Account Name | | when IFILE (OFILE) = 2, the corresponding Area Name contains the two-letter EBCDIC name of the area, Input (Output) Name contains the filename to copy from (to), and Input (Output) Account contains the account name.<br><br>when IFILE (OFILE) = 0, the device name is left-justified and blank filled in the Input (Output) Name field. The corresponding AREANAME contains zeros or the SFILE count. |
| SPACE | = 0 | for printer destined files, the printer is spaced according to the VFC byte (NVFC = 0), or is to be single spaced (NVFC = 1). |
| | = 1 | the printer is to be spaced "Space Count" lines between each line output (NVFC = 1). This field is not used when NVFC = 0. |
| NVFC | = 0 | for printer destined files, the printer is to operate with VFC, and the first byte of every record contains the VFC information. |
| | = 1 | for printer destined files, the printer is to operate without VFC, and the first byte in every record is data. |
| ADD | = 0 | the output tape is to be positioned according to the SFILE count before the copy commences. |
| | = 1 | the output file will follow the last file on magnetic tape or be added to the end of an existing disk file. |
| WEOF | = 0 | two end-of-files are to be written to the output tape after the copy. |
| | = 1 | "WEOF Count" end-of-files are to be written to the tape after the copy. |
| OREW | = 0 | the output tape is to be left positioned after the file. |
| | = 1 | the output tape is to be rewound after the copy and end-of-files (if any) are written. |
| OUNLOAD | = 0 | same as OREW = 0. |
| | = 1 | the output tape is rewound "off-line" after the copy and end-of-files (if any) are written. |
| Space Count | | the number of lines to space the printer between each line of output when NVFC = 1 SPACE = 1. |
| WEOF Count | | the number of end-of-files to write to the output tape after the copy when WEOF = 1. |
| SFILE Count | | the number of files a magnetic tape is to be forward skipped before the copy is started. |

MEDIA key-ins specifying the control functions C, L, I, X communicate directly with the MEDIA task, setting or resetting the appropriate indicators in the resident portion of the MEDIA task.

The MEDIA service call forms a nine-word table from the call's FPT, job name and task name. This table is sent to the MEDIA task as the data packet of a Signal call. A class mask of

X'4000'

is used for the Signal to identify it as a service call packet. The contents of a service call packet are as follows:

<table>
<tr><td>FILE</td><td>DS</td><td>NVFC</td><td>DEL</td><td>0</td><td>0</td><td>0</td><td>MASN</td><td>Area Name</td></tr>
<tr><td colspan="9">File Name</td></tr>
<tr><td colspan="9">File Name</td></tr>
<tr><td colspan="9">Account Name</td></tr>
<tr><td colspan="9">Account Name</td></tr>
<tr><td colspan="9">Job Name</td></tr>
<tr><td colspan="9">Job Name</td></tr>
<tr><td colspan="9">Task Name</td></tr>
<tr><td colspan="9">Task Name</td></tr>
</table>

where

| | | |
|---|---|---|
| File | = 2 | the file is specified by the area and file name. |
| DS | = Ø | printer destined files will be printed single spaced (NVFC = 1), or according to the VFC byte in the record (NVFC = Ø). |
| | = 1 | printer destined files will be double spaced (NVFC = 1). This field is ignored when NVFC = Ø. |
| NVFC | = Ø | printer destined files will be printed with VFC, and the first byte of each record will be used as the VFC byte. |
| | = 1 | printer destined files will be printed without VFC; the first byte of each record is printed. |
| DEL | = Ø | do not delete the file after the copy. |
| | = 1 | delete the file after the copy. |
| MASN | | the MEDIA Action Sequence Number. The next number in sequence is assigned to each request as an identification number. |

Area Name, File Name, Account Name — the area, file and account name as specified in the MEDIA call (FILE = 2).

Task Name, Job Name — the task- and job-name of the task that issued the service call. These will be printed on the burst page of a printer-destined file.

The resident section of the task contains all permanent areas the task requires and a short segment of code that is the main loop of the copy. The contents and structure are as follows:

| Label | Contents | | | |
|---|---|---|---|---|
| MEDRCTRL | MASN | 0———0 | 0 \| 0 \| 0 \| 0 \| END \| STOP \| 0 \| 0 | Status \| ABORT |
| MEDRQINF | 0———0 | 0———0 | 0———0 | 0———0 |
| MEDRJOB | Job Name (8 characters) | | | |
| MEDRTASK | Task Name (8 characters) | | | |
| MEDRITMP | Input device index, or zero | 0———0 | Input Device Control Info. | |
| MEDROTMP | Output device index, or zero | 0———0 | Output Device Control Info. | |
| MEDRBB1 | First blocking buffer control word temp (zero if no blocking buffer) | | | |
| MEDRBB2 | Second blocking buffer control word temp (zero if no blocking buffer) | | | |
| MEDRERRS | Error indicator/error code (4 characters) | | | |
| | TYC code if device error; otherwise, 0 | | | |
| MEDRIDCB | Input DCB | | | |
| MEDRODCB | Output DCB | | | |
| MEDRRA<br>MEDRRB<br>MEDRWA<br>MEDRWB<br>MEDRCHKR<br>MEDRCHKW<br>MEDROC<br>MEDRFPTX | FPTs to read and write "A" and "B" buffers, check Reads and Writes, open and close DCBs; miscellaneous services | | | |
| MEDR900 | No-operation Error Processor | | | |
| MEDRLOOP | Copy code | | | |
| MEDRSTCB | STCB | | | |
| MEDRSTAK | Temp Stack | | | |

where

MASN one byte counter; next number is assigned to each new MEDIA request as an identification number.

End when END = 1, do not initiate a new copy operation. Set by MEDIA control function "L".

Stop = 1 suspend the current copy. Set by control function "S".

= 0 continue or resume the current copy.

Status set by MEDIA when operator intervention is required or when execution must be suspended to control where processing is to be resumed. Values and meanings are:

0 idle.

1 device manual during pre-copy processing.

2 in a copy, stopped by S key-in.

3 device manual during copy.

4 waiting device from Symbionts, exclusive use.

5 waiting tape mount and operator okay.

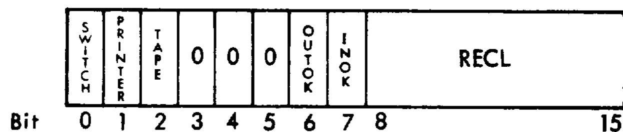
6 printing break pages in part 2.

Abort when ABORT = 1, abort the current copy operation. Set by control function X.

Job Name }
Task Name } the 8-character job and task names of the current copy.

Input/Output Index and Control Words the DCT index of the input and output devices, or zero (0) if to a disk file or null device.

the Control Information halfword has the format:




where

SWITCH switchable device; other devices of the same type may be substituted.

PRINTER line printer device — any model.

TAPE magnetic tape device — any model.

OUTOK valid output device.

INOK valid input device.

RECL maximum record length in words, minus 1.

Input, Output DCBs prototype DCBs for the input and output files.

FPTs two each Read and Write FPTs and two Check FPTs. Input and output are doubled buffered, plus an FPT to OPEN/CLOSE the DCBs and space for pre-processing FPTs.

Copy code the resident copy code.

STCB secondary Task Control Block for the MEDIA task.

Temp Stack push down stack.

The overlay section contains the functions necessary to process the MEDIA CALs and key-ins, effect the MEDIA control functions, acquire the specified input/output devices, open and close the files, and to do any pre- or post-processing.

For each copy, the next operation is selected by searching for a signal, first with a class mask of X'8000' to select the highest priority keyin request and then, if none exists, with a class mask of X'4000' to select a CAL service request.

Having selected a request, the DCBs for both files are formed. Conflict with the SYMBIONT processor is then checked and a delayed request process is initiated if it exists (see below). The input DCB is then opened. An error due to unavailability goes to the delayed request process.

Successful opening of the input allows a similar process to start in the output file. When both files are successfully opened pre-copy preparation can begin.

If it is impossible to obtain both the input and output devices, due either to symbiont conflict or device unavailability, delayed request processing is initiated. This is done by closing the input file, if open, to free the device and avoid deadlock conditions. The "Requested by MEDIA" bits (bit 1) are set in each device's DCTRBM byte. Then, if either device is in conflict with the SYMBIONT processor, the MEDIA task does a foreground WAIT to await a start from the other task when it is finished with the devices. If they are unavailable for any other reason, a five-second timer is initiated and then a foreground WAIT is done. In either case an internal status indicator is set to "acquiring devices".

When the foreground WAIT returns and the "acquiring devices" indicator is set, processing continues as above with checks for input device Symbiont conflicts. This process is repeated until both devices are successfully acquired.

After successfully opening the two files, any pre-copy positioning is done first for the input and then the output file. The resident copy loop is prepared to perform the requested NVFC or SPACE,n processing and the copy initiated.

At copy completion, postcopy processing and positioning are done. Then for both normal and abnormal or abort terminations, the DCBs are closed and the SYMBIONT task started if it has requested either device.

The MEDIA internal status is set to "idle" and a SUPERWAIT with a short timeout is done. When the wait returns, the next copy request is polled as above.

Figure 70 shows the definitions of the field and indicator symbols are defined in the CPREQU System.

```
LABEL                    INDEX
NAME           VALUE     TYPE      COMMENT

                         ECB CLASS FLAGS

MEDKEYCL EQU   X'8000'             A KEY-IN REQUEST
MEDCALCL EQU   X'4000'             A CAL    REQUEST



                       MEDIA CONTROL FLAGS

MEDRGO   EQU   X'8000'             START OR CONTINUE A COPY OPERATION
MEDREND  EQU   X'0800'             END COPY, DO NOT START A NEW COPY
MEDRSTOP EQU   X'0400'             STOP (SUSPEND) COPY IMMEDIATELY
MEDRABRT EQU   X'0001'             ABORT CURRENT COPY OPERATION
MEDRMCTL EQU   M15                 MASK FOR STORING CONTROL FLAGS



               STATUS INDICATORS FOR MEDIA AFTER STARTS

MEDSIDLE EQU   00                  IDLE
MEDSIMOP EQU   1**1                DEVICE INOP, AWAITING READY
MEDSCOPY EQU   2**1                IN A COPY; PROCESS WAS STOPPED
MEDSICPY EQU   3**1                IN A COPY; DEVICE INOP
MEDSAQIR EQU   4**1                ACQUIRING DEVICES FROM SYMBIONTS
MEDSTAPE EQU   5**1                WAITING TAPE MOUNT, 'I' KEY-IN
MEDSHEDR EQU   6**1                PRINTING BREAK PAGES IN PART 2



                  DEVICE CHARACTERISTIC INDICATORS

MEDASW   EQU   X'8000'             OK TO SWITCH TO SIMILAR DEVICE
MEDAPRNT EQU   X'4000'             DEVICE IS A PRINTER
MEDATAPE EQU   X'2000'             DEVICE IS A TAPE
MEDAOUOK EQU   X'0200'             DEVICE LEGAL FOR OUTPUT DEVICE
MEDAINOK EQU   X'0100'             DEVICE LEGAL FOR INPUT DEVICE
MEDA#WRD EQU   X'00FF'             NUMBER OF WORDS IN MAX.LEN.REC.
MEDRSSZ  EQU   145                 SIZE OF STACK AREA IN ROOT



                 MAP FOR COPY REQUEST SIGNAL PACKETS

MEDPLEN  EQU   16                  LENGTH OF A SIGNAL PACKET

MEDPBASE EQU   0         FW        HEADER WORD / BASE OF PACKET
MEDPICTL EQU   1         FW        INPUT  CONTROL WORD / AREA NAME
MEDPIFIL EQU   2         FW        INPUT  FILE-, DEVICE- NAME
MEDPOCTL EQU   9         FW        OUTPUT CONTROL WORD / AREA NAME
MEDPJOBN EQU   6         FW        JOB NAME OF REQUESTING TASK
MEDPOFIL EQU   10        FW        OUTPUT FILE-, DEVICE- NAME
MEDPTSKN EQU   8         FW        TASK NAME OF REQUESTING TASK
MEDPIACN EQU   4         FW        INPUT  FILE ACCOUNT NAME
MEDPOACN EQU   12        FW        OUTPUT FILE ACCOUNT NAME
MEDPMASN EQU   5         BYTE      SEQUENCE / ID NUMBER
MEDPISFL EQU   6         BYTE      SFILE COUNT, INPUT
MEDPOSFL EQU   18        BYTE      SFILE COUNT, OUTPUT
MEDPIOPT EQU   2         HW        INPUT  OPTIONS HW
MEDPIAR  EQU   3         HW        INPUT  AREA NAME
MEDPOOPT EQU   18        HW        OUTPUT OPTIONS HW
MEDPOAR  EQU   19        HW        OUTPUT AREA NAME
```

Figure 70. Field and Indicator Definitions

| LABEL NAME | | VALUE | INDEX TYPE | COMMENT |
|---|---|---|---|---|
| | | **OPTION INDICATORS AS BYTE VALUES** | | |
| MEDOFILE | EQU | X'80' | | FILE IS SPECIFIED |
| MEDOALL | EQU | X'10' | | ALL FILES ON INPUT TO BE COPIED |
| MEDODEL | EQU | X'08' | | DELETE INPUT FILE AFTER COPY |
| MEDOSPAC | EQU | X'20' | | SPACE COUNT SPECIFIED FOR PRINTER |
| MEDONVFC | EQU | X'10' | | DO NO VFC; 1ST DATA BYTE IS DATA<br>IF VFC, 1ST DATA BYTE IS VFC BYTE |
| MEDOADD | EQU | X'08' | | ADD FILES TO EXISTING FILE OR TAPE |
| MEDOWEOF | EQU | X'04' | | WRITE EOFS AFTER COPY |
| MEDOREW | EQU | X'02' | | REWIND INPUT/OUTPUT AFTER COPY |
| MEDOUNLD | EQU | X'01' | | UNLOAD INPUT/OUTPUT AFTER COPY |
| | | **OPTIONS AS HALF-WORDS** | | |
| MEDHFILE | EQU | MEDOFILE**8 | | |
| MEDHALL | EQU | MEDOALL**8 | | |
| MEDHDEL | EQU | MEDODEL**8 | | |
| MEDHREW | EQU | MEDOREW**8 | | |
| MEDHUNLD | EQU | MEDOUNLD**8 | | |
| MEDHSPAC | EQU | MEDOSPAC**8 | | |
| MEDHNVFC | EQU | MEDONVFC**8 | | |
| MEDHADD | EQU | MEDOADD**8 | | |
| MEDHWEOF | EQU | MEDOWEOF**8 | | |
| | | **OFFSETS TO INFORMATION IN RESIDENT TABLES** | | |
| MEDRSN | EQU | 0 | BYTE | NEXT ID NUMBER TO BE ASSIGNED |
| MEDRSTAT | EQU | 3 | BYTE | STATUS INDICATORS |
| MEDR#CAL | EQU | 4 | BYTE | NUMBER OF CAL REQUESTS QUEUED |
| MEDRCFIN | EQU | 5 | BYTE | LAST CAL ID PROCESSED |
| MEDR#KEY | EQU | 6 | BYTE | NUMBER OF KEY-IN REQUESTS QUEUED |
| MEDRKFIN | EQU | 7 | BYTE | LAST KEY-IN ID PROCESSED |
| MEDRFLAG | EQU | 1 | HW | CONTROL FLAGS / STATUS INDICATORS |
| | | **OFFSETS INTO READ/WRITE FPTS** | | |
| MEDRRBUF | EQU | 4 | FW | BUFFER ADDRESS, READ FPT |
| MEDRWBUF | EQU | 4 | FW | BUFFER ADDRESS, WRITE FPT |
| MEDRRBYT | EQU | 5 | FW | NUMBER OF BYTES TO READ |
| MEDRWBYT | EQU | 5 | FW | NUMBER OF BYTES TO WRITE |
| MEDRRBTD | EQU | 6 | FW | BYTE DISPLACEMENT IN 1ST WORD |
| MEDR#BYT | EQU | 7 | FW | ACTUAL NUMBER OF BYTES READ |

Figure 70. Field and Indicator Definitions (cont.)

# 14. EDIT SUBSYSTEM

## Functional Overview

The EDIT processor is a line-at-a-time text file manipulation utility available to the CP-R user either in the background stream or under control of the Terminal Job Entry system executive.

EDIT operates in one of two states: the command state or the active state. The command state is defined as the time in which EDIT is accepting or processing a command. This state is entered when EDIT types its identifying prompt character(s), * or **, requests input, and awaits the next command. On the other hand, the active state is defined as that time in which EDIT is executing commands, processing text, or accepting text. This state is entered when a command starts execution and terminates at the completion of the command.

EDIT is assembled as two modules, one of which is considered as context (writable) and the other as procedure (read and execute). The context module is linked into the processor root. The procedure module is linked as a segment with the characteristics ILOAD, (SHARE, SYSTEM), and (ACCESS, RX). The user may chose to omit the sharability option, but if he includes it (recommended in a multi-usage environment), he must assign a segment number which does not conflict with other system-sharable segments defined at his installation. EDIT should be linked as a secondary foreground task since this permits both foreground and background execution.

The memory allocation for EDIT is shown in Figure 71.

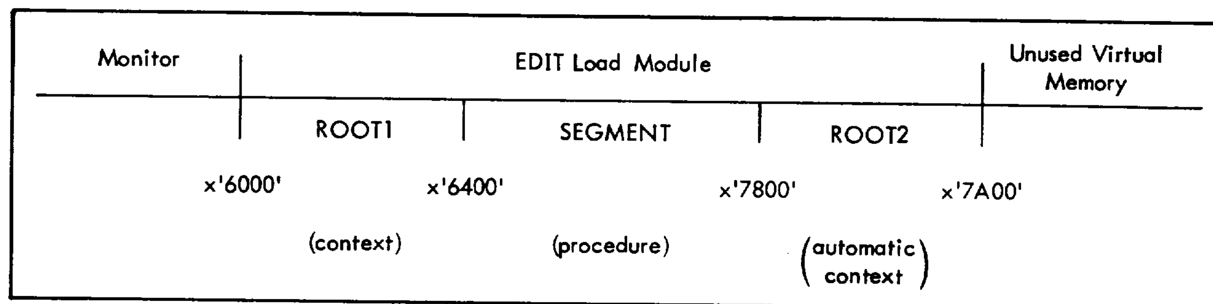


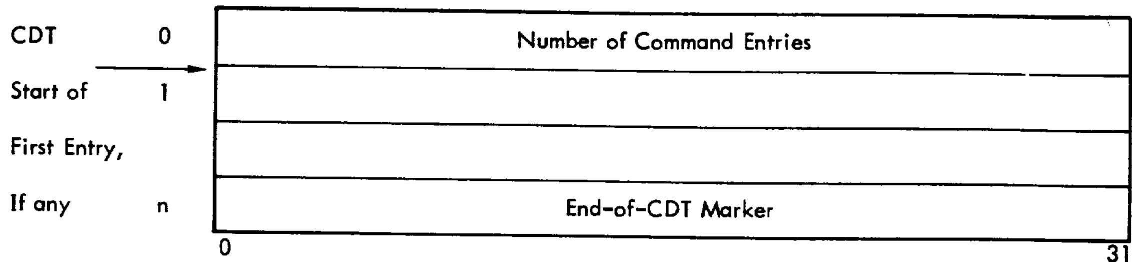

Figure 71. Memory Allocation EDIT

EDIT files are stored on disk as keyed records with the sequence number used as a key. It is assumed to have an implied decimal point such that when the key is converted to EBCDIC for printout purposes the sequence number 1 2 3 4 5 6 7 appears as 1 2 3 4 . 5 6 7 . The largest sequence number allowed is 9999.999.

## Operational Overview

EDIT is organized in a highly modular fashion. Upon entry, BEGINEDITOR performs subsystem initialization after which MASTERPARSER controls input command scan of a line of user commands. From a line of input command(s) the Command Description Table (CDT) is built. Error checks are made and warnings given to the user if necessary. MASTERPARSER uses a number of subroutines to build the Command Description Table: GETNEXTNAME and GETNEXTPARAM to break down text strings; PARSE:I:CMND$INTG to process integer strings; PARSE:I:CMND$STRG to process alphabetic strings in slashes; and routines of the form PARSE: cmnd for command processing. The format of the Command Description Table is given in Figure 72.

On completion of a command line (with possible extension), control is passed to the MASTEREXECUTIVE routine to perform the commands which then reside in the CDT. Figure 73 shows the general processing flow of EDIT. MASTEREXECUTIVE serves as a driver for command processing using F: routines for file commands, R: routines for record commands and I: routines for intra-record command processing.




Notes:

1. Entry Format

| Word | Byte 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | Number of words in entry | Command number (see Table 8.) | Order of occurrence in command line | Number of items |
| 1 | Item 1 type | Item 1 Text Pointer | Item 2 type | Item 2 Text Pointer |
| ⋮ | | | | |
| m | Item 1 (TEXTC form) | | | |
| ⋮ | | | | |
| n | Item 2 (TEXTC form) | | | |
| ⋮ | | | | |

where

Item type =
- 0 for END, carriage return.
- 1 for NAME, a file name, e.g., (EXAMPLE.D3).
- 2 for SEQ, a sequence number, e.g., 1.23.
- 3 for SEQ2, two sequence numbers or two sequence numbers separated by a dash, e.g., 1.2-2.
- 4 for INTG, a numeric string whose value is less than 1000; e.g., 123.
- 5 for STRG, a character string enclosed in slashes; e.g., /BUILD/.
- 6 for ALPH, a character string not enclosed in slashes; e.g., BUILD.
- 7 for COM, comma.
- 8 for SCOL, semicolon.
- 9 for LPAR, left parenthesis.
- 10 for RPAR, right parenthesis.
- 11 for PERIOD.
- 12 for BLANK.

Text pointer is word address of TEXTC form in entry; e.g., for item 1, Text pointer is n.

2. CDT+100=CDTADR, address of current command in CDT.

3. PARAMPSN = next available slot in CDT.

4. PRMBUFSZ = number of words in PARAMBUF to be added.

5. CHARPSN = number of next character to scan.

6. End-of-CDT Marker = X'00000100'.

Figure 72. Command Description Table (CDT)

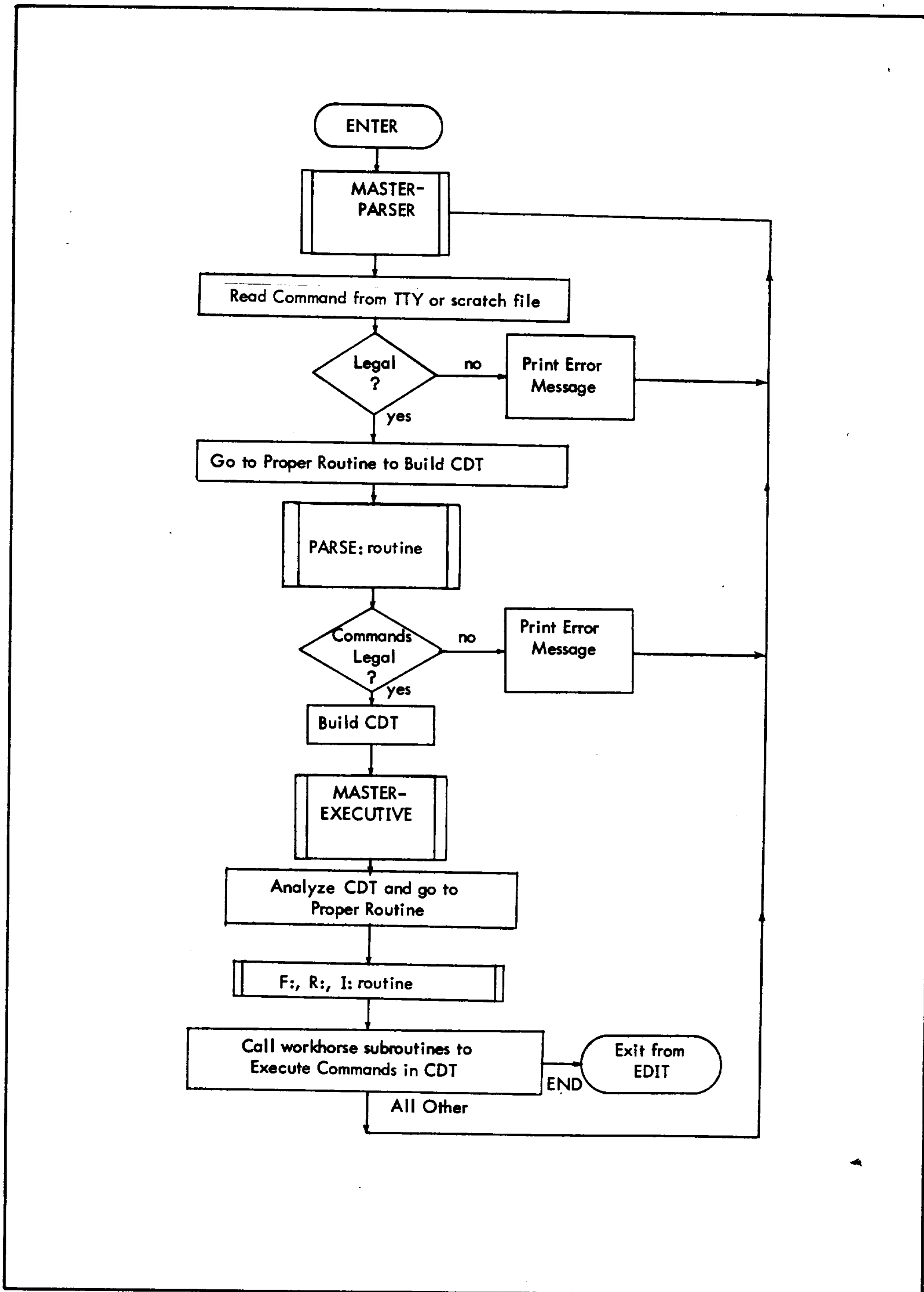



Figure 73. Overall Flow Diagram of EDIT

Table 9. Command Number Table

| Command Number | Command | CDT Builder | CDT Executor | |
|---|---|---|---|---|
| 1 | BP | PARSE:BP | F:BLANK$PRESERV | FILE COMMANDS |
| 2 | BUILD or | PARSE:BUILD | F:BUILD | |
| | SAVE | PARSE:SAVE | F:SAVE | |
| 3 | COPY | PARSE:COPY | F:COPY | |
| 4 | DELETE | PARSE:DELETE | F:DELETE | |
| 5 | EDIT | PARSE:EDIT | F:EDIT | |
| 6 | END | PARSE:END | F:END | |
| 8 | CR or | PARSE:CR | F:CR | |
| | SEQ | PARSE:SEQ | F:SEQ | |
| 9 | MERGE | PARSE:MERGE | F:MERGE | |
| 10 | CM | PARSE:CM | R:COMMENTARY | RECORD COMMANDS |
| 11 | DE | PARSE:DE | R:DELETE | |
| 12 | FD | PARSE:FD | R:FIND$DELETE | |
| 13 | FT | PARSE:FT | R:FIND$TYPE | |
| 14 | IN | PARSE:IN | R:INSERT | |
| 15 | IS | PARSE:IS | R:INSERT$SUP$SEG | |
| 16 | MD | PARSE:MD | R:MOVE$DELETE | |
| 17 | MK | PARSE:MK | R:MOVE$KEEP | |
| 18 | RN | PARSE:RN | R:RENUMBER | |
| 19 | SS | PARSE:SS | R:SET$STEP | |
| 20 | ST | PARSE:ST | R:SET$STEP$TYPE | |
| 21 | TS | PARSE:TS | R:TYPE$SUP$SEQ | |
| 22 | TY | PARSE:TY | R:TYPE | |
| 23 | TC | PARSE:TC | R:TYPE$COMPRESSED | |
| 24 | FS | PARSE:FS | R:FIND$SEQUENCE | |
| 25 | GO | PARSE:GO | R:GO | |
| 26 | RET | PARSE:RET | R:RET | |
| 30 | SE | PARSE:SE | I:SET | INTRALINE COMMANDS |
| 31 | D | PARSE:I:CMND$STRG or PARSE:I:CMND$INTG | I:DELETE | |
| 32 | E | | I:OVERWR$EXTEND | |
| 33 | F | | I:FOLLOW$BY | |
| 34 | L | | I:SHIFT$LEFT | |
| 35 | O | | I:OVERWRITE | |
| 36 | P | | I:PRECEDE$BY | |
| 37 | R | | I:SHIFT$RIGHT | |
| 38 | S | | I:SUBSTITUTE | |
| 39 | JU | PARSE:JU | I:JUMP | |

Table 9. Command Number Table (cont.)

| Command Number | Command | CDT Builder | CDT Executor |
|---|---|---|---|
| 40 | NO | PARSE:NO | I:NO$CHANGE |
| 41 | RF | PARSE:RF | I:REVERSE$BPFLAG |
| 42 | TS | PARSE:TS | I:TYPE$SUP$SEQ |
| 43 | TY | PARSE:TY | I:TYPE |
| 44 | A | PARSE:I:CMND$STRG or PARSE:I:CMND$INTG | I:ALIGN |
| 45 | Y | | I:YES$CONTINUE |
| 46 | N | | I:NO$CONTINUE |
| 47 | DE | | I:DEL$REC |
| 48 | C | PARSE:C | I:COPY$REC |

## Module Analysis

The routine and subroutine descriptions which follow are organized such that the initialization routine BEGINEDITOR appears first followed by MASTERPARSER and its associated PARSE: subroutines in alphabetical order, MASTER-EXECUTIVE and its associated F:, R:, and I: command executor routines in alphabetical order, and lastly the general subroutines in alphabetical order.

In the descriptions which follow register notations have the following meaning:

| Symbol | is | Register | Symbol | is | Register |
|---|---|---|---|---|---|
| X3 | | 1 | T2 | | 9 |
| X4 | | 2 | P3 | | 10 |
| X1 | | 3 | R1 | | 11 |
| X2 | | 4 | R2 | | 12 |
| P1 | | 5 | F:LNK | | 13 |
| P2 | | 6 | R:LNK | | 13 |
| LNK | | 7 | I:LNK | | 13 |
| T1 | | 8 | | | |

## BEGINEDITOR

1. Purpose

   Performs initialization of EDIT.

2. Entry:

   This is the routine whose address is entered in the TCB for EDIT causing it to be the beginning point of execution when control is passed to the EDIT subsystem by Task Initiation.

3. Exit:

   There is no formal exit; it merely branches to next routine, MASTERPARSER1.

**MASTERPARSER1**

1. Purpose:

   Serves as the driver for the command text scanning. It performs initialization of flags, the CDT, and TSTACK. (Alternate entry MASTERPARSER is used to restart after certain types of error.)

2. Entry:

   Initially, execution branches to MASTERPARSER1 from BEGINEDITOR; thereafter it is entered via B MASTERPARSER1 or B MASTERPARSER.

3. Exit:

   If item type (Figure 71) of the first string found is one of the four shown below, the branch is to the indicated CDT builder routine.

| Item Type | CDT Builder Routine |
|---|---|
| INTG | PARSE:I:CMND$INTG |
| STRG | PARSE:I:CMND$STRG |
| CR | CMND$CONT |
| ALPH | PARSE: (routine) |

4. Operation:

   After initialization, one or two asterisks are typed for prompt characters depending on whether EDIT is in step mode. READTELETYPE2 is used to read one line of commands. MASTERPARSER increments CDTADR and the count of CDT entries and resets PARAMPSN. GETNEXTPARAM is used to test command type for one of the following: INTG, STRG, CR and ALPH. If it is none of them, the error message "C1:ILGL SYNTAX" is typed and branch is made to MASTERPARSER. If the command type is ALPH but the command is not one of those shown in Table 9, the message 'C1:UNKN CMND' is typed and branch is made to MASTERPARSER.

The flow of MASTERPARSER is given in Figure 74.

PARSE:CM

1. Purpose:

   Adds an entry to the CDT for CM n, c.

2. Entry:

   B PARSE:CM

   This subroutine is invoked via the CBRCHTBL of MASTERPARSER after a command has been identified.

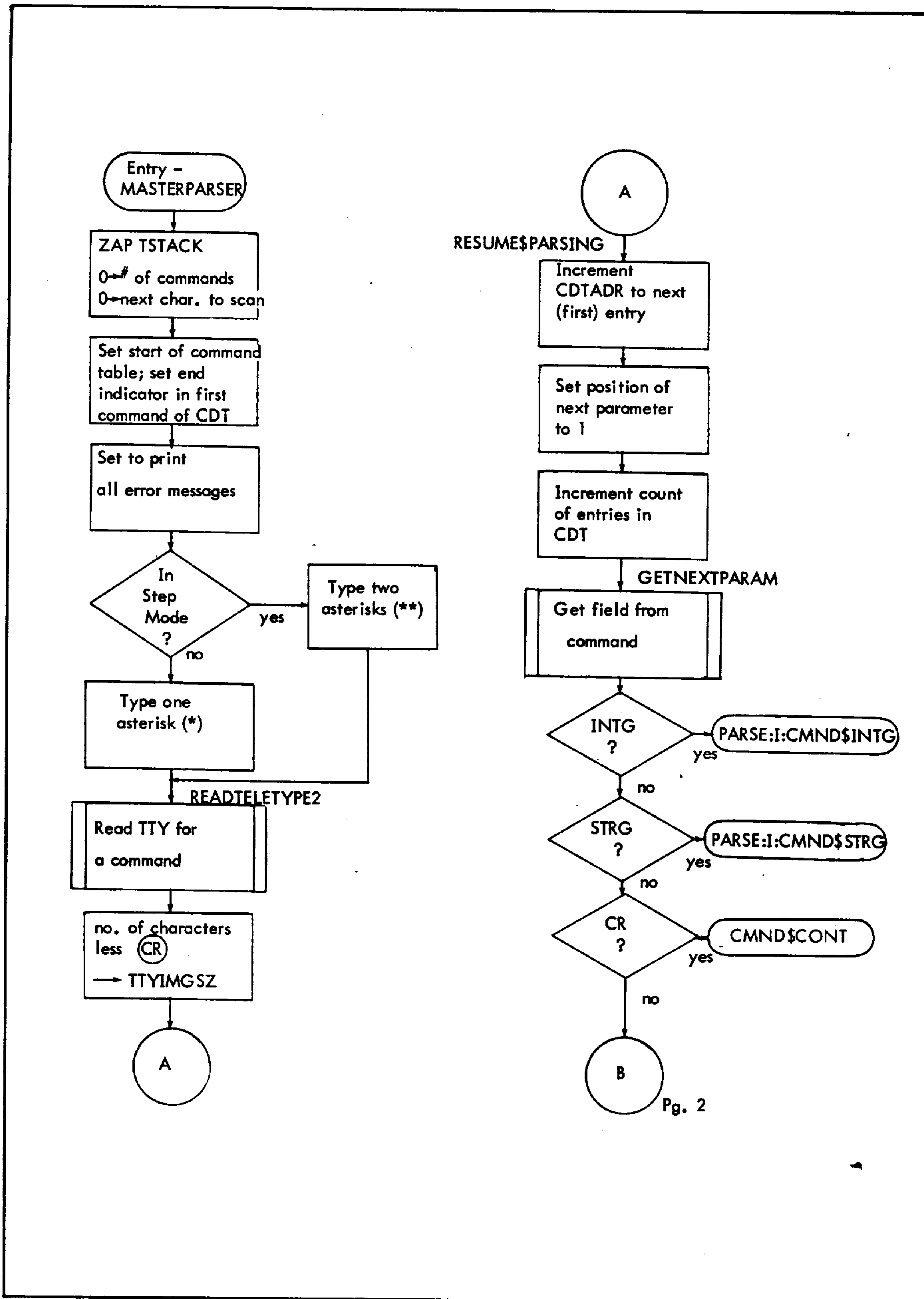



Figure 74. Flow Diagram of MASTERPARSER

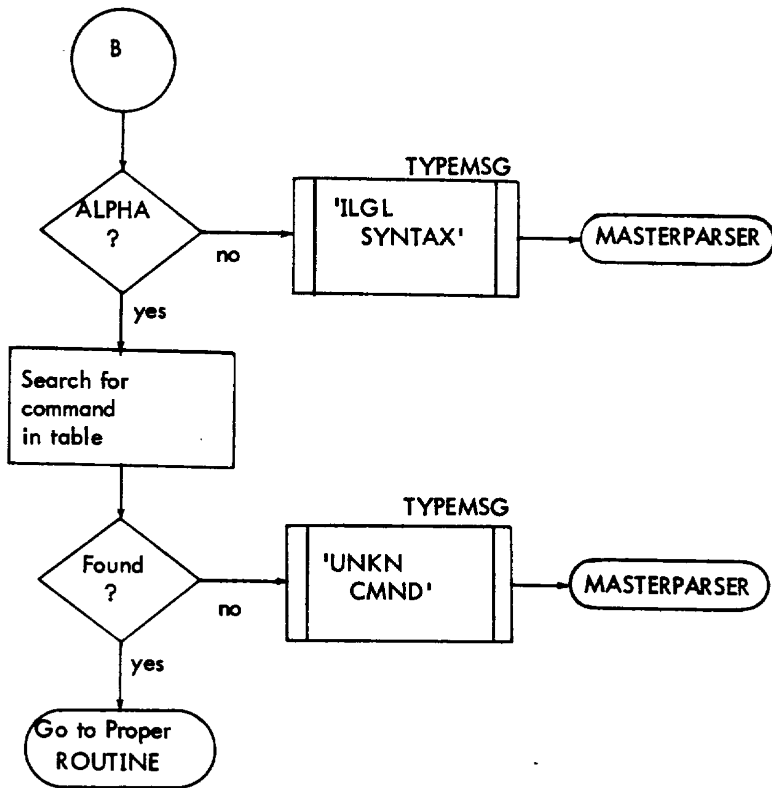



Figure 74. Flow Diagram of MASTERPARSER (cont.)

3. Exit:

Normal return is to MASTEREXECUTIVE upon finding a carriage return. Error exits are:

to ILGL$SEQ2 on finding a second sequence number;

to ILGL$SEMICOLON on finding a ";";

to MASTERPARSER via TYPEPERR on finding any other character in which case it prints one of the following:

'-Pn:NOT SEQ #',

'-Pn:ILGL SYNTAX',

'-Pn:NOT COL #',

or

'-Pn:PARAM MISSING'.

4. Operation:

This subroutine uses NEWCDTENTRY to build new CDT entry, CHECK 1CDTENTRY to make sure "CM" is first command, and GETNEXT PARAM to SCAN command text.

PARSE:DE, PARSE:SE

1. Purpose:

Adds an entry to the CDT for DE n[-m] or SE n [-m][, c[, d]].

2. Entry:

B PARSE:DE for DE

B PARSE:SE for SE

This subroutine is invoked via the CBRCHTBL of MASTERPARSER after a command has been identified.

3. Exit:

Normal return destination (MASTEREXECUTIVE) is given in the calling sequence to NXTPRM and will be used upon recognition of a carriage return in the input command buffer.

Alternate normal return is to RESUME$PARSING if semicolon is encountered after the SE command.

Error exits: to ILGL$SEMICOLON if semicolon found; after DE command: to MASTERPARSER via TYPE-PERR after printing one of the following:

'-Pn:NOT SEQ #',

'-Pn:ILGL SYNTAX',

or

'-Pn:NOT COL #'.

4. Operation:

This subroutine uses:

NEWCDTENTRY to establish an entry in the CDT for this command;

GETNEXTPARAM to scan the input text;

ADJINT to format sequence number as integer * 1000;

REPSEQ to duplicate sequence number if only one given;

CHECK 1CDTENTRY to make sure this is the only entry in the CDT;

and ADDCDTPARAM to add to the CDT.

PARSE:EDIT

1. Purpose:

Adds an entry to the CDT for EDIT fid.

2. Entry:

B PARSE:EDIT

This subroutine is invoked via the CBRCHTBL of MASTERPARSER after a command has been identified.

3. Exit:

Normal return is to MASTEREXECUTIVE upon finding a carriage return. Error returns:

to ILGL$SEMICOLON on finding a ";";

to MASTERPARSER via TYPEPERR on finding any other character where it prints '-Pn:ILGL SYNTAX'.

4. Operation:

This subroutine uses:

NEWCDTENTRY to build new CDT entry;

CHECK 1CDTENTRY to ensure that EDIT is first command in CDT;

GETFILEID;

ADDCDTPARAM to add entry to CDT;

and GETNEXTPARAM to scan for carriage return.

The scan may be extended to obtain an additional file name and a sequence number and increment.

PARSE:END

1. Purpose:

Adds an entry to the CDT for END.

2. Entry:

B PARSE:END

This subroutine is invoked via the CBRCHTBL of MASTERPARSER after a command has been identified.

3. Exit:

Normal return is to MASTEREXECUTIVE on finding a carriage return. Error returns are:

to ILGL$SEMICOLON on finding a semi-colon (;);

to finding MASTERPARSER via TYPEPERR

on finding another character where it prints '-Cn:ILGL SYNTAX'.

4. Operation:

This subroutine uses:

NEWCDTENTRY to build a new CDT entry;

CHECK 1CDENTRY to ensure END is first command;

ADDCDTPARAM to put the "NS" keyword in the CDT;

and get GETNEXTPARAM to scan for carriage return or the "NS" keyword.

PARSE:NO

1. Purpose:

Adds an entry to the CDT for END or NO.

2. Entry:

B PARSE:NO

This subroutine is invoked via the CBRCHTBL of MASTERPARSER after a command has been identified.

3. Exit:

Normal return is to MASTEREXECUTIVE on finding a carriage return. Error returns are:

to ILGL$SEMICOLON on finding a semi-colon (;);

to MASTERPARSER via TYPEPERR on finding another character where it prints '-Cn:ILGL SYNTAX'.

4. Operation:

This subroutine uses:

NEWCDTENTRY to build a new CDT entry:

CHECK 1CDTENTRY to make sure NO is first command;

and GETNEXTPARAM to scan for carriage return.

PARSE:FD, PARSE:FS and PARSE:FT

1. Purpose:

   Adds an entry to the CDT for $\begin{Bmatrix} FD \\ FS \\ FT \end{Bmatrix}$ n [-m],/STRG/[,c[,d]].

2. Entry:

   B PARSE:FD

   B PARSE:FS

   B PARSE:FT

   This subroutine is invoked via the CBRCHTBL of MASTERPARSER after a command has been identified.

3. Exit:

   Normal return is to MASTEREXECUTIVE upon recognition of a carriage return in the input command buffer. Error exits are: to ILGL$SEMICOLON on finding a semi-colon;

   to MASTERPARSER via TYPEPERR after printing one of the following:

   '-Pn:NOT SEQ #',

   '-Pn:ILGL SYNTAX',

   '-Pn:NOT STRG',

   '-Pn:NOT COL #',

   or

   '-Pn:PARAM MISSING'.

4. Operation:

   The following subroutines are used:

   NEWCDTENTRY to build a new CDT entry;

   CHECK 1CDTENTRY to ensure that there is only one CDT entry;

   ADJINT to format sequence number as an integer * 1000;

   REPSEQ to duplicate sequence number if only one given;

   ADDCDTPARAM to add to the CDT;

   GETNEXTPARAM to scan the input text;

   and TYPEPERR to type error message if second parameter is missing.

PARSE:I:CMND$STRG, PARSE:I:CMND$INTG

1. Purpose:

   Process intraline commands.

2. Entry:

This subroutine is used by GETNEXTPARAM to process an intraline command of the form STRG (a character string enclosed in slashes) or INTG (a numeric string whose value is less than 1000). The entry point addresses PARSE:I:CMND$STRG for a character string and PARSE:I:CMND$INTG for a numeric string are passed to GETNEXTPARAM in its calling sequence as the addresses to be branched to on finding such a recognizable string.

3. Exit:

| Branch is made to: | On finding: | Prints message: |
|---|---|---|
| MASTERPARSER | No match | '-Cn:UNKN CMND' |
| MASTERPARSER | Cmnds not in order | '-Cn:ILGL SYNTAX' |
| MASTERPARSER | Order improper | '-Cn:NOT CNT' |
| MASTERPARSER | String not of form/ST1/ where expected | '-Cn:NOT STRG' |
| RESUME$PARSING | Semi-colon | |
| MASTEREXECUTIVE | Carriage return | |

4. Operation:

This subroutine calls NEWCDTENTRY to build new CDT entry (character or integer). It calls ADDCDTPARAM to add the new parameter ( character or integer). It searches table of intraline commands to find a match; if found, it processes appropriate command following the string. It uses GETNEXTPARAM for commands that require further scanning.

The flow of PARSE:I, CMND$STRG, and PARSE:ICMND$INTG is given in Figure 76.

PARSE:IN, PARSE:IS

1. Purpose:

Adds an entry to the CDT for IN n[,i] or IS n[,i].

2. Entry:

B PARSE:IN

B PARSE:IS

This subroutine is invoked via the CBRCHTBL of MASTERPARSER after a command has been identified.

3. Exit:

B GET$SEQ$INCR to process the n[,i]portion of the command.

4. Operation:

This subroutine uses NEWCDTENTRY to build new CDT entry and CHECK1CDTENTRY to make sure IN or IS is first command.

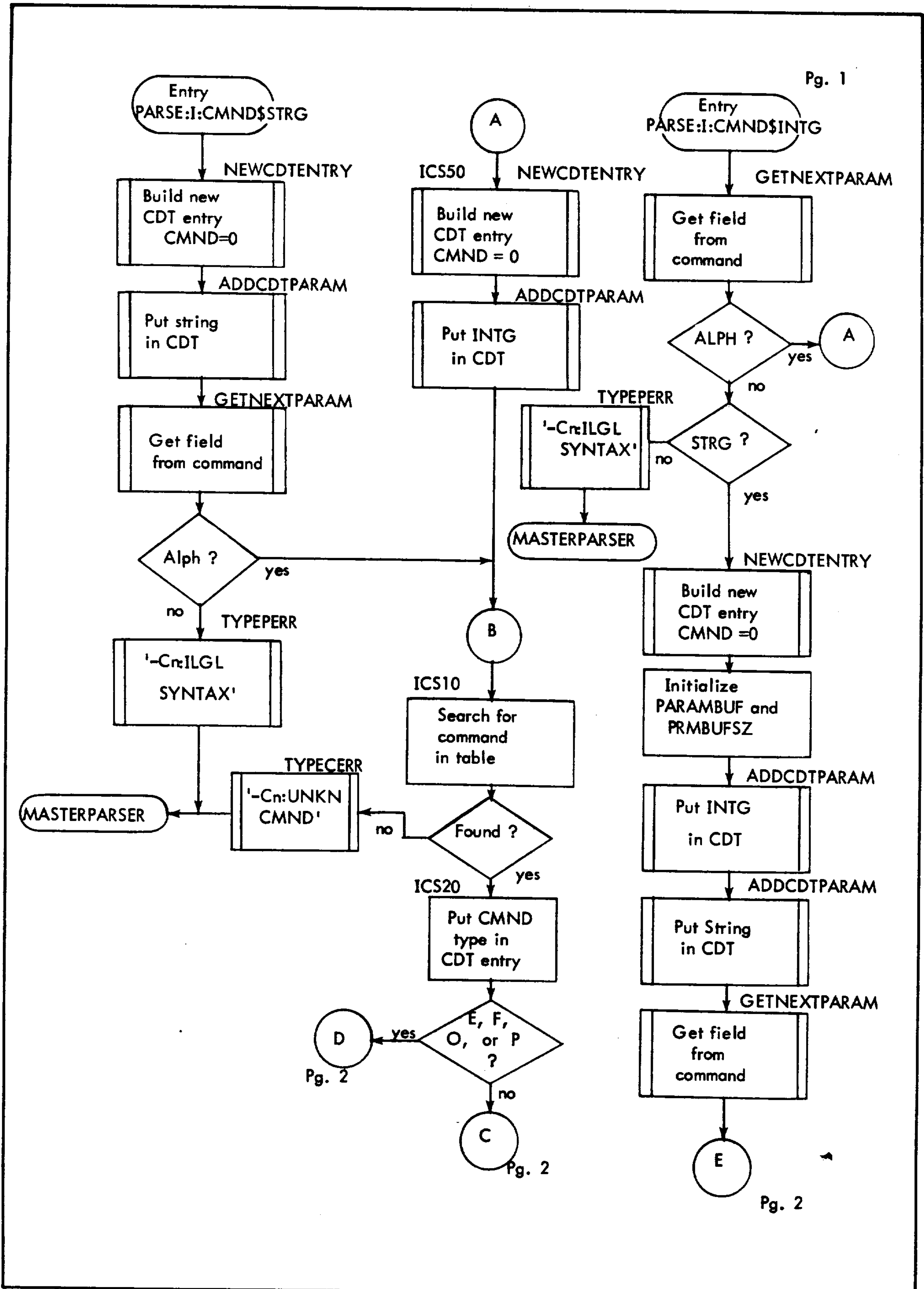



Figure 75. Flow Diagram of PARSE:I:CMND$STRG and PARSE:I:CMND$INTG

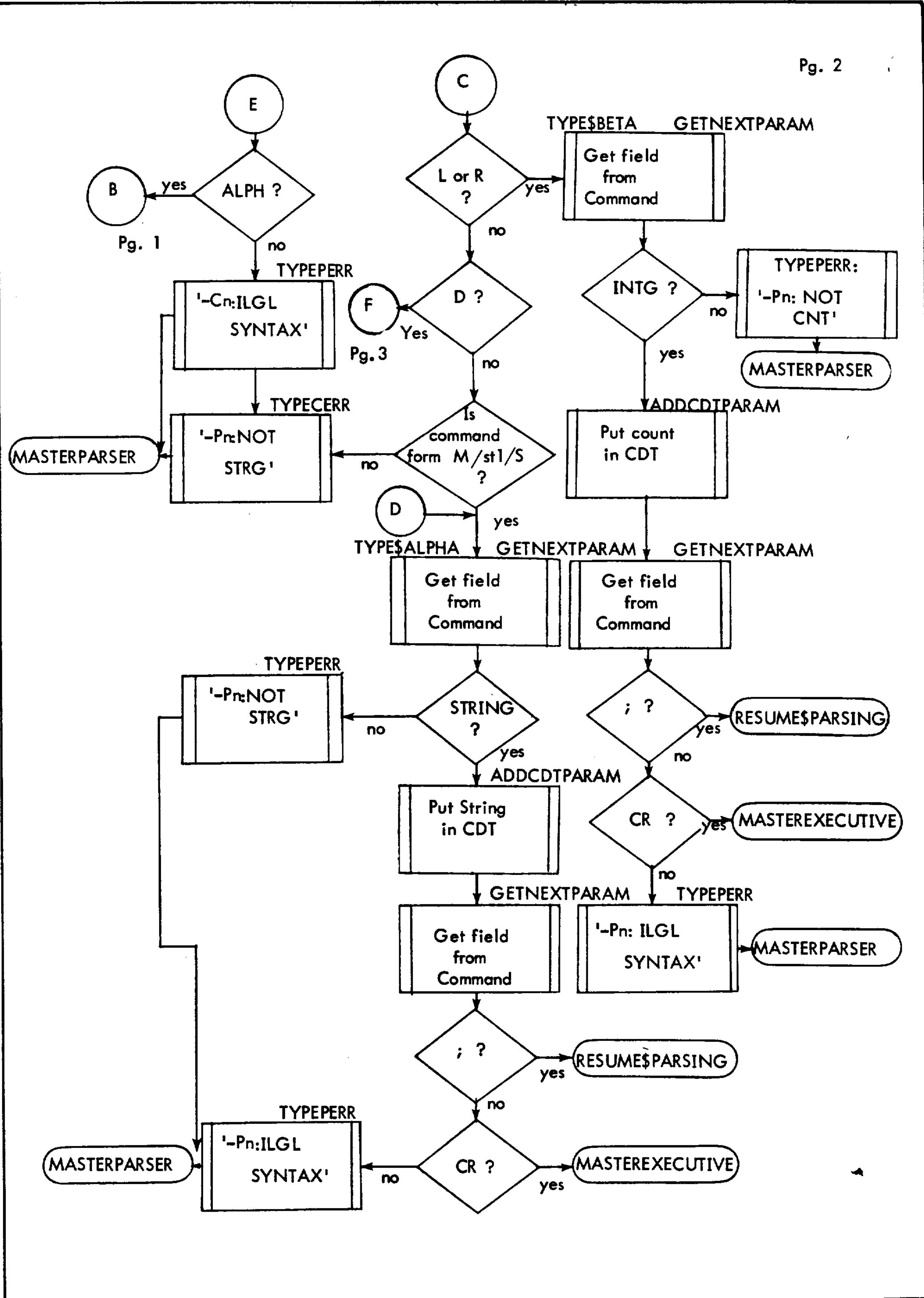



Figure 75. Flow Diagram of PARSE:I:CMND$STRG and PARSE:I:CMND$INTG (cont.)

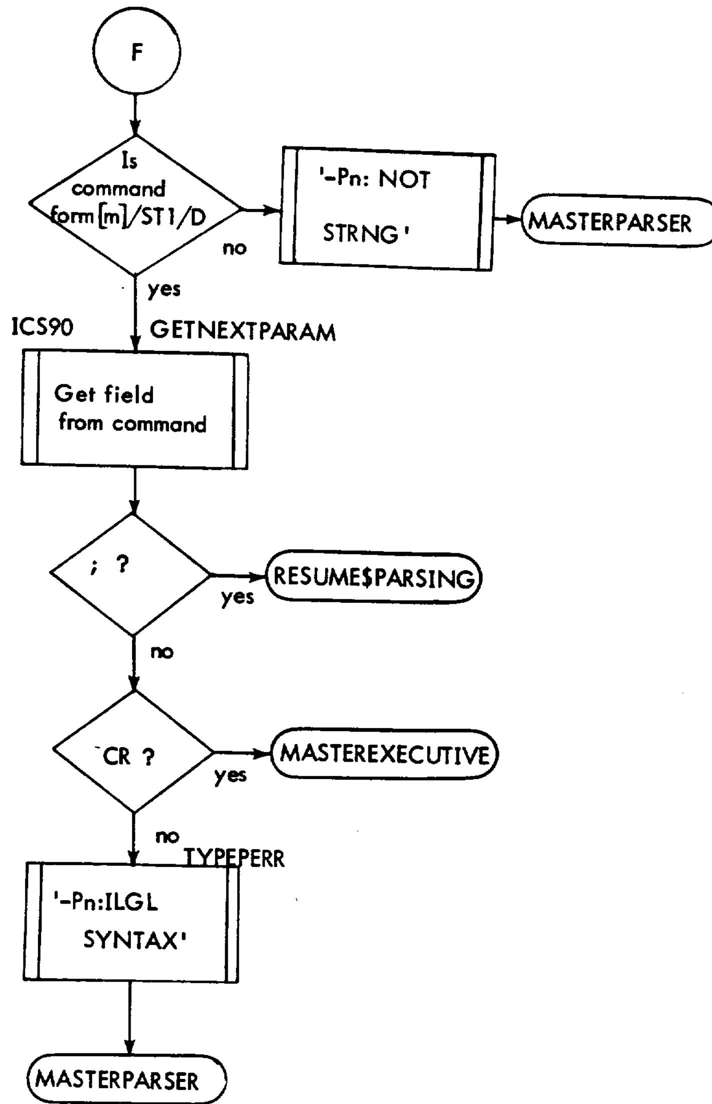



Figure 75. Flow Diagram of PARSE:I:CMND$STRG and PARSE:I:CMND$INTG (cont.)

PARSE:MD, PARSE:MK

1. Purpose:

   Adds an entry to the CDT for MD n [-m], k[-p][,i] or MK n [-m], k[-p][,i].

2. Entry:

   B PARSE:MD

   B PARSE:MK

   This subroutine is invoked via the CBRCHTBL of MASTERPARSER after a command has been identified.

3. Exit:

   Normal return is B GET$INCREMENT. Error exit is to MASTERPARSER via TYPEPERR after printing one of the following:

   '-Pn:NOT SEQ #'.

   '-Pn:ILGL SYNTAX'.

4. Operation:

   This subroutine uses:

   NEWCDTENTRY to add a new entry to the CDT;

   CHECK 1CDTENTRY to make sure this is the first CDT entry;

   ADJINT to format sequence number as an integer * 1000;

   REPSEQ to duplicate sequence number if only one given;

   ADDCDTPARAM to add to the CDT;

   and GETNEXTPARAM to scan the input text.

PARSE:RF

1. Purpose:

   Adds an entry to the CDT for RF.

2. Entry:

   B PARSE:RF

   This subroutine is invoked via the CBRCHTBL of MASTERPARSER after a command has been identified.

3. Exit:

   Normal return is to MASTEREXECUTIVE upon finding a carriage return or to RESUME$PARSING on finding a ";". Error return is to MASTERPARSER via TYPEPERR on finding another character.

4. Operation:

   It uses NEWCDTENTRY to build a new CDT entry and GETNEXTPARAM to scan command text.

PARSE:RN

1. Purpose:

   Adds an entry to the CDT for the command RN, renumber.

2. Entry:

   B PARSE:RN

   This subroutine is invoked via the CBRCHTBL of MASTERPARSER after a command has been identified.

3. Exit:

   Normal return is to MASTEREXECUTIVE upon recognition of a carriage return in the input command buffer.

   Error exit: to ILGL$SEMICOLON on finding a semi-colon;

   to MASTERPARSER via TYPEPERR after printing one of the following:

   '-Pn:NOT SEQ #',

   '-Pn:ILGL SYNTAX',

   or

   '-Pn:PARAM MISSING'.

4. Operation:

   This subroutine uses:

   NEWCDTENTRY to build a new CDT entry;

   CHECK1CDTENTRY to make sure RN is the first command;

   ADDCDTPARAM to add to the CDT;

   GETNEXTPARAM to scan the input text;

   ADJINT to format sequence number as integer * 1000; and

   TYPEPERR to type error message.

PARSE:SS, PARSE:ST and PARSE:JU

1. Purpose:

   Adds an entry to the CDT for SS n [,c[,d]] or ST n [,c[,d]] or JU n.

2. Entry:

   B PARSE:SS

   B PARSE:ST

   B PARSE:JU

   This subroutine is invoked via the CBRCHTBL of MASTERPARSER after a command has been identified.

3. Exit:

Normal return is to MASTEREXECUTIVE on finding a carriage return for JU or to GET$COL#PAIR for SS or ST. Error returns are:

to ILGL$SEQ2 on finding a second sequence number;

to ILGL$SEMICOLON on finding a semi-colon;

to MASTERPARSER via TYPEPERR after printing:

'-Pn:NOT SEQ #'

or

'-Pn:ILGL SYNTAX'.

4. Operation:

This subroutine uses:

NEWCDTENTRY to build a new CDT entry;

GETNEXTPARAM to scan the input text;

ADJINT to format sequence number as integer * 1000;

CHECK1CDTENTRY to make sure this is the only entry in the CDT (if not JU); and

ADDCDTPARAM to add to the CDT.

PARSE:TC, PARSE:TS and PARSE:TY

1. Purpose:

Adds an entry to the CDT for TS [n-m][,c[,d]], TY[n-m][,c[,d]], or TC n[-m][,c[,d]].

2. Entry:

B PARSE:TC

B PARSE:TS

B PARSE:TY

This subroutine is invoked via the CBRCHTBL of MASTERPARSER after a command has been identified.

3. Exit:

For TC: a record number must be specified; normal exit is a B GET$COL#$PAIR; error exit is to MASTERPARSER via TYPEPERR on finding no record number in which case it prints '-Pn:NOT SEQ#'.

For TS and TY: normal exit is either to MASTEREXECUTIVE on finding a carriage return or to GET$COL#$PAIR to process record range; error exit is to MASTERPARSER via TYPEPERR on finding illegal character, in which case it prints '-Cn:ILGL SYNTAX'.

4. Operation:

This subroutine uses:

NEWCDTENTRY to build new CDT entry;

CHECK1CDTENTRY to ensure that TS, TY or TC is first command to be added to CDT;

ADJINT to form sequence number as integer * 1000;

REPSEQ to repeat "n" as "m" if only "n" given; and

ADDCDTPARAM to add to CDT.

## MASTEREXECUTIVE Routine

1. Purpose:

This is the master routine to execute commands in the CDT. It resets CDTADR to point to start of CDT, gets command from start of CDT, checks whether the proper mode is being used for this command, and calls it if so.

2. Entry:

B MASTEREXECUTIVE

This routine's address is given to NXTPRM[†] as the branch location following identification of a carriage return in the input command buffer.

3. Exit:

B MASTERPARSER after finding erroneous data in CDT or after properly executing all commands in CDT.

4. Operation:

MASTEREXECUTIVE performs the following functions: restores last default value of blank perservation flag; sets ALLOK flag to show that "ALL" mode is potentially legal; if a file command, ensures that input file is present and keyed; if intra-record command, ensures that set mode is in effect; executes command from CDT via F: , R: , or I: routines; if all-flag set and command is intra-record, repeats for all occurrences. This routine, in effect, controls execution of the set and step commands in addition to serving as driver for CDT command execution.

The flow for MASTEREXECUTIVE is given in Figure 76.

F:BLANK$PRESERV

1. Purpose:

Sets SUBFLAG for blank preservation mode.

2. Entry:

F:LNK is the linkage register. This subroutine is entered via BAL, F:LNK F:BLANK$PRESERV.

---

[†]A procedure that generates a calling sequence to GETNEXTPARAM.

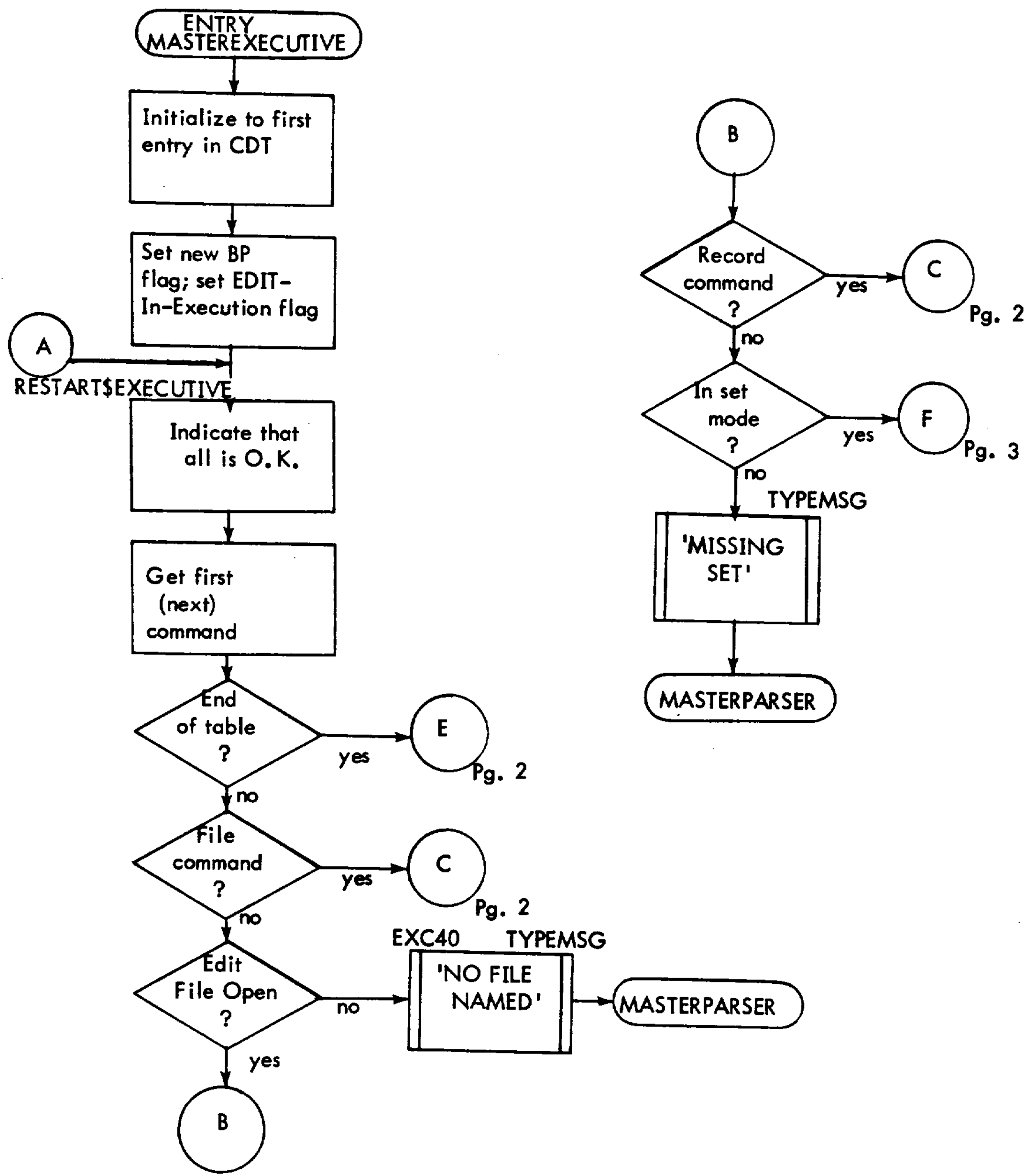



Figure 76. Flow Diagram of MASTEREXECUTIVE

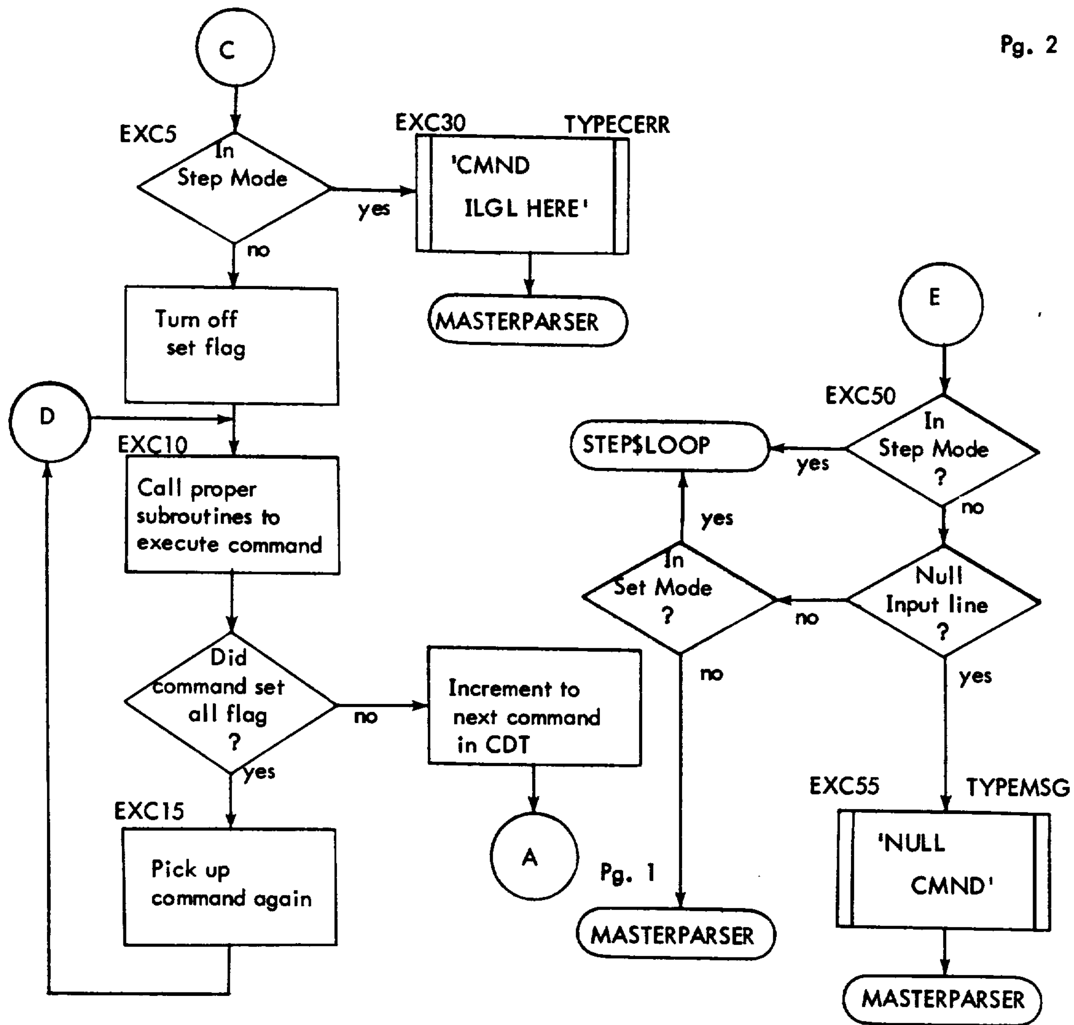



Figure 76. Flow Diagram of MASTEREXECUTIVE (cont.)

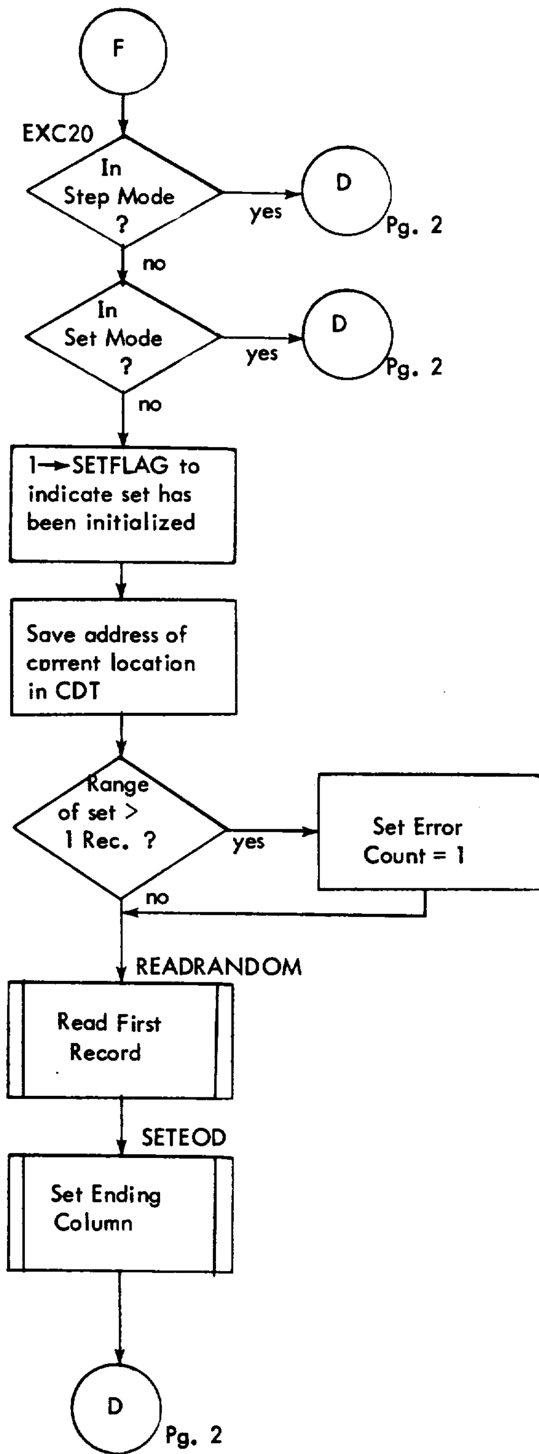



Figure 76. Flow Diagram of MASTEREXECUTIVE (cont.)

3. Exit is via B *F:LNK which results in return to MASTEREXECUTIVE.

4. Operation:

   If mode neither on or off it types "-NOT ON/OFF"; otherwise it sets SUBFLAG to current mode.

F:EDIT

1. Purpose:

   Performs initialization in preparation for editing a file.

2. Entry:

   F:LNK is the linkage register. This subroutine is entered via BAL, F:LNK F:EDIT.

3. Exit:

   B *F:LNK, which results in return to MASTEREXECUTIVE.

4. Operation:

   If an EDIT command is already in effect, the scratch file is closed after a SAVE is performed on the subject file if it exists. If only a scratch file is specified for the new EDIT operation, it is reopened (on the assumption that it has been built previously as an EDIT scratch file) using routine OPENSCR. If a subject file is specified, the method of indexing (specified key start and step, default key start and step, or key contained in each subject file record) is determined, and routine BUILDSCR is called to build the scratch file from the records in the subject file.

   The flow for F:EDIT is given in Figure 77.

F:END

1. Purpose:

   Closes scratch file if open and causes return to the monitor.

2. Entry:

   F:LNK is the linkage register. This subroutine is entered via BAL, F:LNK F:END.

3. Exit:

   It returns to the monitor via CAL3, 6 0.

4. Operation:

   If an EDIT command is not yet in effect, F:END does an EXIT service call. If an EDIT command is in effect and the NS option of END was not specified, F:END calls SAVESCR to save the subject file. It then closes the scratch file and performs the EXIT call. If NS was specified, the save operation is omitted.

   The flow for F:END is given in Figure 78.

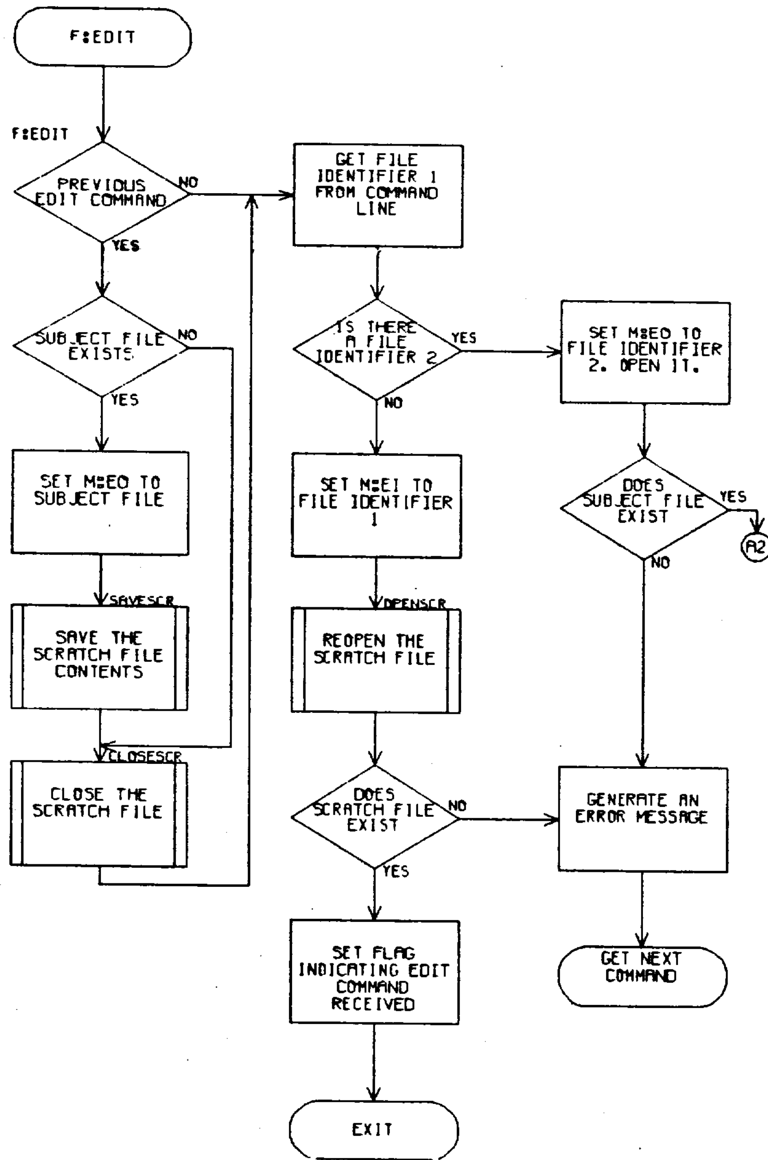



Figure 77. Flow Diagram of F:EDIT

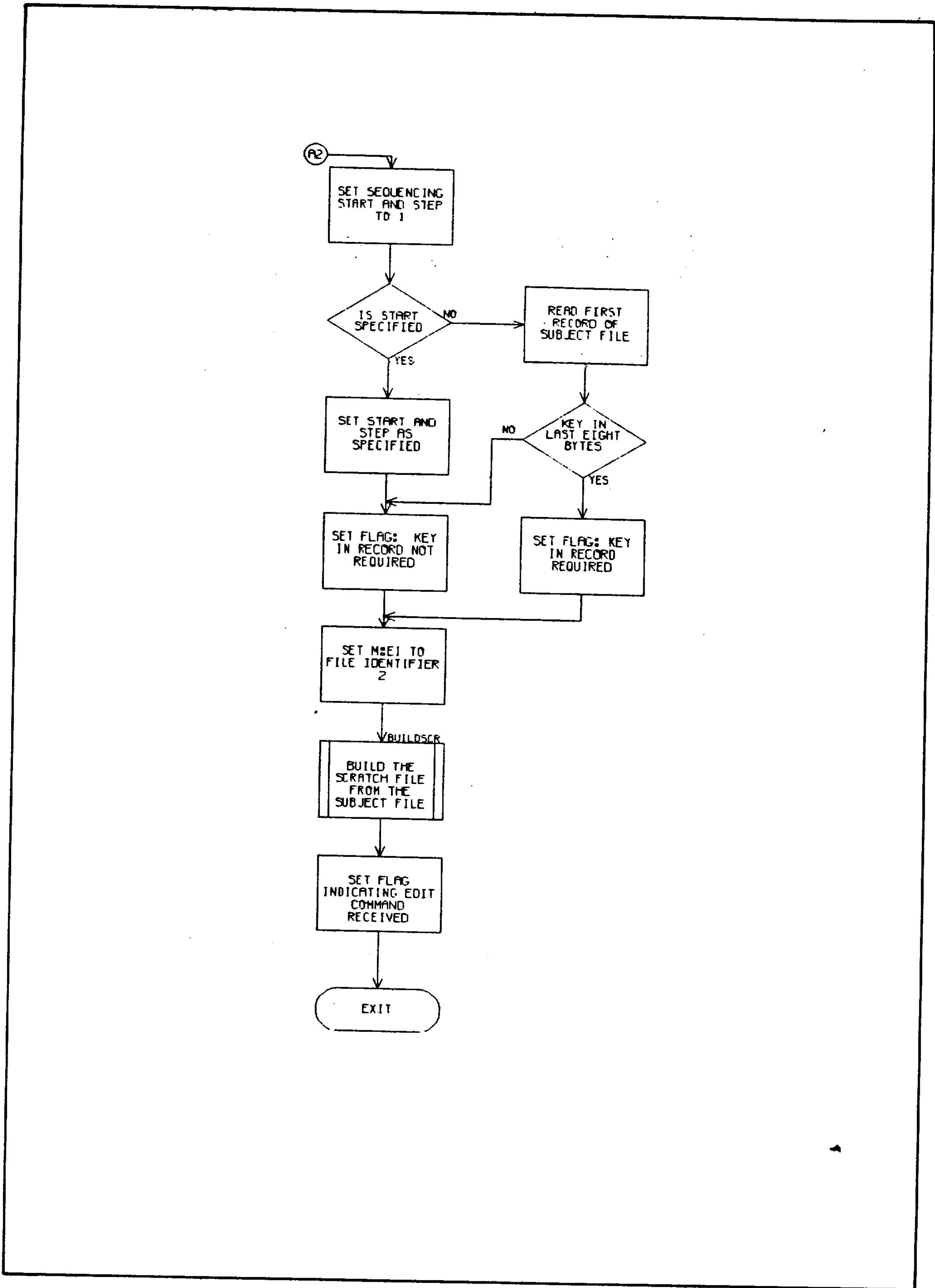



Figure 77. Flow Diagram of F:EDIT (cont.)

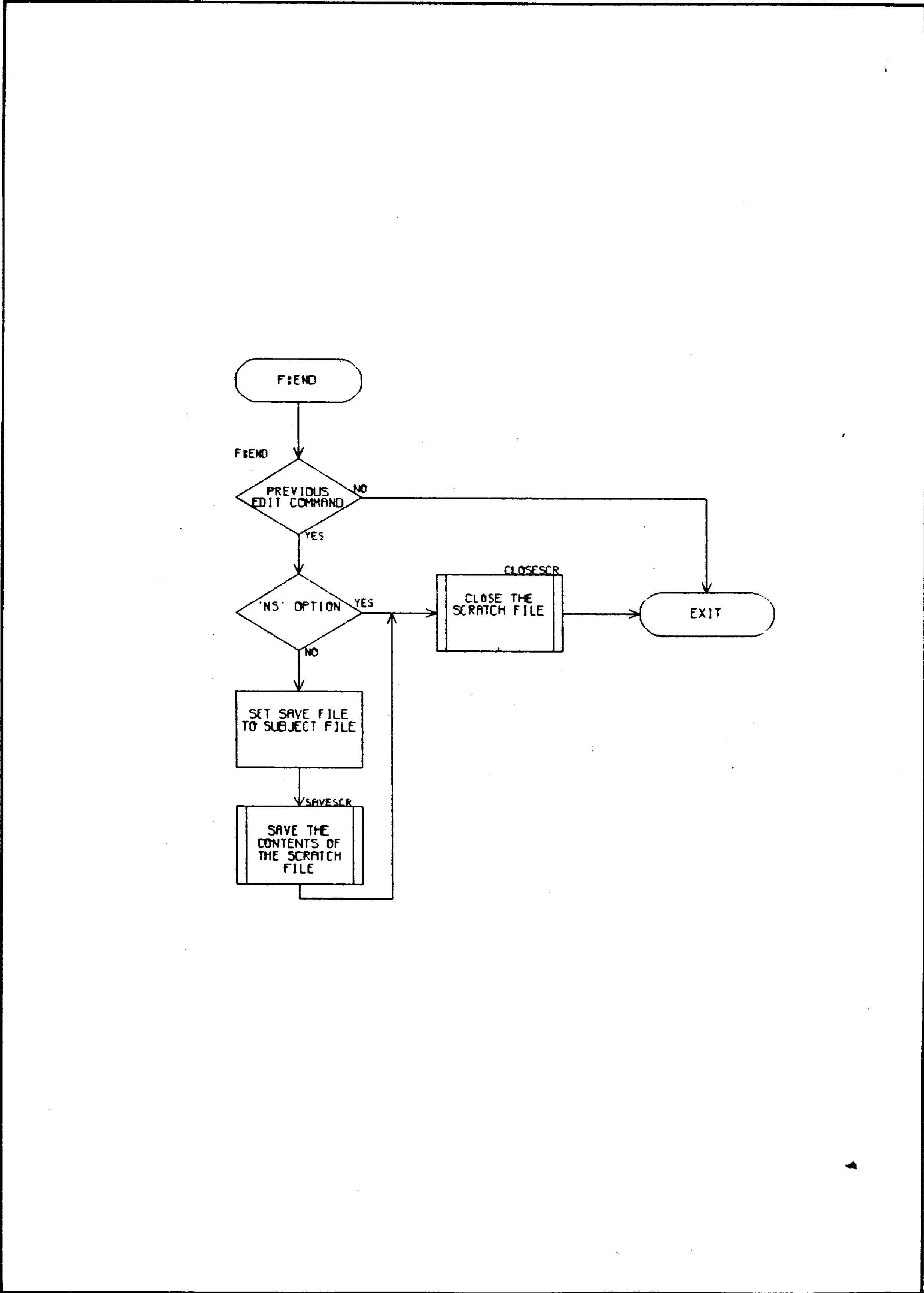



Figure 78. Flow Diagram of F:END

R:COMMENTARY

1. Purpose:

Executes the insert commentary command, CM.

2. Entry:

R:LNK is the linkage register. This subroutine is entered via BAL, R:LNK R:COMMENTARY, from MASTEREXECUTIVE.

3. Exit:

B MASTERPARSER

4. Operation:

If column number exceeds 140, it prints '-P2:COL ERROR' via TYPEMSG and exits. It uses READRANDOM to read specified record: if not found, it prints '-Pn:NO SUCH REC' via TYPEMSG and exits.

It uses: TYPESEQ to type sequence number prompt and READTELETYPE2 to read commentary (if CR, it exits).

It moves commentary into record: if extends beyond 140 characters in all it types '--OVERFLOW' via TYPEMSG and goes on to next record; commentary is blank – filled to the right.

It uses: SETEOD to insert carriage return if CR ON;

WRITERANDOM to write the record; and READSEQUEN to read next record.

It types "--EOF HIT' if EOF found and exits.

It repeats sequence number prompt and continues as previously described until a record having only CR is found or until EOF is encountered.

F:SAVE

1. Purpose:

Executes the SAVE command.

See Figure 79.

R:DELETE

1. Purpose:

Performs a request to delete records.

2. Entry:

R:LNK is the linkage register.

Note: F:LNK and R:LNK are synonymous. This subroutine is entered via BAL, R:LNK R:DELETE in MASTEREXECUTIVE.

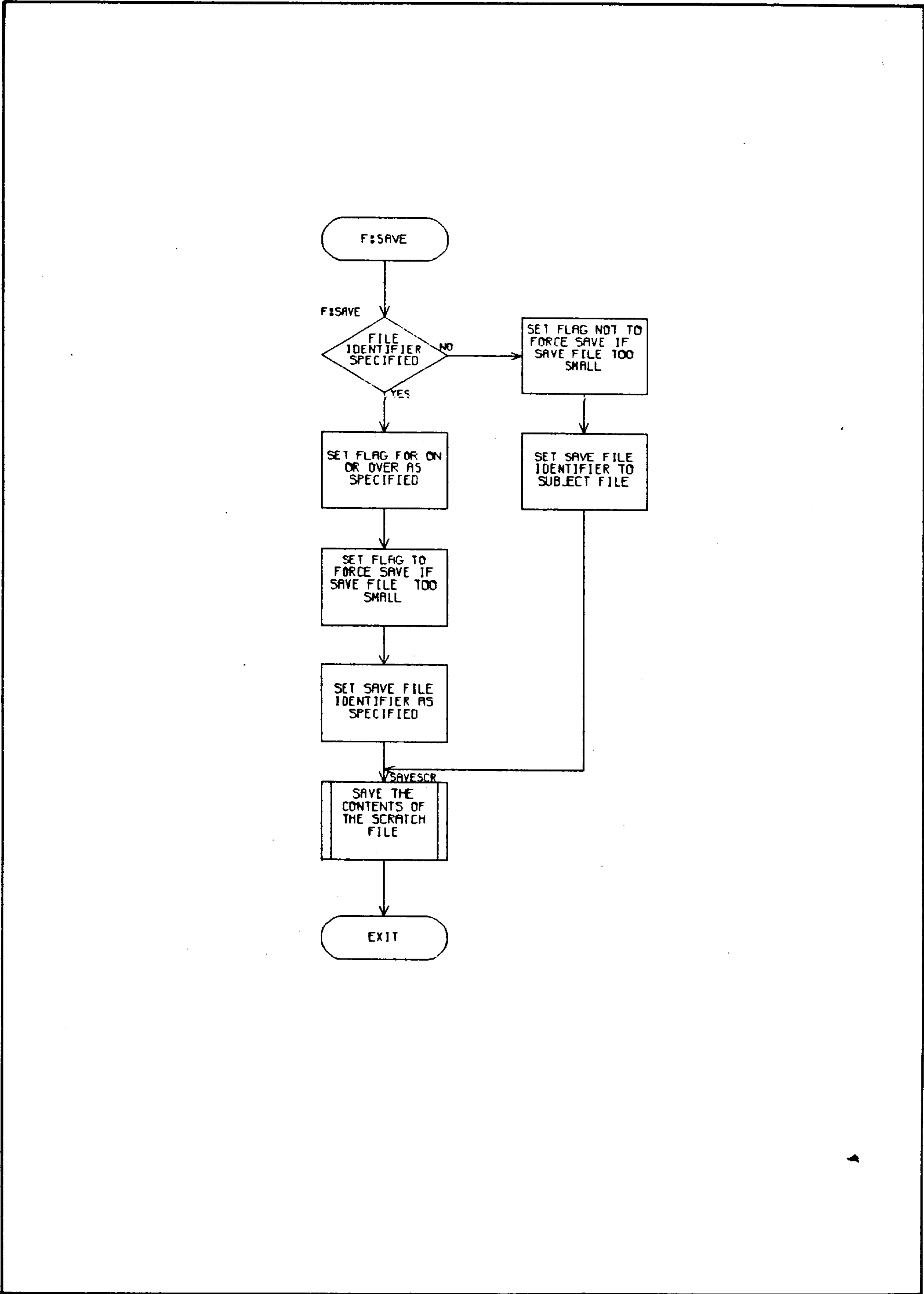



Figure 79. Flow Diagram of SAVE Command

3. Exit:

   B *R:LNK

4. Operation:

   It sets: P1 = first sequence number and P2 = last sequence number.

   It uses DELETE to perform deletion of records specified by the range. If nothing is deleted it uses TYPEMSG to type '-NOTHING TO DE'.

## R:FIND$SEQUENCE, R:FIND$DELETE and R:FIND$TYPE

1. Purpose:

   Processes the record commands FS, FD, and FT.

2. Entry:

   R:LNK is the linkage register. This subroutine is entered via BAL, R:LNK

   { R:FIND$SEQUENCE<br>R:FIND$DELETE<br>R:FIND$TYPE }

   from MASTEREXECUTIVE.

3. Exit:

   B *R:LNK

4. Operation:

   It sets: FIRSTSET = first sequence number in CDT and LASTSET = second sequence number in CDT.

   It uses PROCESSCOL#PAIR to process the column number parameters and READNXTRANDOM to read first sequence number specified or next highest.

   It uses: FINDMATCH to scan the record for the specified string and READSEQUEN to read further if string not in record read via READNXTRANDOM.

   For: FS it uses TYPESEQ to type sequence number of record;

   FT it uses SETEOD to insert (RET) if CR ON;

   FD it uses DELETERECORD to delete the record.

   It repeats the scan for the desired record range.

   It wraps up: for all three command types TYPEMSG is used to type '--NONE' if no matches found for FD TYPEMSG is used to type '--XXX RECS DLTED'.

   If EOF is reached during search, '--EOF HIT' is typed via TYPEMSG.

   The flow of R:FIND$SEQUENCE, R:FIND$DELETE, and R:FIND$TYPE is given in Figure 80.

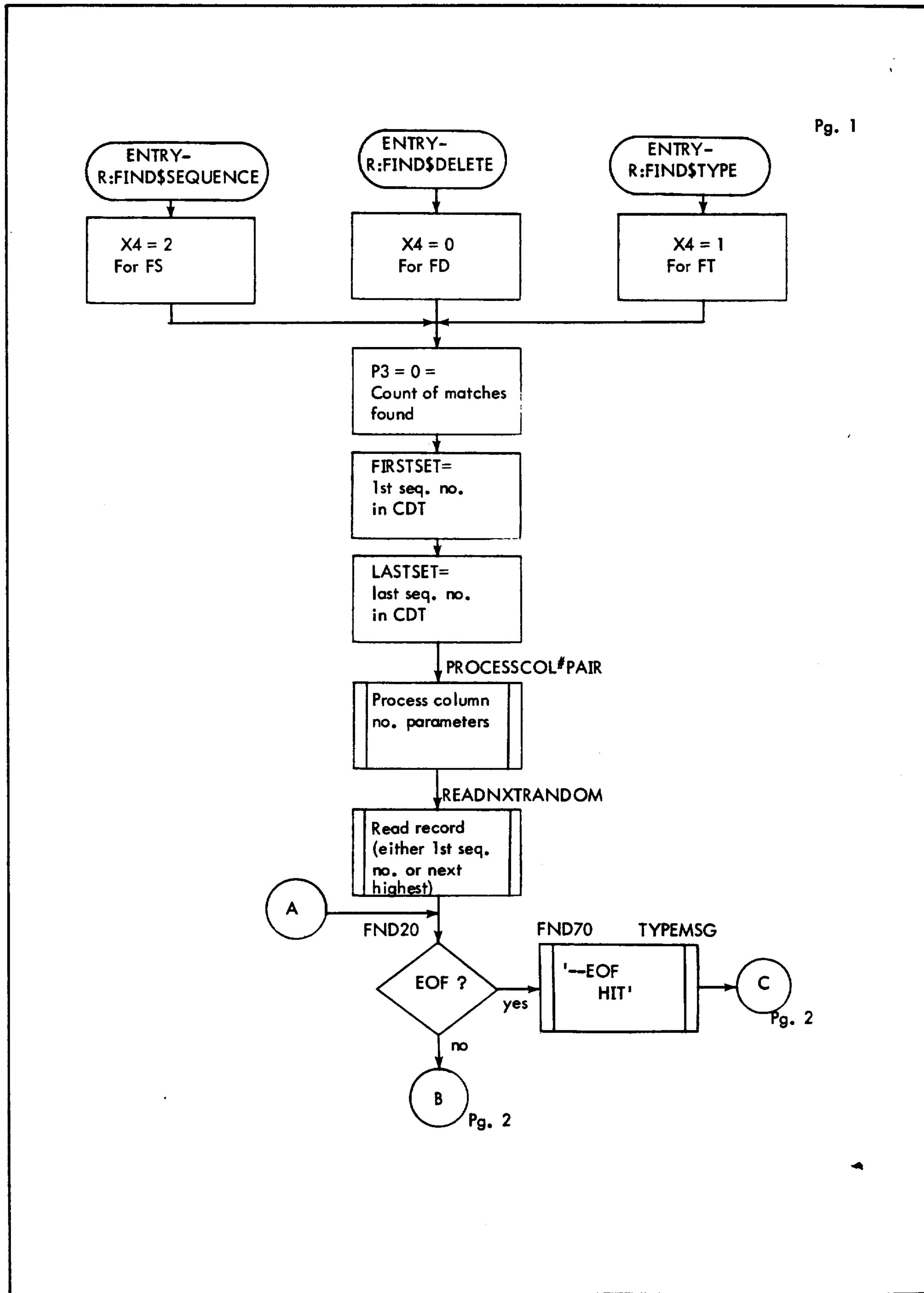



Figure 80. Flow Diagram of R:FIND$SEQUENCE, R:FIND$DELETE, and R:FIND$TYPE

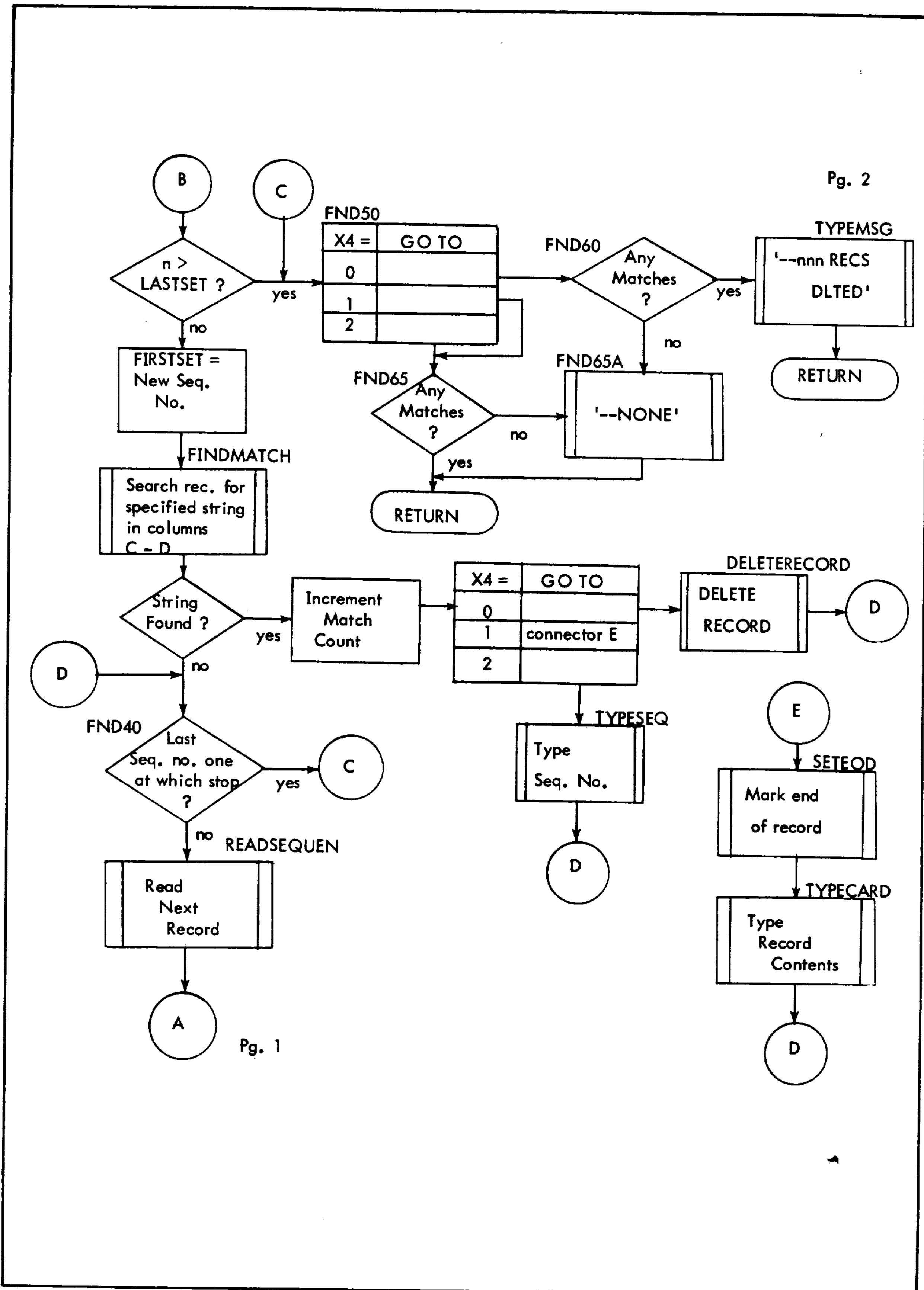



Figure 80. Flow Diagram of R:FIND$SEQUENCE, R:FIND$DELETE, and R:FIND$TYPE (cont.)

R:INSERT$SUP$SEQ, R:INSERT

1. Purpose:

   Inserts records with or without sequence number prompting.

2. Entry:

   R:LNK is the linkage register. This subroutine is entered via BAL, R:LNK R:INSERT (Sets X4=0) or BAL, R:LNK R:INSERT$SUP$SEQ (Sets X4=2) in MASTEREXECUTIVE.

3. Exit:

   B *R:LNK

4. Operation:

   The CDT is examined to pick up starting sequence number and, if present, increment it.

   The subroutine resets DFLTINCR and new default increment. A record is read using READNXTRANDOM. If the record read is the sequence number requested, the next highest record is read via READSEQUEN. If IN command, TYPESEQ is used to prompt with sequence number. The record to be inserted is read via READTELETYPE. If none existed, normal exit occurs. The record terminator is set to blank.

   RECSIZE is set. If record size exceeds 140, '--OVERFLOW' is printed via TYPEMSG. SETEOD is used to insert carriage return if CR ON.

   WRITERANDOM is used to write the new card image. The current sequence number is incremented. If more records will fit in the range, process is repeated starting with sequence number prompting above. Otherwise, the Teletype bell is rung twice via TYPEMSG and normal exit is taken.

   If a record to be inserted has sequence number greater than 9999.999, the message '-MAX. SEQ. NO. EXCEEDED' is printed and command processing is stopped.

R:MOVE$DELETE, R:MOVE$KEEP

1. Purpose:

   Processes the commands Move and Delete (MD) and Move and Keep (MK).

2. Entry:

   R:LNK is the linkage register. This subroutine is entered via BAL, R:LNK R:MOVE$DELETE or BAL, R:LNK R:MOVE$KEEP from MASTEREXECUTIVE or via B MVE58, B MVE56, or B MVE40 from F:MERGE.

3. Exit:

   B *R:LNK

4. Operation:

   This subroutine examines the CDT to get parameters from command form

   $\begin{Bmatrix} \text{MD} \\ \text{MK} \end{Bmatrix}$ n[-m],k[-p][,i].

   It sets DFLTINCR = i, or if i not specified = the most recent increment used (1 if none).

It checks both ranges; errors are: '--EOF HIT' and '-RNG OVERLAP' typed via TYPEMSG. (Ranges must be mutually exclusive).

It uses READNXTRANDOM to attempt to read record n: (at MVE58) if not found it types '-NOTHING to MOVE' via TYPEMSG and exits.

It uses DELETE to delete all records in range k-p. If it was possible to find all of them, it reads ahead using READSEQUEN to mark sequence number of next record.

If record n cannot be found, it types '--EOF HIT'. At MVE56 if record m hit or passed: and if m>p, it types '--CUTOFF AT XXX.X (XX.XX)' and if MD, it deletes records n-m, and then exits.

It reads records from range n-m one by one and writes them with sequence number (key) k-p. At MVE40 normal termination yields message '--DONE AT XX.XX' and for MD, source records in range n-m are deleted before exiting.

The flow of R:MOVE$DELETE and R:MOVE$KEEP is given in Figure 81.

R:RENUMBER

1. Purpose:

   Reads old records having sequence numbers specified and writes them with new sequence numbers, deleteing old records where the sequence number already existed.

2. Entry:

   R:LNK is the linkage register. This subroutine is entered via BAL, R:LNK R:RENUMBER in MASTEREXECUTIVE.

3. Exit:

   B *R:LNK

4. Operation:

   It sets P1 = old sequence number; T1 = new sequence number

   It uses: READRANDOM to read old record; WRITENEWRANDOM to write new record with new sequence number; DELETERECORD to delete old record should sequence number already exist; and TYPEMSG to type '-P1:NO SUCH REC' if old record does not exist and '-P2:REC EXISTS' if new record already exists.

R:SET$STEP, R:SET$STEP$TYPE

1. Purpose:

   Executes the SS (Set and Step) or ST (Set, Step and Type) command.

2. Entry:

   R:LNK is the linkage register. This subroutine is entered via BAL, R:LNK R:SET$STEP or BAL, R:LNK R:SET$STEP$TYPE in MASTEREXECUTIVE.

   Auxiliary entry points: FINISH$STEP$LOOP, which is the entry point from I:JUMP, and STEP$LOOP, which is the entry point from MASTEREXECUTIVE if in a step loop.

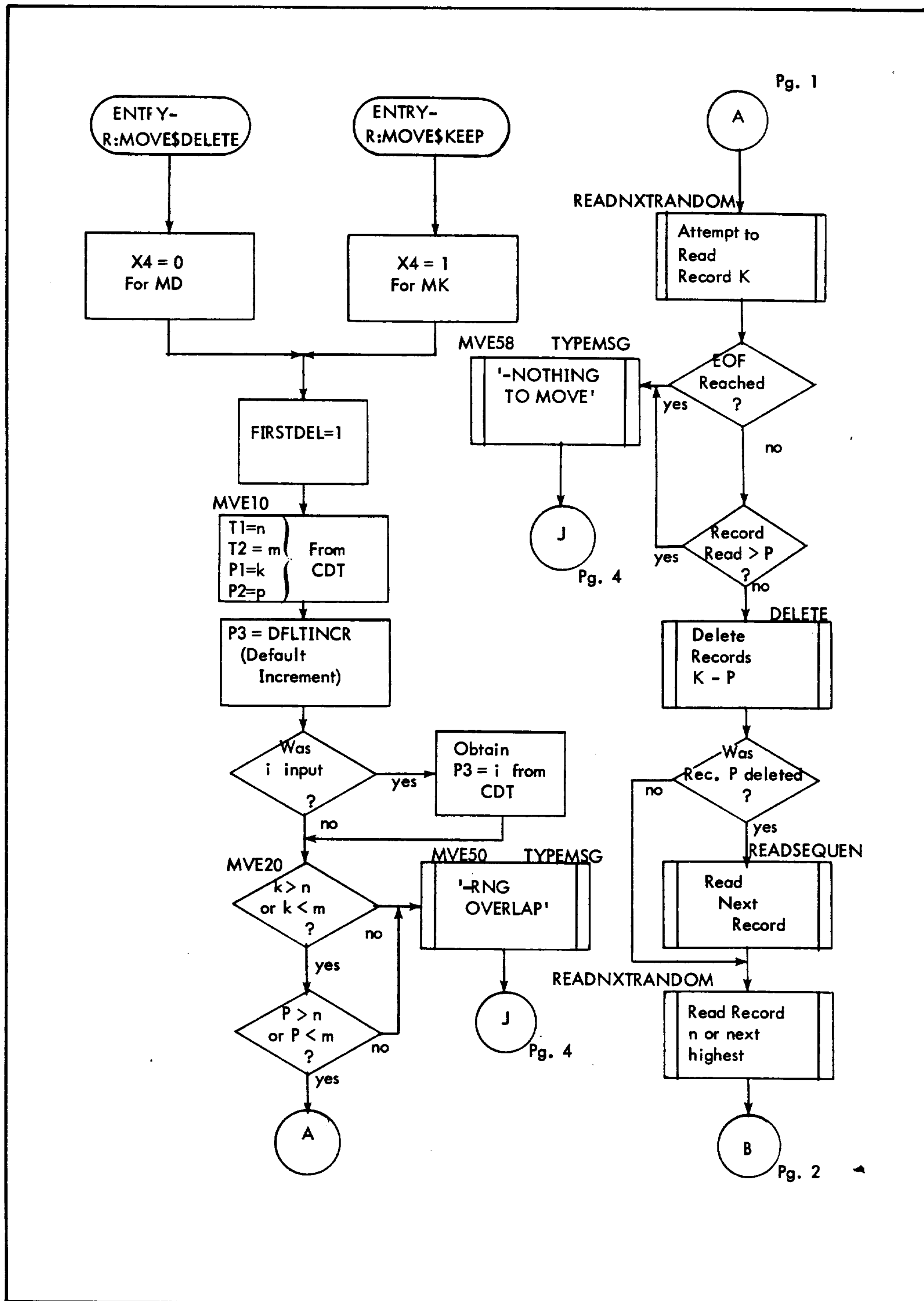



Figure 81. Flow Diagram of R:MOVE$DELETE and R:MOVE$KEEP

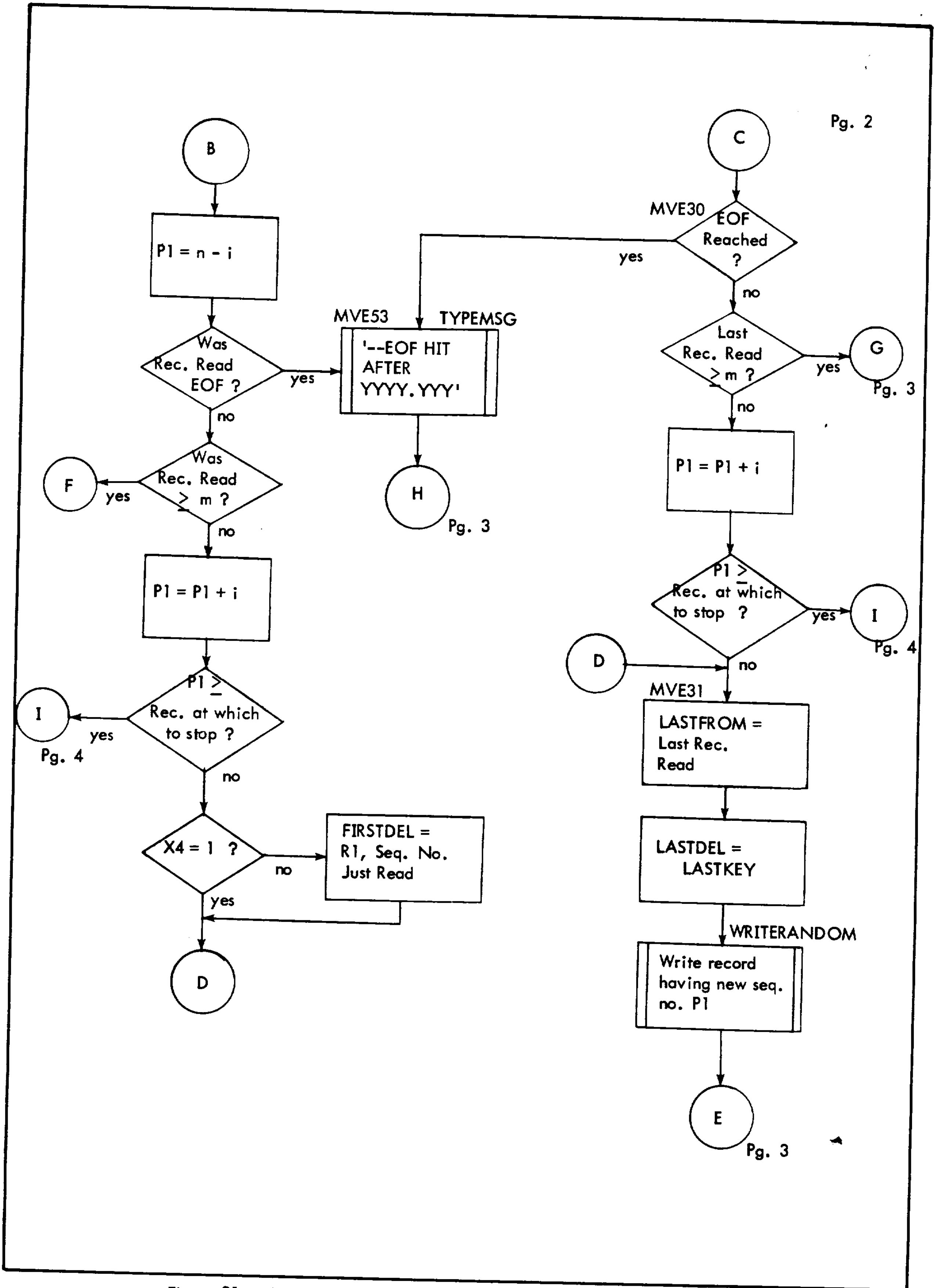
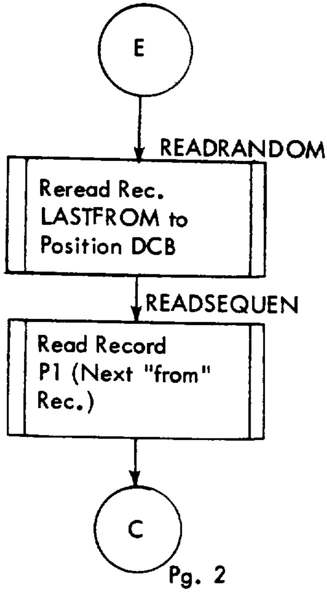



Figure 81. Flow Diagram of R:MOVE$DELETE and R:MOVE$KEEP (cont.)



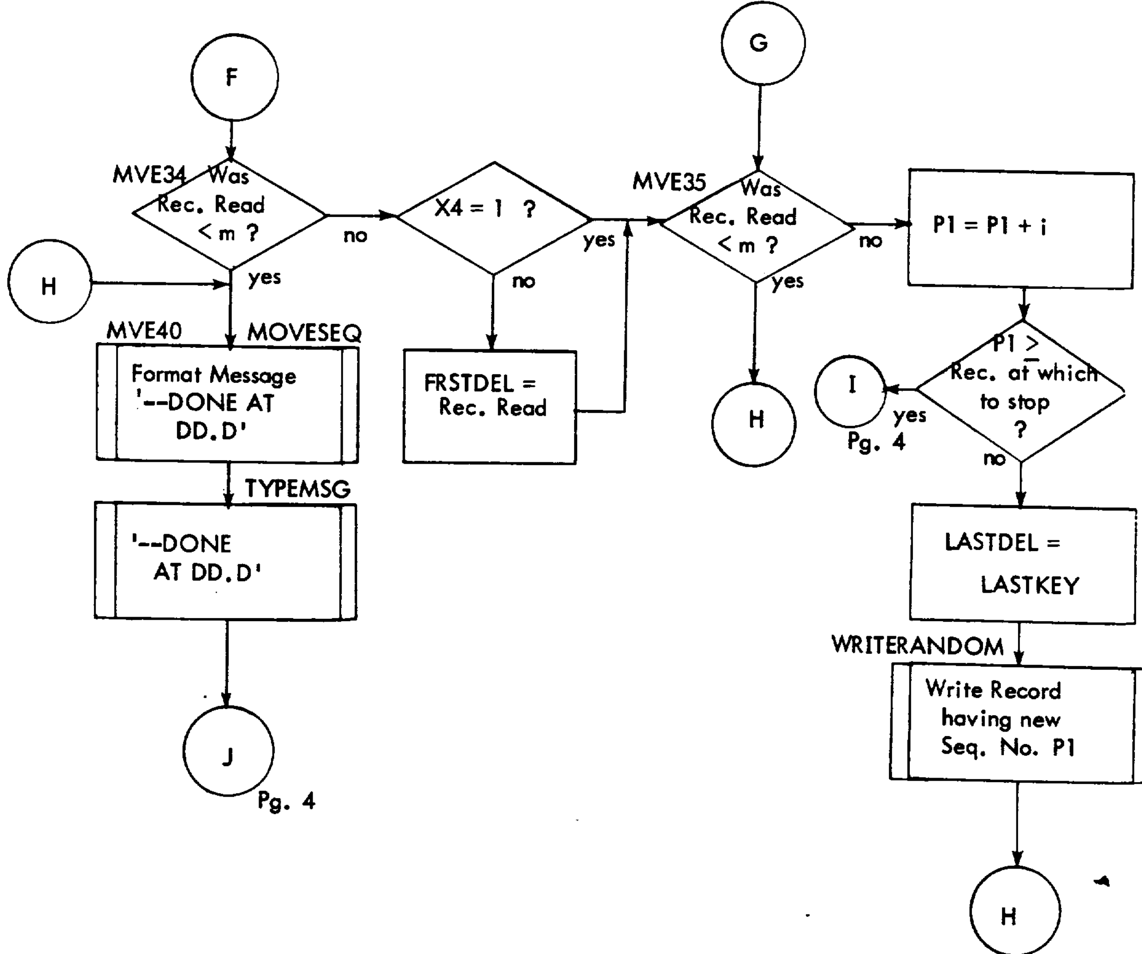




Figure 81. Flow Diagram of R:MOVE$DELETE and R:MOVE$KEEP (cont.)

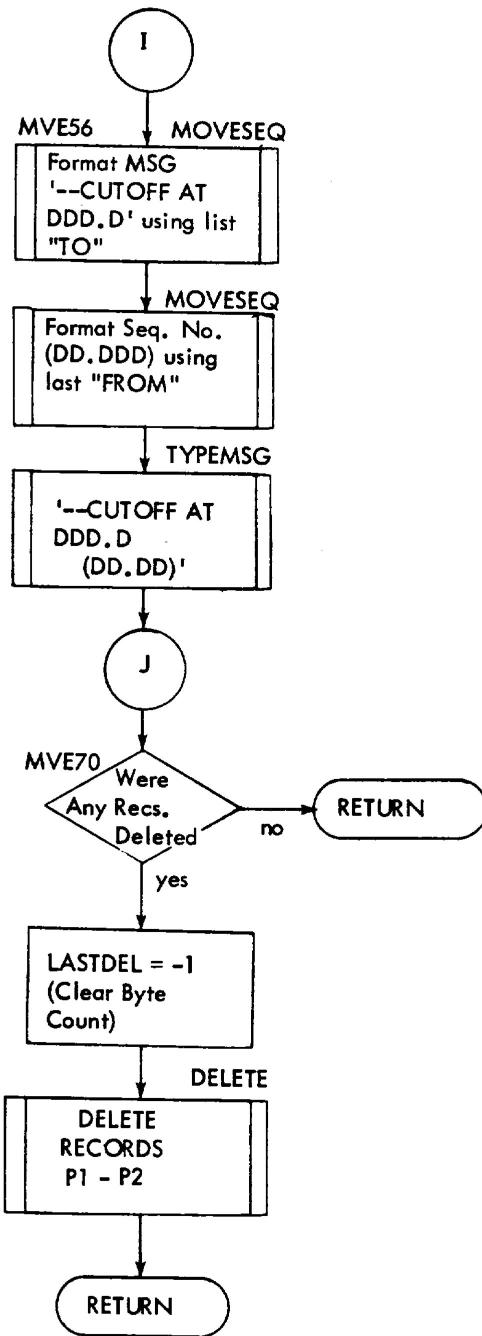



Figure 81. Flow Diagram of R:MOVE$DELETE and R:MOVE$KEEP (cont.)

3. Exit:

B MASTERPARSER in preparation for accepting intra-record commands to update the current record.

4. Operation:

This subroutine sets STEPFLAG = 1 for SS and SETFLAG = 1 for ST.

This CDT is examined to pick up record number and, if present, column number(s). FIRSTSET is set equal to the first sequence number. The column number parameters are processed via PROCESSCOL#PAIR.

An attempt is made to read the record via READRANDOM. At FINISH$STEP$LOOP the following tests are performed: If the record existed: SETEOD is used to insert carriage return if CR ON; record is typed via TYPECARD for ST, or only sequence number is typed via TYPESEQ for SS; and normal exit is made to MASTERPARSER.

If it did not exist: it types '-Pn:NO SUCH REC' via TYPEMSG. It sets STEPFLAG and SETFLAG to zero and exits to MASTEREXECUTIVE.

At STEP$LOOP it writes record via WRITERANDOM (unless command was NO); it exists if null command; otherwise it reads the next record via READSEQUEN. It exits if EOF was hit after typing '--EOF HIT' via TYPEMSG; if not hit, it saves new sequence number in FIRSTSET and proceeds with tests at FINISH$STEP$LOOP.

## R:TYPE$COMPRESSED, R:TYPE and R:TYPE$SUP$SEQ

1. Purpose:

Executes commands: TC (type records compressed), TY (type records), TS (type, suppressing sequence number).

Command forms are: {TY / TC / TS} n [-m][,c][,d] {TS / TY} for intra-record mode.

2. Entry:

R:LNK is the linkage register. This subroutine is entered via

| | |
|---|---|
| BAL, R:LNK | R:TYPE$COMPRESSED, |
| BAL, R:LNK | R:TYPE, |
| BAL, R:LNK | R:TYPE$SUP$SEQ in MASTEREXECUTIVE. |

3. Exit:

B *R:LNK

4. Operation:

This subroutine examines the CDT to find sequence numbers. It uses PROCESSCOL#PAIR to prepare for processing columns c-d and READNXTRANDOM for reading record n. It reads and types records in the range using:

SETEOD to insert carriage return if CR ON, TYPECARD to type record, and READSEQUEN to read next record. If EOF was hit, it types '--EOD HIT' and exits.

The technique for handling typing of column bounds and the compressing of blanks is at TYP40 as a subroutine within this routine. Using FRSTCLMN, LASTCLMN (set by PROCESSCOL#PAIR) it shifts the image in CARDIMG to compress blanks if TC.

I:DELETE

1. Purpose:

   Executes the intraline Delete string command D.

2. Entry:

   I:LNK is the linkage register. This subroutine is entered via BAL, I:LNK I:DELETE in MASTEREXECUTIVE.

3. Exit:

   B *I:LNK

4. Operation:

   This subroutine uses: FINDCOLUMN to find column corresponding to the first parameter, SHIFTLEFT to delete string, ADJUSTALLFLAG to set column 1 at which to resume matching, and SETOD to reset the EOD marker.

I:FOLLOW$BY

1. Purpose:

   Executes the intraline command F (follow X by Y).

2. Entry:

   I:LNK is the linkage register. This subroutine is entered via BAL, I:LNK I:FOLLOW$BY in MASTEREXECUTIVE.

3. Exit:

   B *I:LNK

4. Operation:

   This subroutine calls: FINDCOLUMN to find column corresponding to the first parameter, SHIFTRIGHT to make room for second string, MOVESTRING to move string into hole, ADJUSTALLFLAG to set ALLFLAG = column at which to resume matching (if ALLFLAG on), and SETEOD to reset EOD marker.

I:JUMP

1. Purpose:

   Executes the Jump command, JU (format JU n).

2. Entry:

   I:LNK is the linkage register. This subroutine is entered via BAL, I:LNK I:JUMP in MASTEREXECUTIVE.

3. Exit:

Normal exit is B FINISH$STEP$LOOP. Error exit is B *I:LNK.

4. Operation:

If user is not in step mode, this subroutine uses TYPECERR to type and exits to MASTEREXECUTIVE. It uses WRITERANDOM to write record. It examines CDT to get record number n and uses READRANDOM to read it. If record not found, it uses TYPECERR to type '-Cn:NO SUCH REC', uses READRANDOM to restore old record to the buffer, and exits to MASTEREXECUTIVE.

If record found, it saves sequence number in FIRSTSET and exits to FINISH$STEP$LOOP.

I:NO$CHANGE

1. Purpose:

Sets NOCHGFLG flag in response to the NO command while in step mode.

2. Entry:

I:LNK is the linkage register. This subroutine entered via BAL, I:LNK I:NO$CHANGE in MASTEREXECUTIVE.

3. Exit:

B *I:LNK

4. Operation:

If user is not in step mode this subroutine types '-Cn:CMND ILGL HERE' via TYPECERR and returns. Otherwise, it sets NOCHGFLG=1 and returns.

I:OVERWRITE

1. Purpose:

Executes the intraline command O, overwrite X by Y.

2. Entry:

I:LNK is the linkage register. This subroutine is entered via BAL, I:LNK I:OVERWRITE in MASTEREXECUTIVE.

3. Exit:

B *I:LNK

4. Operation:

This subroutine uses: FINDCOLUMN to find column corresponding to the first parameter, MOVESTRING to overwrite string X with string Y, ADJUSTALLFLAG to set column number at which to resume matching, and SETEOD to set the EOD marker.

I:OVERWR$EXTEND

1. Purpose:

   Executes the overwrite and extend blanks command, E. This command has two forms:

   [j]/$string_1$/E/$string_2$/ or kE/$string_2$/.

2. Entry:

   I:LNK is the linkage register. This subroutine is entered via BAL, I:LNK I:OVERWR$EXTEND in MASTEREXECUTIVE.

3. Exit:

   B *I:LNK

4. Operation:

   This subroutine sets ALLOK $\neq$ 0 to show "ALL" is not valid at this point; uses FINDCOLUMN to find column which corresponds to the first parameter; examines CDT to find address of $string_2$; uses MOVESTRING to place new string in record; updates P1 to point to the column after the last new character; stores blanks in remainder of record; uses SETEOD to insert carriage return if CR ON; and exits to MASTEREXECUTIVE.

I:PRECEDE$BY

1. Purpose:

   Executes the intraline string command P, precede X by Y.

2. Entry:

   I:LNK is the linkage register. This subroutine is entered via BAL, I:LNK I:PRECEDE$BY in MASTEREXECUTIVE.

3. Exit:

   B *I:LNK

4. Operation:

   This subroutine uses: FINDCOLUMN to find column corresponding to the first parameters, SHIFTRIGHT to make room for second string, MOVESTRING to move string Y into hold, ADJUSTALLFLAG to set column number at which to resume matching, and SETEOD to reset the EOD marker.

I:REVERSE$BPFLAG

1. Purpose:

   Execute the RF command.

2. Entry:

   I:LNK is the linkage register. This subroutine is entered via BAL, I:LNK I:REVERSE$BPFLAG in MASTEREXECUTIVE.

3. Exit:

B *I:LNK

4. Operation:

This subroutine reverses BPFLAG and returns.

I:SET

1. Purpose:

Executes the intraline command SE.

2. Entry:

I:LNK is the linkage register. This subroutine is entered via BAL, I:LNK I:SET in MASTEREXECUTIVE. Its auxiliary entry is SET$LOOP from MASTEREXECUTIVE if in SET loop.

3. Exit:

B *I:LNK is normal exit. If entered via SET$LOOP, normal exit is B RESTART$EXECUTIVE. Error exit is B MASTERPARSER.

4. Operation:

It sets SETFLAG = 1. The CDT is examined to pick up first and last sequence numbers. It stores first sequence number in FIRSTSET, and last one in LASTSET. If range covers more than one record it sets ERRORCNT = 1.

PROCESSCOL#PAIR is used to process the column numbers. SETADR is initialized to address in CDT which follows the SE command. An attempt is made to read the record via READRANDOM: if not found, '-Pn:NO SUCH REC' is typed via TYPEMSG, SETFLAG is set to zero, and exit is made to MASTERPARSER; if found, SETEOD is used to insert carriage return if CR ON, and exit is to MASTEREXECUTIVE via B *I:LNK.

If entered at SET$LOOP and the last record has been processed, it uses WRITERANDOM to write it, sets SETFLAG = 1 so that loop will be restarted if another I:CMND follows and exits to MASTERPARSER. If it has not, it uses WRITERANDOM to write current record and READSEQUEN to read next one: if sequence is past range, it sets SETFLAG = 1 and exits; if not past range it puts new sequence number in FIRSTSET, puts contents of SETADR in CDTADR to start I:CMND loop at beginning, uses SETEOD to insert carriage return if needed, and exits to RESTART$EXECUTIVE.

If EOF encountered it uses TYPEMSG to print '--EOF HIT', sets SETFLAG = 1, and exits to MASTERPARSER.

I:SHIFT$LEFT

1. Purpose:

Executes the intraline command L, shift X left by N.

2. Entry:

I:LNK is the linkage register. It is entered via BAL, I:LNK I:SHIFT$LEFT in MASTEREXECUTIVE.

3. Exit:

B *I:LNK

4. Operation:

This subroutine uses: FINDCOLUMN to find column corresponding to the first parameter, SHIFTLEFT to shift string left N places, and SETEOD to reset the EOD marker.

I:SHIFT$RIGHT

1. Purpose:

Executes the intraline string command R, shift X right by N.

2. Entry:

I:LNK is the linkage register. This subroutine is entered via BAL, I:LNK I:SHIFT$RIGHT from MASTEREXECUTIVE.

3. Exit:

B *I:LNK which results in a return to MASTEREXECUTIVE.

4. Operation:

This subroutine uses: FINDCOLUMN to find the column corresponding to the first parameter, SHIFTRIGHT to shift string X right N spaces, and SETEOD to set the EOD markers.

I:SUBSTITUTE

1. Purpose:

Executes the substitute command, S whose format is [j]/$string_1$/S/$string_2$/.

2. Entry:

I:LNK is the linkage register. This subroutine is entered via BAL, I:LNK I:SUBSTITUTE in MASTEREXECUTIVE.

3. Exit:

B *I:LNK

4. Operation:

This subroutine performs the following functions:

a. Uses FINDCOLUMN to find column corresponding to the first parameter.
b. Exits if it is not found.
c. Examines CDT to get address of $string_2$.

d. Sets P1 = character following $string_1$.

e. Uses SHIFTRIGHT to shift rightmost characters to the right if new string is shorter than old.

f. Uses MOVESTRING to put the new string in place.

g. Uses ADJUSTALLFLAG to reset ALLFLAG to the column number at which matching is to be resumed (if ALLFLAG $\neq$ 0).

h. Uses SETEOD to insert carriage return if CR ON.

i. Exits to MASTEREXECUTIVE.

I:TYPE

1. Purpose:

   Type records in response to intraline commands.

2. Entry:

   I:LNK is the linkage register. This subroutine is entered via BAL from MASTEREXECUTIVE; in response to TY command, via BAL, I:LNK I:TYPE; or in response to TS command, via BAL, I:LNK I:TYPE$SUP$SEQ.

3. Exit:

   B *I:LNK

4. Operation:

   For I:TYPE FIRSTSET is set equal to sequence number. TYPECARD is used to type card image with sequence number. For I:TYPE$SUP$SEQ it sets P1 = 1 to indicate sequence number suppression. It uses TYPECARD to type card image without sequence number.

## General Purpose Subroutines

ADDCDTPARAM

1. Purpose:

   Adds a new parameter to the Command Description Table (see Figure 71).

2. Entry:

   LNK is linkage register used. This subroutine is entered via BAL, LNK ADDCDTPARAM from several PARSE:routines when it is desired to add to the CDT. Upon entry P1, PARAMPSN, and PRMBUFSZ are as shown below.

3. Exit:

   B 0, LNK with expanded CDT entry.

4. Operation:

Words are added to the CDT from PARAMBUF according to the format shown previously in Figure 71 using the following input parameters:

P1 = type of parameter.

PARAMPSN = next available slot in CDT.

PRMBUFSZ= number of words to be added.

ADJINT

1. Purpose:

Forms a sequence number as an integer * 1000.

2. Entry:

LNK is the linkage register. This subroutine is entered via BAL, LNK ADJINT.

3. Exit:

B *LNK

4. Operation:

This routine multiplies the sequence number in PARAMBUF by 1000 and stores it back in PARAMBUF.

ADJUSTALLFLAG

1. Purpose:

Sets ALLFLAG.

2. Entry:

LNK is the linkage register. This subroutine is used by a number of the I:routines in processing intraline commands. Upon entry P1 = column number at which to resume matching.

3. Exit:

B 0, LNK to calling routine.

4. Operation:

No action is taken if ALLFLAG < 0. Otherwise this subroutine sets ALLFLAG = P1.

ANLZRIGHT

1. Purpose:

Analyzes the composition of a field to the right.

2. Entry:

LNK is the linkage register. This routine is entered via BAL, LNK ANLZRIGHT from either the SHIFTLEFT or SHIFTRIGHT routine. Upon entry, P1 = column at which to begin analyzing.

3. Exit:

If BO OFF: R1 = number of nonblanks to first blank. R2 = number of blanks – 1 from first blank to next nonblank. If BP ON: R1 = number of characters to last nonblank on card. R2 = number of trailing blanks on card. CC1 = 1 if initial P1 > end of buffer. CC1 = 0 otherwise. ANLZRIGHT returns to calling routine via B 0, LNK.

4. Operation:

If start of field falls past end of buffer, it sets R1 and R2 = 0, clears stack of P1, P2, sets CC1 and exits. EODCLMN is column number containing last nonblank character.

BPFLAG = 1 if BP ON.

BPFLAG = 0 if BP OFF.

MAXCLMN points to the end of buffer.

This subroutine performs character scan based on blank preservation mode to set R1 and R2 as shown above. It sets CC1 and returns to calling routine.

BINTODEC

1. Purpose:

Converts a binary number to decimal.

2. Entry:

LNK is the linkage register. P1 contains binary number. P2 contains byte address where decimal string is to be stored (right-most byte). This subroutine is used by MOVESEQ and TYPESEQ.

3. Exit:

Return is to calling routine via B 0, LNK with decimal quantity properly stored.

4. Operation:

BINTODEC divides binary number by 10, adds zone bits to remainder (i.e., X'F0'), stores it, moves pointer one to left in output string and repeats until seven digits have been converted.

BLANKBUF

1. Purpose:

Stores blanks in CARDIMG.

2. Entry:

LNK is the linkage register. This subroutine is used by READRANDOM and READSEQUEN. It is called using BAL, LNK BLANKBUF.

3. Exit:

Upon exit blanks are stored in CARDIMG. Return is made to calling routine via B 0, LNK.

CHECK 1CDTENTRY

1. Purpose:

Ensures there is only one entry in the CDT.

2. Entry:

LNK is the linkage register. This subroutine is entered via BAL, LNK CHECK1CDTENTRY.

3. Exit:

Normal return is to calling routine (only one entry was present). Error exit is to MASTERPARSER via TYPECERR after printing: '-Cn:CMND ILGL HERE'.

4. Operation:

This subroutine checks entry count in word 1 of CDT; it should be 1.

DELETE

1. Purpose:

Deletes records in a specified range.

2. Entry:

LNK is the linkage register. This routine is used by R:DELETE, and R:MOVE$DELETE. Calling sequence is BAL, LNK DELETE and assumes:

P1 contains sequence number of first record to delete.

P2 contains sequence number of last record to delete.

R1 contains sequence number of last record read.

R2 contains number of records deleted.

3. Exit:

CC1 = 1 if last sequence number was passed. CC1 = 0 otherwise. Return to calling routine using B 0, LNK. Message upon hitting end-of-file is '--EOF HIT'.

4. Operation:

Reads and deletes record i.

It continues until P2 is reached or passed, or EOF hit.

BADIO1

1. Purpose:

   Prints abnormal I/O message.

2. Entry:

   At BADIO, an error code is loaded into X1 from D1. At BADIO1, the error code is assumed to be in X1. Entry is with branch because there is no return to calling routine. BADIO1 is called by various open, read and write routines.

3. Exit:

   It returns to the monitor via the EXIT CAL.

4. Operation:

   It sets up error code in message line and prints message.

FINDCOLUMN

1. Purpose:

   Evaluates the first parameters for intraline commands.

2. Entry:

   LNK is the linkage register. This subroutine is entered via BAL,LNK FINDCOLUMN from one of the following routines:

   I:DELETE

   I:OVERWR$EXTEND

   I:FOLLOW$BY

   I:SHIFT$LEFT

   I:OVERWRITE

   I:PRECEDE$BY

   I:SHIFT$RIGHT

   I:SUBSTITUTE

   Upon entry CDTADR contains address of current command in CDT.

3. Exit:

   Upon exit: P1 = column computed from parameters.

   P2 = width of field at this column.

   X1 =position of next CDT control byte.

CC1 = 1 if no column found.

CC1 = 0 otherwise.

Exit is B 0, LNK

4. Operation:

If not in "ALL" mode: and < 3 parameters: if command form K cmnd , it ensures that FRSTCLMN < K < LASTCLMN. If/string/ cmnd , it uses FINDMATCH to search for string. and = 3 parameters: it sets occurrence count = 1 if it is illegal at this point, and it uses FINDMATCH to search for string. Error messages (printed via TYPECERR) are:

'--Cn:'QLL'IGNORED',

'--Cn:NO SUCH STRING',

'--Cn:COL>LIMIT',

'--Cn:COL<LIMIT'.

## FINDMATCH

1. Purpose:

Finds matching string in record.

2. Entry:

LNK is the linkage register. This subroutine is entered via BAL, LNK FINDMATCH from one of the following routines:

R:FIND$SEQUENCE,

R:FIND$DELETE,

R:FIND$TYPE,

FINDCOLUMN.

Upon entry: P1 = column number.

P2 = address of TEXTC string to be matched.

3. Exit:

Upon exit: R1 = column number at which match occurred.

CC1 = 0 if match found.

CC1 = 1 if match not found.

Exit is via B O, LNK.

4. Operation:

This subroutine:

sets TEXTCADR = C(P2), and

sets STOPCLMN (last column number at which a match can take place) = C(LASTCLMN) - string length.

If P1 < STOPCLMN, it exits with CC1 = 1. It scans record for match with TEXTC string.

GETFILEID

1. Purpose:

Checks syntax of FID and if good, sets PARAMBUF and PRMBUFSZ.

2. Entry:

LNK is the linkage register. This subroutine is entered via BAL, LNK GETFILEID from PARSE:SAVE and :EDIT to obtain the file ID so it can be placed in the Command Description Table.

The file ID is assumed to be in the Teletype input buffer.

3. Exit:

File ID is now in PARAMBUF as follows:

PARAMBUF

TEXTC filename

TEXTC area name

and it has been checked for proper format.

Return is to calling routine via BAL 0, LNK.

Error returns are via GETNEXTNAME: messages include

'-P1:BAD FID'

'-Cn:CMND ILGL HERE'

'-P1:ILGL SYNTAX'.

4. Operation:

This routine rejects a file name which is longer than eight characters. It uses GETNEXTNAME first to build file name in PARAMBUF. It then pushes the area name, if present. It then pulls the entries from the stack and stores them in PARAMBUF and stores in PRMBUFSZ the length in words of the entries in PARAMBUF.

GETNEXTNAME

1. Purpose:

Gets the next name from the Teletype input buffer (TTYIMG) and places it in PARAMBUF.

2. Entry:

LNK is the linkage register. This subroutine is entered by invoking the NXTNAM command procedure resulting in the calling sequence:

| | |
|---|---|
| BAL, LNK | GETNEXTNAME. |
| GEN, 8, 24 | # of branches, address of error message, |
| GEN, 8, 24 | type 1, branch address 1. |
| ⋮ | ⋮ |
| GEN, 8, 24 | type n , branch address n. |

3. Exit:

Upon exit, a name (file or area) resides in PARAMBUF in TEXTC format.

Return is to the branch address in bits 8-31 of the word in the calling sequence of which a match was found on bits 0-7. Error return is to MASTERPARSER.

4. Operation:

Input characters from TTYIMG are scanned and tested to determine whether they are part of a name. They are placed in PARAMBUF in TEXTC format. When a terminator is found (e.g., comma or right parenthesis) the scan is stopped, and the name is padded to the right with three blanks. If an error is found, the error message given in the calling sequence is printed and return is made to MASTER-PARSER via TYPECERR.

The flow of GETNEXTNAME is given in Figure 82.

GETNEXTPARAM

1. Purpose:

Scans the Teletype input buffer to isolate recognizable character strings which comprise EDIT commands and places them in PARAMBUF.

2. Entry:

This routine is called by various PARSE: routines. LNK is the linkage register. The routine is invoked by the NXTPRM procedure which sets up a calling sequence as follows:

| | |
|---|---|
| BAL, LNK | GETNEXTPARAM. |
| GEN, 8, 24 | # of branches, address of error message. |
| GEN, 8, 24 | completion type, branch address 1. |
| ⋮ | ⋮ |
| GEN, 8, 24 | completion type, branch address n. |

3. Exit:

Upon exit, a parameter is in PARAMBUF in TEXTC format. Return is to the branch address in bits 8-31 of the word in the calling sequence of which a match was found on bits 0-7. Error return is to MASTERPARSER via TYPEPERR.

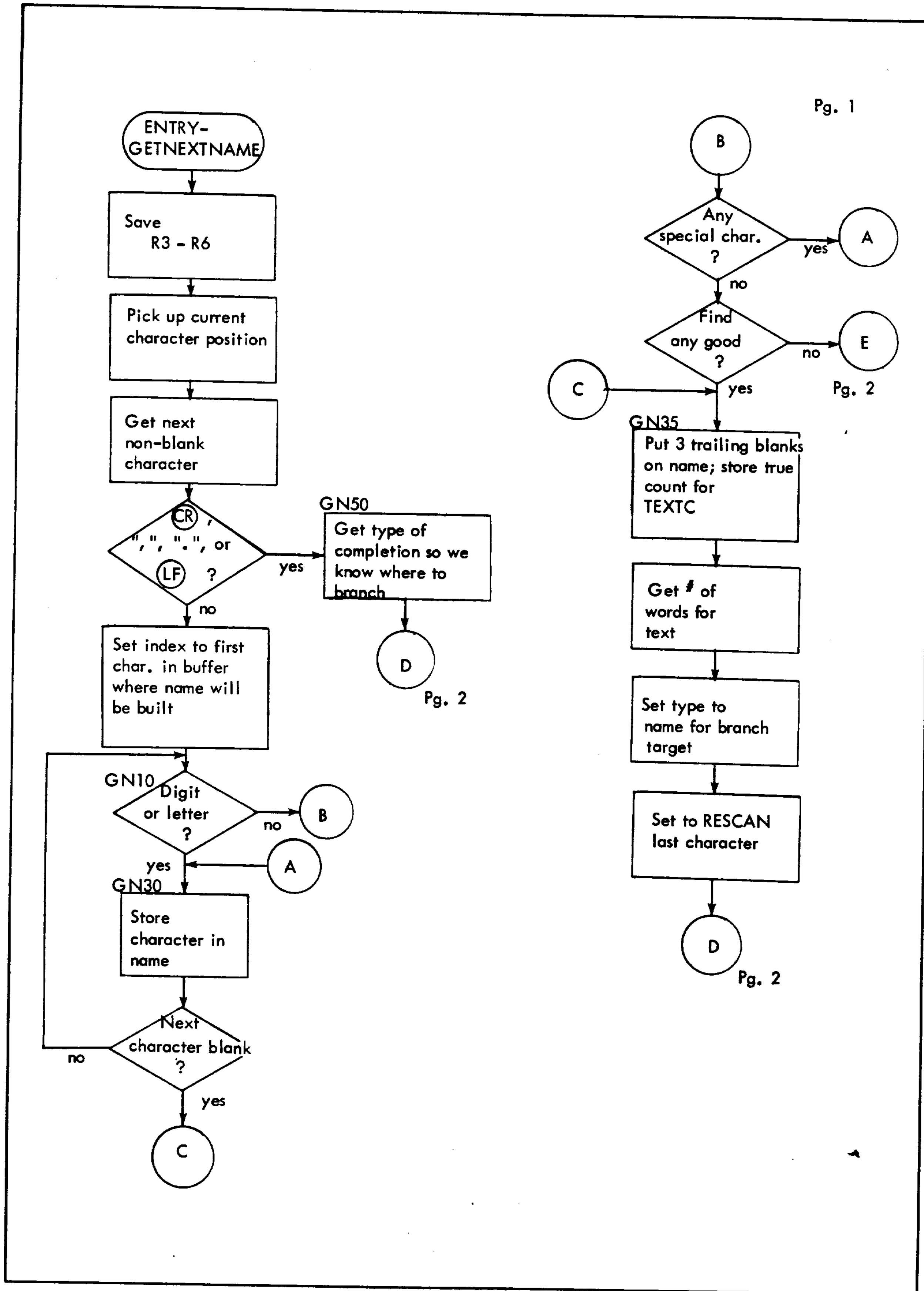



Figure 82. Flow Diagram of GETNEXTNAME

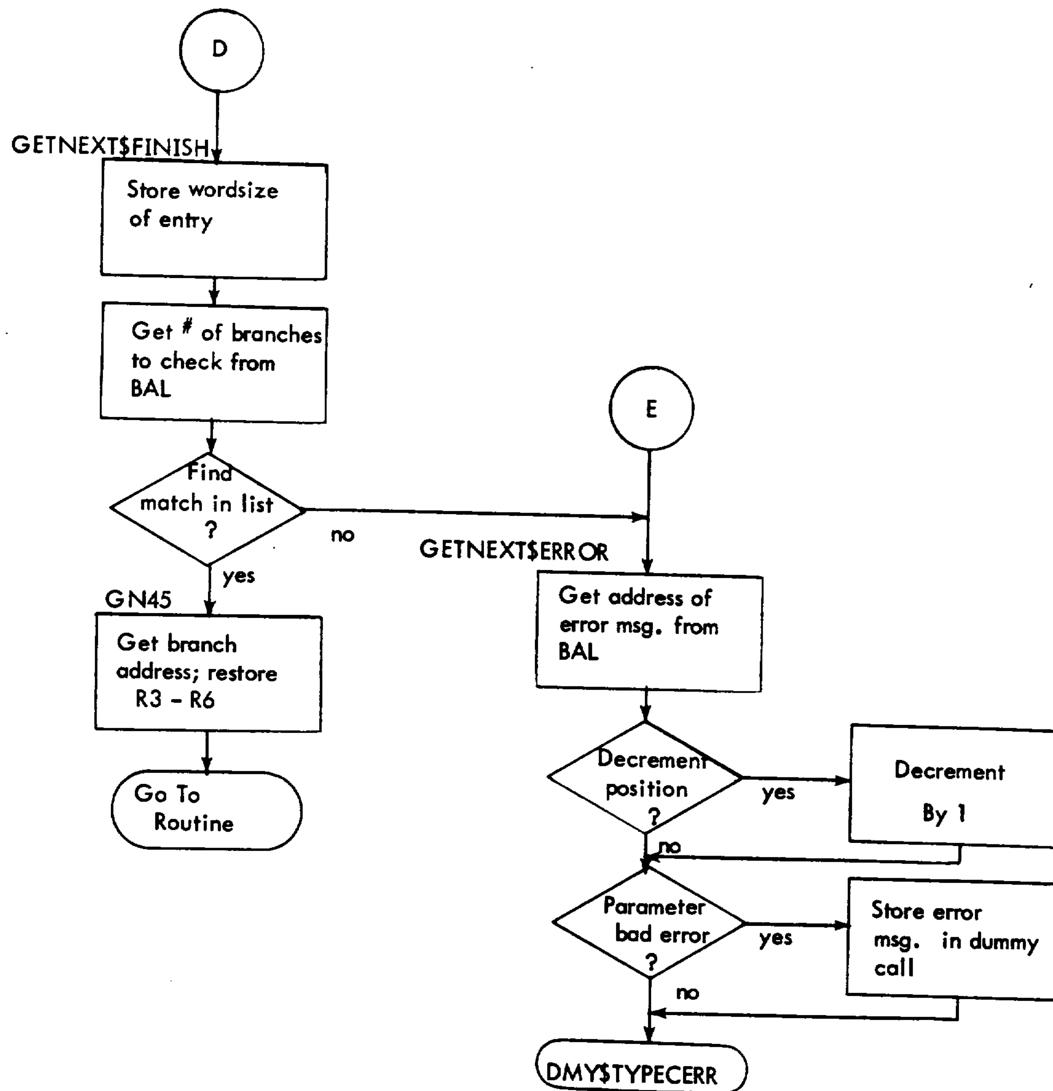



Figure 82. Flow Diagram of GETNEXTNAME (cont.)

4. Operation:

   A scan is made character-by-character, with tests being performed to detect invalid command format. Examples: slash (/) must not be the last character of a command; a sequence number must not exceed three digits; the second sequence number in a range must not be greater than the first. Error messages include:

   '-Pn:ILGL STRG'

   '-Pn:NULL STRG'

   '-Pn:ILGL SEQ#'

   '-Pn:SEQ2<SEQ1'

   in addition to that given in the calling sequence.

   The flow of GETNEXTPARAM is given in Figure 83.

ILGL$SEMICOLON

1. Purpose:

   Output an error message when semi-colon is found following F: or R: CMND (should only be used in intrarecord operations).

2. Entry:

   The address of this routine is entered via the NXTPRM procedure into the calling sequence of a branch to routine GETNEXTPARAM when it is desired that the SCOL (semicolon) type of entry be flagged as an error. When entered it is done via B *D1 at GN45 of GETNEXT$FINISH.

3. Exit:

   Branches to MASTERPARSER.

4. Operation:

   It increments byte 3 of CDT containing CMND #. It uses TYPECERR to type '-Cn:CMND ILGL HERE'.

MOVESEQ

1. Purpose:

   Formats sequence number in EBCDIC as 'XXXX.XXX' having four characters from calling sequences appended.

2. Entry:

   LNK is the linkage register. This subroutine is entered via BAL, LNK MOVESEQ from one of the following routines:

   R:MOVE$DELETE

   R:MOVE$KEEP

   READSEQUEN

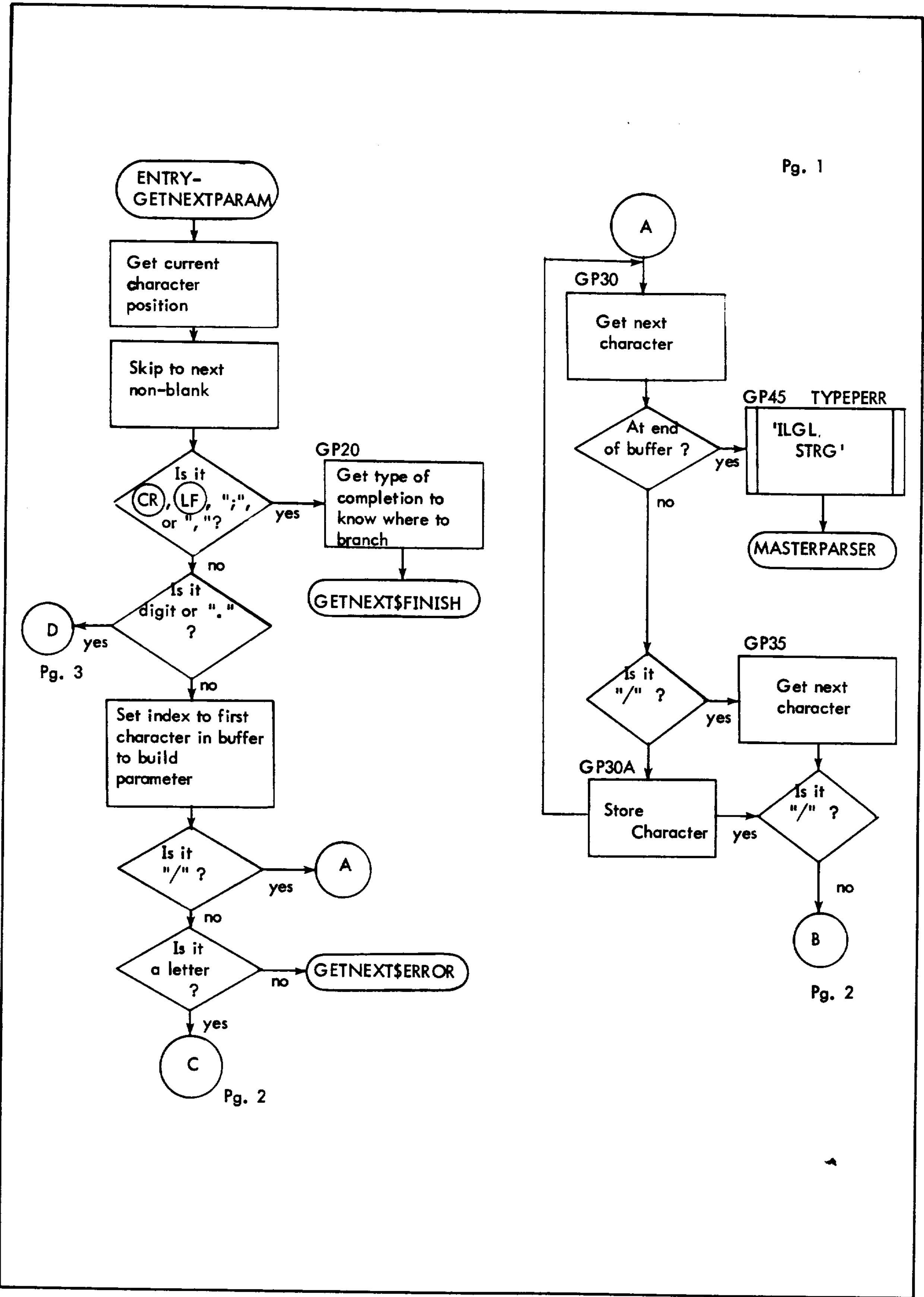



Figure 83. Flow Diagram of GETNEXTPARAM

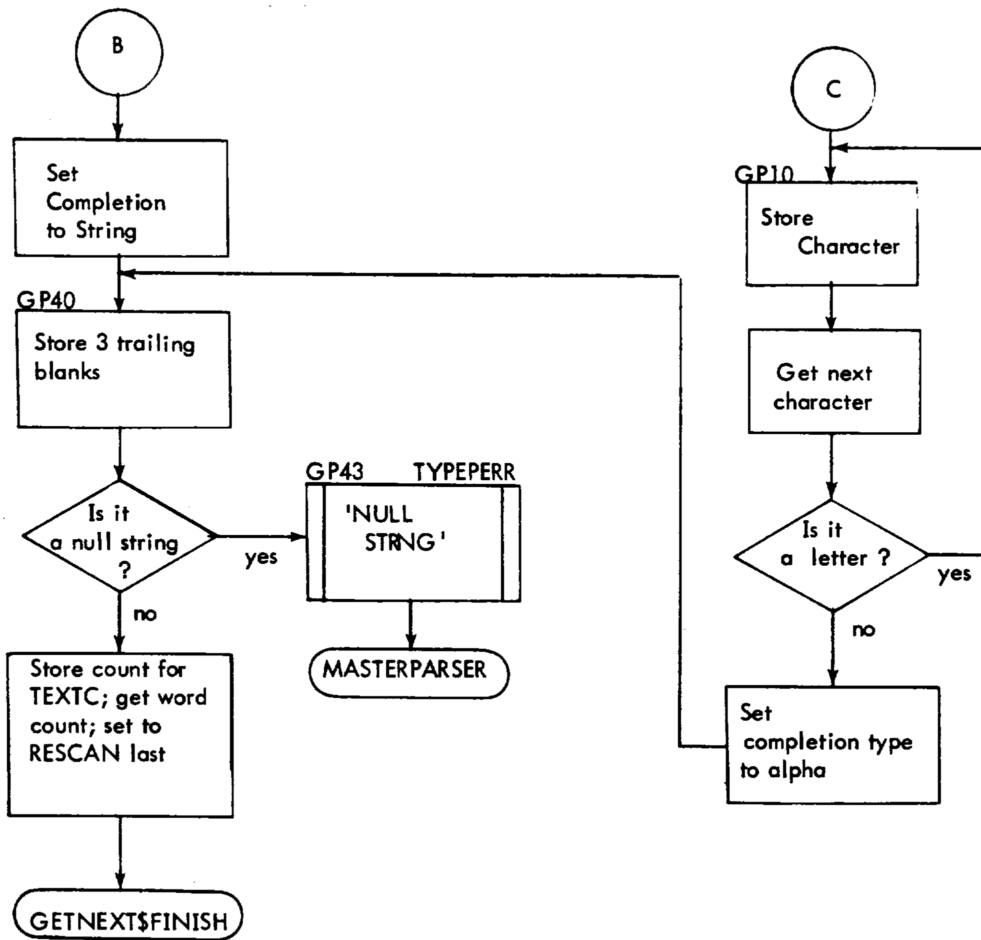



Figure 83. Flow Diagram of GETNEXTPARAM (cont.)

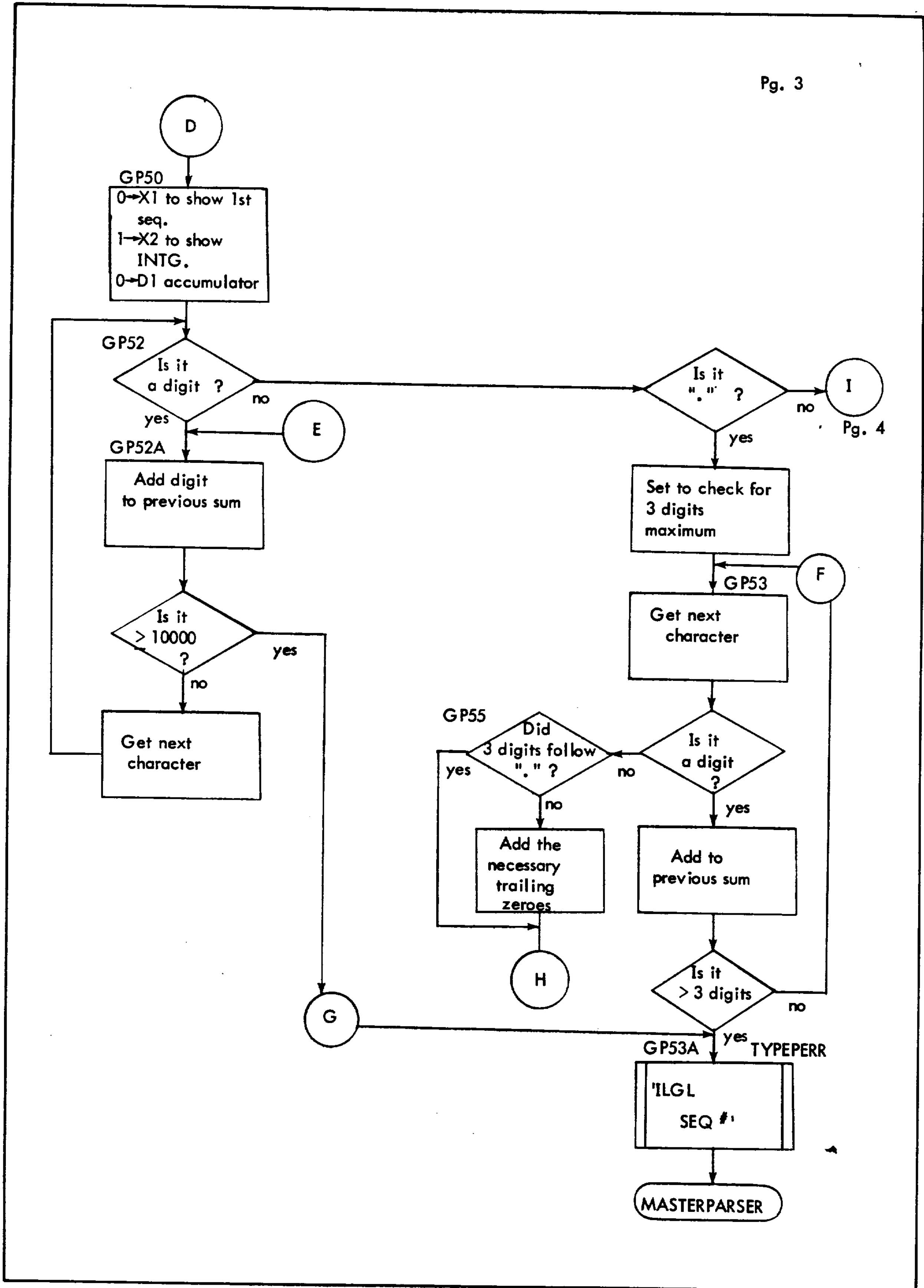



Figure 83. Flow Diagram of GETNEXTPARAM (cont.)

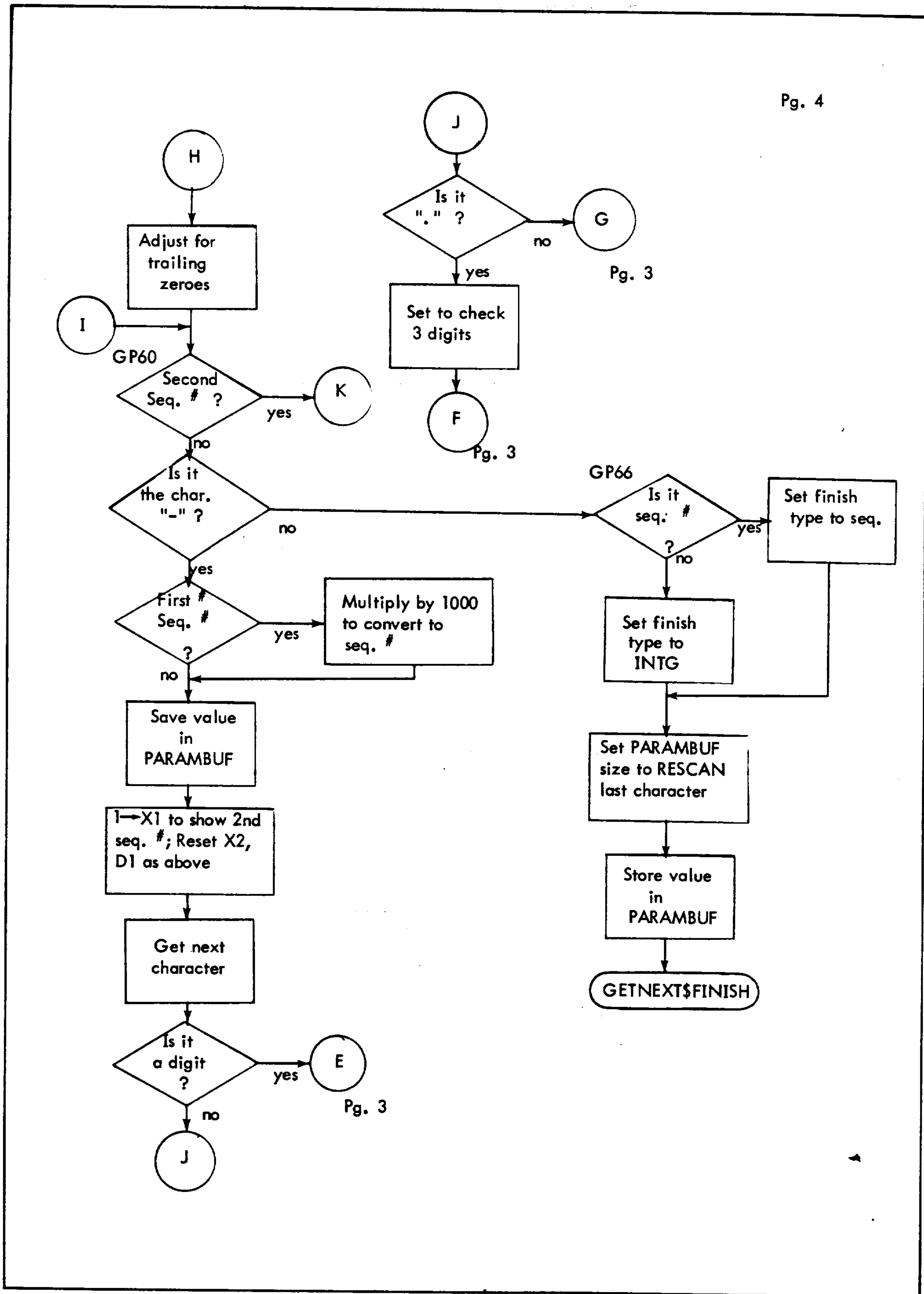



Figure 83. Flow Diagram of GETNEXTPARAM (cont.)

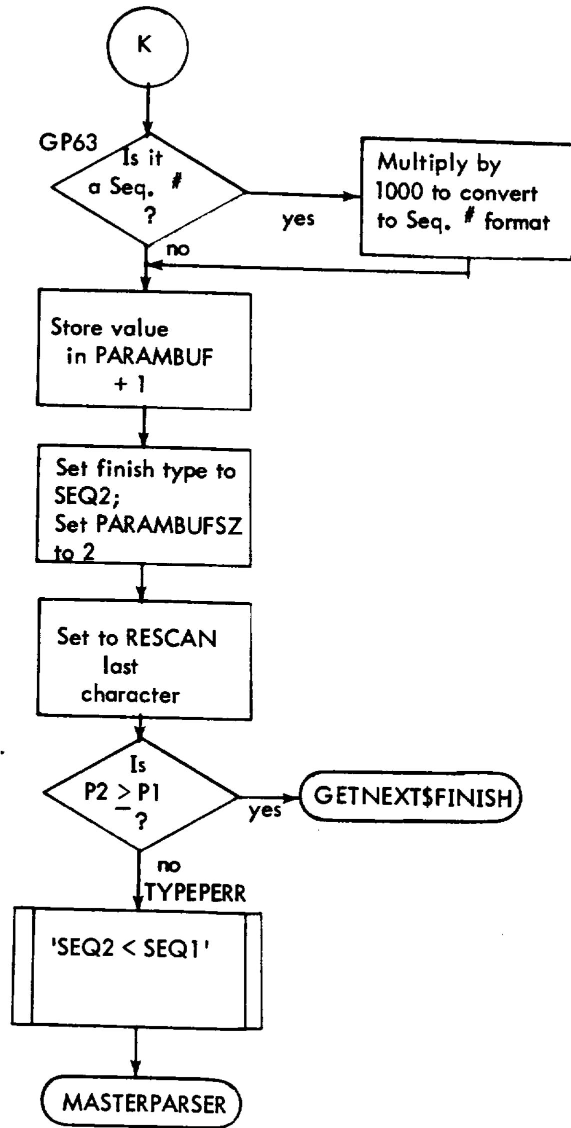



Figure 83. Flow Diagram of GETNEXTPARAM (cont.)

Upon entry: P1 = sequence number to be converted to EBCDIC; P2 = byte address at which to put the string. Word following the BAL contains four characters to be appended to the sequence number.

3. Exit:

Upon exit, R1 contains the number of characters in the resultant string. Exit is B 1, LNK.

4. Operation:

It uses BINTODEC to convert sequence number to EBCDIC, places string in TEMPBLCK with leading zeros blanked and the requested characters appended at the right.

MOVESTRING

1. Purpose:

Moves a character string into the output buffer.

2. Entry:

LNK is the linkage register. This subroutine is entered via BAL, LNK MOVESTRING from several I: routines. Upon entry P1 = column at which string is to be placed; P2 = address of TEXTC string which is to be inserted.

3. Exit:

B 0, LNK

4. Operation:

If starting column is beyond end of record and any character of string is nonblank, it types '-Cn:OVERFLOW' via TYPECERR and exits; otherwise, it exits. If not beyond end of record it moves TEXTC string into buffer, one character at a time. If end of buffer is reached, it types message as above and exits.

NEWCDTENTRY

1. Purpose:

Sets up room in the CDT for a new entry.

2. Entry:

LNK is the linkage register. This subroutine is entered from several PARSE: routines. Upon entry: P1 contains the number of the command type to be added; word following the BAL contains the number of parameters.

3. Exit:

Return is made to $\alpha + 2$ with CDT entry initialized.

4. **Operation:**

CDT entry is initialized as follows:

word 0: byte 0 contains length of entry (= 0 initially); byte 1 contains command type (or number) e.g., 0 for carriage return, 1 for file name, etc. Byte 2 contains number of this entry in the CDT; byte 3 contains number of parameters. Words (1 ⟶ # of parameters/2) : zeroes. Word # of parameters/2 + 1 : X'00000100'.

PROCESSCOL#PAIR

1. Purpose:

Performs internal housekeeping required in processing a pair of column numbers in a record or an intra-record command.

2. Entry:

LNK is the linkage register. This subroutine is entered via BAL, LNK PROCESSCOL#PAIR from the following routines:

R:FIND$SEQUENCE

R:FIND$DELETE

R:FIND$TYPE

R:SET$STEP

R:SET$STEP$TYPE

R:TYPE$COMPRESSED

R:TYPE

R:TYPE$SUP$SEQ

I:SET

Upon entry, X1 points to the location of the next parameter control byte in the CDT.

3. Exit:

Normal exit is B 0, LNK. Error exit is B MASTERPARSER after printing '-BAD COL. NO. PAIR' via TYPEMSG and setting SETFLAG and STEPFLAG = 0.

4. Operation:

It sets starting and stopping columns as follows:

FRSTCLMN = 0 if no starting column given.
= column number c - 1 from command if input.

LASTCLMN = 140 if no stopping column given.
= column number d + 1 from command if input.

If c ≥ d it prints error message and exits.

If d ≥ 140 it prints error message and exits.

READNXTRANDOM

1. Purpose:

   Reads random record or next highest one.

2. Entry:

   LNK is the linkage register. This subroutine is entered via BAL, LNK READNXTRANDOM from the following routines:

   R:FIND$SEQUENCE

   R:FIND$DELETE

   R:FIND$TYPE

   R:INSERT$SUP$SEQ

   R:INSERT

   R:MOVE$DELETE

   R:MOVE$KEEP

   R:TYPE$SUP$SEQ

   R:TYPE$COMPRESSED

   DELETE

   Upon entry, P1 = sequence number of record to be read.

3. Exit:

   R1 = sequence number of record actually read.

   CC1 = 0 if record existed.

   CC1 = 1 otherwise.

   Return is to calling routine via B 0, LNK.

4. Operation:

   This subroutine uses READRANDOM to issue read. If read was successful it sets R1 = sequence number and CC1 = 0. Otherwise it sets CC1 = 1 and returns.

READTELETYPE, READTELETYPE2

1. Purpose:

   Reads an input line.

2. Entry:

LNK is the linkage register. This subroutine is entered via BAL, LNK TELETYPE from one of the following:

R:INSERT

R:INSERT$SUP$SEQ

It is entered via BAL, LNK TELETYPE2 from MASTERPARSER or R:COMMENTARY and stores characters into TTYIMG.

3. Exit:

R1 = number of characters read. It returns via B 0, LNK to calling routine.

4. Operation:

It sets buffer to CARDIMG or TTYIMG, stores blanks in buffer, reads up to the length of the buffer and sets R1. If the value in GOSEQ is negative (EDIT is not in GO mode), the input is taken from M:SI, the control stream. If the value in GOSEQ is non-negative (EDIT is in GO mode), the input is taken from the edit file at the first sequence number greater than the value in GOSEQ.

SETEOD

1. Purpose:

Scans active card image to locate the rightmost nonblank character.

2. Entry:

LNK is the linkage register. This subroutine is used by a number of routines such as MASTEREXECUTIVE and several R: and I: routines. It is called using BAL, LNK SETEOD.

3. Exit:

EODCLMN contains column of last nonblank character or -1 if all blanks. RECSIZE contains a byte count of zero if all blanks. Return is to calling routine using B 0, LNK.

4. Operation:

This subroutine scans record image from right looking for all blank words (up to word 0). If no nonblanks are found, it sets flag to check word zero, byte-by-byte. Otherwise, it sets flag to indicate byte-by-byte checking in the word where a nonblank character was found.

SETLASTKEY

1. Purpose:

Stores key of last record read read in LASTKEY and stores record size in RECSIZE.

2. Entry:

LNK is the linkage register. This subroutine is entered via BAL, LNK SETLASTKEY from READRANDOM and READSEQUEN.

3. Exit:

It returns to calling routine via B 0, LNK with LASTKEY and RECSIZE set.

4. Operation:

It sets LASTKEY and RECSIZE, removes carriage return, if any, and uses SETEOD to append carriage return if CR ON.

SHIFTLEFT

1. Purpose:

Shifts a character string to the left.

2. Entry:

LNK is the linkage register. This subroutine is entered via BAL, LNK SHIFTLEFT from one of the following routines:

I:DELETE

I:SHIFT$LEFT

I:SUBSTITUTE

Upon entry: P1 = column number at which to start the shift.
P2 = width of field starting at this column.
P3 = number of spaces to shift left.

3. Exit:

B 0, LNK

4. Operation:

This subroutine uses ANLZRIGHT to analyze field at P1. If field extends beyond end of card, it prepares to shift in blanks. If shift will push data off beginning of record, it prints '--Cn:UNDERFLOW' via TYPECERR and prepares to shift only to column 0. If width of 0, it bypasses shift. It performs shift, blanking out the cleared characters on the right.

SHIFTRIGHT

1. Purpose:

Shifts a character string to the right.

2. Entry:

LNK is the linkage register. This subroutine is entered via BAL, LNK SHIFTRIGHT from the following routines:

I:FOLLOW$BY

I:PRECEDE$BY

I:SHIFT$RIGHT

I:SUBSTITUTE

Upon entry: P1 = column at which to start shift.

P2 = width of field starting at this column.

P3 = number of places to shift.

3. Exit:

B 0, LNK

4. Operation:

It exits if P1 > MAXCLMN.

It initializes FIELDCNT = number of fields to compress and BLANKCNT = number of blanks to compress. If $field_1$ extends to end of record it types pointers prior to shifting. If field extends beyond end of record, in addition to message, it blanks out P2 characters and exits.

For normal shifts:

It uses ANLZRIGHT to compute R1 = column at end of nonblanks and R2 = number of nonblanks -1 = number to shift for each space to be filled (just rightmost portion of record if BP ON).

The R1 and R2 quantities computed by each entry to ANLZRIGHT are pushed into a stack to be pulled from to perform the shifting or compressing. When all fields have been compressed, blank-fill is performed on the original fields at left in record.

Special cases:

If $field_2$-n is found to spill off end of card: message is output via TYPECERR: '--Cn:OVERFLOW'; and BLANKCNT gets set to the appropriate number of nonblanks to destroy.

It performs special process of compressing fields that do fit until reaching the one which does not, where it then computes a special R1 and R2. This covers cases in which part of a nonblank field gets destroyed.

The flow of SHIFTRIGHT is given in Figure 84.

TYPECARD

1. Purpose:

Types a card image.

2. Entry:

LNK is the linkage register. This subroutine is entered via BAL, LNK TYPECARD from one of the following routines:

R:FIND$SEQUENCE

R:FIND$DELETE

R:FIND$TYPE

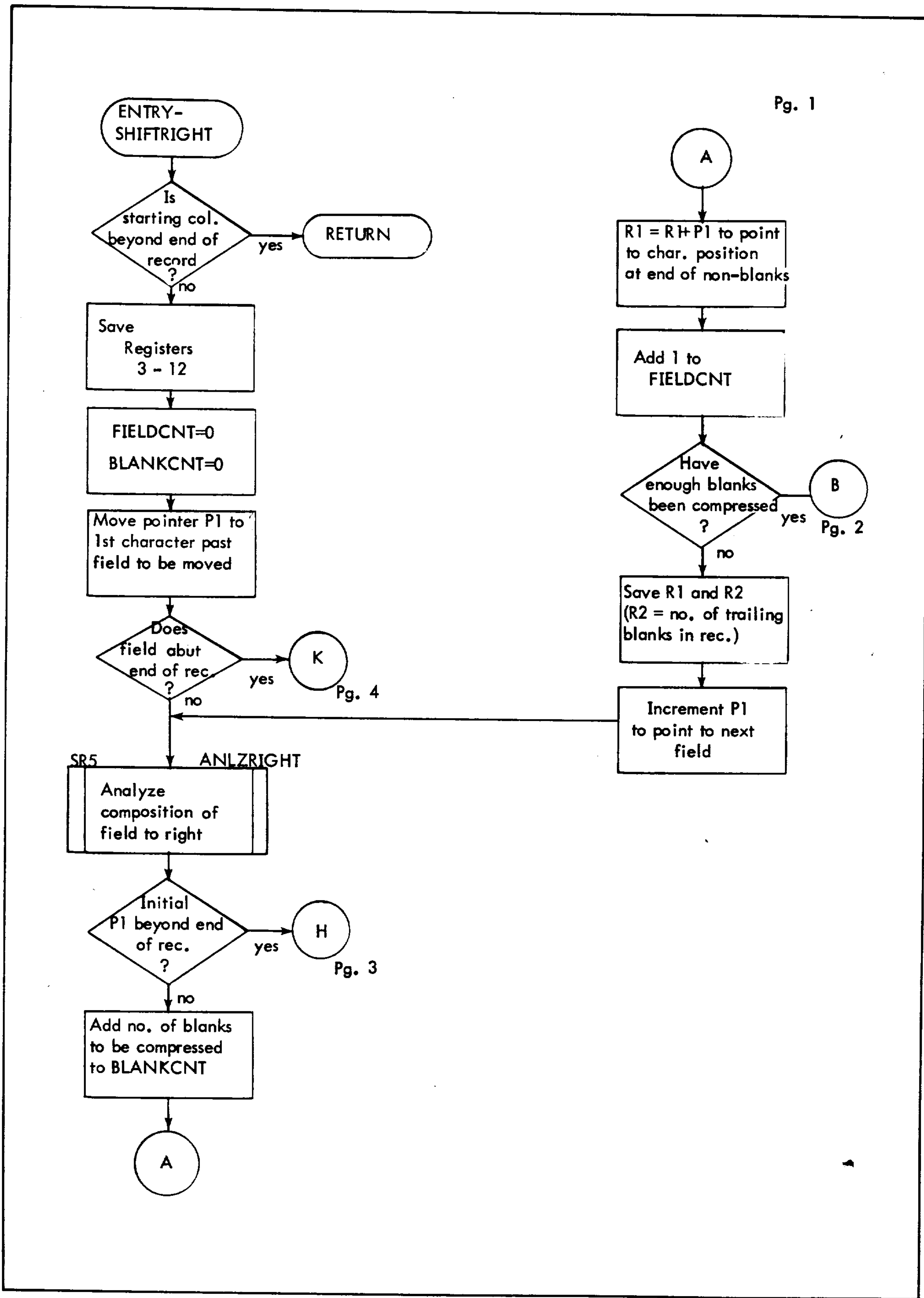



Figure 84. Flow Diagram of SHIFTRIGHT

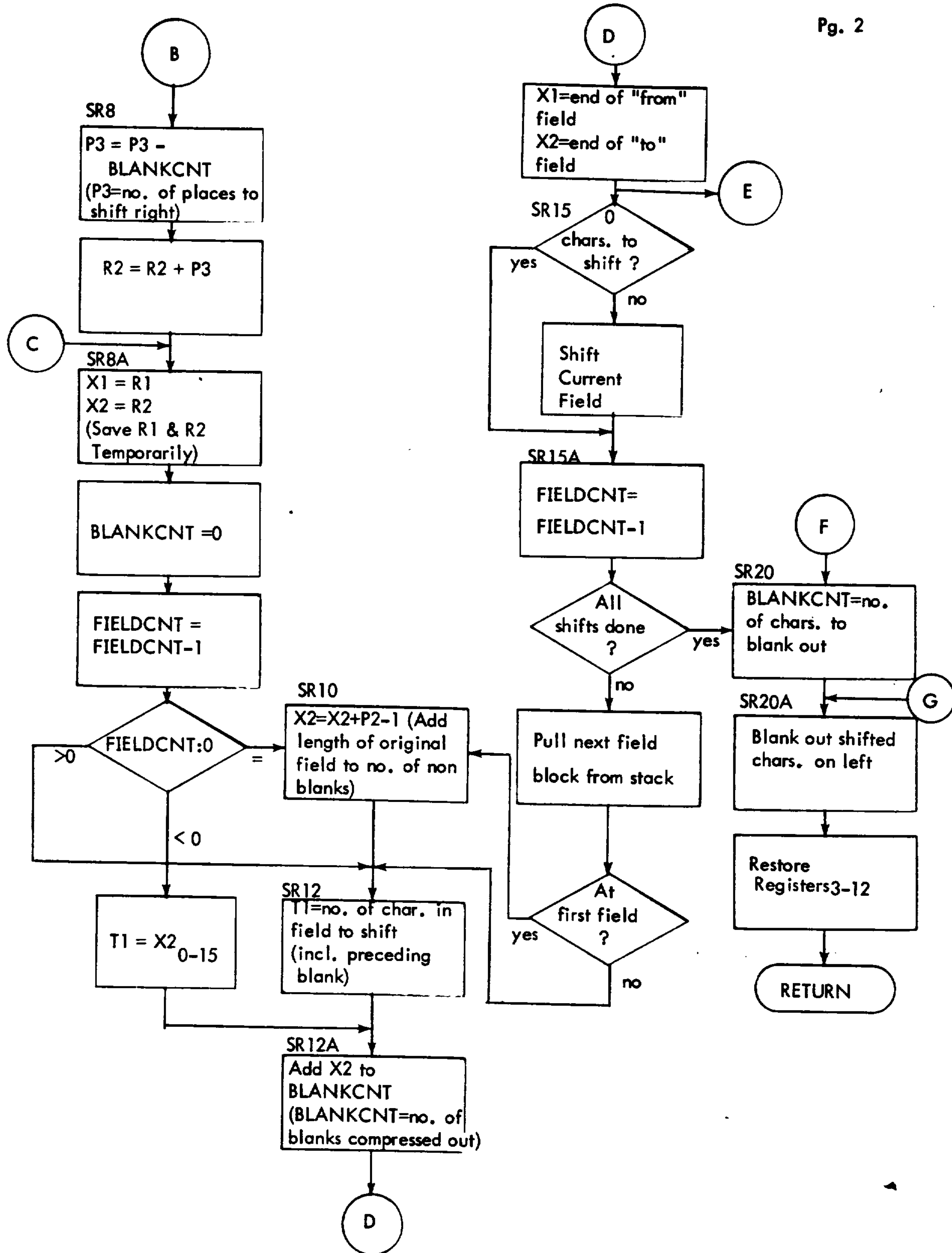



Figure 84. Flow Diagram of SHIFTRIGHT (cont.)

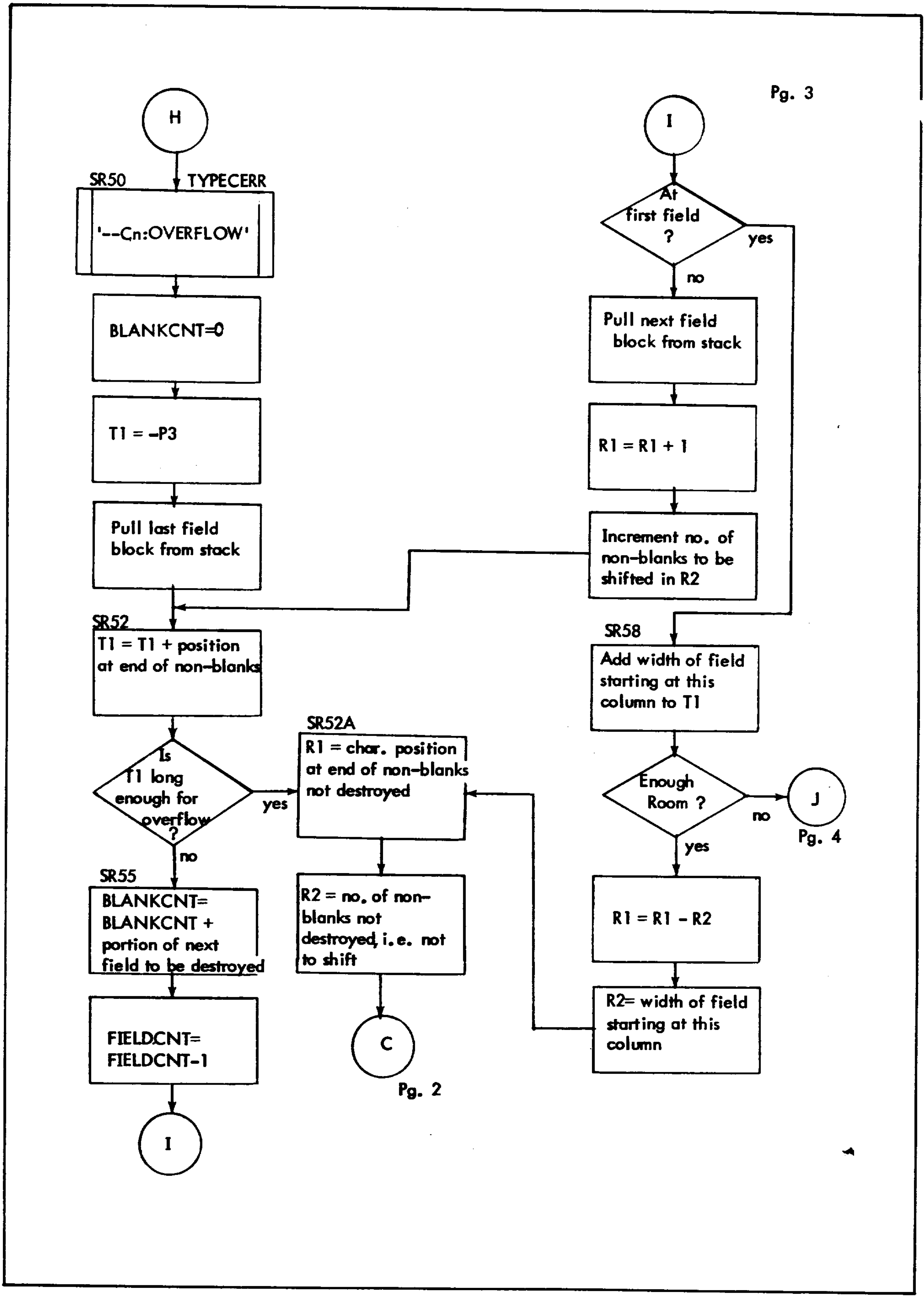



Figure 84. Flow Diagram of SHIFTRIGHT (cont.)

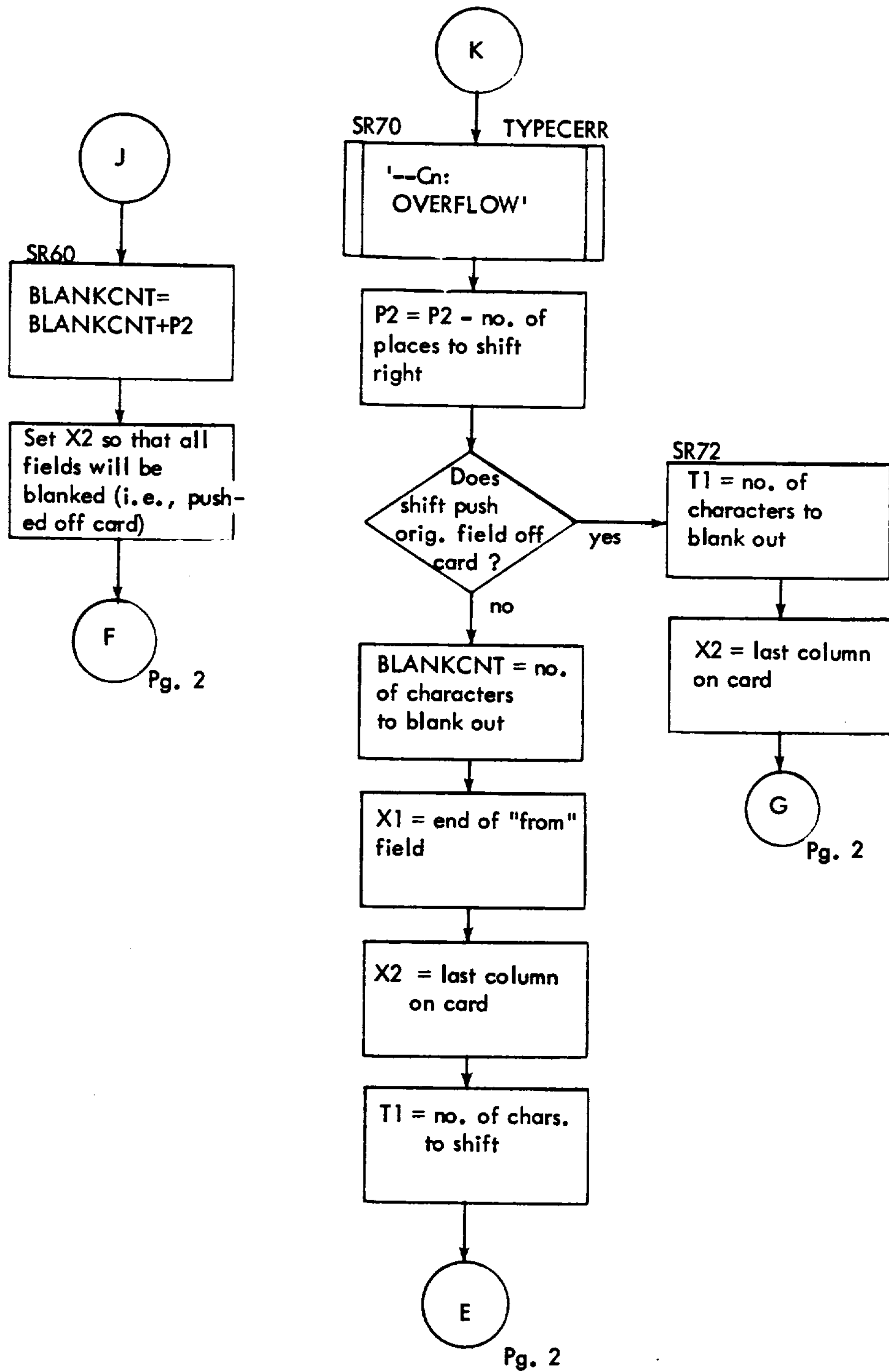



Figure 84. Flow Diagram of SHIFTRIGHT (cont.)

R:SET$STEP

R:SET$STEP$TYPE

R:TYPE$COMPRESSED

R:TYPE$SUP$SEQ

I:TYPE

I:TYPE$SUP$SEQ

Upon entry P1 = sequence number to be typed (< 0 if sequence number is not to by typed).

3. Exit:

   Return is to calling routine via B 0, LNK.

4. Operation:

   It calls MOVESEQ if sequence number is to be typed and calls TYPEMSG to type card contents.

## TYPECERR, TYPEPERR

1. Purpose:

   Types a command error message (TYPECERR) or parameter error message (TYPEPERR).

2. Entry:

   LNK is the linkage register. This subroutine is entered via BAL, LNK TYPECERR or BAL, LNK TYPEPERR from many routines. The word following BAL contains address of message to be printed.

3. Exit:

   B 1, LNK

4. Operation:

   If maximum error messages allowed have been printed it returns. It sets command or parameter number to agree with its place in the command: e.g., '--C2----'. It types message using TYPEMSG.

## TYPEMSG

1. Purpose:

   Remove trailing zeroes and types a message.

2. Entry:

   LNK is the linkage register. This subroutine is entered via BAL, LNK TYPEMSG with next word = word address of TEXTC string.

3. Exit:

   Return is to calling routine via B 1,LNK

4. Operation:

   The Monitor WRITE CAL is used. The DRC mode is normally off, resulting in trailing carriage return and line feed. If the last character of the message is EOM, the mode is changed to suppress carriage control.

TYPESEQ

1. Purpose:

   Types the sequence number 'XXXX.XXX'.

2. Entry:

   LNK is the linkage register. This routine is called via BAL,LNK TYPESEQ from the following routines:

   R:COMMENTARY

   R:FIND$SEQUENCE

   R:FIND$DELETE

   R:INSERT

   R:INSERT$SUP$SEQ

   R:SET$STEP

   R:SET$STEP$TYPE

   TYPECARD

   Upon entry, P1 = sequence number to be typed. The address after the CAL contains four characters to append to sequence number.

3. Exit:

   Return is to calling routine via V 1,LNK.

4. Operation:

   This subroutine calls BINTODEC to convert binary sequence number to decimal; it puts a period between the fourth and fifth digits; it appends the four characters in the calling sequence to the end of sequence number; it suppresses leading zeros in the first three digit positions and it calls TYPEMSG to print sequence number.

## Indexed Scratch File Management

Since the Control Program for Real-time (CP-R) does not support an indexed access method, EDIT provides its own indexed file management when assembled for use under CP-R.

### Indexed File Structure

In the indexing structure used for the CP-R Edit scratch file, the key length is fixed and the data record length is implicit in the data representation. The file is given a granule size of 256 words. Any granule of the file will be unused, used entirely as index, or used entirely as data. The first granule of the file is always an index granule. Whenever more storage is needed for either index or data, the next unused granule of the file is assigned. It will retain this assignment until the indexing structure is discarded.

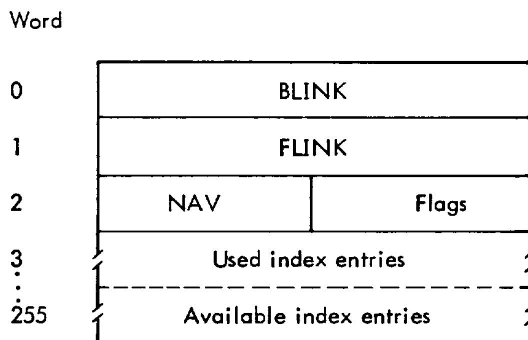
For each record in the file, the index contains an entry that relates the key and the address of the record. The entries are ordered by key value. The granules of the index are linked by pointers in each granule containing the granule numbers of its predecessor and successor.

Data granules of the file are composed of CP-R compressed format records. Data is accessed only via index pointers, so no order relationship or linkage is maintained between data records or granules.

The index entry for each record indicates whether the record is continued or deleted. If a record is continued, it is actually only a fragment of a record. Its index entry is followed by another that describes another record fragment to be appended. Continuation never occurs across an index granule boundary. Instead, all index entries are moved to the next granule, or to an inserted granule, if the next granule has too few available entries. Any service on a record involves all of its fragments treated as a single record. Record continuation is used to provide for the case where a record is overwritten by a longer record. If a record is deleted, it will not be recognized by any indexed file services. Both the index entry and the data space will be reassigned if it is necessary to build a new entry (either original or continuation) at the same point in the index. Otherwise, neither the index nor the data space will be recovered. If a record is deleted, the index entries for all its fragments are marked as deleted.

The formats for the index granule and index entry are given below.

INDEX GRANULE FORMAT




where

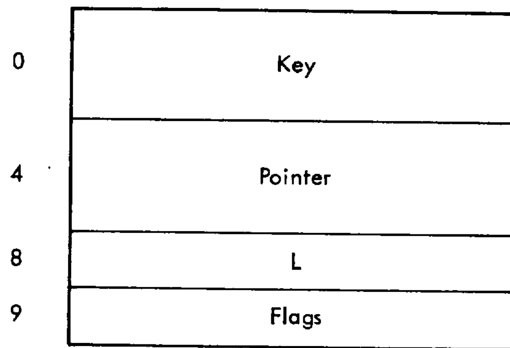
BLINK backwards link, which is the granule number in the file of the previous index granule. BLINK is -1 for the granule containing the smallest key value.

FLINK forwards link, which is the granule number in the file of the next index granule. FLINK is -1 for the granule containing the largest key value.

NAV entry number of the first available index entry. The first entry is number zero.

Flags not currently used.

INDEX ENTRY FORMAT




where

Key is the value of the key for the associated record.

Pointer bits 0-21 is the granule number of the record in the file and bits 22-31 is the byte number of the record in the granule.

L is the length of the record fragment in bytes.

Flags bit 6 indicates the entry is deleted and bit 7 indicates the record is continued in the next entry.

OPENSCR

1. Purpose:

   Opens the CP-R indexed scratch file.

2. Call:

   BAL, LNK OPENSCR

3. Input:

   DCB F:EI must be assigned to the scratch file.

4. Output:

   None.

5. Stack:

   Six.

6. Subroutines:

   READX, OPENSCRI, CLOSESCR, UPKENTRY

7. Operation:

   This subroutine is used when a file is specified for use as the EDIT scratch file but no subject file is named. In this case, it is assumed that the file is either empty or that its contents were generated by previous use as an EDIT scratch file. In the former case, it is initialized as an empty indexed file. In the latter case, its contents are validated, and its next available data byte and next available granule are determined.

   The flow of OPENSCR is given in Figure 85.

CLOSESCR

1. Purpose:

   Closes the CP-R indexed scratch file.

2. Call:

   BAL, LNK CLOSESCR

3. Input:

   DCB F:EI must be assigned to the scratch file.

4. Output:

   None.

5. Stack:

   Two.

6. Subroutines:

   WRGRANS

7. Operation:

   This subroutine writes the current index and data granules to the scratch file if they have been modified. It then closes the F:EI DCB.

   The flow of CLOSESCR is given in Figure 86.

WRGRANS

1. Purpose:

   Writes out index and data granules for CP-R indexed scratch file.

2. Call:

   BAL, LNK WRGRANS

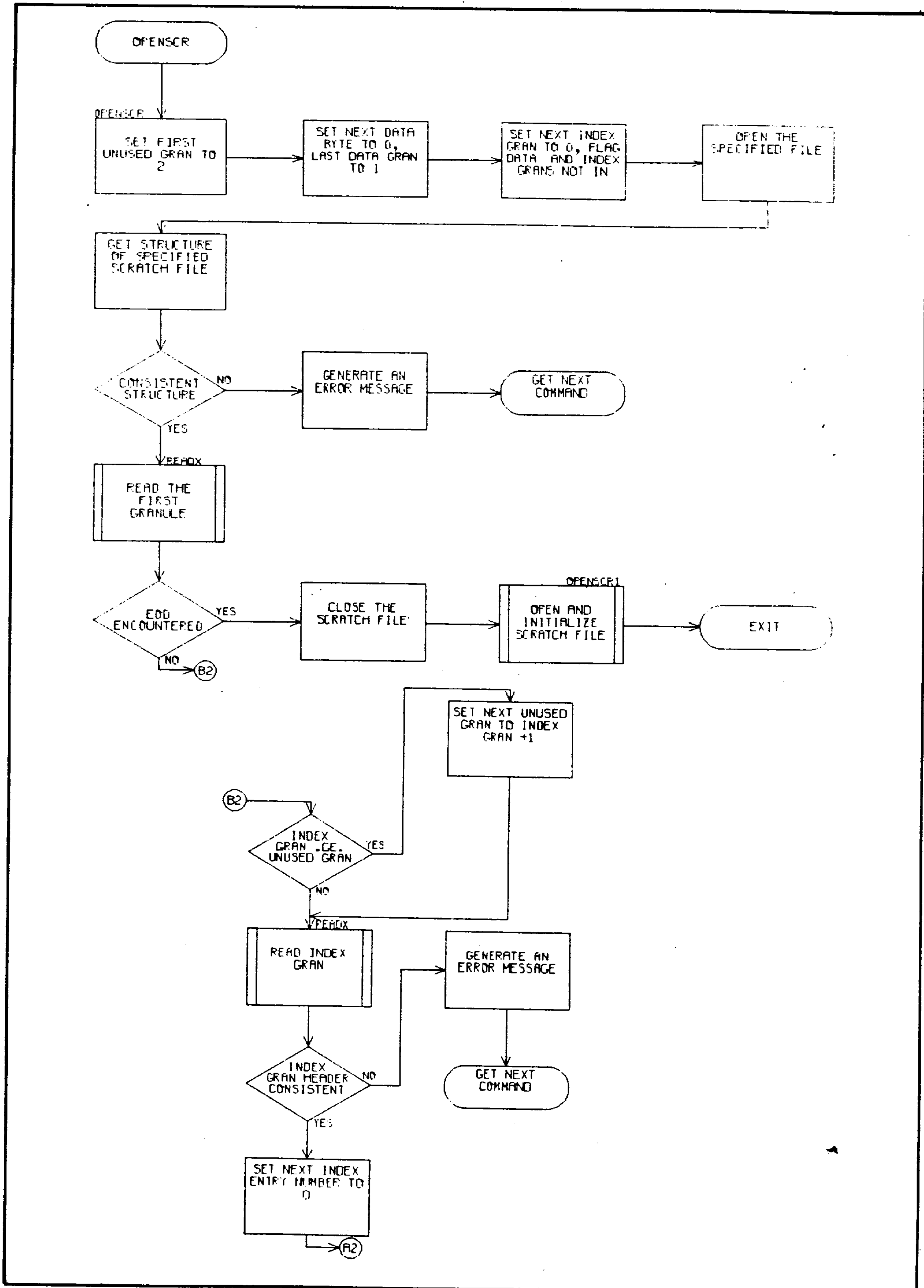



Figure 85. Flow Diagram of OPENSCR

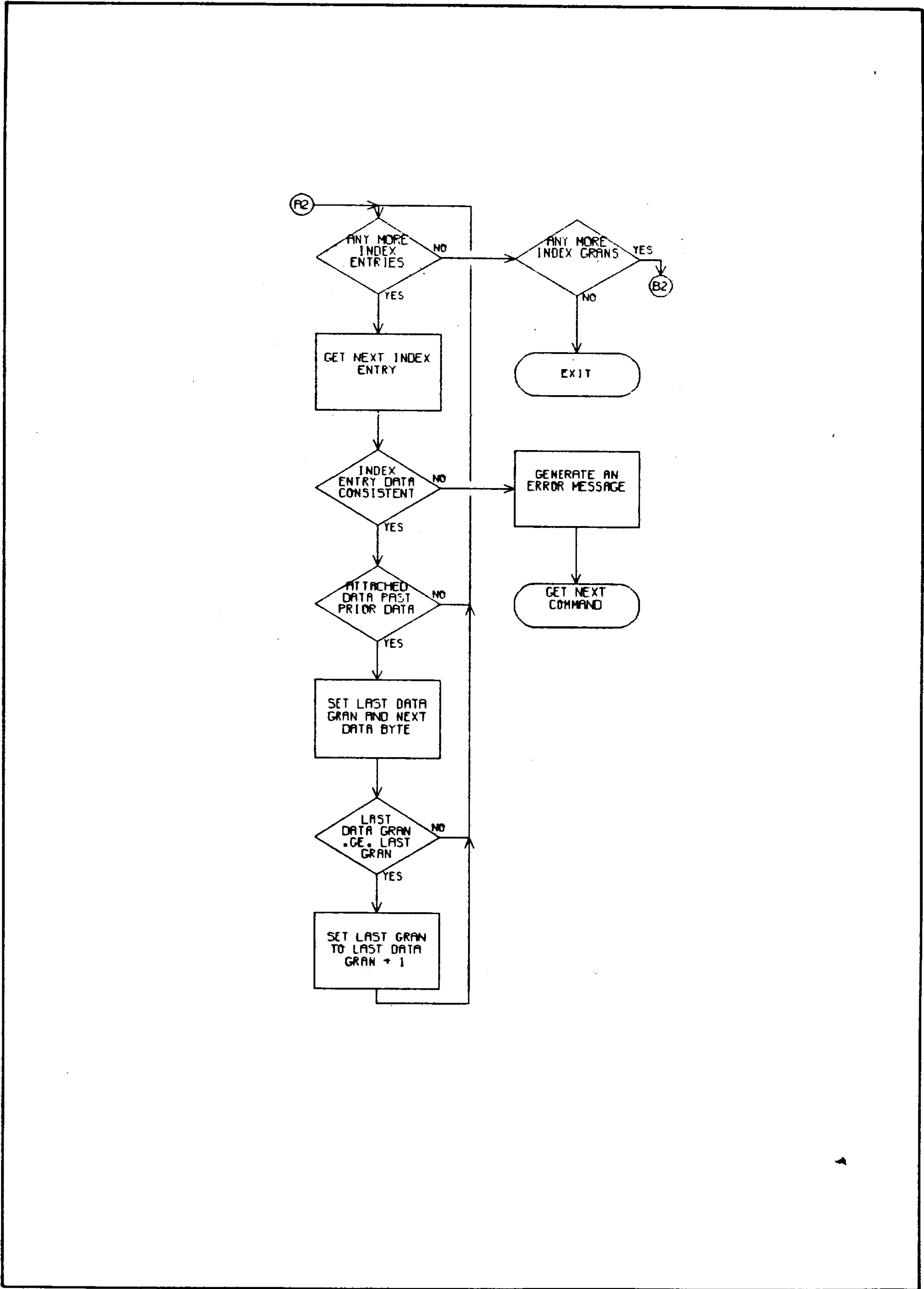



Figure 85. Flow Diagram of OPENSCR (cont.)

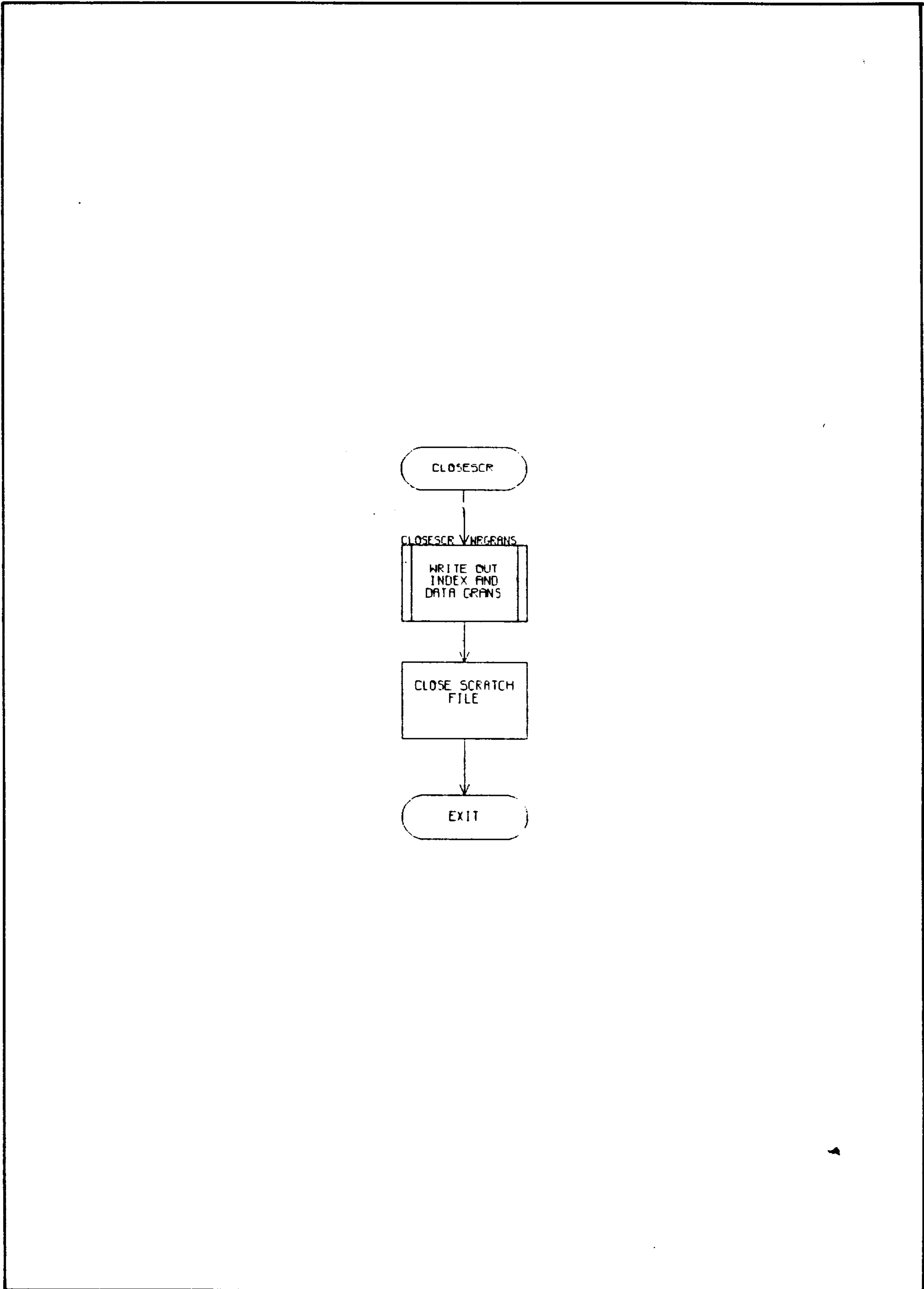



Figure 86. Flow Diagram of CLOSESCR

3. Input:

   None.

4. Output:

   None.

5. Stack:

   One.

6. Subroutines:

   None.

7. Operation:

   For both the current index and current data granule, this subroutine writes the granule to the scratch file if the granule has been modified since it was last read.

   The flow of WRGRANS is given in Figure 87.

OPENSCRI

1. Purpose:

   Opens and initializes the CP-R indexed scratch file.

2. Call:

   BAL, LNK OPENSCRI

3. Input:

   DCB F:EI must be assigned to the scratch file.

4. Output:

   None.

5. Stack:

   Two.

6. Subroutines:

   WRGRANS

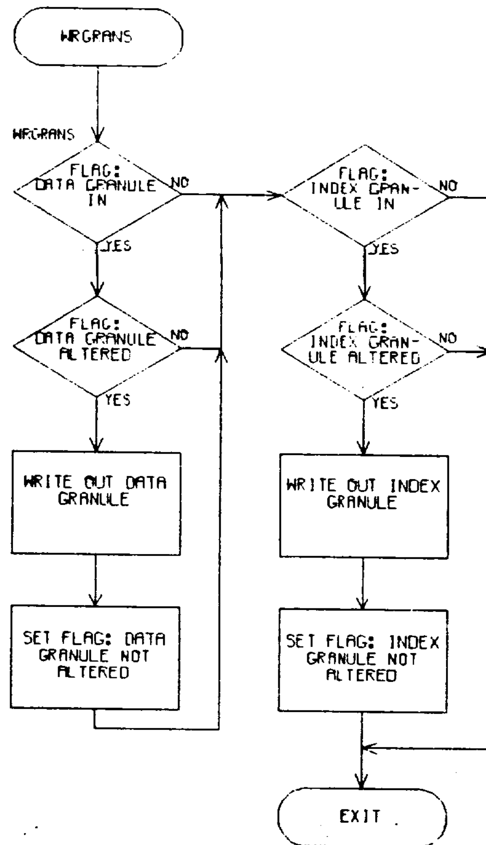



Figure 87. Flow Diagram of WRGRANS

7. Operation:

DCB F:EI is opened. A rewind and a write-end-of-file are issued to the file. The index granule buffer is set to the contents of granule zero of an empty scratch file. Indicators are set such that scratch file granule zero is in the buffer and modified, that byte 0 of granule 1 is the next available data byte, and that granule 2 is the next available granule.

The flow of OPENSCRI is given in Figure 88.

## READD

1. Purpose:

   Reads in a data granule from CP-R scratch file.

2. Call:

   BAL, LNK READD

3. Input:

   P1 = granule number.

4. Output:

   Normal: 10 = 0.

   Scratch file overflow: X'1C' in register 10 byte 0.

5. Stack:

   Two.

6. Subroutines:

   WRGRANS

7. Operation:

   If the required data granule is already in the data granule buffer, this routine returns immediately. If it is not, WRGRANS is called to write out any altered data or index information and the required granule is read into the data granule buffer. Overflow is reported if the required granule is past the file EOT.

   The flow of READD is given in Figure 89.

## READX

1. Purpose:

   Reads in an index granule from CP-R scratch file.

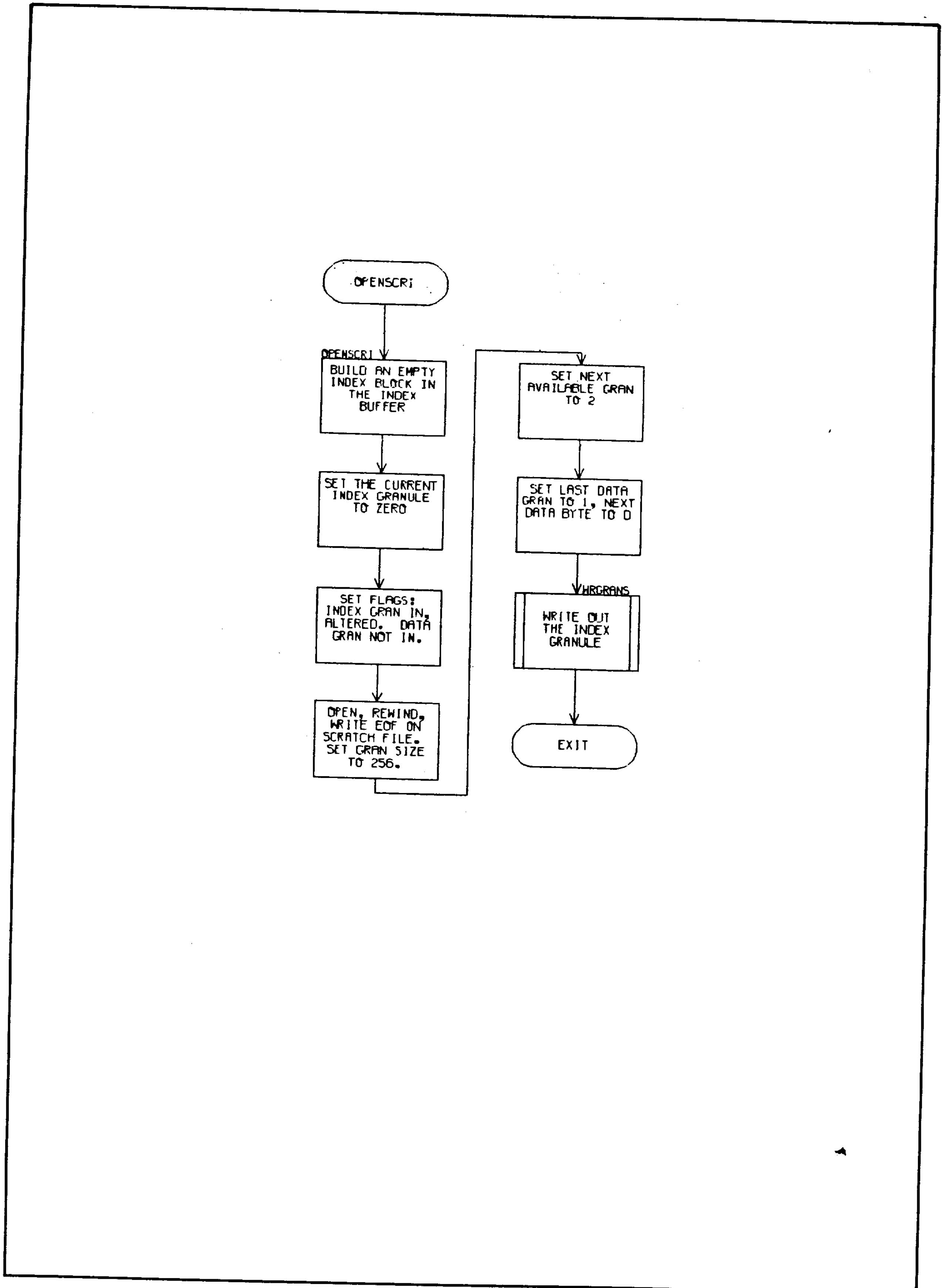



Figure 88. Flow Diagram of OPENSCRI

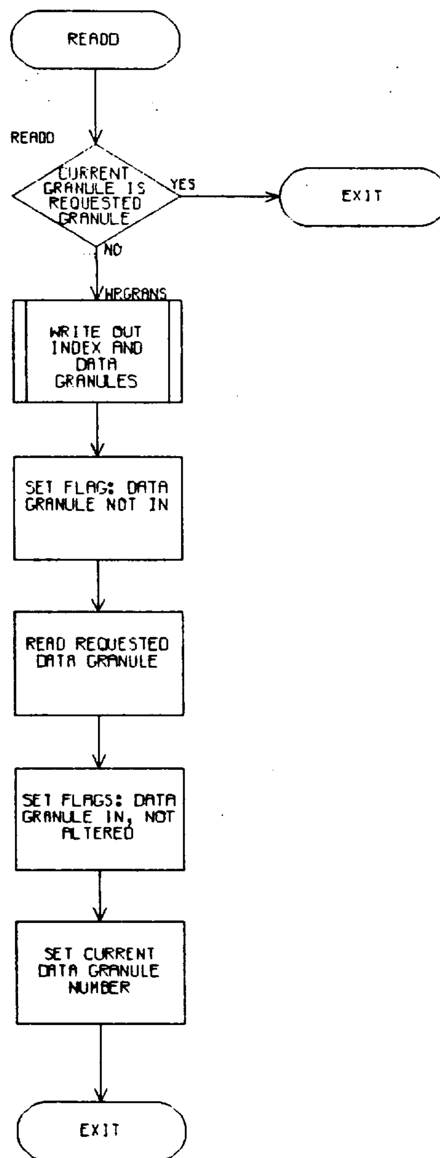



Figure 89. Flow Diagram of READD

2. Call:

BAL, LNK READX

3. Input:

P1 = granule number.

4. Output:

Normal: 10 = 0.

Scratch file overflow: X'1C' in register 10 byte 0.

5. Stack:

Two.

6. Subroutines:

WRGRANS

7. Operation:

If the required granule is already in the index granule buffer, this routine returns immediately. If it is not, WRGRANS is called to write out any modified data or index information, and the required granule is read into the index granule buffer. Overflow is reported if the required granule is beyond the file EOD.

The flow of READX is given in Figure 90.

UPKENTRY

1. Purpose:

Unpacks an index entry from the CP-R scratch file.

2. Call:

BAL, LNK UPKENTRY

3. Input:

P1 = entry number in granule.

4. Output:

None.

5. Stack:

Two.

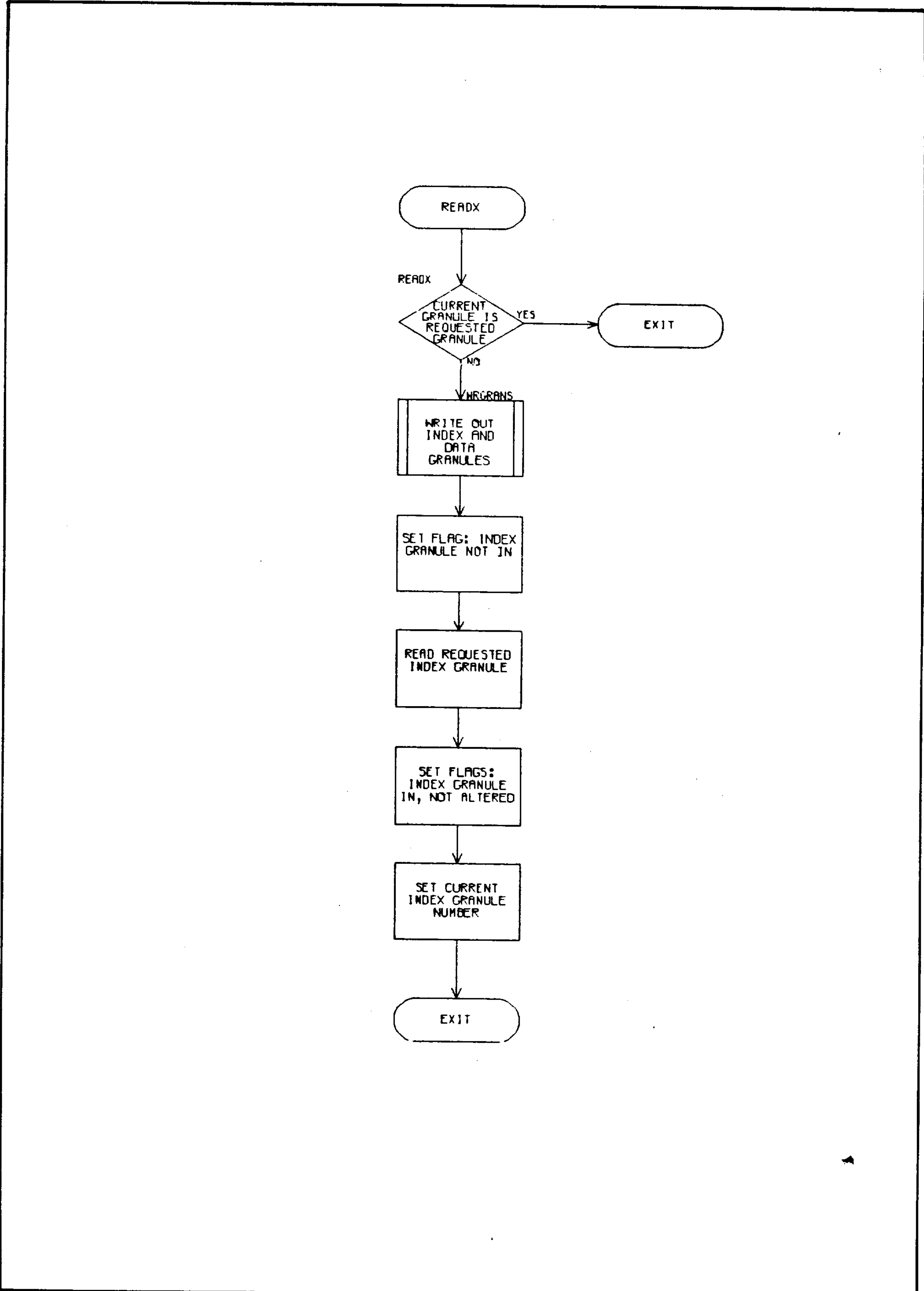



Figure 90. Flow Diagram of READX

6. Subroutines:

None.

7. Operation:

Each field of the indicated entry in the current index granule is moved into a full-word area in the EDITOR context.

PKENTRY

1. Purpose:

Packs an index entry into the CP-R scratch file index buffer.

2. Call:

BAL, LNK PKENTRY

3. Input:

P1 = entry number in granule.

4. Output:

None.

5. Stack:

Two.

6. Subroutines:

None.

7. Operation:

Each field of the indicated entry in the current index granule is overwritten with the data from a full-word area in the EDITOR context.

FINDX

1. Purpose:

Finds index entry for specified key in CP-R scratch file.

2. Call:

BAL, LNK FINDX

3. Input:

P1 = key value.

4. Output:

   R1 = entry number of entry returned.

   If key is found, entry for key is returned, and unpacked; 10 = 0.

   If key found but deleted, entry for key is returned and unpacked; 10 = X'43'.

   Entry not found; not past end of file; entry for first key following specified one is returned and unpacked; 10 = '43'.

   Entry not found and past EOF; next available entry in last index granule is returned, not unpacked (since not yet written); 10 = X'43'.

5. Stack:

   Six.

6. Subroutines:

   READX, UPKENTRY

7. Operation:

   The index granule chain is scanned in reverse from the current granule until a granule is found that is the first index granule or does not entirely follow the given key. Then the chain is scanned forward until a granule is found which is the last index granule or does not entirely precede the given key. Then the entries of this granule are searched forward until one is found which is equal to or past the given key. This entry is returned.

   The flow of FINDX is given in Figure 91.

FINDNXX

1. Purpose:

   Finds index entry for next key after specified one. If none, indicates entry for last key in file.

2. Input:

   P1 = key for which to find successor.

3. Output:

   Successor found:

   R1 = entry number of successor; XBUFF contains correct index granule; 10 = 0.

   Successor not found; R1 = entry number of last key of file; XBUFF contains its index granule; 10=X'06' in byte zero.

4. Stack:

   Four.

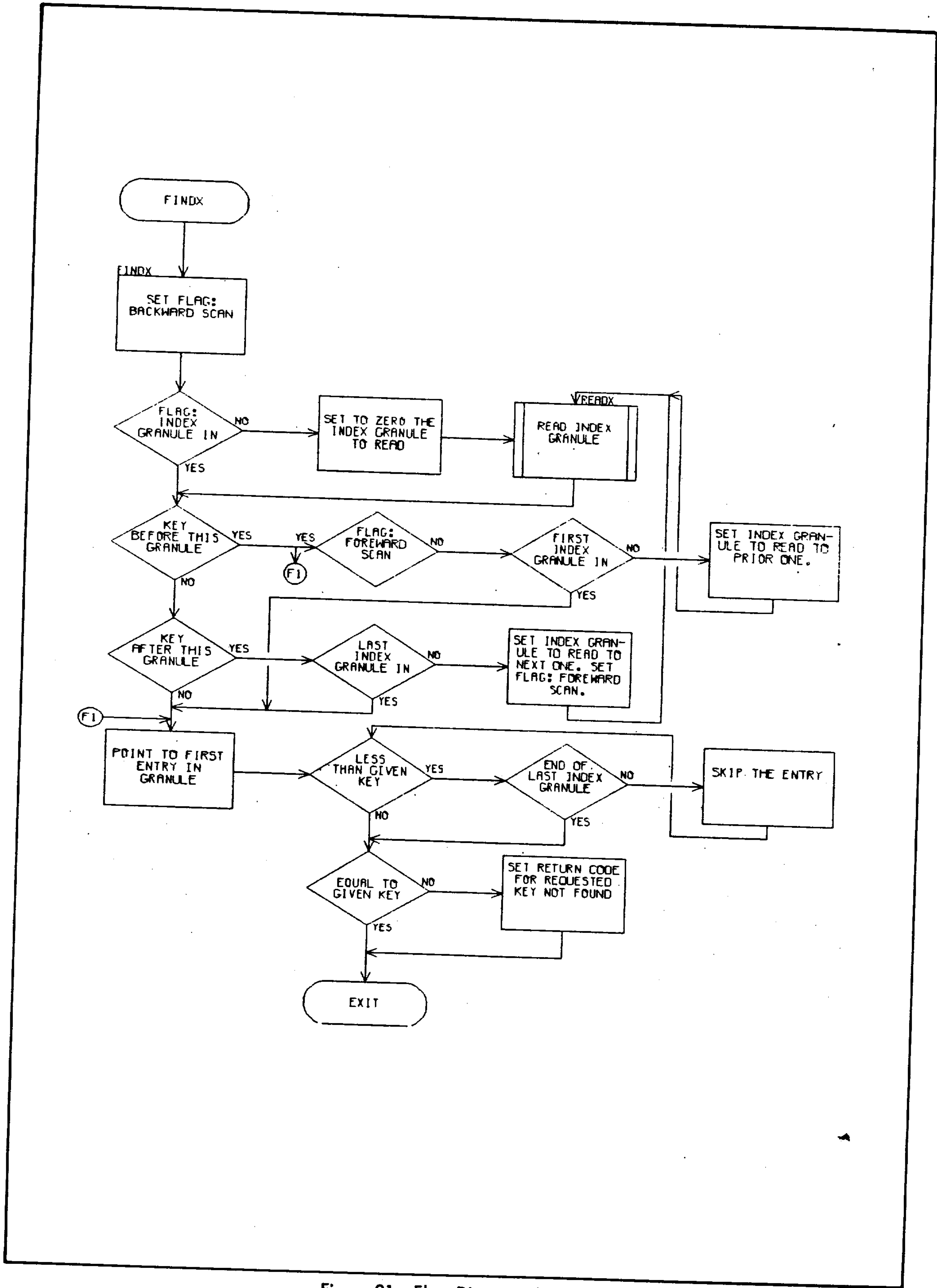



Figure 91. Flow Diagram of FINDX

5. Subroutines:

   FINDX, READX, UPKENTRY

6. Operation:

   FINDX is called for the specified key. If the returned key is the same or is deleted, the index is scanned forward to the first nondeleted entry. If at any time the end of the index is encountered, end-of-file is reported, and the index is scanned in reverse until the first undeleted entry is found. This is the returned entry.

   The flow of FINDNXX is given in Figure 92.

GETX

1. Purpose:

   Gets the necessary index entries for writing a record to the CP-R scratch file. It uses existing and deleted entries, when possible to ensure that the entries obtained include enough attached data space.

2. Call:

   BAL, LNK GETX

3. Input:

   P1 = key value. Record text is in the buffer labeled CARDIMG.

4. Output:

   Normal: 10 = 0 if key previously existed; = X'43' if not correct index granule read in. R1 = entry number for first entry obtained.

   Scratch file overflow; 10 = X'06' in byte zero.

5. Stack:

   Six.

6. Subroutines:

   FINDX, READX, WRGRANS, PKENTRY

7. Operation:

   The amount of data space needed is determined. Any existing entries for the key are assigned. If more space is needed, adjacent deleted entries are also assigned. If still more space is needed, a new index entry is inserted at the front of the assigned set of entries, and the balance of the data space needed is attached to it. To make room for the new entry, other entries may be shifted to the following index granule or a new index granule may be created.

   The flow of GETX is given in Figure 93.

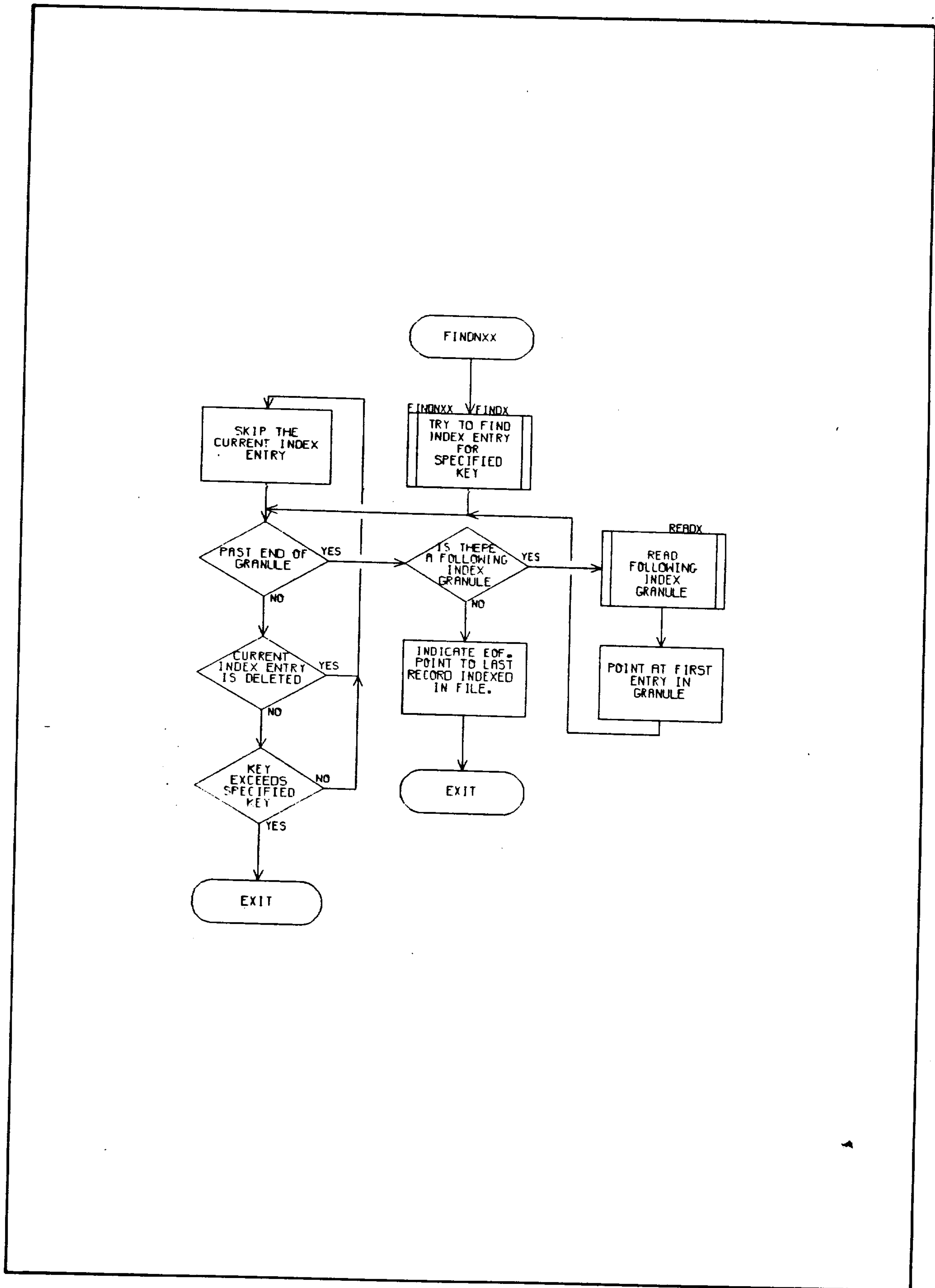



Figure 92. Flow Diagram of FINDNXX

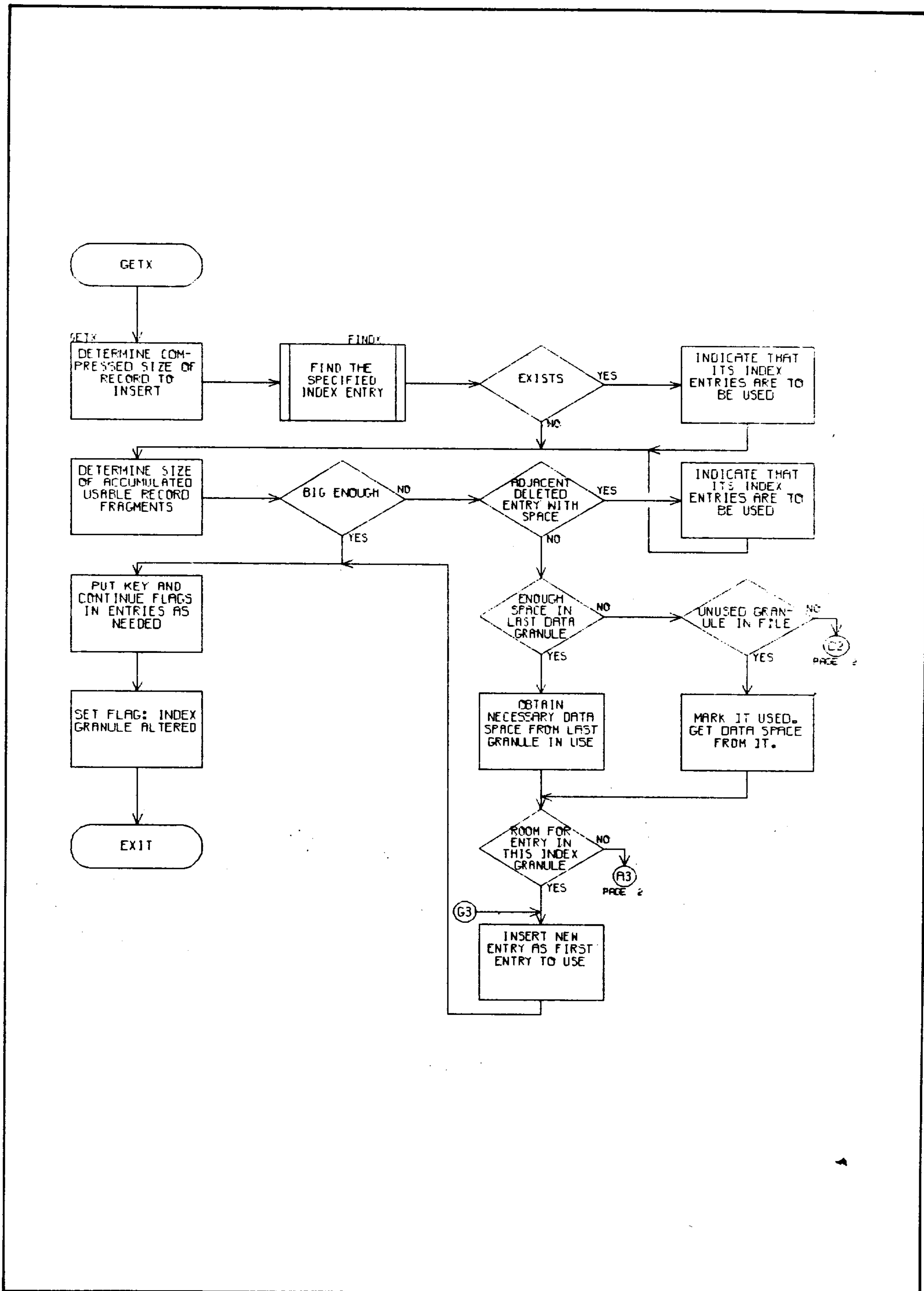



Figure 93. Flow Diagram of GETX

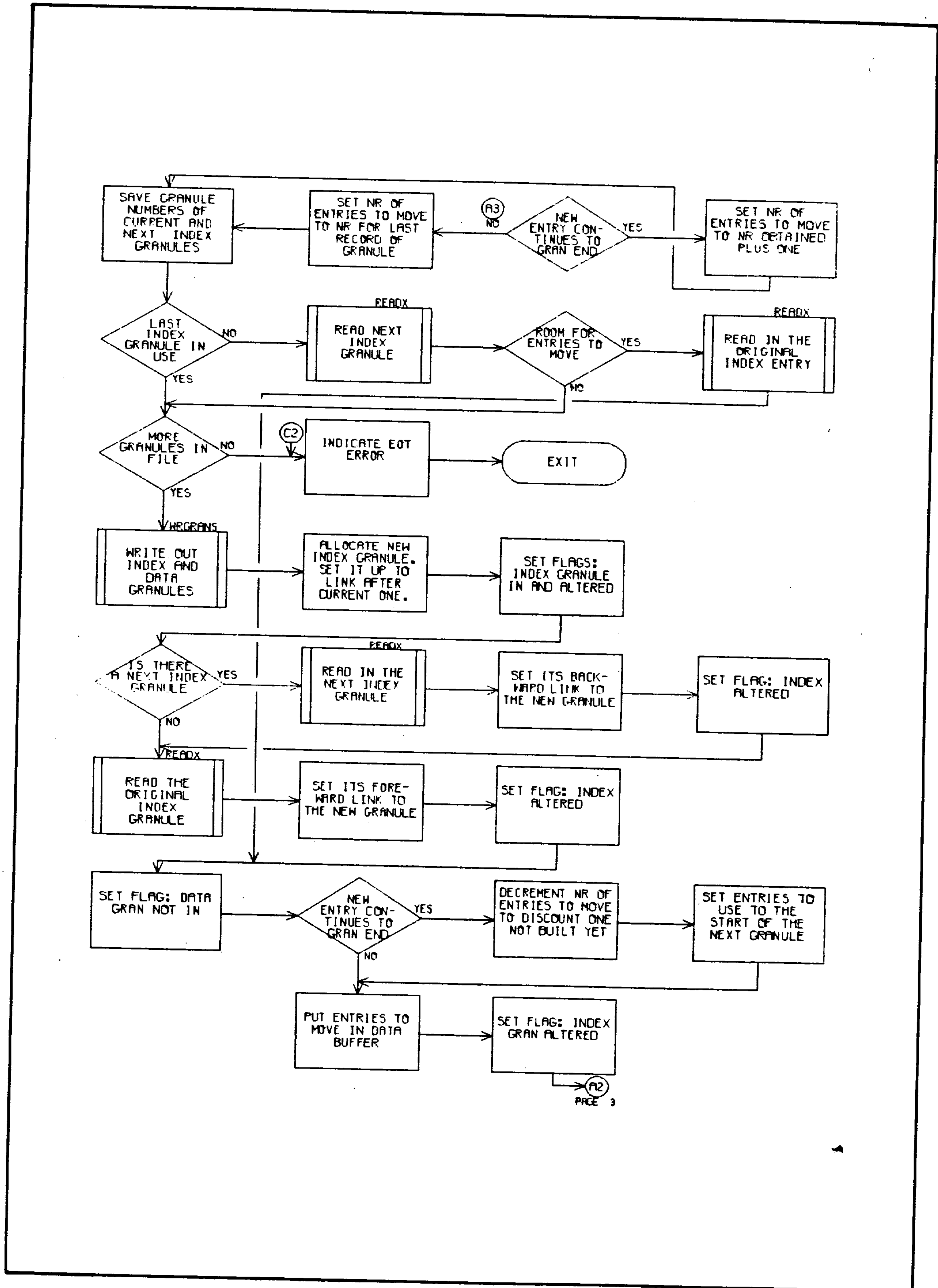



Figure 93. Flow Diagram of GETX (cont.)

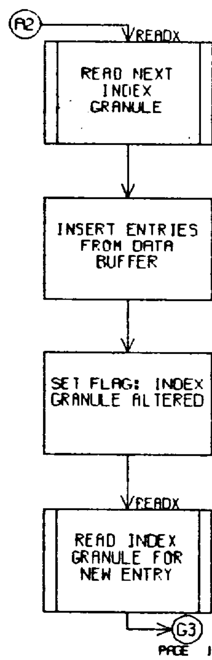



Figure 93. Flow Diagram of GETX (cont.)

## DATA PACK/UNPACK CONVENTIONS

The following definitions describe register use conventions in the subroutines used to move and pack/unpack data between the user buffers and the data granule buffer. These subroutines are GETREC, PUTREC, GETRBYTE and PUTRBYTE.

The conventions are as follows:

| | |
|---|---|
| BLANKCT = P2 | Blank count for multiple blank expansion/compression. |
| NEXTX = P1 | Next index entry to access when current data area used up. |
| STRPTR = X1 | Pointer to next byte in user I/O byte string. |
| STRCT = X2 | Remaining byte count for user I/O byte string. |
| RECPTR = X3 | Pointer to next byte in data granule buffer. |
| RECCT = X4 | Remaining byte count for current block of data in data granule buffer. |

## GETREC

1. Purpose:

   Gets a data record given its index entries.

2. Call:

   BAL, LNK GETREC

3. Input:

   P1 = NEXTX = entry number for first index entry to use.

4. Output:

   None.

5. Stack:

   Eight.

6. Subroutines:

   GETRBYTE

   Note: See Data Pack/Unpack Conventions above.

7. Operation:

   The registers are set up by the Data Pack/Unpack Conventions. The routine then loops calling GETRBYTE to get bytes from the record, expanding multiple blank string representations, and storing the results in the record buffer until the requested count is exhausted or an end-of-record character is obtained.

   The flow of GETREC is given in Figure 94.

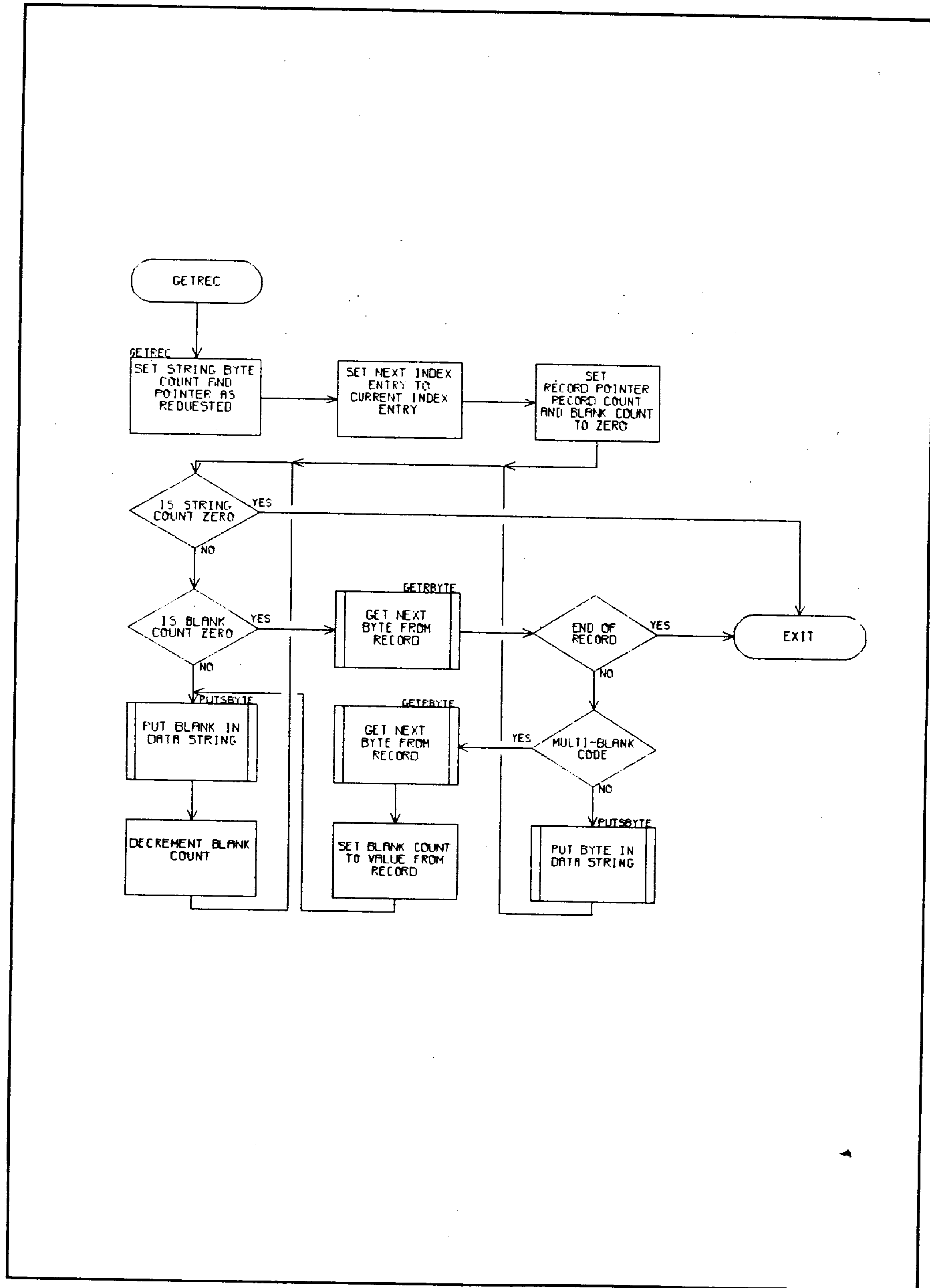



Figure 94. Flow Diagram of GETREC

PUTREC

1. Purpose:

   Puts a data record into the CP-R indexed scratch file, given the index entries for the record.

2. Call:

   BAL, LNK PUTREC

3. Input:

   P1 = NEXTX = entry number for first index entry to use.

4. Output:

   None.

5. Stack:

   Eight.

6. Subroutines:

   PUTRBYTE

   See Data Pack/Unpack Conventions above.

7. Operation:

   The registers are set up by the Data Pack/Unpack Conventions. Characters are obtained from the record buffer, multiple blank strings are compressed, and the results are inserted in the data buffer using PUTRBYTE, until the requested character count is exhausted. Finally, an end-of-record character is inserted.

   The flow of PUTREC is given in Figure 95.

GETRBYTE/PUTRBYTE

1. Purpose:

   Gets/puts byte of CP-R indexed file data.

2. Call:

   BAL, LNK GETRBYTE

   or

   BAL, LNK PUTRBYTE

3. Input:

   Registers set up as in Data Pack/Unpack Conventions. For PUTRBYTE only; T1 = byte to insert.

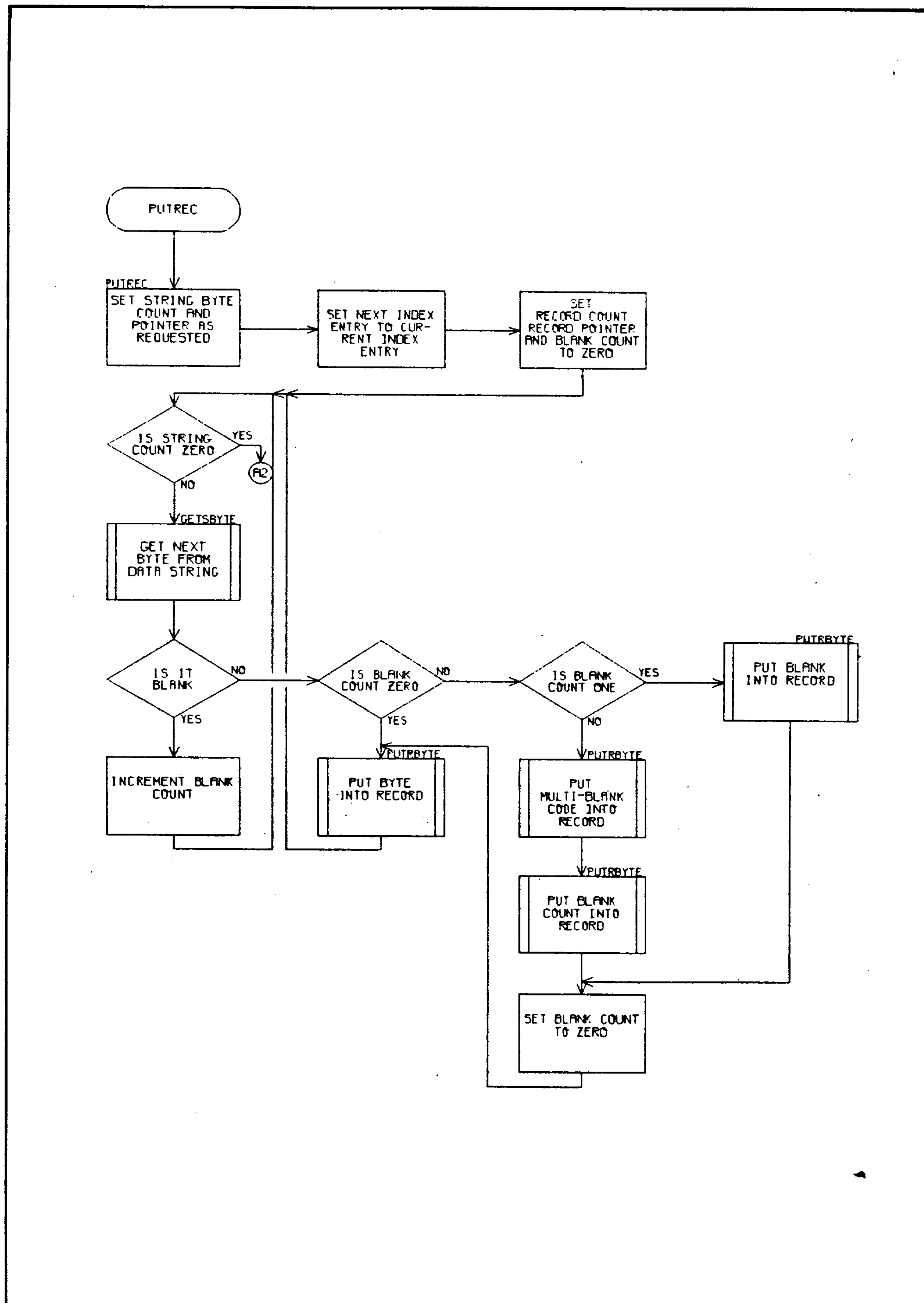



Figure 95. Flow Diagram of PUTREC

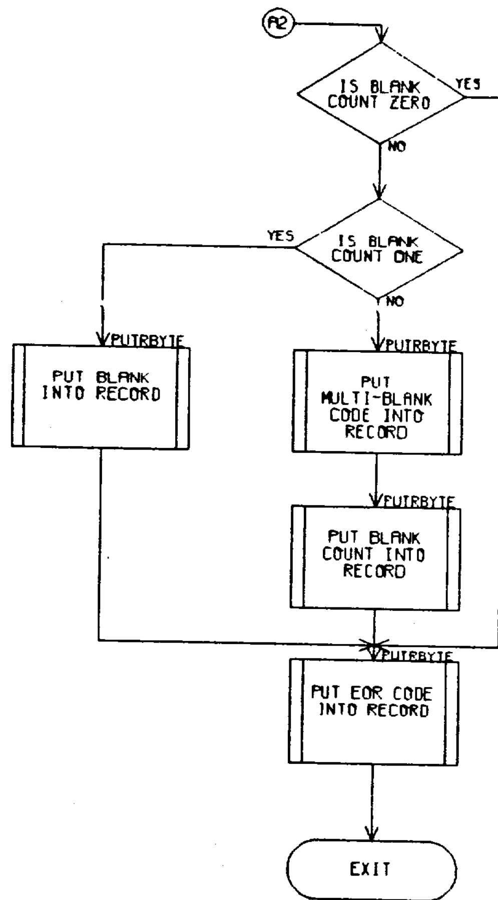



Figure 95. Flow Diagram of PUTREC (cont.)

4. Output:

   For GETRBYTE only; T1 = byte obtained.

5. Stack:

   One.

6. Subroutines:

   READD.

7. Operation:

   When no data remains in the current data block either routine accesses the next index entry, and reads the indicated data granule to get the next block of data for a record. It then transfers the data in the block, one character per call, until the block is exhausted.

   The flow of GETRBYTE/PUTRBYTE is given in Figure 96.

DELETERECORD

1. Purpose:

   Deletes the most recently read record.

2. Call:

   BAL, LNK   DELETERECORD

3. Input:

   None.

4. Output:

   None.

5. Stack:

   Four.

6. Subroutines:

   FINDX, UPKENTRY, PKENTRY

7. Operation:

   Sets the "DELETED" flag in all index entries for the record, and indicates that the current index granule has been altered.

   The flow of DELETERECORD is given in Figure 97.

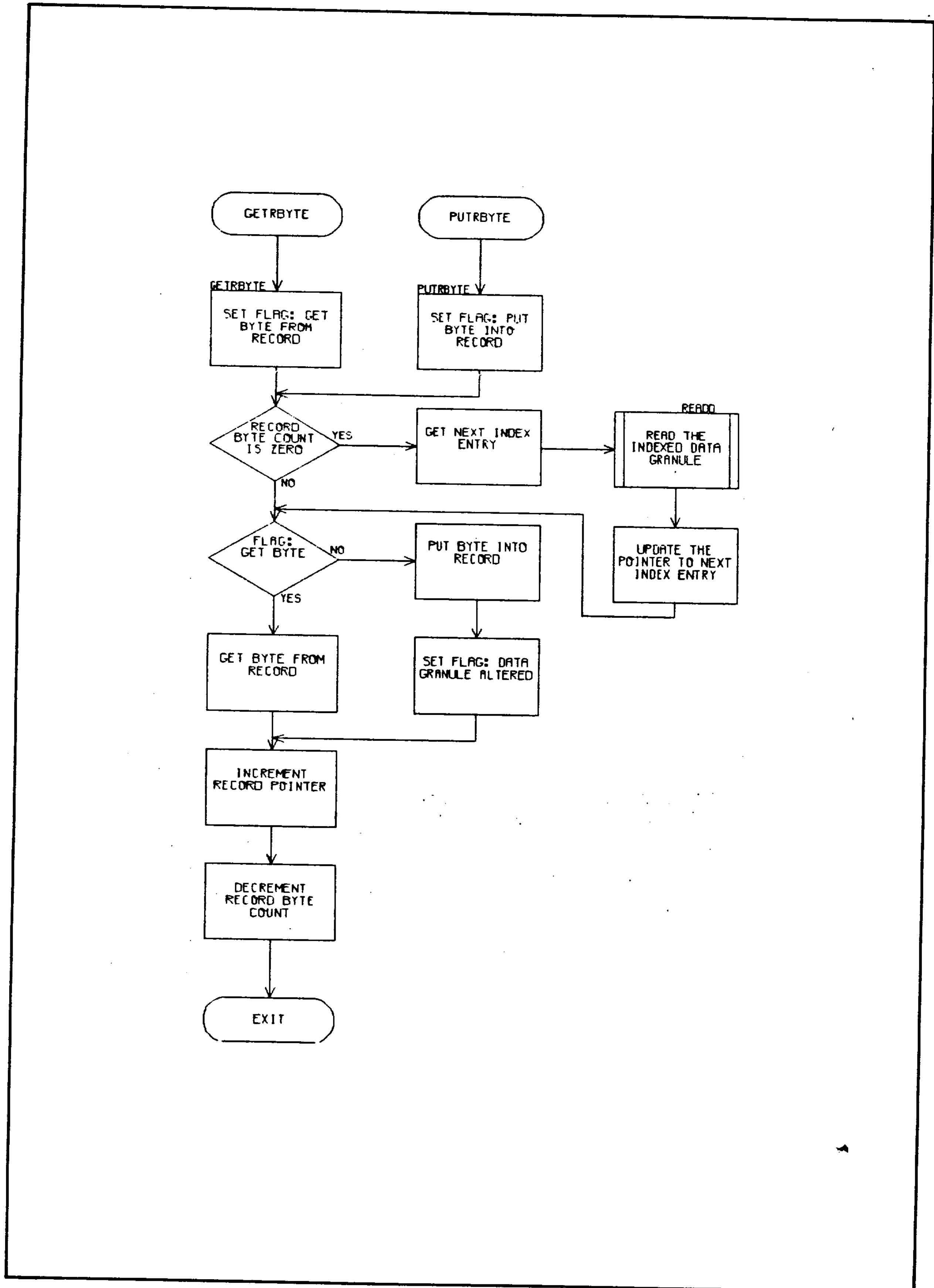



Figure 96. Flow Diagram of GETRBYTE/PUTRBYTE

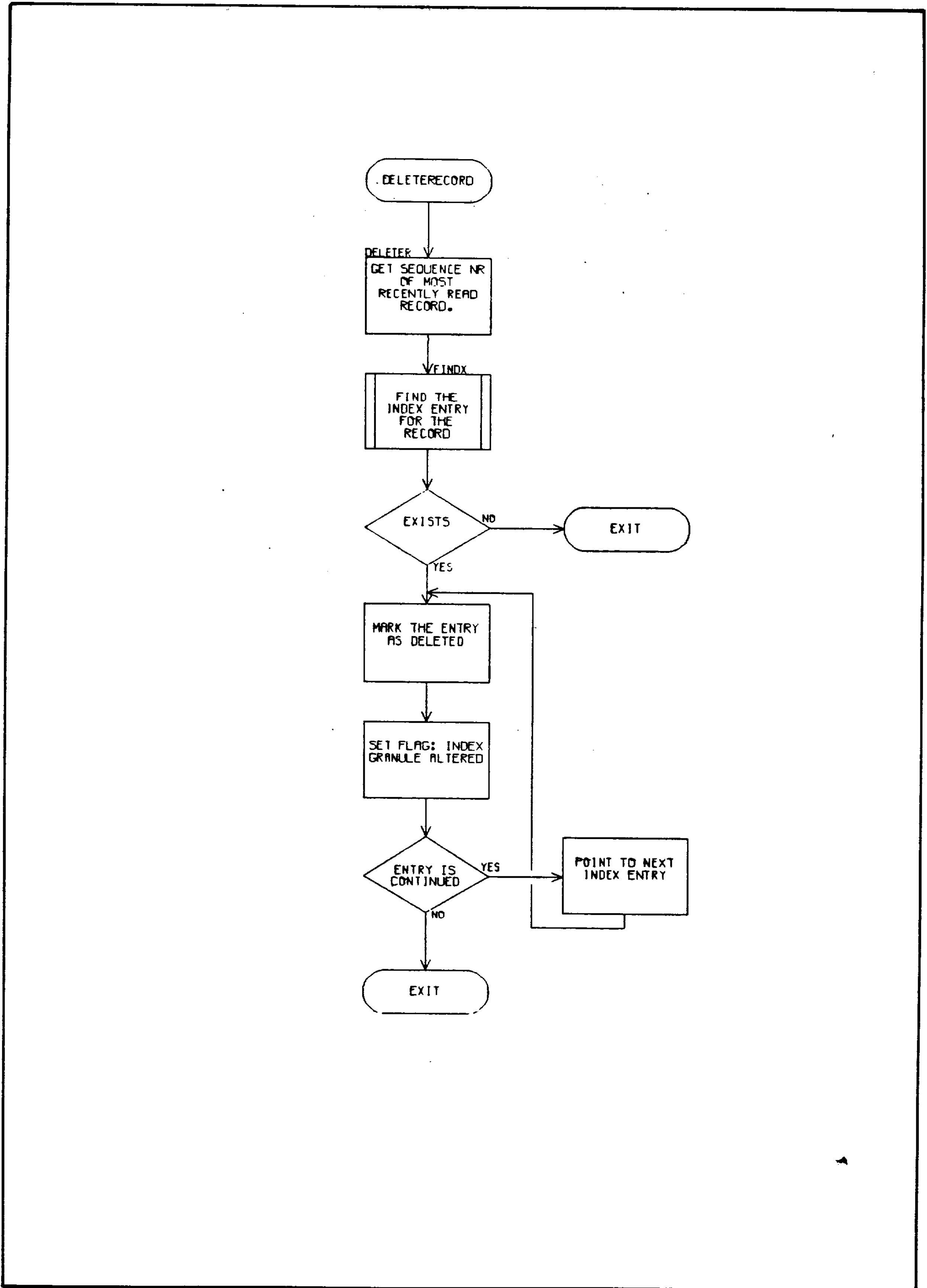



Figure 97. Flow Diagram of DELETERECORD

WRITERANDOM

1. Purpose:

   Writes a record into the CP-R indexed scratch file.

2. Call:

   BAL, LNK WRITERANDOM

3. Input:

   P1 = key. Data to write in CARDIMG. Record length in RECSIZE.

4. Output:

   None.

5. Stack:

   Four.

6. Subroutines:

   GETX, PUTREC, SCROFLO

7. Operation:

   This routine calls GETX to get an index entry(ies) for the record to be written. If the scratch file does not overflow, WRITERANDOM uses PUTREC to move the record into the data space attached to the entry(ies) obtained.

   The flow of WRITERANDOM is given in Figure 98.

WRITENEWRANDOM

1. Purpose:

   Writes a new record into the CP-R indexed scratch file.

2. Call:

   BAL, LNK WRITENEWRANDOM

3. Input:

   P1 = key. Data to write in CARDIMG. Record length in RECSIZE.

4. Output:

   New entry; CC = 0. Not new entry; CC = 8.

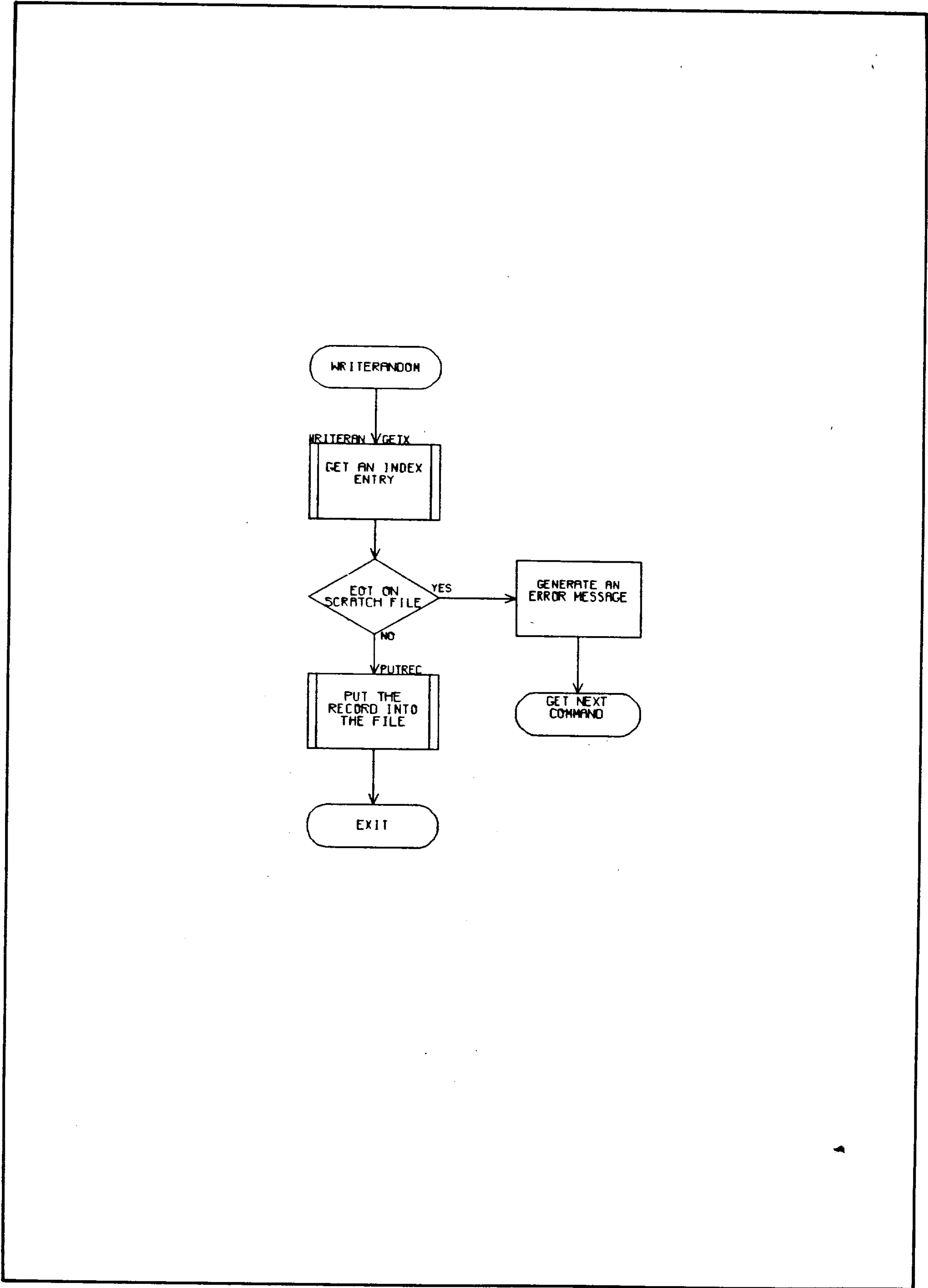



Figure 98. Flow Diagram of WRITERANDOM

5. Stack:

   Four.

6. Subroutines:

   FINDX, GETX, PUTREC, SCROFLO

7. Operation:

   This routine first calls FINDX to determine if the specified routine already exists. If it does not, WRITENEWRANDOM calls GETX to get an index entry for the record, and PUTREC to move the record into the data space obtained with the index entry.

   The flow of WRITENEWRANDOM is given in Figure 99.

READRANDOM

1. Purpose:

   Reads a record from the CP-R indexed scratch file.

2. Call:

   BAL, LNK READRANDOM

3. Input:

   P1 = key. Data byte length in RECSIZE.

4. Output:

   Key found:

   CC = 0.

   Data in CARDIMG.

   Key not found:

   CC = 8.

5. Stack:

   Four.

6. Subroutines:

   BLANKBUF, FINDX, GETREC, SETLASTKEY

7. Operation:

   This routine calls FINDX to determine the location of the record. If the record exists, READRANDOM moves it into the record buffer by calling GETREC.

   The flow of READRANDOM is given in Figure 100.

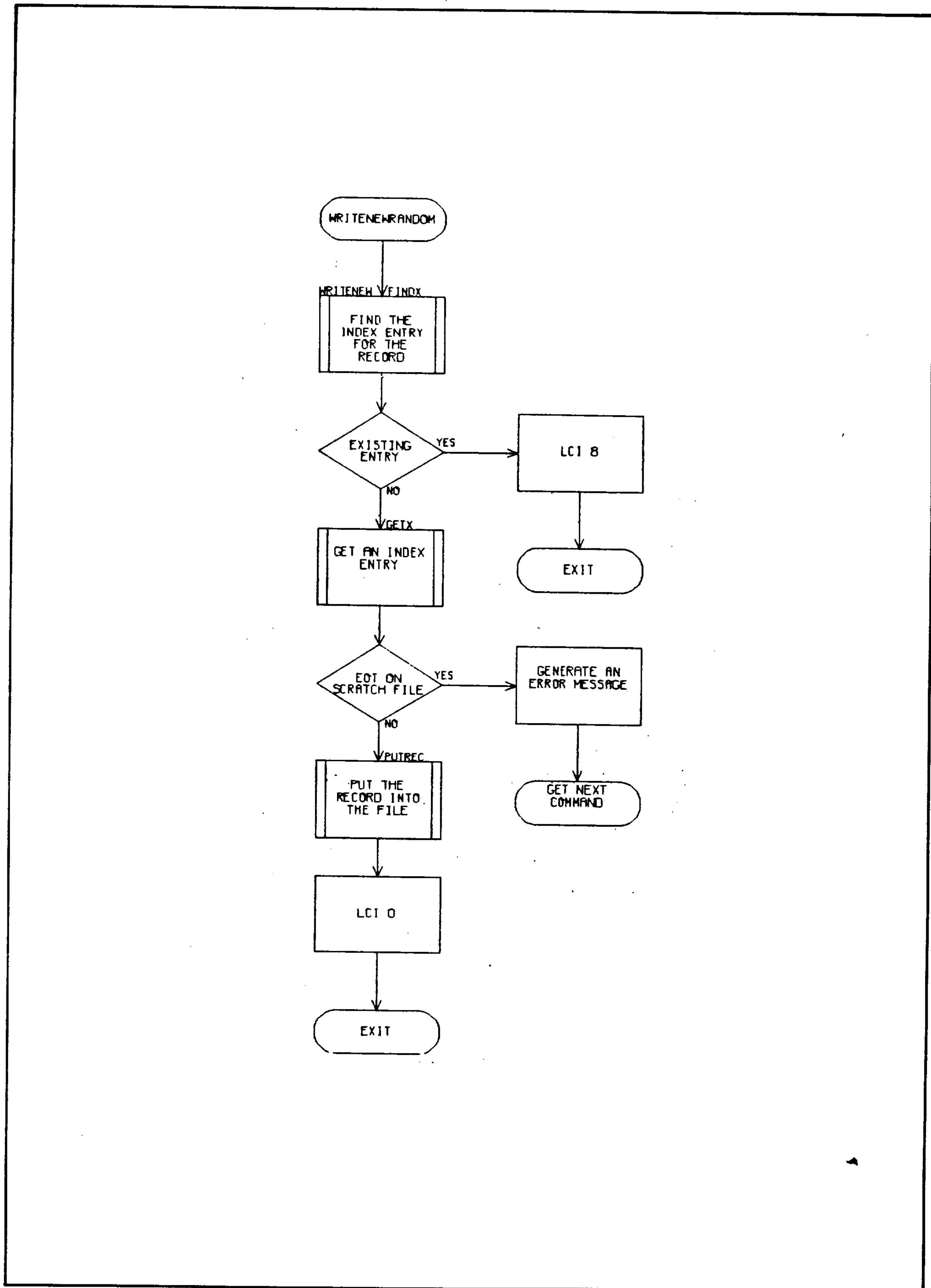



Figure 99. Flow Diagram of WRITENEWRANDOM

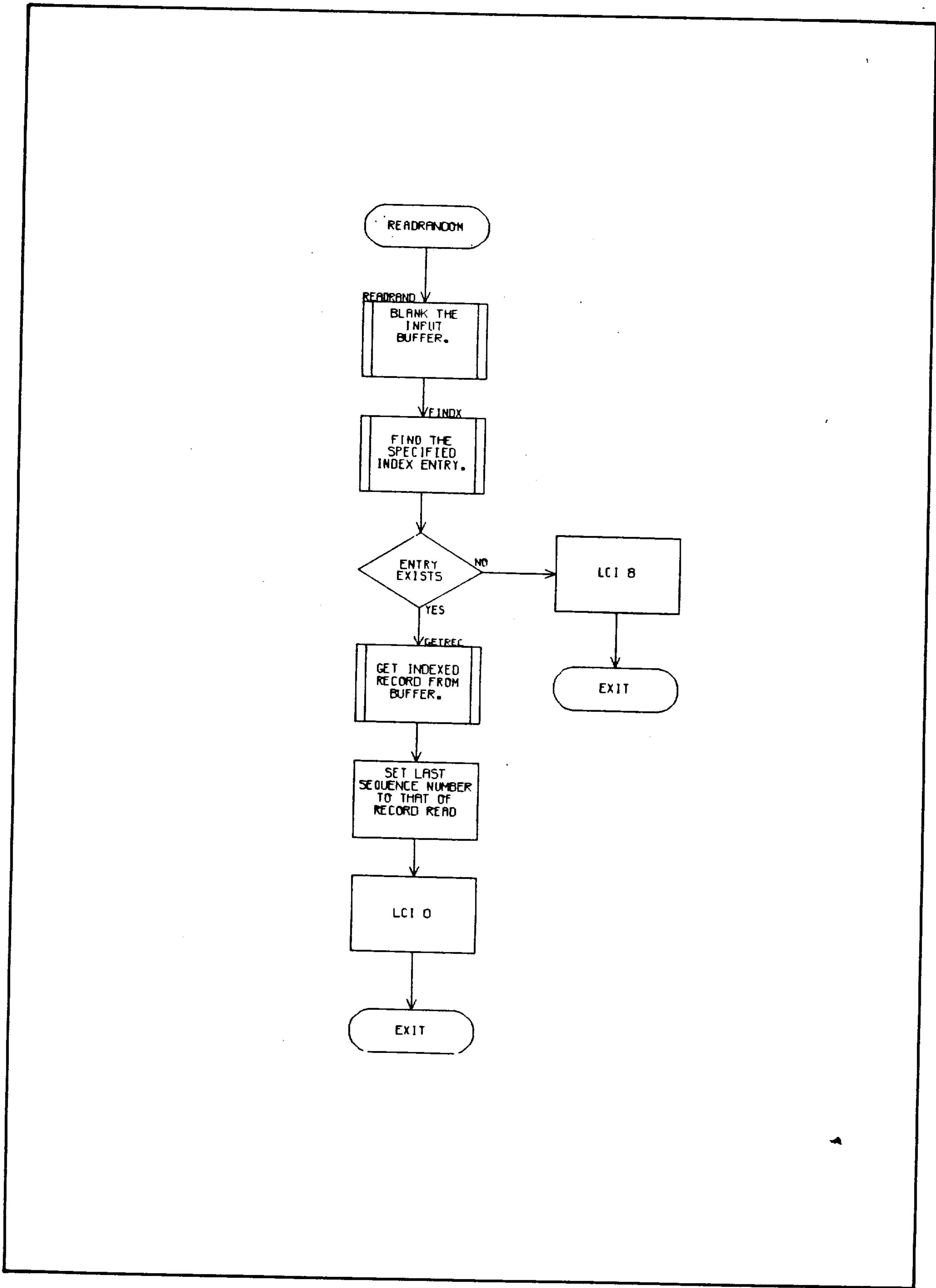



Figure 100. Flow Diagram of READRANDOM

## READSEQUEN

1. Purpose:

   Reads sequentially a CP-R indexed scratch file.

2. Call:

   BAL, LNK   READSEQUEN

3. Input:

   Data byte length in RECSIZE.

4. Output:

   Data in CARDIMG.  R1 = key of record read.

5. Stack:

   Six.

6. Subroutines:

   BLANKBUF, FINDNXX, GETREC, SETLASTKEY

7. Operation:

   This routine uses FINDNXX to locate the record that succeeds the last one read or written. If there is a successor, READSEQUEN moves it into the record buffer by calling GETREC. If there is no successor, READSEQUEN indicates an end-of-file.

   The flow of READSEQUEN is given in Figure 101.

## BUILDSCR

1. Purpose:

   Builds the CP-R indexed scratch file from the subject file.

2. Call:

   BAL, LNK  BUILDSCR

3. Input:

   M:EI set to scratch file, M:EO set to subject file.  Both DCBs closed.

4. Output:

   None.

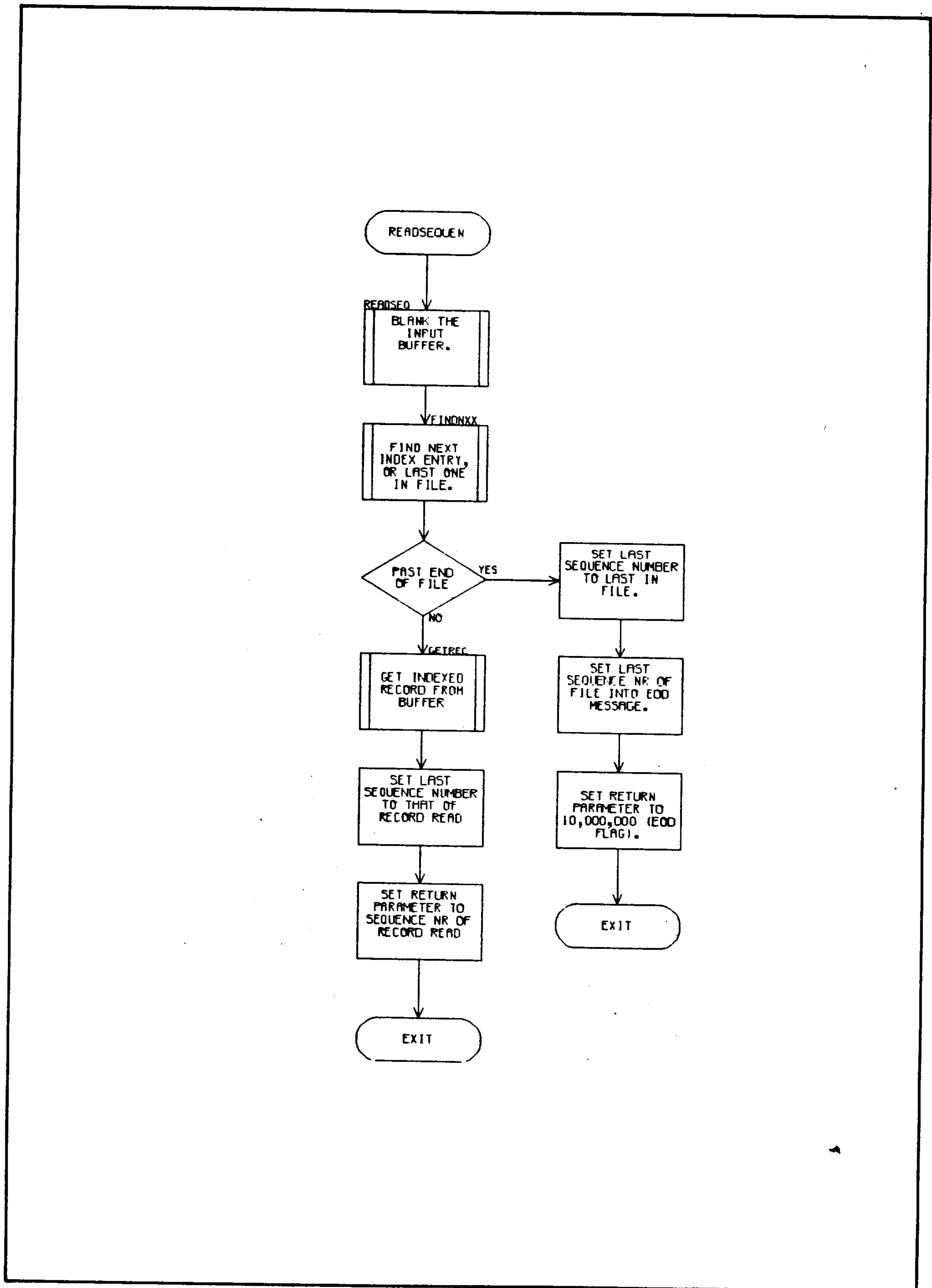



Figure 101. Flow Diagram of READSEQUEN

5. Stack:

   Seven.

6. Subroutines:

   OPENSCRI, WRITENEWRANDOM, WRITERANDOM

7. Operation:

   This routine reads the subject file sequentially, indexes the records read, and writes them (using WRITE-NEWRANDOM and WRITERANDOM) into the scratch file. The key value for each record is determined either by the trailing eight bytes of the record, or by a fixed increment applied to the prior key written. If the save file sequencing mode is on, a key in the last eight bytes of any record is removed.

   The flow of BUILDSCR is given in Figure 102.

INSEQNR

1. Purpose:

   Translates a line number in a subject file record.

2. Call:

   BAL, LNK INSEQNR

3. Input:

   None.

4. Output:

   If found:

   10 = 0.

   R1 = value times 1000.

   If not found:

   10 = 1.

5. Stack:

   Three.

6. Subroutines

   Internal only.

7. Operation:

   Starting with the eighth byte from the end of the record, the routine skips leading blanks, skips leading zeros, accumulates the integer part of a key up to four characters, and tests for a decimal point. If none is found,

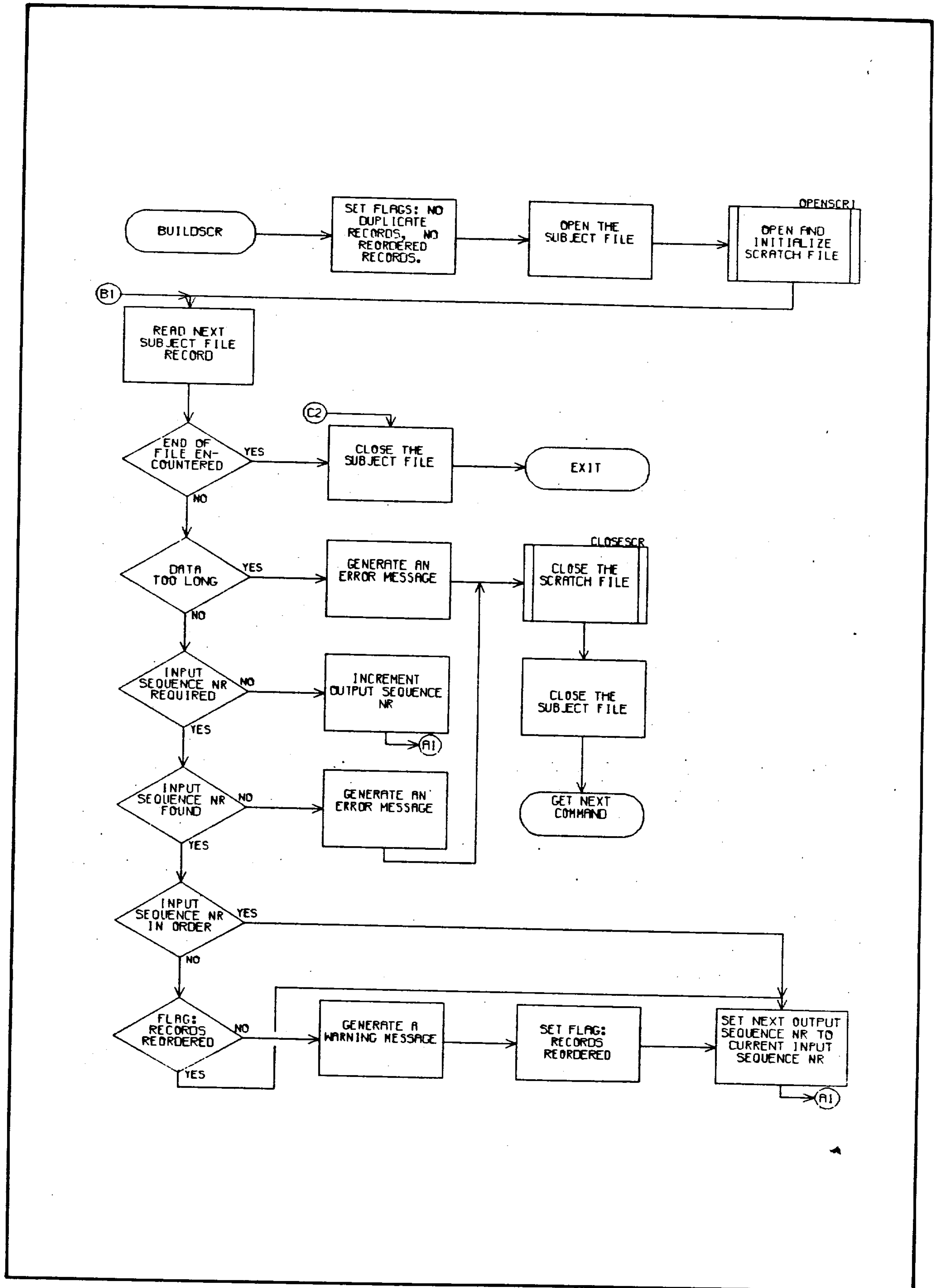



Figure 102. Flow Diagram of BUILDSCR

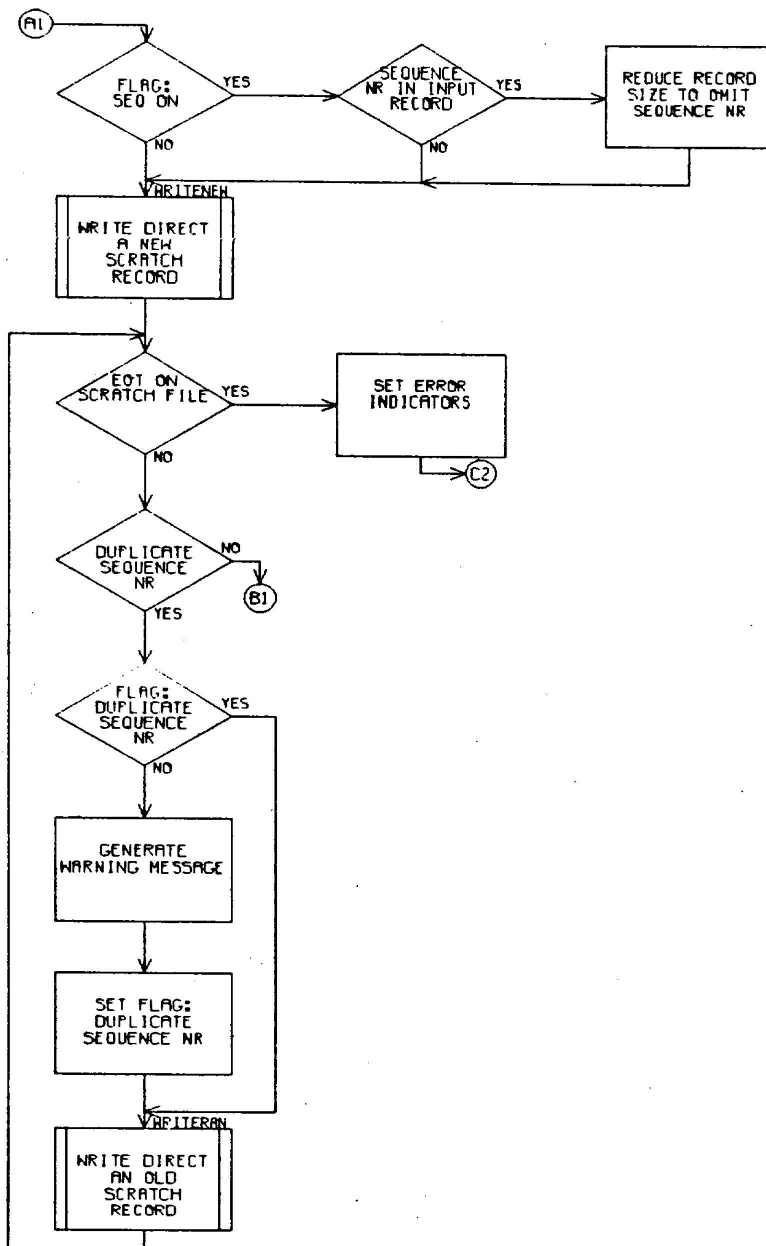



Figure 102. Flow Diagram of BUILDSCR (cont.)

the routine then skips blanks. If a decimal point is found, the routine accumulates the fractional part of the key up to three characters and then skips blanks. If no digits were encountered, or if the scan did not reach the end of the record, the routine reports that no sequence number was found.

SAVESCR

1. Purpose:

   Builds a standard CP-R file from the CP-R indexed scratch file.

2. Call:

   BAL, LNK SAVESCR.

3. Input:

   M:EI and M:EO assigned; M:EI open.

4. Output:

   None.

5. Stack:

   Ten.

6. Subroutines:

   READSEQUEN, READX, UPKENTRY

7. Operation:

   By counting the number of records in the scratch file and considering the structure of the save file, the routine determines how big the save file must be. Using this estimate, the routine can allot the save file or abort a save on an inadequate save file before it is altered. The save is conducted by reading the scratch file sequentially (using READSEQUEN) and writing to the save file until an end-of-file condition is reported for the scratch file. If the save file sequencing mode is on, each record will have its key inserted in its last eight bytes.

   The flow of SAVESCR is given in Figure 103.

GETEO

1. Purpose:

   Determines the nature of the EO file.

2. Call:

   BAL, LNK GETEO

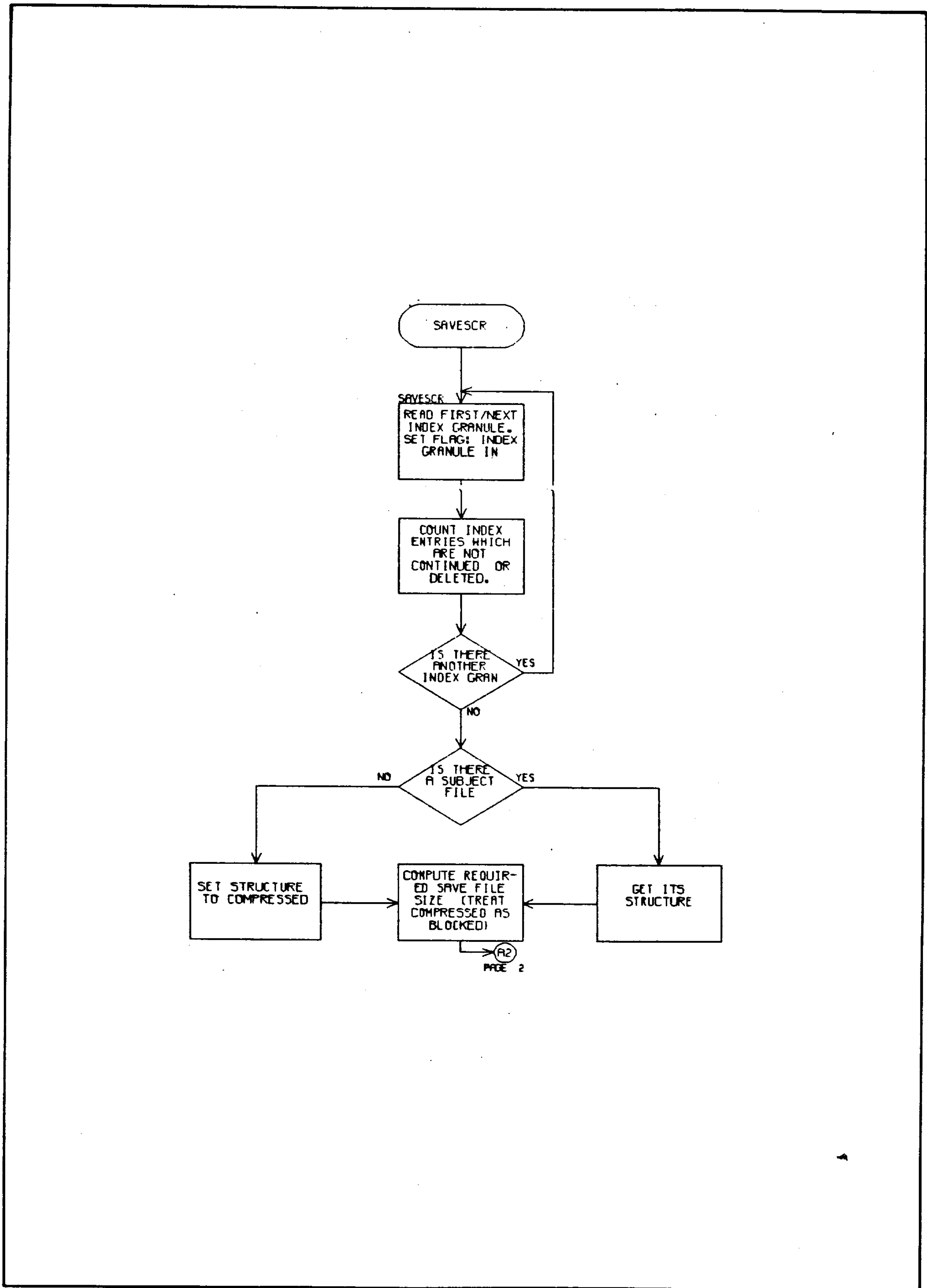



Figure 103. Flow Diagram of SAVESCR

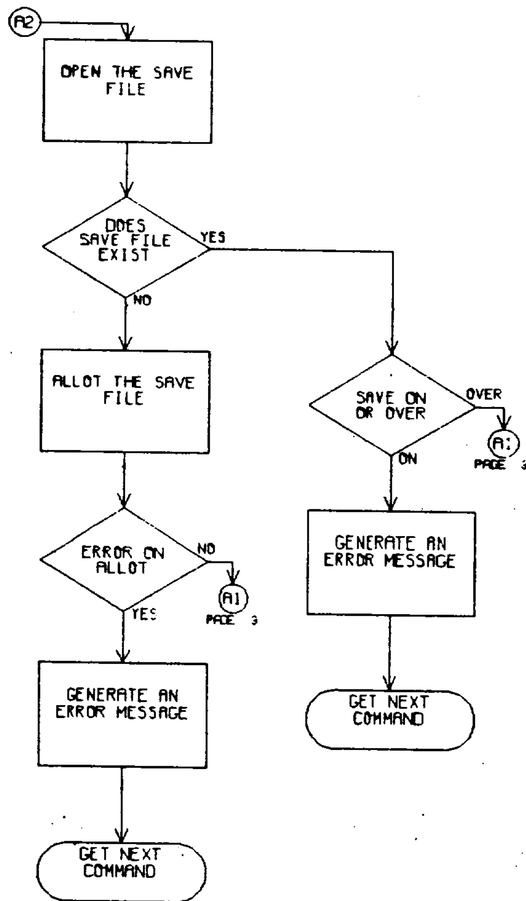



Figure 103. Flow Diagram of SAVESCR (cont.)

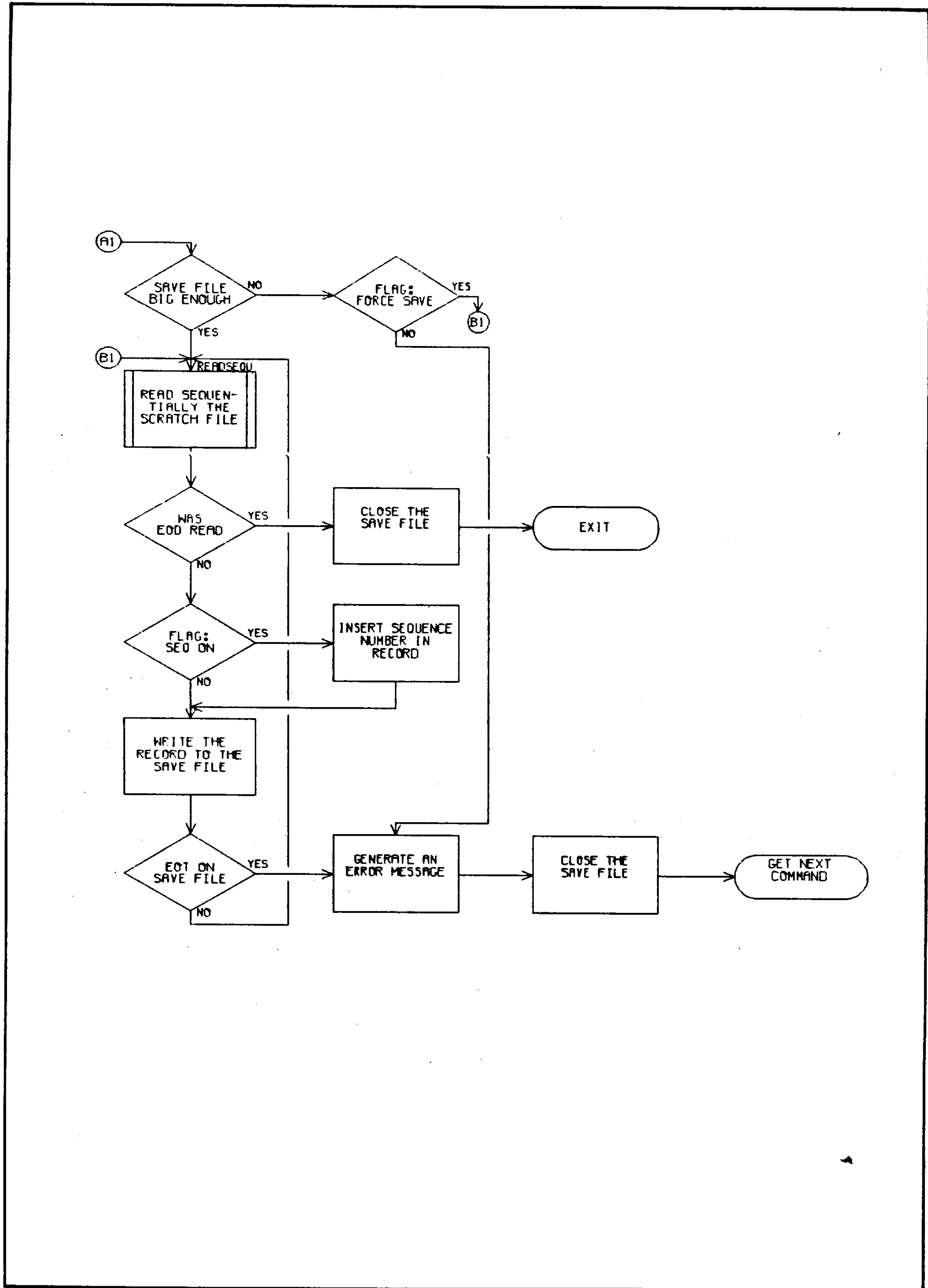



Figure 103. Flow Diagram of SAVESCR (cont.)

3. **Input:**

M:EO assigned to a file.

4. Output:

If file and area exist, R1 = File organization code; = 0 for unblocked; = 1 for blocked; = 2 for compressed.

R2 = record size if R1 = 1.

R3 = file size in sectors.

If file or area nonexistent, R3 = 0.

5. Stack:

One.

6. Subroutines:

None.

7. Operation:

The CP-R "GET DEVICE/FILE/OPLABEL INDEX" service is used to obtain the required information. If the device involved is a 720X, the number of sectors is divided by three to reflect an assumed 256-word granule size on a device with 90-word sectors.

# 15. SYSTEM GENERATION

## Overview

The System Generation program is assembled in absolute, using the ASECT directive, and is ORG'd (origined) at two locations:

1. The first ORG at location X'140' allocates and defines the system flags and pointers. It is the first location that cannot be used for an external interrupt. The system flags and pointers are a group of cells that provide communication between SYSGEN, all portions of the Monitor, and the system processors and service routines. Since these cells are in fixed, predetermined locations, they are defined via the EQU directive in all programs that reference them. Note that these cells must not be changed, deleted, or altered in any way in the SYSGEN listing unless the EQU directives are also changed in all programs that reference the cells. The system flags and pointers are followed by a skeleton of the Master Dictionary. The Master Dictionary is not necessarily fixed at its assembled location since it may be moved to the unused interrupt cells if sufficient space exists.
2. The next ORG (based on assembly parameters) fixes the start of the SYSGEN program. SYSGEN is ORG'd such that the program will occupy the highest address portion in memory. This provides the SYSGEN Loader with the maximum amount of room to load the Monitor and its overlays in the lower address portion of memory. If a user adds a significant amount of code to the Monitor, this ORG may have to be moved to a higher location to prevent the Monitor from overflowing SYSGEN during the load.

The System Generation program is divided into two sections designated as SYSGEN and SYSLOAD. SYSGEN processes all the SYSGEN control commands and allocates and initializes all the Monitor tables from the information on the control commands. It also builds a symbol table for SYSLOAD that contains the name and absolute address of all the Monitor tables. Optionally, SYSGEN will output on a rebootable deck containing the Monitor tables and SYSLOAD on cards, paper tape, or magnetic tape. The SYSGEN phase can be overwritten during the loading of the Monitor, and terminates by exiting to SYSLOAD.

SYSLOAD loads the Monitor, all optional resident routines, the CP-R overlays, the Job Control Processor, and then writes these in to the CP-R file in the SP area. A map containing the CP-R table allocation and disk allocation is output upon request. SYSLOAD terminates by reading in the RAD Bootstrap and exiting to it, simulating a booting of the system from the disk.

Figure 104 illustrates the core layout of SYSGEN and SYSLOAD after the absolute object module is loaded by the Stand-Alone SYSGEN Loader.

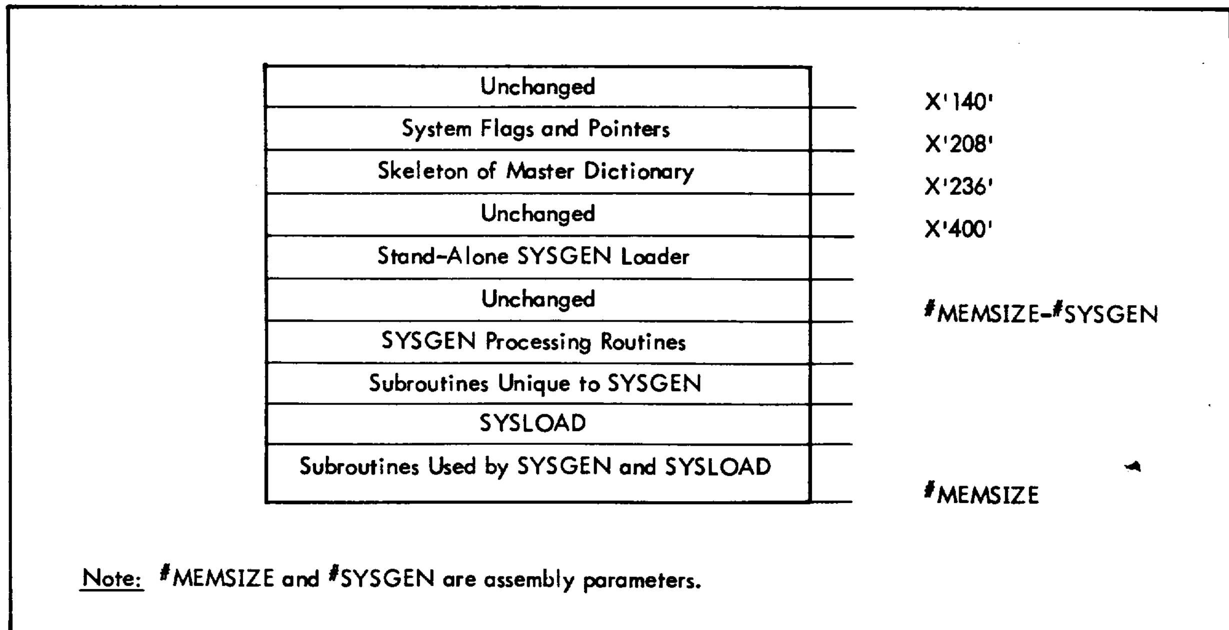



Figure 104. SYSGEN and SYSLOAD Layout before Execution

Figure 106 depicts a typical core layout after SYSGEN and SYSLOAD have executed.

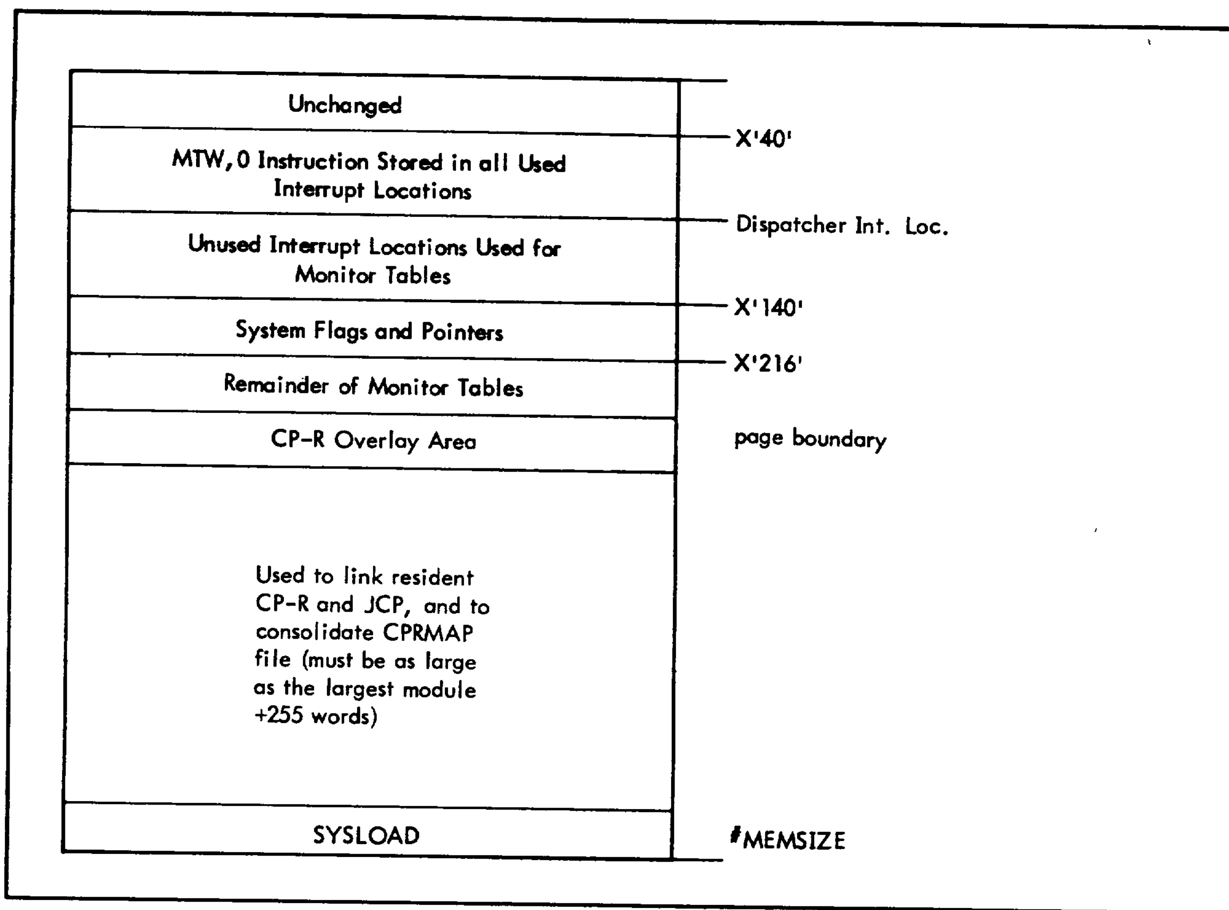



Figure 105. SYSGEN and SYSLOAD Layout after Execution

## SYSGEN/SYSLOAD Flow

The flowcharts in Figure 106 depict the overall flow of SYSGEN and SYSLOAD. The labels used correspond to the labels in the program listing.

## Loading Simulation Routines, CP-R, and CP-R Overlays

SYSGEN/SYSLOAD contains a loader that loads the instruction simulation packages, CP-R, the CP-R overlays, and the Job Control Processor (JCP). Each object module loaded must have one DEF directive that identifies the object module to the loader.† The DEFs listed in Table 10 are recognized by the Loader.

†This DEF must be the first load item in the ROM.

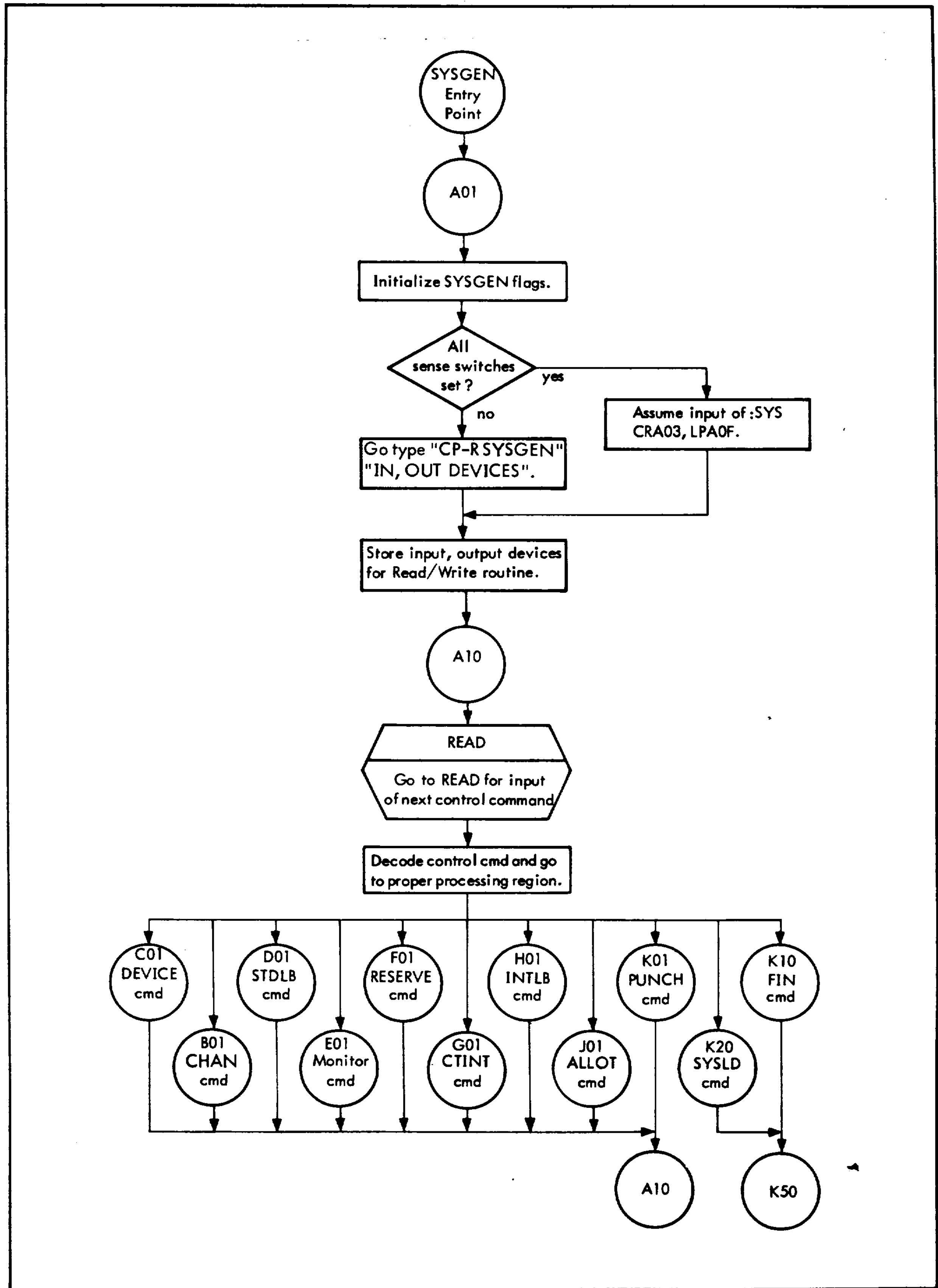



Figure 106. SYSGEN/SYSLOAD Flow

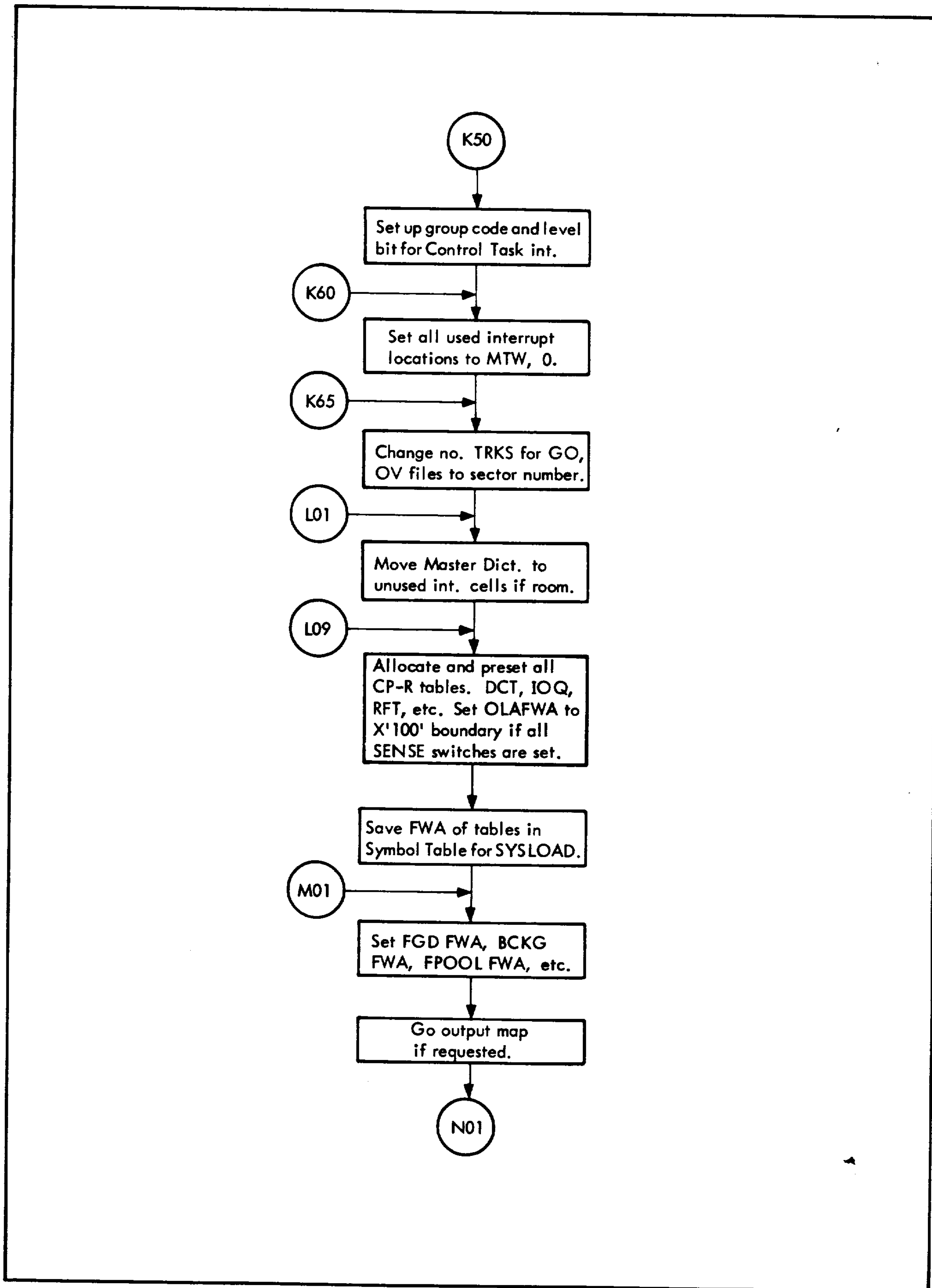



Figure 106. SYSGEN/SYSLOAD Flow (cont.)

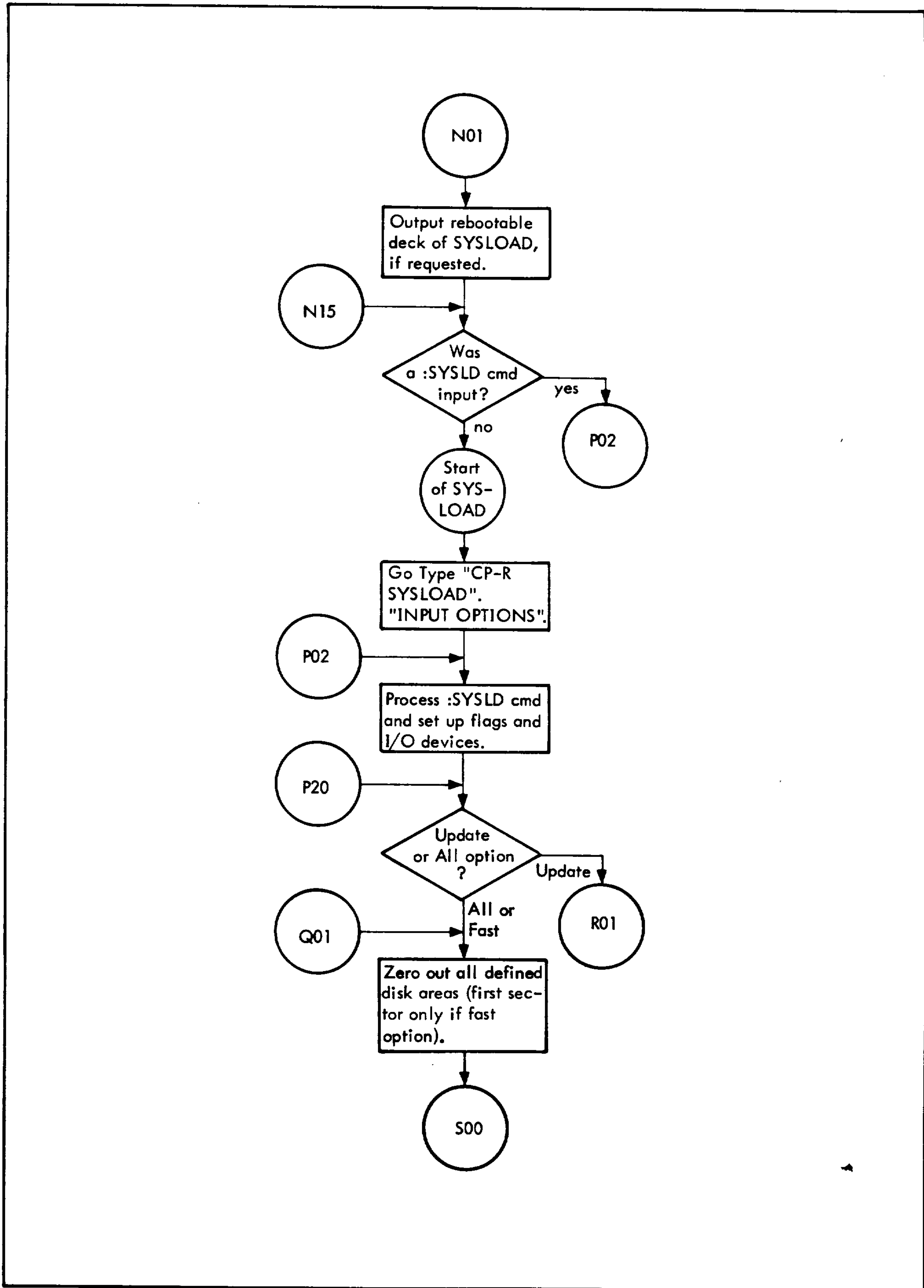



Figure 106. SYSGEN/SYSLOAD Flow (cont.)

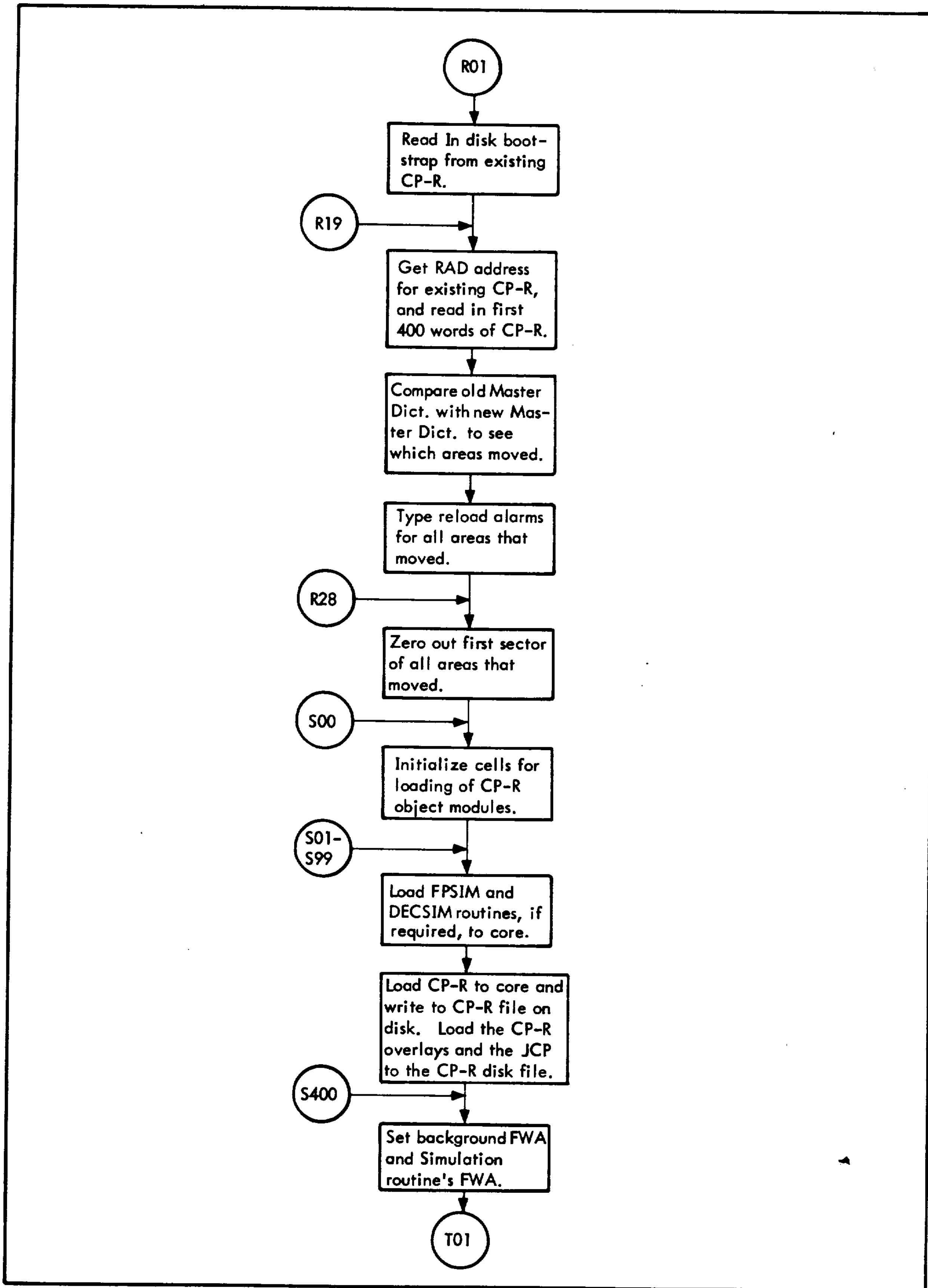



Figure 106. SYSGEN/SYSLOAD Flow (cont.)

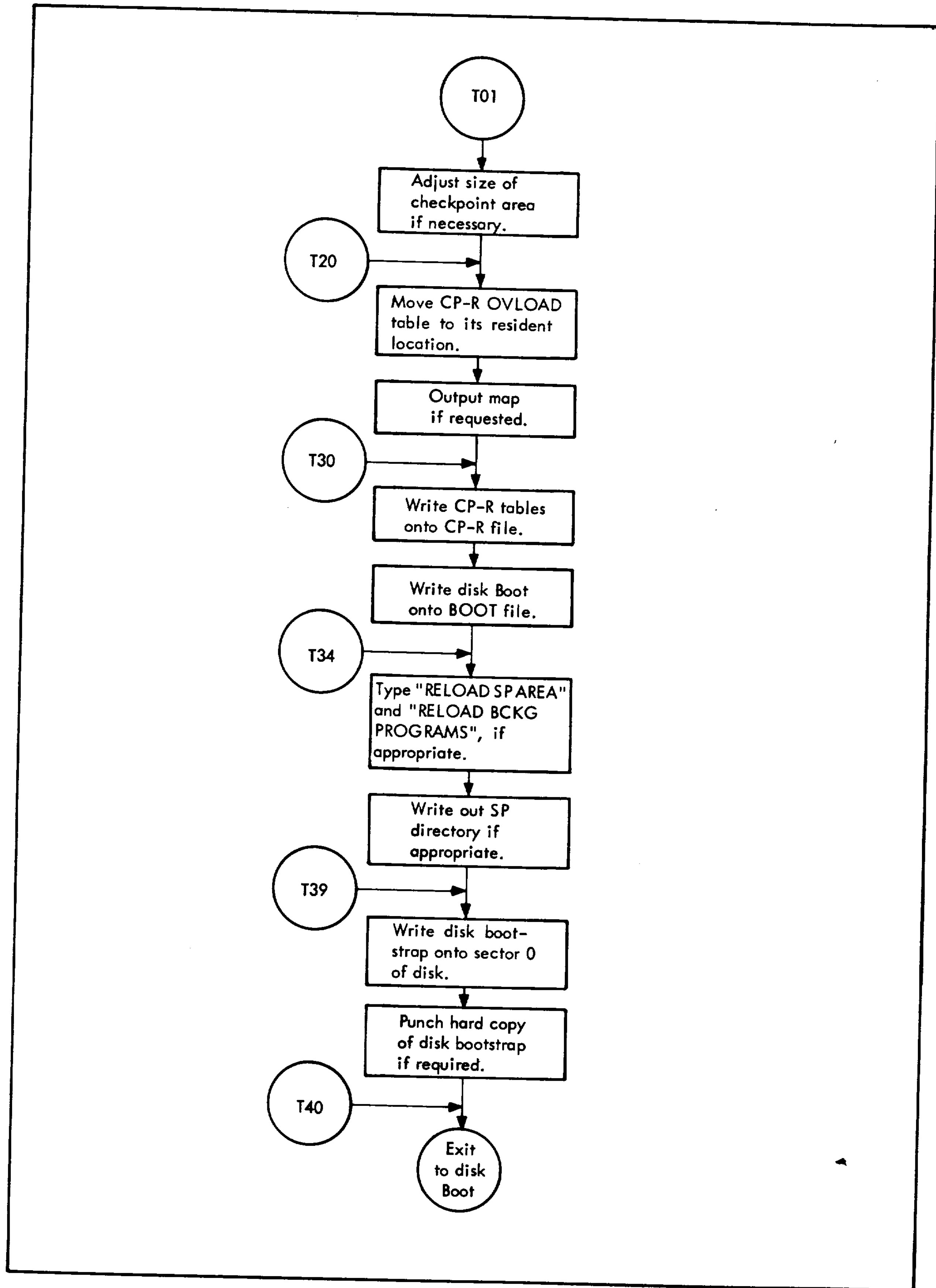



Figure 106. SYSGEN/SYSLOAD Flow (cont.)

Table 10. Standard System Modules

| DEF Name | Program |
|---|---|
| ABEX | Background Abort/Exit |
| ALLOT | ALLOT Service Calls |
| ARM | ARM/DISARM/CONNECT/DISCONNECT |
| BKL1 | Background Loader |
| CHECK | Check service calls |
| CKD | Crash dump to LP |
| CKD2 | Crash KDUMP to LP |
| CLOSEX | Close a DCB |
| COCIO | I/O routines for COC |
| CPR | Main CP-R module |
| CRD | Crash dump to BI |
| CRS | Crash SAVE |
| CRS2 | Crash SAVE |
| DBC1 | Debug functions |
| DBC2 | Debug functions |
| DBC3 | Debug functions |
| DBDW | Debug data and entries |
| DBS1 | Debug scan |
| DBS2 | Debug functions |
| DBS3 | Debug scan |
| DELETE | Service call |
| DEVI | Device service calls |
| DISC | Disk handlers |
| DUMP | Postmortem dump |
| ENQ | Enqueue/dequeue a resource |
| ESU | Error summary |
| EXTM | Termination service calls |
| FGL1 | Run-time Loader |
| FGL2 | Run-time Loader |

Table 10. Standard System Modules (cont.)

| DEF Name | Program |
| --- | --- |
| FGL3 | Run-time Loader |
| GETNRT | I/O subroutines |
| INIT | Boot-time initialization |
| IOEX | IOEX service calls |
| IPLMM | Memory Management Initialization |
| IPLSYM | SYMBIONT Initialization |
| JOB1 | Job service calls |
| JOB2 | Job service calls |
| KEYSCN | Command syntax scanners |
| KEY1 | Keyin processor |
| KEY2 | Keyin processor |
| KEY3 | Keyin processor |
| KEY4 | Keyin processor |
| KEY5 | Keyin processor |
| KEY6 | Keyin processor |
| KEY7 | Keyin processor |
| KEY8 | Keyin processor |
| LOG | Error Logger |
| LP | Line Printer Handlers |
| MEDIA | Media service calls |
| MED1 | Media service calls |
| MED2 | Media service calls |
| MMROOT | Memory Management data and subroutines |
| MMO1 | Memory Management service calls |
| MMO2 | Memory Management service calls |
| MMO3 | Memory Management subroutines |
| MMO4 | Memory Management exec |
| MMO5 | Memory Management subroutines |
| MMO6 | Memory Management service calls |
| OPENX | Open a DCB |
| PINIT | INIT service calls |
| PLO1 | Public libraries |

Table 10. Standard System Modules (cont.)

| DEF Name | Program |
|---|---|
| PRINT | Print service calls |
| READWR | Read/Write service calls |
| REWDEV | Rewind on devices |
| REWIND | Rewind service calls |
| RUN | Run service calls |
| RWBFIL | Blocked File I/O |
| RWDEV | Read/Write device I/O |
| RWFILE | Read/Write file I/O |
| SCHED | Periodic Scheduler |
| SCMSG | Periodic Scheduler Subroutines |
| SDBUF | Side buffering routines |
| SEX | Symbiont Exec |
| SIGNAL | Signal handler |
| SJOB | SJOB/KJOB service calls |
| SNAM | SETNAME service calls |
| STDLB | STDLB service calls |
| SYM1 | Symbiont routines |
| SYM2 | Symbiont routines |
| SYM3 | Symbiont routines |
| TAPE | Magnetic Tape handlers |
| TEL | Terminal Executive Language |
| TEL1 | TEL routines |
| TEL2 | TEL routines |
| TERM | Task Termination |
| TEX | Terminal Exec |
| TEX1 | Terminal Exec routines |
| TEX2 | Terminal Exec routines |
| TIO1 | Secondary Task Initiation |
| TIO2 | Secondary Task Initiation |
| TIO3 | Task Initiation Data and subroutines |
| TMGETP | Task/ECB subroutines |
| TMTYC | Task/ECB subroutines |
| TRAPS | Trap handling |
| TT | Task termination |
| WAIT | Wait service calls |

## Rebootable Deck Format

If a :PUNCH control command is read by SYSGEN, a rebootable deck is output that includes the CP-R tables with their initialized values, SYSLOAD, and the CP-R Symbol Table.† This deck can be used to load a new version of CP-R without re-inputting all the SYSGEN control commands.

The first card in the rebootable deck consists of a one-card bootstrap program that loads the next two cards in the deck. These next two cards consist of a program that loads the remainder of the deck, consisting essentially of the CP-R Table, SYSLOAD, and the CP-R Symbol Table in core image format.

The two cards containing the Core Image Loader have the following format:

| Byte No. | Contents |
|---|---|
| 0 | X'FF'(for card 1)  X'9F'(for card 2) |
| 1, 2, 3 | Unused (all zeros) |
| 4, 5, 6, 7 | Complement checksum of entire card (carry out of bit 0 is ignored in computing checksum) |
| 8, 9 | Unused (all zeros) |
| 10, 11 | Load address, minus one, for following data |
| 12-119 | Loader in absolute core image format |

The core image format of the Two-Card Loader is

| Word | Bits 0–7 | Bits 8–15 | Bits 16–31 |
|---|---|---|---|
| word 1 | X'FF' or X'9F' | | |
| word 2 | Complement checksum of entire 29 words on card | | |
| word 3 | | | Load address - 1 |
| word 4 | | | |
| (words 4-30 contain the Two-Card Loader in absolute core image format.) | ⋮ | | |
| word 30 | | | |

The CPR Tables, SYSLOAD, and the CPR Symbol Table are output in the core image format

| Word | Bits 0–7 | Bits 8–15 | Bits 16–31 |
|---|---|---|---|
| word 1 | X'FF' or X'9F' | | Sequence number (0-n) |
| word 2 | Load address - 1 | | Complement checksum (not incl. halfword 0) |
| word 3 | | | |
| (words 3-30 contain the above-mentioned data in core image format.) | ⋮ | | |
| word 30 | | | |

† If the rebootable deck is output to paper tape, there are no special additional characters. That is, the paper tape contains an exact card image.

All cards contain an X'FF' in byte 0 except the last card. The last card contains an X'9F' in byte 0 and the SYSLOAD entry address in place of the load address in word 1. The last card contains no data other that the SYSLOAD entry address, the sequence number, and checksum.

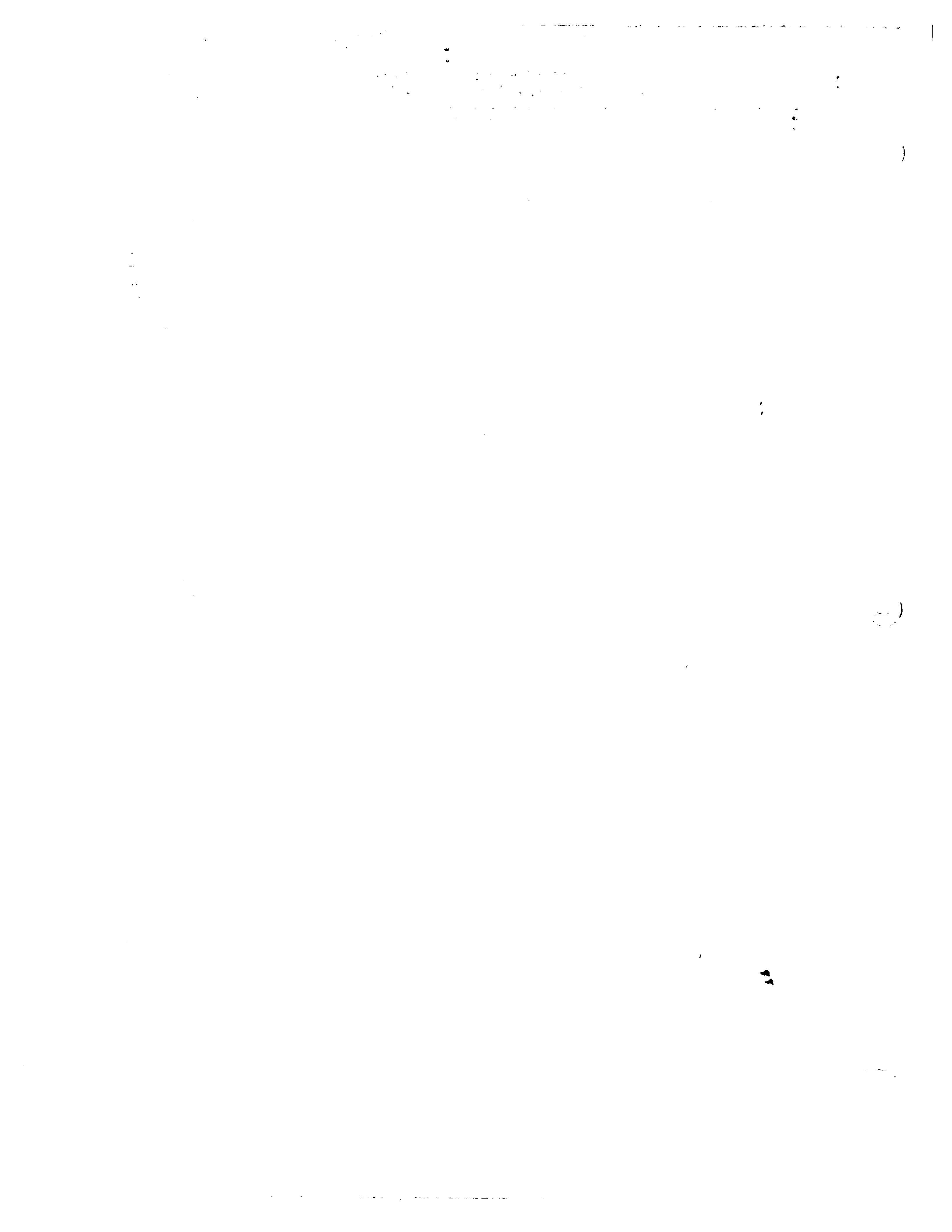
## Stand-Alone SYSGEN Loader

The Stand-Alone SYSGEN Loader is a small loader specifically created to load the SYSGEN absolute object module. Since SYSGEN is assembled in absolute, the SYSGEN Loader will only load absolute load items and handles only the small subset of the Sigma Object Language required to load SYSGEN.

The SYSGEN Loader I/O routine is similar to the SYSGEN I/O, with the code performing the actual loading being similar to the code in the SYSGEN Loader.

## SYSGEN LOADER LOADER

Each BI tape/deck is preceded with a 26-record bootstrap that loads the SYSGEN Loader into memory from the same device it was booted from.



# APPENDIX A. CP-P SYSTEM FLAGS AND POINTERS

Table A-1. CP-R System Flags and Pointers

| Name | Location | Description |
|---|---|---|
| K:SYSTEM | X'2B' | Monitor Identification (RBMIDENT) have the following meaning:<br>Bits 0-7 System-identification (X'80' = CPR).<br>Bits 8-11 Version (C=3, D=4, etc.).<br>Bits 12-15 Update (1, 2, 3, etc.).<br>Bits 16-23 Reserved.<br>Bits 24-25 00 - Sigma 5.<br>01 - Sigma 6/7.<br>10 - Sigma 9<br>11 - Xerox 550<br>Bit 26 Reserved.<br>Bit 27 Reserved.<br>Bit 28 Reserved.<br>Bit 29 Real-Time Routines.<br>Bit 30 Reserved.<br>Bit 31 Symbionts included. |
| K:BACKBG | X'140' | Beginning address of background. |
| K:BCKEND | X'141' | Ending address of SMM background. |
| K:FGDBG1 | X'142' | Beginning address of non-monitor real memory. |
| K:FGDEND | X'143' | Ending address of addressable real memory. |
| K:CCBUF | X'144' | Address of Control Card Buffer. |
| K:BPOOL | X'145' | Unused in mapped system. |
| K:FGDBG2 | X'146' | Unused in mapped system. |
| K:FMBOX | X'147' | Start address of FGD Mailboxes. |
| K:FPOOL | X'148' | Start address of FGD Blocking Buffer Pool. |
| K:UNAVBG | X'149' | Memory size + 1. |
| K:MASTD | X'14A' | Start address of MDFLAG table in Master Dictionary. |
| K:NUMDA | X'14B' | Highest valid index for Master Dictionary. |
| K:VRSION | X'14C' | CP-R version. |
| K:ACCNT | X'14D' | Job Accounting flag. |
| K:OV | X'14E' | Permanent and current sizes of OV. |
| K:KEYST | X'14F' | Post status of key-in read here. |
| K:JCP1 | X'150' | JCP and Control Task.<br>Bits have the following meaning:<br>Bit 0 = 1, JCP is executing.<br>Bit 1 = 1, Background is active.<br>Bit 2 = 1, Background is checkpointed on the disk.<br>Bit 3 = 1, Background is being used by Foreground but was not checkpointed.<br>Bit 4 = 1, Waiting for key-in response.<br>Bit 5 = 1, Skip to next JOB card.<br>Bit 6 = 1, Set by ABORT for CALEXIT.<br>Bit 7 = 1, Set by CALEXIT for ABORT. |

Table A-1. CP-R System Flags and Pointers (cont.)

| Name | Location | Description |
|---|---|---|
| K:JCP1 (cont.) | | Bits 8-15, Previous assign. of C device (for TY key-in).<br>Bits 16-21, Unused.<br>Bit 22= 1, System processor executing.<br>Bit 23= 1, Execute BKGD Debug.<br>Bits 24-25, 0 means no PMD requested.<br>1 means conditional PMD.<br>2 means unconditional PMD.<br>Bit 26, Flag for CKPT that alarm typed.<br>Bit 27= 1, CP-R Initialize routine is running.<br>Bit 28= 1, FG key-in active.<br>Bit 29= 1, TY key-in active.<br>Bit 30= 1, Attend command was input.<br>Bit 31= 1, JOB command was input. |
| K:CTST | X'151' | Flags to execute Control Task subtask. Bits have the following meaning:<br><br>Bit 0= 1, Execute CHECKPOINT.<br>Bit 1= 1, Execute FGD Loader/Releaser.<br>Bit 2= 1, Execute Restart.<br>Bit 3= 1, Time to service all devices.<br>Bit 4= 1, Execute ABORT/EXIT.<br>Bit 5= 1, Execute key-in.<br>Bit 6= 1, Execute PMD.<br>Bit 7= 1, BCKG is IDLE.<br>Bit 8= 1, Execute BCKG load.<br>Bit 9= 1, Load JCP.<br>Bit 10= 1, Load BCKG (Program not JCP).<br>Bit 11= 1, Key-in required by higher priority subtask.<br>Bit 12= 1, Recycle FGL1/2 to FGL1 for possible RLS.<br>Bit 13= 1, Execute error logger.<br>Bit 14= 1, CKPT deferred during BCKG abort.<br>Bit 15= 1, BCKG in wait following attended mode abort.<br>Bit 26= 1, KEY2 doing STDLB RAD file OPEN/CLOSE.<br>Bit 27= 1, FGL1 called from FGL2.<br>Bit 28= 1, Control Task is operating.<br>Bit 29= 0, Execute ABORT part of ABORT/EXIT.<br>Bit 29= 1, Execute EXIT part of ABORT/EXIT.<br>Bit 30= 1, PMD from key-in request.<br>Bit 31= 1, PMD from PMD command. |
| K:SY | X'152' | Nonzero if SY key-in active. |
| K:BPEND | X'153' | End of load area for SMM BCKG program. |
| K:CTWD | X'154' | WD code for Control Task. Byte 0 nonzero means CT was triggered. |
| K:CTGL | X'155' | Group level for Control Task. |
| K:BLOAD | X'156' | Name in BCD of BCK program to load (two words). |
| K:BAREA | X'158' | Index of area to load BCK program from. |
| K:ASSIGN | X'159' | Address of ASSIGN table. |
| K:RUNF | X'15A' | Post run status here for FGD !RUN or !ROV command. |
| K:HIINT | X'15B' | HW0 = Control task interrupt number.<br>HW1 = Highest address used for interrupt. |

Table A-1. CP-R System Flags and Pointers (cont.)

| Name | Location | Description |
|---|---|---|
| K:FGDBG3 | X'15C' | Unused in mapped system. |
| K:PMD | X'15D' | Cells to dump for PMD as DW address (5 words). |
| K:DCB | X'162' | DCB for Control Task to load in overlays (7 words). Always assigned to RBM File. |
| K:KEYIN | X'169' | Key-in control words. |
| K:FGDBG4 | X'16F' | Unused in mapped system. |
| K:DELTA | X'170' | Entry point for Delta. |
| K:QUEUE | X'171' | Address of Queue routine. Byte 0 = Nonzero, Stop I/O on BCKG. |
| K:BTFILE | X'172' | Status of BT Files<br>Bits 0 - 8, 1 bit for each X1 file. Bit set to 1 means SAVE file.<br>Bits 16 - 31, LWA to use for non-SAVE files. |
| K:GO | X'173' | Permanent and current sizes of GO. |
| K:PAGE | X'174' | Byte 0 = Number of lines per page. |
| K:RDBOOT | X'175' | FWA and device Number of RADBOOT. |
| K:DCT1 | X'176' | Addresses of tables. |
| K:DCT16 | X'177' | |
| K:OPLBS1 | X'178' | |
| K:OPLBS3 | X'179' | |
| K:RFT4 | X'17A' | |
| K:RFT5 | X'17B' | |
| K:SERDEV | X'17C' | Address of SERDEV. |
| K:REQCOM | X'17D' | Address of REQCOM. |
| K:INITX | X'17E' | Address to return to after INIT runs. |
| K:FGLD | X'17F' | Byte 0 = Nonzero, XEQ FGD Load/RLS. |
| K:PMD1 | X'180' | Flags for dumps. |
| K:CTDR7 | X'181' | Location to save context pointer during Control Task dump. |
| K:DBTS | X'182' | Context pointer for background PMD. |
| K:KEYDCB | X'183'-X'187' | DCB to read operator key-ins. |
| K:CLK1 | X'188' | Clock cells must start on a DW boundary: there are counters for 4 clocks – 2 words/clock.† |
| K:CLK2 | X'18A' | Word 2 gets stored into word 1 when Counter = 0. |
| K:CLK3 | X'18C' | |

†The user never needs to access Clock 4.

Table A-1. CP-R System Flags and Pointers (cont.)

| Name | Location | Description |
|---|---|---|
| K:ABTLOC | X'18E' | Abort location. |
| K:MSG1 | X'190' | KEY-IN. |
| K:MSG2 | X'193' | KEY ERR. |
| K:MSG3 | X'196' | RLS NAME NA. |
| K:MSG4 | X'19A' | FILE NAME ERR. |
| K:MSG5 | X'19E' | FGD AREA ACTIVE. |
| K:MSG6 | X'1A3' | NOT ENUF BCKG SPACE. |
| K:MSG7 | X'1A9' | UNABLE TO DO ASSIGN. |
| K:MSG8 | X'1AF' | BCKG CKPT. |
| K:MSG9 | X'1B2' | BCKG IN USE BY FGD. |
| K:MSG10 | X'1B7' | BCKG RESTART. |
| K:MSG11 | X'1BB' | CK AREA TOO SMALL. |
| K:MSG12 | X'1C0' | I/O ERR ON CKPT. |
| K:MSG13 | X'1C5' | JOB ABORTED AT xxxxx. |
| K:MSG14 | X'1CB' | LOADED PROG NAME. |
| K:MSG15 | X'1CF' | UNABLE TO LOAD BCKG PUB LIB. |
| K:MSG16 | X'1D7' | CKPT WAITING FOR BCKG I/O RUNDOWN. |
| K:XITSIM | X'1E6' | Unimplemented instruction normal return. |
| K:TRPSIM | X'1E7' | Unimplemented instruction trap return. |
| K:PPGMOT | X'1E8' | Unimplemented instruction memory-protection error return. |
| K:MONTH | X'1EA' | Table of days/month and BCD names. |
| K:DATE1 | X'1F6' | Number days in current year; current year - 1900. |
| K:DATE2 | X'1F7' | Day of year. |
| K:TIME | X'1F8' | Time of day in seconds. |
| K:ELTIM1 | X'1F9' | FGD saves BCKG elapsed time here. |
| K:LIMIT | X'1FA' | Maximum execution time for BCKG. |
| K:ACCNAM | X'1FB' | Account entry for AL file (8 words). |
| K:ELTIM2 | X'202' | Last word of account entry (elapsed time). |
| K:PTCH | X'207' | Beginning address of patch area. |
| K:PTCHND | X'208' | Ending address of patch area. |
| K:IOWD | X'209' | I/O trigger values. |
| K:IOGL | X'20A' | |
| K:CPWD | X'20B' | CP trigger values. |
| K:CPGL | X'20C' | |
| K:IOLOCK | X'20D' | |
| K:RMPT | X'20E' | RMPT location and length. |
| K:BMEM | X'20F' | Maximum number of BCKG pages. |
| K:JAET | X'210' | Number of allocatable DCT entries. |
| K:RTS | X'211' | CP-R stack pointer. |

Table A-1. CP-R System Flags and Pointers (cont.)

| Name | Location | Description |
|---|---|---|
| K:MDNAME | X'212' | Byte 0: Number of Master Dictionary entries.<br>Bytes 1-3: Address of MDNAME table. |
| K:DCT1X | X'213' | Address of DCT1 table. |
| K:RBMEND | X'214' | LWA of resident CP-R. |
| K:RUNJ | X'215' | Status from JCP run CAL. |
| K:DEBUG | X'216' | Debug communication LOC. |
| K:FSMM | X'217' | Pages, end address for foreground SMM. |
| K:MDBOA | X'218' | Address of MDBOA table. |
| K:MDEOA | X'219' | Address of MDEOA table. |
| K:MDDCTI | X'21A' | Address of MDDCTI table. |

# APPENDIX B. XEROX STANDARD OBJECT LANGUAGE

## INTRODUCTION

### GENERAL

The Xerox standard object language provides a means of expressing the output of any language processor in standard format. All programs and subprograms in this object format can be loaded by the Monitor's relocating loader.† Such a loader is capable of providing the program linkages needed to form an executable program in core storage. The object language is designed to be both computer-independent and medium-independent; i.e., it is applicable to any Xerox computer having a 32-bit word length, and the same format is used for any output medium.

### SOURCE CODE TRANSLATION

Before a program can be executed by the computer, it must be translated from symbolic form to binary data words and machine instructions. The primary stages of source program translation are accomplished by a processor. However, under certain circumstances, the processor may not be able to translate the entire source program directly into machine language form.

If a source program contains symbolic forward references, a single-pass processor such as the Xerox Symbol assembler can not resolve such references into machine language. This is because the machine language value for the referenced symbol is not established by a one-pass processor until after the statement containing the forward reference has been processed.

A three-pass processor, such as the Xerox Assembly Program (AP), is capable of making "retroactive" changes in the object program before the object code is output. Therefore, a two-pass processor does not have to output any special object codes for forward references. An example of a forward reference in a Symbol source program is given below.

```
        :
Y       EQU     $ + 3
        :
        CI, 5   Z
        :
        LI, R   Z
        :
Z       EQU     2
        :
        BG      Z
        :
R       EQU     Z + 1
        :
```

†Although a discussion of the object language is not directly pertinent to the CP-R, it is included in this manual because it applies to all processors operating under CP-R.

In this example the operand $ + 3 is not a forward reference because the assembler can evaluate it when processing the source statement in which it appears. However, the operand Z in the statement

```
CI,5    Z
```

is a forward reference because it appears before Z has been defined. In processing the statement, the assembler outputs the machine-language code for CI,5, assigns a forward reference number (e.g., 12) to the symbol Z, and outputs that forward reference number. The forward reference number and the symbol Z are also retained in the assembler's symbol table.

When the assembler processes the source statement

```
LI,R    Z
```

it outputs the machine-language code for LI, assigns a forward reference number (e.g., 18) to the symbol R, outputs that number, and again outputs forward reference number 12 for symbol Z.

On processing the source statement

```
Z    EQU    2
```

the assembler again outputs symbol Z's forward reference number and also outputs the value, which defines symbol Z, so that the relocating loader will be able to satisfy references to Z in statements CI,5 Z and LI,R Z. At this time, symbol Z's forward reference number (i.e., 12) may be deleted from the assembler's symbol table and the defined value of Z equated with the symbol Z (in the symbol table). Then, subsequent references to Z, as in source statement

```
BG    Z
```

would not constitute forward references, since the assembler could resolve them immediately by consulting its symbol table.

If a program contains symbolic references to externally defined symbols in one or more separately processed subprograms or library routines, the processor will be unable to generate the necessary program linkages.

An example of an external reference in a Symbol source program is shown below.

```
REF     ALPH
:
LI,3    ALPH
:
```

When the assembler processes the source statement

```
REF     ALPH
```

it outputs the symbol ALPH, in symbolic (EBCDIC) form, in a declaration specifying that the symbol is an external reference. At this time, the assembler also assigns a declaration name number to the symbol ALPH but does not output the number. The symbol and name number are retained in the assembler's symbol table.

After a symbol has been declared an external reference, it may appear any number of times in the symbolic subprogram in which it was declared. Thus, the use of the symbol ALPH in the source statement

```
LI,3    ALPH
```

in the above example, is valid even though ALPH is not defined in the subprogram in which it is referenced.

The relocating loader is able to generate interprogram linkages for any symbol that is declared an external definition in the subprogram in which that symbol is defined. Shown below is an example of an external definition in a Symbol source program.

```
              DEF     ALPH
               :
              LI,3    ALPH
               :
      ALPH    AI,4    X'F2'
               :
```

When the assembler processes the source statement

```
DEF    ALPH
```

it outputs the symbol ALPH, in symbolic (EBCDIC) form, in a declaration specifying that the symbol is an external definition. At this time, the assembler also assigns a declaration name number to the symbol ALPH but does not output the number. The symbol and name number are retained in the assembler's symbol table.

After a symbol has been declared an external definition it may be used (in the subprogram in which it was declared) in the same way as any other symbol. Thus, if ALPH is used as a forward reference, as in the source statement

```
LI,3      ALPH
```

above, the assembler assigns a forward reference number to ALPH, in addition to the declaration name number assigned previously. (A symbol may be both a forward reference and an external definition.)

On processing the source statement

```
ALPH     AI,4     X'F2'
```

the assembler outputs the declaration name number of the label ALPH (and an expression for its value) and also outputs the machine-language code for AI,4 and the constant X'F2'.

## OBJECT LANGUAGE FORMAT

An object language program generated by a processor is output as a string of bytes representing "load items". A load item consists of an item type code followed by the specific load information pertaining to that item. (The detailed format of each type of load item is given later in this appendix.) The individual load items require varying numbers of bytes for their representation, depending on the type and specific content of each item. A group of 108 bytes, or fewer, comprises a logical record. A load item may be continued from one logical record to the next.

The ordered set of logical records that a processor generates for a program or subprogram is termed an "object module". The end of an object module is indicated by a module-end type code followed by the error severity level assigned to the module by the processor.

## RECORD CONTROL INFORMATION

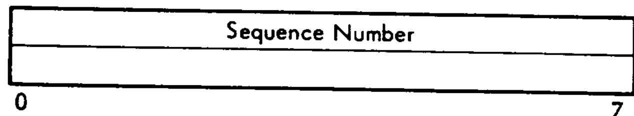
Each record of an object module consists of 4 bytes of control information followed by a maximum of 104 bytes of load information. That is, each record, with the possible exception of the end record, normally consists of 108 bytes of information (i.e., 72 card columns).

The 4 bytes of control information for each record have the form and sequence shown below.

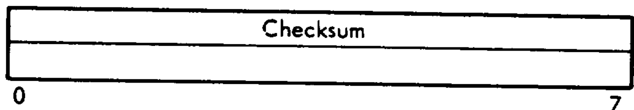
Byte 0

| Record Type | | | Mode | | Format | | |
|---|---|---|---|---|---|---|---|
| | | | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

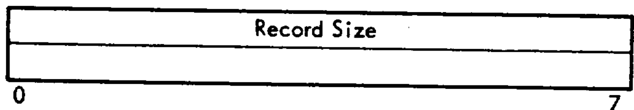
Byte 1




Byte 2




Byte 3




Record Type   specifies whether this record is the last record of the module:

000 means last
001 means not last

Mode   specifies that the loader is to read binary information. This code is always 11.

Format   specifies object language format. This code is always 100.

Sequence Number   is 0 for the first record of the module and is incremented by 1 for each record thereafter, until it recycles to 0 after reaching 255.

Checksum   is the computed sum of the bytes comprising the record. Carries out of the most significant bit position of the sum are ignored.

Record Size   is the number of bytes (including the record control bytes) comprising the logical record (5 ≤ record

size ≤ 108). The record size will normally be 108 bytes for all records except the last one, which may be fewer. Any excess bytes in a physical record are ignored.

### LOAD ITEMS

Each load item begins with a control byte that indicates the item type. In some instances, certain parameters are also provided in the load item control byte. In the following discussion, load items are categorized according to their function:

1. Declarations identify to the loader the external and control section labels that are to be defined in the object module being loaded.
2. Definitions define the value of forward references, external definitions, the origin of the subprogram being loaded, and the starting address (e.g., as provided in the AP END directive).
3. Expression evaluation load items within a definition provide the values (such as constants, forward references, etc.) that are to be combined to form the final value of the definition.
4. Loading items cause specified information to be stored into core memory.
5. Miscellaneous items comprise padding bytes and the module-end indicator.

## DECLARATIONS

In order for the loader to provide the linkage between subprograms, the processor must generate for each external reference or definition a load item, referred to as a "declaration", containing the EBCDIC code representation of the symbol and the information that the symbol is either an external reference or a definition (thus, the loader will have access to the actual symbolic name).

Forward references are always internal references within an object module. (External references are never considered forward references.) The processor does not generate a declaration for a forward reference as it does for externals; however, it does assign name numbers to the symbols referenced.

Declaration name numbers (for control sections and external labels) and forward reference name numbers apply only within the object module in which they are assigned. They have no significance in establishing interprogram linkages, since external references and definitions are correlated by matching symbolic names. Hence, name numbers used in any expressions in a given object module always refer to symbols that have been declared within that module.

The processor must generate a declaration for each symbol that identifies a program section. Although the Xerox Symbol assembler used with the Monitor allows only a standard control section (i.e., program section), the standard object language includes provision for other types of control sections (such as dummy control sections). Each object module produced by the Symbol processor is considered to consist of at least one control section. If no section is explicitly identified in a Symbol source program, the assembler assumes it to be a standard control section (discussed below). The standard control section is always assigned a declaration name number of 0. All other control sections (i.e., produced by a processor capable of declaring other control sections) are assigned declaration name numbers (1, 2, 3, etc.) in the order of their appearance in the source program.

In the load items discussed below, the access code, pp, designates the memory protection class that is to be associated with the control section. The meaning of this code is given below.

| pp | Memory Protection Feature[†] |
|---|---|
| 00 | Read, write, or access instructions from. |
| 01 | Read or access instructions from. |
| 10 | Read only. |
| 11 | No access. |

Control sections are always allocated on a doubleword boundary. The size specification designates the number of bytes to be allocated for the section.

Declare Standard Control Section

Byte 0

| Control byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Byte 1

| Access code | | | | Size (bits 1 through 4) | | | |
|---|---|---|---|---|---|---|---|
| p | p | 0 | 0 | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Byte 2

| Size (bits 5 through 12) | |
|---|---|
| | |
| 0 | 7 |

Byte 3

| Size (bits 13 through 20) | |
|---|---|
| | |
| 0 | 7 |

This item declares the standard control section for the object module. There may be no more than one standard control section in each object module. The origin of the standard control section is effectively defined when the first reference to the standard control section occurs, although the declaration item might not occur until much later in the object module.

---

[†]"Read" means a program can obtain information from the protected area; "write" means a program can store information into a protected area; and, "access" means the computer can execute instructions stored in the protected area.

This capability is required by one-pass processors, since the size of a section cannot be determined until all of the load information for that section has been generated by the processor.

Declare Nonstandard Control Section

Byte 0

| Control byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Byte 1

| Access code | | | | Size (bits 1 through 4) | | | |
|---|---|---|---|---|---|---|---|
| P | P | 0 | 0 | | | | |
| 0 | 1 | 2 | 3 | 4 | | | 7 |

Byte 2

| Size (bits 5 through 12) |
|---|
| 0 ... 7 |

Byte 3

| Size (bits 13 through 20) |
|---|
| 0 ... 7 |

This item declares a control section other than standard control section (see above). The loader is capable of loading object modules (produced by other processors, such as assemblers or compilers) that do contain this item.

Declare Page-Bounded Control Section

Byte 0

| Control Byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Byte 1

| Access code | | | | Size (bits 1 through 4) | | | |
|---|---|---|---|---|---|---|---|
| P | P | 0 | 0 | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Byte 2

| Size (bits 5 through 12) |
|---|
| 0 ... 7 |

Byte 3

| Size (bits 13 through 20) |
|---|
| 0 ... 7 |

This item declares a nonstandard control section beginning on a memory page boundary.

Declare Dummy Section

Byte 0

| Control byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Byte 1

| First byte of name number |
|---|
| 0 ... 7 |

Byte 2

| Second byte of name number† |
|---|
| 0 ... 7 |

Byte 3

| Access code | | | | Size (bits 1 through 4) | | | |
|---|---|---|---|---|---|---|---|
| P | P | 0 | 0 | | | | |
| 0 | 1 | 2 | 3 | 4 | | | 7 |

Byte 4

| Size (bits 5 through 12) |
|---|
| 0 ... 7 |

Byte 5

| Size (bits 13 through 20) |
|---|
| 0 ... 7 |

This item comprises a declaration for a dummy control section. It results in the allocation of the specified dummy section, if that section has not been allocated previously by another object module. The label that is to be associated with the first location of the allocated section must be a previously declared external definition name. (Even though the source program may not be required to explicitly designate the label as an external definition, the processor must generate an external definition name declaration for that label prior to generating this load item.)

Declare External Definition Name

Byte 0

| Control byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

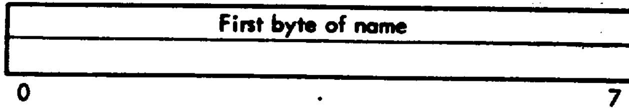
Byte 1

| Name length, in bytes (K) |
|---|
| 0 ... 7 |

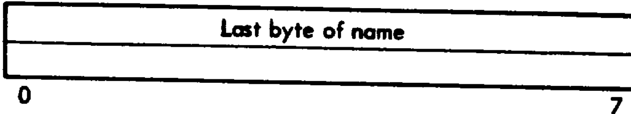
---

†If the module has fewer than 256 previously assigned name numbers, this byte is absent.

Byte 2



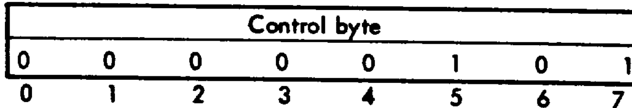

Byte K+1



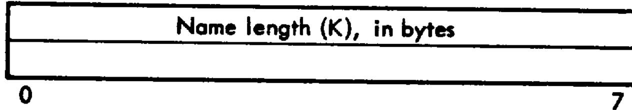

This item declares a label (in EBCDIC code) that is an external definition within the current object module. The name may not exceed 63 bytes in length.

Declare Primary External Reference Name

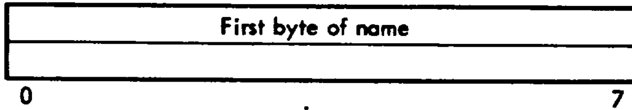
Byte 0



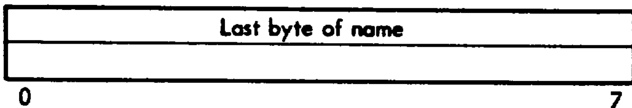

Byte 1




Byte 2




Byte K+1

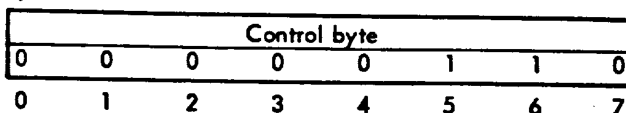



This item declares a symbol (in EBCDIC code) that is a primary external reference within the current object module. The name may not exceed 63 bytes in length.

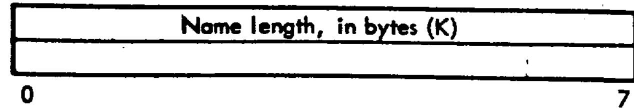
A primary external reference is capable of causing the loader to search the system library for a corresponding external definition. If a corresponding external definition is not found in another load module of the program or in the system library, a load error message is output and the job is errored.

Declare Secondary External Reference Name

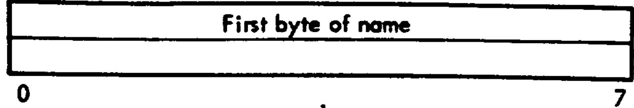
Byte 0



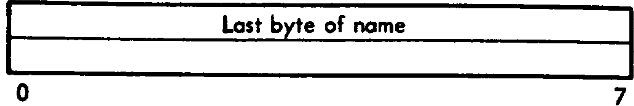

Byte 1




Byte 2




Byte K+1




This item declares a symbol (in EBCDIC code) that is a primary external reference within the current object module. The name may not exceed 63 bytes in length.

A secondary external reference is not capable of causing the loader to search the system library for a corresponding external definition. If a corresponding external definition is not found in another load module of the program, the job is not errored and no error or abnormal message is output.

Secondary external references often appear in library routines that contain optional or alternative subroutines, some of which may not be required by the user's program. By the use of primary external references in the user's program, the user can specify that only those subroutines that are actually required by the current job are to be loaded. Although secondary external references do not cause loading from the library, they do cause linkages to be made between routines that are loaded.

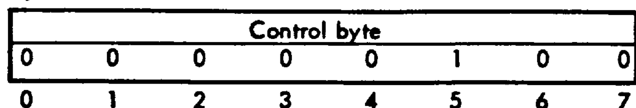
## DEFINITIONS

When a source language symbol is to be defined (i.e., equated with a value), the processor provides for such a value by generating an object language expression to be evaluated by the loader. Expressions are of variable length, and terminate with an expression-end control byte (see Section 4 of this appendix). An expression is evaluated by the addition or subtraction of values specified by the expression.

Since the loader must derive values for the origin and starting address of a program, these also require definition.

Origin

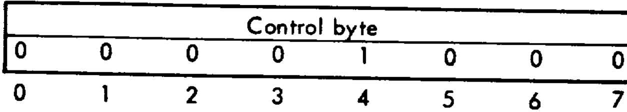
Byte 0



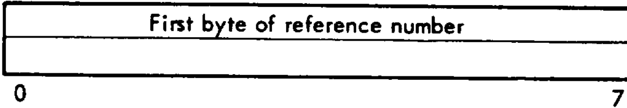
This item sets the loader's load-location counter to the value designated by the expression immediately following the origin control byte. This expression must not contain any elements that cannot be evaluated by the loader (see Expression Evaluation which follows).

Forward Reference Definition

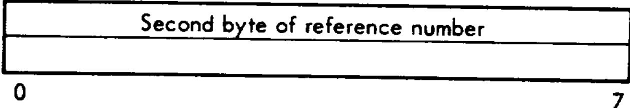
Byte 0




Byte 1



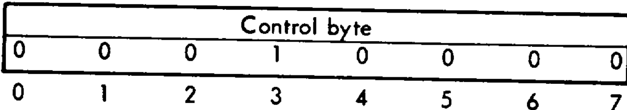

Byte 2



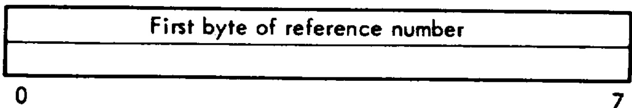

This item defines the value (expression) for a forward reference. The referenced expression is the one immediately following byte 2 of this load item, and must not contain any elements that cannot be evaluated by the loader (see Expression Evaluation which follows).

Forward Reference Definition and Hold

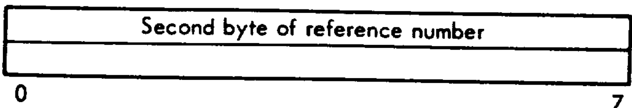
Byte 0




Byte 1




Byte 2

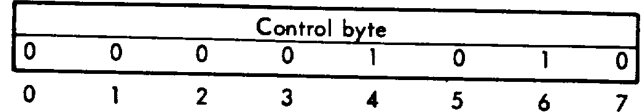



This item defines the value (expression) for a forward reference and notifies the loader that this value is to be retained in the loader's symbol table until the module end is encountered. The referenced expression is the one immediately following the name number. It may contain values that have not been defined previously, but all such values must be available to the loader prior to the module end.

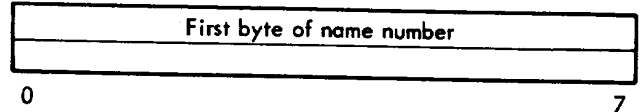
After generating this load item, the processor need not retain the value for the forward reference, since that responsibility is then assumed by the loader. However, the processor must retain the symbolic name and forward reference number assigned to the forward reference (until module end).

External Definition

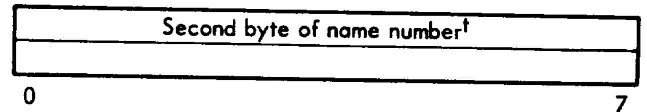
Byte 0




Byte 1



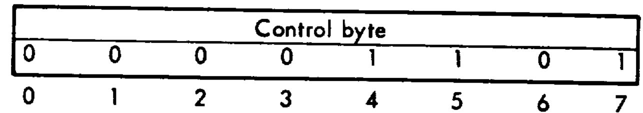

Byte 2




This item defines the value (expression) for an external definition name. The name number refers to a previously declared definition name. The referenced expression is the one immediately following the name number.

Define Start

Byte 0




This item defines the starting address (expression) to be used at the completion of loading. The referenced expression is the one immediately following the control byte.

## EXPRESSION EVALUATION

A processor must generate an object language expression whenever it needs to communicate to the loader one of the following:

1. A program load origin.
2. A program starting address.

†If the module has fewer than 256 previously assigned name numbers, this byte is absent.

3. An external definition value.
4. A forward reference value.
5. A field definition value.

Such expressions may include sums and differences of constants, addresses, and external or forward reference values that, when defined, will themselves be constants or addresses.

After initiation of the expression mode, by the use of a control byte designating one of the five items described above, the value of an expression is expressed as follows:

1. An address value is represented by an offset from the control section base plus the value of the control section base.
2. The value of a constant is added to the accumulated sum by generating an Add Constant (see below) control byte followed by the value, right-justified in four bytes.

   The offset from the control section base is given as a constant representing the number of units of displacement from the control section base, at the resolution of the address of the item. That is, a word address would have its constant portion expressed as a count of the number of words offset from the base, while the constant portion of a byte address would be expressed as the number of bytes offset from the base.

   The control section base value is accumulated by means of an Add Value of Declaration (see below) or Subtract Value of Declaration load item specifying the desired resolution and the declaration number of the control section base. The loader adjusts the base value to the specified address resolution before adding it to the current partial sum for the expression.

   In the case of an absolute address, an Add Absolute Section (see below) or Subtract Absolute Section control byte must be included in the expression to identify the value as an address and to specify its resolution.
3. An external definition or forward reference value is included in an expression by means of a load item adding or subtracting the appropriate declaration or forward reference value. If the value is an address, the resolution specified in the control byte is used to align the value before adding it to the current partial sum for the expression. If the value is a constant, no alignment is necessary.

Expressions are not evaluated by the loader until all required values are available. In evaluating an expression, the loader maintains a count of the number of values added or subtracted at each of the four possible resolutions. A separate counter is used for each resolution, and each counter is incremented or decremented by 1 whenever a value of the corresponding resolution is added to or subtracted from the loader's expression accumulator. The final accumulated sum is a constant, rather than an address value, if the final count in all four counters is equal to 0. If the final count in one (and only one) of the four counters is equal to +1 or -1, the accumulated sum is a "simple address" having the resolution of the nonzero counter. If more than one of the four counters have a nonzero final count, the accumulated sum is termed a "mixed-resolution expression" and is treated as a constant rather than an address.

The resolution of a simple address may be altered by means of a Change Expression Resolution (see below) control byte. However, if the current partial sum is either a constant or a mixed-resolution value when the Change Expression Resolution control byte occurs, then the expression resolution is unaffected.

Note that the expression for a program load origin or starting address must resolve to a simple address, and the single nonzero resolution counter must have a final count of +1 when such expressions are evaluated.

In converting a byte address to a word address, the two least significant bits of the address are truncated. Thus, if the resulting word address is later changed back to byte resolution, the referenced byte location will then be the first byte (byte 0) of the word.

After an expression has been evaluated, its final value is associated with the appropriate load item.

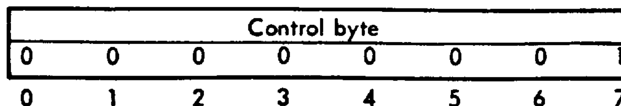
In the following diagrams of load item formats, RR refers to the address resolution code. The meaning of this code is given in the table below.

| RR | Address Resolution |
|---|---|
| 00 | Byte |
| 01 | Halfword |
| 10 | Word |
| 11 | Doubleword |

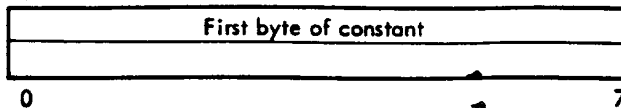
The load items discussed in this appendix, "Expression Evaluation", may appear only in expressions.

Add Constant

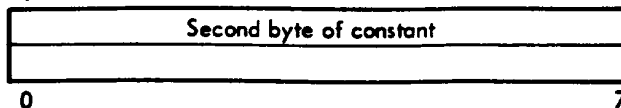
Byte 0



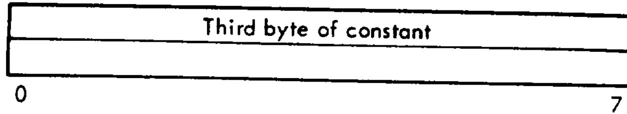

Byte 1



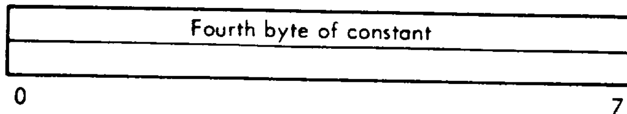

Byte 2



Byte 3



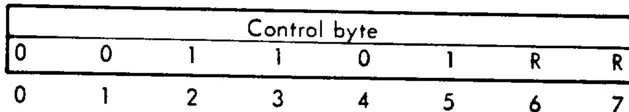

Byte 4




This item causes the specified 4-byte constant to be added to the loader's expression accumulator. Negative constants are represented in two's complement form.

Add Absolute Section

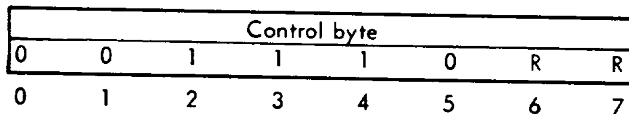
Byte 0




This item identifies the associated value (expression) as a positive absolute address. The address resolution code, RR, designates the desired resolution.

Subtract Absolute Section

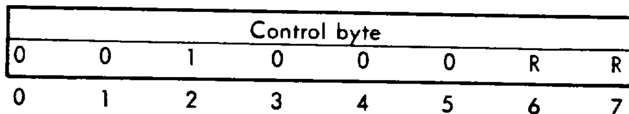
Byte 0



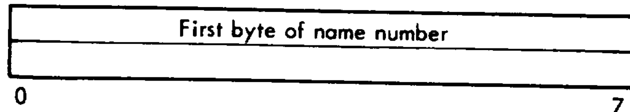

This item identifies the associated value (expression) as a negative absolute address. The address resolution code, RR, designates the desired resolution.

Add Value of Declaration

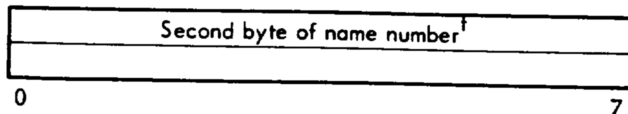
Byte 0




Byte 1




Byte 2



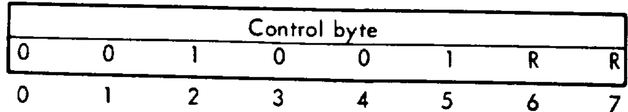

†If the module has fewer than 256 previously assigned name numbers, this byte is absent.

This item causes the value of the specified declaration to be added to the loader's expression accumulator. The address resolution code, RR, designates the desired resolution, and the name number refers to a previously declared definition name that is to be associated with the first location of the allocated section.

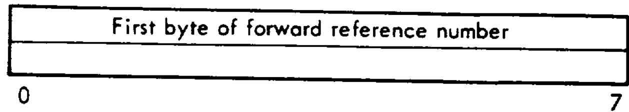
One such item must appear in each expression for a relocatable address occurring within a control section, adding the value of the specified control section declaration (i.e., adding the byte address of the first location of the control section).

Add Value of Forward Reference

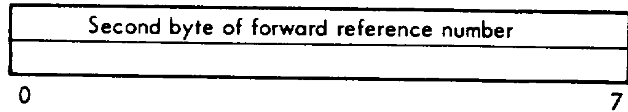
Byte 0




Byte 1



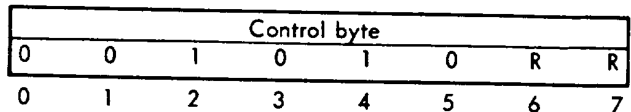

Byte 2



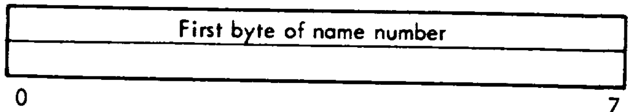

This item causes the value of the specified forward reference to be added to the loader's expression accumulator. The address resolution code, RR, designates the desired resolution, and the designated forward reference must not have been defined previously.

Subtract Value of Declaration

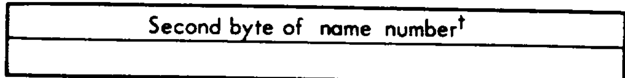
Byte 0




Byte 1




Byte 2




This item causes the value of the specified declaration to be subtracted from the loader's expression accumulator. The address resolution code, RR, designates the desired resolution, and the name number refers to a previously declared definition name that is to be associated with the first location of the allocated section.

†If the module has fewer than 256 previously assigned name numbers, this byte is absent.

Subtract Value of Forward Reference

Byte 0

| Control byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 1 | R | R |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Byte 1

| First byte of forward reference number | |
|---|---|
| 0 | 7 |

Byte 2

| Second byte of forward reference number | |
|---|---|
| 0 | 7 |

This item causes the value of the specified forward reference to be subtracted from the loader's expression accumulator. The address resolution code, RR, designates the desired resolution, and the designated forward reference must not have been defined previously.

Change Expression Resolution

Byte 0

| Control byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 | R | R |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

This item causes the address resolution in the expression to be changed to that designated by RR.

Expression End

Byte 0

| Control byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

This item identifies the end of an expression (the value of which is contained in the loader's expression accumulator).

## FORMATION OF INTERNAL SYMBOL TABLES

The three object code control bytes described below are required to supply the information necessary in the formation of Internal Symbol Tables.

In the following diagrams of load item formats, Type refers to the symbol types supplied by the object language and maintained in the symbol table. IR refers to the internal resolution code. Type and resolution are meaningful only when the value of a symbol is an address. In this case, it is highly likely that the processor knows the type of value that is in the associated memory location, and the type field identifies it. The resolution field indicates the resolution of the location counter at the time the symbol was defined. The following tables summarize the combinations of value and meaning.

Symbol Types

| Type | Meaning of 5-Bit Code |
|---|---|
| 00000 | Instruction |
| 00001 | Integer |
| 00010 | Short floating point |
| 00011 | Long floating point |
| 00110 | Hexadecimal (also for packed decimal) |
| 00111 | EBCDIC text (also for unpacked decimal) |
| 01001 | Integer array |
| 01010 | Short floating-point array |
| 01011 | Long floating-complex array |
| 01000 | Logical array |
| 10000 | Undefined symbol |

Internal Resolution

| IR | Address Resolution |
|---|---|
| 000 | Byte |
| 001 | Halfword |
| 010 | Word |
| 011 | Doubleword |

Type Information for External Symbol

Byte 0

| Control byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Byte 1

| Type field | | IR field | |
|---|---|---|---|
| 0 | 4 | 5 | 7 |

Byte 2

| Declaration number | |
|---|---|
| 0 | 7 |

Byte 3 (if required)

| Declaration number (continued) | |
|---|---|
| 0 | 7 |

This item provides type information for external symbols. The Type and IR fields are defined above. The declaration number field consists of one or two bytes (depending on the current declaration count) which specifies the declaration number of the external definition.

Type and EBCDIC for Internal Symbol

Byte 0

| Control byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | |

Byte 1

| Type field | IR field |
|---|---|
| | |

0 4 5 7

Byte 2

| Length of name (EBCDIC characters) |
|---|
| |

0 7

Byte 3

| First byte of name in EBCDIC |
|---|
| |

0 7

Byte n

| Last byte of name in EBCDIC |
|---|
| |

0 7

Byte n + 1

| Expression defining value of internal symbol |
|---|
| |

0 7

This item supplies type and EBCDIC for an internal symbol. The load items for Type and IR are as above. Length of name specifies the length of the EBCDIC name in characters. The name, in EBCDIC, is specified in the required number of bytes, followed by a byte containing the expression defining the internal symbol

EBCDIC for an Undefined Symbol

Byte 0

| Control byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Byte 1

| Length of name (EBCDIC characters) |
|---|
| |

0 7

Byte 2

| First byte of name in EBCDIC |
|---|
| |

0 7

Byte n

| Last byte of name in EBCDIC |
|---|
| |

0 7

Byte n + 1

| First byte of symbol associated forward reference number |
|---|
| |

0 7

This item is used to associate a symbol with a forward reference. The length of name and name in EBCDIC are the same as in the above item. The last two bytes specify the forward reference number with which the above symbol is to be associated.

## LOADING

Load Absolute

Byte 0

| Control byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | N | N | N | N |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Byte 1

| First byte to be loaded |
|---|
| |

0 7

⋮

Byte NNNN

| Last byte to be loaded |
|---|
| |

0 7

This item causes the next NNNN bytes to be loaded absolutely (NNNN is expressed in natural binary form, except that 0000 is interpreted as 16 rather than 0). The load location counter is advanced appropriately.

Load Relocatable (Long Form)

Byte 0

| Control byte | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | Q | C | R | R |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Byte 1

| First byte of name number |
|---|
| |

0 7

Byte 2

| Second byte of name number† |
|---|
| |

0 7

This item causes a 4-byte word (immediately following this load item) to be loaded, and relocates the address field according to the address resolution code, RR. Control bit C designates whether relocation is to be relative to a forward reference (C = 1) or relative to a declaration (C = 0). Control bit Q designates whether a 1-byte (Q = 1) or a 2-byte (Q = 0) name number follows the control byte of this load item.

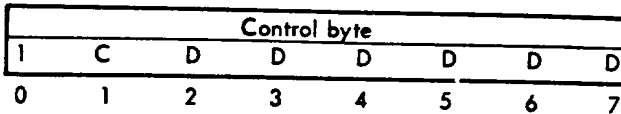
If relocation is to be relative to a forward reference, the forward reference must not have been defined previously. When this load item is encountered by the loader, the load location counter can be aligned with a word boundary by loading the appropriate number of bytes containing all zeros (e.g., by means of a load absolute item).

---

†If the module has fewer than 256 previously assigned name numbers, this byte is absent.

### Load Relocatable (Short Form)

Byte 0

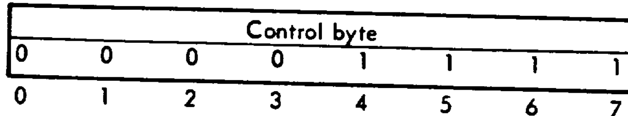



This item causes a 4-byte word (immediately following this load item) to be loaded, and relocates the address field (word resolution). Control bit C designates whether relocation is to be relative to a forward reference (C = 1) or relative to a declaration (C = 0). The binary number DDDDDD is the forward reference number or declaration number by which relocation is to be accomplished.

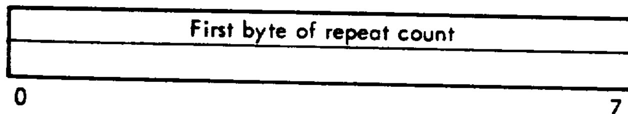
If relocation is to be relative to a forward reference, the forward reference must not have been defined previously. When this load item is encountered by the loader, the load location counter must be on a word boundary (see "Load Relocatable (Long Form)", above).

### Repeat Load

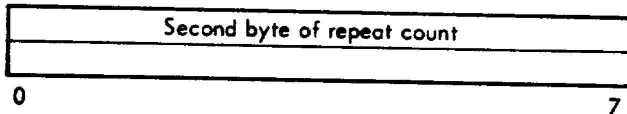
Byte 0




Byte 1



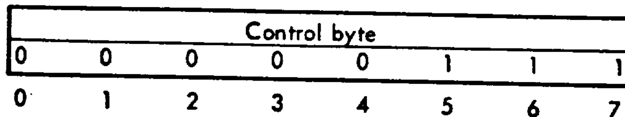

Byte 2



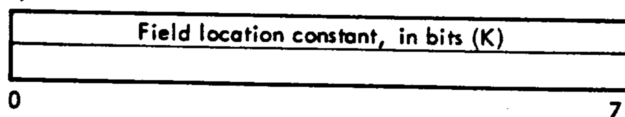

This item causes the loader to repeat (i.e., perform) the subsequent load item a specified number of times. The repeat count must be greater than 0, and the load item to be repeated must follow the repeat load item immediately.

### Define Field

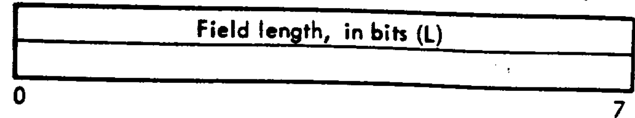
Byte 0




Byte 1




Byte 2




This item defines a value (expression) to be added to a field in previously loaded information. The field is of length L ($1 \le L \le 255$) and terminates in bit position T, where:

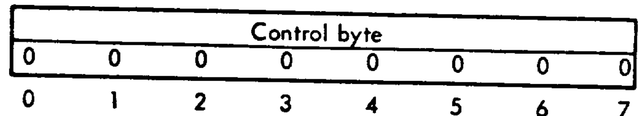
T = current load bit position −256 +K.

The field location constant, K, may have any value from 1 to 255. The expression to be added to the specified field is the one immediately following byte 2 of this load item.

## MISCELLANEOUS LOAD ITEMS

### Padding

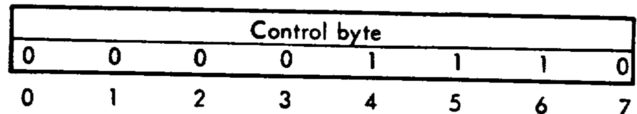
Byte 0



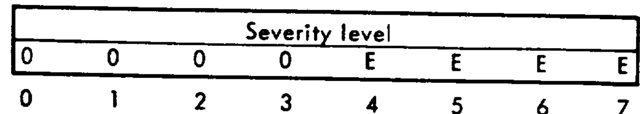

Padding bytes are ignored by the loader. The object language allows padding as a convenience for processors.

### Module End

Byte 0




Byte 1




This item identifies the end of the object module. The value EEEE is the error severity level assigned to the module by the processor.

## OBJECT MODULE EXAMPLE

The following example shows the correspondence between the statements of a Symbol source program and the string of object bytes output for that program by the assembler. The program, listed below, has no significance other than illustrating typical object code sequences.

Example

| Line | Location | Value | Flag | Label | Command | Argument | Comment |
|---|---|---|---|---|---|---|---|
| 1 | | | | | DEF | AA,BB,CC | CC IS UNDEFINED BUT CAUSES NO ERROR |
| 2 | | | | | REF | RZ,RTN | EXTERNAL REFERENCES DECLARED |
| 3 | 00000 | | | ALPHA | CSECT | | DEFINE CONTROL SECTION ALPHA |
| 4 | 000C8 | | | | ORG | 200 | DEFINE ORIGIN |
| 5 | 000C8 | 22000000 | N | AA | LI, CNT | 0 | DEFINES EXTERNAL AA; CNT IS A FWD REF |
| 6 | 000C9 | 32000000 | N | | LW, R | RZ | R IS A FORWARD REFERENCE; |
| 7 | | | | * | | | RZ IS AN EXTERNAL REFERENCE, |
| 8 | | | | * | | | AS DECLARED IN LINE 2 |
| 9 | 000CA | 50000000 | N | RPT | AH, R | KON | DEFINES RPT; R AND KON ARE |
| 10 | | | | * | | | FORWARD REFERENCES |
| 11 | 000CB | 69200000 | F | | BCS, 2 | BB | BB IS AN EXTERNAL DEFINITION |
| 12 | | | | * | | | USED AS A FORWARD REFERENCE |
| 13 | 000CC | 20000001 | N | | AI, CNT | 1 | CNT IS A FORWARD REFERENCE |
| 14 | 000CD | 680000CA | | | B | RPT | RPT IS A BACKWARD REFERENCE |
| 15 | 000CE | 68000000 | X | | B | RTN | RTN IS AN EXTERNAL REFERENCE |
| 16 | 000CF | 0001 | A | KON | DATA, 2 | 1 | DEFINES KON |
| 17 | | 00000003 | | R | EQU | 3 | DEFINES R |
| 18 | | 00000004 | | CNT | EQU | 4 | DEFINES CNT |
| 19 | 000D0 | 224FFFFF | A | BB | LI, CNT | -1 | DEFINES EXTERNAL BB THAT HAS |
| 20 | | | | * | | | ALSO BEEN USED AS A FORWARD |
| 21 | | | | * | | | REFERENCE |
| 22 | 000C8 | | | | END | AA | END OF PROGRAM |

CONTROL BYTES (In Binary)

| Begin Record | Record number: 0 | | |
|---|---|---|---|
| 00111100 | Record type: not last, Mode binary, Format: object language. | | Record control information not part of load item |
| 00000000 | Sequence number 0 | | |
| 01100011 | Checksum: 99 | | |
| 01101100 | Record size: 108 | | |
| | 03020101 (hexadecimal code comprising the load item) | | Source Line 1 |
| 00000011 | Declare external definition name (2 bytes) Name: AA | Declaration number: 1 | |
| | 03020202 | | |
| 00000011 | Declare external definition name (2 bytes) Name: BB | Declaration number: 2 | |
| | 03020303 | | |
| 00000011 | Declare external definition name (2 bytes) Name: CC | Declaration number: 3 | |
| | 0502D9E9 | | Source Line 2 |
| 00000101 | Declare primary reference name (2 bytes) Name RZ | Declaration number: 4 | |
| | 0503D9E3D5 | | |
| 00000101 | Declare primary reference name (3 butes) Name: RTN | Declaration number: 5 | |
| | 0A010100000320200002 | | Source Line 5[†] |
| 00001010 | Define external definition | | |
| | Number 1 | | |
| 00000001 | Add constant: 800 X'320' | | |
| 00100000 | Add value of declaration (byte resolution) | | |
| | Number 0 | | |
| 00000010 | Expression end | | |
| | 040100000320200002 | | Source Line 4 |
| 00000100 | Origin | | |
| 00000001 | Add constant: 800 X'320' | | |
| 00100000 | Add value of declaration (byte resolution) | | |
| | Number 0 | | |
| 00000010 | Expression end | | |
| | 4422000000 | | Source Line 5 |
| 01000100 | Load absolute the following 4 bytes: X'22000000' | | |
| | 07EB0426000002 | | |
| 00000111 | Define field | | |
| | Field location constant: 235 bits | | |
| | Field length: 4 bits | | |
| | Add the following expression to the above field: | | |
| 00100110 | Add value of forward reference (word resolution) | | |
| | Number 0 | | |
| 00000010 | Expression end | | |

[†] No object code is generated for source lines 3 (define control section) or 4 (define origin) at the time they are encountered. The control section is declared at the end of the program after Symbol has determined the number of bytes the program requires. The origin definition is generated prior to the first instruction.

| Code | Description | Source |
|---|---|---|
| | 8432000000 | Source Line 6 |
| 10000100 | Load relocatable (short form). Relocate address field (word resolution) | |
| | Relative to declaration number 4 | |
| | The following 4 bytes: X'32000000' | |
| | 07EB0426000602 | |
| 00000111 | Define field | |
| | Field location constant: 235 bits | |
| | Field length: 4 bits | |
| | Add the following expression to the above field: | |
| 00100110 | Add value of forward reference (word resolution) | |
| | Number 6 | |
| 00000010 | Expression end | |
| | CC50000000 | Source Line 9 |
| 11001100 | Load relocatable (short form). Relocate address field (word resolution) | |
| | Relative to forward reference number 12 | |
| | The following 4 bytes: X'50000000' | |
| | 07EB0426000602 | |
| 00000111 | Define field | |
| | Field location constant: 235 bits | |
| | Field length: 4 bits | |
| | Add the following expression to the above field: | |
| 00100110 | Add value of forward reference (word resolution) | |
| | Number 6 | |
| 00000010 | Expression end | |
| | D269200000 | Source Line 11 |
| 11010010 | Load relocatable (short form). Relocate address field (word resolution) | |
| | Relative to forward reference number 18 | |
| | The following 4 bytes: X'69200000' | |
| | 4420000001 | Source Line 13 |
| 01000100 | Load absolute the following 4 bytes: X'20000001' | |
| | 07EB0426000002 | |
| 00000111 | Define field | |
| | Field location constant: 235 bits | |
| | Field length: 4 bits | |
| | Add the following expression to the above field: | |
| 00100110 | Add value of forward reference (word resolution) | |
| | Number 0 | |
| 00000010 | Expression end | |
| | 80680000CA | Source Line 14 |
| 10000000 | Load relocatable (short form). Relocate address field (word resolution) | |
| | Relative to declaration number 0 | |
| | The following 4 bytes: X'680000CA' | |
| | 8568000000 | Source Line 15 |
| 10000101 | Load relocatable (short form). Relocate address field (word resolution) | |
| | Relative to declaration number 5 | |
| | The following 4 bytes: X'68000000' | |
| | 08 | Source Line 16 |
| 00001000 | Define forward reference (continued in record 1) | |

| Begin Record | Record number 1 | |
|---|---|---|
| 00011100 | Record type: last, Mode: binary, Format: object language. | Record Control Information |
| 00000001 | Sequence number 1 | |
| 11101100 | Checksum: 236 | |
| 01010001 | Record size: 81 | |
| | 000C010000033C200002 (continued from record 0) | Source Line 16 |
| | Number 12 | |
| 00000001 | Add constant: 828 X'33C' | |
| 00100000 | Add value of declaration (byte resolution) | |
| | Number 0 | |
| 00000010 | Expression end | |
| | 42001 | |
| 01000010 | Load absolute the following 2 bytes: X'0001' | |
| | 0800060100000000302 | Source Line 17 |
| 00001000 | Define forward reference | |
| | Number 6 | |
| 00000001 | Add constant: 3 X'3' | |
| 00000010 | Expression end | |
| | 0800000100000000402 | Source Line 18 |
| 00001000 | Define forward reference | |
| | Number 0 | |
| 00000001 | Add constant: 4 X'4' | |
| 00000010 | Expression end | |
| | 0F00024100 | Advance to Word Boundary |
| 00001111 | Repeat load | |
| | Repeat count: 2 | |
| 01000001 | Load absolute the following 1 bytes: X'00' | |
| | 0800120100000340200002 | Source Line 19 |
| 00001000 | Define forward reference | |
| | Number 18 | |
| 00000001 | Add constant: 832 X'340' | |
| | Add value of declaration (byte resolution) | |
| | Number 0 | |
| 00000010 | Expression end | |
| | 0A020100000340200002 | |
| 00001010 | Define external definition | |
| | Number 2 | |
| 00000001 | Add constant: 832 X'340' | |
| 00100000 | Add value of declaration (byte resolution) | |
| | Number 0 | |
| 00000010 | Expression end | |
| | 44224FFFFF | |
| 01000100 | Load absolute the following 4 bytes: X'224FFFFF' | |
| | 0D0100000320200002 | Source Line 22 |
| 00001101 | Define start | |
| 00000001 | Add constant: 800 X'320' | |
| 00100000 | Add value of declaration (byte resolution) | |
| | Number 0 | |
| 00000010 | Expression end | |

| | |
|---|---|
| | 0B000344 |
| 00001011 | Declare standard control section declaration number: 0<br>Access code: Full access. Size 836 X'344' |
| | 0E00 |
| 00001110 | Module end<br>Severity level: X'0' |

A table summarizing control byte codes for object language load items is given below.

| Object Code Control Byte | Type of Load Item |
|---|---|
| 0 0 0 0 0 0 0 0 | Padding |
| 0 0 0 0 0 0 0 1 | Add constant |
| 0 0 0 0 0 0 1 0 | Expression end |
| 0 0 0 0 0 0 1 1 | Declare external definition name |
| 0 0 0 0 0 1 0 0 | Origin |
| 0 0 0 0 0 1 0 1 | Declare primary reference name |
| 0 0 0 0 0 1 1 0 | Declare secondary reference name |
| 0 0 0 0 0 1 1 1 | Define field |
| 0 0 0 0 1 0 0 0 | Define forward reference |
| 0 0 0 0 1 0 0 1 | Declare dummy section |
| 0 0 0 0 1 0 1 0 | Define external definition |
| 0 0 0 0 1 0 1 1 | Declare standard control section |
| 0 0 0 0 1 1 0 0 | Declare nonstandard control section |
| 0 0 0 0 1 1 0 1 | Define start |
| 0 0 0 0 1 1 1 0 | Module end |
| 0 0 0 0 1 1 1 1 | Repeat load |
| 0 0 0 1 0 0 0 0 | Define forward reference and hold |
| 0 0 0 1 0 0 0 1 | Provide type information for external symbol |
| 0 0 0 1 0 0 1 0 | Provide type and EBCDIC for internal symbol |
| 0 0 0 1 0 0 1 1 | EBCDIC and forward reference number for undefined symbol |
| 0 0 0 1 1 1 1 0 | Declare page-bounded control section |
| 0 0 1 0 0 0 R R | Add value of declaration |
| 0 0 1 0 0 1 R R | Add value of forward reference |
| 0 0 1 0 1 0 R R | Subtract value of declaration |
| 0 0 1 0 1 1 R R | Subtract value of forward reference |
| 0 0 1 1 0 0 R R | Change expression resolution |
| 0 0 1 1 0 1 R R | Add absolute section |
| 0 0 1 1 1 0 R R | Subtract absolute section |
| 0 1 0 0 N N N N | Load absolute |
| 0 1 0 1 Q C R R | Load relocatable (long form) |
| 1 C D D D D D D | Load relocatable (short form) |

# APPENDIX C. XEROX STANDARD COMPRESSED LANGUAGE

The Xerox Standard Compressed Language is used to represent source EBCDIC information in a highly compressed form.

Several Xerox processors will accept this form as input or output, will accept updates to the compressed input, and will regenerate source when requested. No information is destroyed in the compression or decompression.

Records may not exceed 108 bytes in length. Compressed records are punched in the binary mode when represented on card media. Therefore, on cards, columns 73 through 80 are not used and are available for comment or identification information. This form of compressed language should not be output to "compressed" files since the I/O compression may cause loss of data.

The first four bytes of each record are for checking purposes. They are as follows:

Byte 1 Identification (00L11000). L = 1 for each record except the last record, in which case L = 0.

Byte 2 Sequence number (0 to 255 and recycles).

Byte 3 Checksum, which is the least significant eight bits of the sum of all bytes in the record except the checksum byte itself. Carries out of the most significant bit are ignored. If the checksum byte is all 1's, do not checksum the record.

Byte 4 Number of bytes comprising the record, including the checking bytes (≤108).

The rest of the record consists of a string of six-bit and eight-bit items. Any partial item at the end of a record is ignored.

The following six-bit items (decimal number assigned) comprise the string control:

| Six-Bit Decimal Item | Function |
|---|---|
| 0 | Ignore. |
| 1 | Not currently assigned. |
| 2 | End of line. |
| 3 | End of file. |
| 4 | Use eight-bit character which follows. |
| 5 | Use n + 1 blanks, next six-bit item is n. |
| 6 | Use n + 65 blanks, next six-bit item is n. |
| 7 | Blank. |
| 8 | 0 |
| 9 | 1 |
| 10 | 2 |
| 11 | 3 |
| 12 | 4 |
| 13 | 5 |
| 14 | 6 |
| 15 | 7 |
| 16 | 8 |
| 17 | 9 |
| 18 | A |
| 19 | B |
| 20 | C |
| 21 | D |
| 22 | E |
| 23 | F |
| 24 | G |
| 25 | H |
| 26 | I |
| 27 | J |
| 28 | K |
| 29 | L |
| 30 | M |
| 31 | N |
| 32 | O |
| 33 | P |
| 34 | Q |
| 35 | R |
| 36 | S |
| 37 | T |
| 38 | U |
| 39 | V |
| 40 | W |
| 41 | X |
| 42 | Y |
| 43 | Z |
| 44 | . |
| 45 | < |
| 46 | ( |
| 47 | + |
| 48 | \| |
| 49 | & |
| 50 | $ |
| 51 | * |
| 52 | ) |
| 53 | ; |
| 54 | ¬ |
| 55 | - |
| 56 | / |
| 57 | , |
| 58 | % |
| 59 | ␣ |
| 60 | > |
| 61 | : |
| 62 | ' |
| 63 | = |

# APPENDIX D. SYSTEM OVERLAY ENTRY POINTS

Table D-1 is a list of all entry points into the various overlays, the overlay containing each entry point, and a brief description of the function of the entry point.

Table D-1. System Overlay Entry Points

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| :#ALTENT | DBDW | Enter debug from alternate PSD |
| :#ASSGN | DBC2 | Process debug assign command |
| :#BRNCH | DBS3 | Branch into user program from debug |
| :#CALENT | DBDW | Enter debug from CAL exit |
| :#CONSG | DBC2 | Process debug connect seg. command |
| :#DMPRET | DBC1 | Return from single dump request |
| :#DODMP | DBC3 | Do a dump request |
| :#DOINIT | DBS2 | Entry for initialization |
| :#DORET | DBS2 | Prepare for user return |
| :#DOSNAP | DBS2 | Entry for snap execution |
| :#DOTRAP | DBS2 | Entry after trap |
| :#DOVAL | DBC3 | Do evaluation of name |
| :#DUMP | DBC1 | Process debug dump command |
| :#EXCT | DBS3 | Process debug exec. control command |
| :#INSRT | DBC1 | Process debug insert command |
| :#LOOK | DBC1 | Process debug look command |
| :#MODFY | DBC3 | Process debug modify command |
| :#NAME | DBC1 | Process debug name command |
| :#PATCH | DBC3 | Process debug patch command |
| :#QUIT | DBS3 | Process debug quit command |
| :#REMOV | DBC2 | Process debug remove command |
| :#RERCHK | DBC3 | Process read error check |
| :#SCAN | DBS1 | Scan the input command |
| :#SCNDR | DBC3 | Scan a dump request |
| :#SCNLCX | DBS3 | Scan location forms |

Table D-1. System Overlay Entry Points (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| :#SCNLWX | DBS3 | Scan for location or word |
| :#SCNWDX | DBS3 | Scan ford forms |
| :#SNAP | DBC1 | Process debug snap command |
| :#SNAPIT | DBC1 | Execute snap command |
| :#TINIT | DBC2 | Trap initialization processing |
| :#TRAPEX | DBS2 | Exit from debug trap processing |
| :#TRAPIN | DBC2 | Process trap control |
| :#WKSQZ | DBS1 | Squeeze unused space from debug work space |
| ABEX | ABEX | Process abort and exit CALs for background |
| ABORT | TERM | Process all abort CALs |
| ACTV | IOEX | Process activate CALs |
| ALLOT | ALLOT | Process allot CALs |
| ANALYSE | TEX1 | Analyse errors of TEX |
| ARM | ARM | Process connect,arm,disconnect,disarm CALs |
| ASSIGN | ASSIGN | Process assign CAL |
| BKGSEQ | ABEX | Initiate background sequencing ('C' from IDLE) |
| BKLASSN | BKL1 | Does background DCB assignments |
| BKL1 | BKL1 | Perform background loading functions |
| BREAK | SNAM | Process INT CALs |
| CALLQ | | Sub to CALL QUEUE and wait for I/O completion |
| CALLQP | | Entry to CALLQ with preset priority |
| CFUPDIR | CLOSEX | Update directory entry for altered file and write it to disk |
| CHECK | CHECK | Process CHECK CALs |
| CHECKA | CHECK | Second-CHECK routine |
| CHKBAL | CHECK | Entry to CHECK via BAL |
| CHKBALA | CHECK | Alternate internal entry to CHECK, via a BAL |
| CKD | CKD | Crash dump from CK area |
| CKD2 | CKD2 | Crash dump from CK area, continued |
| CKENACT | TMTYC | Get and test end-action |

Table D-1. System Overlay Entry Points (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| CKENACTS | TMTYC | Get and test end-action in standard FPT |
| CKENACT1 | TMTYC | Test and convert end-action parameter |
| CKENACT2 | TMTYC | Same as CKENACT1(TMTYC) |
| CKINTADR | TMTYC | Test and convert interrupt address |
| CKINTLAB | TMTYC | Test and convert interrupt label |
| CLOSE | READWR | Process CLOSE CALs |
| CLOSEDCB | READWR | Entry to close via BAL |
| CLOSEX | CLOSEX | Routine to close DCBs |
| CLOSRFIL | CLOSEX | Routine to close a DCB assigned to a RAD file |
| COCIO | COCIO | Queue equivalent for COC I/O |
| COCRIP | COCIO | RIPOFF equivalent for COC I/O |
| COCSRDV | COCIO | SERDEV equivalent for COC I/O |
| COCTIME | COCIO | Five second line checking routine |
| COOP | GETNRT | Intercept BKG I/O requests to symbiont dedicated devices |
| CORRES | DEVI | Process correspondence CALs |
| CRD | CRD | Crash dump from SE op-label |
| CRFIL | CLOSEX | Release blocking buffer and RFT entry for closing a file |
| CRS | CRS | Crash save to SE op-label from CK area |
| CRS2 | CRS2 | Continuation of CRS |
| CSEARCH | DBS2 | Debug scan routine to search for commands |
| DBDW | DBDW | Start of data/workspace for debug |
| DBKG | ABEX | Background dump driver |
| DCBBUSY | READWR | SUB to check for an I/O request to a busy DCB |
| DEACTV | IOEX | Process deactivate CALs |
| DEBUG | DBDW | Debug CAL processor |
| DELETE | DELETE | Process DELETE CALs |
| DELFPT | CHECK | Same as CHECK(SIGNAL) entry point |
| DEQ | ENQ | Process DEQUEUE CALs |
| DEVI | DEVI | Process 'set' portion of device CALs |

Table D-1. System Overlay Entry Points (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| DEVN | DEVI | Process 'get' portion of device CALs |
| DFGD | DUMP | CT return to CT dump after break |
| DFGDBAL | DUMP | DUMP break to check for other CT work |
| DFM | DEVI | Process device file mode CALs |
| DISARM | ARM | Same entry point as ARM(ARM) |
| DRC | DEVI | Process device DIR. Record format CALs |
| DUMP | DUMP | Performs a memory DUMP |
| DVF | DEVI | Process device vertical format CALs |
| EMARECB | TMGETP | Sub. to chain an ECB to the R-task |
| EMARECBX | TMGETP | Sub. to chain an ECB to the R-task in reverse priority |
| EMBLDECB | TMGETP | Sub. to build an ECB from a standard FPT |
| EMDATAI | TMGETP | Sub. to process a data area into an ECB |
| EMDATAO | TMGETP | Sub. to remove a data area to users receiving area |
| EMGETECB | TMTYC | Sub. to create a new ECB linked to the current task |
| EMGETEM | TMTYC | Sub. to create a new ECB linked to any task |
| EMGETFPT | TMTYC | Sub. to get an original FPT address |
| EMSETR3 | CHECK | Set R3 to an FPT addr based on FPT addr in an ECB |
| EMSETR3A | CHECK | Set R3 to an FPT addr based on FPT addr in R3 |
| EMWAIT | TMTYC | Sub. to control wait states |
| ENQ | ENQ | Process ENQUEUE CALs |
| ENQABNM | ENQ | Abnormal condition sub. for ENQUEUE ECBs |
| ENQCHK | ENQ | Sub. to check ENQUEUE ECBs |
| ERRSEND | LOG | Routine to put an operator message into the Error Log |
| ESU | ESU | Process error summary key-in |
| EXTM | EXTM | Process exterminate CALs |
| FGLBADLM | FGL2 | Abort primary load module initiation |
| FGLMEMCK | FGL3 | Check availability of unmapped memory region |
| FGLMSG | FGL1 | Output a message for the primary loader |
| FGLOKLM | FGL2 | Complete primary load module initiation |

Table D-1. System Overlay Entry Points (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| FGL1 | FGL1 | Primary program release |
| FGL2 | FGL2 | Primary program load (initialize tables) |
| FGL3 | FGL3 | Primary program load (read in root and PUBLIBS) |
| FINDBB | FINDBB | Get a blocking buffer |
| FINDDIR | FINDBB | FIND (or allot) extent O of a file |
| FINDDIRX | FINDBB | FIND (or allot) extent N of a file |
| FMBLDECB | GETNRT | Build an I/O ECB |
| FMCHECK | CHECK | Process I/O CHECK CALs |
| FMCKWP | RWFILE | Check for write protection violations |
| FMCK1 | CHECK | Internal entry to FMCHECK |
| FMCK2 | CHECK | Internal entry to FMCHECK |
| FMCK3 | CHECK | Internal entry to FMCHECK |
| FMDELETE | DELETE | DELETE file (extent) entries from permanent directory |
| FMGETEXT | RWEXT | Get next extent of an extended file |
| FMJCL | TTJOB | Clean up RFT and DCT entries at job termination |
| FMMASTX | RWFILE | Determine MASTD index for an area |
| FMOPL2AD | GETNRT | Get caller's OPLBS2 table address |
| FMTCL | TT | Cleanup files for a terminating task |
| FPTBSY | READWR | Check for an I/O request to a busy FPT |
| GENCHARS | PRINT | PRINT expanded text for break pages |
| GETANAME | ESU | Subroutine to get account name |
| GETDCBAD | GETNRT | Get DCB address from FPT |
| GETDCTX | GETNRT | Get device index from DCB |
| GETIOID | KEYSCN | Scan an I/O designator (FILE, OPLABEL, or DEVICE) |
| GETNRT | GETNRT | Internal entry to read/write processing |
| GETOPT | | Get options for key-ins, in KEY3 - KEY7 |
| GETTIME | SIGNAL | Process GETTIME CALs |
| HOURLOG | LOG | Log hourly timestamp |
| IBBPARAM | RWBFIL | Sub to increment the file position in a blocked file |

Table D-1. System Overlay Entry Points (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| INIT | | Perform boot-time initialization of CPR |
| INITLOG | LOG | Routine to initialize the error log file when DT keyin is done |
| INSDBUF | SDBUF | Input side buffering logic |
| IOEX | IOEX | Process all IOEX CALs |
| IPLMM | IPLMM | Do memory management initialization at boot time |
| IPLSYM | IPLSYM | Initialize symbiont areas and job number |
| JMTENQ | TTJOB | Clean up job level ENQs |
| JMTERM | TTJOB | Destroy a job when last task has terminated |
| JOBDLTE | JOB2 | Sub. to delete a job's files in an area |
| JOBDLTEA | JOB2 | Sub. to selectively delete a job's files in an area |
| JOBMSG | JOB2 | Sub. to output messages to OC device |
| JOBSCAN | JOB1 | Validate a JOB card for symbiont input |
| JOB1 | JOB1 | Process a M:JOB CAL, Part 1 |
| JOB2 | JOB2 | Process a M:JOB CAL, Part 2 |
| JSCAN | JOB1 | Validate and format a JOB card |
| JTRAP | TRAPS | Process job trap CAL |
| KEY1 | KEY1 | Decode key-in keyword, branch to proper overlay for processing |
| KEY1A04 | KEY1 | Process key-err message typeouts |
| KEY2 | KEY2 | Process key-ins in KEY2 overlay |
| KEY3 | KEY3 | Process key-ins in KEY3 overlay |
| KEY4 | KEY4 | Process key-ins in KEY4 overlay |
| KEY5 | KEY5 | Process key-ins in KEY5 overlay |
| KEY6 | KEY6 | Process key-ins in KEY6 overlay |
| KEY7 | KEY7 | Process key-ins in KEY7 overlay |
| KJOB | EXTM | Process KJOB CALs |
| LOAD | | Entry to JCP loader |
| LOADACI | MMROOT | Sub. to load ACI for a task |
| LOADMAP | MMROOT | Sub. to load MAP and ACE for a task |

Table D-1. System Overlay Entry Points (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| LOG | LOG | Move error log records from log stack to ER OP label |
| MEDIA | MEDIA | Resident copy loop code and DCB's, FPT's |
| MEDIACAL | SNAM | Media CAL processor |
| MEDIATSK | MED1 | Start of media task |
| MEDRLOOP | MEDIA | Start of main copy loop in resident module |
| MEDRLOPA | MEDIA | Alt. entry to main loop for 'ALL' copies |
| MEDR900 | MEDIA | Common CAL error routine (do nothing routine) |
| MEDOEXIT | MED1 | Entry into MED1 module from resident copy loop if no errors |
| MEDO90 | MED1 | Error routine, reading input file |
| MEDO94 | MED1 | Error routine, writing output file |
| MEDO99 | MED1 | Error routine, DCB abnormal and reading shared files |
| MED1 | MED1 | Part 1 of media task; request selection and initiation |
| MED2 | MED2 < | Part 2 of media task; post processing clean-up |
| MED600 | MED2 | Entry to Part 2 of media task |
| MED800 | MED2 | Convert media ID number to EBCDIC in output line |
| MED810 | MED2 | Acquire a device for media use |
| MED820 | MED2 | Copy preample (printer break page print) |
| MED830 | MED2 | Copy postamble (printer clean-up) |
| MED840 | MED2 | Copy postamble (tape positioning) |
| MED880 | MED2 | Wait routine for device manual, symbiont device conflict |
| MMABNM | MM05 | Sub. to process abnormal ECB exits |
| MMACT | MM01 | Process activate CALs |
| MMCAL | MM01 | Process all memory management CALs |
| MMCHECK | MM04 | Sub. to check memory management ECBs |
| MMDEACT | MM02 | Process deactivate CALs |
| MMERASE | MM02 | Process erase CALs |
| MMEXEC | MM04 | Memory management executive task |
| MMFETCH | MMROOT | Sub. to fetch one word from any real address |

Table D-1. System Overlay Entry Point (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| MMFMP | MMROOT | Sub. to find a memory partition |
| MMFOV | MM01 | Sub. to find the OVLOAD entry for a segment |
| MMGETP | MM02 | Process GETPAGE CALs |
| MMGJRP | MM03 | Subroutine to get job reserved pages |
| MMGP | MM01 | Sub. to get pages for a segment |
| MMGPPS | MM03 | Sub. to get preferred partition pages |
| MMGSTM | MMROOT | Sub. to get one page of real memory |
| MMGTRP | MM03 | Sub. to get task reserved pages |
| MMICHK | MM04 | Internal entry into MMCHECK |
| MMLOCK | MM06 | Process LOCK CALs |
| MMMOVE | MMROOT | Move contents of a real page to another real page |
| MMOMFPP | MMROOT | Call overlay manager to release pages |
| MMPOST | MM04 | Sub. to post a memory management ECB |
| MMRDS | MM01 | Reset disp skip flag and RLS exclu. use of a SD |
| MMRECB | MM04 | Sub. to create a memory management ECB |
| MMRELP | MM02 | Process RELPAGE CALs |
| MMRELS | MM02 | Sub. to clean up mapped task at termination |
| MMRELSD | MM02 | Sub. to free up SD space at termination |
| MMRFILE | MM05 | Sub. to get roll out file space |
| MMRILW | MMROOT | Sub. to start roll in of a long wait task |
| MMRISEG | MM04 | Sub. to request roll-in of a segment |
| MMRJRP | MM03 | Sub. to release job reserved pages |
| MMROLL | MM04 | Sub. to request memory pages from the memory exec |
| MMROLLIN | MM05 | Sub. to roll in a segment |
| MMROOT | MMROOT | Context block for memory management |
| MMROUT | MM05 | Sub. to roll out a segment |
| MMRP | MM02 | Sub. to release pages in a SD |
| MMRPPS | MM03 | Sub. to release preferred partition pages |
| MMRPPSI | MM03 | Internal entry into sub. MMRPPS |

Table D-1. System Overlay Entry Point (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| MMRPREF | MM06 | Sub. to recover preferred partitions pages |
| MMRREAD | MM05 | Sub. to read a segment from the roll out file |
| MMRRFILE | MM05 | Sub. to release roll out file space |
| MMRSTM | MMROOT | Sub. to release one page of real memory |
| MMRTRP | MM03 | Sub. to release task reserved pages |
| MMRWRITE | MM05 | Sub. to write a segment to the roll out file |
| MMSAC | MM01 | Sub. to set access codes |
| MMSDS | MM01 | Set disp skip flag and wait for exclu. use of a SD |
| MMSEGCK | MM01 | Sub. to verify a segment number |
| MMSETVPN | MM01 | Sub. to generate virtual page number arguments |
| MMSTART | MM01 | Sub. to start the memory management executive |
| MMSTOP | MM04 | Sub. to stop the memory management executive |
| MMSTORE | MMROOT | Sub. to store one word into any real address |
| MMSWAP | MMROOT | Memory management swap control |
| MMSWLK | MM03 | Sub. to set write locks |
| MMTJOB | MM02 | Sub. to do job level cleanup at termination |
| MMTPRIM | MM03 | Sub. to free pages acquired by a primary task |
| MMTSEC | MM02 | Sub. to release ACI and AST space at termination |
| MMUNLOCK | MM06 | Process unlock CALs |
| MMVVPN | MM01 | Sub. to verify a virtual page number |
| MODIFY | EXTM | Same entry as status |
| OFFVERBG | TEX1 | Construct OFF message verbage |
| ONOFFMSG | TEX2 | Write out OFF messages |
| OPEN | READWR | Process OPEN CALs |
| OPENDCB | READWR | Routine to open a DCB |
| OPENX | OPENX | Internal entry to OPENDCB |
| OSEARCH | DBS2 | Debug scan routine to search for machine operations |
| OUTSDBUF | SDBUF | Output side buffering logic |
| PFIL | REWIND | Process all PFIL CALs |

Table D-1. System Overlay Entry Point (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| PFILDEV | REWDEV | Position file on tape devices |
| PINIT | PINIT | Process INIT CALs |
| PINTABNM | PINIT | Sub. to process abnormal ECB exits |
| PLO1 | PLO1 | Routine processes all PUBLIBS |
| PMD | ABEX | Dispatch BKGD to dump itself |
| POLL | SIGNAL | Process all POLL CALs |
| POLLABNM | SIGNAL | Routine to process POLL ECB abnormal conditions |
| POLLCHK | SIGNAL | Routine to process checks on POLL services |
| PPOST | SIGNAL | Process POST CALs |
| PRECDEV | REWDEV | Position records on tape devices |
| PRECORD | REWIND | Process PRECORD CALs |
| PREFMODE | MMO6 | Process PREFMODE CALs |
| PRINT | PRINT | Process PRINT CALs |
| PROMPT | DEVI | Process set PROMPT character CALs |
| PUBLIB | PLO1 | Same as PLO1 |
| RBLOCK | RWBFIL | Sub. to read a block into a blocking buffer |
| RCCUPF | SYM5 | Clean up previous job's input files for the COOP |
| RCCUPJ | SYMS | Clean up previous job's input files for the COOP, Part 2 |
| RCEXU | SYM3 | Process EXU command in the input COOP |
| RCFEOD | SYM3 | Process EOD command in the input COOP |
| RCGETF | SYM3 | Sub. to get next file for the input COOP |
| RCJOB | SYM3 | Sub. to process a JOB card in the input COOP |
| RCOOP | SYM4 | Process input requests to a symbiont dedicated device |
| READDIR | FINDBB | Sub. to read a directory sector |
| READWR | READWR | Process Read/Write CALs |
| RECALARM | CRS | ALARM Receiver CAL |
| RELADBUF | RWBFIL | Release a Blocking Buffer |
| RETELING | TEL2 | |
| REWDEV | REWDEV | Process REWIND CALs on devices |

Table D-1. System Overlay Entry Point (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| REWIND | REWIND | Process REWIND CALs |
| RLS | EXTM | Process RELEASE CALs |
| RUN | RUN | Process all RUN CALs |
| RWBFIL | RWBFIL | Read/Write blocked or compressed RAD files |
| RWDEV | RWDEV | Process Read/Write to devices |
| RWEXTDIR | RWEXT | Get next extent while processing a direct access Read/Write |
| RWEXTSEQ | RWEXT | Get next extent while processing a sequential Read/Write |
| RWFILE | RWFILE | Read/Write processor for disk file |
| SCAN | KEYSCN | Common scan routine for all key-in routines |
| SCEMPTY | SCNEXT | SCHED sub. to find next empty SCHED file entry |
| SCFIND | SCNEXT | SCHED sub. to find match SCHED file entry |
| SCHED | SCHED | Control task entrance to periodic scheduler |
| SCHEDC | RUN | CAL processing for periodic scheduling |
| SCMSG | SCNEXT | SCHED sub. for output of messages |
| SCNEXT | SCNEXT | SCHED sub. to find next executable candidate |
| SCUPDATE | SCNEXT | SCHED sub. to change SCHED file |
| SDBUF | SDBUF | Side buffering processor prolay dummy entry point |
| SEARCHAI | ESU | Initiate AI file search |
| SEGLOAD | EXTM | Process SEGLOAD CALs |
| SETNAME | SNAM | Process SETNAME CALs |
| SETOVR | GETNRT | Subr. to test/set abort override in I/O CALs |
| SETPRI | JOB2 | Sub. to set a job's priority in the directory |
| SETUP | REWIND | Sub. to open a DCB and get its assignment |
| SEX | SEX | Symbiont executive resident code and context |
| SIGABNM | SIGNAL | Routine to process SIGNAL ECB abnormal conditions |
| SIGCHK | SIGNAL | Routine to process checks on signal services |
| SIGNAL | SIGNAL | Process SIGNAL CALs |
| SIGNAL1 | SIGNAL | Internal SIGNAL CAL processor entry point |

Table D-1. System Overlay Entry Points (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| SIMIKEY | SYM3 | Simulate 'SCRYDD,I' key-in for symbionts |
| SJOB | SJOB | Process SJOB CALs |
| SMBACKUP | SYM5 | Perform 'R' and 'B' symbiont key-in options functions |
| SMBCDHEX | SYM4 | Convert number from EBCDIC to hex |
| SMBSYF | SYM3 | Process a 'busy file' error return in symbiont processor |
| SMCLOSF | SYM2 | Sub. to close a symbiont file |
| SMDEVERR | SYM2 | Sub. to process unrecoverable I/O errors in symbionts |
| SMDFIS | JOB2 | Sub. to delete a job's files in the 'IS' area |
| SMDFOS | JOB2 | Sub. to delete a job's files in the 'OS' area |
| SMFEOD | SYM2 | Sub. to handle EOD in output symbiont |
| SMFINDF | SYM3 | Sub. to check for a busy file |
| SMGETF | SYM2 | Sub. to get next file for SYMB/COOP to read |
| SMGETOF | SYM2 | Sub. to get next file for output symbiont to process |
| SMGETQF | SYM2 | Sub. to find a file associated with 'SYYNDD,Q' key-in |
| SMHEXBCD | SYM4 | Convert number from hex to EBCDIC |
| SMINIT | SYM3 | Sub. to initialize context for a symbiont device |
| SMJOBFIN | SYM3 | Sub. to process JOB/ FIN in input symbiont |
| SMWMSG | SYM2 | Format and output symbiont messages to OC |
| SMXKEY | SYM2 | Sub. to process the 'SYYNDD,X' key-in |
| SMXTND | SYM5 | Allot a symbiont extension file |
| SNAM | SNAM | Process setname CALs |
| SNAP | CRS | SNAP key-in processing |
| START | SIGNAL | Process START CALs |
| STATUS | EXIM | Process STATUS CALs |
| STDLB | STDLB | Process STDLB CALs |
| STIMABNM | SIGNAL | Routine to process STIMER ECB abnormal conditions |
| STIMER | SIGNAL | Process STIMER CALs |
| STLBCHK | STDLB | Routine to process checks on STDLB services |
| STOP | SIGNAL | Process STOP CALs |

Table D-1. System Overlay Entry Points (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| STPIO1 | IOEX | Process STOPIC/STARTIO CALs |
| STPIO2 | IOEX | Same entry as STPIO1(IOEX) |
| STRTIO1 | IOEX | Same entry as STPIO1(IOEX) |
| STRTIO2 | IOEX | Same entry as STPIO1(IOEX) |
| SYMCUP | SYM2 | Sub. to do symbiont device clean-up |
| SYM1 | SYM1 | Symbiont executive |
| TAPE | TAPE | Tape handler prolay dummy entry point |
| TDLOAD | MMROOT | Sub. to do actual loading of map and ACI |
| TELCNTRL | TEL2 | Entry for TEL control/break |
| TELERROR | TEL1 | Common TEL error processing |
| TELEXEC | TEL1 | Execution of a TEL command |
| TELREAD | TEL2 | Reads a TEL command |
| TEL3 | TEL3 | Initialize TEL work area |
| TERM | TERM | Process TERM CALs |
| TEST | WAIT | Process TEST CALs |
| TESTBUF | GETNRT | Sub. to test the validity of caller's Read/Write buffer |
| TESTLOOP | TEX2 | Initiates TEX line testing |
| TESTWT4 | GETNRT | Routine to test for delete-on-post I/O request |
| TEX | TEX | Terminal executive context |
| TEXBUFFR | TEX1 | Gets BLK. BUF. for TEX workspace |
| TEXEXEC | TEX2 | Initiates TJE/TEX processing |
| TEXIT | TRAPS | Process trap exit CALs |
| TI | TIO1 | Secondary task initiation Part 1 |
| TICRASH | TIO2 | Secondary task initiation crash routine |
| TIME | WAIT | Process TIME CALs |
| TIO2ABEN | TIO2 | Sub. for abnormal end conditions during sec. task INIT |
| TIO3DBUG | TIO3 | Routine sets up DEBUG controls before task entry |
| TIRFT | TIO3 | Sub. to initialize LIRFT table |
| TISAST | TIO2 | Sub. to sort and store an AST entry |

Table D-1. System Overlay Entry Points (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| TISCHN | TIO3 | Sub. to dechain chained temp space |
| TISD | TIO3 | Sub. to build a segment descriptor |
| TISDECHN | TIO3 | Sub. to chain a segment descriptor |
| TISEARCH | TIO3 | Sub. to search for shared segments |
| TISREAD | TIO2 | Sub. to perform all file reads for task initiation |
| TIS15 | TIO2 | Secondary task initiation Part 2 |
| TIS21 | TIO2 | Entry point from TIO1 for tasks with no segments |
| TMABORT | TERM | Sub. to abort a foreground task |
| TMABRTT | TERM | Sub. to abort a load module |
| TMCKADP | TMTYC | Sub. to check a range of addresses |
| TMCKADR | TMTYC | Sub. to check an address and convert to real if virtual |
| TMDCBERR | EXTM | Sub. to process DCB errors |
| TMDELAET | ENQ | Sub. to free an AET and the EDT if idle |
| TMDEQ | ENQ | Sub. to dequeue an item |
| TMENQ | ENQ | Sub. to enqueue an item |
| TMFINDJ | TMGETP | Sub. to get job ID by JOBNAME |
| TMFINDT | TMGETP | Sub. to get task ID by task name |
| TMGETIDS | TMGETP | Sub. to get job and task identification |
| TMGETJID | TMGETP | Sub. to get job ID from P11 and P12 in FPT |
| TMGETP | TMGETP | Sub. to fetch priority from an FPT |
| TMGETTID | TMGETP | Sub. to get task ID from P3 and P4 in FPT |
| TMGRA | TMTYC | Get the real address and protection for a virtual add address |
| TMLM | TERM | Subroutine to terminate or abort one load module |
| TMSETE | EXTM | Sub. to set R8 and R10 in RTS if CAL processing error |
| TMSETPSD | CHECK | Sub. to alter PSD in RTS |
| TMSETREG | CHECK | Sub. to alter R8 and R10 in RTS |
| TMSTOP | SIGNAL | Internal entry into STOP CAL processor |
| TMTERM | TERM | Sub. to terminate a foreground task |
| TMTRMJ | TERM | Subroutine to terminate all load modules in a job |

Table D-1. System Overlay Entry Points (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| TMTRMT | TERM | Sub. to terminate a load module |
| TMTYC | TMTYC | Sub. to SFT FPT type completion word parameter |
| TMTYCB | TMTYC | Sub. to set FPT type completion word busy |
| TMTYCS | TMTYC | Sbroutine to set FPT type completion in stand, FPT |
| TMTYC15 | TMTYC | Sub. to set TYC in R15 into FPT TYC word |
| TMTYC15S | TMTYC | Sub. to set TYC in R15 into TYC word in stand, FPT |
| TMVADR | TMTYC | Sub. to check a virtual address (no conversion) |
| TMWALL | WAIT | Sub. to do wait all on SECBS |
| TRAPCRSH | TRAPS | Trap crash entry |
| TRAPS | TRAPS | Trap handler entry |
| TRAPS | TRAPS | Internal entry for trap handling |
| TRAP70 | TRAPS | Process TRAP CAL |
| TRTN | TRAPS | Process TRAP return CAL |
| TRTY | TRAPS | Process TRAP retry CALs |
| TRUNCATE | DELETE | Process TRUNCATE CALs |
| TT | TT | Sub. to do secondary task terminations |
| TTDEBUG | DBS1 | Task termination cleanup for DEBUG |
| TTJOB | TTJOB | Sub. to clean job controls for task termination |
| TTLN | TT | Subroutine to determine if this is a TJE job |
| TTPRIM | TT | Sub. to do misc. task cleanup for primary terminations |
| TYPE | PRINT | Process all TYPE CALs |
| USEARCH | DBS2 | Debug scan routine to search for user name symbols |
| VERACCNT | TEX1 | Verify account format |
| WAIT | WAIT | Process WAIT CALs |
| WAITALL | WAIT | Process WAITALL CALs |
| WAITANY | WAIT | Process WAITANY CALs |
| WBLOCK | RWBFIL | Sub. to write out a blocking buffer |
| WCGETJOB | SYM3 | Sub. to define next job file for output COOP |
| WCOOP | SYM4 | Process output requests to a symbiont dedicated device |

Table D-1. System Overlay Entry Points (cont.)

| Entry Point Name | Overlay Name | Description |
|---|---|---|
| WCSTSYM | SYM3 | Sub. to free file space for output COOP |
| WEOF | REWIND | Process WEOF CALs |
| WEOFDEV | REWDEV | Process WEOF to devices |
| WLBLOCK | RWBFIL | Sub. to write the current block of a RAD file |
| WRITDIR | FINDBB | Sub. to write a directory sector |

XEROX

# Reader Comment Form

We would appreciate your comments and suggestions for improving this publication

| Publication No. | Rev. Letter | Title | Current Date |
|---|---|---|---|
| | | | |

How did you use this publication?

- [ ] Learning
- [ ] Installing
- [ ] Sales
- [ ] Reference
- [ ] Maintaining
- [ ] Operating

Is the material presented effectively?

- [ ] Fully Covered
- [ ] Well Illustrated
- [ ] Well organized
- [ ] Clear

What is your overall rating of this publication?

- [ ] Very Good
- [ ] Fair
- [ ] Very Poor
- [ ] Good
- [ ] Poor

What is your occupation?

Your other comments may be entered here. Please be specific and give page, column, and line number references where applicable. To report errors, please use the Xerox Software Improvement or Difficulty Report (1188) instead of this form.

Your name & Return Address

Thank You For Your Interest. (fold & fasten as shown on back, no postage needed if mailed in U.S.A.)

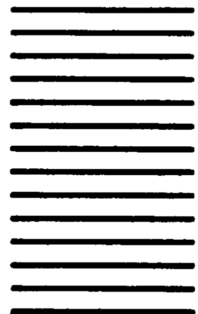
PLEASE FOLD AND TAPE –
NOTE: U. S. Postal Service will not deliver stapled forms

First Class
Permit No. 59153
Los Angeles, CA

BUSINESS REPLY MAIL
No postage stamp necessary if mailed in the United States

Postage will be paid by

Honeywell Information Systems
5250 W. Century Boulevard
Los Angeles, CA 90045



*Attn: Programming Publications*

Fold