Xerox Corporation
701 South Aviation Boulevard
El Segundo, California 90245
213 679-4511

XEROX

Xerox Real-Time Batch Monitor (RBM)

Sigma 5-9 Computers

System

Technical Manual

90 17 00E

May 1975

Price: $8.25

© Xerox Corporation, 1973, 1975

Printed in U.S.A.

REVISION

This publication is a revision of the Xerox Real-Time Batch Monitor (RBM)/System Technical Manual for Sigma 5-9 Computers, Publication Number 90 17 00D (dated October 1973). The revision reflects the C04 version of the system. Changes in the text for C04 are indicated by a vertical line in the margin of the page.

RELATED PUBLICATIONS

Title	Publication No.
Xerox Sigma 5 Computer/Reference Manual	90 09 59
Xerox Sigma 6 Computer/Reference Manual	90 17 13
Xerox Sigma 7 Computer/Reference Manual	90 09 50
Xerox Sigma 8 Computer/Reference Manual	90 17 49
Xerox Sigma 9 Computer/Reference Manual	90 17 33
Xerox Real-Time Batch Monitor (RBM)/RT,BP Reference Manual	90 15 81
Xerox Real-Time Batch Monitor (RBM)/OPS Reference Manual	90 16 47
Xerox Real-Time Batch Monitor (RBM)/RT,BP User's Guide	90 16 53
Xerox Availability Features (CP-R) 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-H/LN Reference Manual	90 09 66
Xerox Extended FORTRAN IV-H/OPS Reference Manual	90 11 44
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 viii

1. RBM INITIALIZATION ROUTINE 1

2. RBM CONTROL TASK 3

Structure 3
Function and Implementation 3
Resident Control Task 3
Key-In Processor 4
Load Module Control (Formerly "Foreground Loader") 4
Background Sequencing 5
Checkpoint Restart (CKPT) 6
Control Task Dump 7

3. I/O HANDLING METHODS 8

Channel Concept 8
Handling Devices 8
Single Interrupt Mode 8
Interrupt-to-Interrupt Mode 8
System Tables 8
IOQ (Request Information) 8
DCT (Device Control) 9
CIT (Channel Information) 9
Handler Tables 9
I/O Control System Overview 10
Interfaces 10
Interfaces into the IOCS 10
Interfaces out of the IOCS 12
IOCS Control Sequence/Example 12
Register Conventions 32
QUEUE 32
CALLSD 32
SERDEV 32
RIPOFF 32
STARTIO 33
CLEANUP/IOSCU 33
REQCOM 33
I/O Error Logging 34
I/O Statistics 34
Side Buffering 34
Output Side Buffering 34
Input Side Buffering 34
IOEX 35
Queued IOEX 35
Dedicated IOEX 35
Disk Pack Track-by-Track Logic 35
Disk Pack Seek Separation 35
Disk Pack Arm-Position Queue Optimization 35
Disk Angular-Position Queue Optimization 36
User I/O Services 36
OPEN 36
CLOSE 36
READ/WRITE 37
PRINT 38
TYPE 38
DFM 38
DVF 38
DEVICE 38
CORRES 38
REWIND 39
WEOF 39
PREC 39
PFILE 39
ALLOT 40
DELETE 40
TRUNCATE 40

4. JOB CONTROL PROCESSOR 41

Overview 41
ASSIGN Command Processing 41
JCP Loader 55
Job Accounting 57
Background TEMP Area Allocation 57

5. FOREGROUND SERVICES 60

Implementation 60
RUN 60
RLS 60
MASTER/SLAVE 60
STOPIO/STARTIO 60
DEACTIVATE/ACTIVATE 61
IOEX 61
TRIGGER, DISABLE, ENABLE, ARM DISARM, CONNECT, DISCONNECT 61
Task Control Block (TCB) 61

6. MONITOR INTERNAL SERVICES 64

RBM Overlays 64
Entry and Exit Point Inventory (EPI) 66
Overlay Inventory (OVI) 66
Event Control Block and Event Control Services 67
Overview of ECB Usage 67
CAL Processor Usage 67
Task-Termination Usage 68
ECB and Data-Area Formats 68
ECBDATA (Word 0) 69
ECBFPT (Word 1) 69
ECBSECB (Word 2) 70
ECBRECB (Word 3) 70
ECBPC (Word 4) 71
ECBENDAC (Word 5) 71
ECBTIME/ECBCOMPL (Word 6) 71
ECBCTLS (Word 7) 71

Dynamic Space — 72
Dynamic-Space Service Calls — 72
GETTEMP — 72
RELTEMP — 72
SYSGEN Considerations — 73
Dispatcher — 73

7. MISCELLANEOUS SERVICES 74

SEGLOAD — 74
Trap Handling — 74
Trap CAL and JTRAP CAL — 74
Trap Processing — 74
TRTN (Trap Return) — 76
TRTY (Trap Retry) — 76
TEXIT (Trap Exit) — 76

8. RBM TABLE FORMATS 77

General System Tables — 77
Disk File Table (RFT) — 77
Device Control Table (DCT) — 78
DCT Format — 78
SYSGEN DCT Consideration — 81
Channel Information Table (CIT) — 81
I/O Queue Table (IOQ) — 82
Blocking Buffers — 84
Master Dictionary — 86
Operational Label Table (OPLBS) — 87
OVLOAD Table (for RBM Overlays Only) — 87
Write Lock Table (WLOCK) — 87
RBM Dispatcher Level Inventory (RDLI) — 88
Associative Enqueue Table (AET) — 89
Purpose — 89
Type — 89
Logical Access — 89
Overview of Usage — 89
Associative Enqueue Table (AET) Format — 90
Task-Controlled Tables — 90
Load Module Inventory (LMI) — 90
LMINAME (LMI1) — 93
LMIPCB, LMIFWA (LMI2) — 93
LMIJID, LMILWA (LMI3) — 93
LMIPL, LMICTXT (LMI4) — 94
LIMSTAT (LMI5) — 94
LMISTD (LMI6) — 94
LMIRTS (LMI7) — 94
LMIMAXS (LMI8) — 94
LMIMAXR (LMI9) — 94
LMIAET — 95
LMISECB — 95
LMIRECB — 95
System Task Inventory (STI) — 95
Purpose — 95
Type — 95
Logical Access — 95
Overview of Usage — 96
STISPCE — 97
STIXRTS — 97
STIRTSB — 97
STIJID — 97
STILMID — 97
STIPRIO — 97
STITCB — 98
STIOVID — 98
STICOUNT — 98
STITIME — 98
STISTAT — 98
STIDNXT — 98
Task Control Block (TCB) — 99
Purpose — 99
Type and Location — 99
Logical Access — 99
Overview of Usage — 99
Task Control Block (TCB) Format — 100
Secondary Task Control Block (STCB) — 101
Purpose — 101
Location and Type — 101
Logical Access — 101
Overview of Usage — 101
Secondary Task Control Block (STCB) Format — 102
Job-Controlled Tables — 104
System Job Inventory (SJI) Table — 104
Purpose — 104
Type — 104
Logical Access — 104
Overview of Usage — 104
System Job Inventory (SJI) Table Format — 106
Job Control Block (JCB) — 106
Purpose — 106
Type — 106
Logical Access — 106
Overview of Usage — 107
Job Control Block Format — 107
Job Program Table (JPT) — 108
Purpose — 108
Type — 108
Logical Access — 108
Overview of Usage — 108
JPT Table Format — 109
Enqueue Definition Table (EDT) — 109
Purpose — 109
Type and Location — 109
Logical Access — 109
Overview of Usage — 109
Enqueue Definition Table (EDT) Format — 111
EDTNAME — 111
EDTEDT — 111
EDTRECB — 111
Load-Module Data Structures — 111
Load Module Headers — 112
Task Load Module Header — 112
PUBLIB Load Module Header — 113
OVLOAD Table (for Load Modules) — 114

9. OVERLAY LOADER 115

Overlay Structure — 115
Overlay Loader Execution — 115

Dynamic Table Area 116
Dynamic Table Order 117
T:SYMBOL and T:VALUE 117
T:VALUE Entry Formats 118
T:SYMBOL Entry Formats 119
T:PUBVAL and T:PUBSYM 119
T:PUBVAL Entry Formats 119
T:PUBSYM Entry Formats 120
T:VALX 120
T:DCB 121
T:SEG 122
B:MT 124
T:DECL 124
T:CSECT 124
T:FWD 125
T:FWDX 125
T:MODULE 126
T:ROMI 126
T:DCBV 127
T:MODIFY 127
Use of the Dynamic Table Area During LIB 128
T:LDEF 129
T:LROM 129
MODULE File 130
EBCDIC File 130
MODIR File 130
DEFREF File 130
Use of Dynamic Table Area During PASSTWO 131
T:GRAN 131
T:ASSN 132
MAP Use of Dynamic Table Area 132
DIAG Use of Dynamic Table Area 133
Root Tables 133
T:PL 134
T:DCBF 134
Scratch Files 135
Program File Format 136
Logical Flow of the Overlay Loader 137
Logical Flow of CCI 137
Logical Flow of PASSONE 137
Logical Flow of LIB 138
Logical Flow of PASSTWO 138
Logical Flow of MAP 138
Logical Flow of DIAG 139
Loader-Generated Table Formats 139
PCB 139
DCBTAB 140
OVLOAD 140
Loading Overlay Loader 140

10. RADEDIT 150

Functional Flow 150
Permanent Disk Area Maintenance 150
Permanent File Directory 150
Control Commands 152
:ALLOT 152
:DELETE 153
:TRUNCATE 153
:SQUEEZE 153
Library File Maintenance 153
Algorithms for Computing Library File Lengths 153
Library File Formats 155
MODIR File 156
MODULE File 156
EBCDIC File 156
DEFREF File 157
Command Execution 157
:ALLOT 158
:COPY 158
:DELETE 158
:SQUEEZE 158
Bad Track Handling 158
Command Execution 158
:BDTRACK 158
:GDTRACK 159
Utility Functions 159
:MAP 159
:LMAP 161
:SMAP 161
:CLEAR 162
:COPY 162
:DPCOPY 162
:DUMP 162
:SAVE 163
:RESTORE 164

11. SYSTEM GENERATION 176

Overview 176
SYSGEN/SYSLOAD Flow 177
Loading Simulation Routines, RBM, and RBM Overlays 177
SYSGEN I/O 185
Rebootable Deck Format 185
Stand-Alone SYSGEN Loader 186

APPENDIXES

A. RBM SYSTEM FLAGS AND POINTERS 187

B. PAPER TAPE STANDARD FORMAT 192

C. ERROR LOGGING 193

D. XEROX STANDARD OBJECT LANGUAGE 201

Introduction 201
General 201
Source Code Translation 201
Object Language Format 202
Record Control Information 202
Load Items 203
Declarations 203
Definitions 205
Expression Evaluation 206

Formation of Internal Symbol Tables 209
Loading 210
Miscellaneous Load Items 211
Object Module Example 211

E. XEROX STANDARD COMPRESSED LANGUAGE 217

F. SYSTEM OVERLAY ENTRY POINTS 218

FIGURES

1. Initialize Routine Core Layout 1
2. RBM Initialize Routine Overall Flow 2
3. Overall IOCS Organization 11
4. IOCS: QUEUE Routine 13
5. IOCS: SERDEV Routine 15
6. IOCS: CLOCKIO Routine 17
7. IOCS: RIPOFF Subroutine 18
8. IOCS: STARTIO Routine 19
9. IOCS: IOINT Routine 21
10. IOCS: IOALT Routine 23
11. IOCS: CLEANUP Routine 24
12. IOCS: REQCOM Routine 26
13. IOCS: ENDAC Subroutine 28
14. IOCS: IOERROR Subroutine 29
15. IOCS: IOLOG Subroutine 30
16. IOCS: PUSHLOG Subroutine 31
17. JCP General Flow 42
18. JOB Command Flow 44
19. FIN Command Flow 45
20. ASSIGN Command Flow 45
21. DAL Command Flow 46
22. ATTEND Command Flow 46
23. MESSAGE Command Flow 46
24. PAUSE Command Flow 47
25. CC Command Flow 47
26. LIMIT Command Flow 47
27. STDLB Command Flow 48
28. NAME Command Flow 49
29. RUN Command Flow 51
30. ROV Command Flow 51
31. POOL Command Flow 51
32. ALLOBT Command Flow 52
33. LOAD Command Flow 53
34. PMD Command Flow 54
35. PFIL, PREC, SFIL, REWIND, and UNLOAD Command Flows 54
36. WEOF Command Flow 54
37. Core Layout During JCP Execution 55
38. Pre-PASS1 Core Layout 56
39. ARM, DISARM, and CONNECT Function Flow 62
40. Arrangement of SYSLOAD Input ROMs 65
41. ECB Format and Chained Data Areas 68
42. Relationship of Task Controlled Data 91
43. Relationship Between a Primary Task Control Block and Other Control Blocks 99
44. Relationship Between Secondary Task Control Block and Other System Control Data 102
45. Relationship of Job Associated Control Tables 105
46. Enqueue/Dequeue Table Relationship 110
47. Overlay Structure of the Overlay Loader 115
48. Overlay Loader Core Layout 116
49. LIB Reorganization of Dynamic Table Area 128
50. PASSTWO Reorganization of Dynamic Table Area 131
51. MAP Table Reference 133
52. Program File Format 136

53. Overlay Loader Flow, !OLOAD ____ 141
54. Overlay Loader Flow, CCI ____ 141
55. Overlay Loader Flow, PASSONE ____ 142
56. Overlay Loader Flow, PASSTWO ____ 145
57. Overlay Loader Flow, MAP ____ 147
58. Overlay Loader Flow, RDIAG ____ 148
59. Overlay Loader Flow, RDIAGX ____ 148
60. Overlay Loader Flow, DIAG ____ 149
61. RADEDIT Functional Flow ____ 151
62. Permanent Disk Area ____ 154
63. RADEDIT Flow, ALLOT ____ 165
64. RADEDIT Flow, COPY ____ 166
65. RADEDIT Flow, SQUEEZE ____ 171
66. RADEDIT Flow, SAVE ____ 173
67. RADEDIT Flow, RESTORE ____ 175
68. SYSGEN and SYSLOAD Layout Before Execution ____ 176
69. SYSGEN and SYSLOAD Layout After Execution ____ 177
70. SYSGEN/SYSLOAD Flow ____ 178

TABLES

1. ASSIGN Table ____ 41
2. Disk File Table Allocation ____ 78
3. DCT Subtable Formats ____ 79
4. IOQ Allocation and Initialization ____ 82
5. Overlay Loader Segment Functions ____ 115
6. T:DCBF Entries ____ 134
7. Background Scratch Files ____ 135
8. Standard SYSLOAD DEFs ____ 183
A-1. RBM System Flags and Pointers ____ 187

PREFACE

The primary purpose of this manual is to provide a guide for better comprehension of the program listings supplied with the Xerox Real-Time Batch Monitor (RBM) operating system. The programs and processors included are the System Generation program, the Monitor and its associated tasks and subprocessors such as the Job Control Processor, Overlay Loader, and RADEDIT.

The manual is intended for Sigma RBM users who require an in-depth knowledge of the structure and internal functions of the RBM operating system for system maintenance purposes. Since the RBM Technical Manual and program listings are complementary, it is recommended that the listings be readily available when this manual is used.

1. RBM INITIALIZATION ROUTINE

The RBM Initialize routine is entered from the disk bootstrap every time the system is booted from the disk, and it sets up core prior to the execution of RBM. It also modifies the resident RBM system (including all system tables), the RBM overlays, and the Job Control Processor. Modifications may be made from the C, OC, or SI device that is selected by a corresponding sense switch setting (1, 2, or 3). If sense switch 4 is reset, the Initialize routine loads all programs on the FP area of the disk designated as resident foreground into the foreground area. The Initialize routine extends 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 RBM). The Initialize routine terminates by triggering the RBM Control Task.

The general flow of the Initialize routine, from entry from disk bootstrap to triggering the Control Task interrupt, is illustrated in Figure 2.

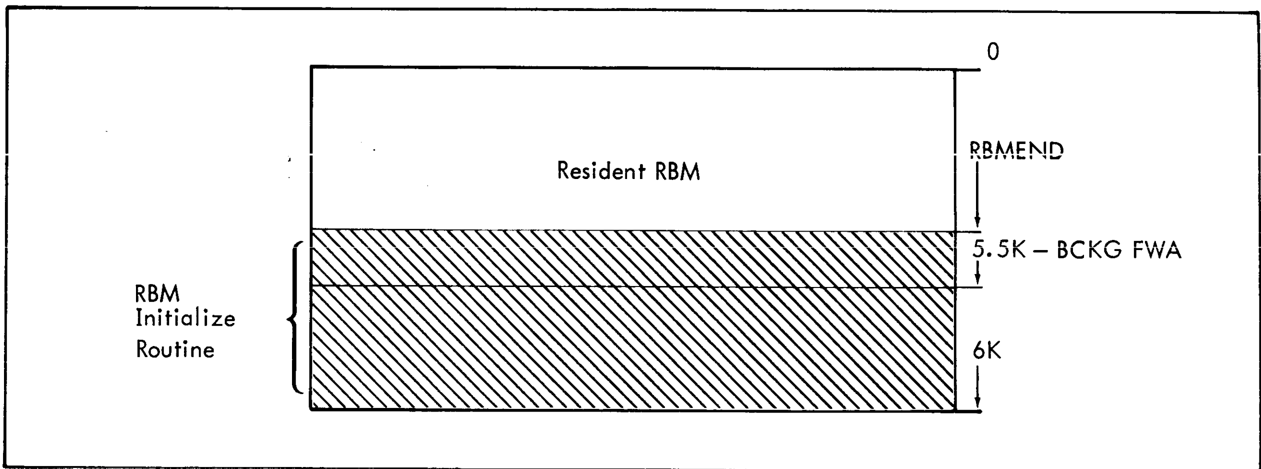


Figure 1. Initialize Routine Core Layout

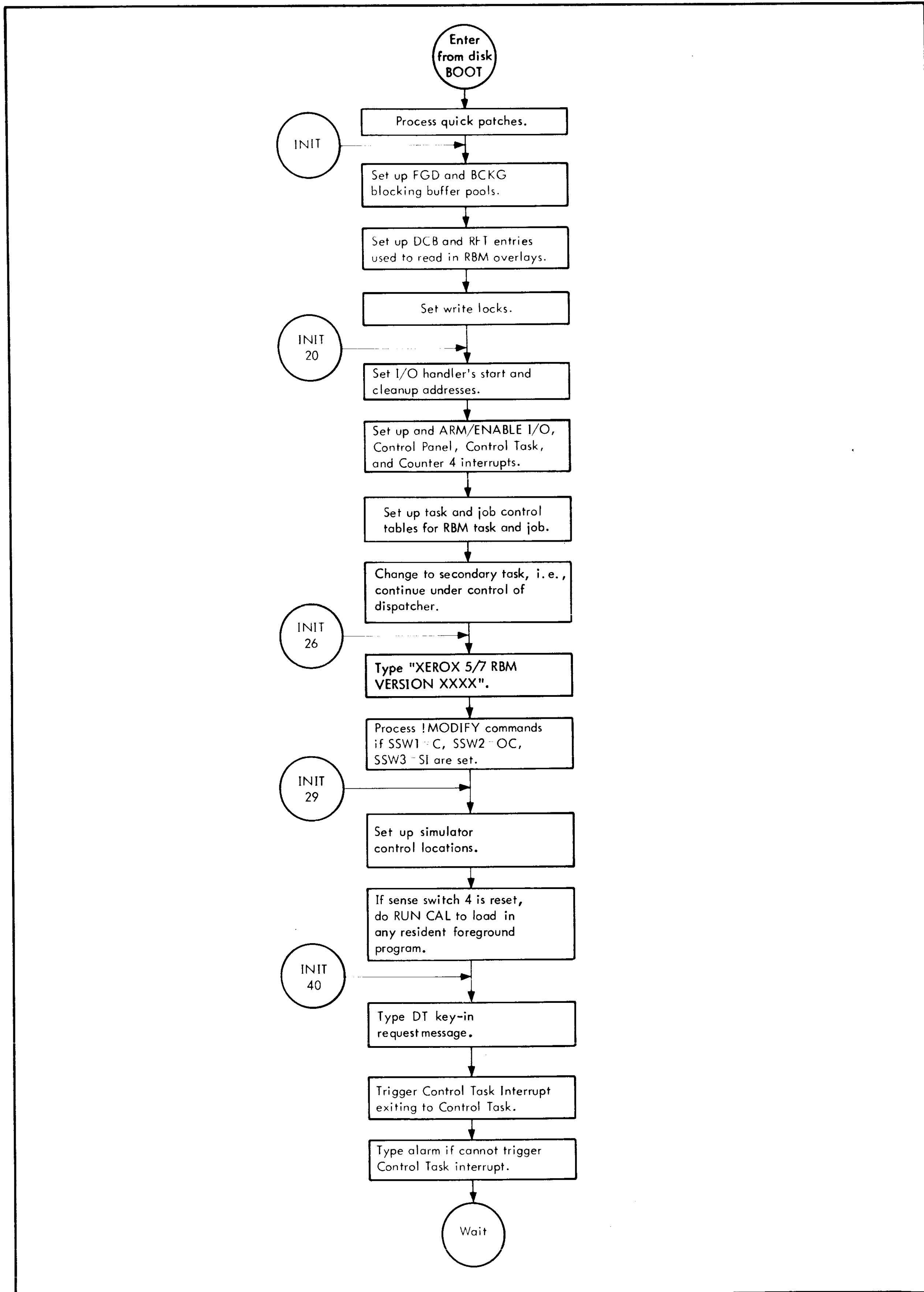


Figure 2. RBM Initialize Routine Overall Flow

2. RBM CONTROL TASK

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

- Key-in processing
- Foreground program "RUN" and "INIT"
- Foreground program "RELEASE" and "EXTM"
- Background program Load
- Background Checkpoint
- Background Restart
- Background Exit
- Background Abort
- Background Wait
- Background Postmortem Dump
- Keyin initiated dumps
- Deferred I/O processing
- Periodic service of all devices
- Crash data handling
- I/O error log handling

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)
- Checkpoint/Restart (CKPT)
- Background Abort/Exit (ABEX)
- Postmortem and Keyin Dumps
- Seven-Part Key-in Processor (KEY1-KEY7)
- Error logger (LOG)
- Error summary (ESUM)
- Crash saver (CRS)
- Crash-save dumper (CRD)
- Direct crash dumper part 1 (CKD1)
- Direct crash dumper part 2 (CKD2)

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 four 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 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 load modules to be released. If a releasable load module is found, FGL1 releases it. The 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 load modules, FGL1 also finds all 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 an 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. If the load module is foreground, its initialization sequence is executed. Normal completion posting is effected for the originating RUN or INIT request.

If FGL3 is unsuccessful at loading a foreground program 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, or the task was background, its tables are deleted, and the originating request is posted as abnormally completed.

FGL3 acquires dynamic memory for load module header input, and for background load modules reserves a blocking buffer as well (background headers may be as large as a full sector). 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. If it is, the module must also load into the correct area of memory (background or foreground). If it does not, the load is again terminated unsuccessfully, but if a foreground load module is concerned, checkpoint is requested. The root segments of the load module are read into their execution locations. If any PUBLIB is required, the LMI is searched. If the PUBLIB has no LMI entry, it must be loaded. Its header is read. From header data, it is determined if the PUBLIB loads into the foreground area, and does not overlap an existing program or PUBLIB. If these conditions are met, the PUBLIB is loaded and given an LMI entry. If not, the loading of the original program load module is terminated unsuccessfully. If a PUBLIB is already loaded, its use count is incremented.

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, or if the background is checkpointed. In the former case, the flag in K:CTST for BKL1 execution is cleared. At this point, Background Sequencing has terminated. In the latter case, the flag is not cleared so that the Control Task will continue to enter BKL1 until the checkpoint condition is cleared allowing BKL1 to proceed.

When BKL1 is called to initiate either JCP or another background program, the general process is to delete the background blocking buffer pool, reset the background-foreground boundary, reallocate the background buffers, 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 Load Module Control to do the task initiation.

The resetting of the background-foreground boundary is done if an FMEM key-in has been received that changes the boundary, and no foreground program or PUBLIB would lie in background as a result of the change. The change is effected by altering the boundary address pointer (K:FGDBG1) and resetting the write locks. If the change cannot be made because of existing programs in the foreground, the FMEM request is deleted and a message is sent to the OC device, but background initiation is attempted anyway.

The reallocation of buffers before initiating the background task provides a fixed number of blocking buffers (two) for use during initiation processing. Additionally, if a load module other than JCP is being initiated, the contents of the control command buffer and the ASSIGN buffer are moved from the locations they occupied when the prior background task (JCP) terminated. Note that if the background-foreground boundary is changed, BKL1 must not exit until it has performed these buffer moves, or Load Module Control could load over the old buffers, destroying the data needed.

The final path through BKL1 is taken after completion of the INIT service requested in either of the first two paths. Load Module Control, 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, ASSIGNs are done as indicated in the ASSIGN table. If the background program was not JCP, blocking buffers are reallocated either according to a POOL command if one was received; or to the number of blocked files indicated in the program's DCBs, if possible; or as few as one, if memory space is not adequate for the default number. If JCP is being initiated, it keeps the two blocking buffers allocated before the INIT request. BKL1 then zeros unused background memory, 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 in K:JCP indicating whether the termination is an exit or an abort. The Control Task is then triggered.

If ABEX is entered while the background is checkpointed, it exits immediately, so that – like BKL1 – it is reentered at each pass through the Control Task until the checkpoint is cleared.

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 sets the flag in K:CTST to run the PMD overlay, resets its own flag, and exits. When the dump is complete, PMD will set the ABEX flag to allow ABEX to finish. 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.

Checkpoint/Restart (CKPT)

This overlay performs both the Checkpoint and Restart functions. Checkpoint is accomplished by waiting for outstanding background I/O requests to run to completion and then writing the entire background portion of core to the CK area of the RAD. When the background has been successfully written to the RAD, the message

!!BCKG CKPT

is output on OC. At conclusion of the checkpoint, the background portion of memory is given to the foreground by setting the boundary pointers K:FGDBG1 and K:BCKEND and setting the Write locks appropriately.

The following self-explanatory messages may be output during checkpoint:

!!CKPT ABORT, I/O HUNG

!!BCKG USED BY FGD

!!CK AREA TOO SMALL

!!I/O ERR ON CKPT

Restart is accomplished by resetting the boundary pointers K:FDGBG1 and K:BCKEND, and by resetting the Write locks to their precheckpoint settings. The message

!!BCKG RESTART

is output on the OC device and the control bits indicating that the background is checkpointed are reset (K:JCP1 bits 2, 3). Control is then transferred to the resident Control Task, and when all specified subtasks are completed, the Control Task will exit to the proper point in the background.

Control Task Dump

This overlay performs core dumps. Any Dump key-in requests in effect at entry are performed first, and when these are exhausted, the background PMD requests are satisfied (maximum of four ranges). K:JCP1 bit 6 is set prior to completing the ABORT/EXIT, and the PMD is then done. After the PMD is completed, the Control Task returns to the ABORT/EXIT overlay and completes background cleanup.

The dump format is either hexadecimal or optionally both hexadecimal and EBCDIC, with the registers being retrieved from their storage area and dumped as locations 0 through X'F'. Subroutines are included in the overlay that perform hexadecimal to EBCDIC conversion and move bytes into the print image.

After printing each line, control is returned to the resident Control Task to enable other subtasks to be performed without waiting for total completion of the dump. The resident Control Task returns control to PMD after performing any higher priority subtasks.

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 the devices 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 RBM 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 RAD 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).

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.

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/L !!yyndd 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

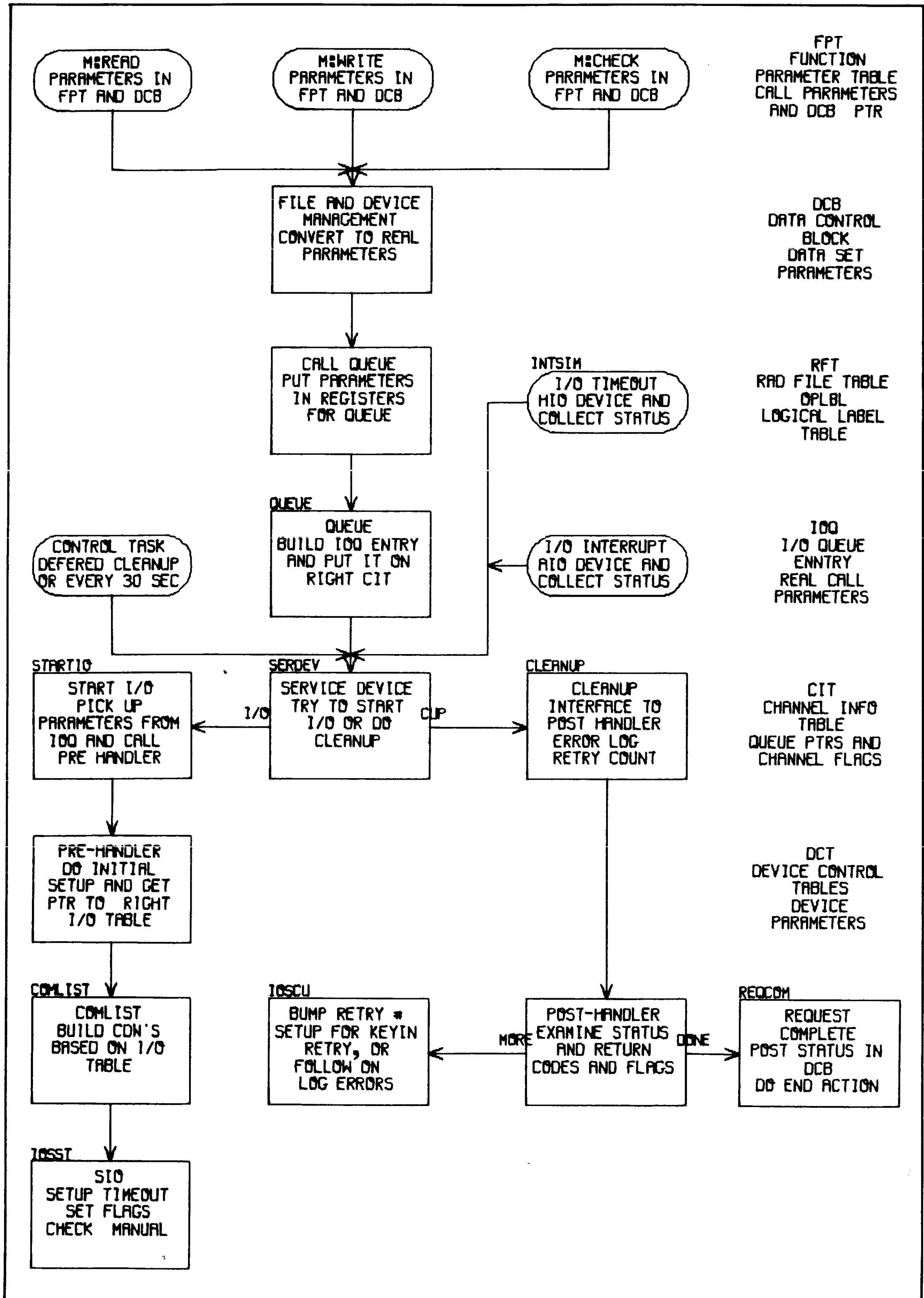


Figure 3. Overall IOCS Organization

Interfaces out of the IOCS

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

- o A data control block (DCB) may be posted with the actual record size (ARS) and type-of-completion code.
- o A post word may be posted with the ARS and type-of-completion code.
- o An external interrupt level may be triggered.
- o A user subroutine may be entered with the ARS and type-of-completion code in registers.

The last two options are only available for privileged, foreground, real-time tasks.

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 the system doesn't 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.

QUEUE

QUEUE ENTER AN I/O REQUEST

DEVICE IOEX DCT5 BIT7 SET — YES / NO

IS THIS AN IOEX Q REQUEST — YES / NO

DEVICE DOWN DCT3 BIT2 SET — YES / NO

$0047 PUSH 8,R0 → A2

EXIT FROM QUEUE (ERROR)

A2 → $0043 INITIALIZE R1 TO MAX DCT INDEX AND BACKGROUND PRIORITY

B2 → $0042 DISABLE

IS FREE IOQ CHAIN EMPTY — YES / NO

IS THIS THE LAST Q ENTRY — NO / YES

IS THIS AN OMAN REQUEST — NO / YES

$0042A IS REQUEST BACKGROUND — NO / YES

IS BKGRND IOQ CNT MAXIMUM — NO / YES

$0046 ENABLE PUSH 15,R4 → A3

A3 → SERDEV ATTEMPT TO DRIVE A REQUEST TO COMPLETION AND FREE A Q

PULL 15,R4 DECREMENT R1

IS R1 DCT PTR ZERO — NO → B2 / YES

$0045 INCREMENT BACKGROUND IOQ COUNT

$0040 DE-CHAIN IOQ ENTRY FROM FREE CHAIN

ENABLE

$0044 FILL-IN IOQ ENTRY FROM REGISTERS SET INITAL ACCESS KEY IN IOQ3

PUT BYTE CNT IN POST WORD IF FPT POSTING OR IN DCB-ARS IF DCB POSTING

$0049 DISABLE

IS CHANNEL Q EMPTY — YES → D7 / NO

$0053 IS REQ HIGHER PRIORITY — YES → C6 / NO

GET NEXT Q ENTRY

IS IT NUL — YES → D5 / NO

Figure 4. IOCS: QUEUE Routine

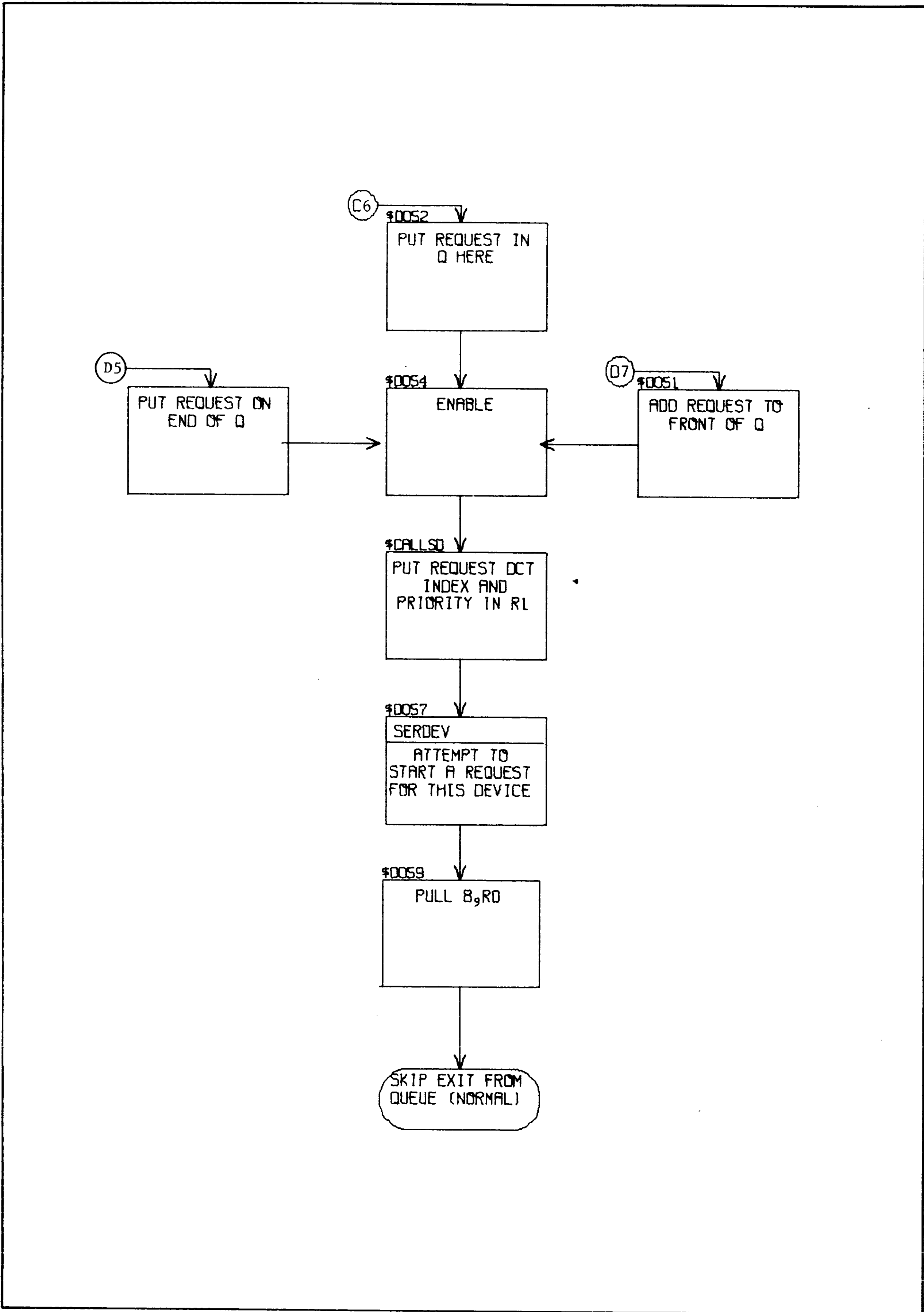


Figure 4. IOCS: QUEUE Routine (cont.)

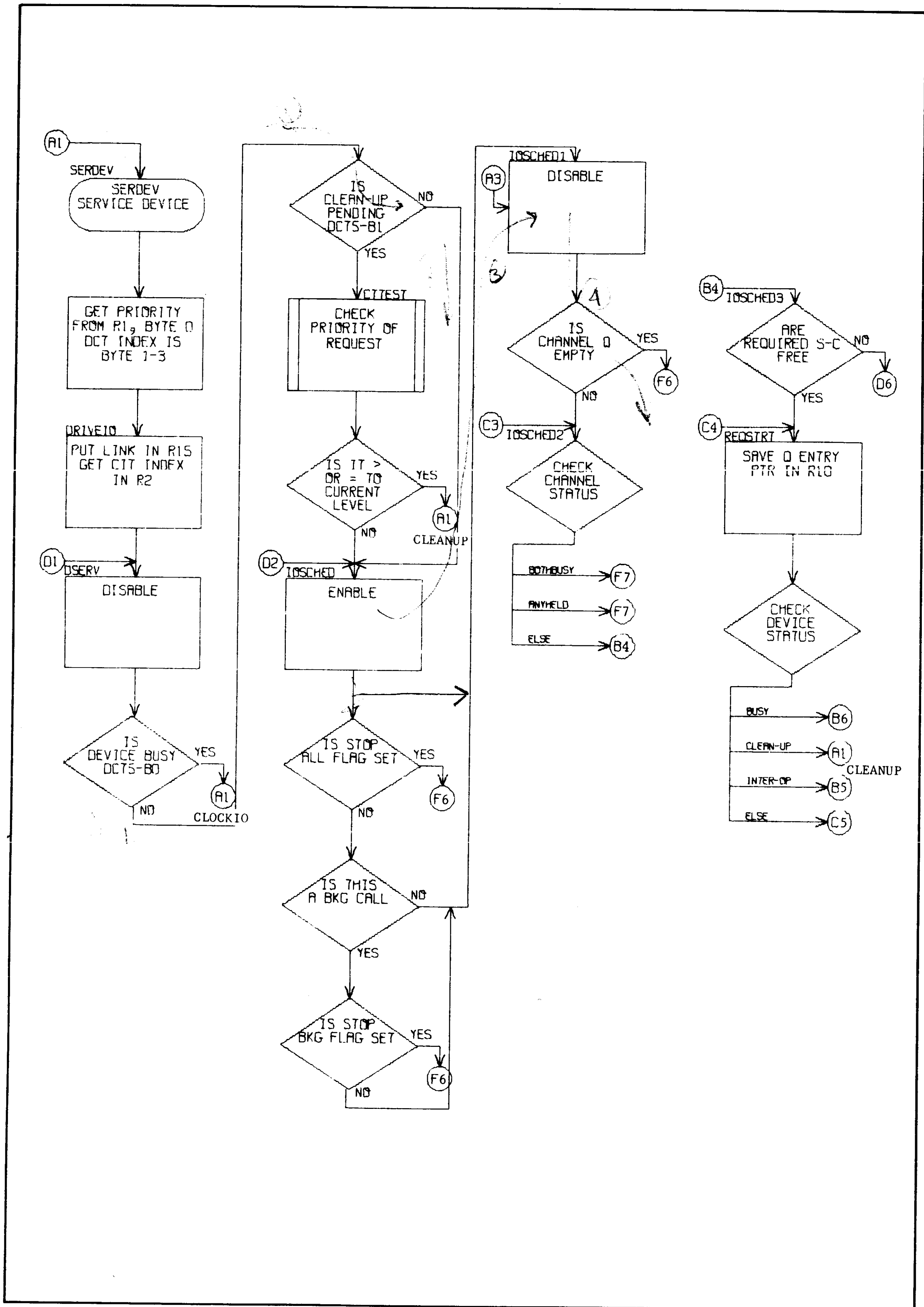


Figure 5. IOCS: SERDEV Routine

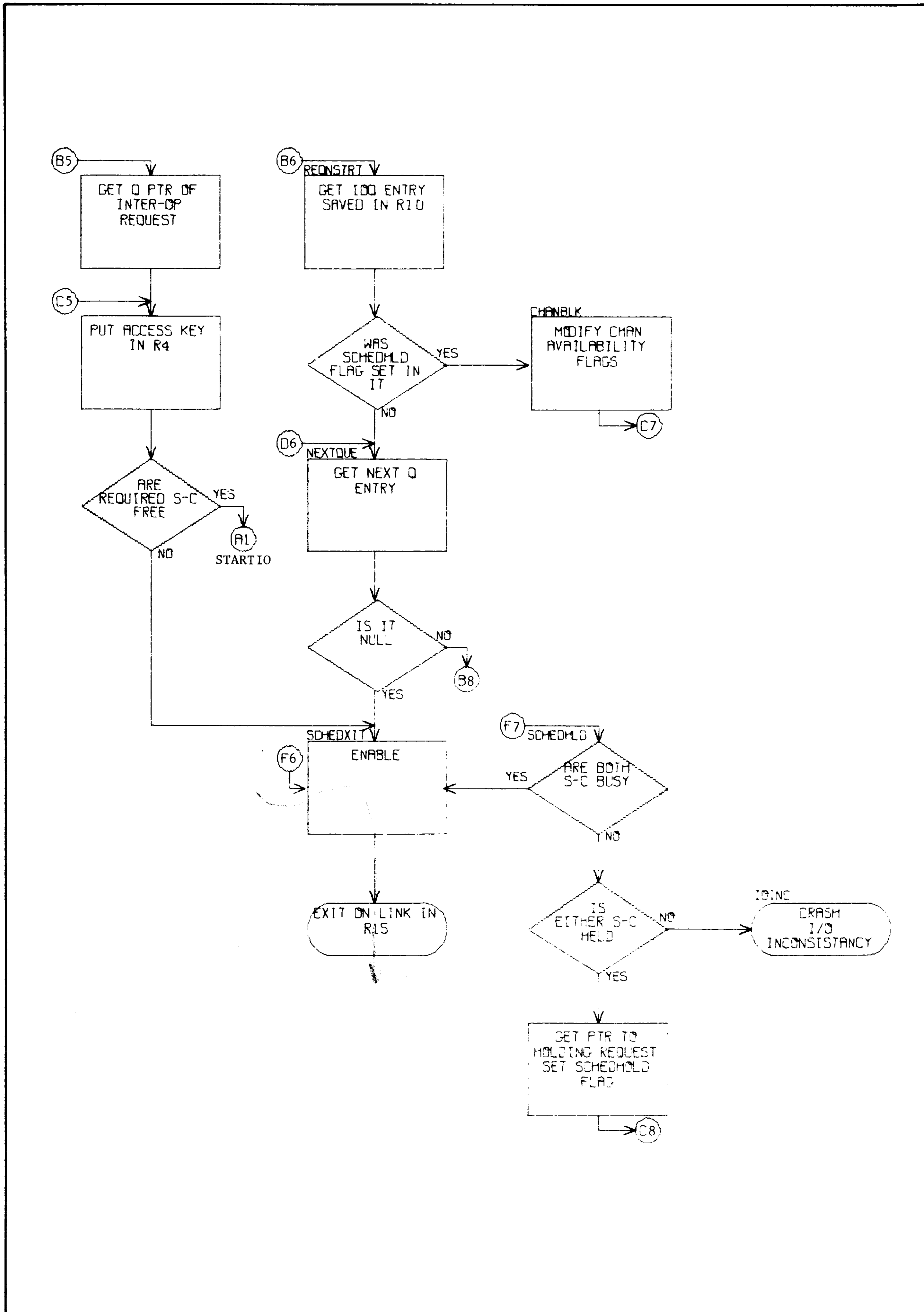


Figure 5. IOCS: SERDEV Routine (cont.)

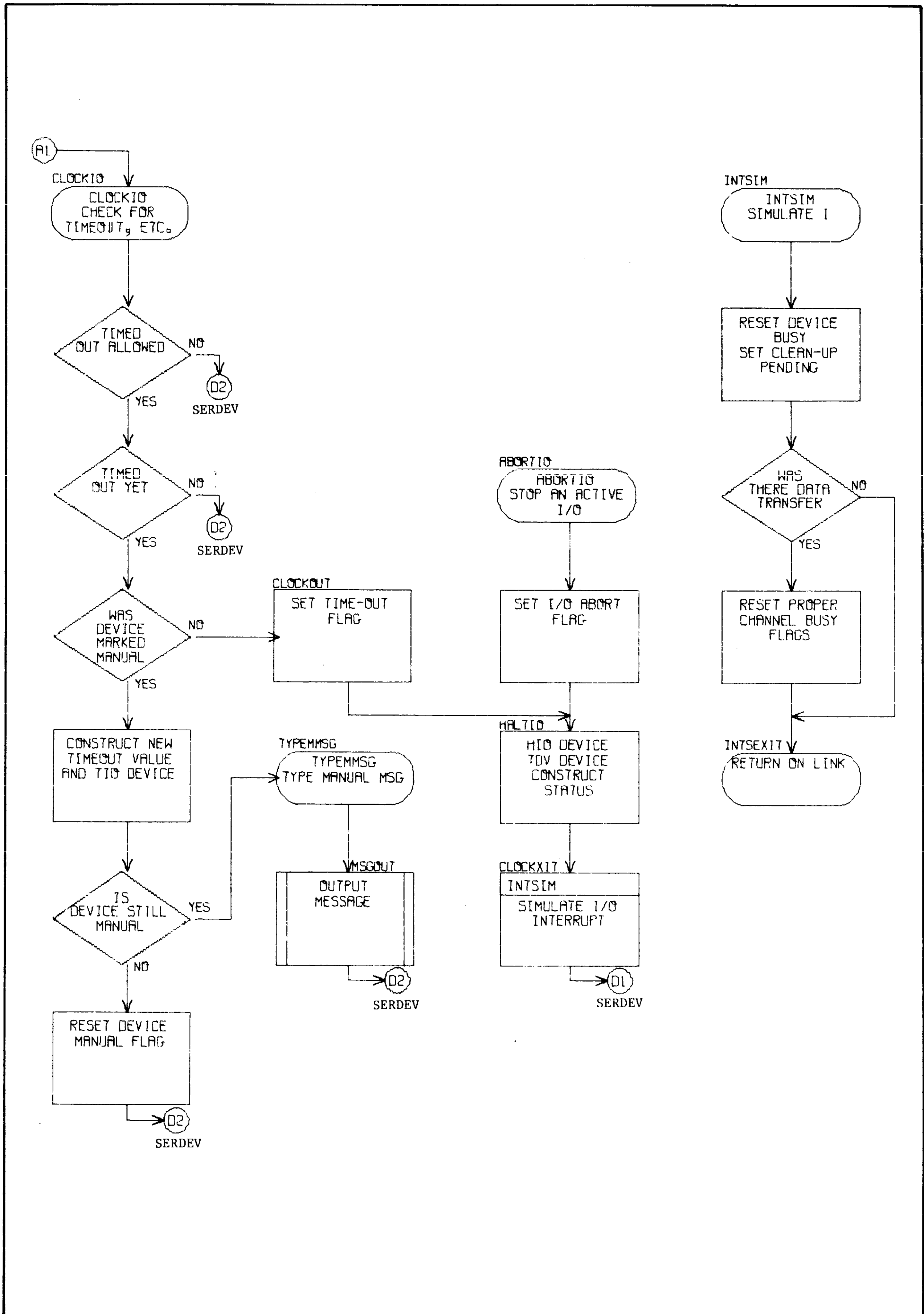


Figure 6. IOCS: CLOCKIO Routine

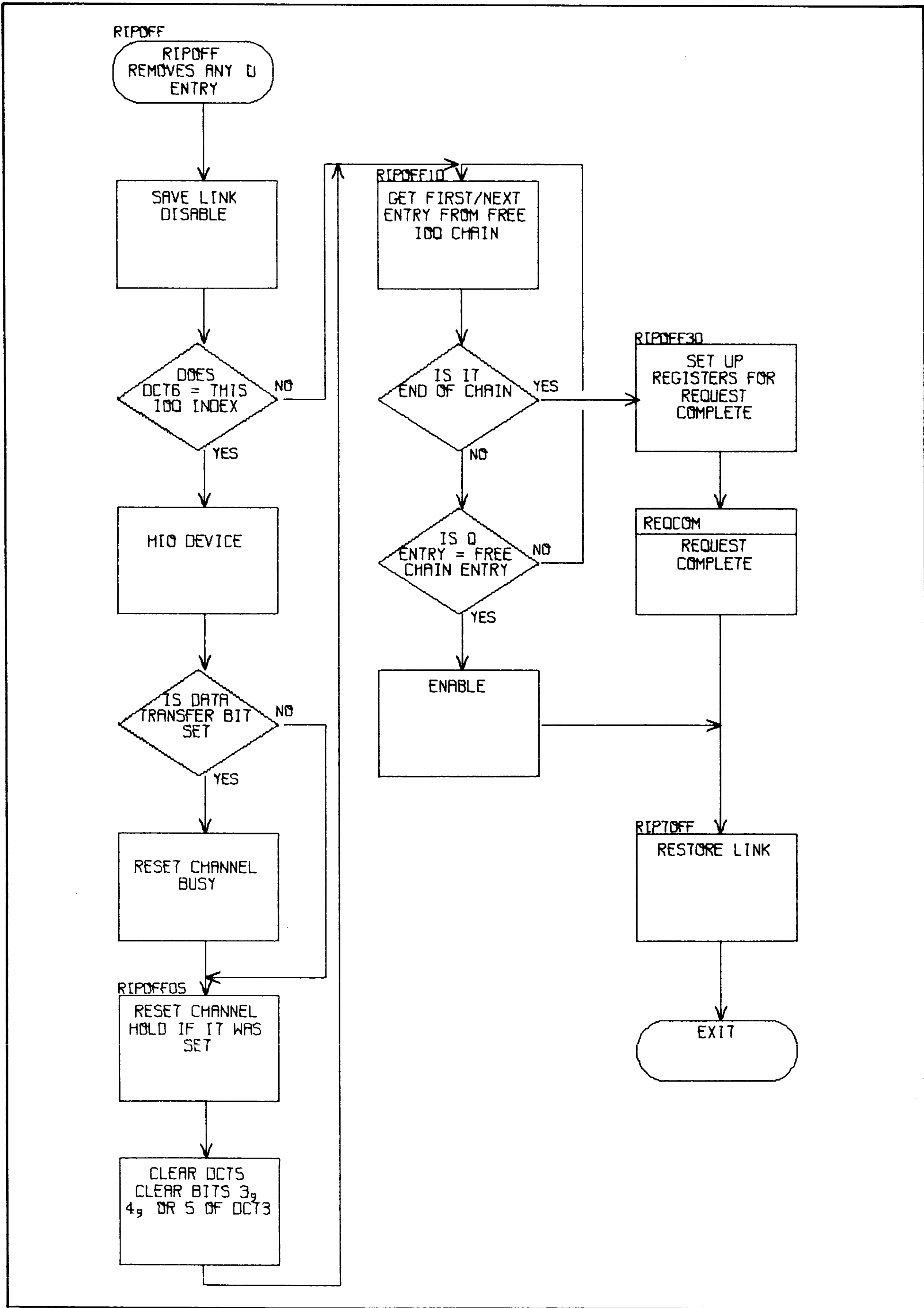


Figure 7. IOCS: RIPOFF Subroutine

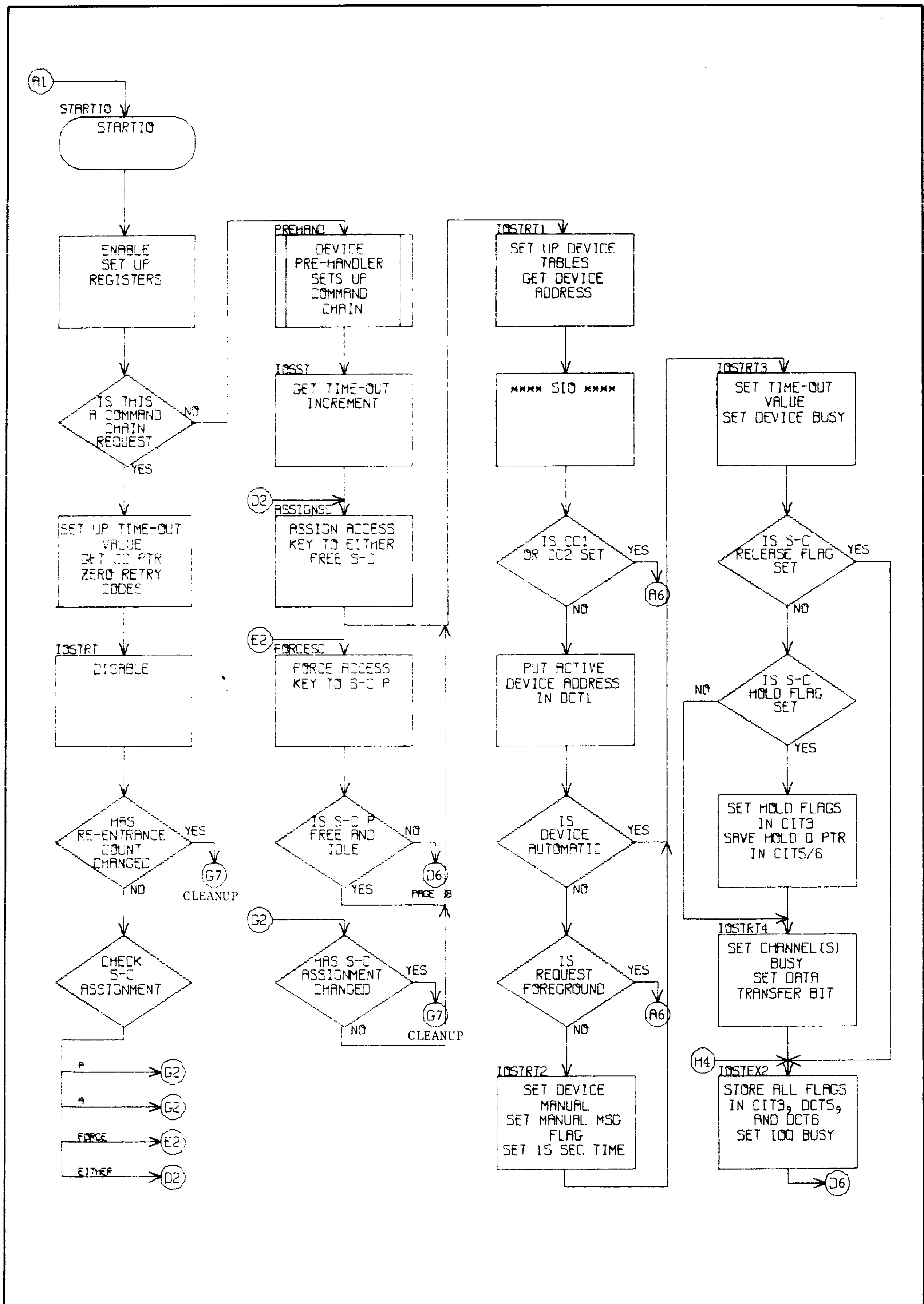


Figure 8. IOCS: STARTIO Routine

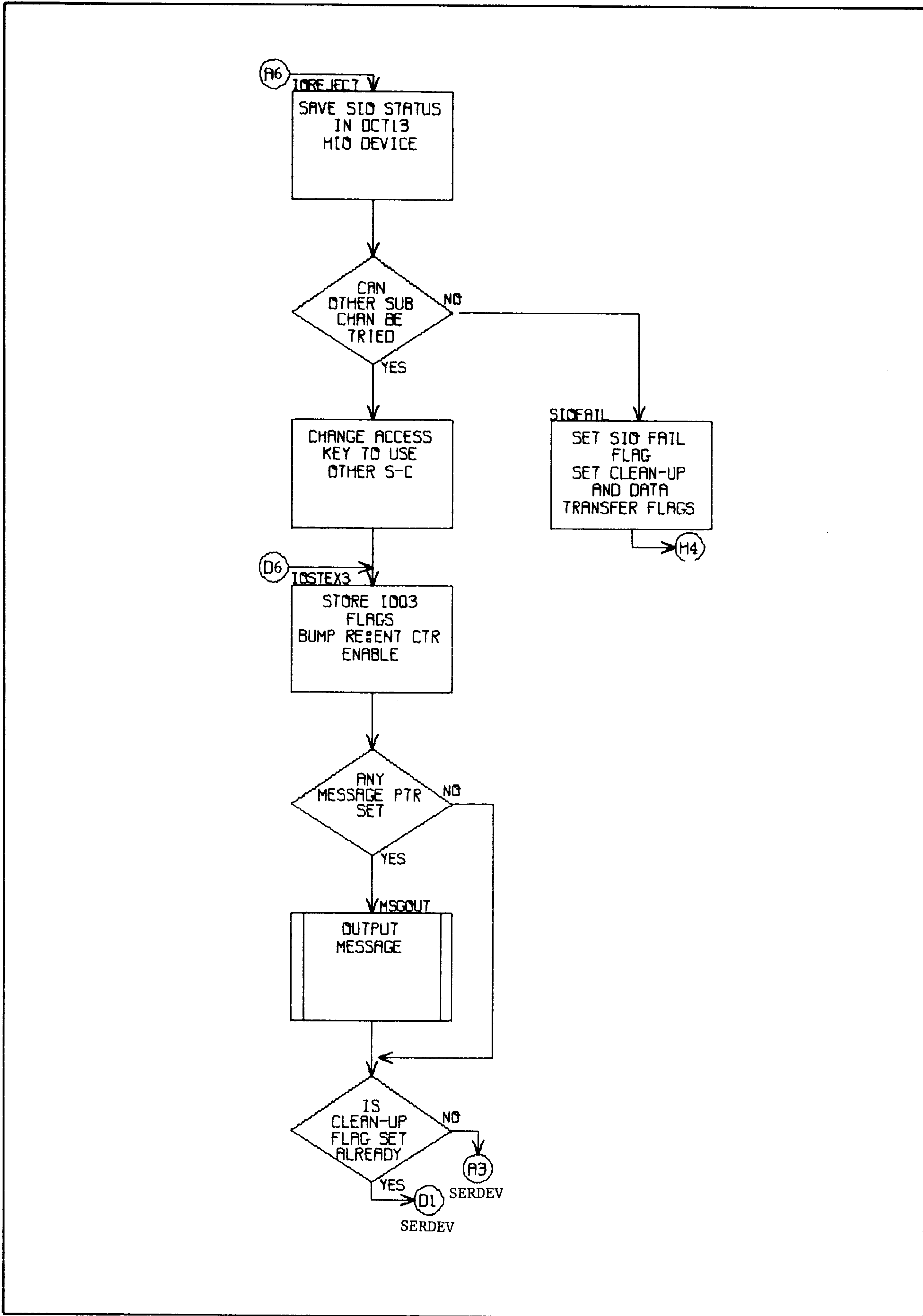


Figure 8. IOCS: STARTIO Routine (cont.)

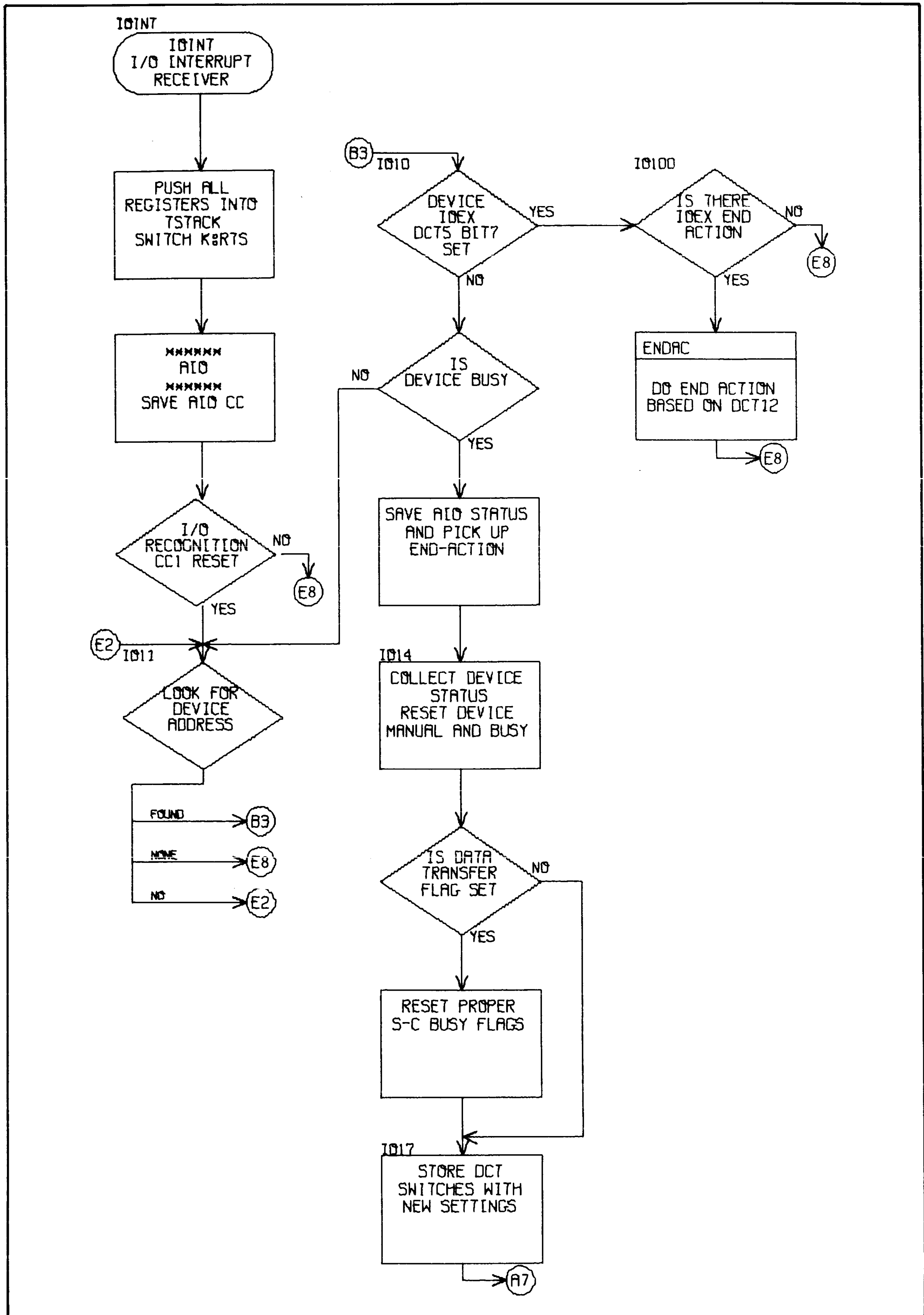


Figure 9. IOCS: IOINT Routine

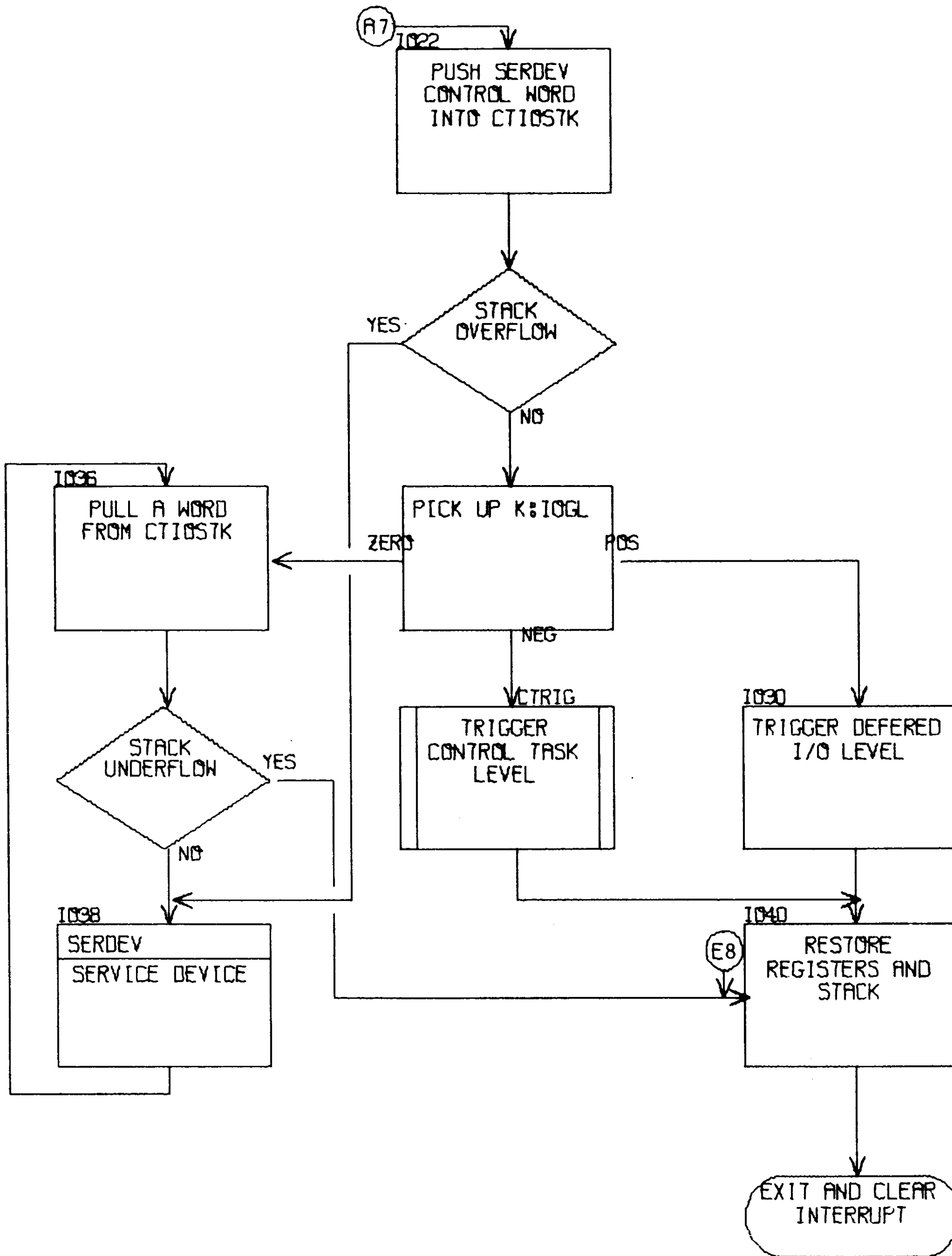


Figure 9. IOCS: IOINT Routine (cont.)

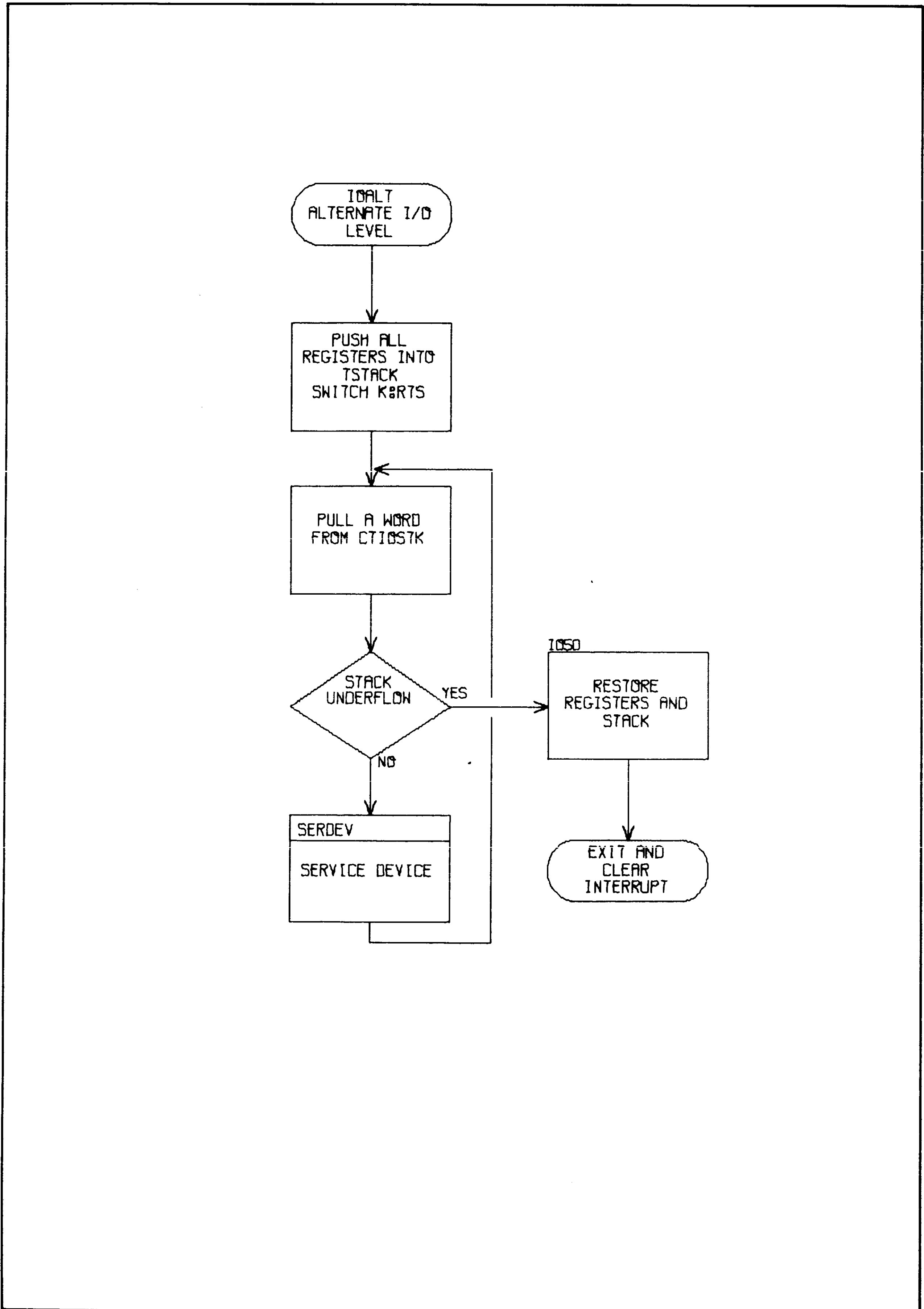


Figure 10. IOCS: IOALT Routine

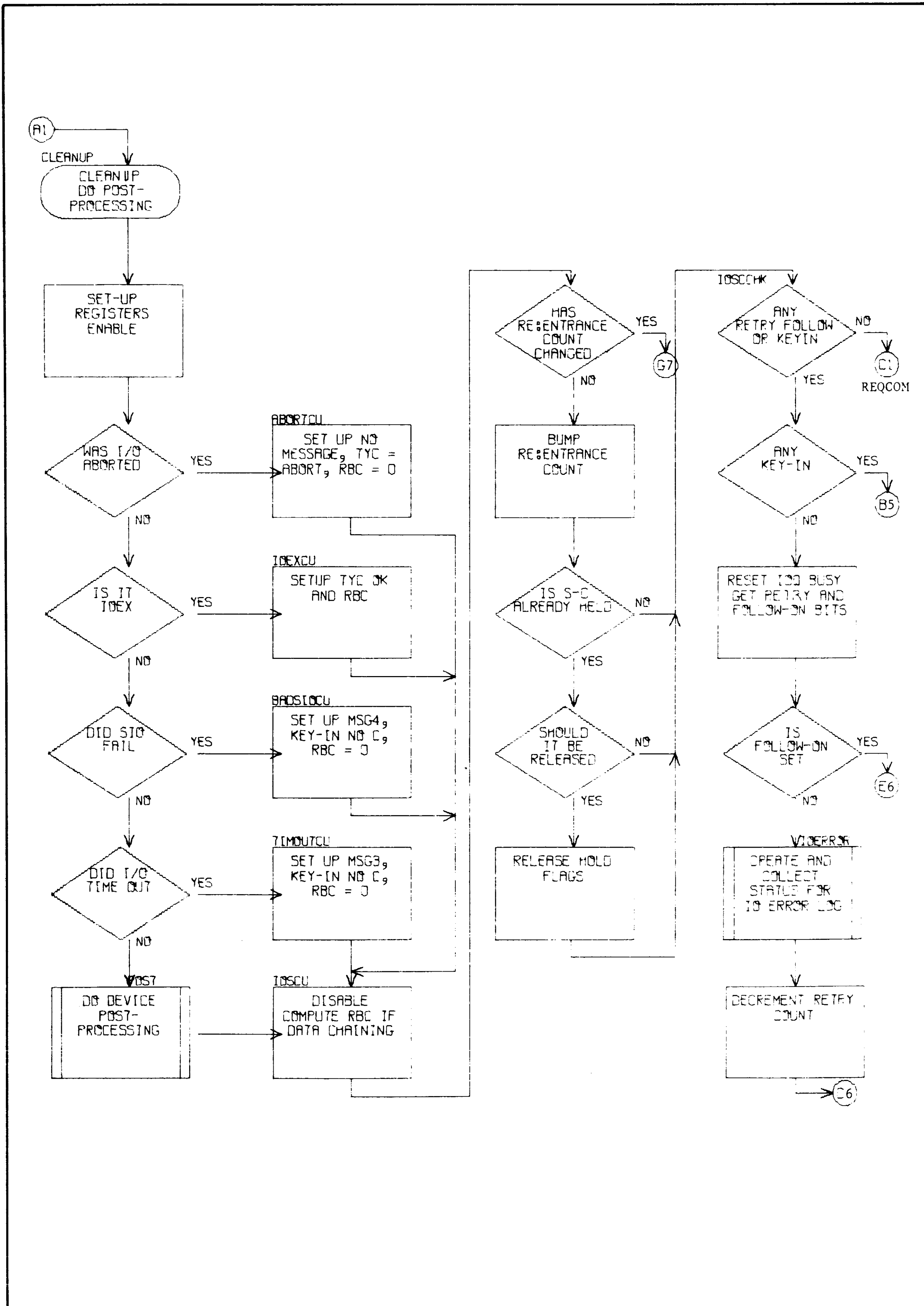


Figure 11. IOCS: CLEANUP Routine

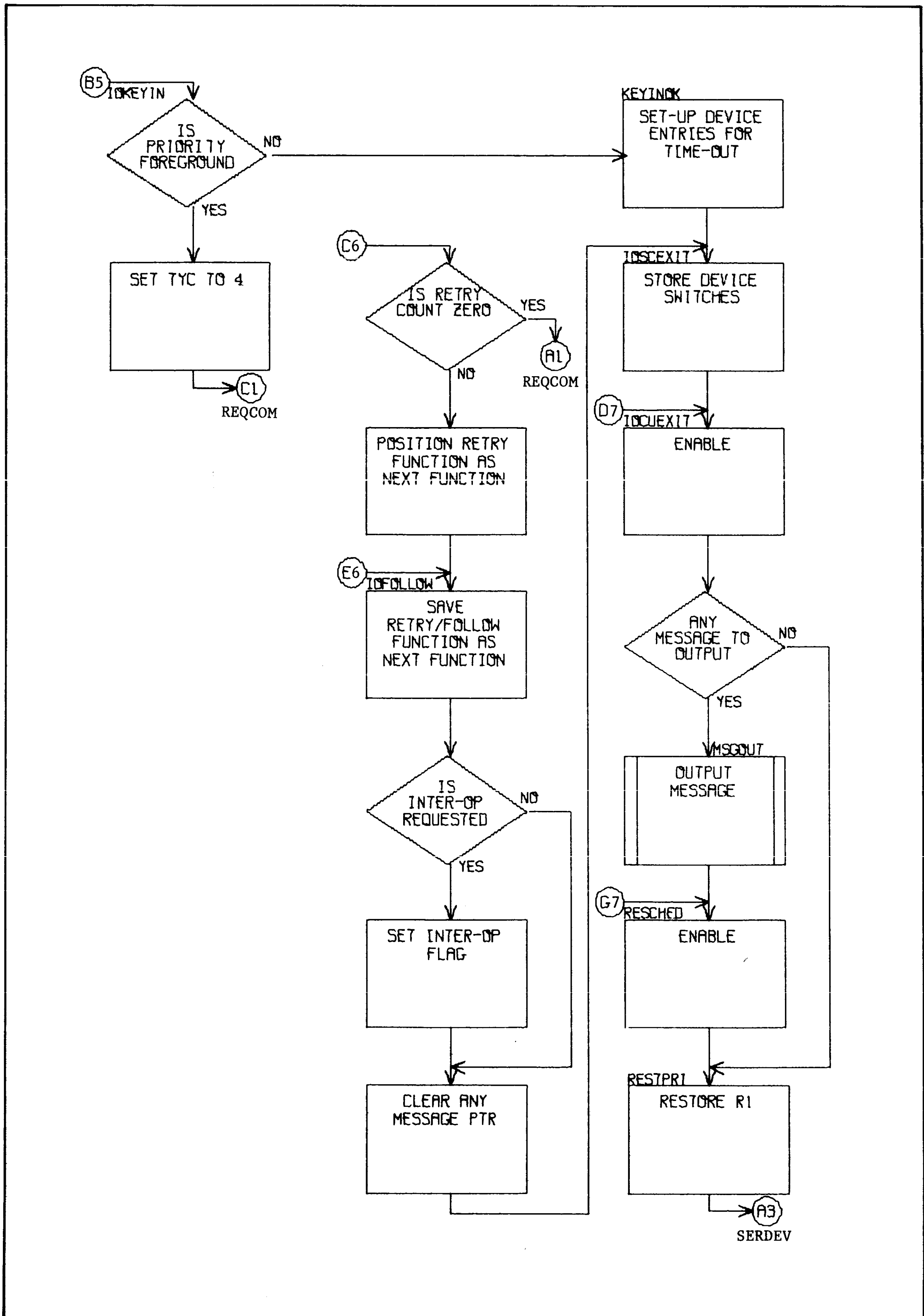


Figure 11. IOCS: CLEANUP Routine (cont.)

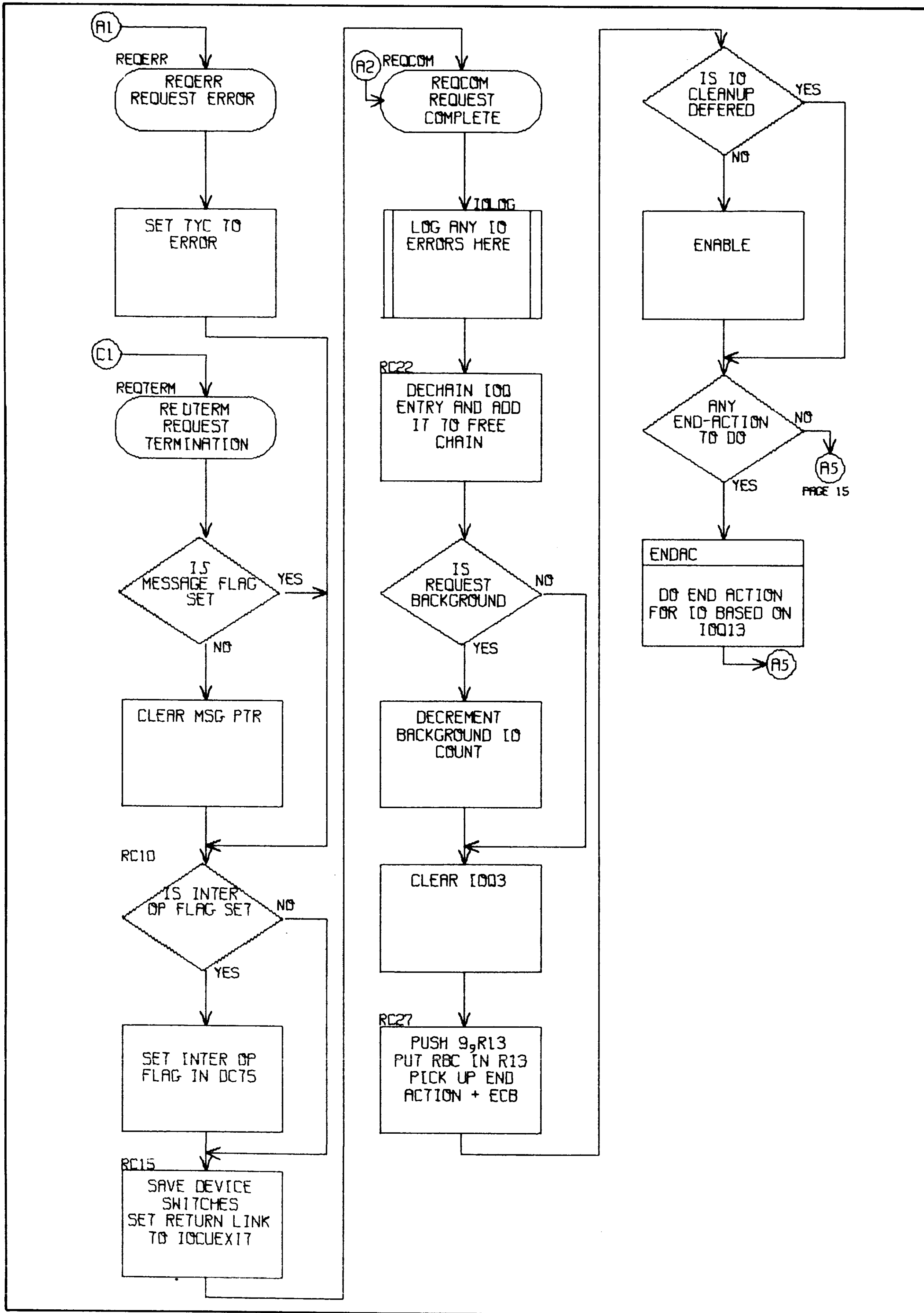


Figure 12. IOCS: REQCOM Routine

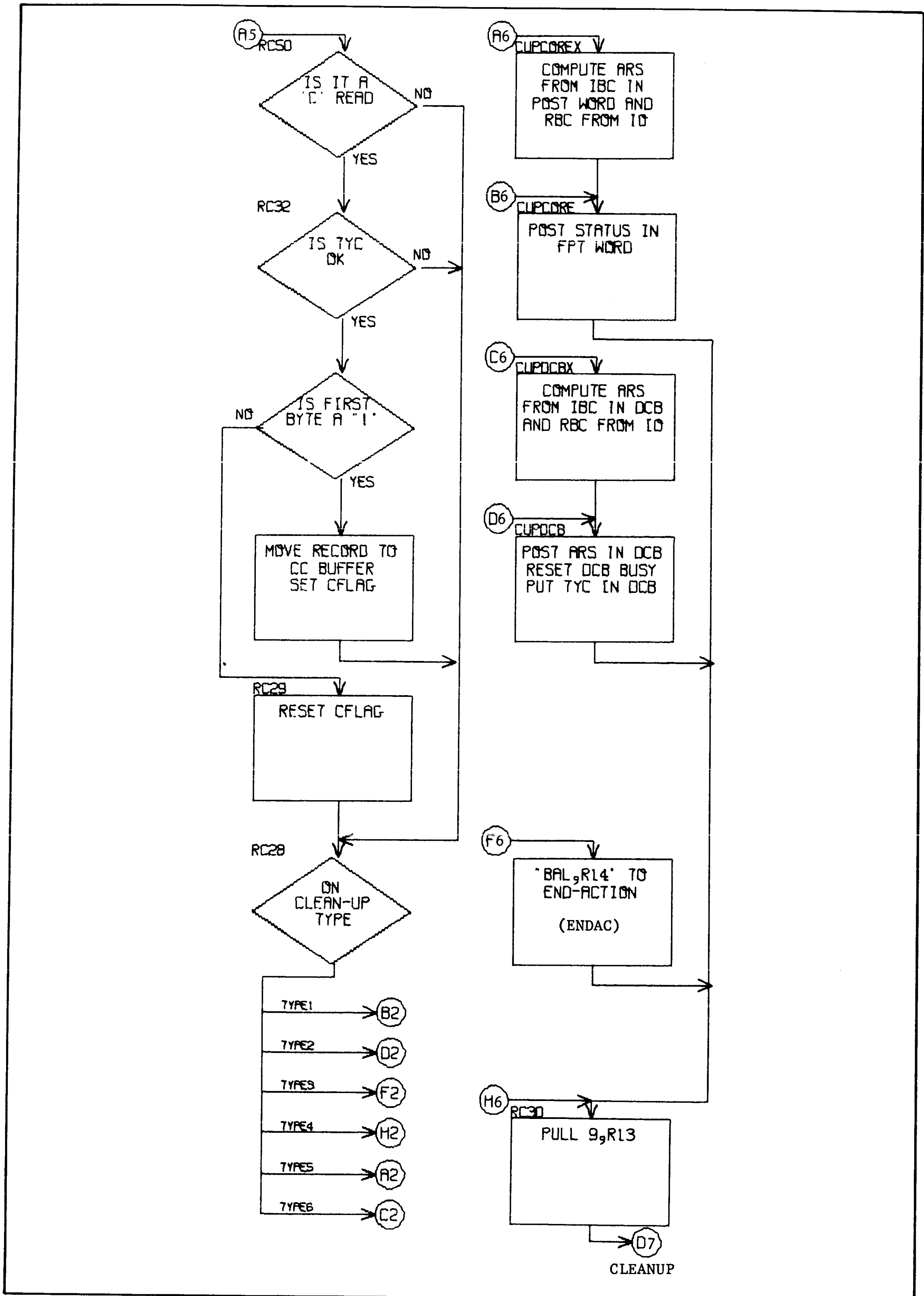


Figure 12. IOCS: REQCOM Routine (cont.)

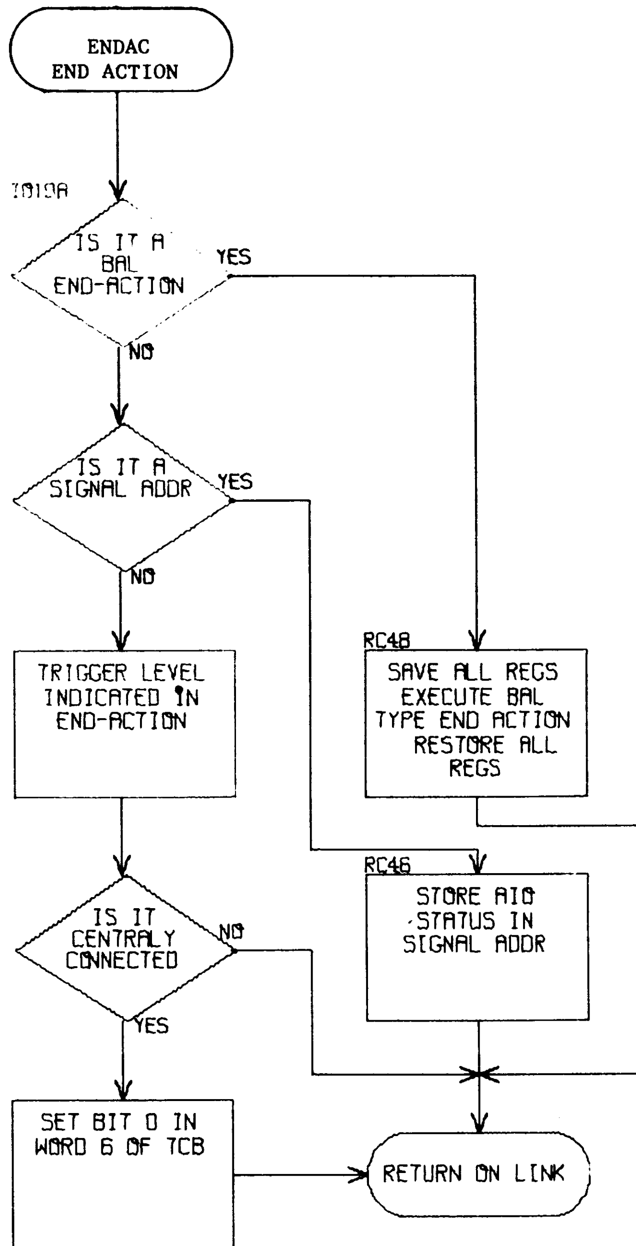


Figure 13. IOCS: ENDAC Subroutine

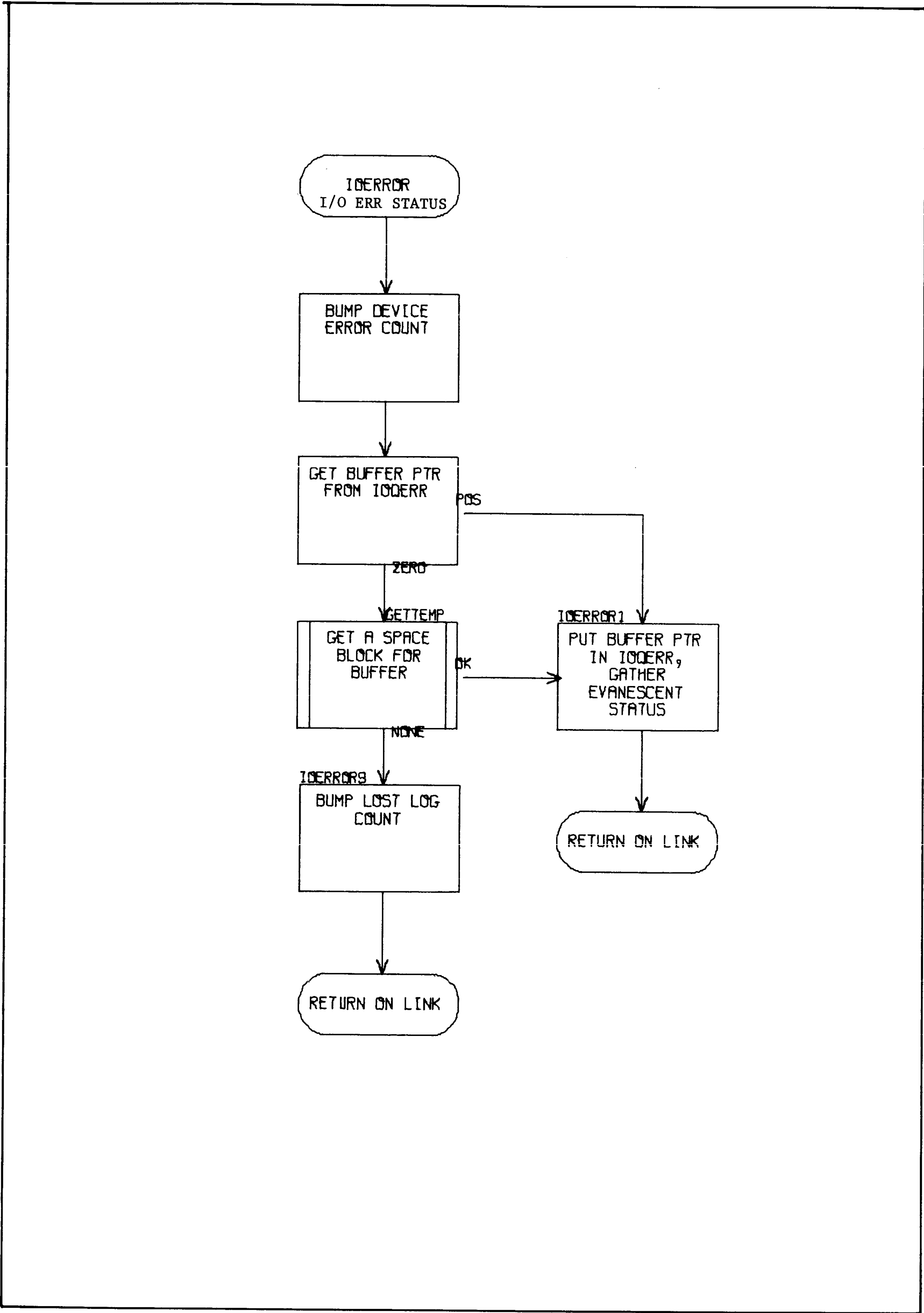


Figure 14. IOCS: IOERROR Subroutine

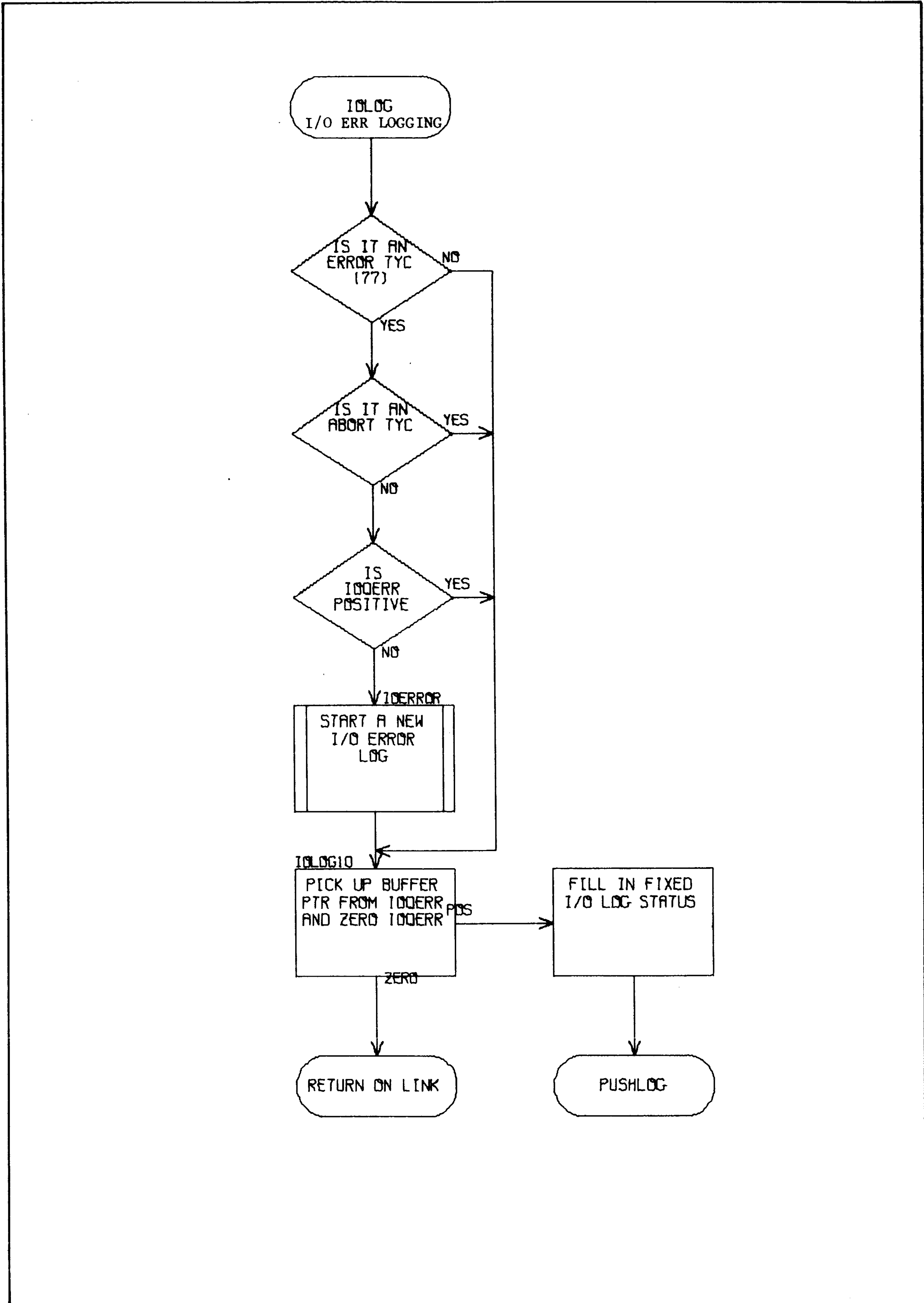


Figure 15. IOCS: IOLOG Subroutine

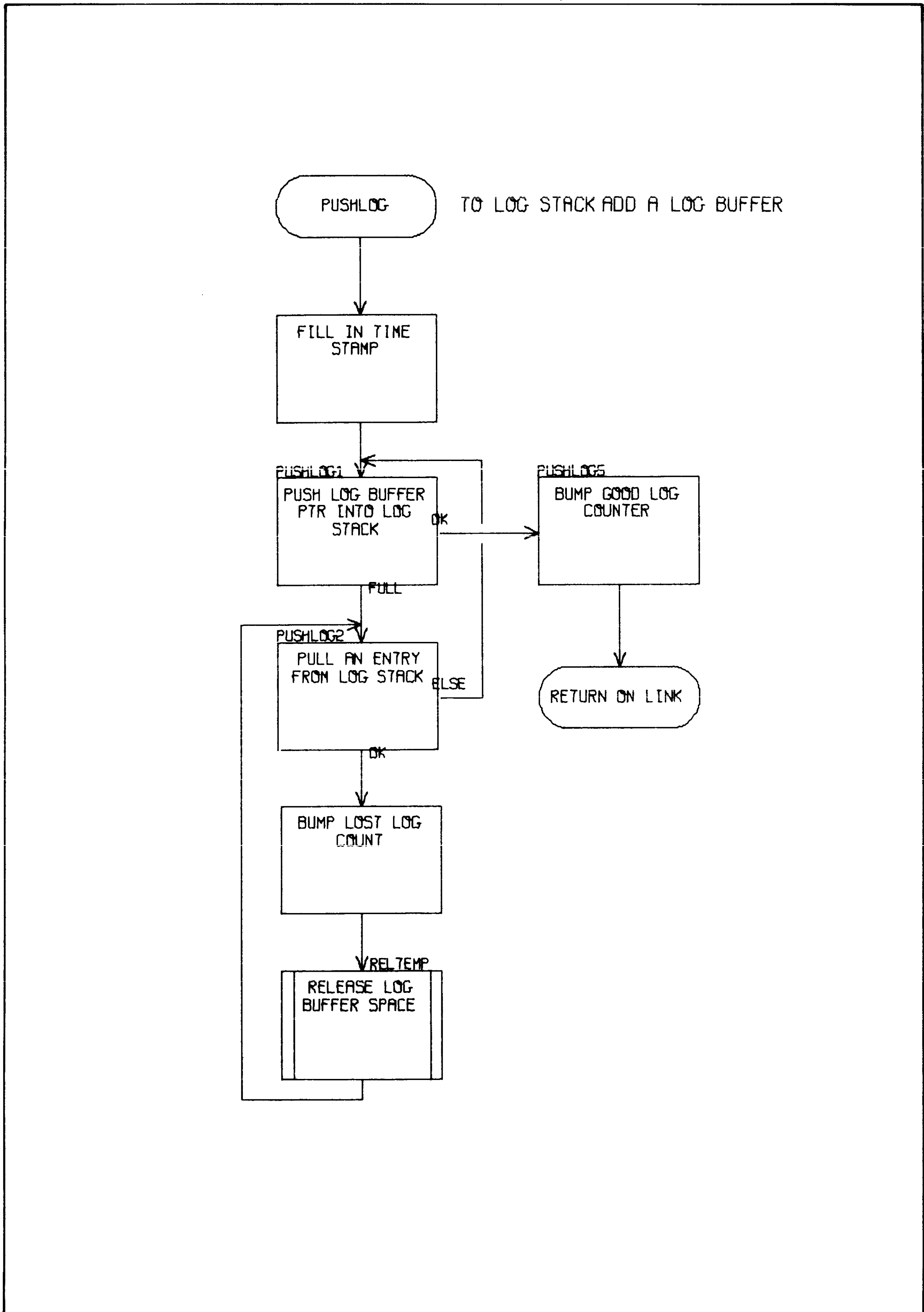


Figure 16. IOCS: PUSHLOG Subroutine

Register Conventions

QUEUE

Routine returns +1 if device IOEX, +2 otherwise

At entry:

R2	ECB ID
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	RAD seek address or number of records to pass (MT)
R13	Priority

Registers R0 – R7 preserved; R8 – R15 clobbered

CALLSD

At entry:

R1	FPI code
R2	DCB address
R3	FPI address
R5	Link

R1 – R7 preserved; R0, R8 – R15 clobbered

SERDEV

At entry:

R1	Bits 0–7 = priority Bits 8–31 = DCT index
R2	Link

All clobbered

RIPOFF

At entry:

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

All registers are clobbered.

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	Contrinue 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, R3, R4 set as for CLEANUP):

R1	DCT and priority
R3	IOQ pointer
R4	CIT pointer
R5	Link
R11	RBC
R12	TYC

R13 - R15, R0 - R4 preserved; R5 - R12 clobbered.

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 Appendix C "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 devices 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.

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 which cause an I/O transfer to cross one or more track boundaries will be broken into single-track operations. This is done within the disk pack pre- and post-device handler, and does not generate multiple IOQ entries.

There are three advantages to this method:

1. Long disk transfers by a lower priority task do not block a higher priority request more than one track time.

2. Flawed-track recovery is feasable, allowing alternate tracks to be assigned to damaged tracks.

3. Data transfers which cross cylinder boundaries are not allowed by the hardware. This problem is avoided by making only single-track transfers.

There is one disadvantage:

Because of processing time, the next-track operation cannot be begun in time not to lose a revolution between tracks.

Therefore, there is no time advantage in requesting more than one track of data per transfer.

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.

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-handler. 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.

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.

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.

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.

Background Blocking Buffers are handled dynamically. If a blocked I/O request is made and all allocated Background Blocking Buffers are in use by other files, one of the blocking buffers will be taken from its associated file

(after writing the block to the file, if necessary) and used for the current request. This blocking buffer is now associated with the file that most recently used it. When a request is made for I/O on the original file, the system recognizes that no Background Blocking Buffer is associated with the file and it will locate a buffer for this file by borrowing one from another file if necessary. One Background Blocking Buffer is sufficient for any background program.

Foreground Blocking Buffers are not handled dynamically.

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

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.

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.

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.

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 RAD 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) parameters.

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:

1. A magnetic tape 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.

2. Disk File Backward. File position (RFT11) is set to zero; the blocking buffer position (RFT10) is set to zero; the current record number is set to 1; and the blocking buffer control word address (RFT17) is set to zero.

3. Unblocked Disk File Forward. 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.

4. Blocked Disk File Forward. 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.

5. Compressed Disk File Forward. 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.

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

The new entry is added to the current sector of the directory (identification entry with A = 0) at the location specified by "address" in the identification entry. The BOT of the new entry is set equal to the "next available sector". The EOT is computed, using the FSIZE, RSIZE, and FORMAT parameters. The identification entry is updated to reflect the new entry. The "next available sector" is set = EOT of the new entry + 1, and "address" is incremented by 5.

If there is insufficient space in the current sector of the directory for another entry, "A" in the identification entry is set to 1; "address" is set = "next available sector" and that sector address is used for the new sector of the directory. A new identification entry is built by setting "A" = 0; "address" = 6; and "next available sector" = EOT of the new entry + 1.

If there is insufficient space to allocate to the file, the file directory is searched for deleted entries (file name = 0). The smallest deleted entry that frees sufficient space is selected for the new entry. Disk space is lost if the deleted entry frees more space than is required by the new entry. (This space can subsequently be made available for allocation 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 four words of the file directory entry are zeroed out. The last word of the entry (BOT and EOT) remains unaltered. 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 unused and is not reallocated until a RADEDIT :SQUEEZE command is executed.

4. 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 RAD. The JCP is loaded from RAD 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 RBM 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 5K of core to execute, which means that the smallest possible core space allocated to the background must be at least 5K. Approximately one third of the JCP code consists of the JCP Loader, which is used to load the Overlay Loader at System Generation time.

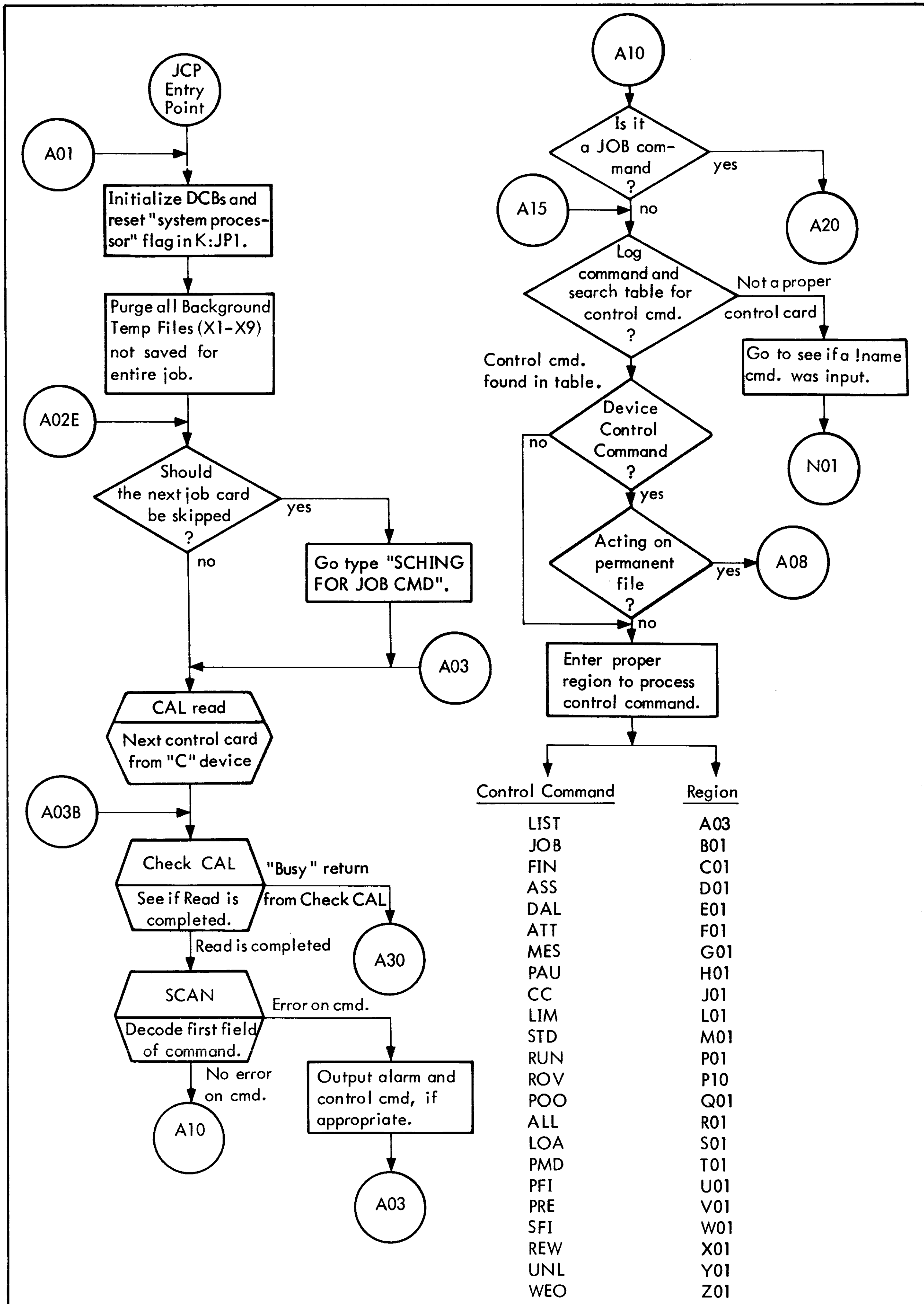
The flowchart illustrated in Figure 17 depicts the overall flow of the JCP, and Figures 18 through 36 illustrate the JCP commands. The labels used in the flowcharts correspond to the labels in the program listing.

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 background memory 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.

Table 1. ASSIGN Table

Words	Contents
1	Number of entries in table (each entry of ten words contains data from one !ASSIGN command). This word is always on an odd boundary; K:ASSIGN contains the address of word 1.
2,3	Name of DCB to change in EBCDIC. This pair of words and the next four pairs of words are on a doubleword boundary.
4	This word contains changes to the items in word 0 of the DCB.
5	Mask for items being changed in word 0. The Background Loader does an STS instruction (using words 4 and 5) to change the items in word 0 of the DCB.
6	Changes for word 1 of DCB.
7	Mask for items being changed in word 1.
8	Changes for word 3 of DCB.
9	Mask for items being changed in word 3.
10,11	File name in EBCDIC if DCB is assigned to a RAD file; otherwise, these words equal zero.
Words 2 through 11 contain one entry in the ASSIGN table and are repeated for each !ASSIGN command.	



Control Command	Region
LIST	A03
JOB	B01
FIN	C01
ASS	D01
DAL	E01
ATT	F01
MES	G01
PAU	H01
CC	J01
LIM	L01
STD	M01
RUN	P01
ROV	P10
POO	Q01
ALL	R01
LOA	S01
PMD	T01
PFI	U01
PRE	V01
SFI	W01
REW	X01
UNL	Y01
WEO	Z01

Figure 17. JCP General Flow

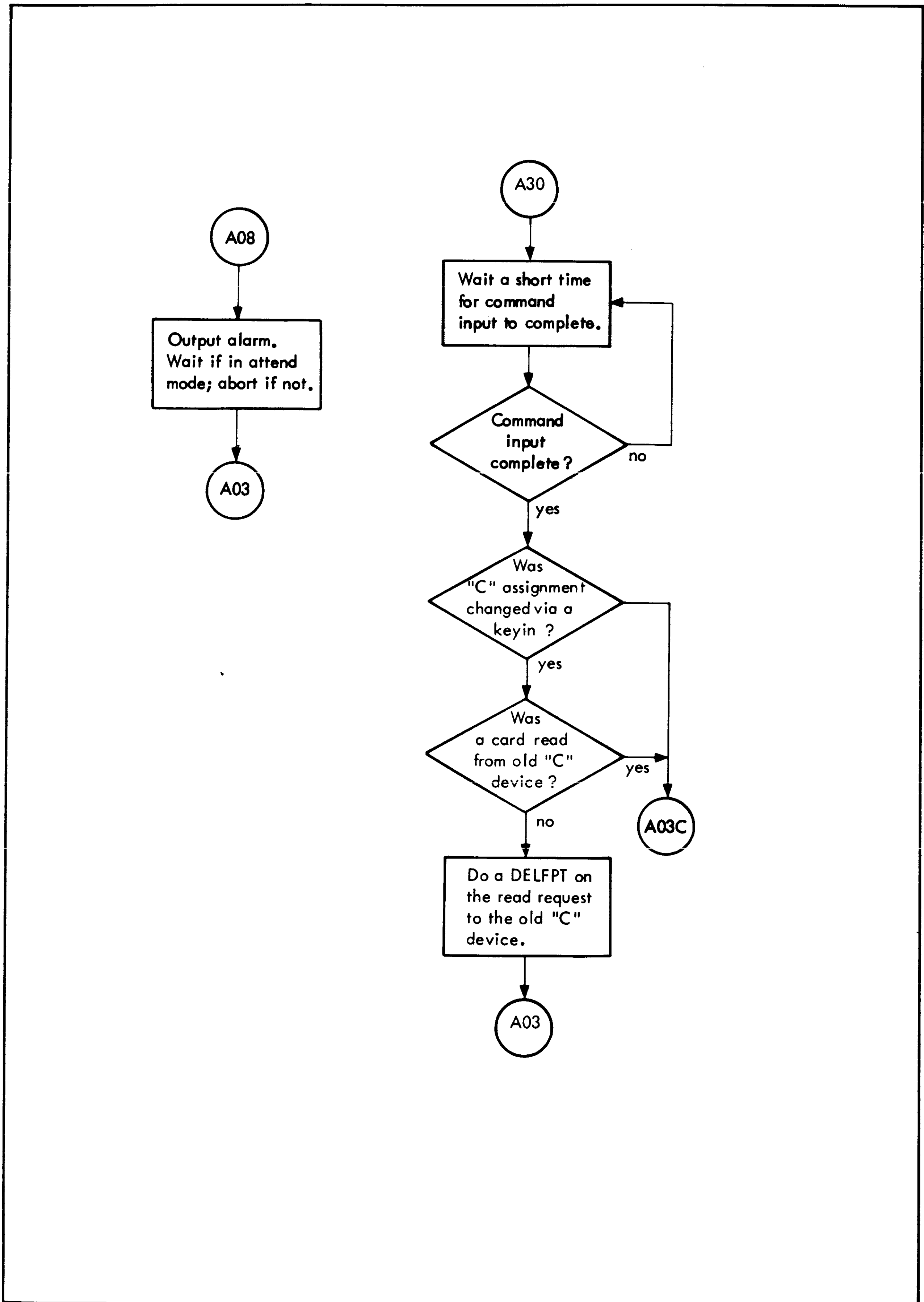


Figure 17. JCP General Flow (cont.)

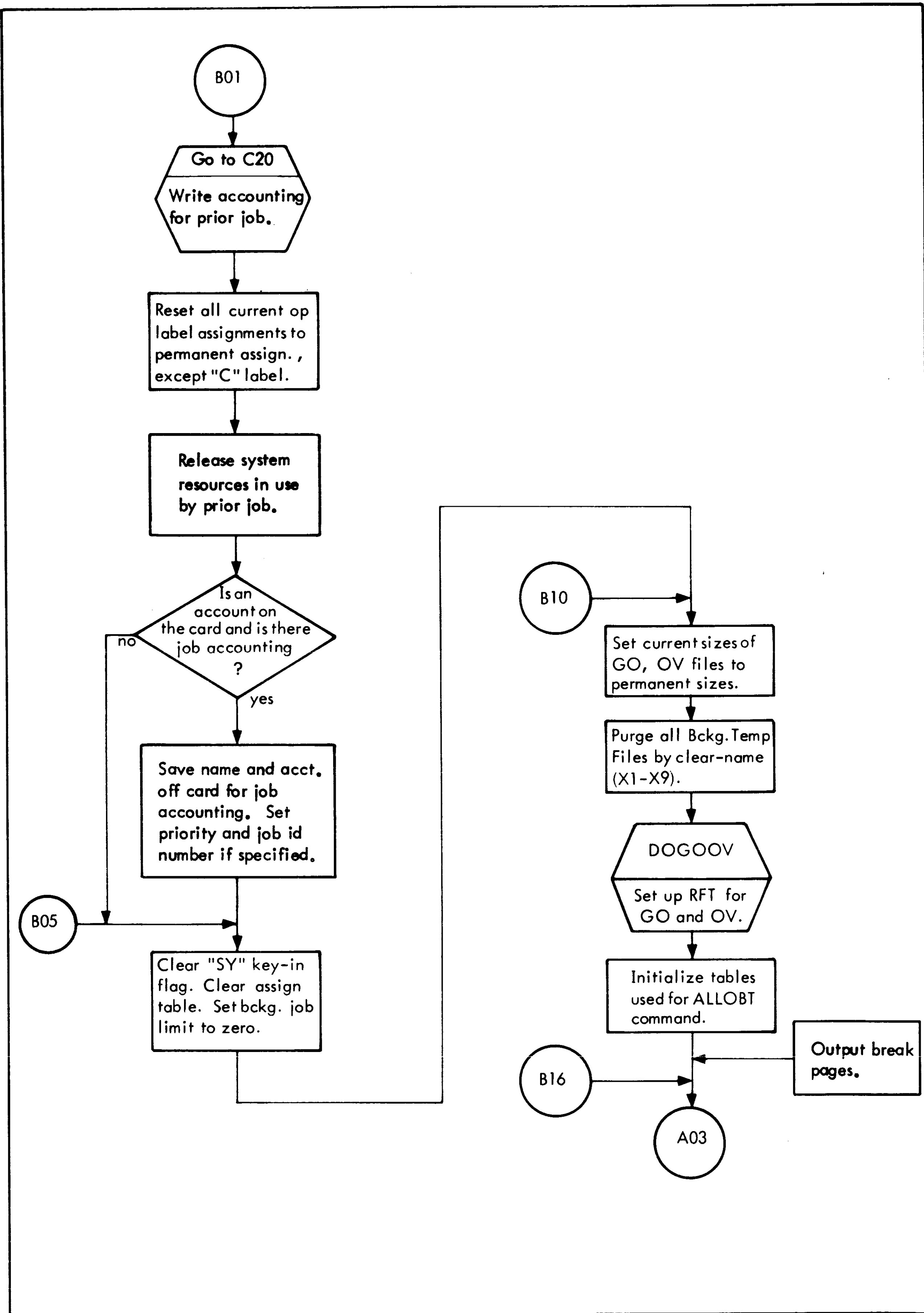


Figure 18. JOB Command Flow

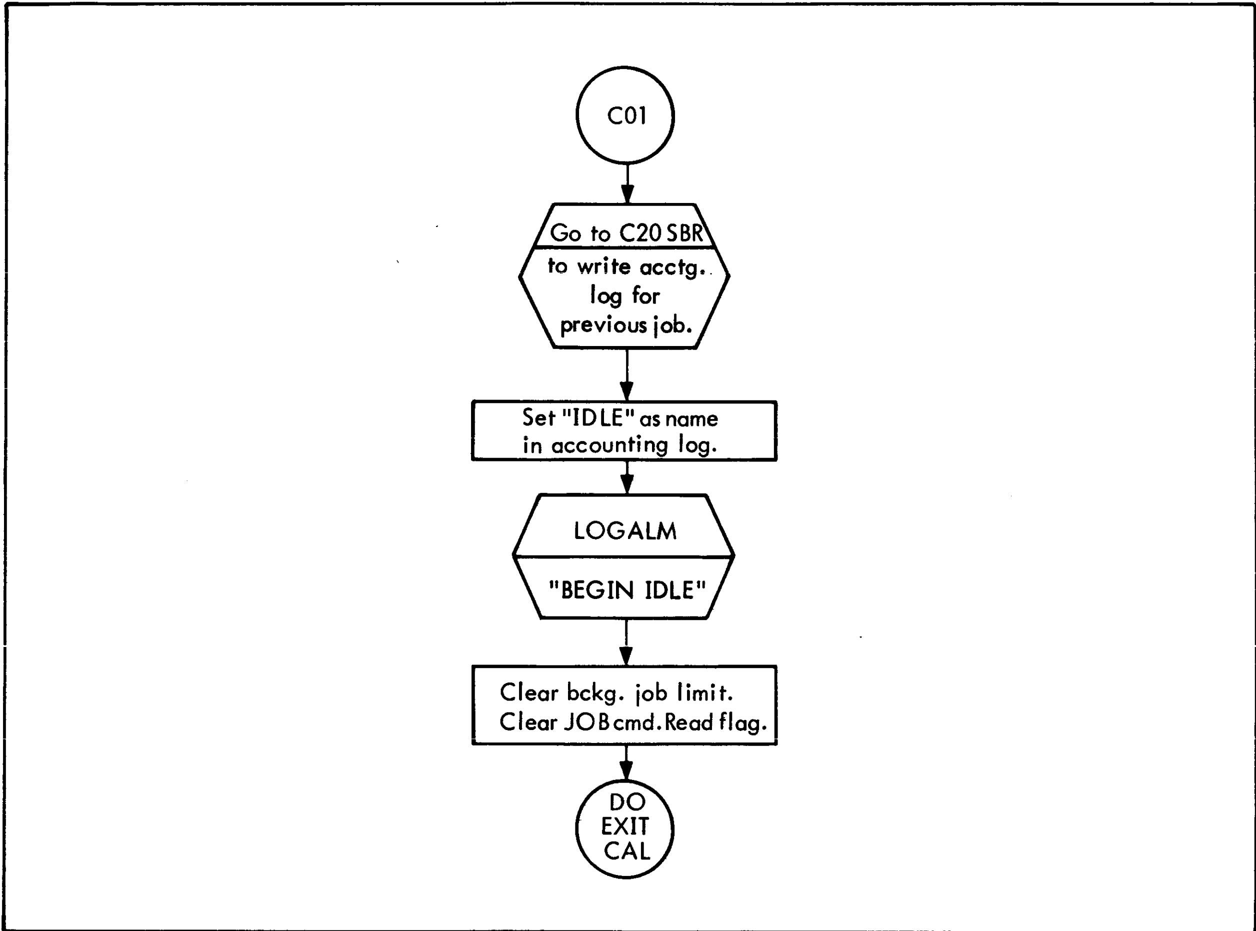


Figure 19. FIN Command Flow

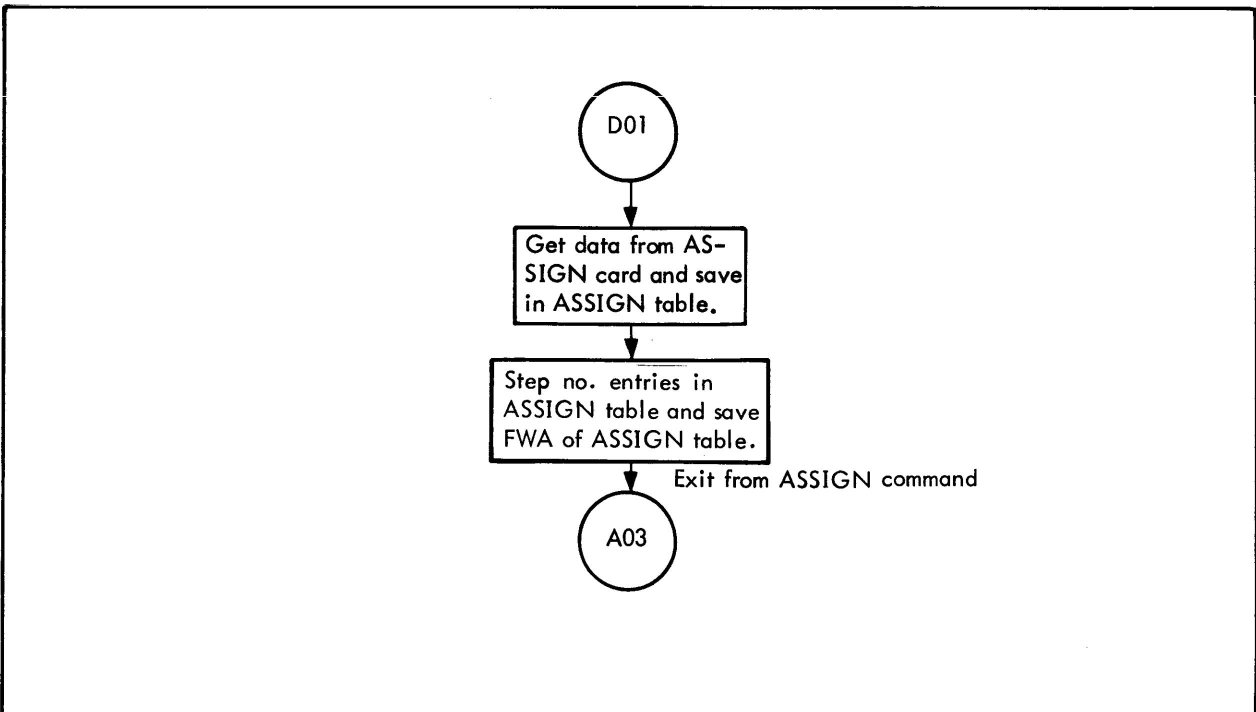


Figure 20. ASSIGN Command Flow

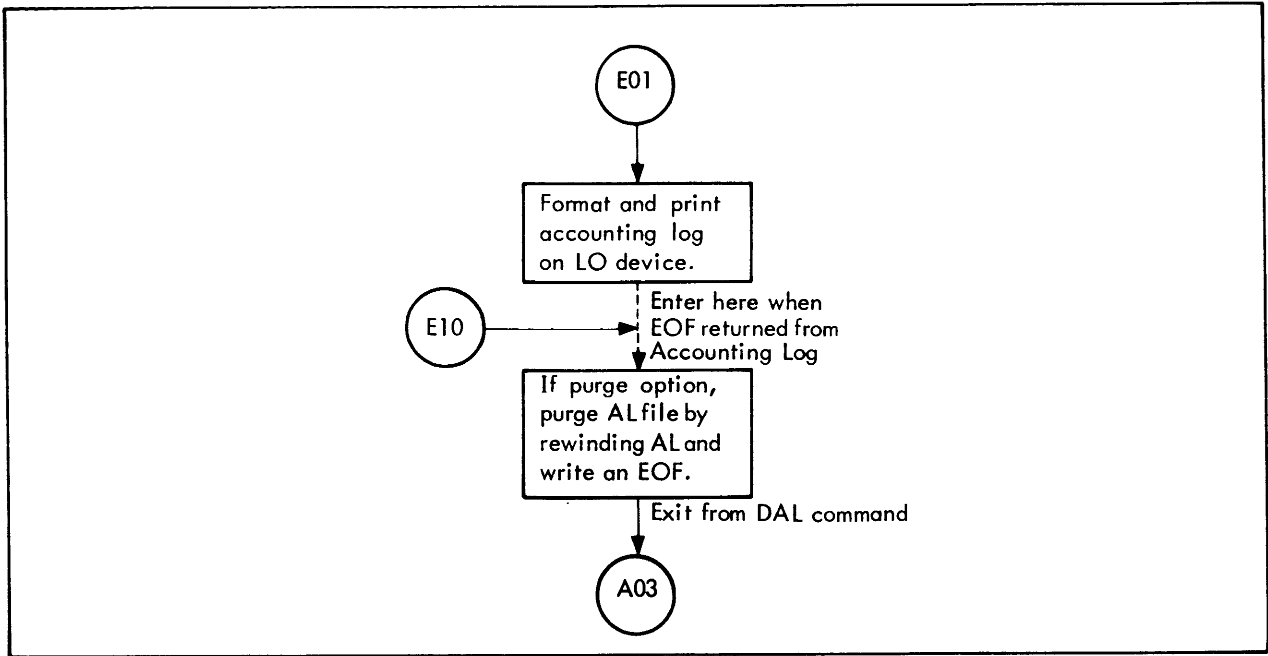


Figure 21. DAL Command Flow

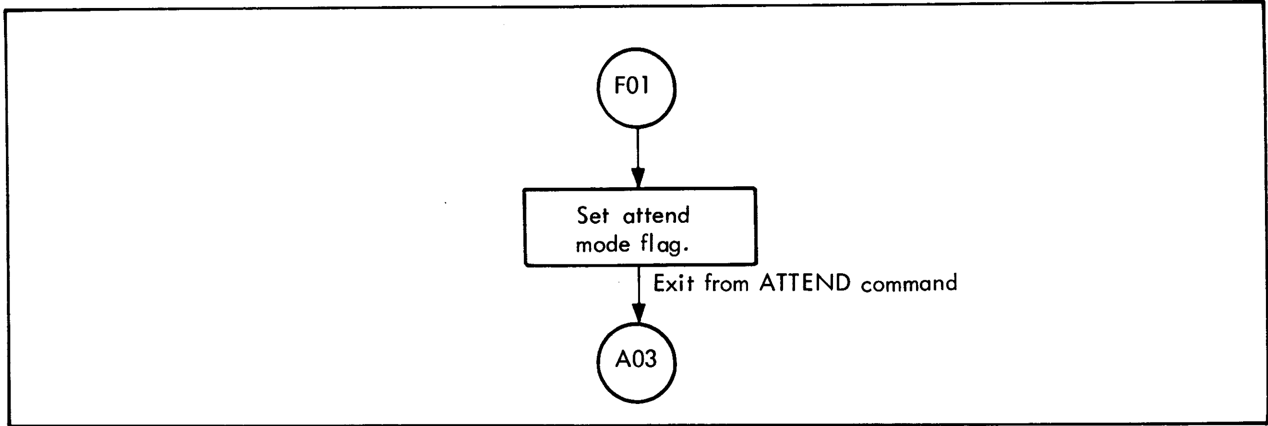


Figure 22. ATTEND Command Flow

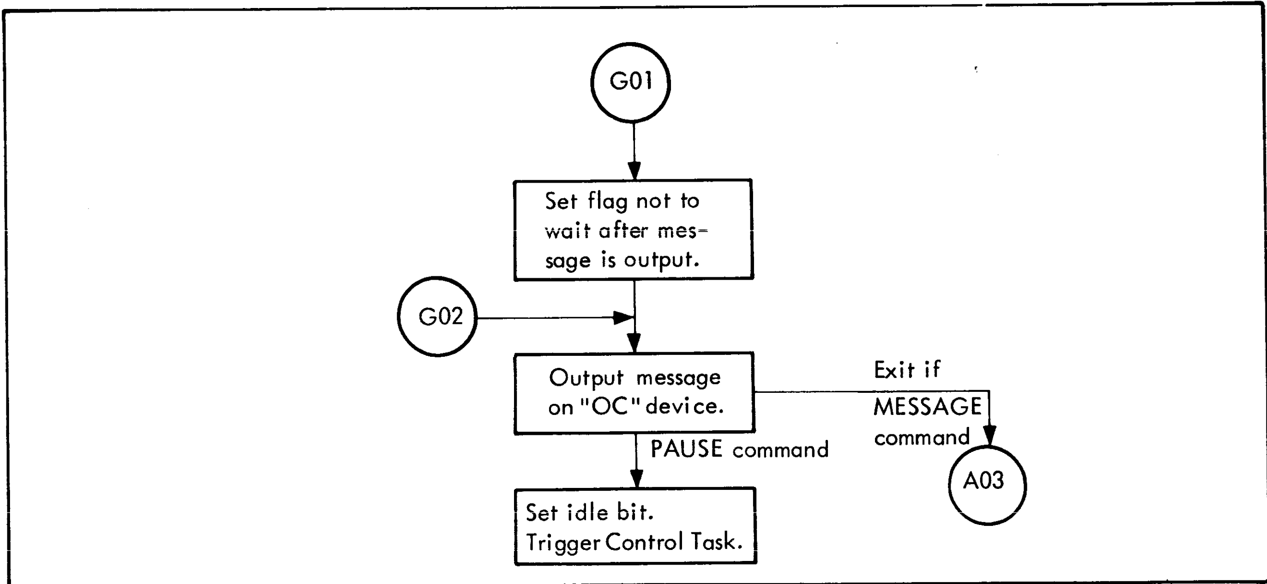


Figure 23. MESSAGE Command Flow

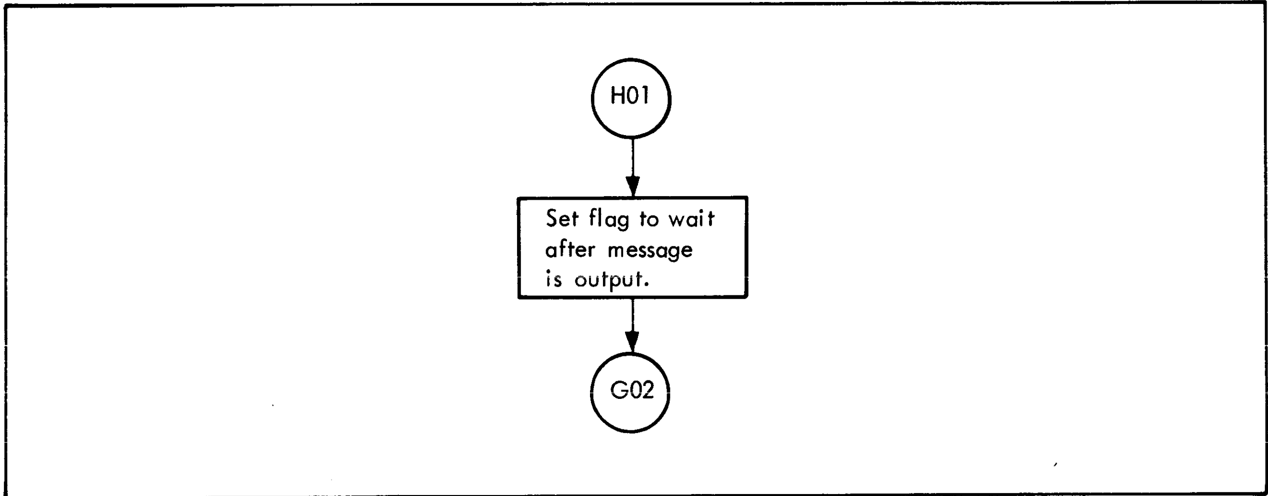


Figure 24. PAUSE Command Flow

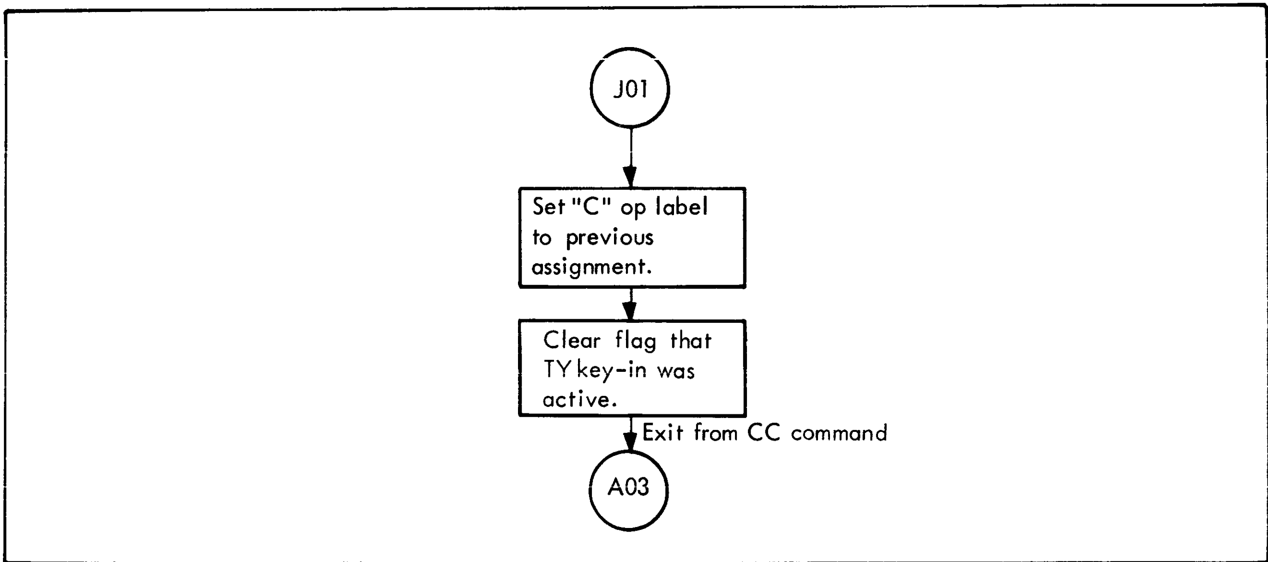


Figure 25. CC Command Flow

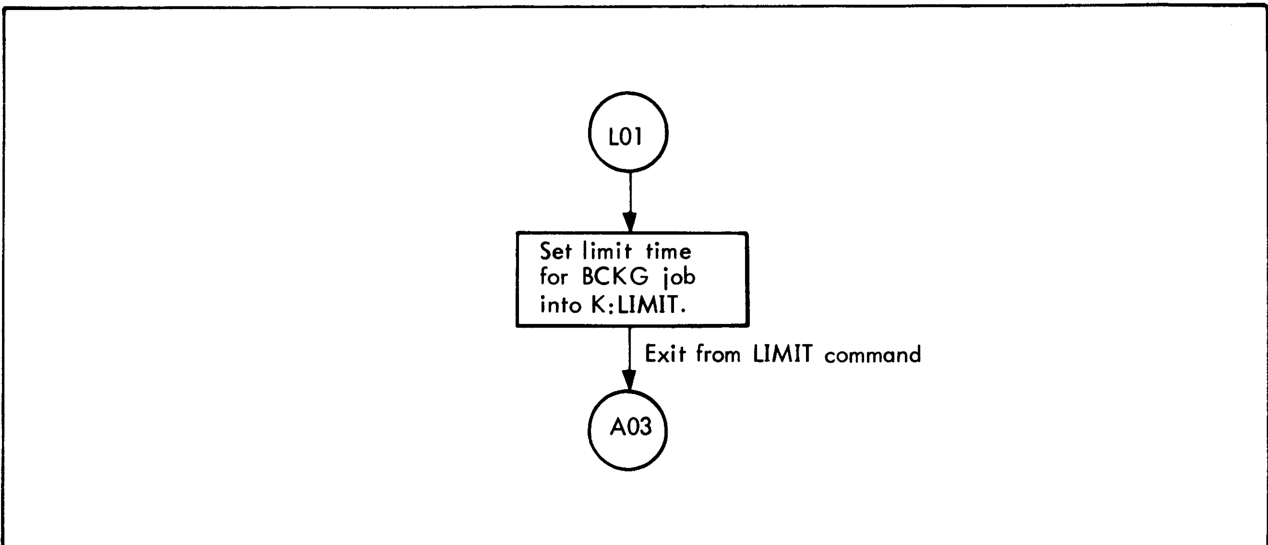


Figure 26. LIMIT Command Flow

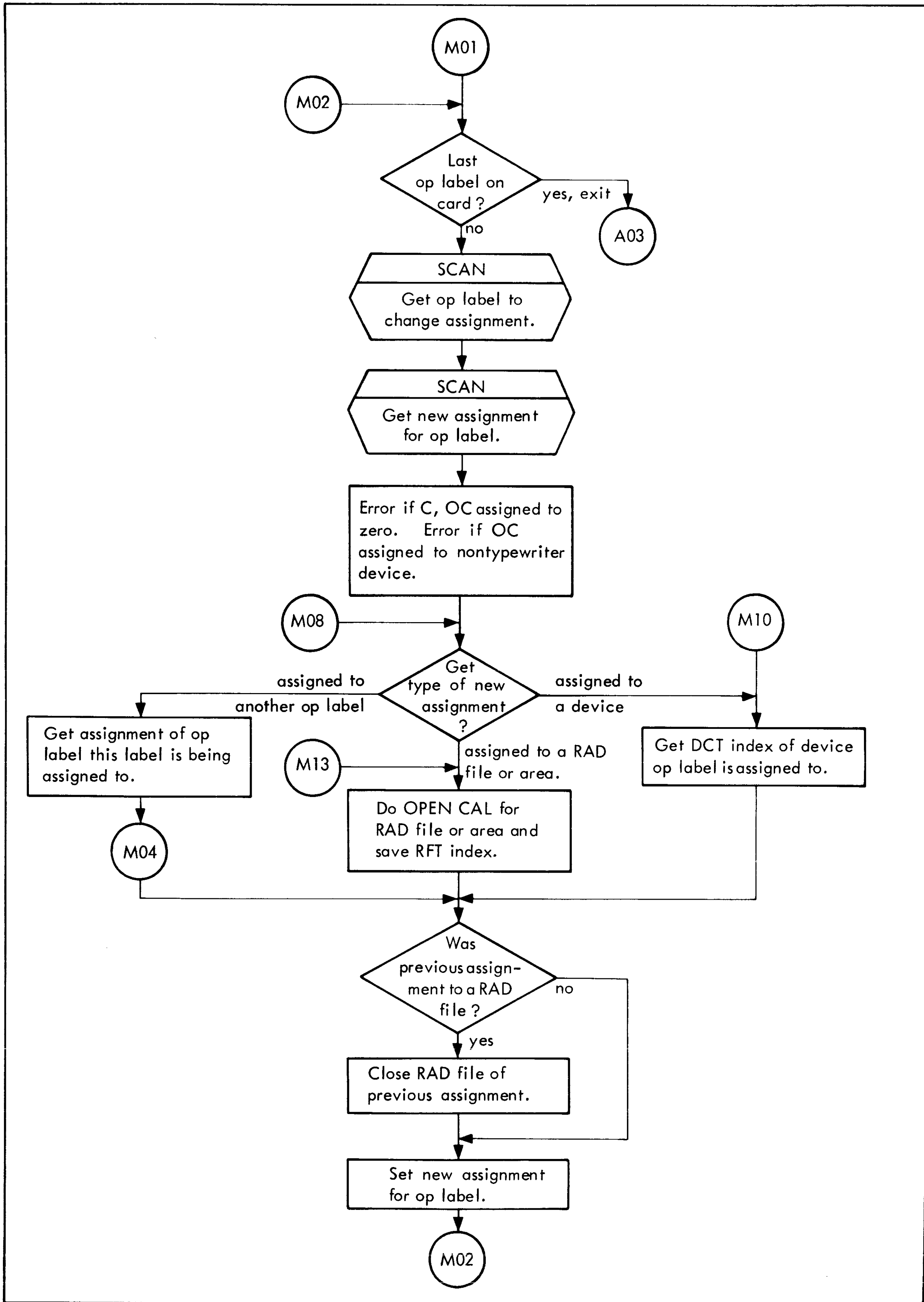


Figure 27. STDLB Command Flow

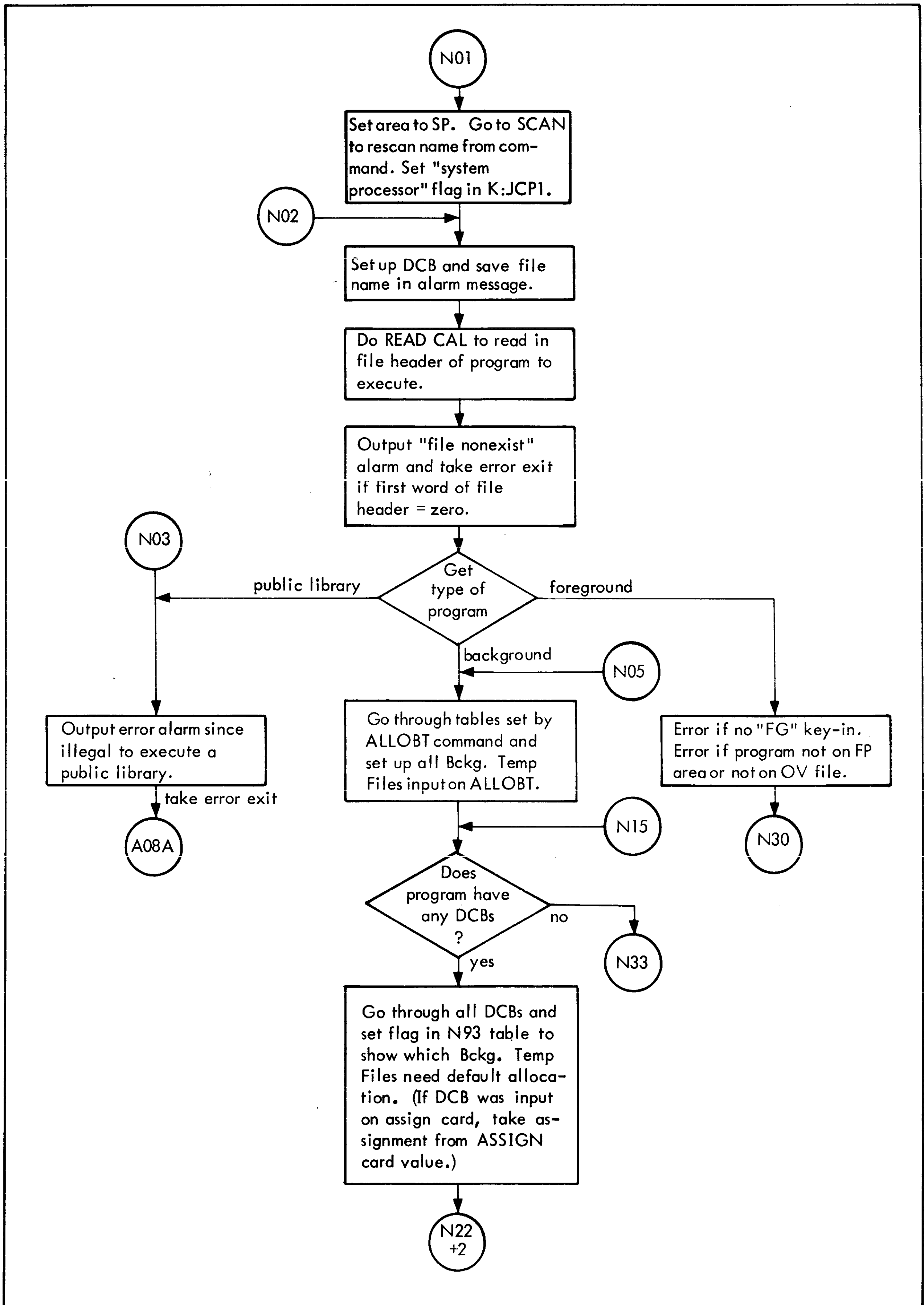


Figure 28. NAME Command Flow

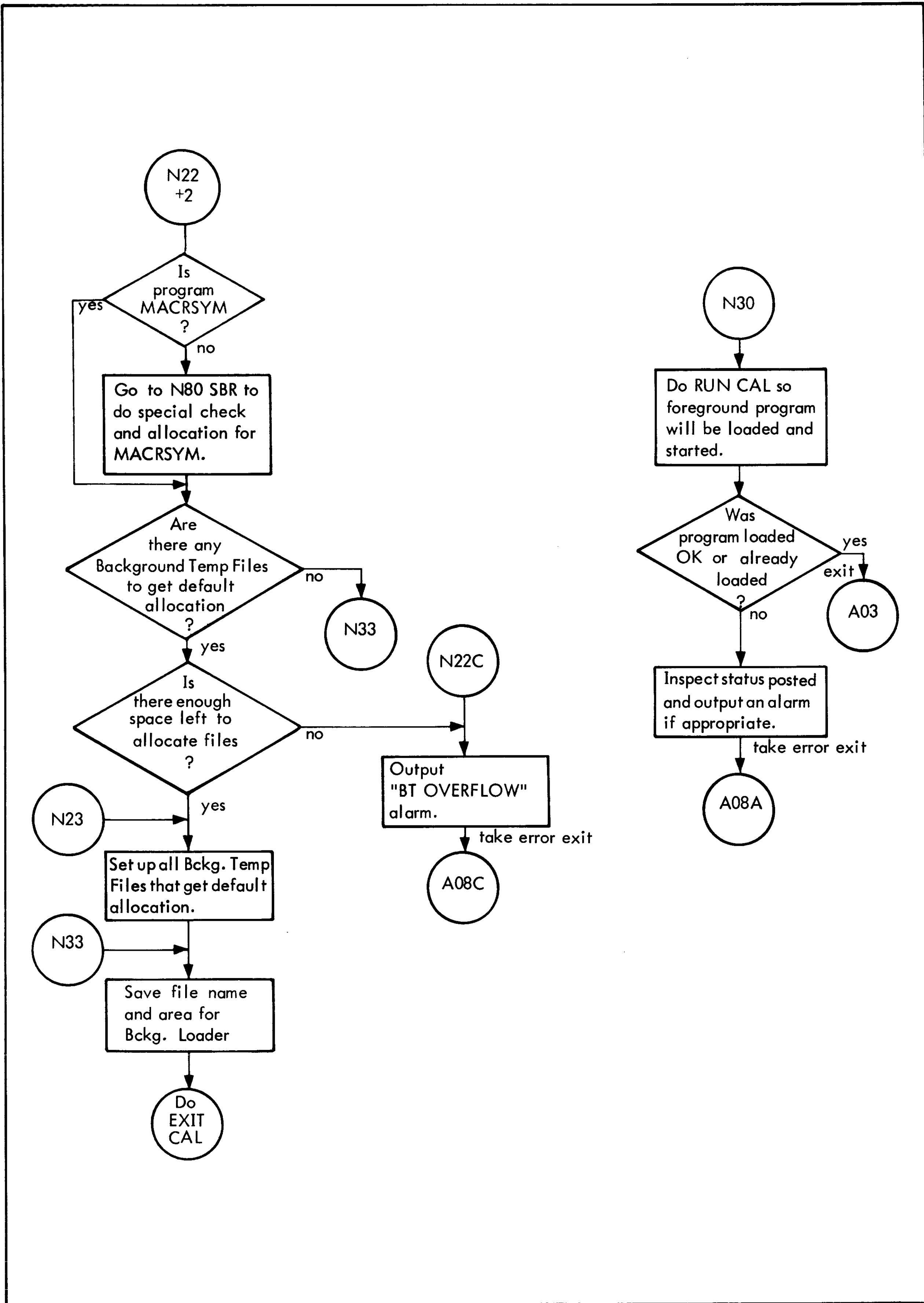


Figure 28. NAME Command Flow (cont.)

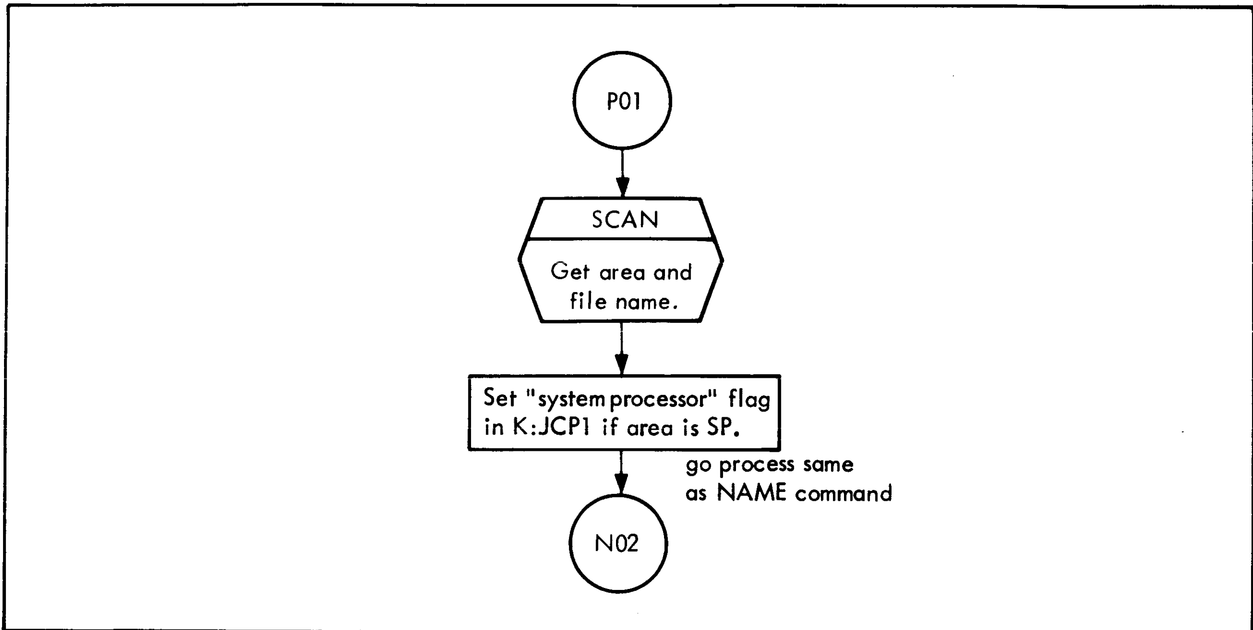


Figure 29. RUN Command Flow

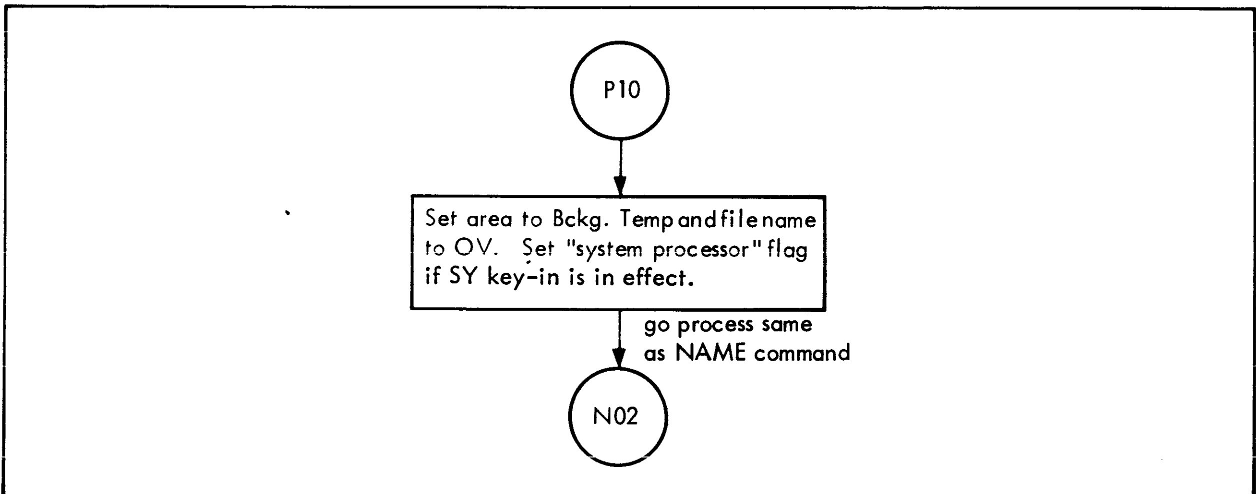


Figure 30. ROV Command Flow

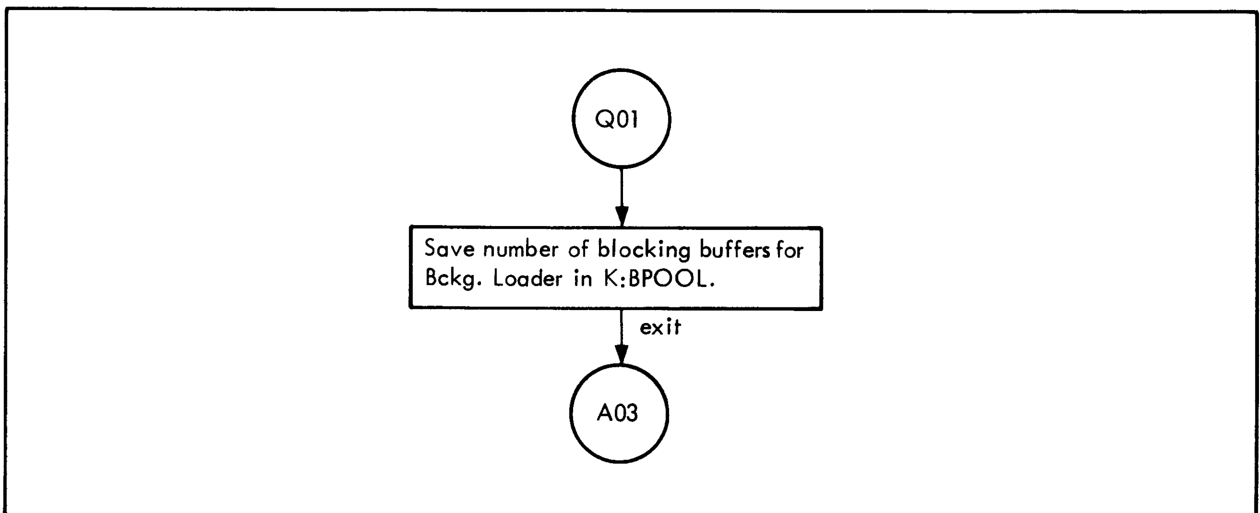


Figure 31. POOL Command Flow

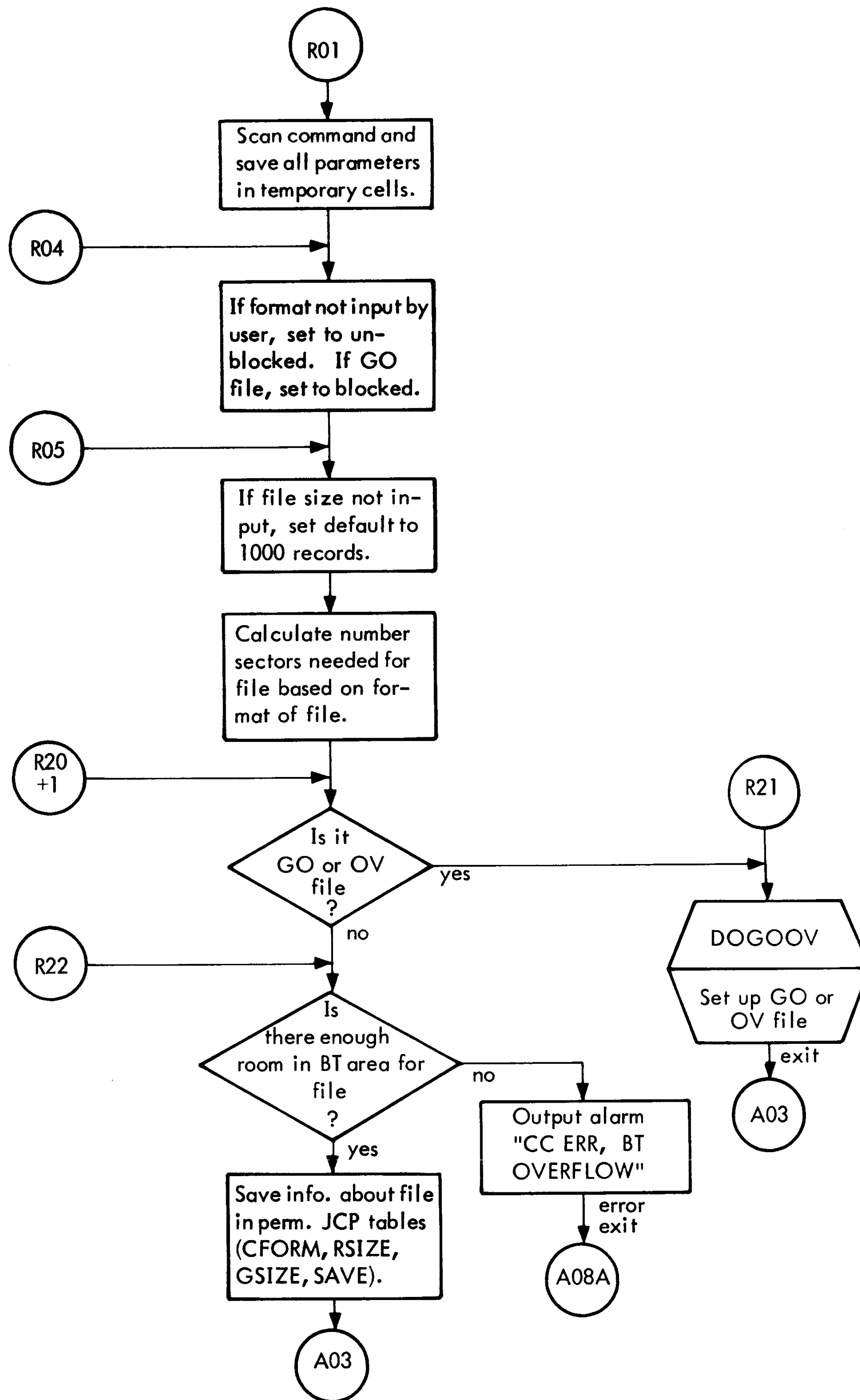


Figure 32. ALLOBT Command Flow

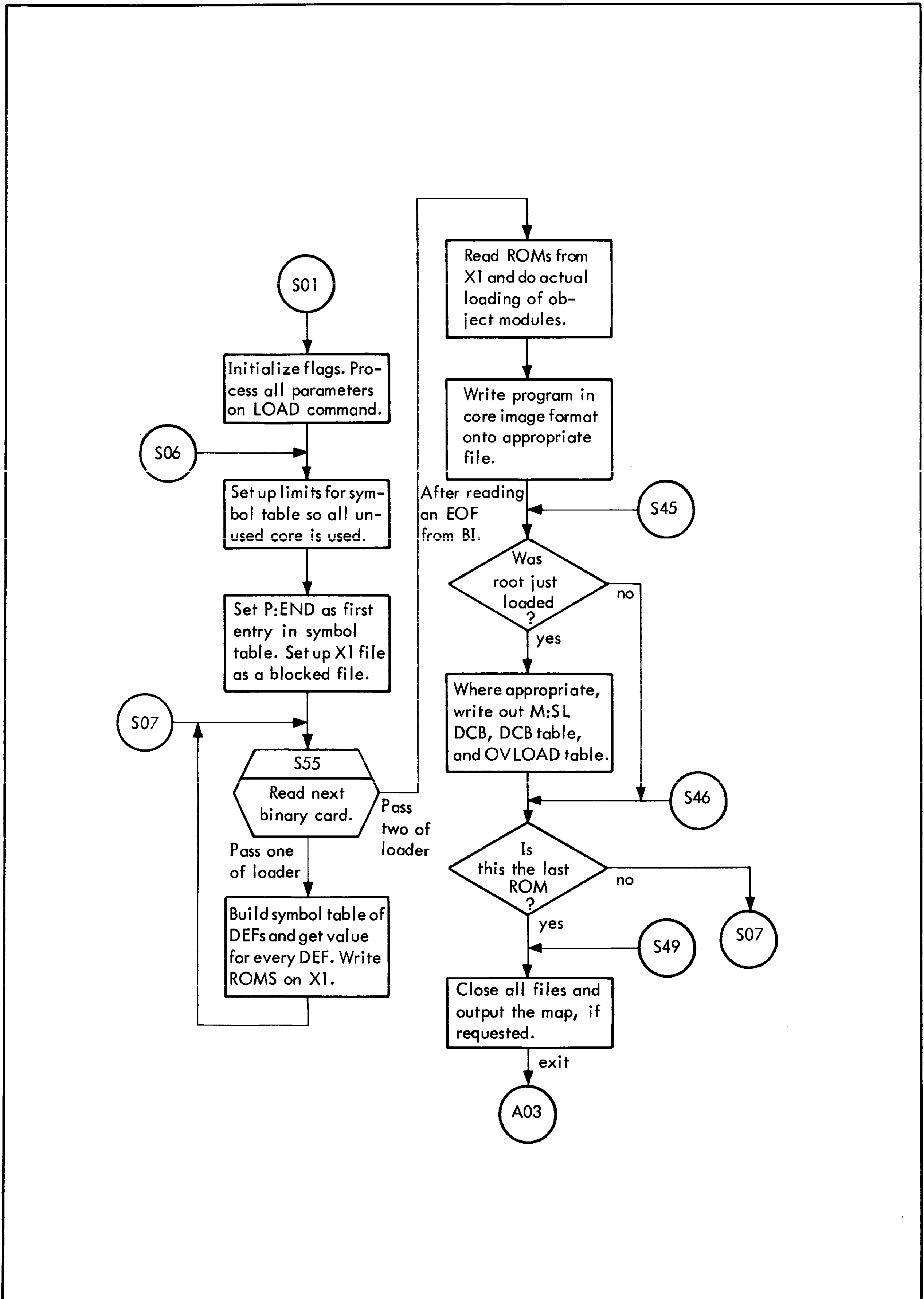


Figure 33. LOAD Command Flow

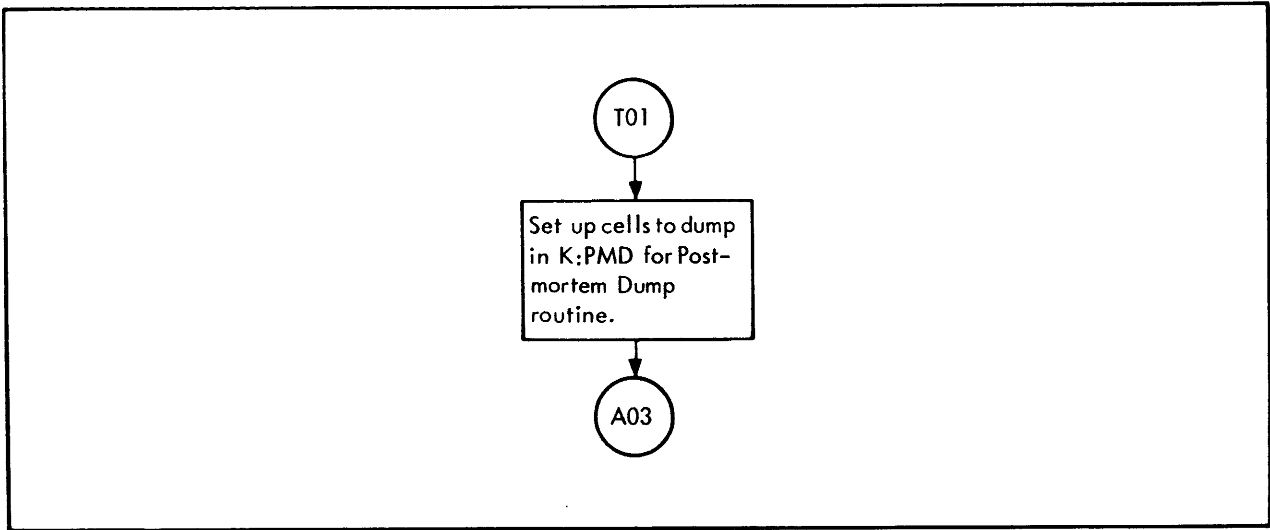


Figure 34. PMD Command Flow

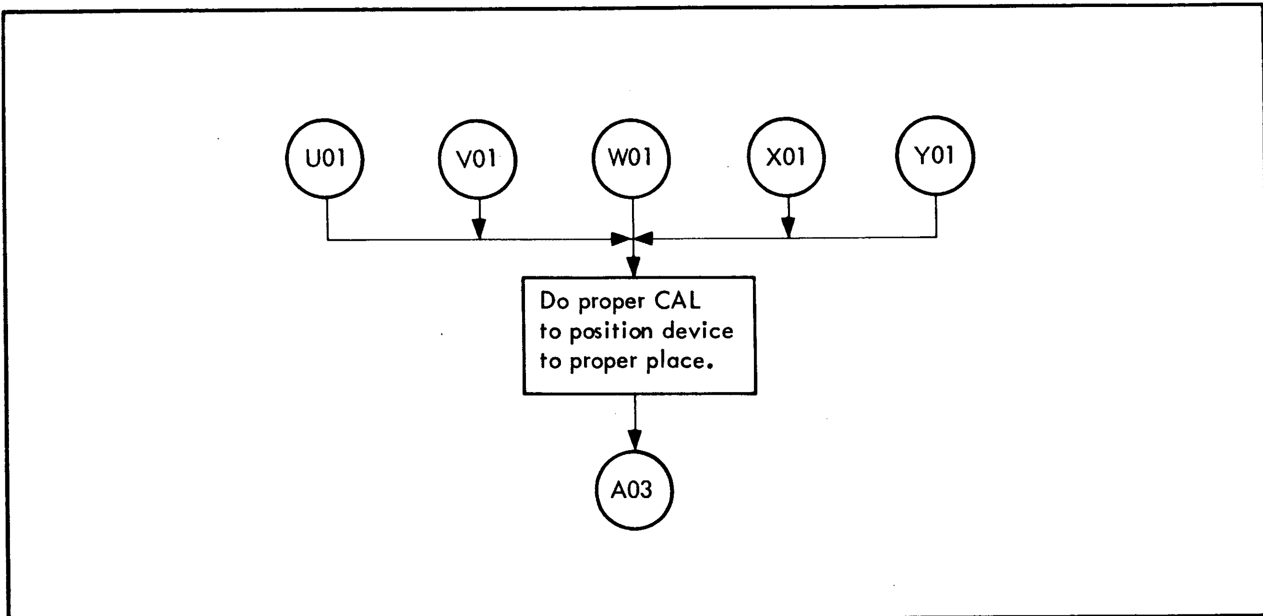


Figure 35. PFIL, PREC, SFIL, REWIND, and UNLOAD Command Flows

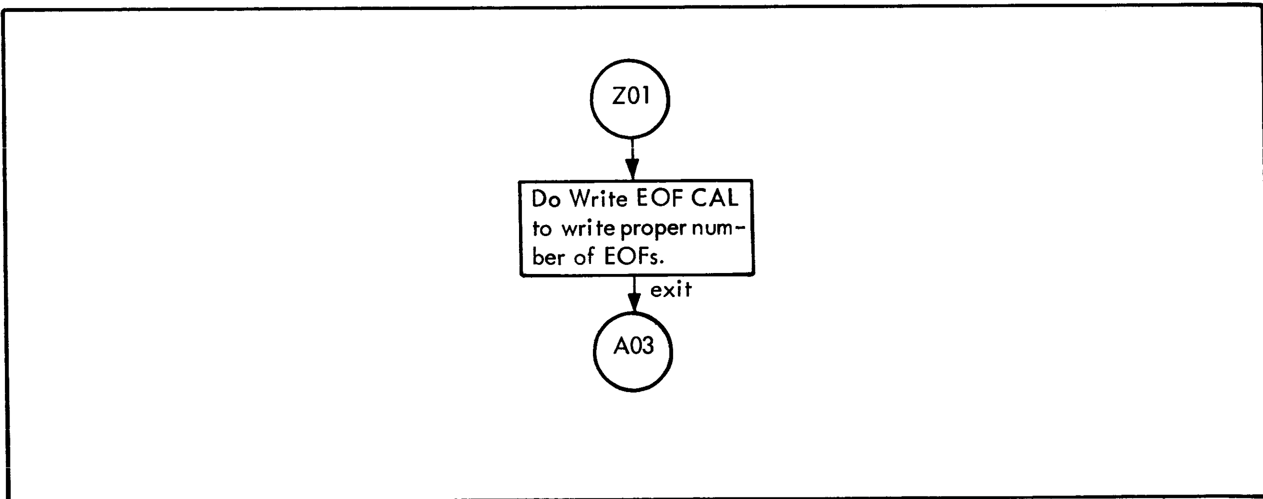


Figure 36. WEOF Command Flow

The diagram in Figure 37 depicts the core layout as the JCP executes.

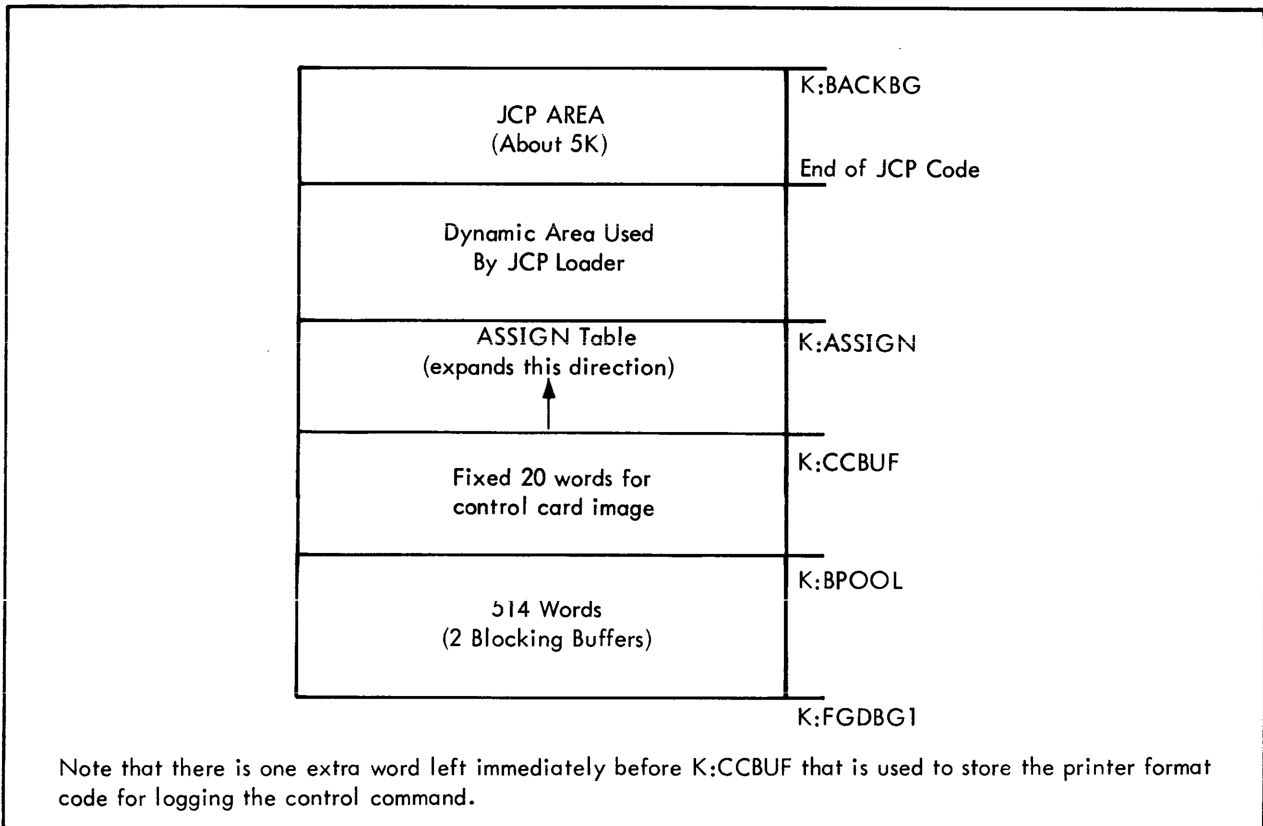


Figure 37. Core Layout During JCP Execution

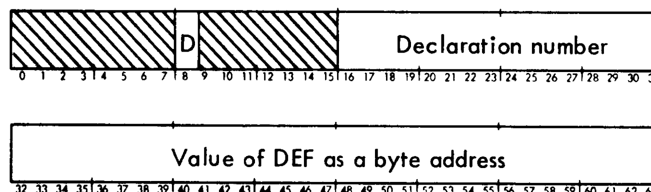
JCP Loader

The JCP Loader loads Relocatable Object Modules (ROMs) or groups of object modules that use a subset of the Xerox Sigma 5/9 Object Language. Initially, the Loader processes all parameters on the !LOAD 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. 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 later in this 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 38 depicts the core layout before PASS1 of the JCP Loader.

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.

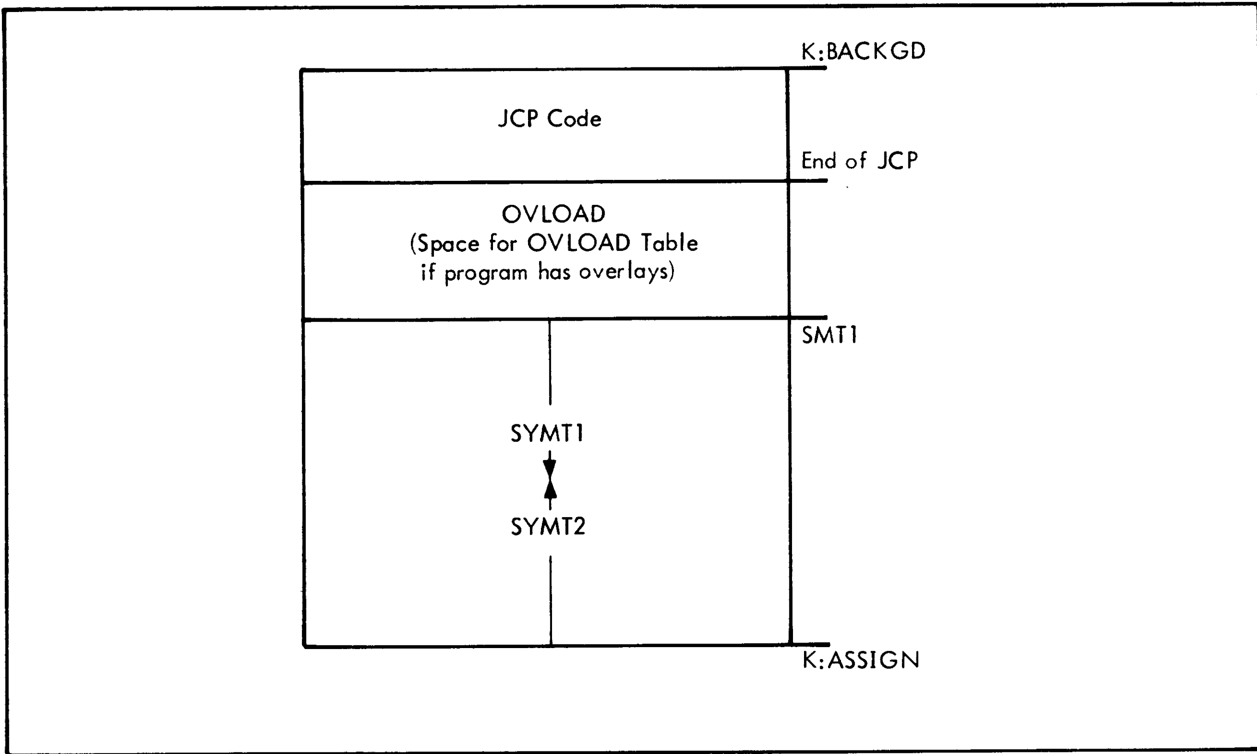
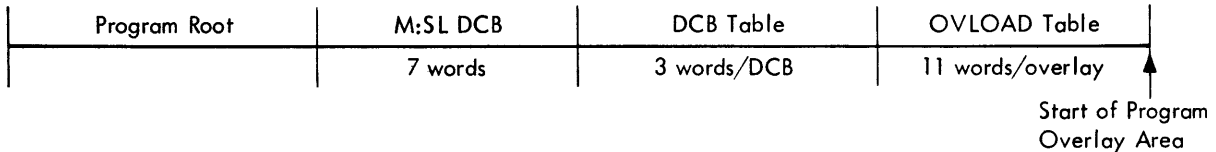


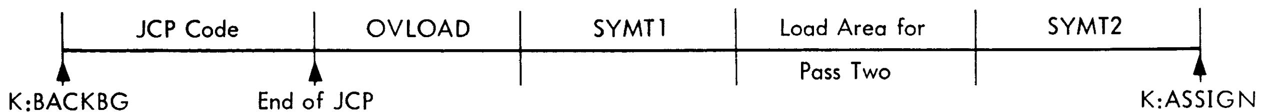
Figure 38. Pre-PASS1 Core Layout

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 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 read the next control command. The format of the OVLOAD table is described in the "RBM Table Formats" chapter.

Job Accounting

Job accounting is an option selected at SYSGEN time. An accounting file will be kept on the RAD by the JCP if the accounting option was chosen. The file must be defined by the user; must have the name "AL"; and must be in the D1 area of the disk.

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 RAD consists of a series of eight-word records, where a new eight-word record is added for each job. The first record in the file is reserved for the IDLE account and is the only record that is ever rewritten. The elapsed time in the IDLE account is incremented by the appropriate amount anytime a !JOB command is input after a prior !FIN command, and the IDLE entry is then rewritten on the disk. 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

The IDLE account has an account number of "IDLE" and a name consisting of all EBCDIC blanks.

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.

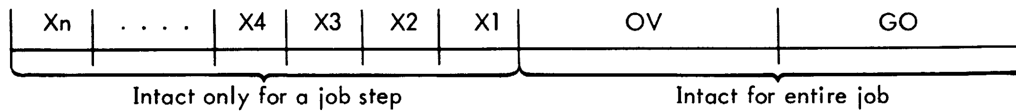
Background TEMP Area Allocation

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 redefined at the time the command is processed.

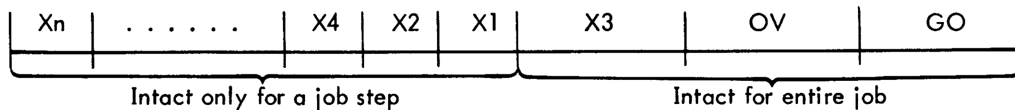
The files in the BT area are allocated so that files remaining intact only for that job step are allocated at the front of the BT area. Files that remain intact for the entire job are allocated at the back of the BT area. Normally, this means that X1 through X9 are allocated at the front of the BT area, 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 ($1 \leq i \leq 9$) 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 a search is made of all user M:Xi DCBs, and 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 and if a M:Xi DCB is found for which a file has not been allocated, a "BT OVERFLOW" alarm will be output and the job aborted.

The following examples depict the allocation of BT as previously described:

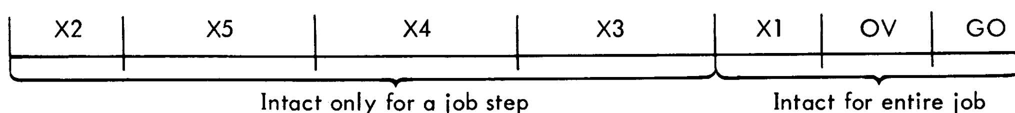
Example 1:

1. An !ALLOBT command for X1 file with SAVE option.

2. An !ALLOBT command for X2 file.

3. A user program with M:X1, M:X2, M:X3, M:X4, and M:X5 DCBs.

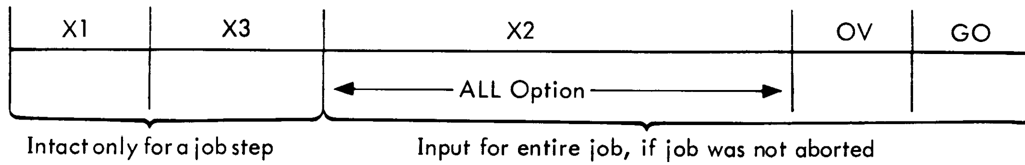
In this case, the BT area would be allocated as



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

3. A user program with M:X1, M:X2, M:X3, and M:X4 DCBs.

The BT area in this case would be allocated as



In this example, the job would be aborted because there is no remaining room to allocate the M:X4 DCB, since the "ALL" option was used for the X2 file. If the "ALL" option is used for file size, all Xi files used by the program must be allocated via the !ALLOBT command.

The JCP does special allocation of the BT area for the AP processor, since the scratch space requirements of this processor depend on the parameters of its 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 the AP processor in the background. Other special allocation tables could be added for other processors requiring nonstandard allocations.

5. 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 RBM, 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.

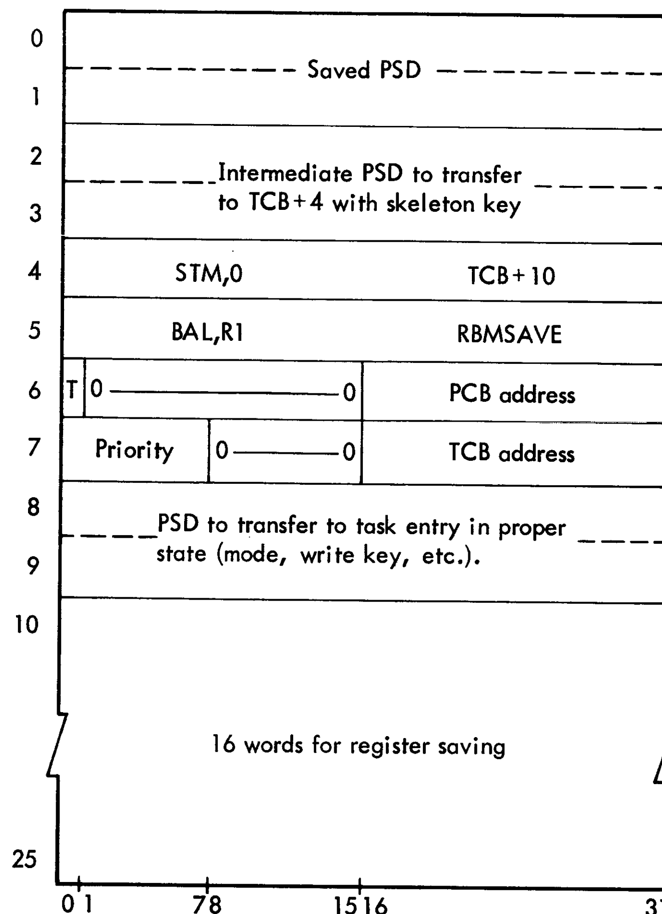
For HIO, the IOEX bit (DCT5, bit 7) is tested. If the bit is set, the HIO is executed and status and condition codes are returned to the user. If the IOEX bit is not set, the monitor routine RIPOFF is called which will eliminate any ongoing or queued requests for the device. The user receives status 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 39.

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:



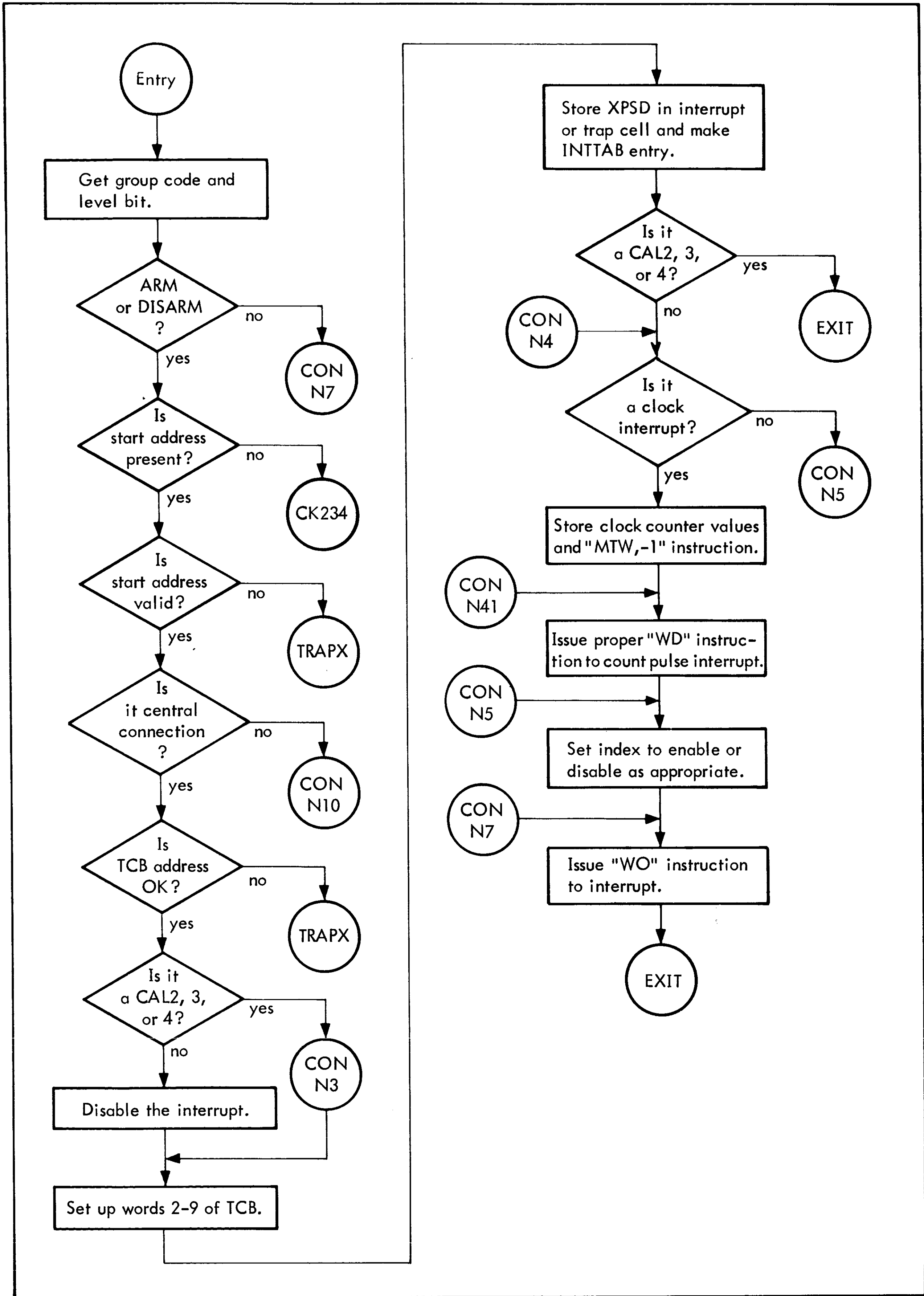


Figure 39. ARM, DISARM, and CONNECT Function Flow

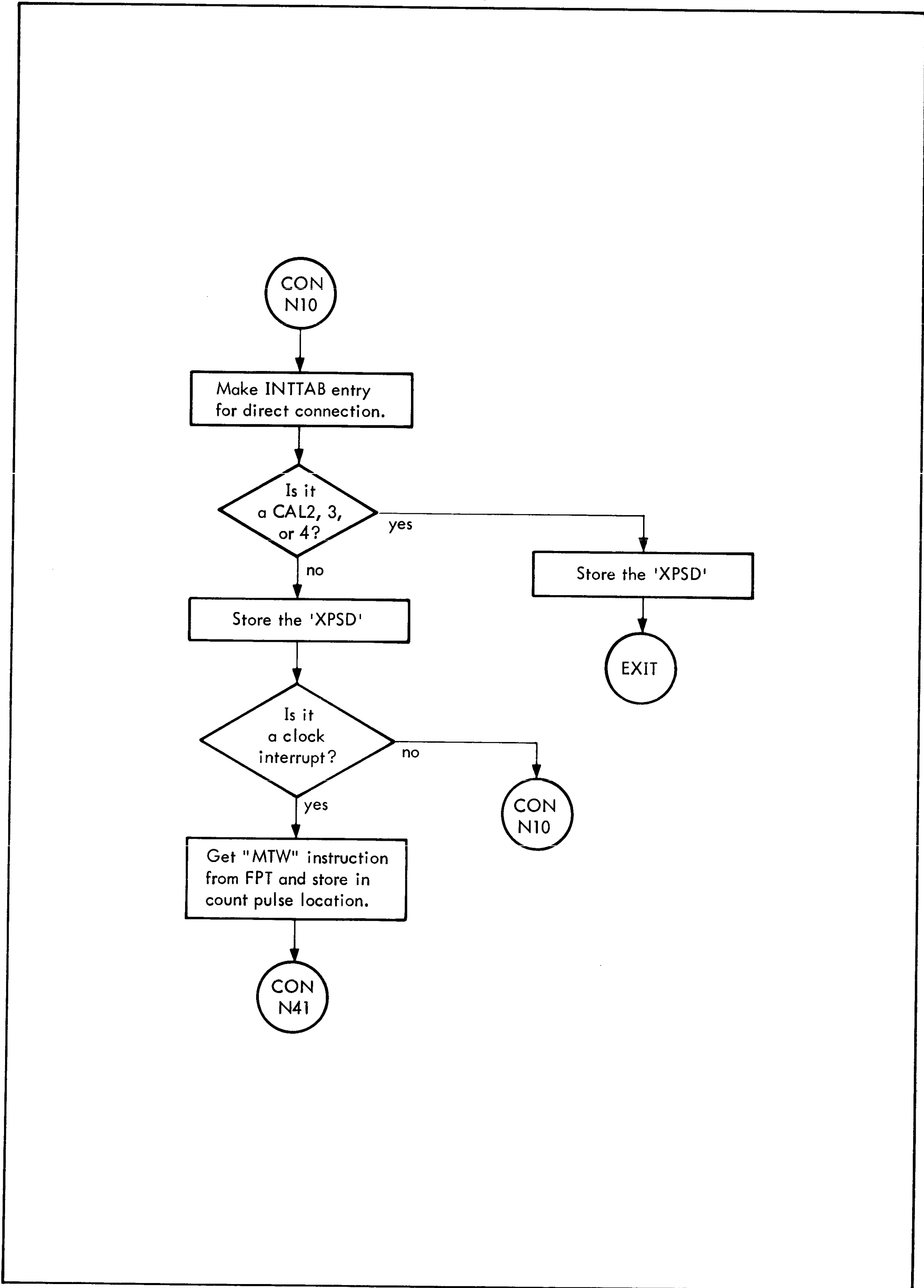


Figure 39. ARM, DISARM, and CONNECT Function Flow (cont.)

6. MONITOR INTERNAL SERVICES

RBM Overlays

All RBM 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.

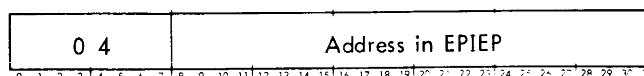
In order to create an overlay the programmer must include DEF's in the overlay ROM for all possible ENTRY points and all possible EXIT points. An ENTRY point is defined as a point at which one would enter the overlay via any type of branching instruction (BAL, BCR, BCS, LPSD, etc.). An EXIT point is 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 RBM 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, of course, is written out with the monitor. The last two ROMs on the SYSLOAD input device must be INIT (presently assembled with the monitor) and JCP in that order. Figure 40 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.

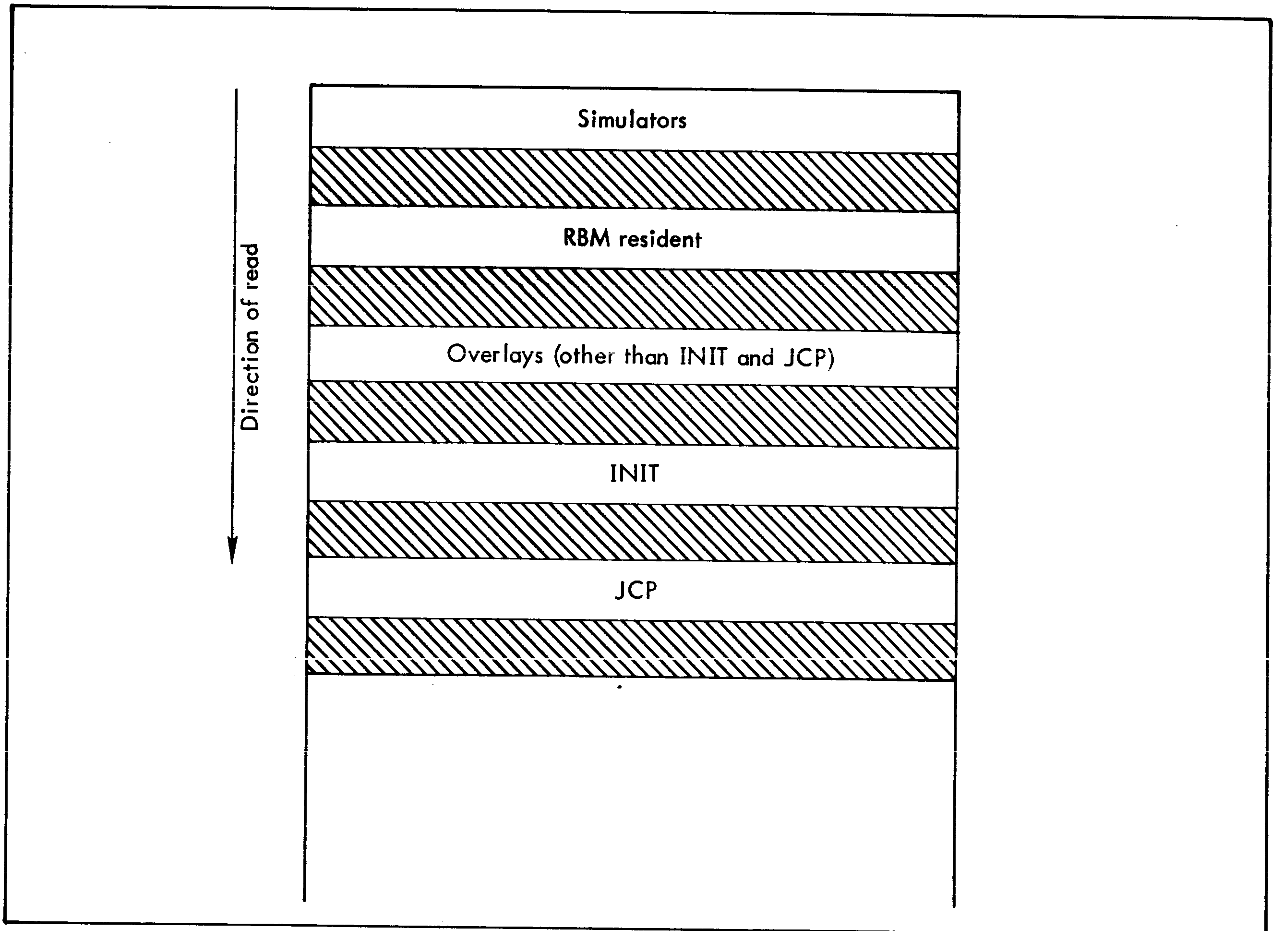


Figure 40. Arrangement of SYSLOAD Input ROMs

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.

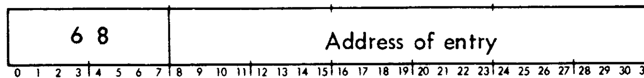
Entry and Exit Point Inventory (EPI)

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.

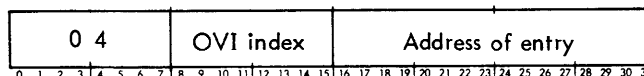
Type: Parallel in RBM table space with a fixed number of entries. Generated by SYSLOAD.

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.

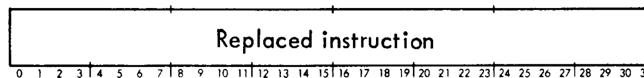
EPIEP: An EPI table entry can have one of three forms. If the entry is an ENTRY point to a resident overlay:



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)

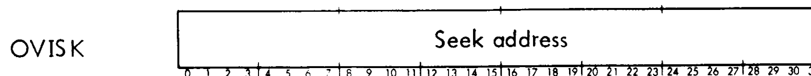
Purpose: The OVI replaces the table previously defined as OVLOAD. It is used by OMAN to load overlays for both primary and secondary tasks. For each overlay it contains the sector address, length, and name.

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

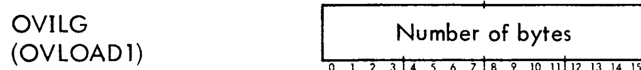
Logical Access: The EPI (Entry and Exit Point Inventory) has a subfield of EPIEP which indexes the proper overlay for that Entry Point.

Entries:

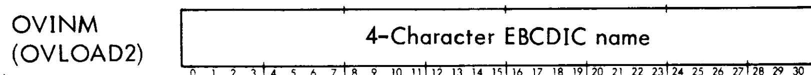
OVISK



OVILG (OVLOAD1)



OVINM (OVLOAD2)



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

OVILG is the length of this overlay in bytes (≤512).

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.

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 RBM.
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	CLOSE	DEVICE
INIT	READ	PRINT
ENQ	WRITE	TYPE
SIGNAL	REW	ALLOT
STIMER	UNLOAD	TRUNCATE
POLL	WEOF	DELETE
SEGLOAD	PFIL	STDLB
OPEN	PREC	

In addition to the above CAL processors, RBM 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

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 exits.

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

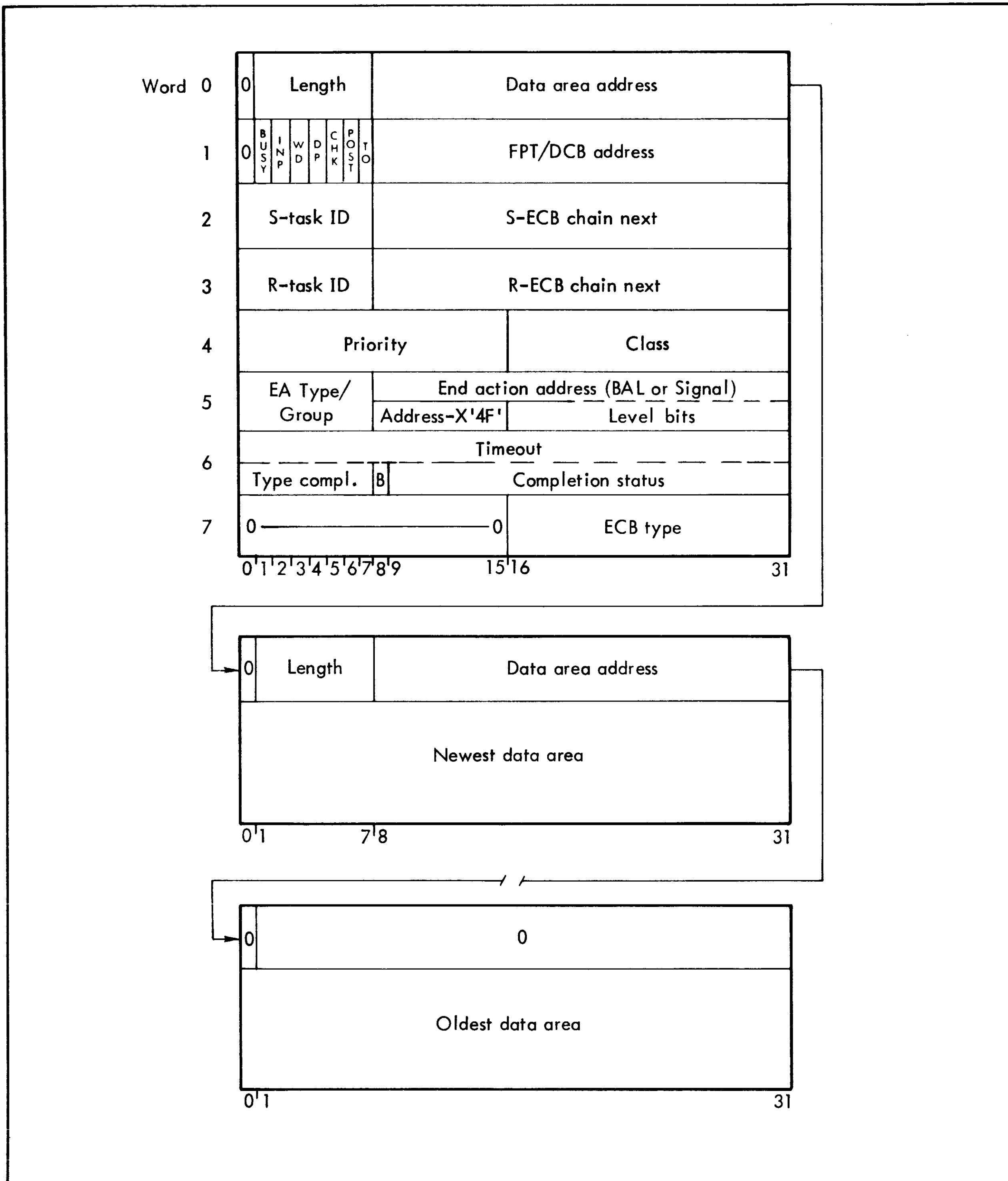


Figure 41. ECB Format and Chained Data Areas

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.

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 RBM 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). 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 delete 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 RBM 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. RBM tasks will have a load-module-inventory entry to head these chains. The chain is in priority order, with the oldest 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 RBM 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.

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 data, 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).

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 (I/O only).
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 RBM 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.

Dynamic Space

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, 6, 16, 32, 64, 128, 256) only. Thus if a user asks for 19 words he 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

The dispatcher in RBM is used to schedule secondary tasks. These include Background and the RBM Control Task, both of which actually run at the null priority level, and any other foreground tasks linked as secondary tasks. The level specified at SYSGEN time on the :CTINT control command is used to give control to the dispatcher when a change of scheduling may be desired. This may occur when a secondary task does an asynchronous operation with wait, or exits, traps, or aborts; or when a timeout occurs, an asynchronous operation completes, or a 30-second hardware time-check occurs.

When the dispatcher receives control, it searches the STI (from bottom up) for any secondary tasks that are not stopped and have an STICOUNT of zero. When one is found its STCB (Secondary Task Control Block) is set up and control is transferred to RBMEXIT. This causes control to be given to the secondary task, or to the Overlay Manager if an overlay reload is necessary.

If the dispatcher has nothing to do, it WAITs at the null priority level.

7. 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 R0 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 (not in background for background program, or in foreground for foreground user), 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

being output on OC and LL

where

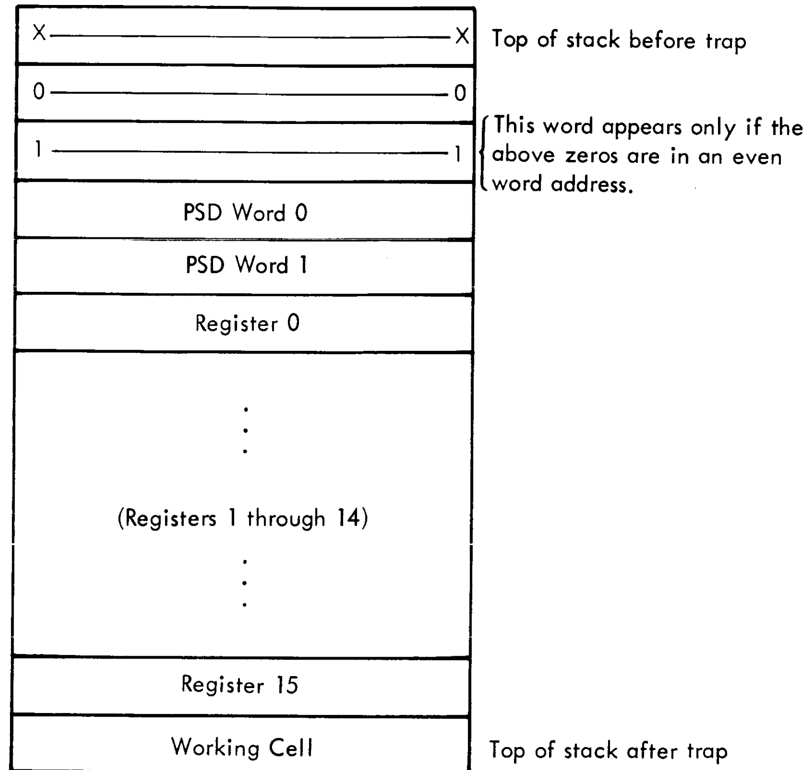
xx is the Error Code in hexadecimal (00 if none).

yyyyy is the address of the CAL.

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.

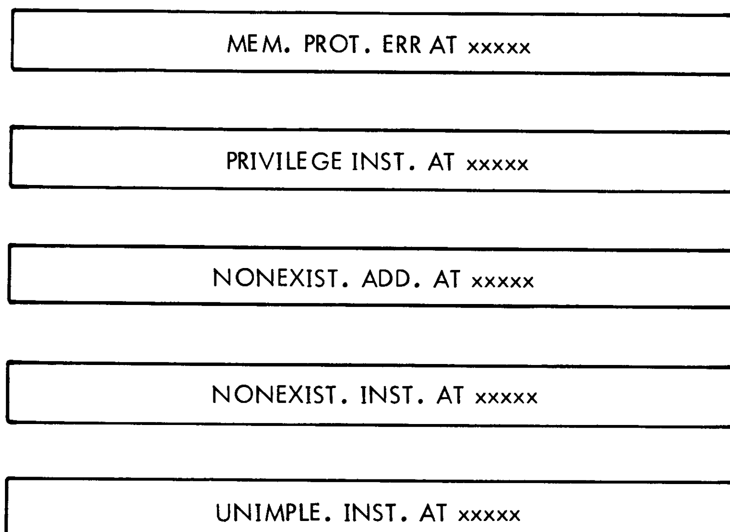
The registers and PSD are saved in the user Temp Stack in the following format:



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

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 due to incorrect parameters being input. After the message is output, the task will be aborted unless the user has provided a trap handler for this trap. If a trap handler is provided, 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.

8. RBM TABLE FORMATS

General System Tables

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

Disk File Table (RFT)

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

File 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.
RFT2 index 0	Entry contains the total number of RFT entries.
RFT3 index 0	Entry contains the maximum number of RFT entries allowed for background use.
RFT4 index 0	Entry contains the current number of background file entries.
RFT5 index 0	Entry is used as the RFT activity count for reentrance tests.
RFT11 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.

Table 2. Disk File Table Allocation

Address	Contents	Initial Value	Length
RFT1	File Name	0	Doubleword
RFT2	Beginning Sector Address (Relative to Area)	X	Halfword
RFT3	Ending Sector Address (Relative to Area)	X	Halfword
RFT4	Granule size (in bytes)	X	Halfword
RFT5	Record size (in bytes)	X	Halfword
RFT6	File Size (in records)	X	Word
RFT7	Switches where Bit 0 = 1 means sequentially written Bit 1 = 1 means directly written Bit 3 = 1 means compressed 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	Halfword
RFT12	Current Record Number	X	Word
RFT13	Number of Open DCBs (total)	X	Byte
RFT14	Not used	X	Byte
RFT15	Number of BGND 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

Device Control Table (DCT)

DCT Format

The Device Control Table (DCT) is composed of several parallel subtables (see Table 3). 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

```
LH, R     DCT1, X
```

DCT2 would be accessed by

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

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

Table 3. 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
	Format of DCT1, DCT1P, DCT1A: bits 0–4 = 0—0; bits 5–7 = 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. Bit 1 = 1 means input is legal for this device. Bit 2 = 1 means device has been marked down and is inoperative. Bit 3 = 1 means device timed out. Bit 4 = 1 means SIO has failed. ~~Bit 5 = 1 means the I/O has aborted.~~ Bits 6/7 = 00 – "Busy" both subchannels. = 01 – Use the P subchannel only. = 10 – Use the A subchannel only. = 11 – Use either subchannel.	Byte
DCT4	Device Type 0 = NO (IOEX) 1 = TY 2 = PR 3 = PP 4 = CR 5 = CP 6 = LP 7 = DC 8 = 9T 9 = 7T 10 = CP (Low Cost) 11 = LP (Low Cost) 12 = DP 13 = PL	Byte
DCT5	Status Switches Bit 0 = device busy. Bit 1 = waiting for cleanup. Bit 2 = between inseparable operations. Bit 3 = data being transferred.	Byte

Table 3. DCT Subtable Formats (cont.)

Subtable Address	Contents	Length
DCT5 (cont.)	Bit 4 = error message given (key-in pending). Bit 5 = unused Bit 6 = SIO was given while device was in manual mode. 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 IO for reserved devices.	Byte
DCTRBM	Bit 6 = 1 means DED DPndd, R keyin is in effect.	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	" (NL) !!yyndd" where yyndd comes from the :DEVICE command.
DCTSDBUF	Pointer to side buffer.
DCTMOD	EBCDIC model number.
DCTMODX	Decimal model number.
DCTJID	X'FF' if reserved device; otherwise 0.

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

DCT7 points to the space allocated by SYSGEN for the command list for the device. The area must begin on a doubleword boundary and have a word length as follows:

Magnetic Tape (7T and 9T)	8 words
Keyboard/Printer	10 words
Card Reader	2 words
Card Punch (7160)	74 words
Card Punch (7165)	2 words
Disk	4 words
Disk Pack	6 words
Paper Tape	8 words
Other Devices	8 words
Line Printer	4 words
Plotter	2 words

Halfword 0 of DCT1 is set by SYSGEN to contain the number of devices (DCT entries) in the DCT table.

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: Bit 0 – Subchannel P busy Bit 1 – Subchannel A busy	Byte

Address	Usage	Size
CIT3 (cont.)	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 4.

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 4. IOQ Allocation and Initialization

Address	Contents	Initial Value	Length
IOQ1	Backward Link	0	Byte
IOQ2	Forward Link	Entry M contains M + 1 for $N > M \geq 0$. Entry N contains 0. N is the number of queue entries.	Byte
IOQ3	Switches Bit 0 = 1 – request busy. Bits 5-7: = 000 – Both subchannels required. = 001 – Subchannel P only. = 010 – Subchannel A only. = 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	Bit 1 = 0 – byte address of buffer. Bit 1 = 1 – DW address of command chain (Queued IOEX).	0	Word

Table 4. IOQ Allocation and Initialization (cont.)

Address	Contents	Initial Value	Length
IOQ9	If IOQ8 bit 1 = 0 – byte count of buffer. If IOQ8 bit 1 = 1 – time-out value for command chain.		
IOQ10	Maximum retry Count	0	Byte
IOQ11	Retry count	0	Byte
IOQ12	Seek Address	0	Word
IOQ13	End-Action data Word 1 Byte 0 is cleanup code where value: 1 = Post status in FPT. 2 = Post status in DCB. 3 = Transfer to address specified in bits15 - 31. 4 = No end action (only available to the monitor). Bit 8 = control device read. Bit 9 = end action data in word 2. Bit 15-31 = FPT completion-status word address for cleanup-code 1; DCB address for cleanup-code 2. Word 2 If word 2 = 0, parameter not present. If byte 0 = X'7F', bits 15-31 are user's signal address. If byte 0 = X'FF', bits 15-31 are user's endaction address. If word 2 ≠ 0, and byte 0 ≠ X'FF' or X'7F', byte 0 = end-action interrupt group code, byte 1 = interrupt address minus X'4F', bits 15-31 contain level bit for interrupt.	0	Doubleword
IOQ14	Priority	0	Byte
IOQECB	ECB pointer in an ECB system (i.e., when the #ECB assembly option is set). Otherwise (#ECB option not set) it has the following format: Bits 0-7 Load module ID. Bits 8-11 Cleanup code. Bit 12 = 1 if original request was a PFIL.	0	Word

Table 4. IOQ Allocation and Initialization (cont.)

Address	Contents	Initial Value	Length
IOQECB (cont.)	Bit 13 = 1 if bits 15-31 contain a pointer to a completion status word. = 0 if bits 15-31 contain an FPT address (cleanup code = 1) or a DCB address (cleanup code = 2). Bit 14 Not used. Bits 15-31 DCB/FPT address or pointer to completion status word.		
IOQERROR	Error-log buffer pointer	0	Word

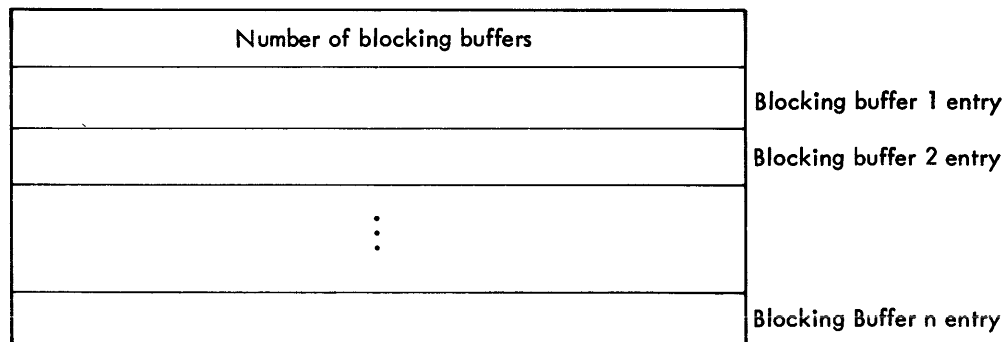
Since the 0th entry is never used in subtables whose entries are words or doublewords, it is not necessary to allocate space for this entry. If the 2N words for IOQ13 are allocated beginning at location ALPHA, IOQ13 is given value ALPHA-2. Thus, IOQ13 may actually point into another table but presents 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 Tables begin on doubleword boundaries.

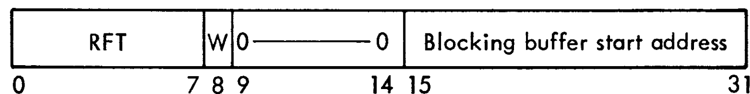
Blocking Buffers

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

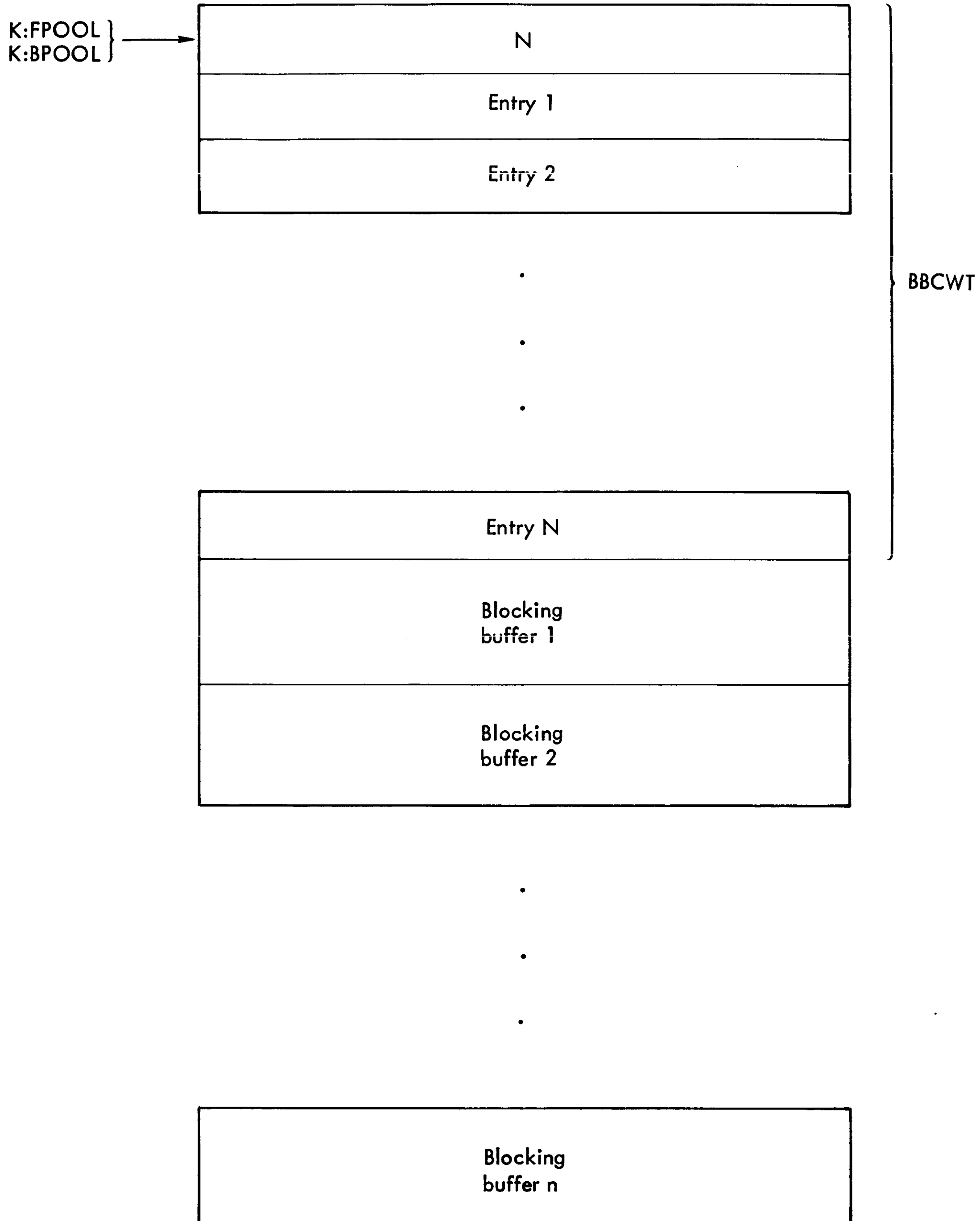
RFT is the index of the RFT entry for the file currently using this buffer. 0 signifies that the buffer is not in use. X'FF' means the buffer is in use, but not by any particular file.

W is set if the blocking buffer has been written in

Foreground and background tasks have different blocking buffer pools and, therefore, have different BBCW tables. K:FPOOL contains an address pointer to the BBCW table used by all foreground tasks in the system. The number and location of blocking buffers available to foreground tasks is determined at SYSGEN by the FFPOOL parameter and cannot be changed except by SYSGEN.

K:BPOOL contains a pointer to the BBCW table used by background tasks. The number and location of the background blocking buffers may vary from job step to job step.

The foreground/background blocking buffer structure is shown below:

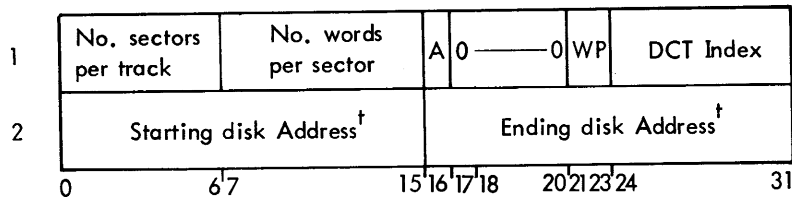


Master Dictionary

K:MASTD (location X'14A'), contains the address of the Master Dictionary. This serial table is indexed by area number where:

Area	DW Index Value	Write Protect Code (WP shown below)
SP	0	4
FP	1	4
BP	2	4
BT	3	2
XA	4	5
CK	5	3
D1	6	1 or 2 (specified during SYSGEN)
D2	7	1 or 2
.	.	.
.	.	.
.	.	.
DF	20	1 or 2

The format of the Master Dictionary (2 words/entry) is



where

A = 0 – area is not allocated.
  = 1 – area is allocated.

N = 0 – directory for this area is not in use; may be updated.
  = 1 – directory for this area is in use; may not be updated.

WP = 1 – (F) only foreground can write in this area (unless SY key-in).
   = 2 – (B) only background can write in this area (unless SY key-in).
   = 3 – (M) only the Monitor can write in this area.
   = 4 – (N) no one can write in this area unless SY key-in.
   = 5 – (X) only IOEX can write in this area.

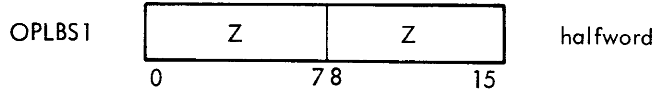
If the system is assembled to include large capacity disks (#MDSHIFT>0), the sector numbers will be shifted one or more bit positions as determined by the assembly parameter #MDSHIFT.

---

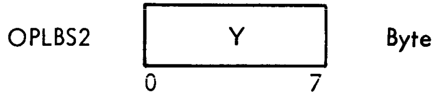
† Starting and ending disk address is given as a sector number (relative to start of the disk).

Operational Label Table (OPLBS)

The Operational Label Table is a parallel table with the format



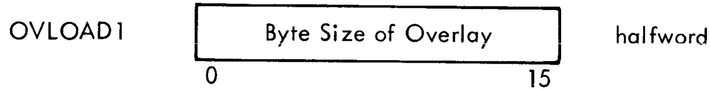
where ZZ is the operational label in EBCDIC

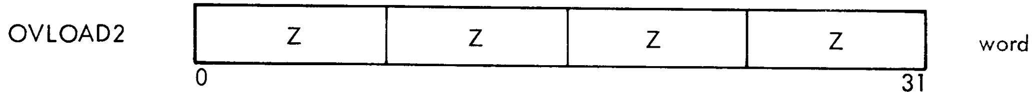


where Y is the DCT or RFT index of the permanent assignment (bit 0 = 0 if DCT index; bit 0 = 1 if RFT index). There is an OPLBS2 table for each active job, which is accessed by an address pointer in the associated job's JCB. The OPLBS2 table for the RBM job contains the permanent assignment of each operational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

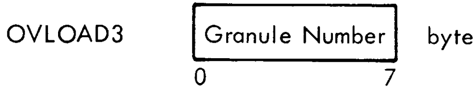
OVLOAD Table (for RBM Overlays Only)

The OVLOAD Table is a parallel table with the format





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

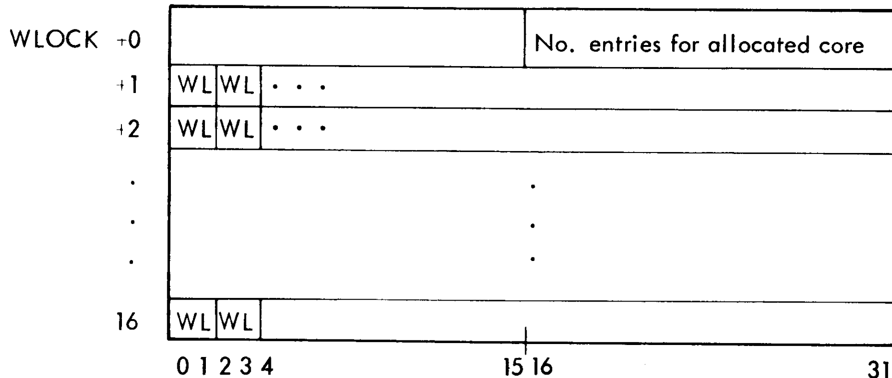


where the specified Granule Number is in the file RBM.

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

Write Lock Table (WLOCK)

Assuming no checkpoint, WLOCK contains write locks for the current core allocation. After a checkpoint the write locks will be restored from this table.



WLOCK + 1 always contains the write locks for the first 8K of memory. The table is always 17 words in length but the first word reflects the number of registers that must be output following a checkpoint.

RBM Dispatcher Level Inventory (RDLI)

RDLIPRIO

Priority
0–7

RDLISTI

STI Index
0–7

RDLITCB

STI Index	TCB Address
0–7	8–31

RDLIADD

RDL Interrupt location
0–15

RDLILVL1

Level Bits (RDL)
0–15

RDLIGRP1

000010————0	Group
0	15

RDLILVL2 (Zero if null)

Level Bits (STL)
0–15

RDLIGRP2 (Zero if null)

000010————0	GRP
0	15

where

RDLIPRIO is the priority, in internal byte format, to which RDL is connected. This is the RDL interrupt location -X'4F'. All tasks with RDLIPRIO + n < Priority < RDLIPRIO + n + 1 fall within level n. Legal secondary priority levels are C(RDLIPRIO) + 1. Entry 0 or 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 access controlled items.

Type

Serial in the JCB or linked from the JCB depending on space requirements or linked from the LMI. Monitor cell K:JAET contains the maximum number of ENQs allowed for each job on system-level controlled items. The system value S:TENQ is equated to the maximum number of ENQs allowed for each load module on job-level controlled items.

Logical Access

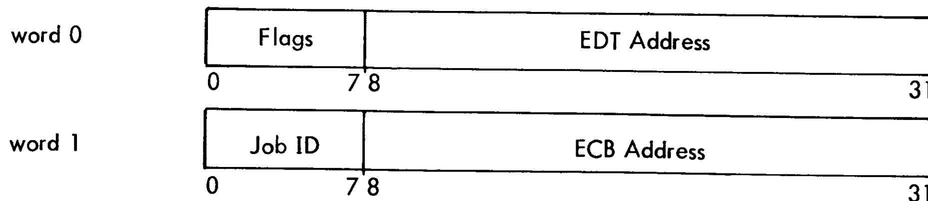
The AET is located via a pointer in a fixed position in the JCB or through a pointer in the LMI. Byte zero of the pointer word contains the number of words in AET.

Overview of Usage

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

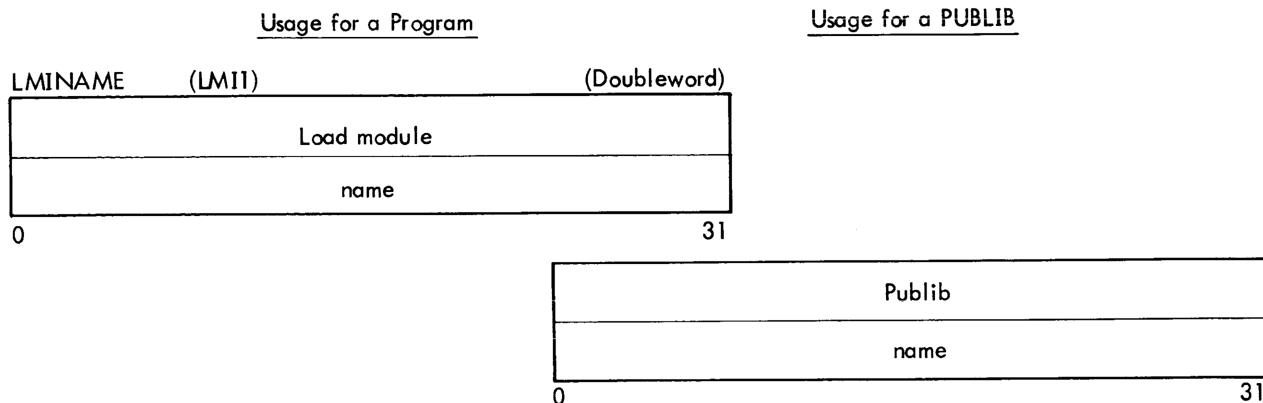
EDT Address: The location of the EDT of the controlled item which was enqueued.

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

Task-Controlled Tables

The tables shown in the subsection are task controlled, i.e., contain task related data. Figure 42 shows the overall relationship of the task-controlled tables and data.

Load Module Inventory (LMI)



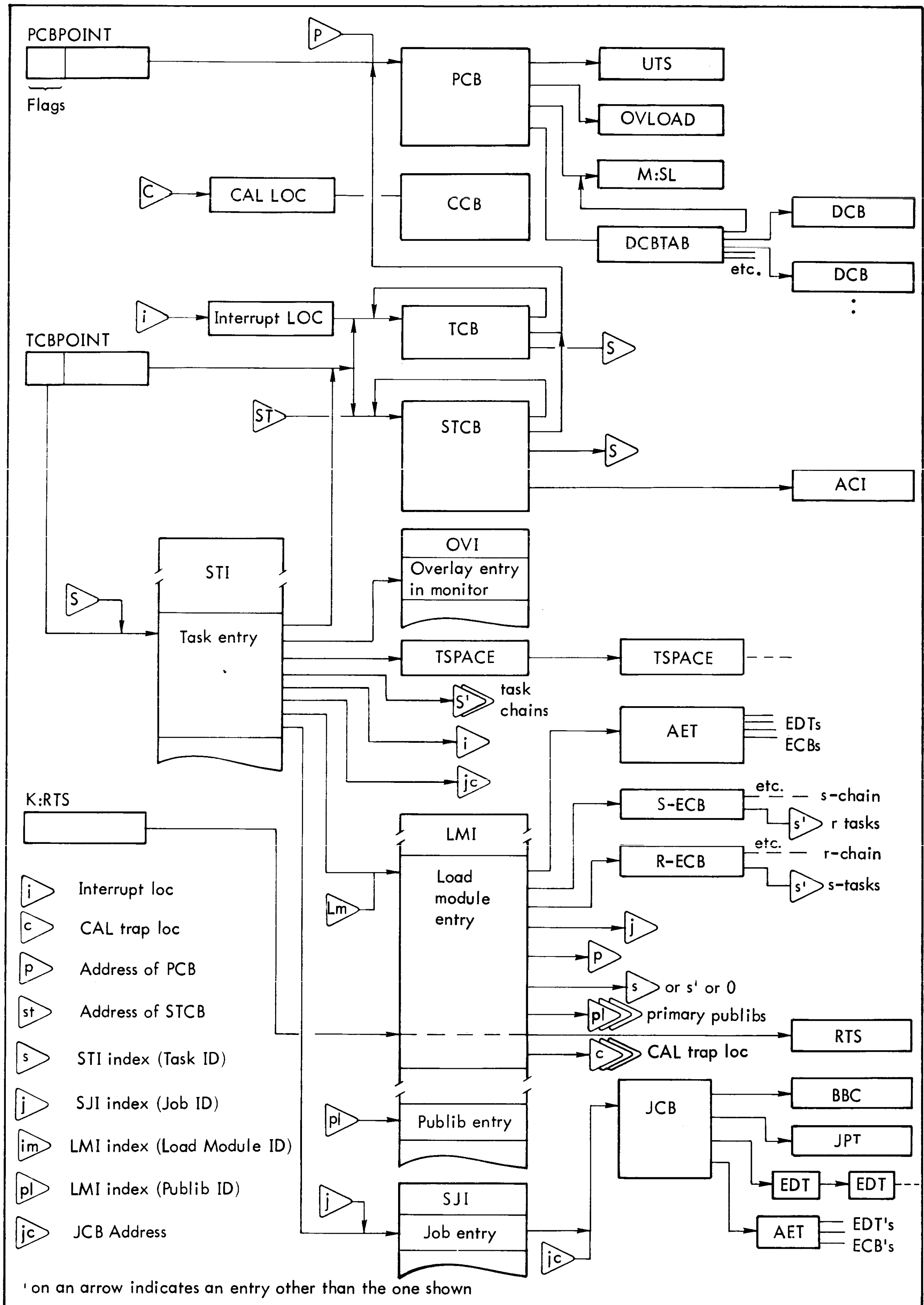
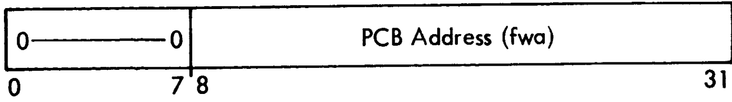
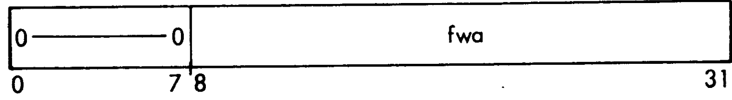


Figure 42. Relationship of Task Controlled Data

Usage for a Program | Usage for a PUBLIB

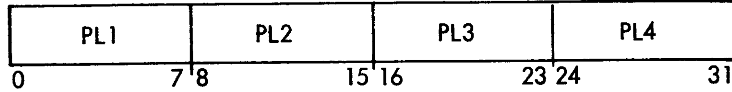
LMIPCB, LMIFWA (LMI2) (Word)

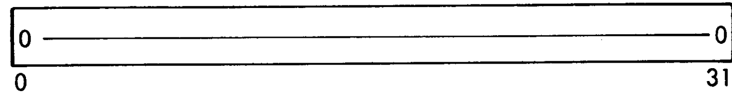




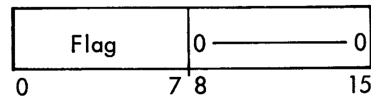
LMIJID, LMILWA (LMI3) (Word)

LMIPL, LMICTXT (LMI4) (Word)

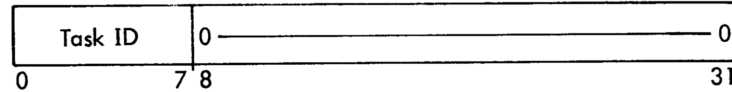


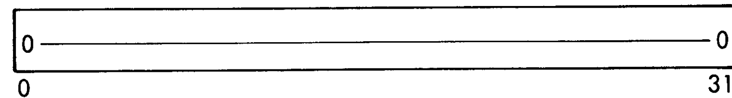


LMISTAT (LMI5) (Halfword)

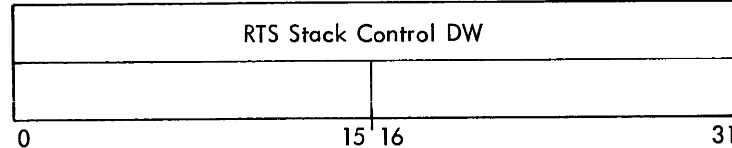


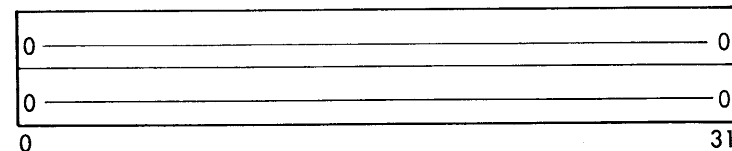
LMISDT (LMI6) (Word)



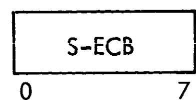


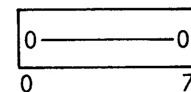
LMIRTS (LMI7) (Doubleword)





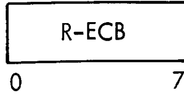
LMIMAXS (LMI8) (Byte)

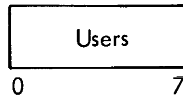




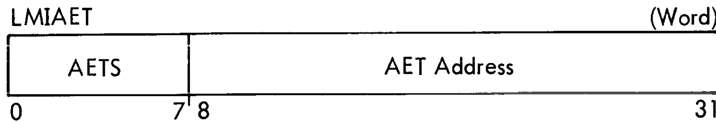
Usage for a Program | Usage for a PUBLIB

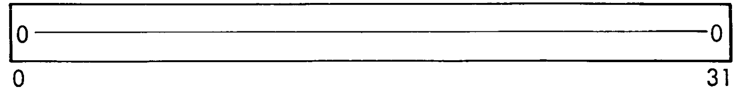
LMIMAXR, LMIUSE (LMI9) (Byte)



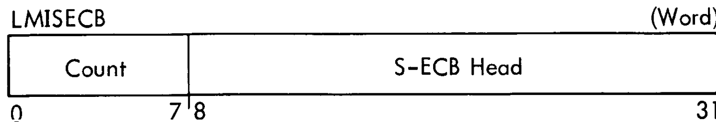


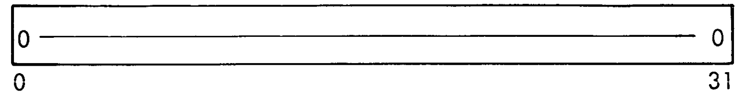
LMIAET (Word)



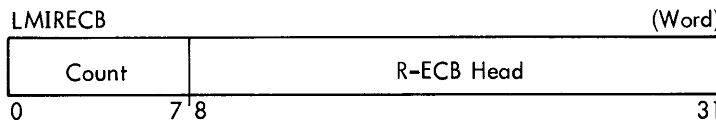


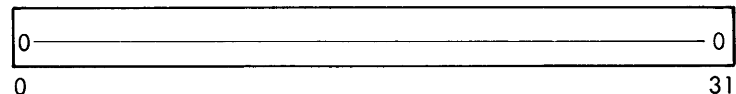
LMISECB (Word)





LMIRECB (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 initiation and remains unaltered during task execution.

LMIPCB, LMIFWA (LMI2)

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 user 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 user 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.

LMISTAT (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.
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 to be loaded.
9	PUBLIB that may be used by foreground.
10	PUBLIB that may be used by background.
11	Termination (normal or not) requested.
12	PUBLIB that is to be released.
13	Load module that is running.
14	Load module that is waiting for memory to load (RUN queued).

LMISDT (LMI6)

For user load modules – Task ID: STI index for the task if it is attached to a dispatcher. (This is the case for background, the RBM task, and foreground tasks during initiation.) Otherwise, zero.

LMIRTS (LMI7)

For user load modules – RTS Stack Control DW: The stack control doubleword for the load module's RBM 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.

LMI9, entry zero, contains the number of entries in LMI.

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 task-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.

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 RBM 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 monitor TSPACE

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:

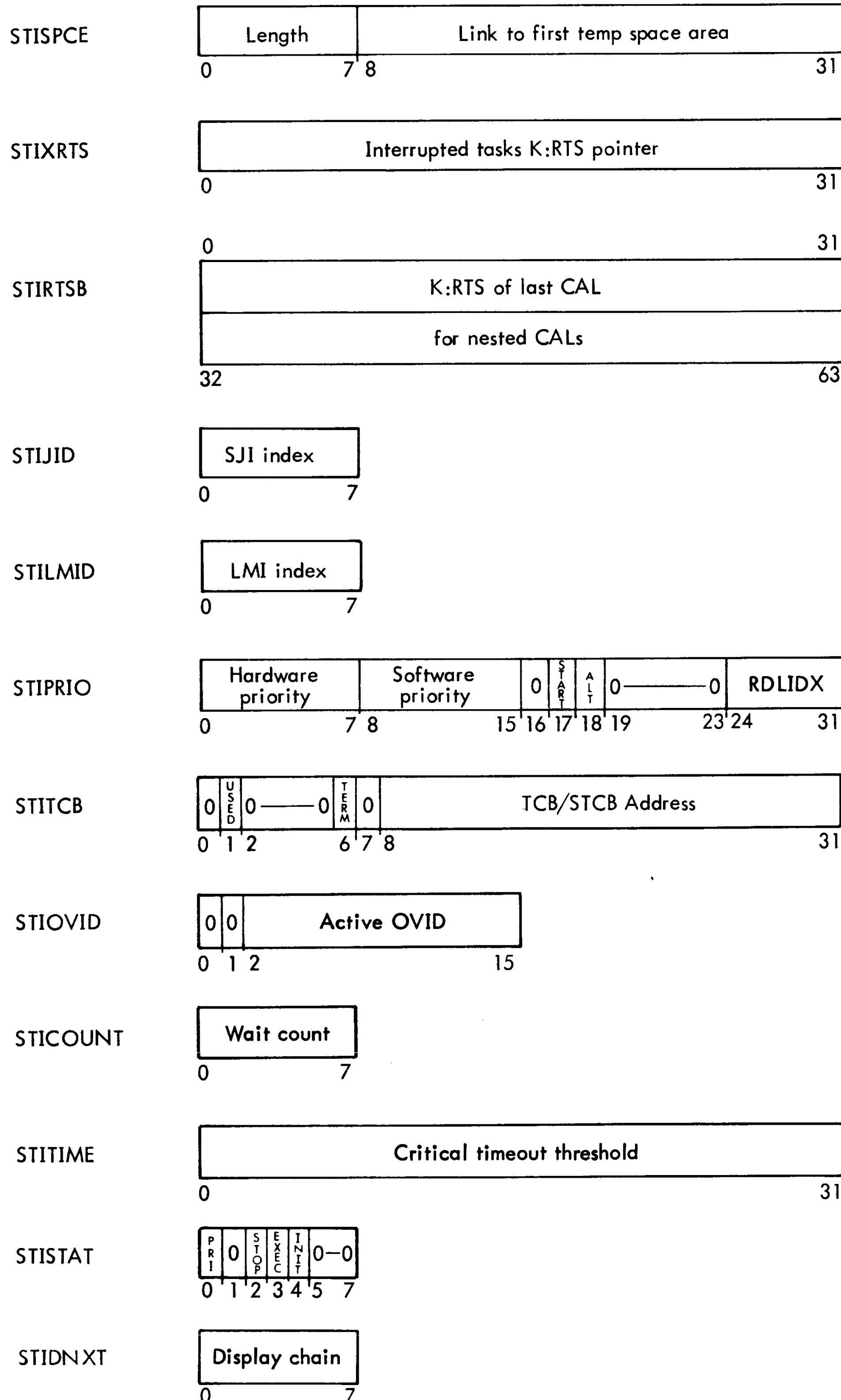
Primary tasks can be uniquely identified by a search for equality on the interrupt priority to which they are connected.

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 RBM tasks. The RBM 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.



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 RBM 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.

RDLIDX (RDLI index): Dispatcher ID.

STITCB

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

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

TCB/STCB address is TCB address if task is primary, STCB 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

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
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

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.

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 43 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 interrupt 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.

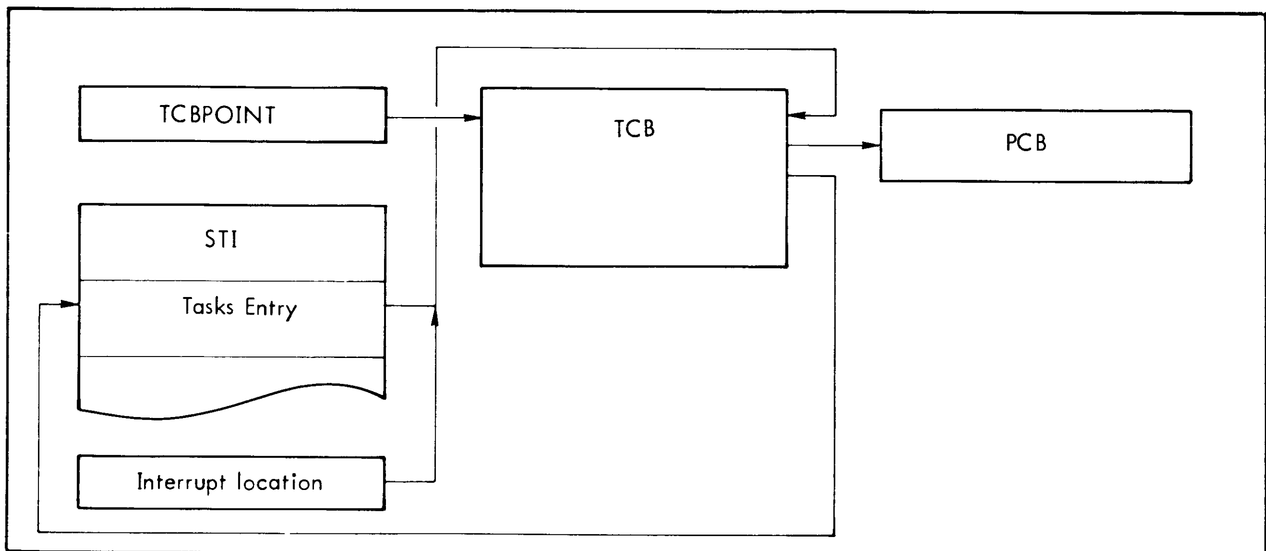
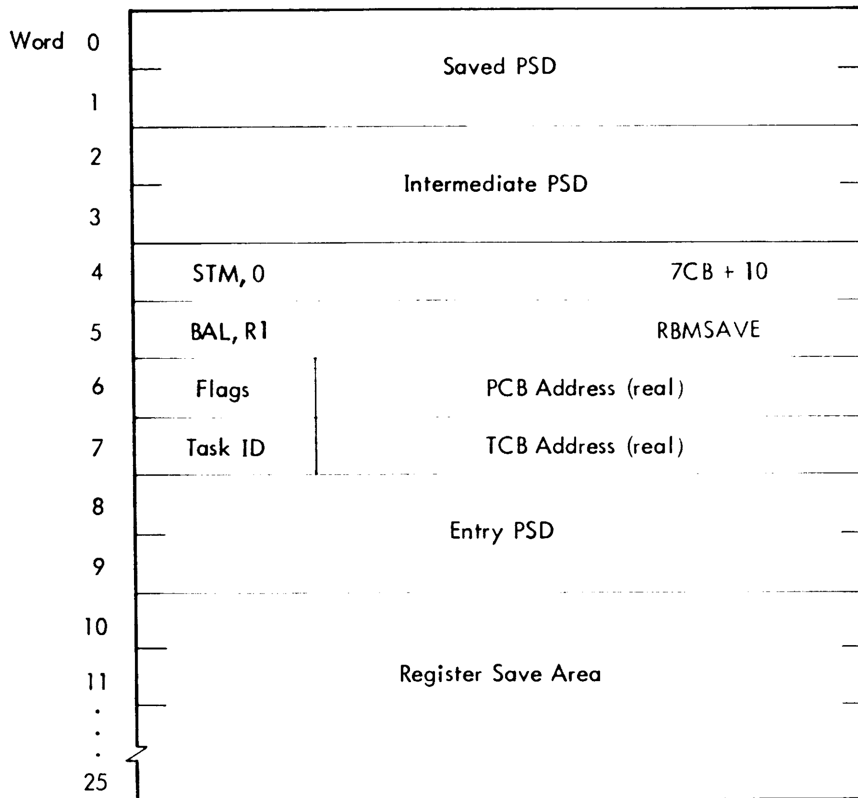


Figure 43. Relationship Between a Primary Task Control Block and Other Control Blocks

Task Control Block (TCB) Format



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 RBM task

1 = 0 for foreground task
= 1 for background task

2 = 0 for primary task
= 1 for secondary task

3 reserved

4 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 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 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 RBM Dispatcher Level (RDL) immediately above the Task Level (STD) (during execution only).

Figure 44 illustrates the logical links between the STCB and other system control data.

Overview of Usage

A user STCB is created by task initiation if the load module requested is secondary. RBM 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 RBM control functions and dispatcher during the life of the task. STCB space is released by task termination.

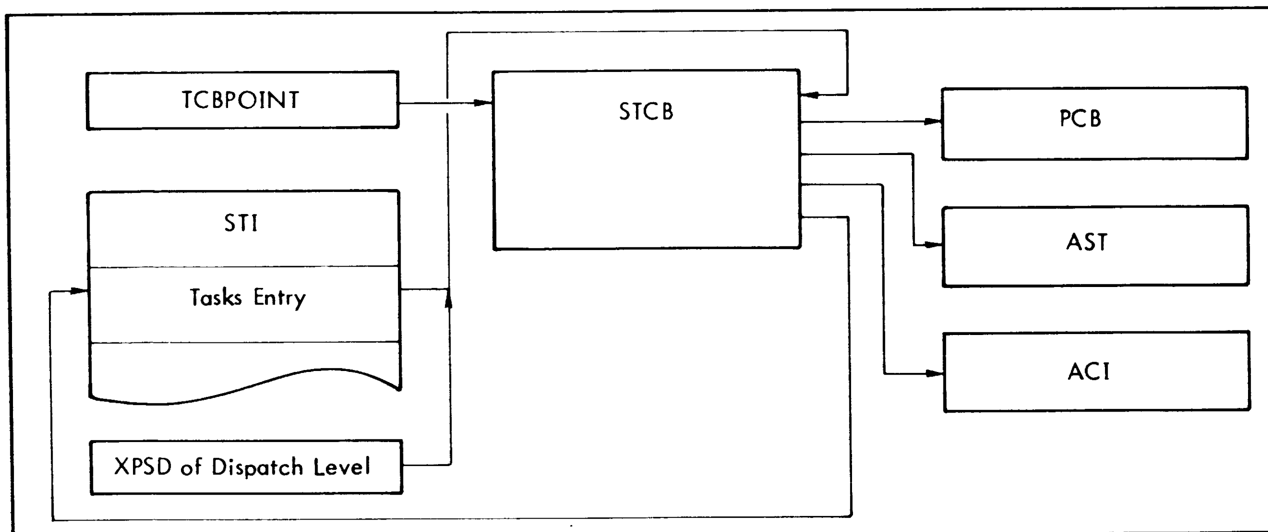


Figure 44. Relationship between Secondary Task Control Block and Other System Control Data

Secondary Task Control Block (STCB) Format

Word		
0–1	**Current PSD, Secondary Task**	
2–3	**Intermediate PSD**	
4	STM, 0	**STCB + 10**
5	BAL, R1	**RBMSAVE**
6	Flags	**PCB Address**
7	Task ID	**STCB Address**
8–9	**Entry PSD to Post Dispatch Processing**	
10 ... 25	Current Registers, Secondary Task	
26	–	
27	–	
28	RDL Group Code	RDL Level Bit
29	–	
30–31	Alternate PSD	

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.

IA load module entry address.

Write Key = 10 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 and are not altered.

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

Bit 0 = 0 for user task
= 1 for RBM task

1 = 0 for foreground task
= 1 for background task

2 = 1 for secondary task

3 reserved

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.

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. "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.

Words 26, 27 spare.

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.

Job-Controlled Tables

The tables shown in this subsection are job controlled, i.e., contain data associated with the job level of control. Figure 45 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 RBM 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 RBM 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 RBM is the first entry. The SJI entry for the background is the second entry (i.e., the RBM SJI index is 1; the background SJI index is 2).

Overview of Usage

The SJI space is allocated by SYSGEN from RBM 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 RBM system job (not one of the number specified). The background entry is also always made and is the default (1 entry plus the RBM entry) if no limit is specified.

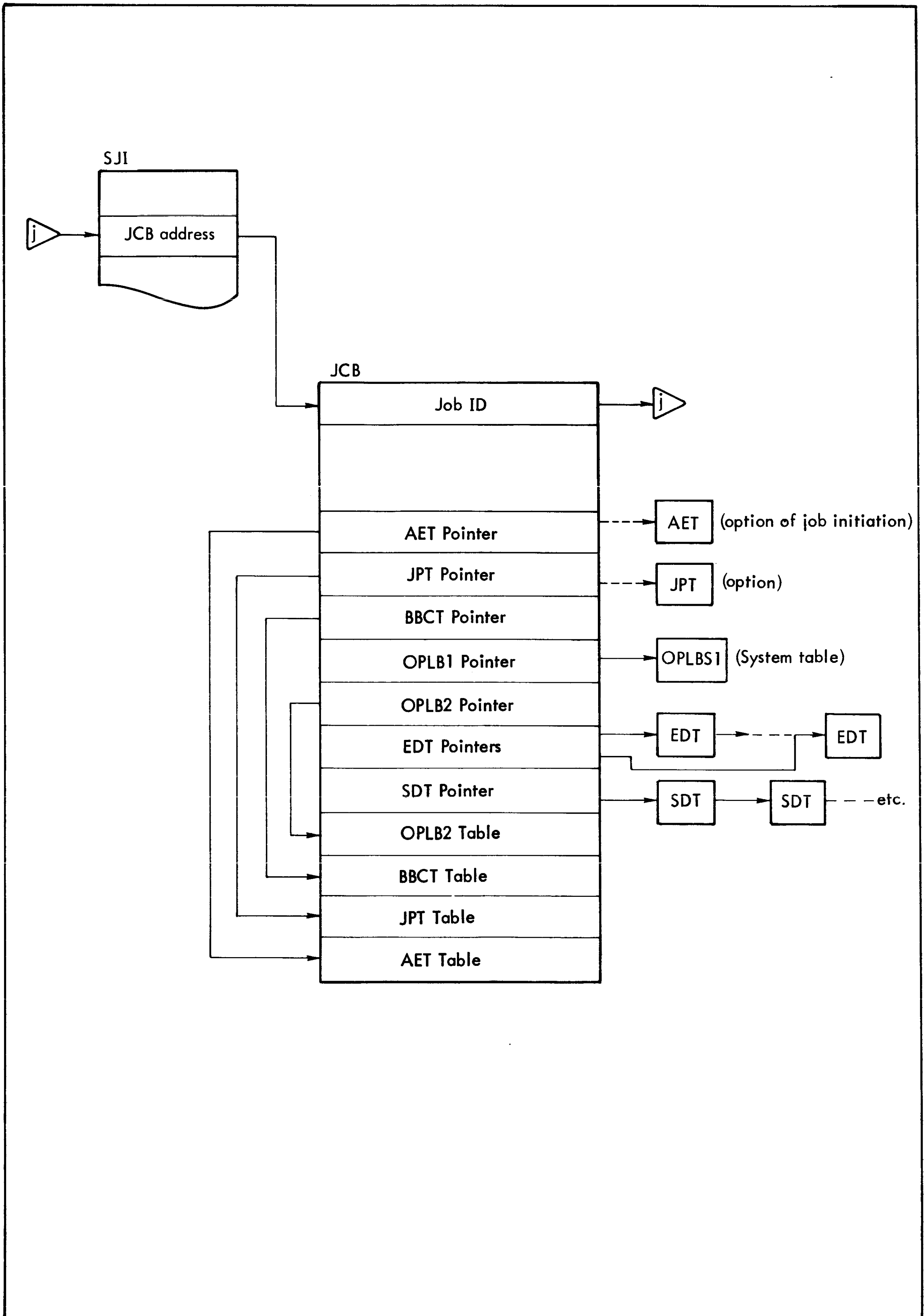


Figure 45. Relationship of Job-Associated Control Tables

The RBM and background entries are initialized by RBM 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	Bits 0: 0; 1–7: No. of words in JCB; 8–31: JCB Address
SJI2	Bits 0–31 / 32–63: EBCDIC job name
SJI3	Bits 0: 0; 1: L; 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 RBM 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 RBM TSPACE. Space is acquired when the job is initiated and released when the job is terminated. The JCBs for the RBM job and the background are established in RBM 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 0–31)
0	bits 0–1: 0 0; bit 2: A; bit 3: J; bits 4–7: 0——0; bits 8–13: (blank); bit 14: T; bit 15: S; bits 16–23: Job Priority; bits 24–31: Job ID
1	0 —— 0
2	Flags (bits 0–7); JTrap Address (bits 8–31)
3	0 —— 0
4	0; No. Entries; OPLB1 Pointer → OPLBS1 system table
5	0; No. Entries; OPLB2 Pointer (points to word 37)
6	0; No. Entries; BBCT Pointer
7	0; Max. length; JPT Pointer
8	0; Max. length; AET Pointer
9	EDT Pointer – head
10	EDT Pointer – tail
11	0 —— 0
12 ... 36	BBCT Blocking Buffer Control Table (25 words)
37 ... N	OPLB2 Operational Label RFT or DCT index byte table
N+1 ... M	JPT Job Program Table (quadruple-word entries, DW bound)
M+1 ... L	AET Enqueue Table for job level enqueues (DW entries, DW bound)

Note on JPT and AET: These tables may not be contiguous to the JPT or to each other, in order that dynamic space may be more efficiently used.

Word 0	Flags	
A	(bit 2)	Indicator of whether AET is contained in fixed portion of JCB or is external to JCB: = 0 AET in fixed portion of JCB. = 1 AET linked from JCB.
J	(bit 3)	Indicator of whether JPT is contained in fixed portion of JCB or is external to JCB: = 0 JPT in fixed portion of JCB. = 1 JPT linked from JCB.
T	(bit 14)	Job-being-terminated bit.
S	(bit 15)	Job-being-initiated bit.

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.

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 RBM location S:EDT (system level ENQs and all device resources) or in the JCB (job level ENQs). Figure 46 shows the overall relationship between system tables that indirectly or directly affect the EDT.

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.

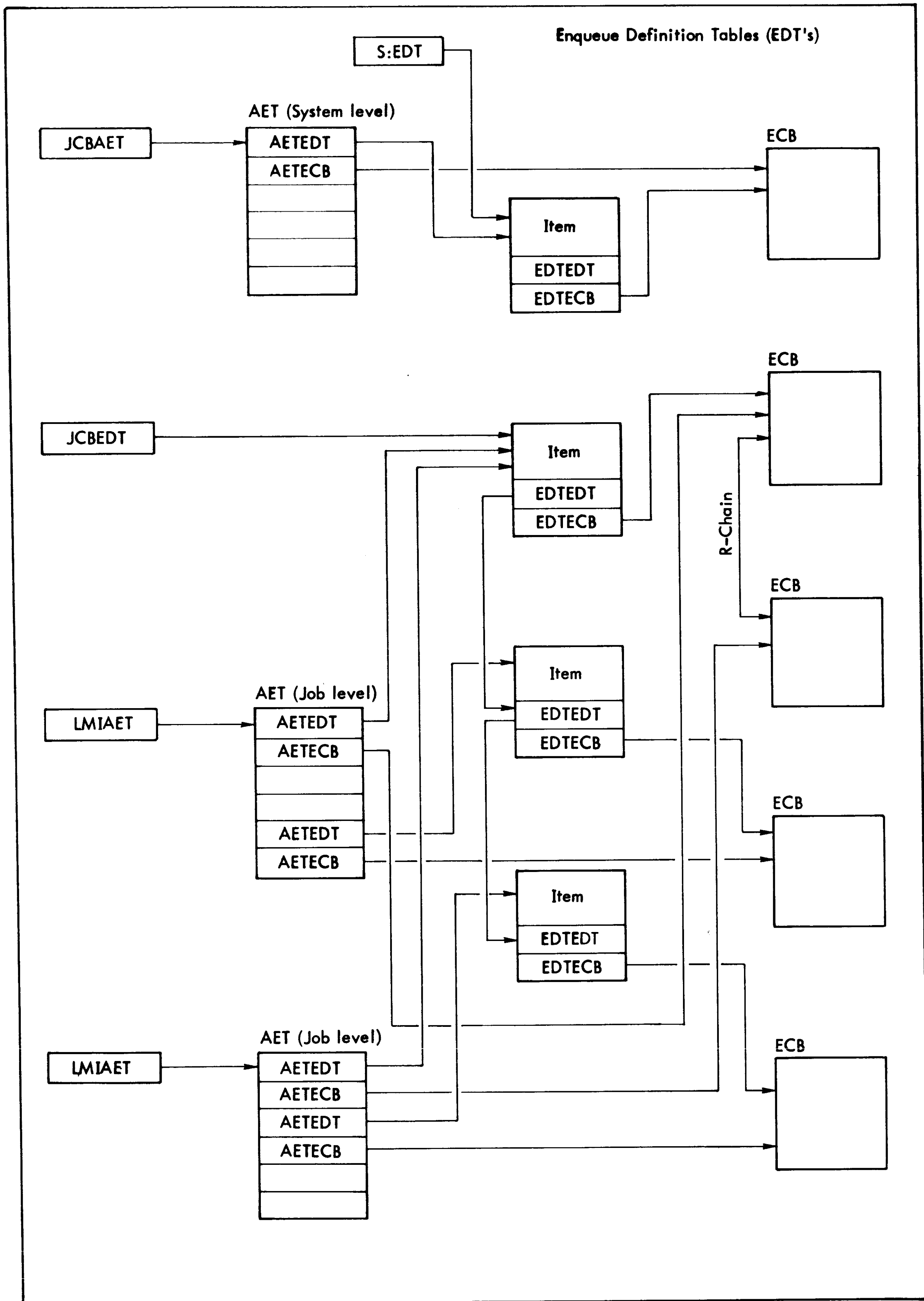
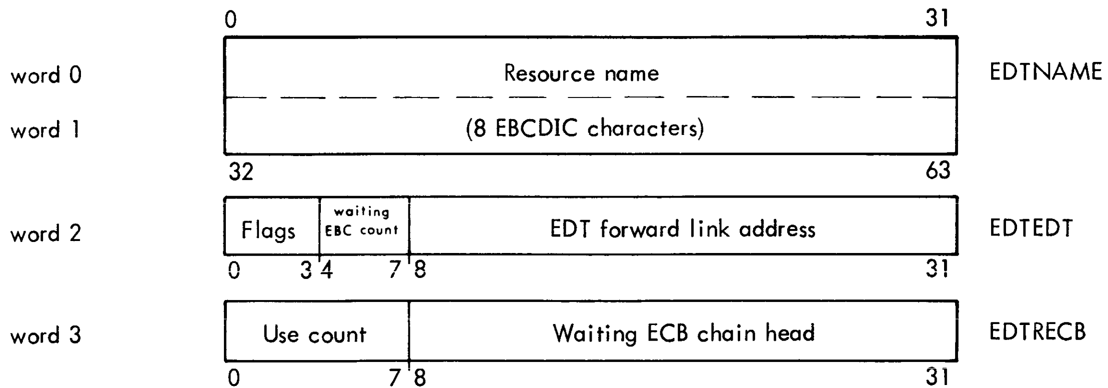


Figure 46. Enqueue/Dequeue Table Relationship

Enqueue Definition Table (EDT) Format



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.

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 that currently have acquired use of the item. If the enqueue is exclusive, this count will be 1. If the enqueue 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.

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

Word	byte 0	1 – 3
0	F | 0 | 0 | L | P | 0 | 0	Task First Word Address
1	MSECB	Task Last Word Address
2	MRECB	Task Entry Word Address
3	MENQ	Root Part one VM BL
4	NSEGS	Root Part one VM WO
5	0	Root Part one LM BL
6		Root Part two VM BL
7		Root Part two VM WO
8		Root Part two LM BL
9		Root Part two LM GO
A		0 ——— 0
B	0	0 ——— 0
C	Stack control doubleword prototype	
D	for the RBM temp stack	
E	Names of PUBLIB load modules required (up to 5 at 8 bytes each)	
	0 ——— 0	
	Remainder of granule 0 is unused	

where

F = 0 for a background task.
  = 1 for a foreground task.

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 solicited ECBs;
X'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

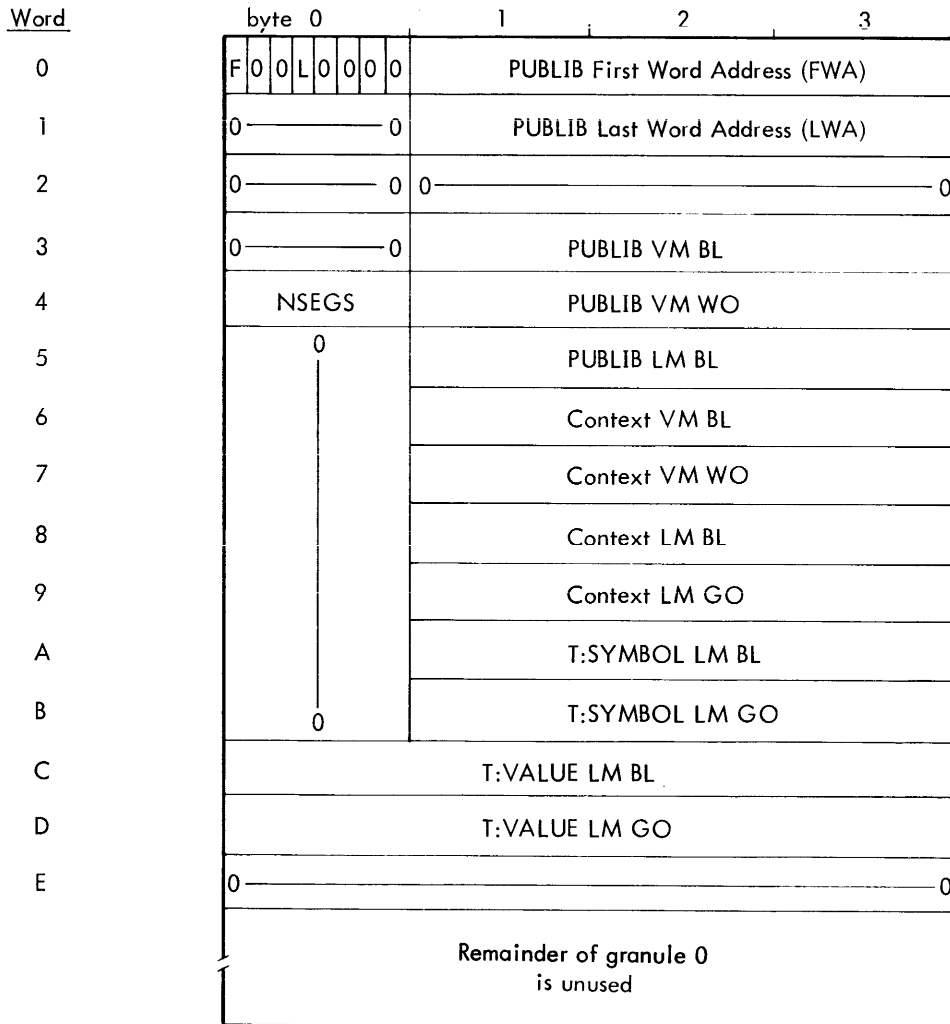
GO Granule origin

LM Load module

VM Virtual memory

WO Word origin

PUBLIB Load Module Header



where

F = 1 for a foreground load module.

L = 1 for a PUBLIB load module (not a task load module).

NSEGS = 1 for PUBLIB only; = 2 for PUBLIB with context segment.

Legend:

BL Byte length

GO Granule origin

LM Load module

VM Virtual memory

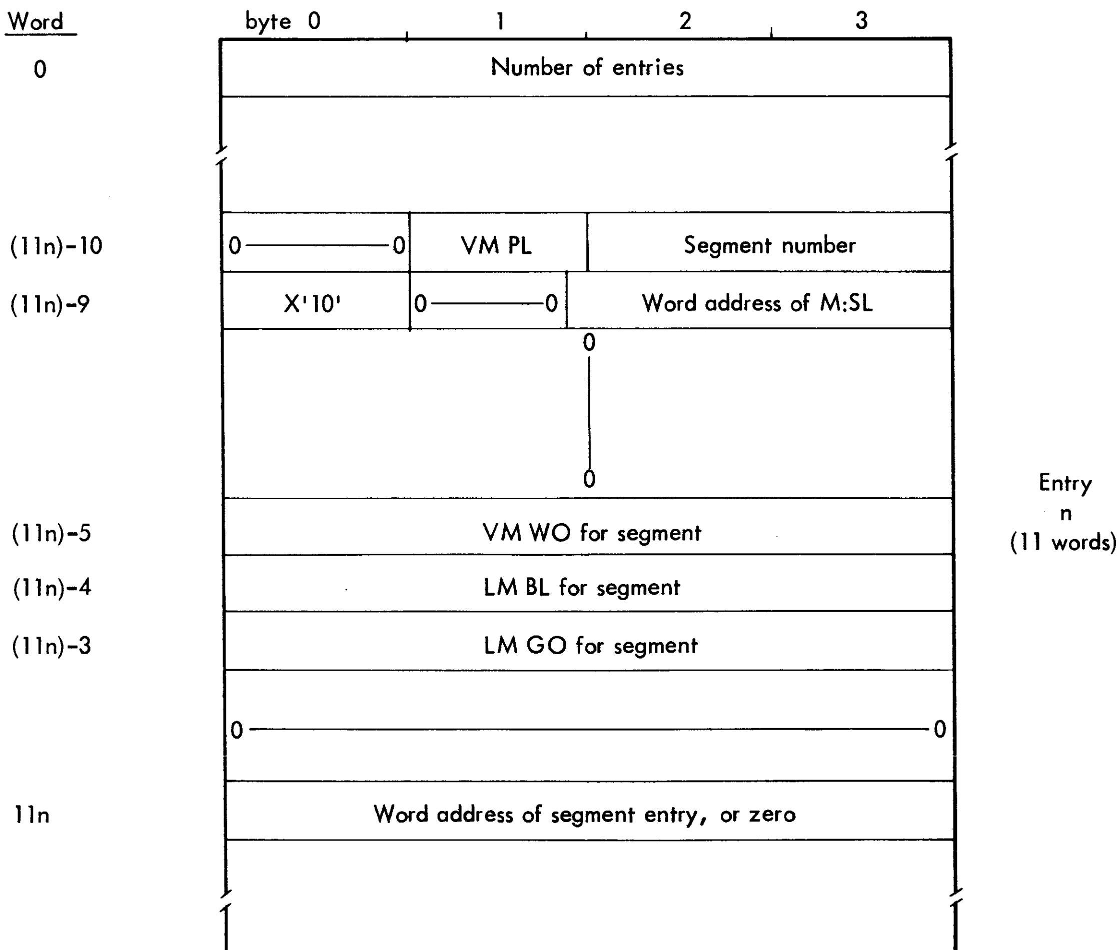
WO Word origin

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.



VM = Virtual Memory
BL = Byte Length
WO = Word Origin

LM = Load Module
PL = Page Length
GO = Granule Origin

9. OVERLAY LOADER

Overlay Structure

The Overlay Loader is itself an overlayed program, with a root and the six segments illustrated in Figure 47.

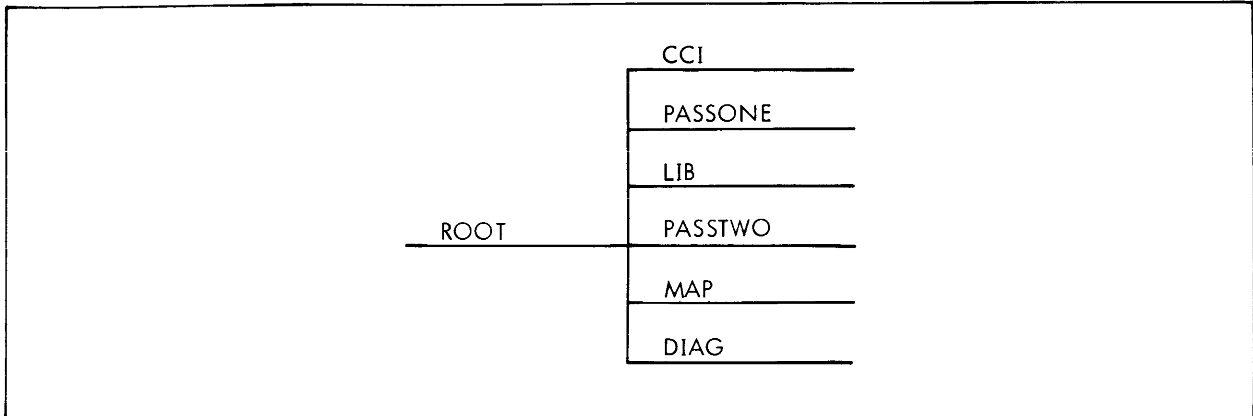


Figure 47. Overlay Structure of the Overlay Loader

The functions of the Root and segments are given in Table 5.

Table 5. 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 RBM 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 RAD 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 48.

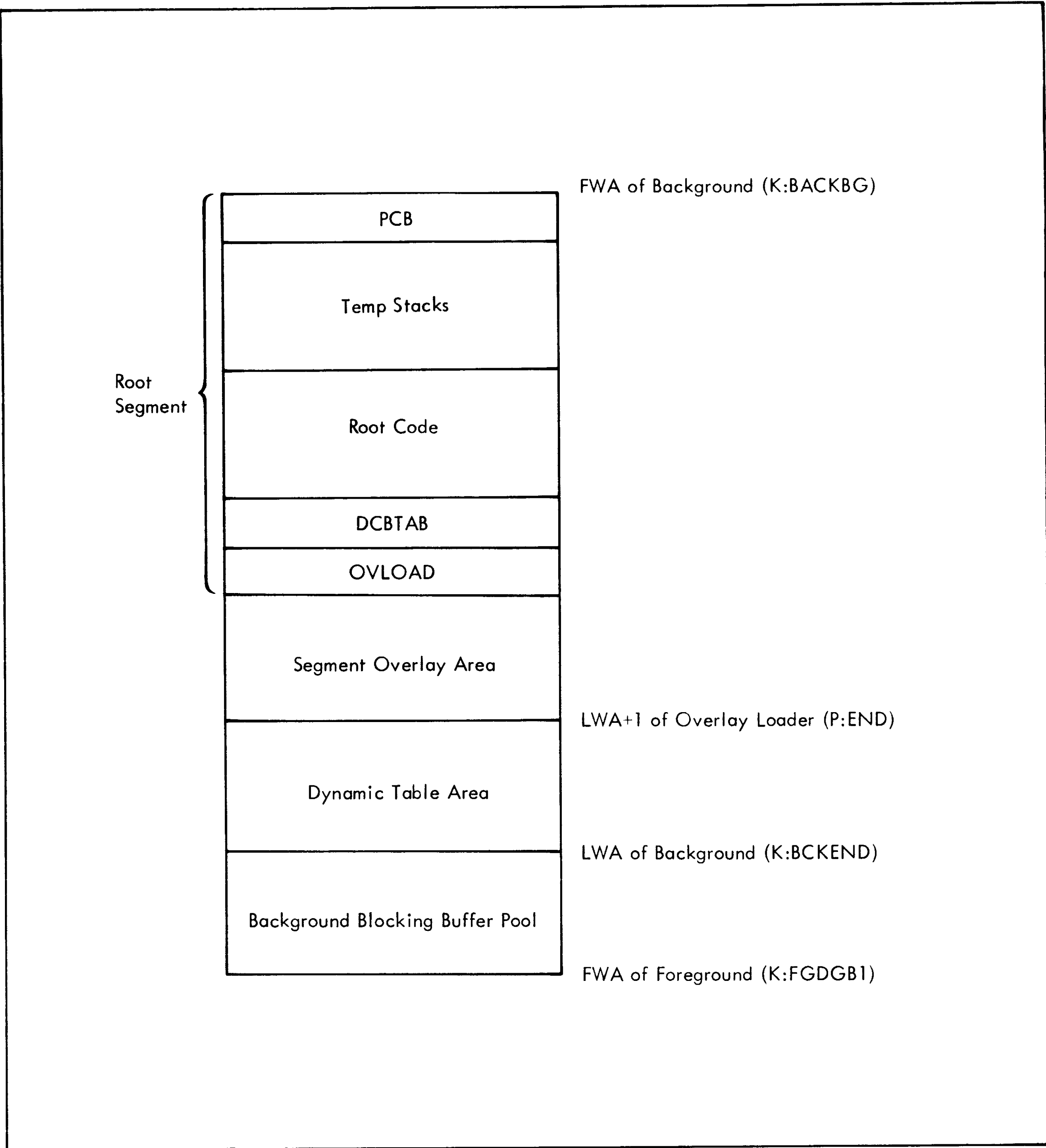


Figure 48. Overlay Loader Core Layout

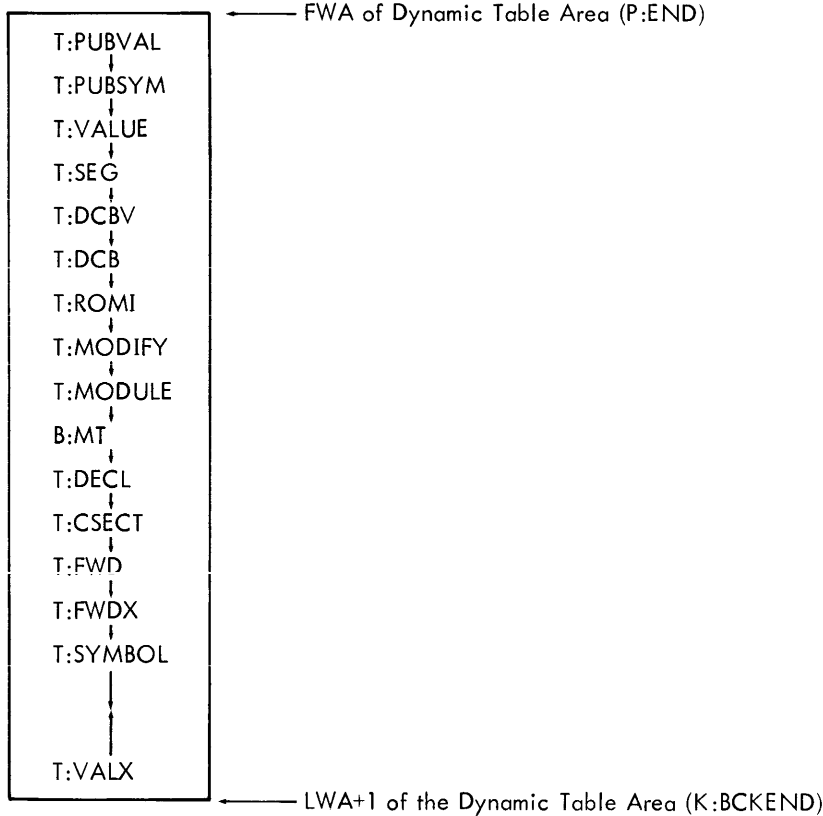
Dynamic Table Area

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

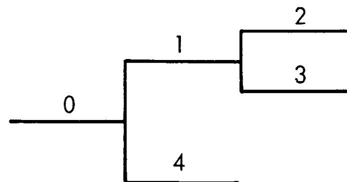
During the first pass over the object modules, the 16 table areas have a fixed order as follows:



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:

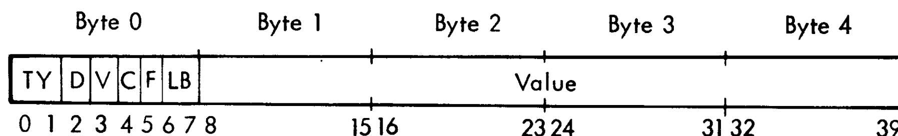
For Root	For Seg 1	For Seg 2	For Seg 3	For Seg 4
0	0	0	0	0
	1	1	1	4
		2	3	

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 RAD 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.

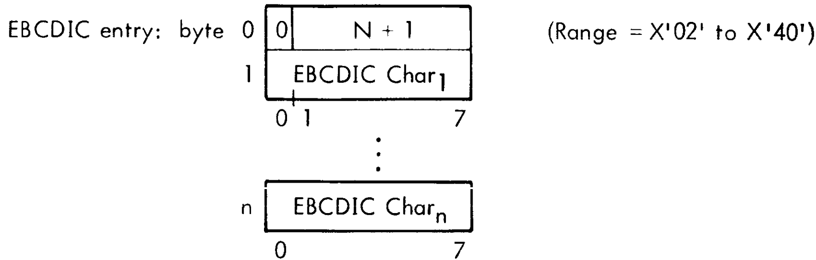
Value is initially set to zero; usage is dependent upon D, V, and C flags.

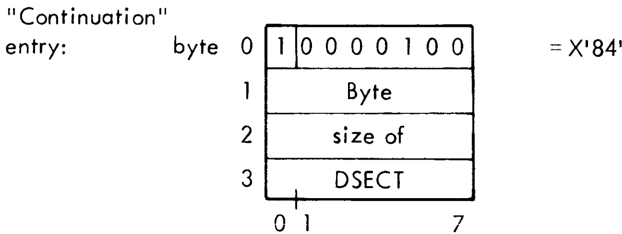
Since the T:VALUE entries are kept as small as possible, unused bit combinations are reserved to define the following two intermediate external types:

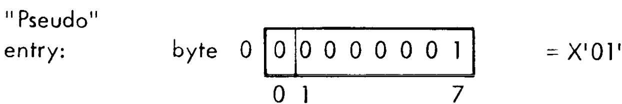
1. If TY = PREF, C = 0, and V = 1, the external is an "excluded pref" which means that the PREF will cause neither library loading nor linkage (including the Public Library). Instead, the PREF will be satisfied by a DEF in a segment further up the path.
2. If TY = DSECT, D = 1, and V = 0, the external 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:







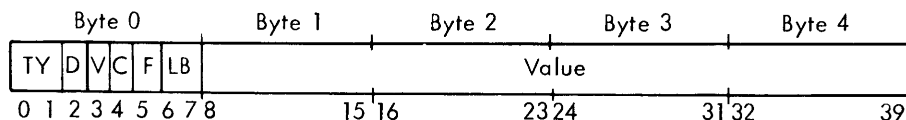
Note that the first byte contains the byte count of the entry (in bits 1-7).

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 !OLOAD 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.

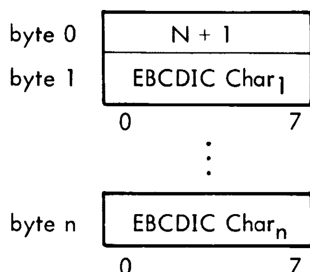
F = 0 not a duplicate DEF.

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



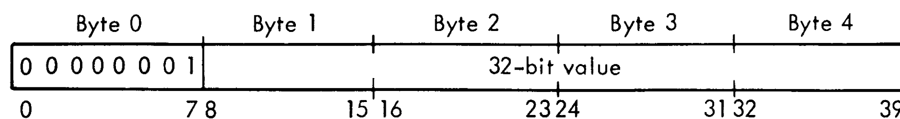
T:VALX

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.

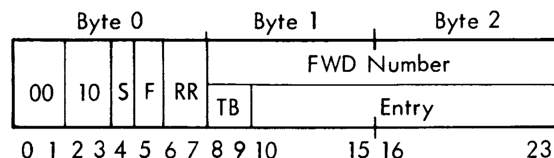
Expressions have a variable size and are made up of expression bytes, combined in any order. The formats for the T:VALX expression bytes (slightly different than the object language) are

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.

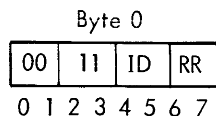
RR = 01 halfword address resolution.

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).

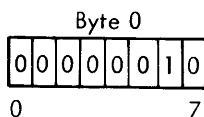
RR = 00 byte address resolution.

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	0–7	8–15	16–23	24–31
0	TY (0–1), D (2), V (3), C (4), F (5), LB (6–7)	(unused, 8–12); Byte Address (13–31)		
1	E1	E2	E3	E4
2	E5	E6	E7	E8

Bit positions: 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	0–15	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 (0), L (1), W (2), F (3), I (4), M (5–6), S (7–8), P (9), (10–12), E A (13–14)	Entry Address (15–31)
8	Byte Length of Library Routines in SEG	
9	Byte length of load-module image of segment	

Bit positions: 0 1 2 3 4 5 6 7 8 9 12 13 14 15 16 31

where

Gran no. the granule number in the RAD file where the table begins. If the RAD 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.
= 0	error severity level reset on every ROM in the segment.
L = 1	load error (duplicate DEFs, unsatisfied REFs, etc.).
= 0	no loading errors in SEG.
W = 1	T:VALUE (I) and T:SYMBOL (I) output on X4, X5.
= 0	T:VALUE (I) and T:SYMBOL (I) not output on X4, X5.
F = 1	segment is fixed in real memory (FIX option).
= 0	segment may be mapped on any available real memory.
I = 1	segment is to be initially loaded with the root (ILOAD option).
= 0	segment will be loaded only on explicit request.
M = 00	segment is any-access.
= 01	segment is read-and-execute.
= 10	segment is read-only.
= 11	segment is no-access.
S = 00	segment is nonsharable.
= 01	segment is job-level sharable.
= 10	segment is system-level sharable.
= 11	unused.
P = 1	segment must be pre-loaded sharable (PRELOAD option).
= 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.
= 01	unused.

EA = 10 SEG's entry address input from CC and value in bits 15-31 is the entry address (in words).

= 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.

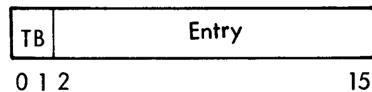
B:MT

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 TempFile on the RAD 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 RAD 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).

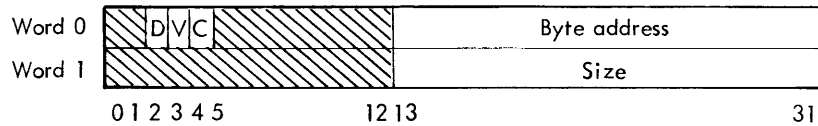
TB = 11 Entry points to T:PUBSYM/T:PUBVAL

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.

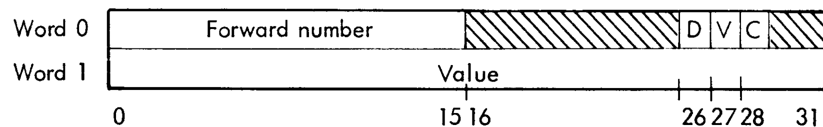
Byte address first byte address of the control section.

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).

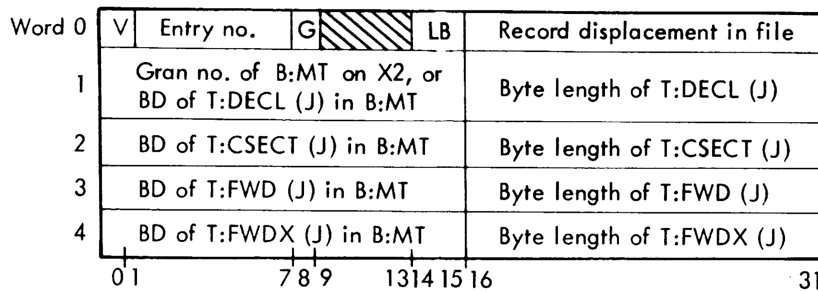
C = 0 value is a positive or negative address with byte resolution (only meaningful if V = 1).

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



where

V = 1 Entry no. in bits 1-7 points to T:DCBV.

V = 0 Entry no. in bits 1-7 points to T:DCBF.

Entry no. the entry number of the DCB (in either T:DCBV or T:DCBF) that points to the RAD file where the ROM is located.

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 RAD.

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 RAD).

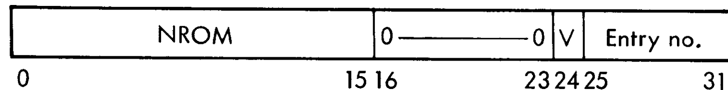
LB = 10 ROM from User Library (FP area of RAD).

Record displacement in the MODULE file (only meaningful for library ROMs.)

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 RAD files (built by CCI):



where

NROM is the number of ROMs to input or contains -5, which means to input until !EOD is encountered. This halfword is used as a decreasing counter by PASSONE and eventually equals zero.

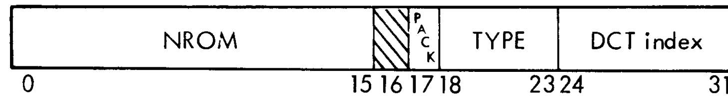
Bits 16-23 always equal zero to specify entry type.

V = 1 Entry no. in bits 25-31 points to T:DCBV.

V = 0 Entry no. in bits 25-31 points to T:DCBF.

Entry no. is the entry number of the DCB (in either T:DCBV or T:DCBF) that points to the RAD file where the ROM is located.

Entry for ROMs input from a specified device or OPLB (built by CCI):



where

Bits 16-23 always equal nonzero to specify entry type.

NROM is described above.

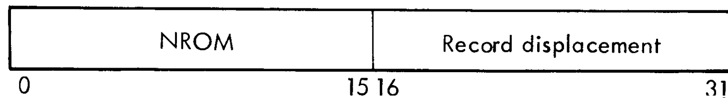
PACK is the PACK flag (bit 22 of word 0) in DCB.

TYPE is the device type code (bits 18-23 of word 1) in DCB.

DCT index is the DCT index of the device (bits 24-31 of word 1) in DCB.

PASSONE will store the information in F:DEVICE and input the ROMs via that DCB. Note that OPLBs are converted to their assigned devices.

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 DCBs assigned to the various RAD files specified (other than GO) on the input options of the :ROOT and :SEG, or :PUBLIB control commands. One DCB is created for each unique file name specified. T:DCB is resident at all times. T:DCBV entries are numbered from 1 to n, and have the standard seven-word DCB format.

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 absoutely 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 49.

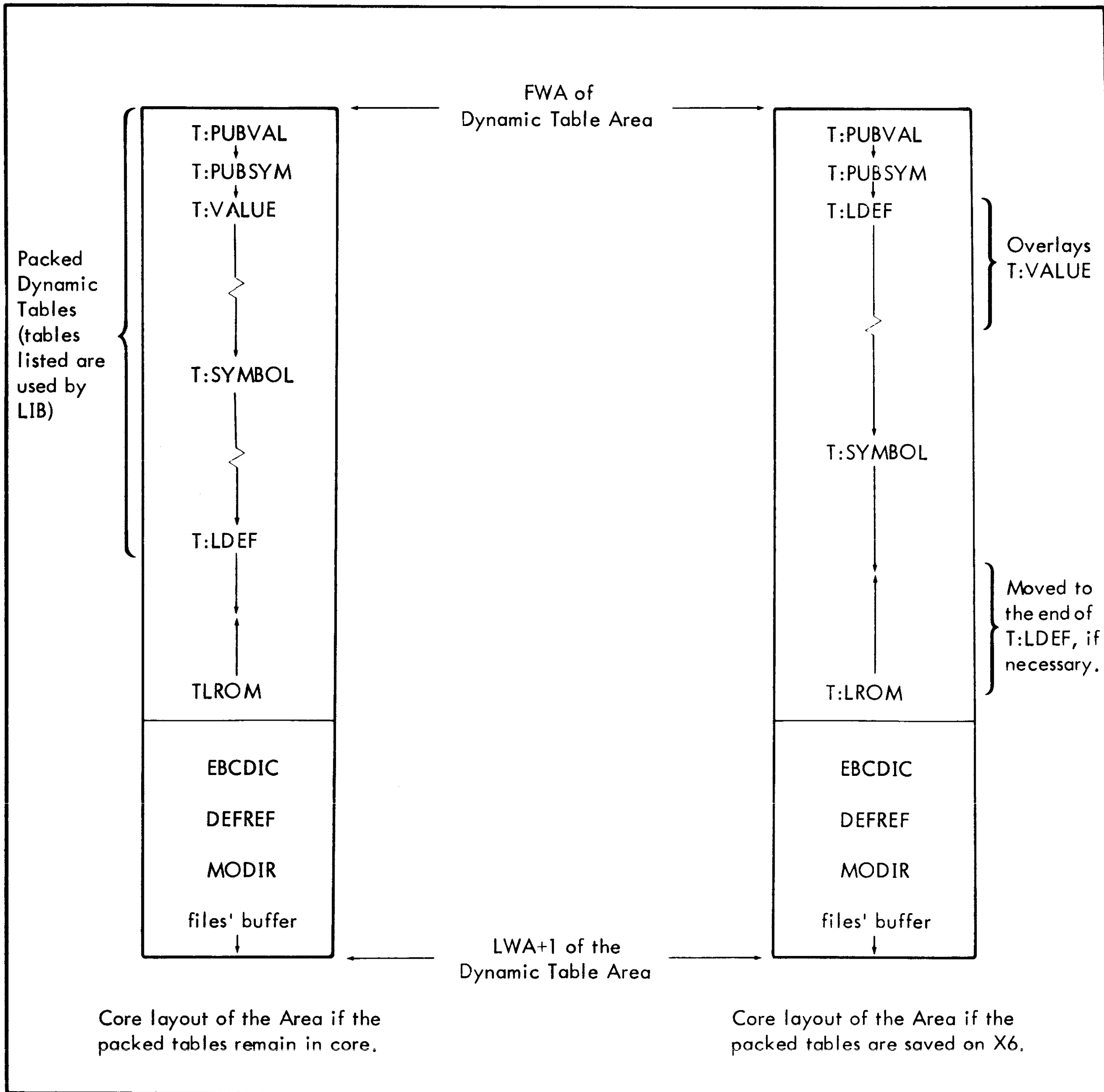


Figure 49. LIB Reorganization of Dynamic Table Area

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.
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

	T:LDEF entry count	
entry 0	T:LDEF entry count	
	0	15
⋮	⋮	
entry n	DR | Value	
	0 1 2	15

where

DR = 00 null entry.

DR = 01 DEF or excluded PREF.

DR = 10 unsatisfied PREF.

DR = 11 DSECT.

Value entry number in 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

entry n	Value	
↑	0 ↑	15
⋮	⋮	
entry 0	T:LROM entry count	
	0	15

where

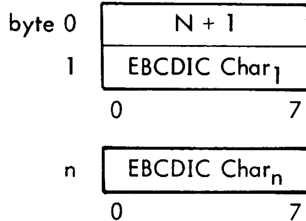
value halfword offset of the library ROM's entry in the MODIR file, which is later changed to the starting record number of the ROM in the MODULE file.

MODULE File

The MODULE file is a blocked sequential file, with 120 bytes per record, that contains the Library's ROMs.

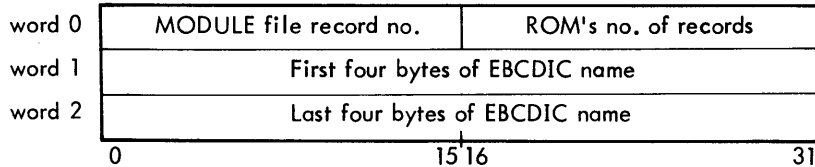
EBCDIC File

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



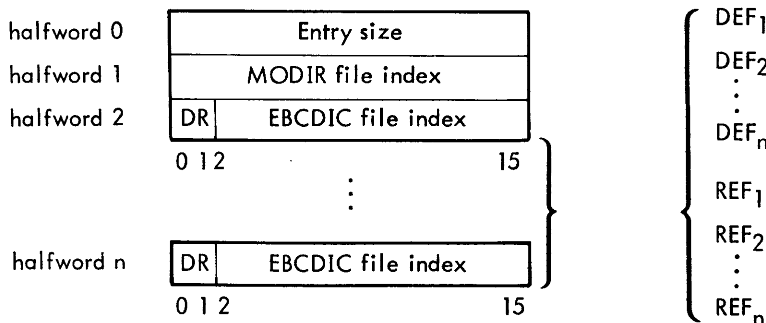
MODIR File

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 50.

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

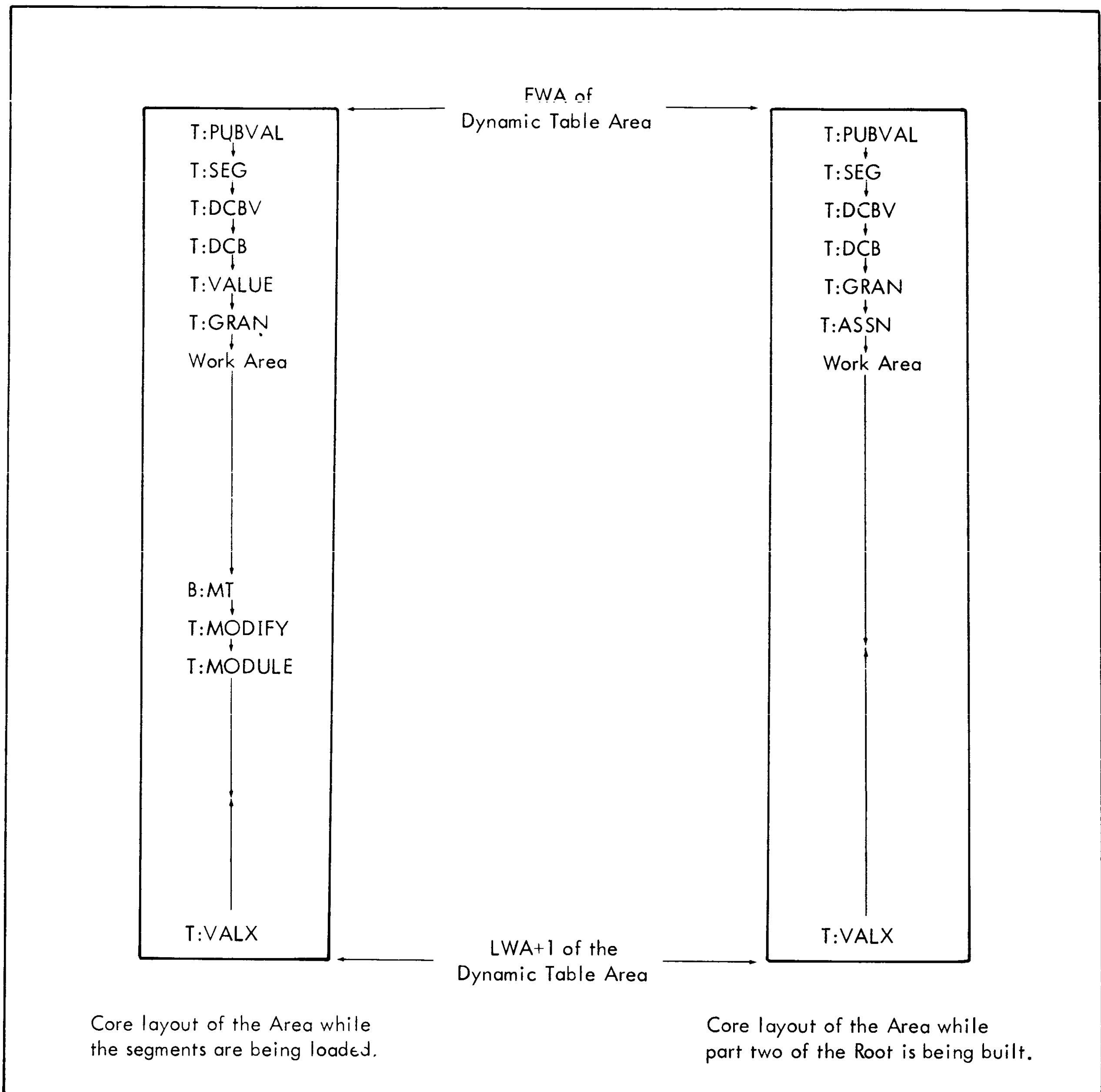
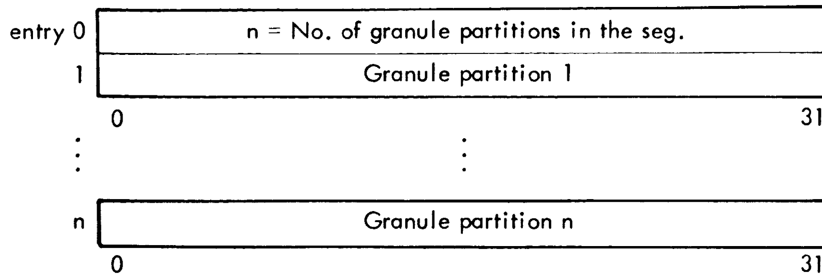


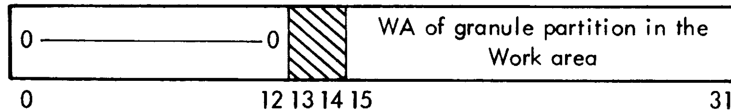
Figure 50. PASSTWO Reorganization of Dynamic Table Area

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:

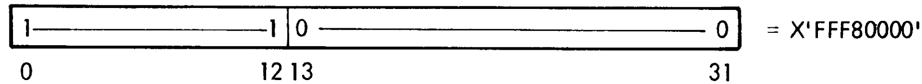


T:GRAN entries have a fixed size of one word with three different formats.

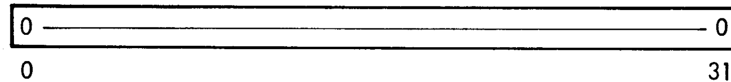
If the granule partition exists in the Work Area:



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

Word	Contents (bits 0–31)
0	Byte address of DCB's execution location
1	Word address of DCB's entry in T:DCB
2	Changes for word 0 of DCB
3	Mask for word 0 of DCB
4	Changes for word 1 of DCB
5	Mask for word 1 of DCB
6	Changes for word 3 of DCB
7	Mask for word 3 of DCB
8	First four EBCDIC bytes of file name or zero
9	Last four EBCDIC bytes of file name or zero

MAP Use of Dynamic Table Area

MAP moves the resident tables T:SEG andT: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 51.

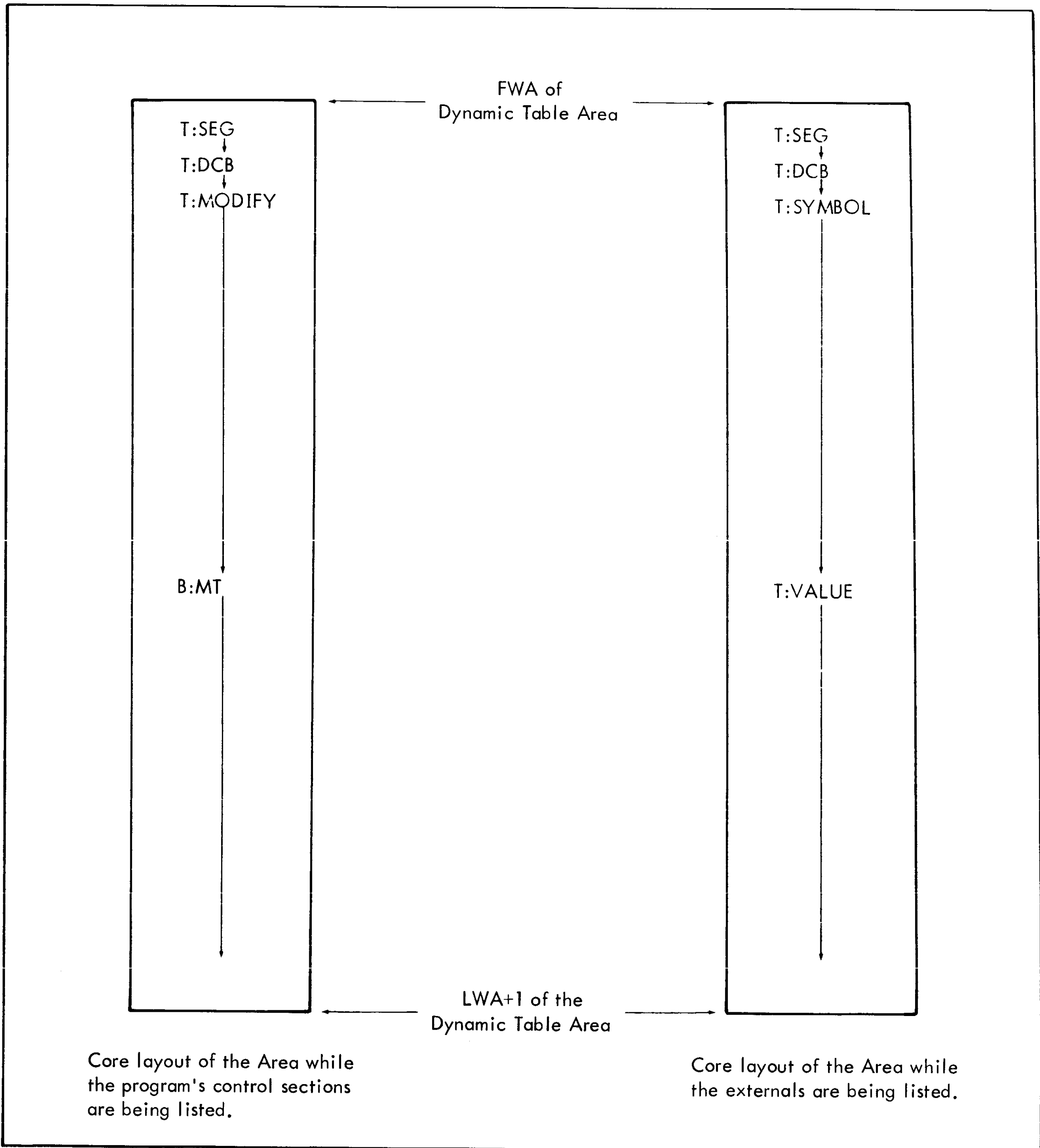


Figure 51. MAP Table Reference

DIAG Use of Dynamic Table Area

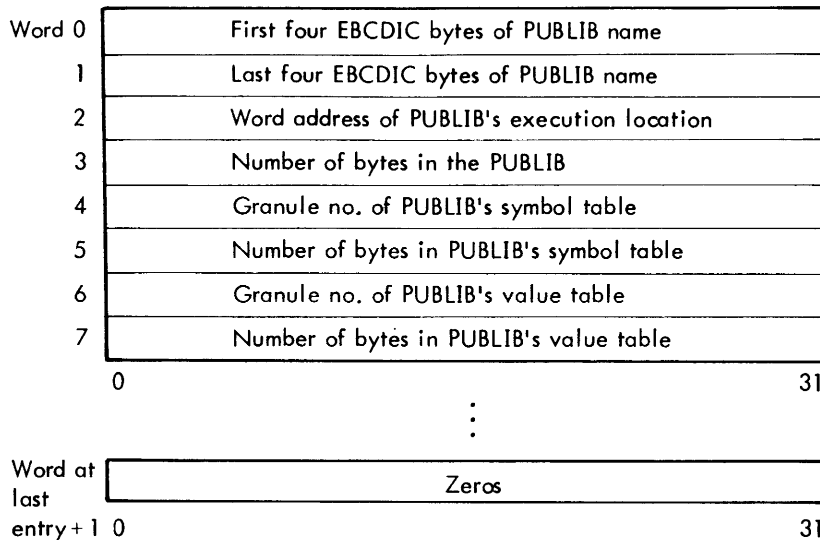
DIAG only uses the Dynamic Table area to reference T:SEG and T:MODULE.

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.

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



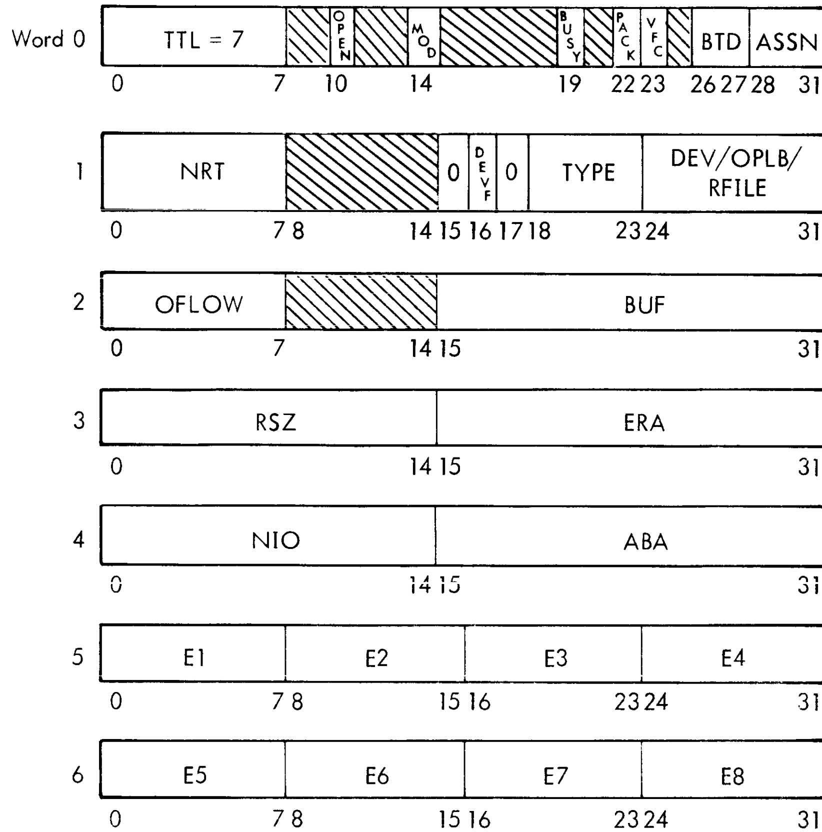
T:DCBF

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 6.

Table 6. 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 seven-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 seven-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 RAD 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 7.

Table 7. 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.

Table 7. Background Scratch Files (cont.)

File Name	Loader Use
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 52.

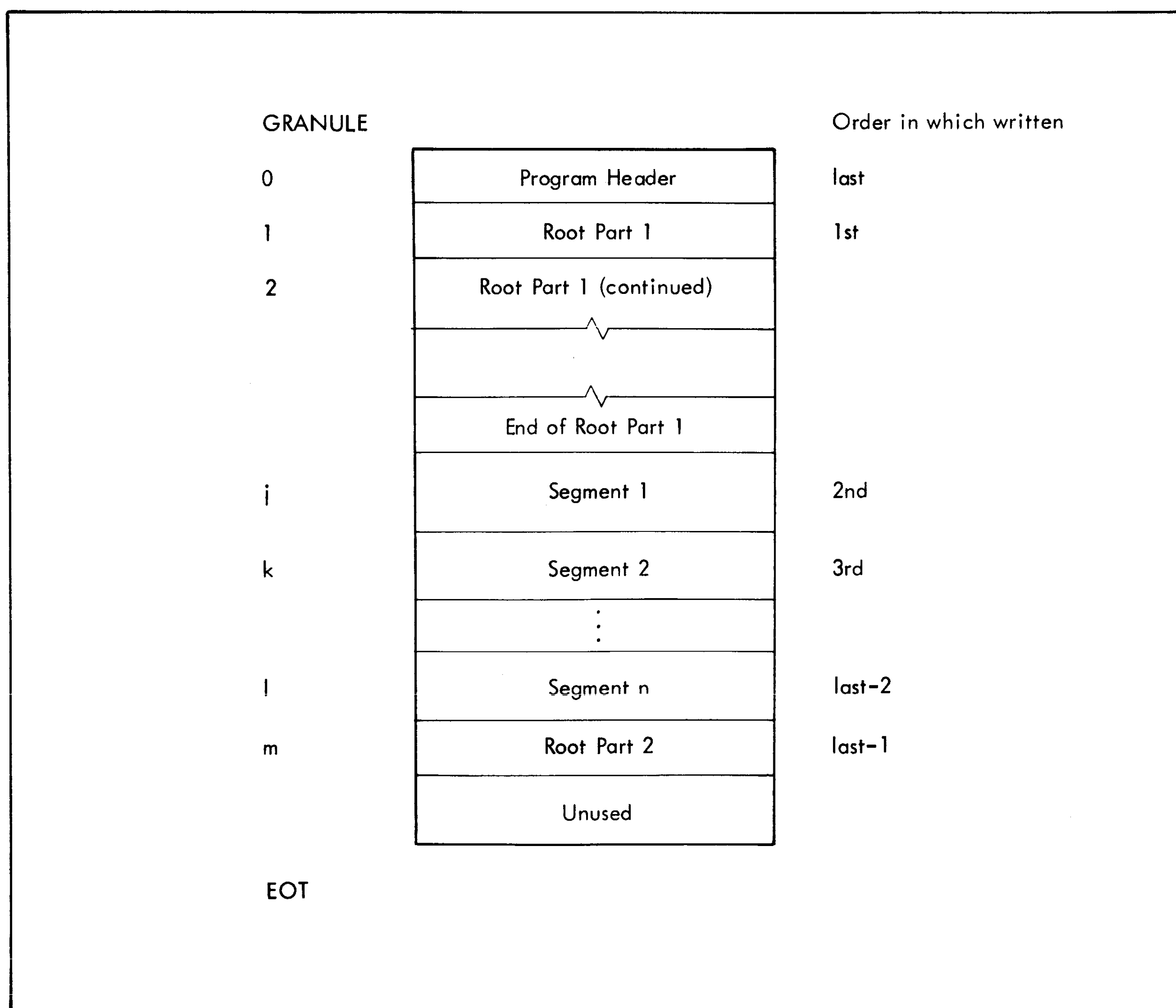


Figure 52. Program File Format

The foreground/background program-header format is described in the "RBM Tables Format" chapter. The Public Library (PUBLIB) header format is also described in that chapter.

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.

: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 RAD scratch files; and sets the 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-RAD 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 RAD scratch file, X6.) LIB then creates T:LDEF, starting from the end of the "pack".

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 RAD (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 RAD 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 RAD file overflowed or there were loading errors when a Public Library was being created.

Logical Flow of DIAG

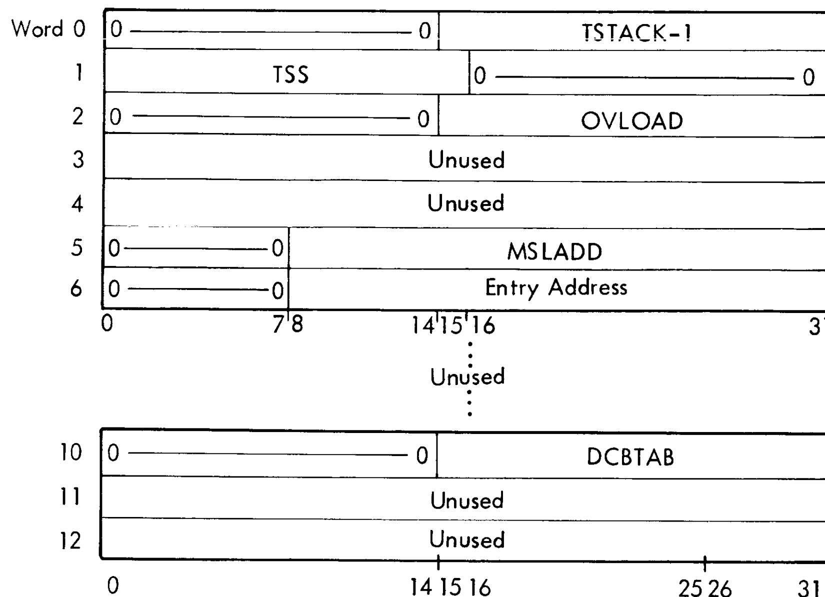
When the DIAG overlay is called, the environment of the calling program is unchanged. Since the DIAG segment overlays the calling segment, all the temporary and permanent storage cells used by the calling segment are located in either the Root or the Dynamic Table area. DIAG is called by the RDIAG subroutine 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.

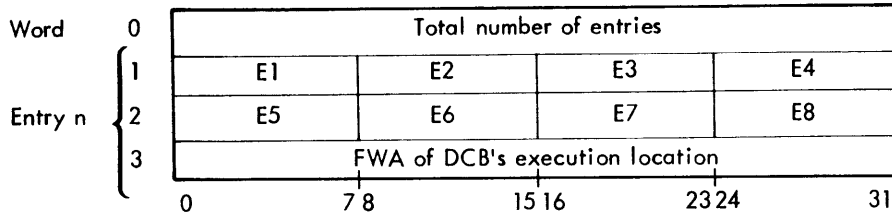
OVLOAD is the address of the table used by the SEGLOAD function to read in overlay segments or zero.

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 "RBM 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 the RAD Editor. It is loaded by the JCP Loader with the !LOAD 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 53 through 60.

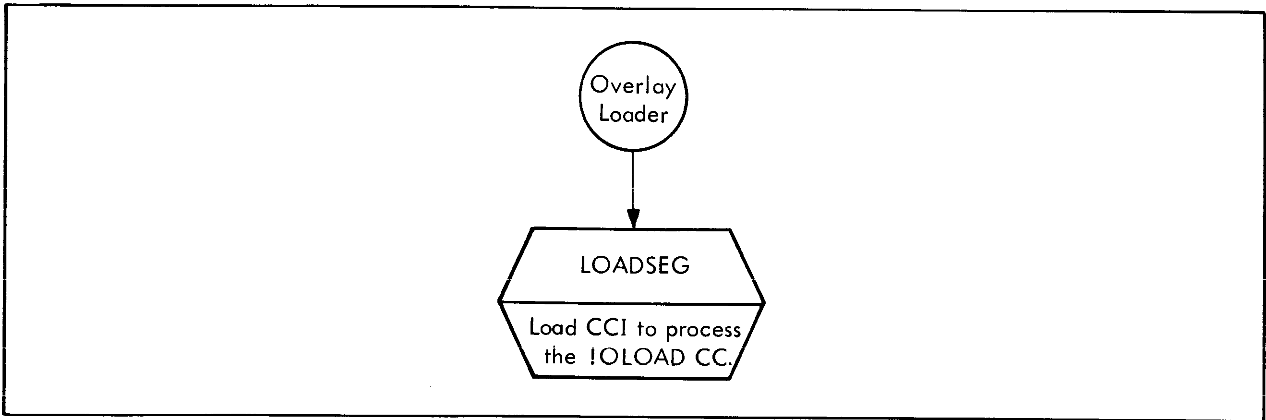


Figure 53. Overlay Loader Flow, !OLOAD

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 54. Overlay Loader Flow, CCI

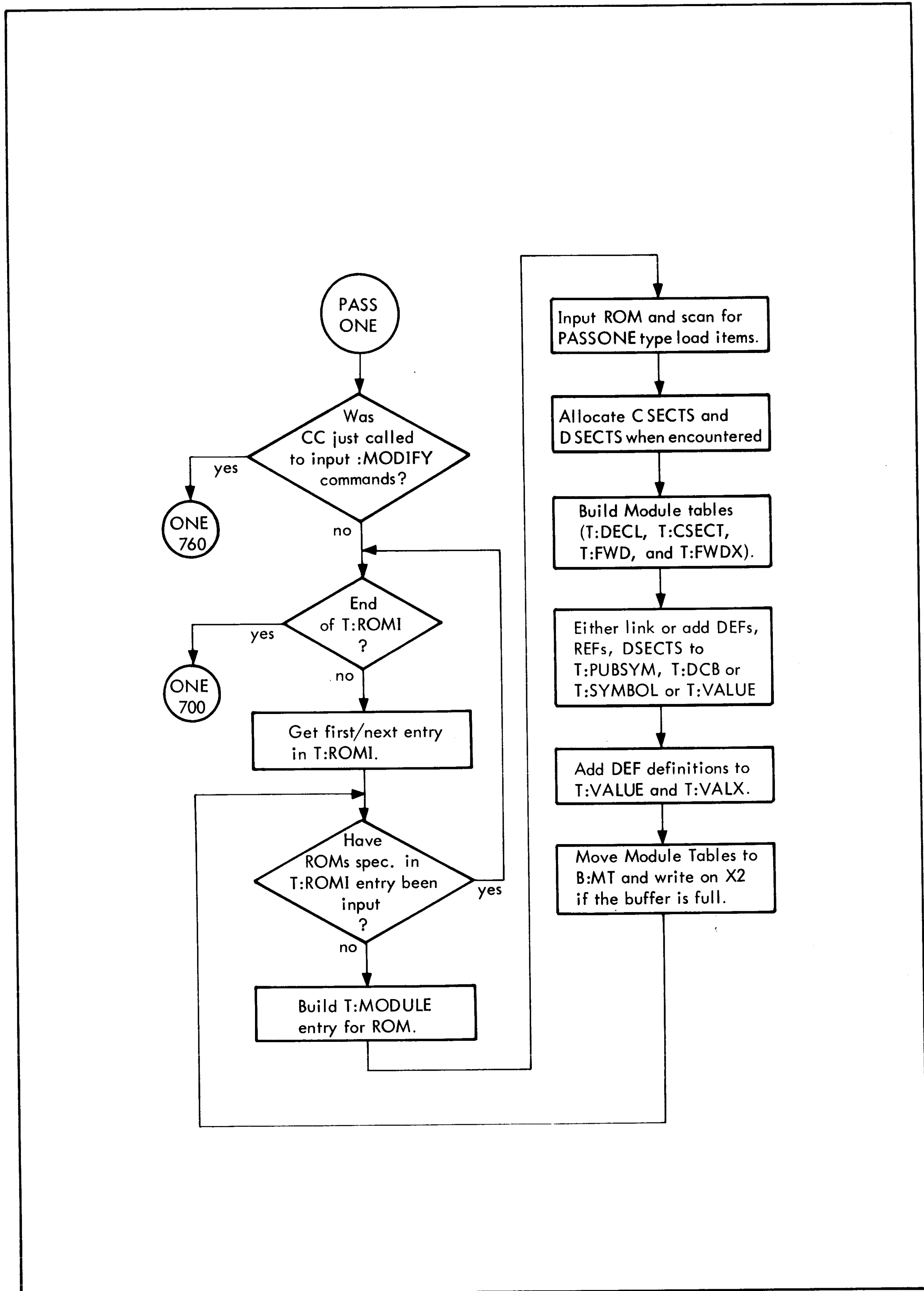


Figure 55. Overlay Loader Flow, PASSONE

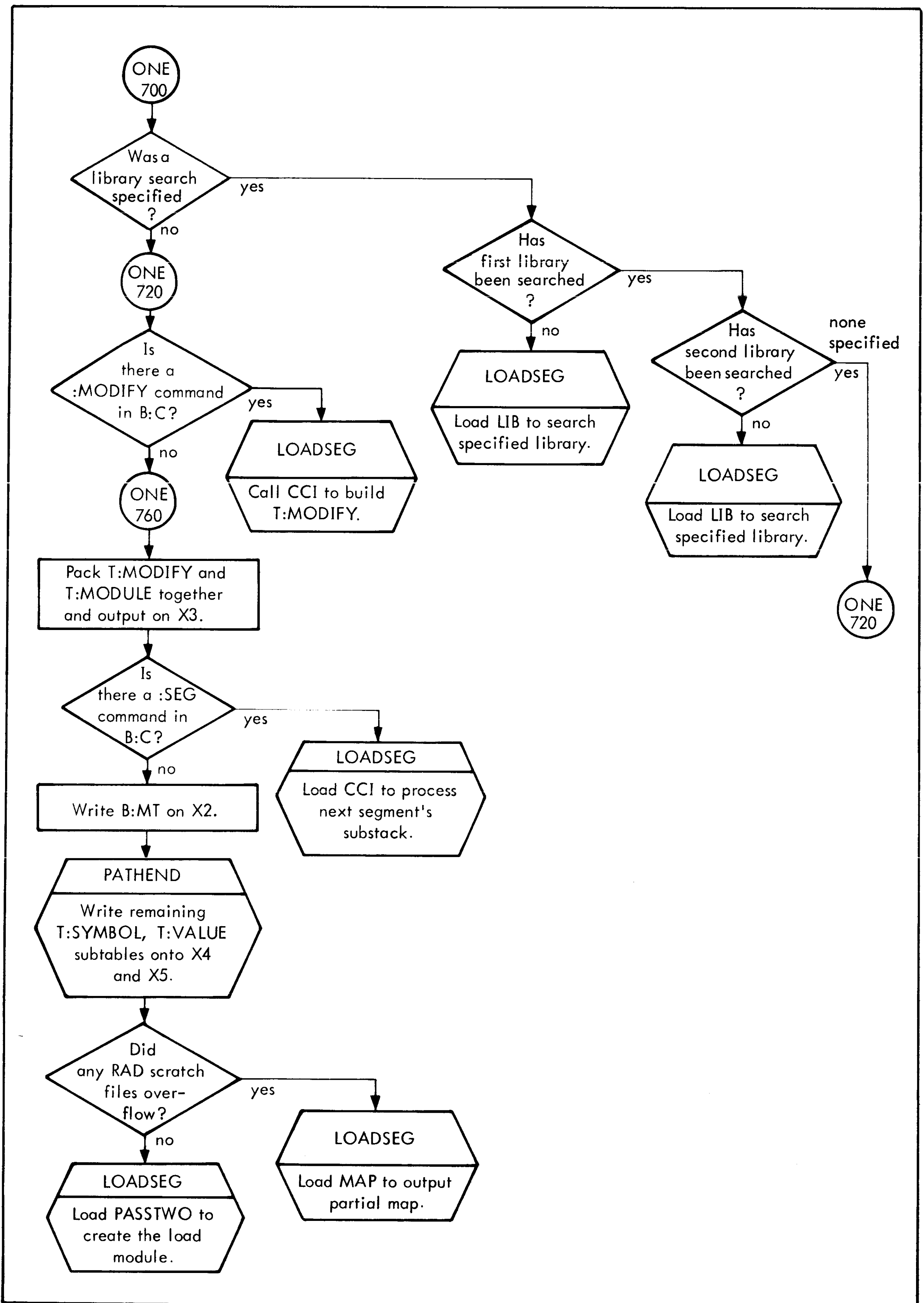


Figure 55. Overlay Loader Flow, PASSONE (cont.)

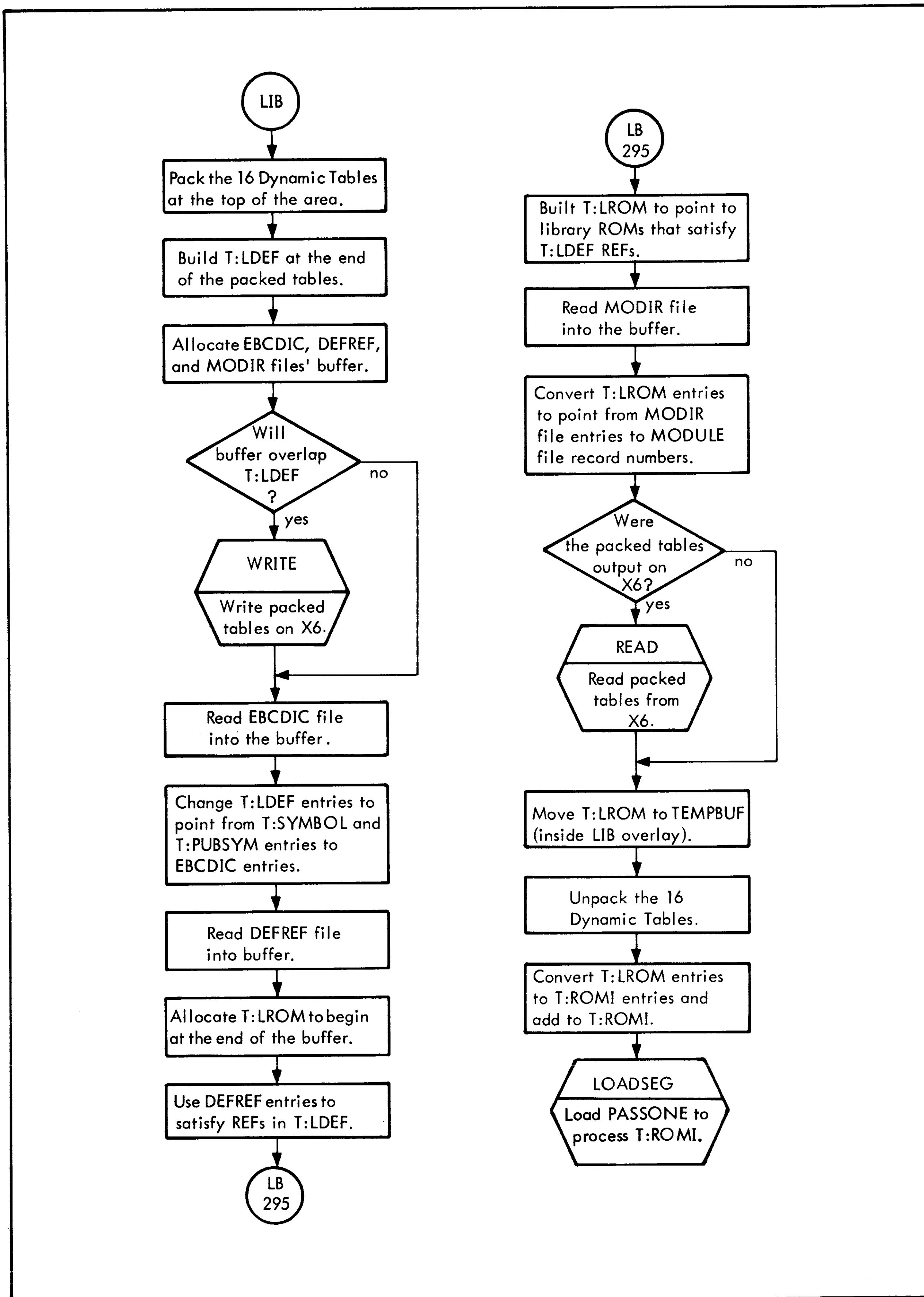


Figure 55. Overlay Loader Flow, PASSONE (cont.)

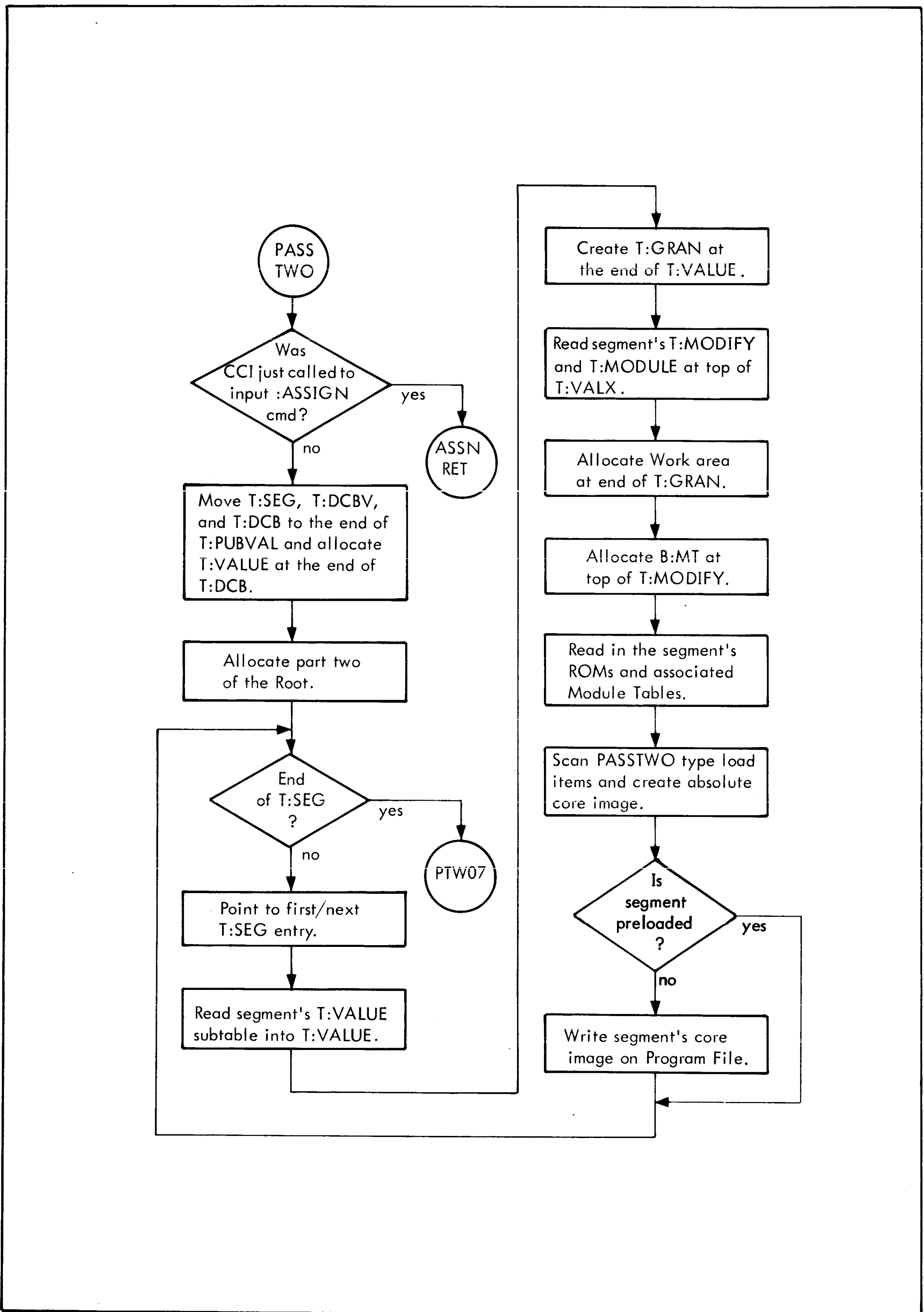


Figure 56. Overlay Loader Flow, PASSTWO

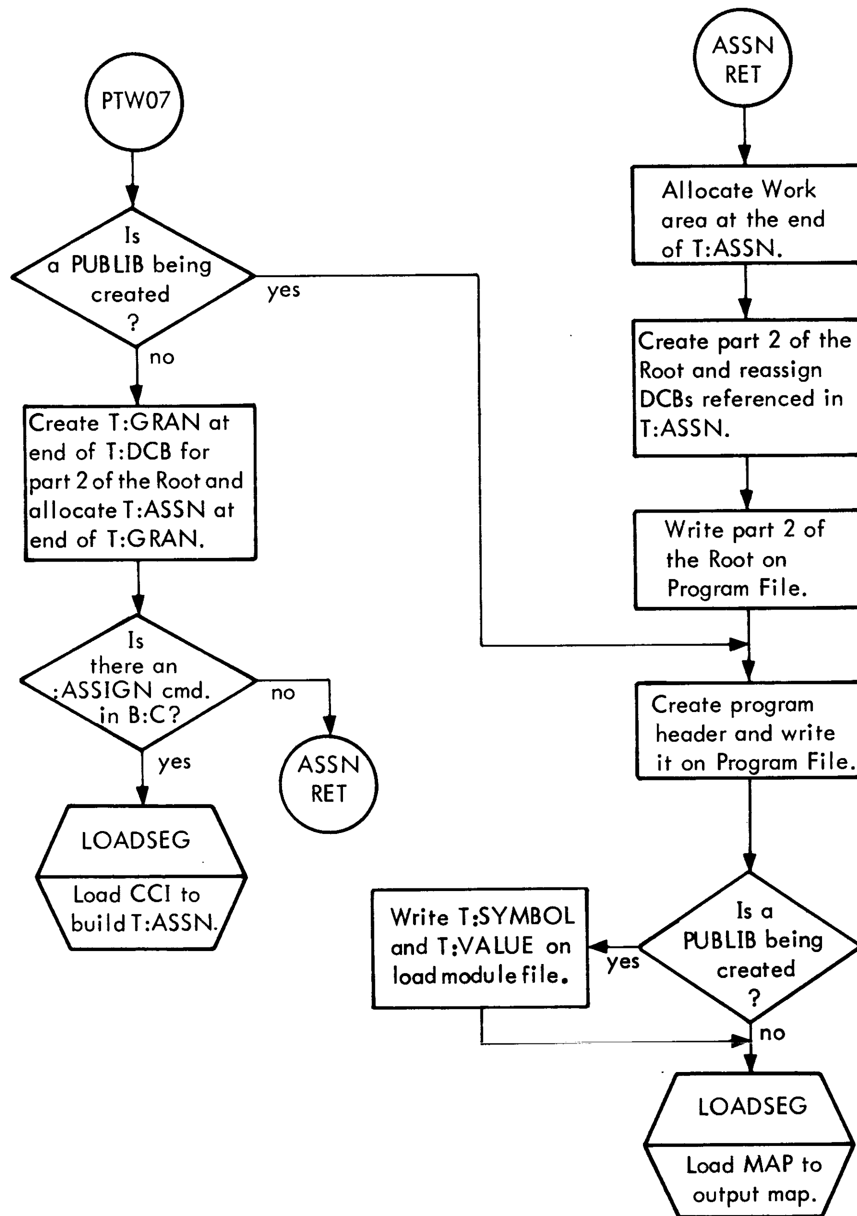


Figure 56. Overlay Loader Flow, PASSTWO (cont.)

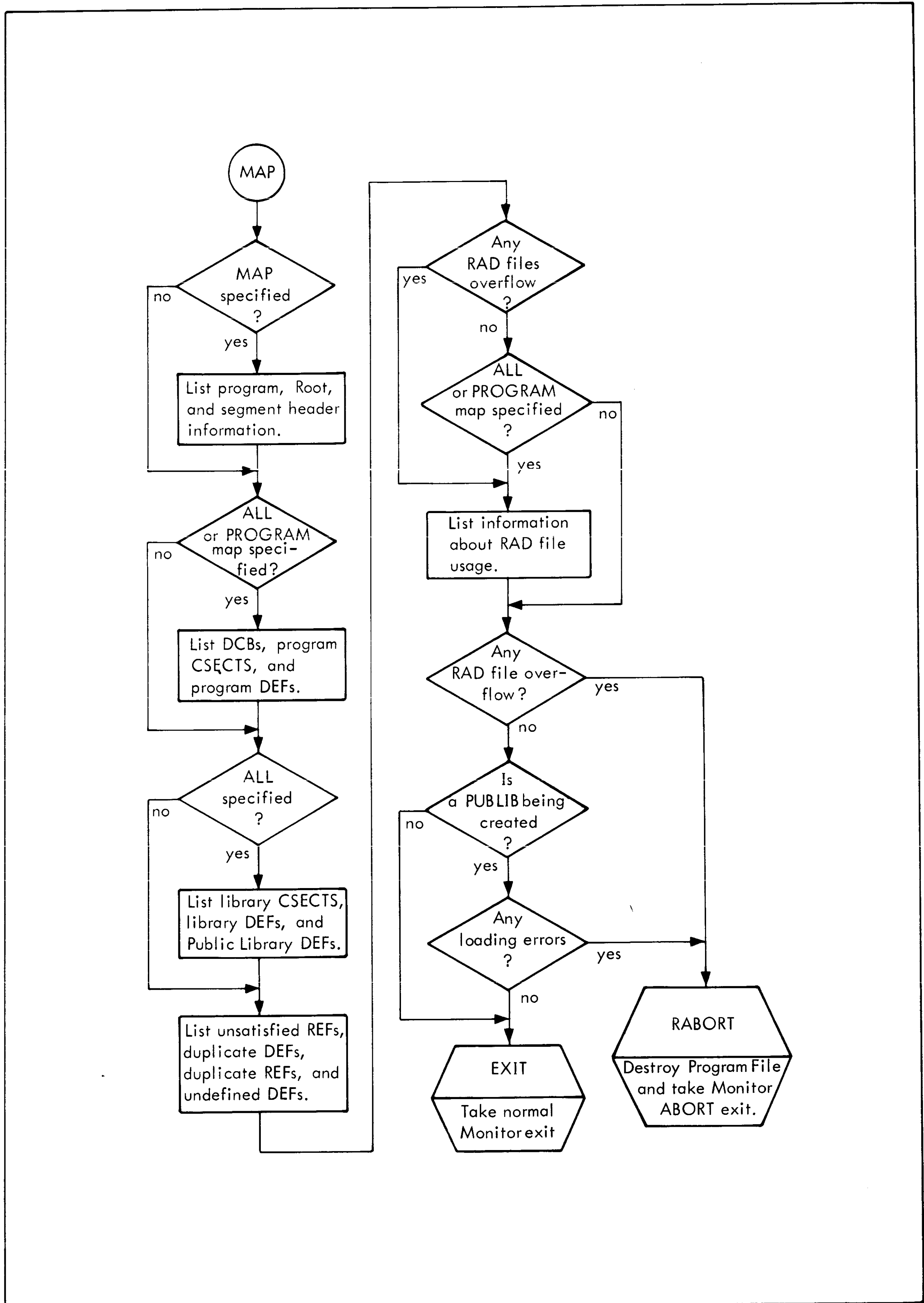


Figure 57. Overlay Loader Flow, MAP

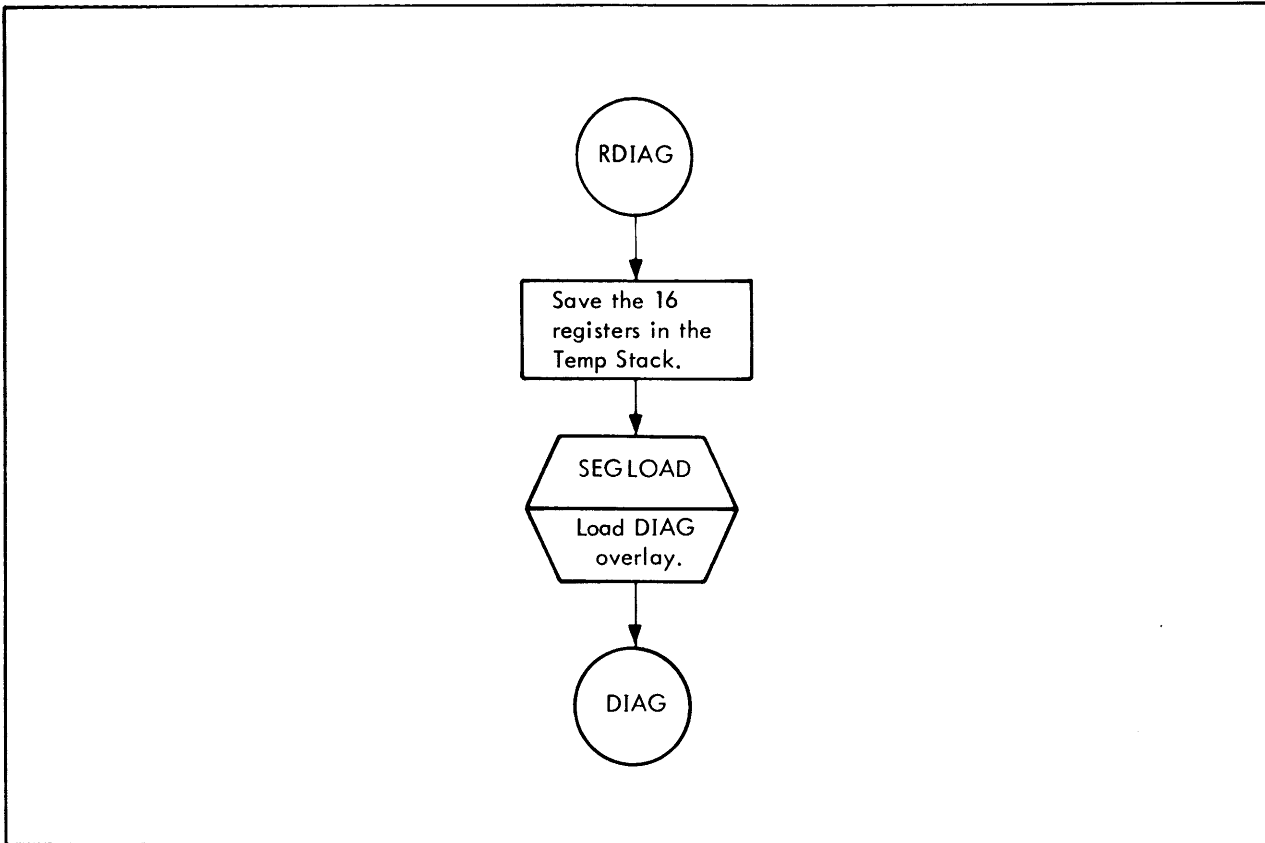


Figure 58. Overlay Loader Flow, RDIAG

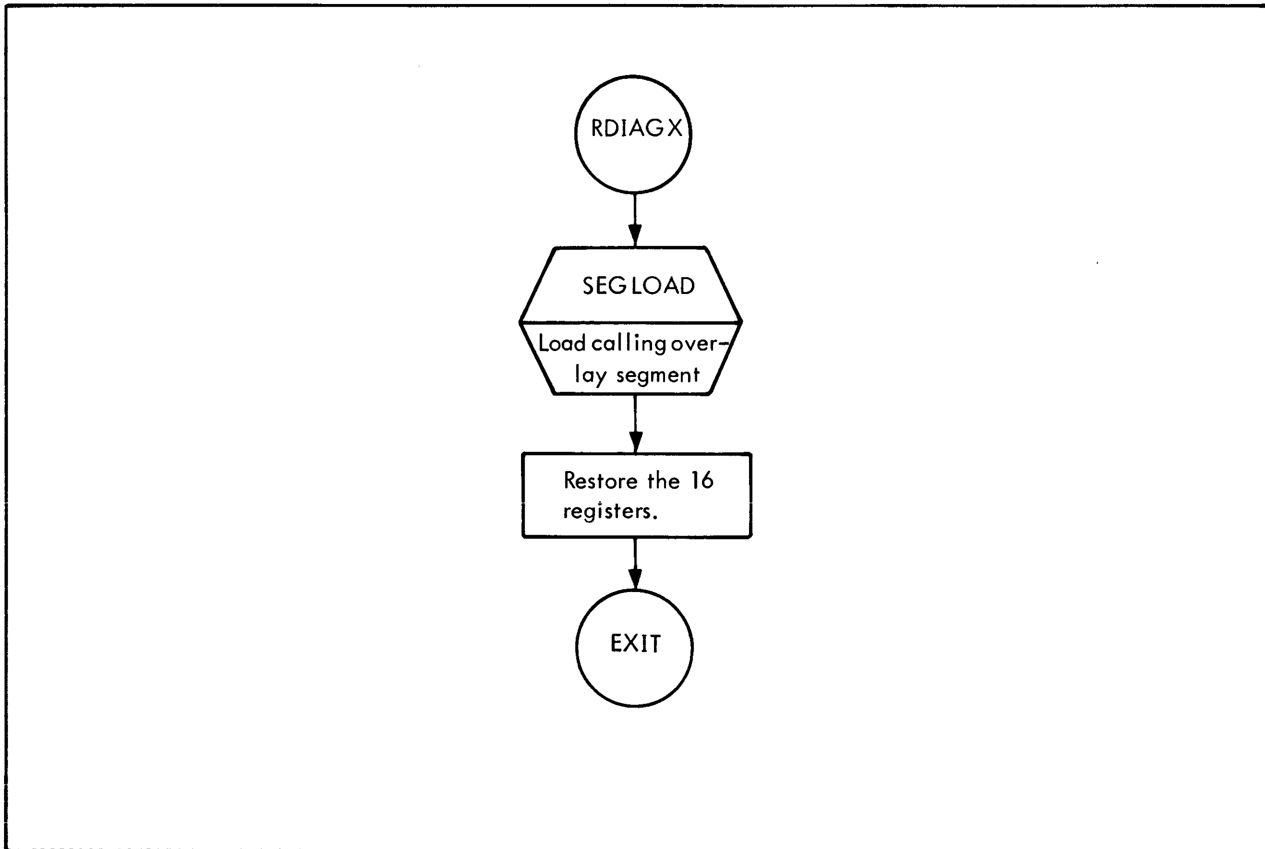


Figure 59. Overlay Loader Flow, RDIAGX

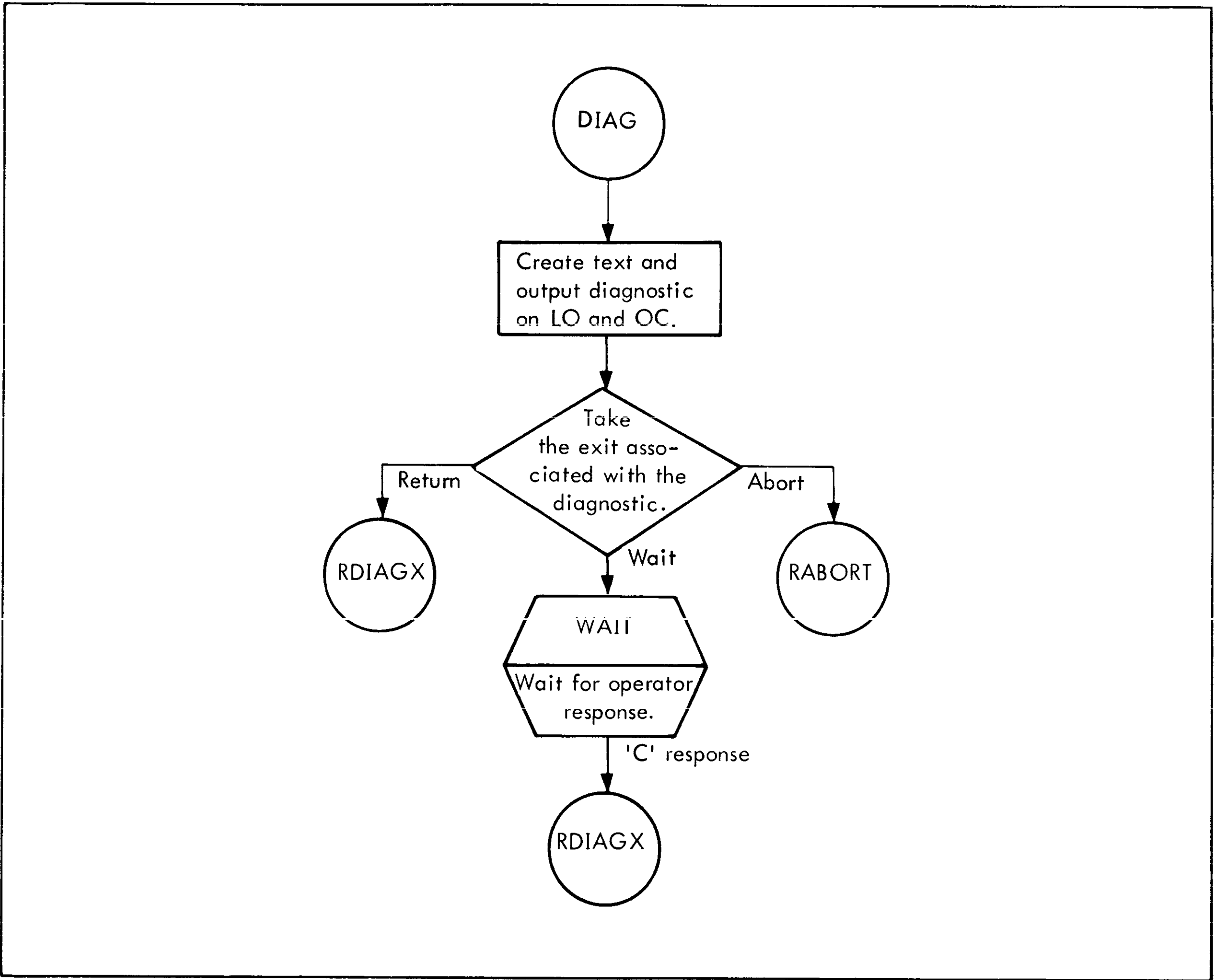


Figure 60. Overlay Loader Flow, DIAG

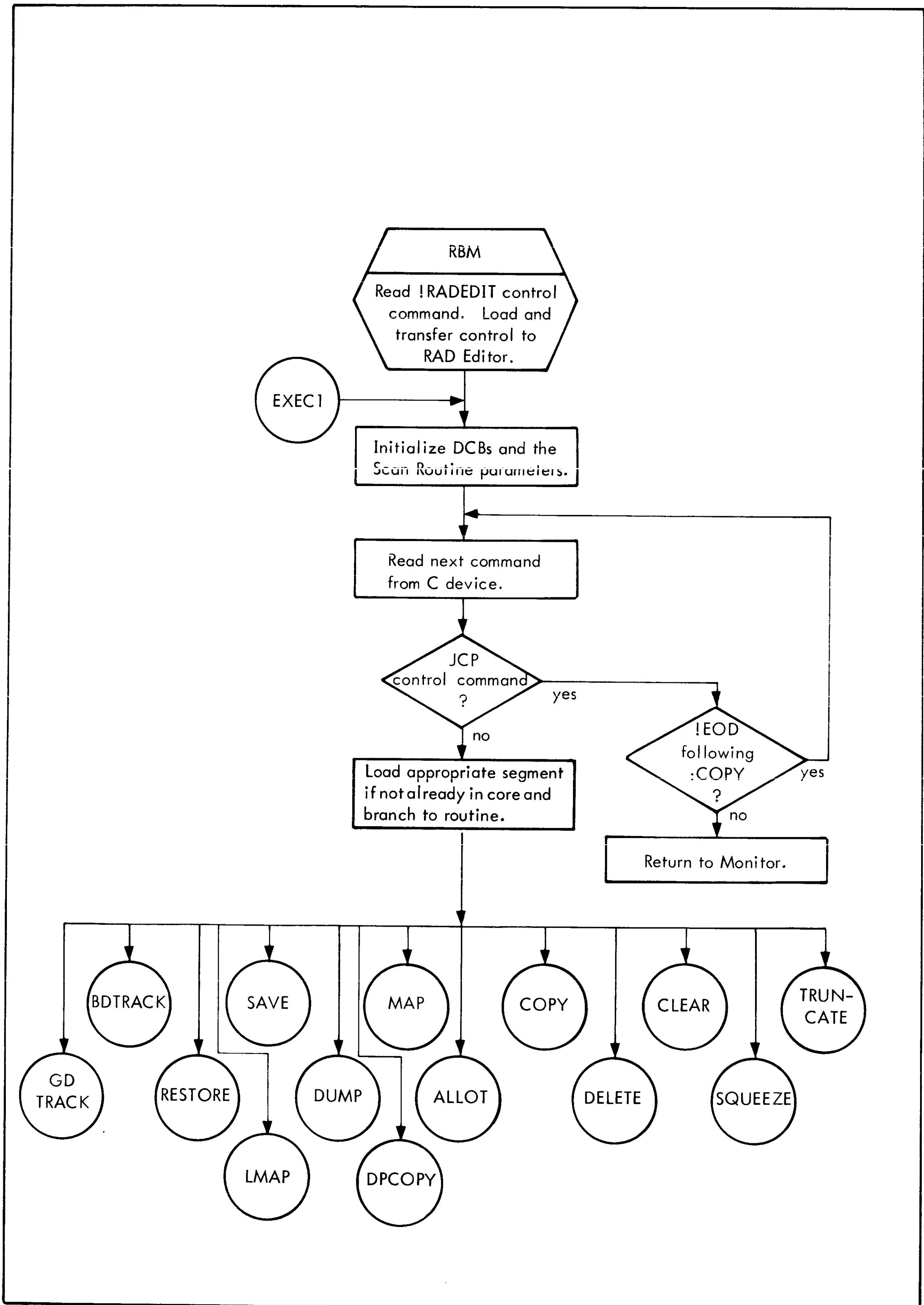


Figure 61. RADEDIT Functional Flow

C If C = 1, compressed records.

B If B = 1, blocked records.

RF If RF = 0, background or nonresident foreground program; if RF = 1, resident foreground program.

GSIZE is the granule size, in bytes, to be used for direct accessing.

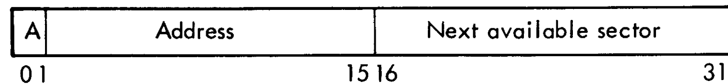
FSIZE is the current number of records in file.

RSIZE is the number of bytes per logical record.

BOT is the relative disk address of first sector defined for the file.

EOT is the relative disk address of last sector defined for the file.

No entry extends over a sector boundary. After a sector of directory is filled, the next available sector within the permanent disk area is allocated as a continuation of the directory. Sectors of a directory are linked by means of a one-word identification entry which is the first word of every sector of the directory. It has the form



where

A If A = 0, the directory ends in this sector; if A = 1, the directory is continued on another sector.

Address If A = 0, "Address" contains the relative location within the sector available for the next entry; if A = 1, "Address" is the relative disk address of the sector where the directory is continued.

Next available sector is the relative disk address of the first unused sector in the area. This word is meaningful only for the last sector of directory.

Space within the permanent disk area is allocated sequentially. The first file in an area, which corresponds to the first entry in the sector of directory, begins in the second sector and extends over an integral number of sectors. Every file begins and ends on a sector boundary.

Control Commands

The permanent disk areas are maintained through the execution of :ALLOT, :DELETE, :TRUNCATE, and :SQUEEZE commands.

The permanent file directories are maintained so that the directory entry defining a file is always contained in a sector of directory that has a lower sector address than the file it defines. To facilitate maintenance, files always appear in the same order as the entries in the file directory.

:ALLOT The permanent disk area specified on the command determines the area in which a file is to be allocated. The FILE, FORMAT, FSIZE, RSIZE, GSIZE, and RF parameters are used to form a new directory entry.

The new entry is added to the current sector of directory (identification entry with A = 0) at the location specified by "Address" in the identification entry. The BOT of the new entry is set equal to the "Next available sector". The EOT is computed, using the FSIZE, RSIZE, and FORMAT parameters. The identification entry is updated to reflect the new entry. The "Next available sector" is set = EOT of the new entry +1, and the "Address" is incremented by 5.

If there is insufficient space in the current sector of directory for another entry, "A" in the identification entry is set to 1; "Address" is set = "Next available sector" and that sector address is used for the new sector of directory. A new identification entry is built by setting "A" = 0; "Address" = 6; and "Next available sector" = EOT of the new entry +1.

If there is insufficient space to allocate for a file, the file directory is searched for deleted entries (file name = 0). The deleted entry that allocates sufficient space and the least amount of space is selected for the new entry. Disk space is lost if the deleted entry allocates more area than is required by the entry. This space can be made available for allocating by executing a :SQUEEZE command. The area allocated by a new entry is zeroed out.

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 = disk sector size in words.

:DELETE The DELETE system call is used to delete a disk file. The call is built from the area and file name parameters on the :DELETE command. The space formerly allocated by the entry becomes unused until either a :SQUEEZE command is executed, or an :ALLOT command is executed with insufficient space on the end of an area to allocate. Space is then allocated by using a deleted entry.

:TRUNCATE The permanent disk area specified on the command determines the area in which a file(s) is to be truncated, with the file name specified being used 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 (K: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 unused and is not reallocated until a :SQUEEZE command is executed.

:SQUEEZE The parameters on the :SQUEEZE command determine which permanent disk area to squeeze. Truncating or deleting a file that is subsequently reallocated may cause a loss of space that cannot then be allocated. That is, the current permanent file directory entry allocates less space than allocated by the original entry. Executing a :SQUEEZE regains all unused space. The directories are compacted and the files themselves are moved to regain the unused space. The BOT and EOT entries (of the permanent file directory) are updated as they are compacted to indicate the area occupied by the moved file. Figure 62 illustrates the permanent disk area before and after squeezing.

Library File Maintenance

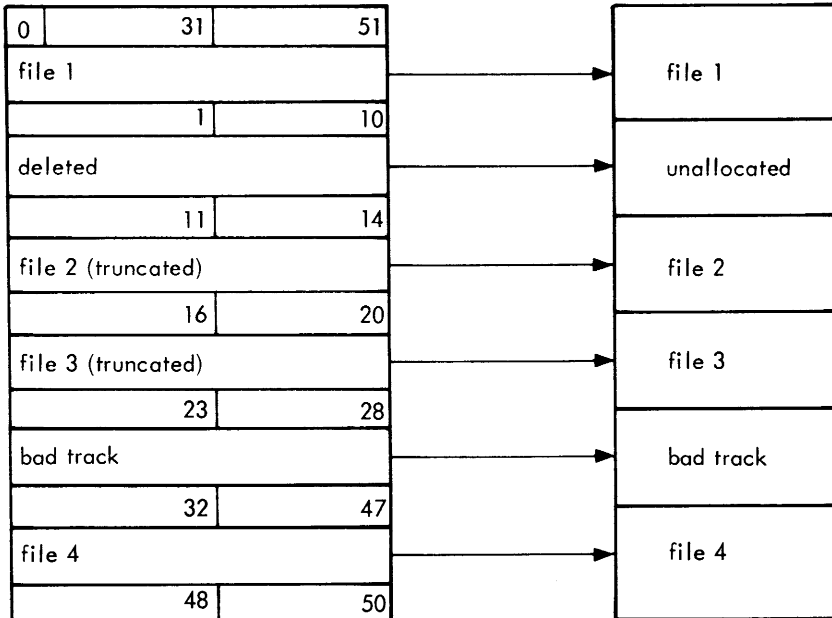
Both the System Library files residing in the SP area and the User Library files residing in the FP area have the same file structure. Each library consists of one blocked Module File (MODULE) and three unblocked files: the Module Directory File (MODIR), EBCDIC File (EBCDIC), and DEFREF File (DEFREF).

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

Permanent RAD Area Before Squeezing



Permanent RAD Area After Squeezing

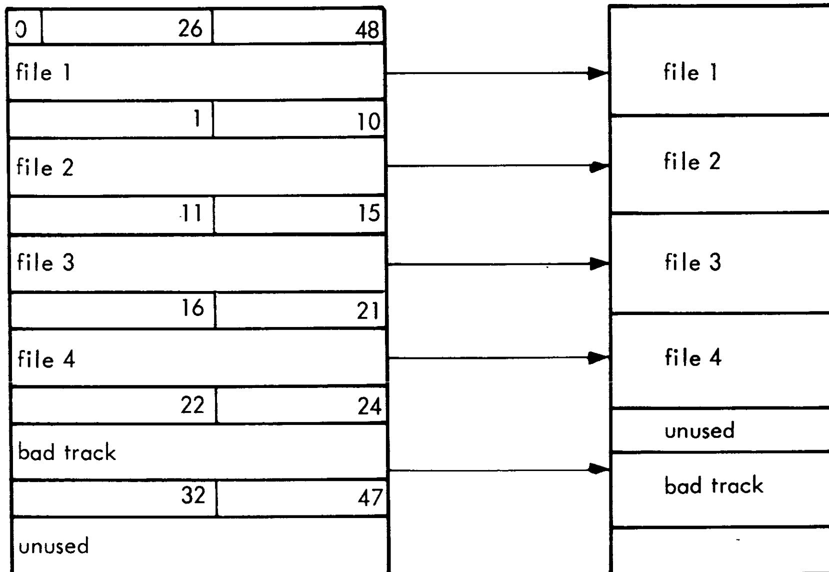


Figure 62. Permanent Disk Area

command. The approximate number of sectors (n_{MODIR}) required in the MODIR File is

$$n_{MODIR} = \frac{3(i)}{s}$$

where

3 is the length of a MODIR File entry, in words.

i is the number of modules to be placed in the library.

s is the disk sector size, in words.

The approximate number of sectors (n_{EBCDIC}) $= \frac{2(d)}{s}$

where

2 is the average length of an EBCDIC File entry, in words.

d is the unique number of DEFs in the library.

s is the disk sector size in words.

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 RAD 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	0–15	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	
6–8	⋮	
9–12	⋮	

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	2–15	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

0 1 . 15 16 17 18 . 31

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 required to be 30 words and FORMAT must be blocked. If MODIR, DEFREF or EBCDIC is specified, FORMAT must be 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 RFTS (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 permanent disk area on the :SQUEEZE command is used to determine the library to be squeezed. Permanent disk areas containing libraries are squeezed in the same way as other areas with the following exception: after the permanent file directories are compacted and files are moved to regain the unused space, a search is made of the MODIR File. All existing modules are copied from the MODULE File to the Temporary File X1. Using X1 as the source of input, the library files MODIR, EBCDIC, and DEFREF are regenerated.

Bad Track Handling

Bad tracks within permanent file areas on a disk are removed from use by making special entries to the appropriate file directory. All bad tracks can be handled in this manner except those 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 tracks are handled through execution of B:DTRACK and :GDTRACK commands. The :BDTRACK command removes the track from use by allocating the track. The :GDTRACK command returns the track for use by deleting the entry made by :BDTRACK.

:BDTRACK The permanent file area that encompasses the bad track is determined by the disk or disk pack (DP) and bad track specified on the command. A check is made to determine if a sector of directory falls within the bad track. If it does, the bad track is not eliminated from use. A search of the file directory is made to determine if the bad track is allocated. If it is, the entry(s) that allocates the track is eliminated and replaced by a bad track entry. If it is not allocated, a bad track entry is added to the end of the file directory. A bad track entry consists of the "file name" being set to -1, and the BOT and EOT being set to the starting and ending sector of the bad track. The appearance of files in the same order as the entries in the file directory is maintained.

:GDTRACK The permanent file area that encompasses the good track is determined by the RAD or disk pack (DP) and bad track specified on the command. A search of the file directory is made for the entry that allocates the track specified on the command. The entry is deleted (file name set = 0), making the track available for allocating.

Utility Functions

The following utility functions are performed by the 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 to another disk.
- 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-ADDRESS	WORDS/SECTOR	SECTORS/TRACK	BEGIN SECTOR	END SECTOR	WRITE 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 number of words per sector, in decimal.

ttttt is the number of 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.

F is write-permitted by foreground only unless SY key-in.

B is write-permitted by background only unless SY key-in.

M is write-permitted by the monitor only.

N is write-permitted only if SY key-in.

X is write-permitted by IOEX only.

Section 2 of the map has the format

FILENAME	ORG	(AREA RELATIVE) BEGIN SECTOR	END SECTOR	GRANULE SIZE (BYTES)	RECORD SIZE (BYTES)	FILE SIZE (RECS)	APPROX RECORDS REMAINING
nnnnnnnn	f	sssss	ttttt	ggggg	rrrrr	ℓℓℓℓℓ	aaaaa

where

nnnnnnnn is the name of a file in the permanent disk area.

f is the file organization:

U specifies unblocked.

B specifies blocked.

C specifies compressed.

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.

aaaaa is the approximate number of additional records the file can contain.

Section 3 of the map gives statistics on the utilization of the area and has the format

REMAINING SECTORS: xxxxx
SECTORS RECOVERABLE: yyyyy

where

xxxxx is the number of unused sectors in the area, i.e., the sectors 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 done.

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 otuput 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 track entry is encountered, "BADTRACK" 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 Directory 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 by the subroutine UNPKDIRE. 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. Error 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) < 0.

Whenever any of these conditions are found, the processing of the area is terminated by the message

. "AREA CONTAINS NO FILES."

The period is used to indicate that the directory is not valid.

:LMAP This command functions exactly as the :MAP function with the following exceptions:

- SP and FP are the only areas allowed with this command.
- The map consists of up to four sections. The first three sections are as shown in the :MAP description. The additional fourth section gives information about object modules in the library files.

Section 4 of the 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 3 per line.)

rrrrrrrr is the name of an external reference. (Up to 3 per line.)

:SMAP This command is similar 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:

FILENAME	RECORDS
nnnnnnnn	ℓℓℓℓℓ

The mapping of the areas is performed in the same steps as under MAP.

:CLEAR The permanent RAD 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 RAD.

: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 RAD 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 RAD 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 RAD area.

ssss is the relative sector number, and begins with 0.

dddddddd is a data word in hexadecimal.

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.

: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 or paper tape.
3. An 88-byte RBM bootstrap used for booting the disk.
4. Records containing data to be restored.

Each record to be restored is preceded by a five-word header with the format

No. words per sector	LRA	LRT	0 —— 0	DP	Area ident.
No. sectors in record	Device number				
First-word address of area					
No. sectors per track	No. sectors (zero) to write				
CKSM (2's complement form)					

0 15 16 17 18 23 24 31

where

No. words per sector is the size of the sector.

LRA is a flag to indicate the last record of an area if LRA = 1, last record.

LRT is a flag to indicate the last record of the tape if LRT = 1, last record.

DP indicates that the device is a disk pack if DP = 1.

Area ident. is the area to which the record belongs.

No. sectors in record is the size of record (in sectors).

Device number is the physical device number of the RAD or disk pack.

FWA of area is the absolute address where the data records should begin being restored.

No. sectors per track/No. sectors (zero) to write is the number of sectors containing all zeros preceding nonzero data in the record.

CKSM is the checksum of this record in the 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 RBM RAD bootstrap is written. The FWA and device number for the header is obtained from K:RDBOOT.
3. The image of the RBM RAD 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.

: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 RBM 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 overall flow of the RAD Editor is illustrated in Figures 63 through 67.

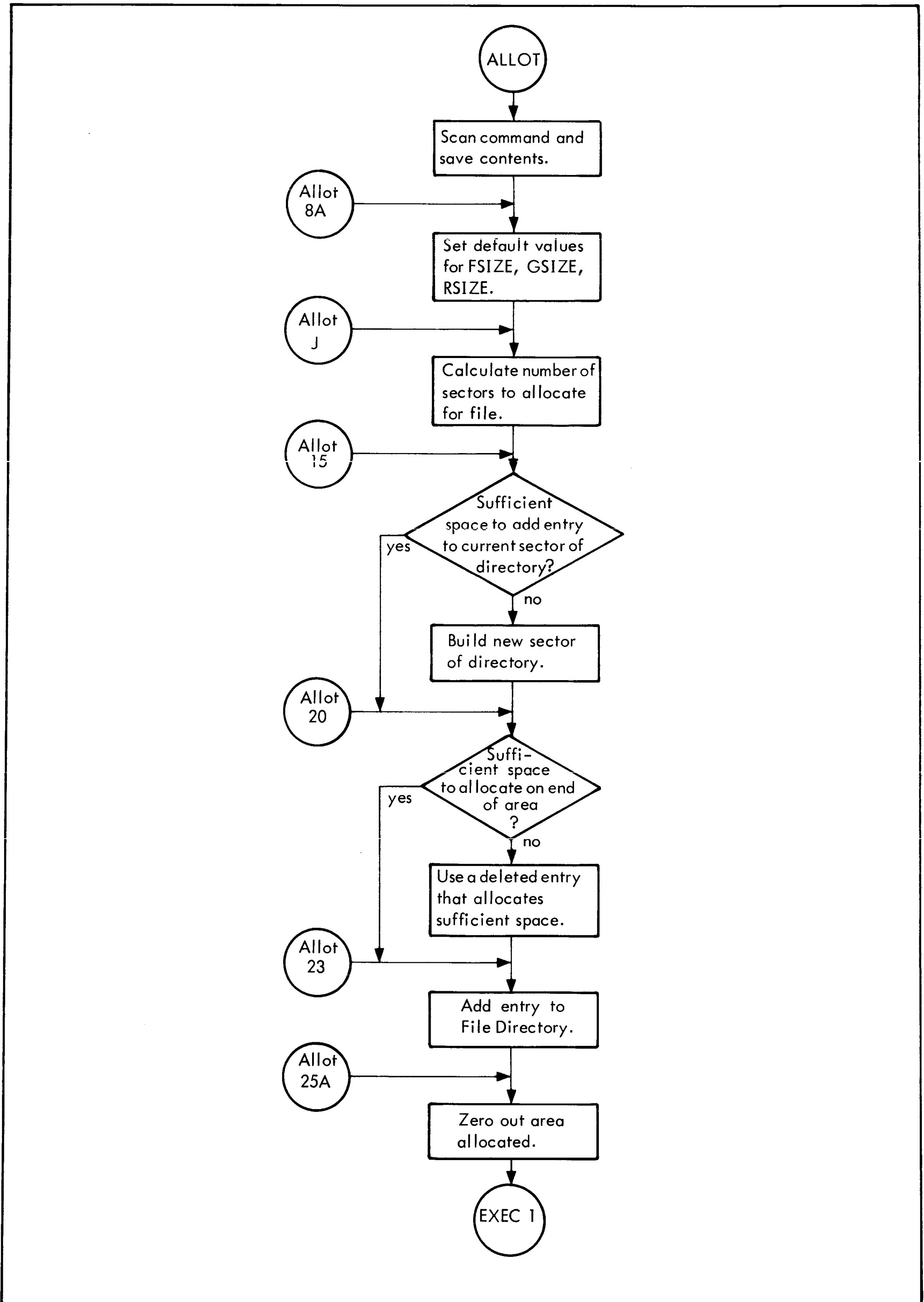


Figure 63. RADEDIT Flow, ALLOT

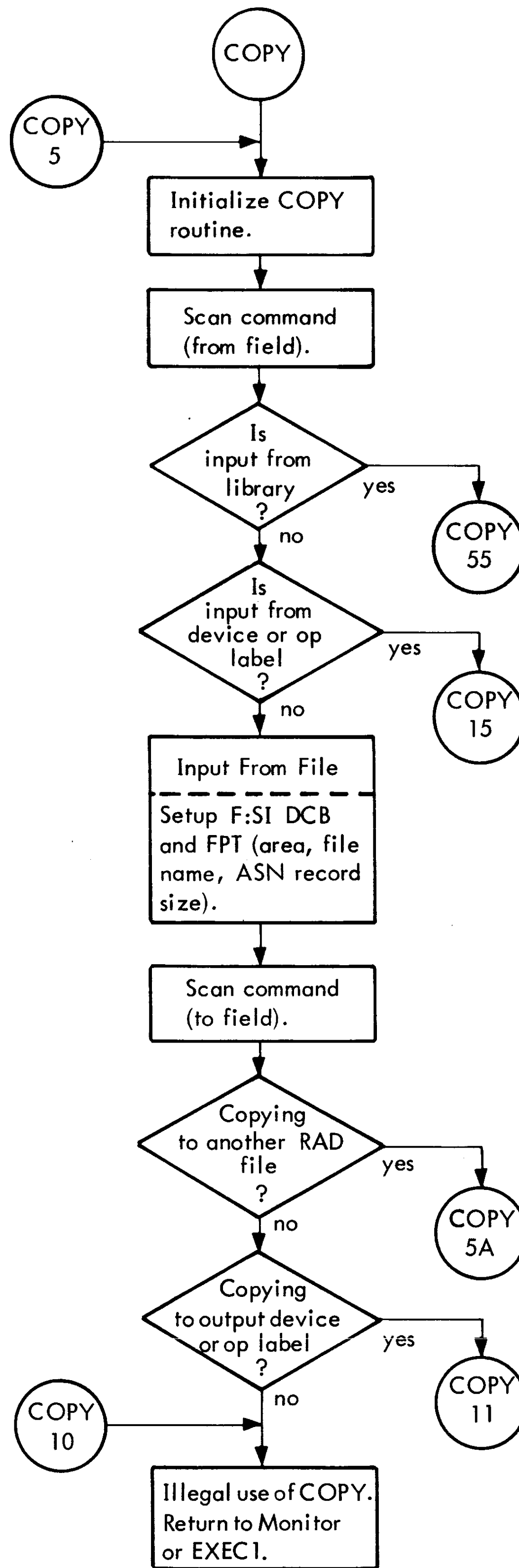


Figure 64. RADEDIT Flow, COPY

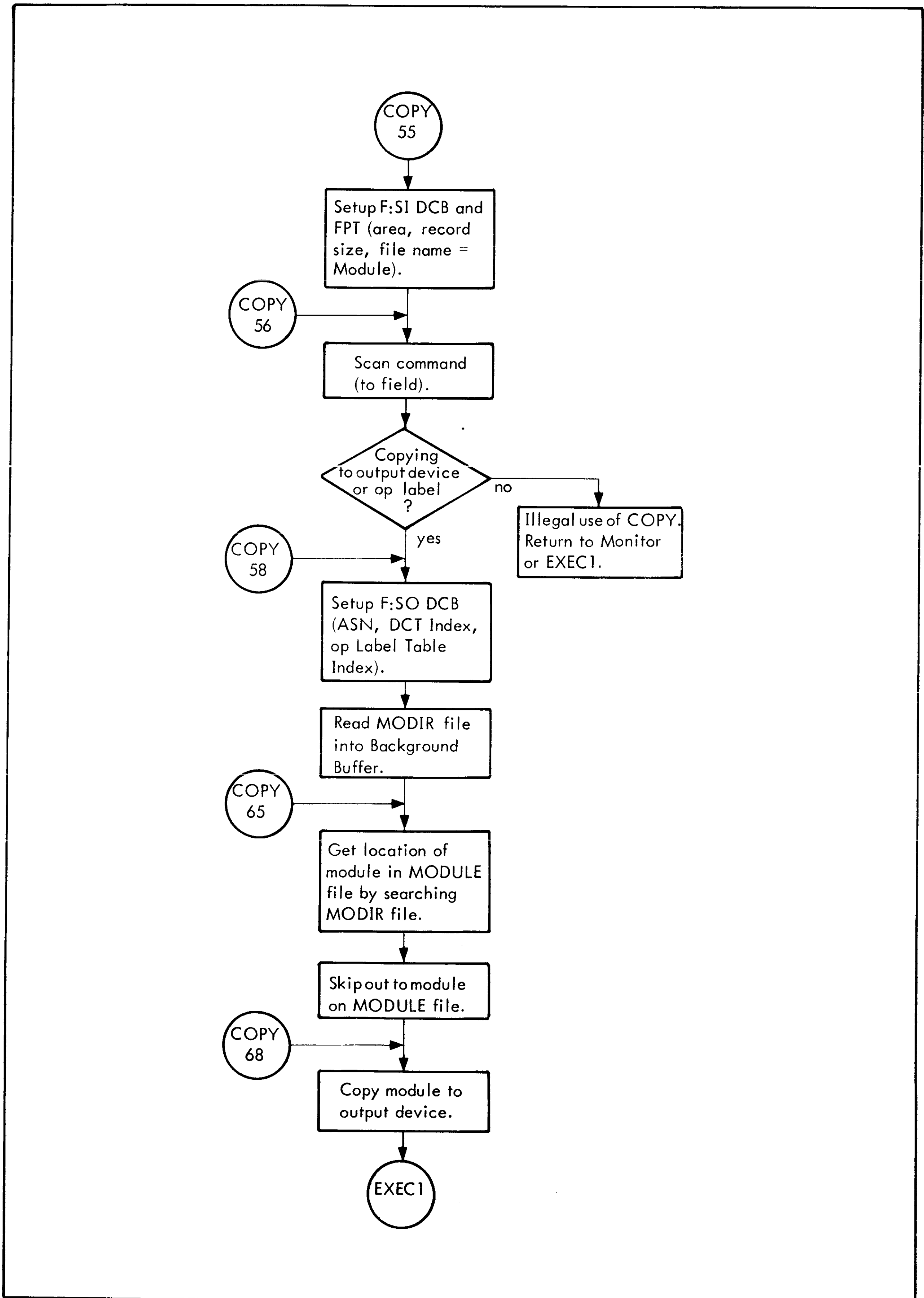


Figure 64. RADEDIT Flow, COPY (cont.)

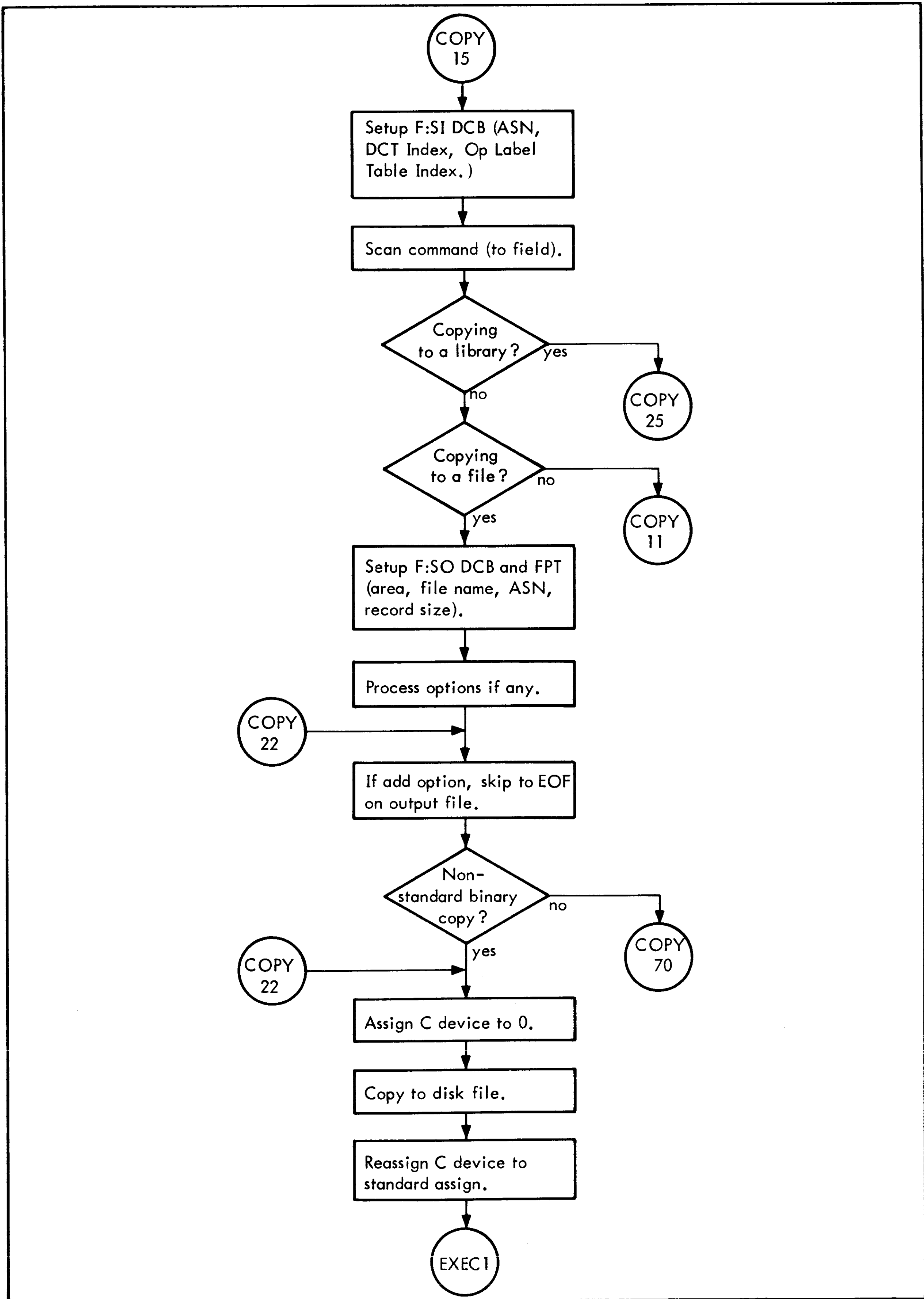


Figure 64. RADEDIT Flow, COPY (cont.)

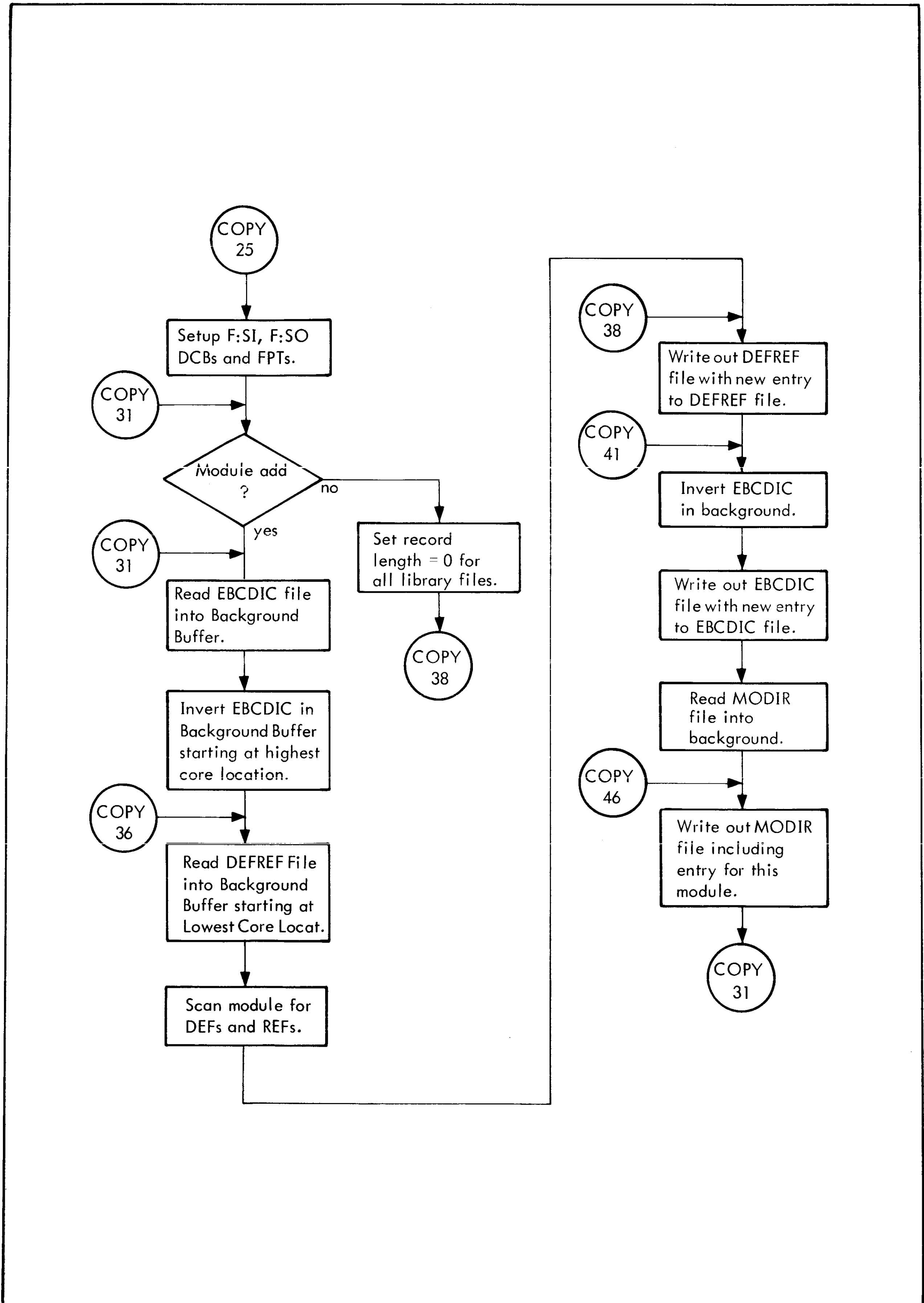


Figure 64. RADEDIT Flow, COPY (cont.)

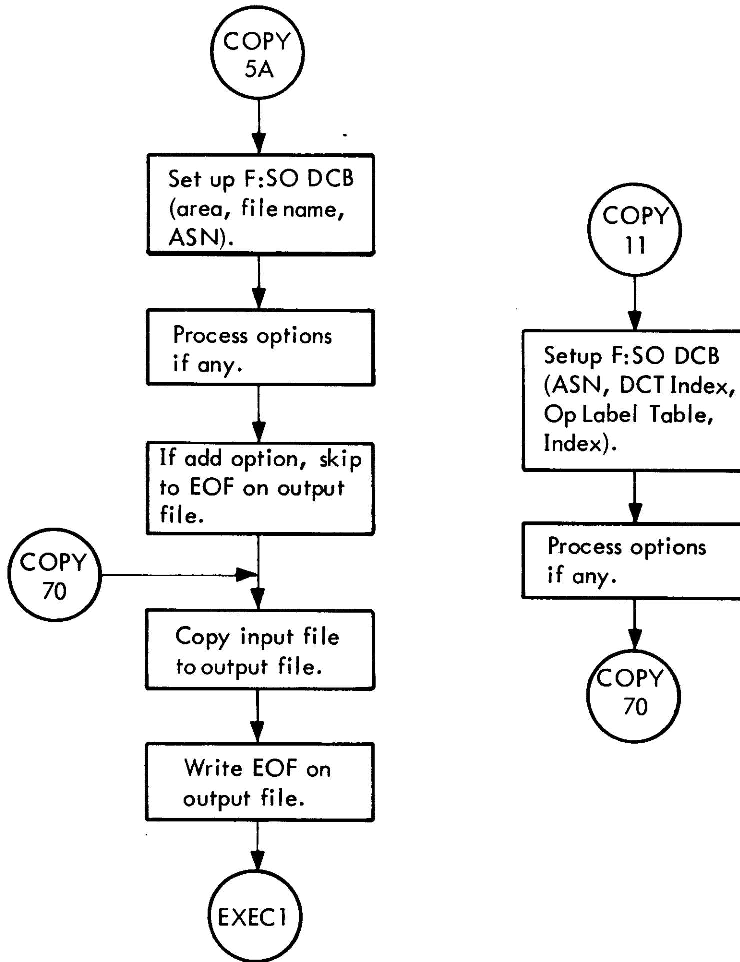


Figure 64. RADEDIT Flow, COPY (cont.)

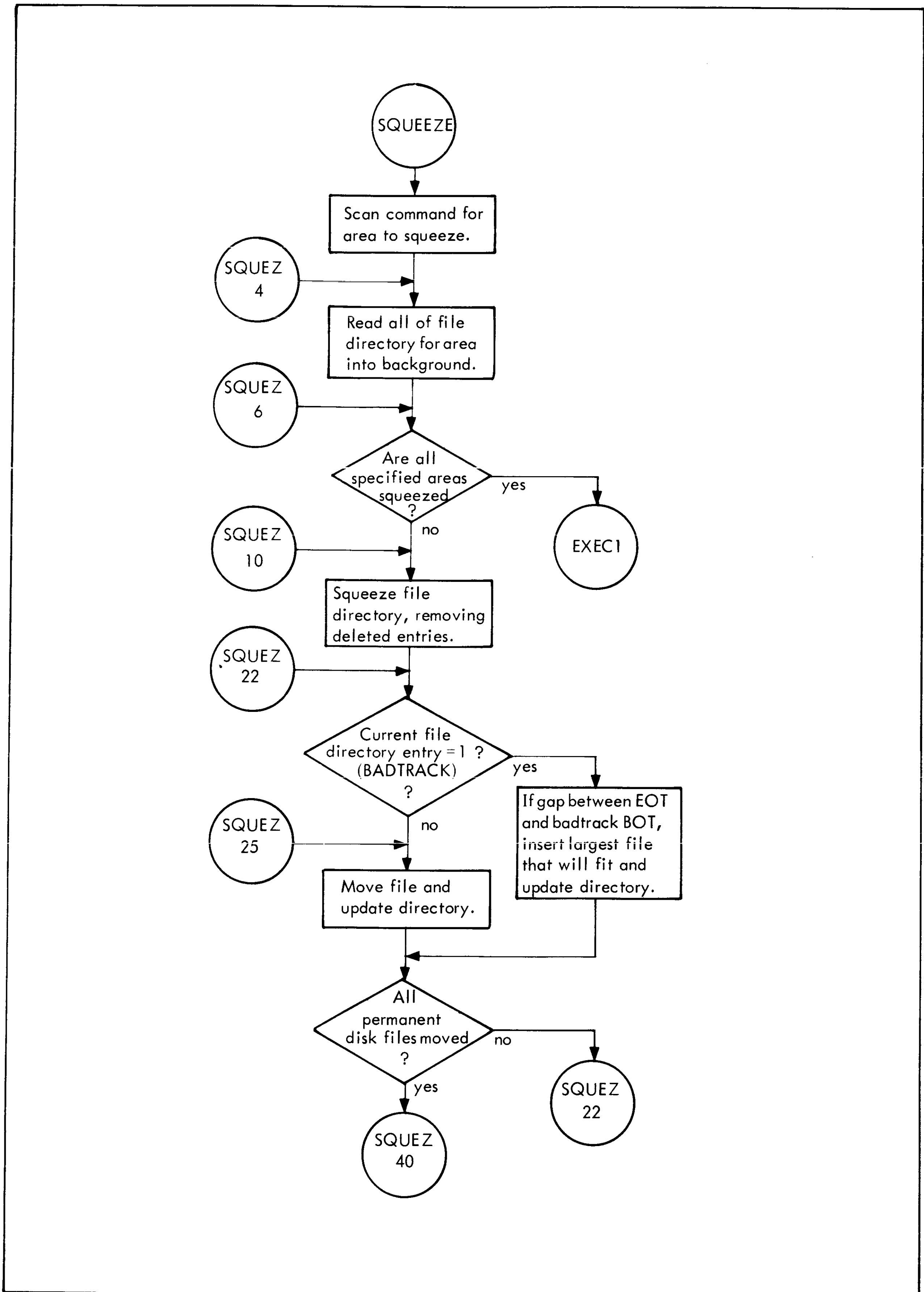


Figure 65. RADEDIT Flow, SQUEEZE

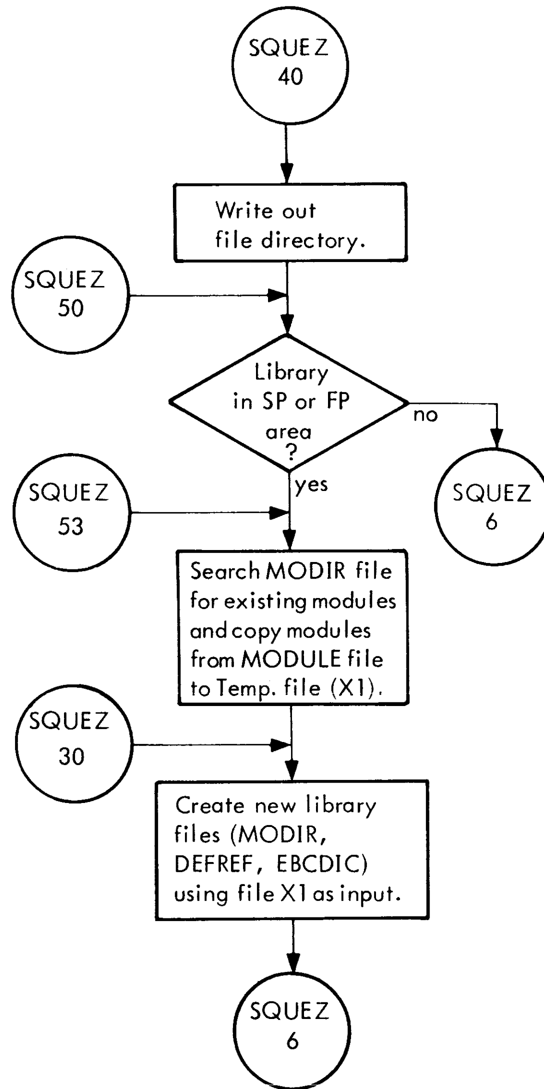


Figure 65. RADEDIT Flow, SQUEEZE (cont.)

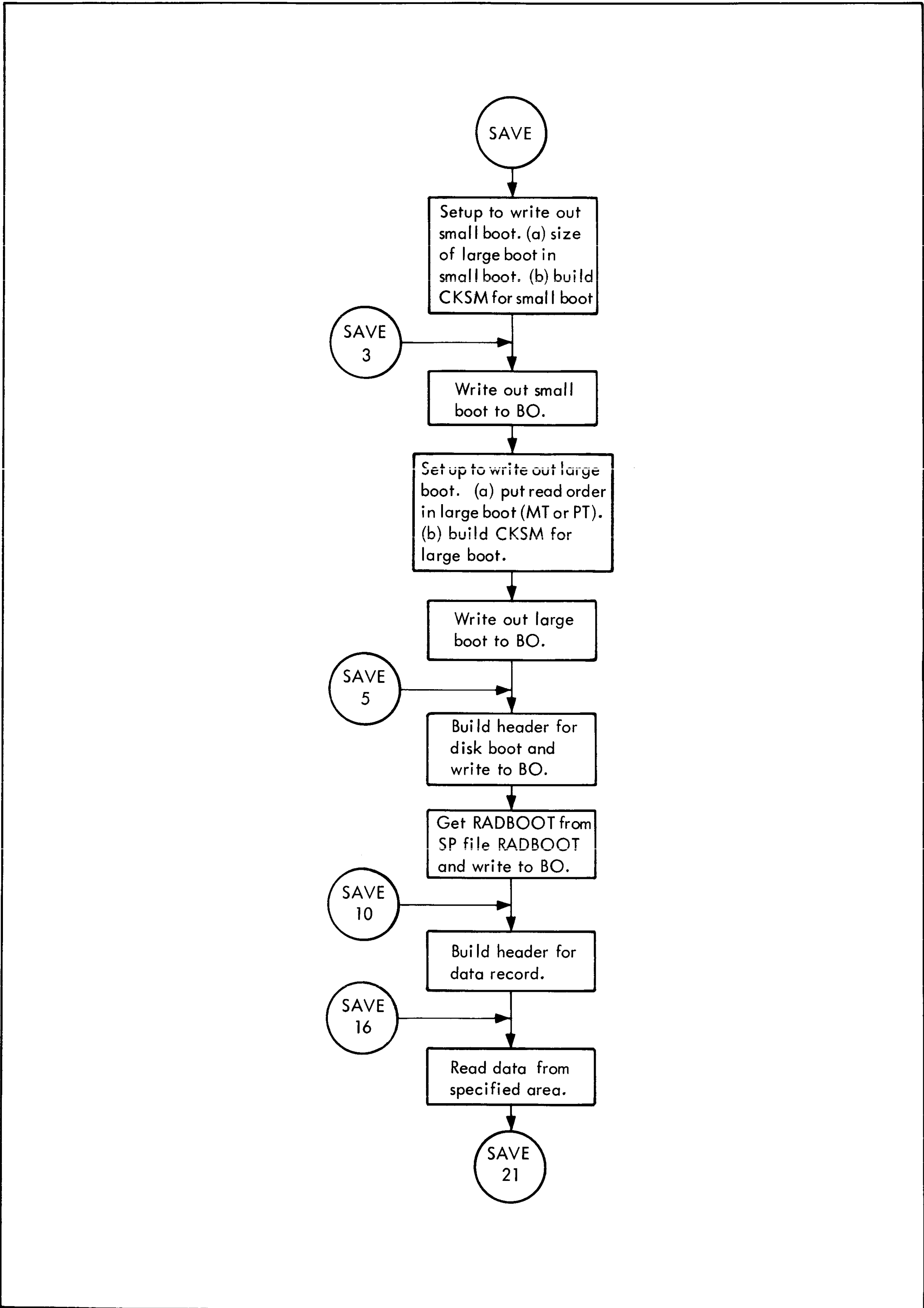


Figure 66. RADEDIT Flow, SAVE

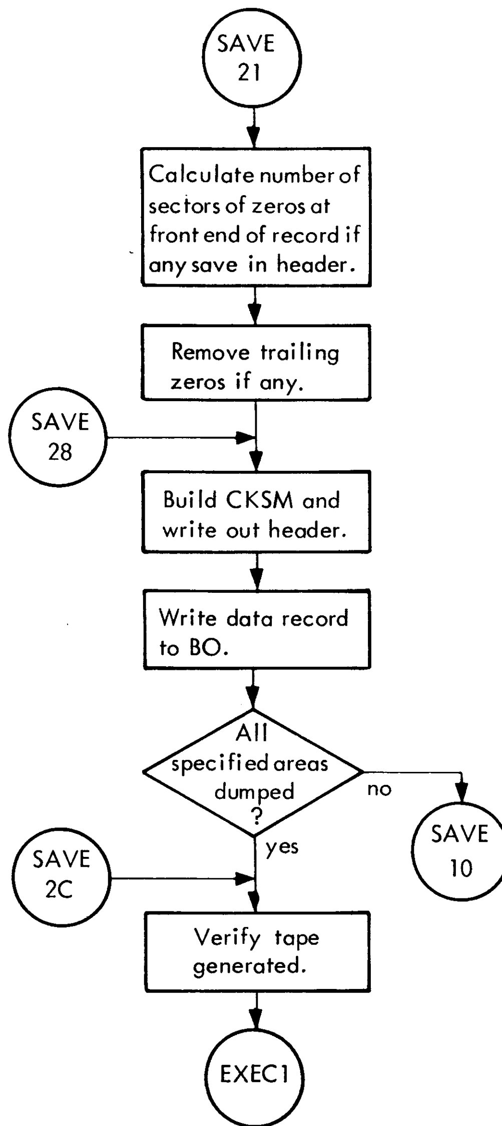


Figure 66. RADEDIT Flow, SAVE (cont.)

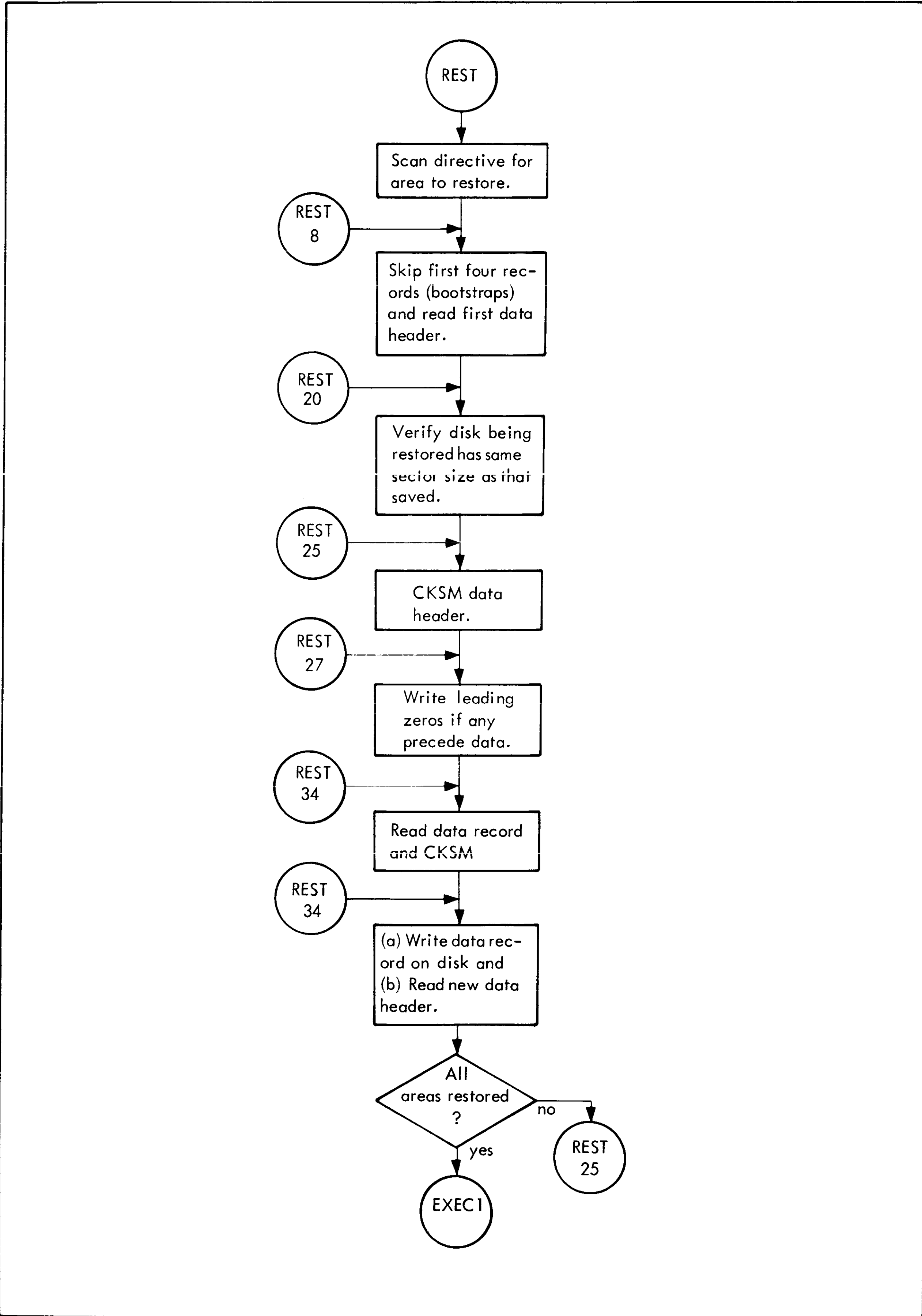


Figure 67. RADEDIT Flow, RESTORE

11. 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 RBM overlays, the Job Control Processor, and then writes these in to the RBM file in the SP area. A map containing the RBM table allocation and RAD allocation is output upon request. SYSLOAD terminates by reading in the disk Bootstrap and exiting to it, simulating a booting of the system from the disk.

Figure 68 illustrates the core layout of SYSGEN and SYSLOAD after the absolute object module is loaded by the Stand-Alone SYSGEN Loader.

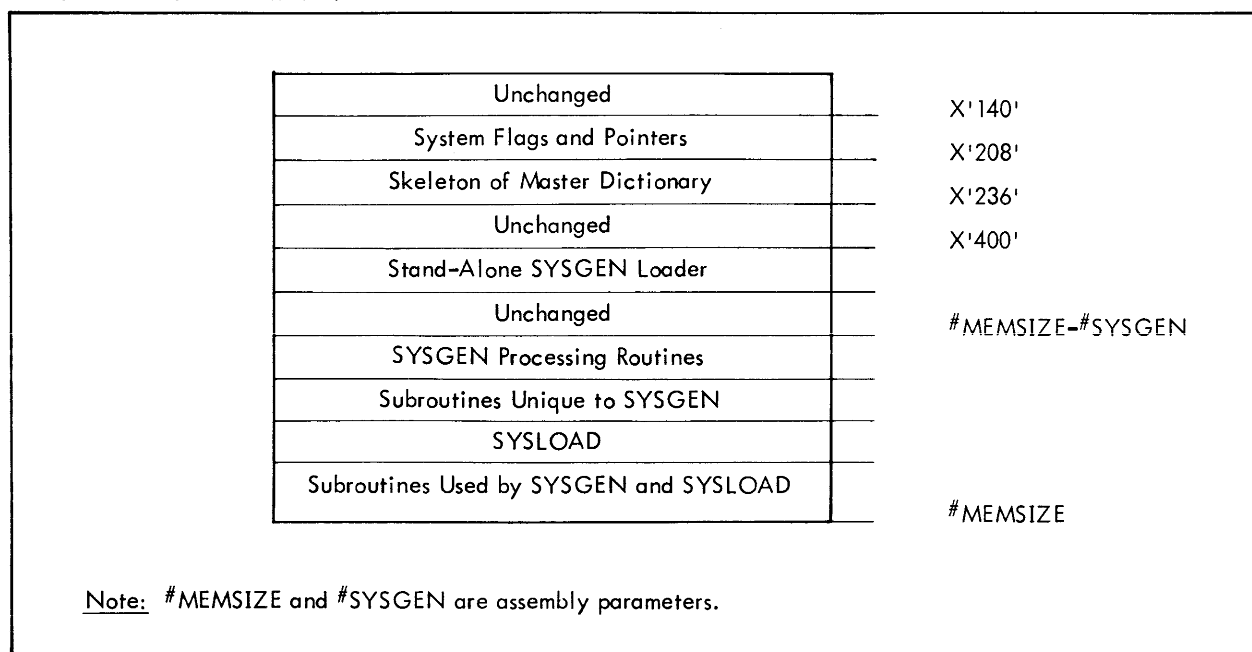


Figure 68. SYSGEN and SYSLOAD Layout Before Execution

Figure 69 depicts a typical core layout after SYSGEN and SYSLOAD have executed.

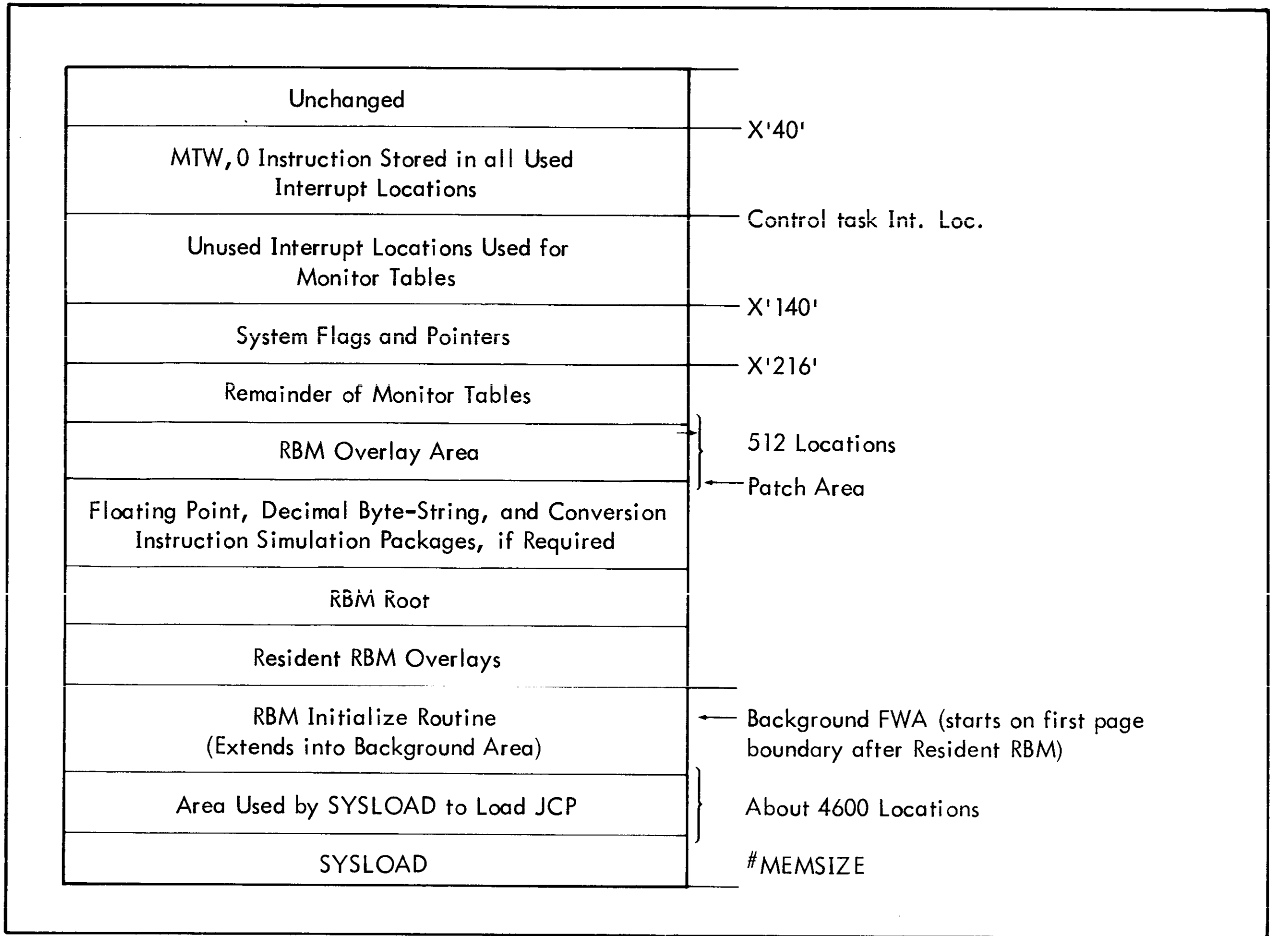


Figure 69. SYSGEN and SYSLOAD Layout After Execution

SYSGEN/SYSLOAD Flow

The flowcharts in Figure 70 depict the overall flow of SYSGEN and SYSLOAD. The labels used correspond to the labels in the program listing.

Loading Simulation Routines, RBM, and RBM Overlays

The S region of the SYSLOAD listing contains a loader that loads the instruction simulation packages, RBM, the RBM 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 8 are recognized by the Loader.

[†]This DEF must be the first load item in the ROM.

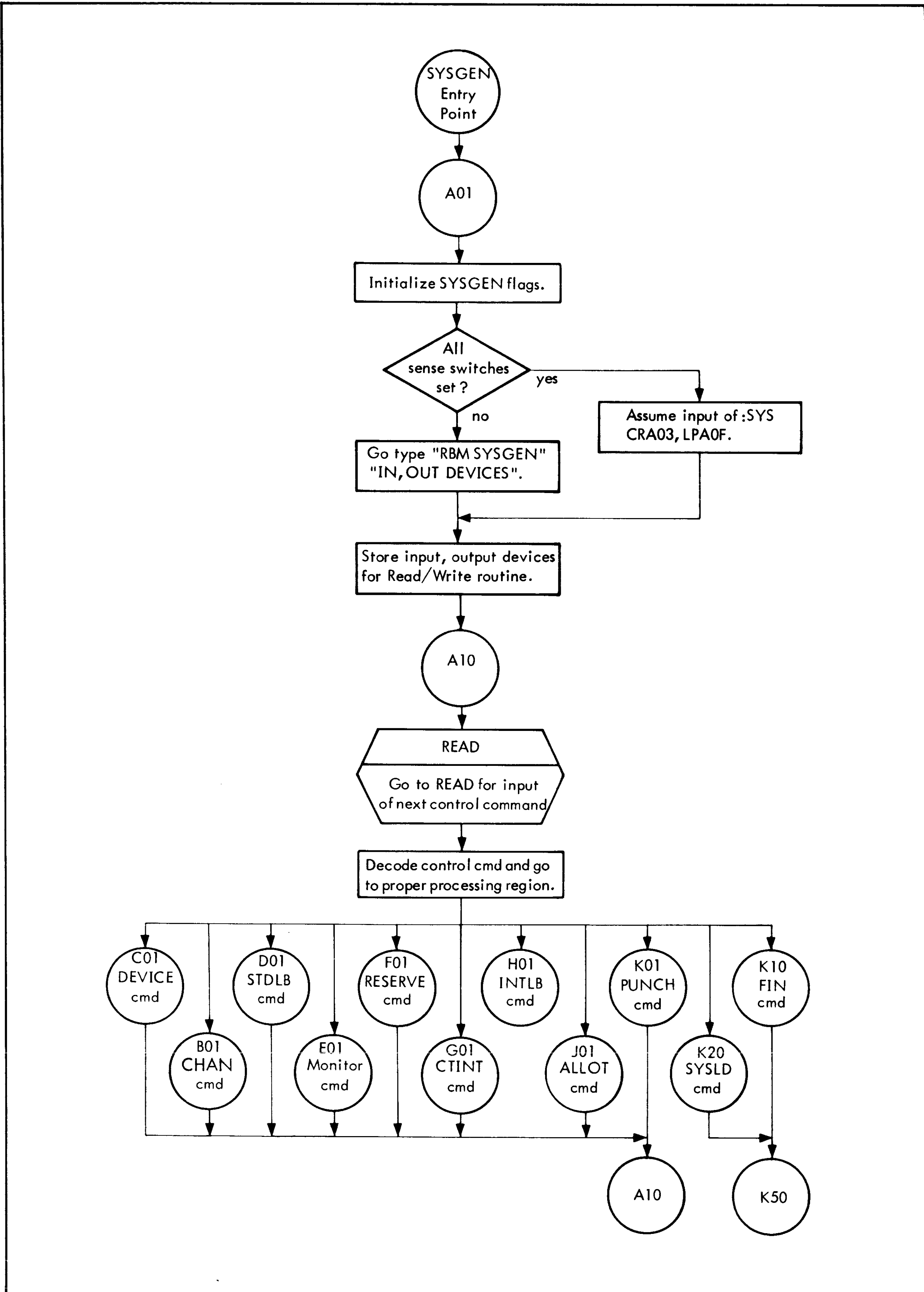


Figure 70. SYSGEN/SYSLOAD Flow

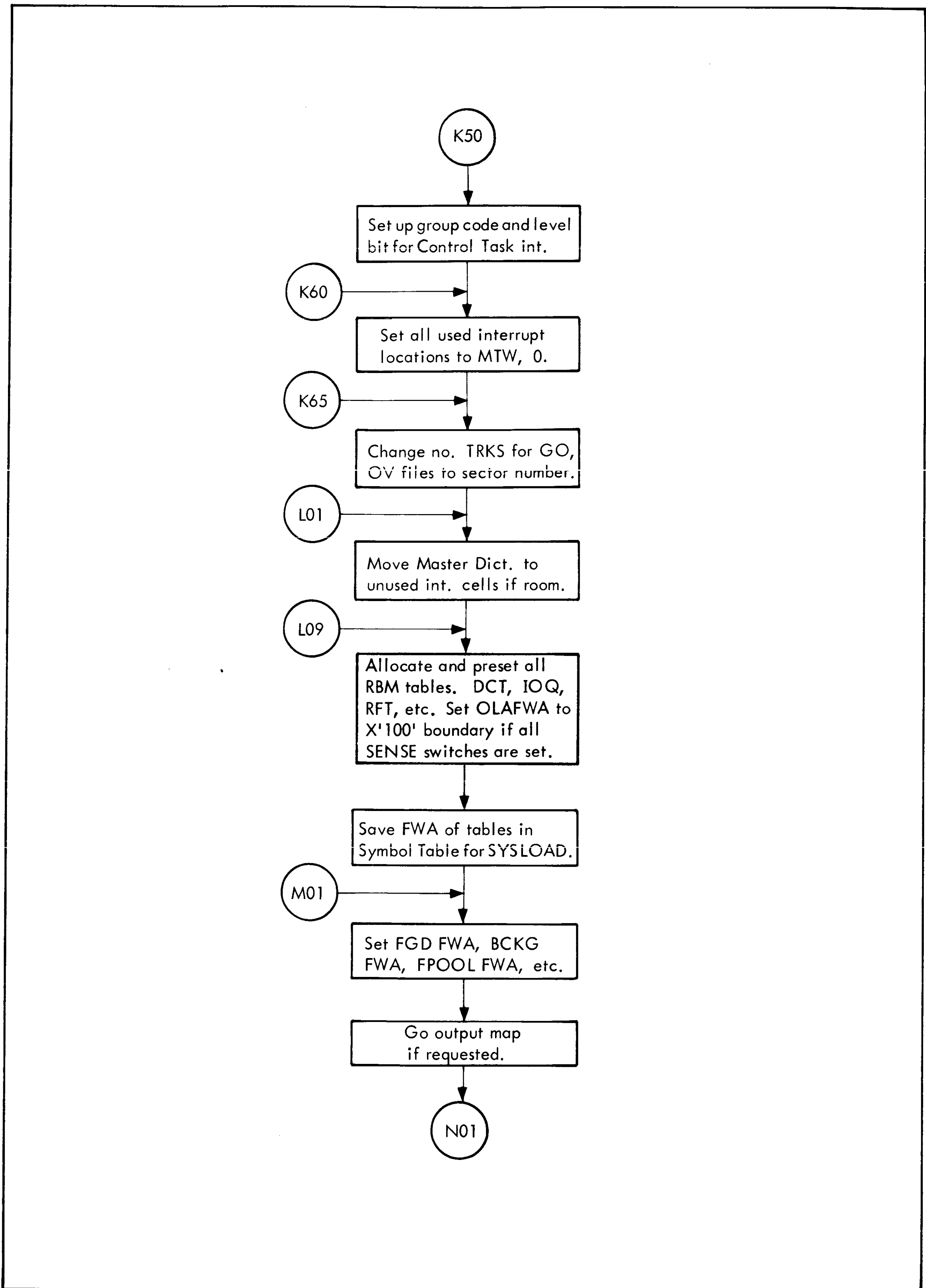


Figure 70. SYSGEN/SYSLOAD Flow (cont.)

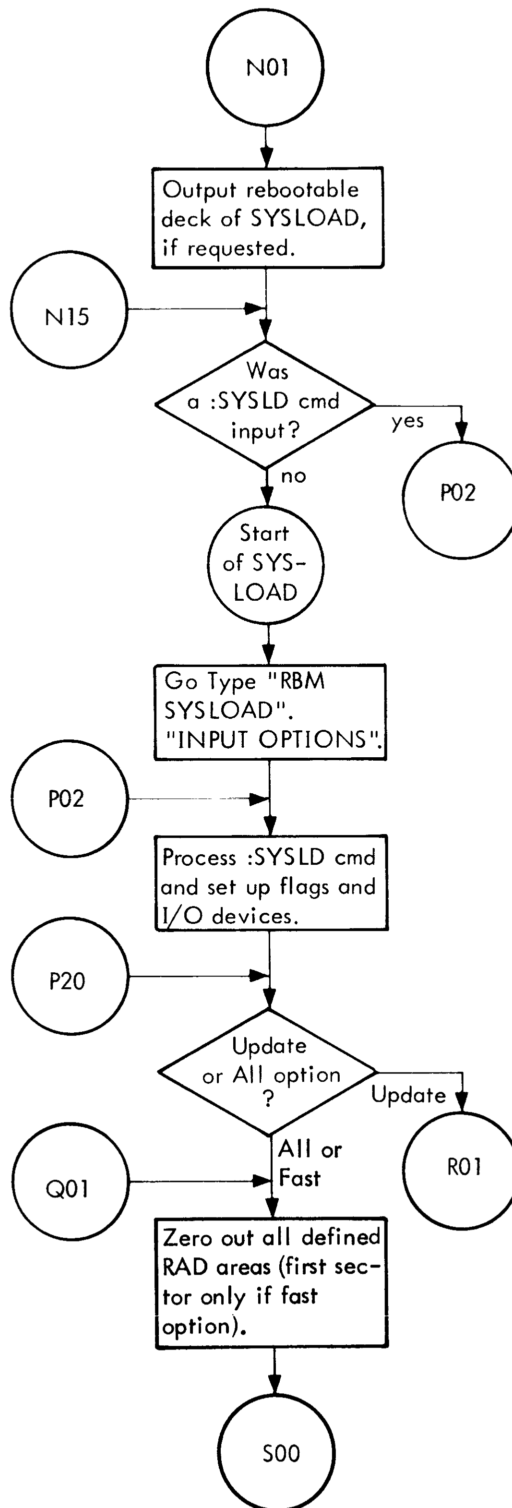


Figure 70. SYSGEN/SYSLOAD Flow (cont.)

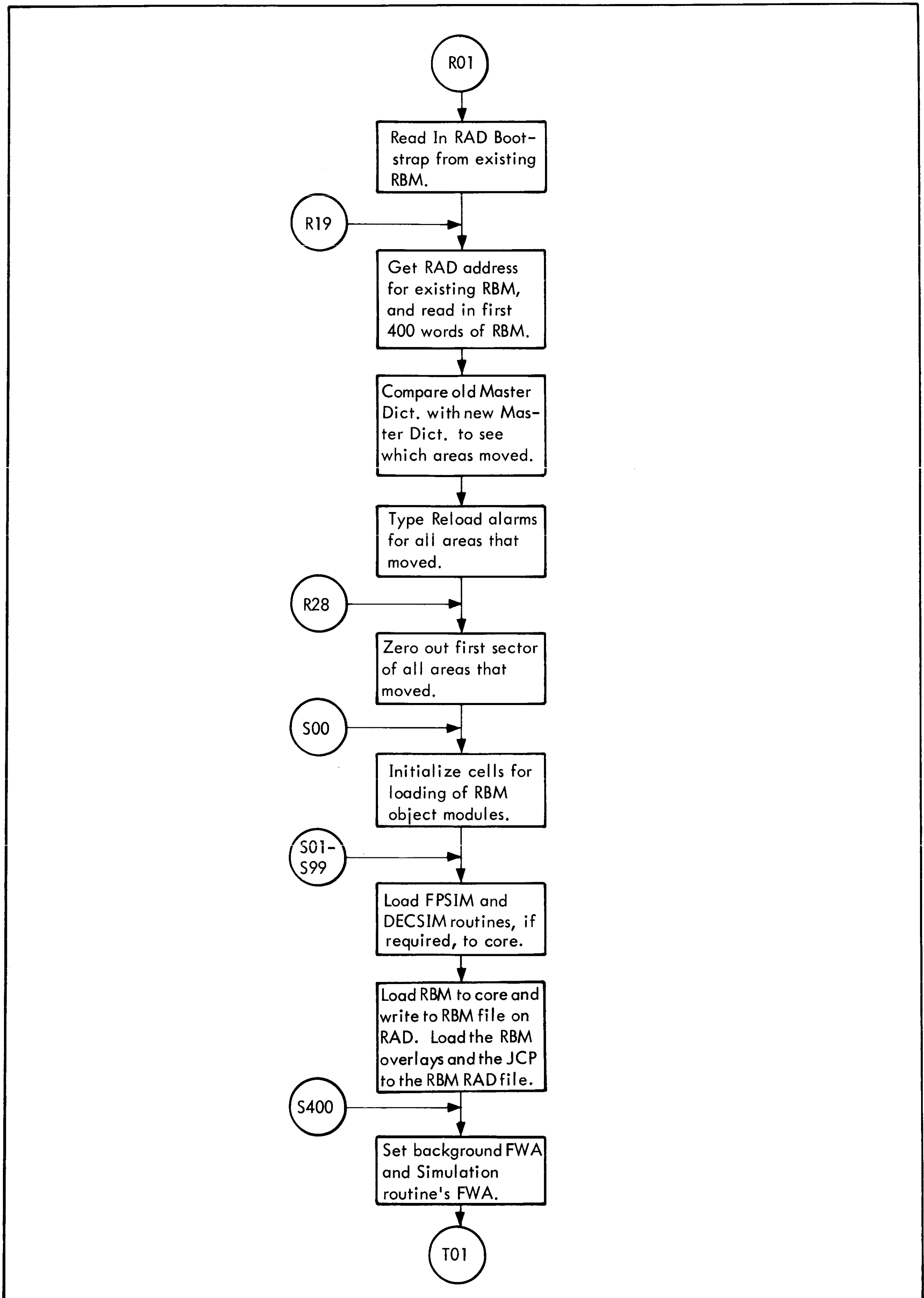


Figure 70. SYSGEN/SYSLOAD Flow (cont.)

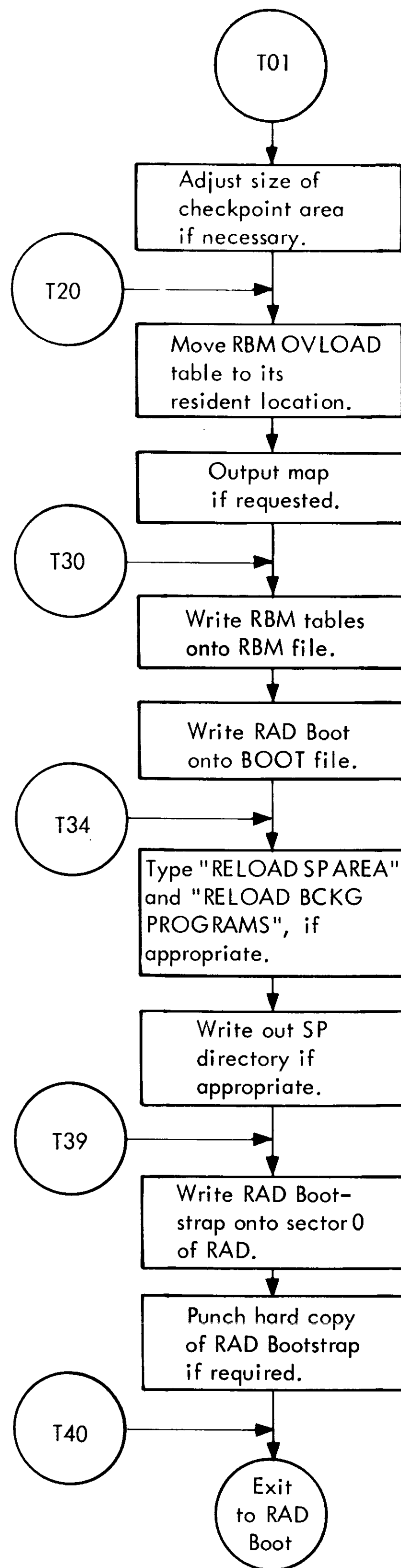


Figure 70. SYSGEN/SYSLOAD Flow (cont.)

Table 8. Standard SYSLOAD DEFs

DEF Name	Program
ABEX	Background abort/exit
ALLOT	ALLOT Service Calls
ARM	ARM/DISARM/CONNECT/DISCONNECT
BKL1	Background Loader
BYTSIM	Byte String Instruction Simulation Routine
CHECK	CHECK Service Calls
CKD	Crash Dump to LP
CKD2	Crash Dump to LP
CKPT	Checkpoint
CLOSEX	Close a DCB
CRD	Crash Dump to BI
CRS	Crash Save
CVSIM	Convert Instruction Simulation Routine
DECSIM	Decimal Instruction Simulation Routine
DELETE	Service Call
DEVI	DEVICE Service Calls
ENQ	Enqueue/Dequeue
ESU	Error Summary
EXTM	Termination Service Calls
FGL1	Run-time Loader
FGL2	Run-time Loader
FGL3	Run-time Loader
FPSIM	Floating Point Simulation Routine
GETNRT	I/O Subroutines
INIT	Boot time initialization
IOEX	IOEX Service Calls
KEY 1	Keyin Processor
KEY2	Keyin Processor
KEY3	Keyin Processor

Table 8. Standard SYSLOAD DEFs (cont.)

DEF Name	Program
KEY4	Keyin Processor
KEY5	Keyin Processor
KEY6	Keyin Processor
KEY7	Keyin Processor
LOG	Error Logger
LP	Line Printer Handlers
OPENX	Open a DCB
PINIT	INIT Service Calls
PRINT	PRINT Service Calls
READWRIT	READ/WRITE Service Calls
REWIND	REWIND Service Calls
RUN	RUN Service Calls
RWBFILE	Blocked File I/O
RWDEVF	Unblocked File I/O
SDBUF	Side Suffering Routines
SIGNAL	Signal Handler
SJOB	SJOB/KJOB Service Calls
SNAM	SETNAME Service Calls
STDLB	STDLB Service Calls
TAPE	Magnetic Tape Handlers
TERM	Task Termination
TMGETP	Task/ECB Subroutines
TMTYC	Task/ECB Subroutines
TRAPS	Trap Handling
TT	Task Termination
WAIT	WAIT Service Calls

The loader satisfies references to any of the RBM tables and RBM DEFs in the object modules it loads. References that can be satisfied are contained in the RBM Symbol Table. The address of each RBM table is stored in the Symbol Table by SYSGEN when the table is allocated. The address of each RBM DEF is stored when it is encountered during loading of the RBM object module.

All other references are treated as overlay entry-point references, and saved in the RBM Program Table. A more detailed discussion is given in the "Monitor Internal Services" chapter.

SYSGEN I/O

SYSGEN and SYSLOAD perform all of their own I/O via the READ/WRITE routine except for the typing of alarms performed by TYPE. The READ/WRITE routine will handle all standard SIGMA peripheral devices.

The READ/WRITE routine makes extensive use of tables (called IOT0 through IOT18) that fully describe the characteristics of each peripheral device. (See the comments in the program listing for descriptions of the READ/WRITE routine and the tables.) The paper tape format used by SYSGEN on read operations is identical to the format used by RBM described in Appendix B.

Rebootable Deck Format

If a :PUNCH control command is read by SYSGEN, a rebootable deck is output that includes the RBM tables with their initialized values, SYSLOAD, and the RBM Symbol Table.† This deck can be used to load a new version of RBM without re-inputting all the SYSGEN control commands.

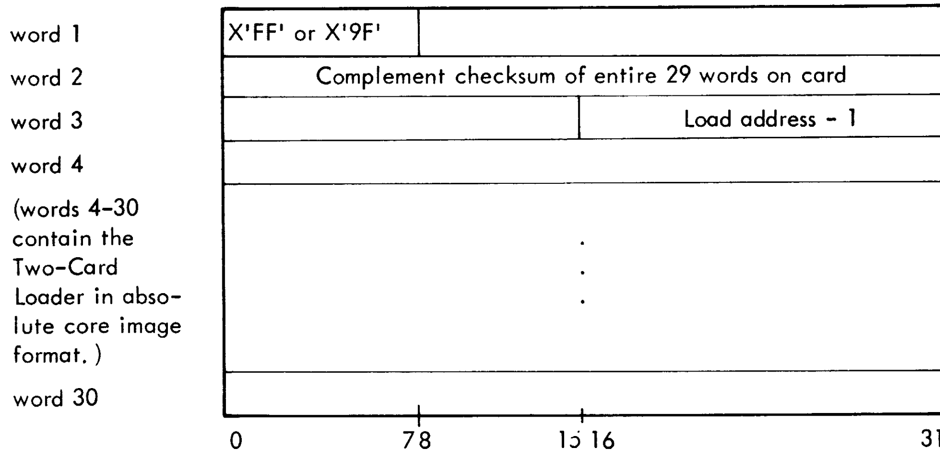
The first card in the rebootable deck consists of a one-card bootstrap program that loads the next two cards in the deck. These next two cards consist of a program that loads the remainder of the deck, consisting essentially of the RBM Table, SYSLOAD, and the RBM Symbol Table in core image format.

The two cards containing the Core Image Loader have the following format:

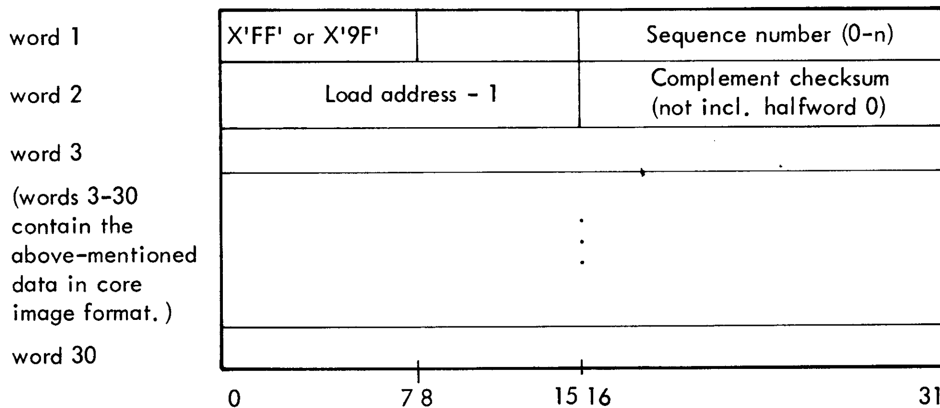
Byte No.	Contents
0	X'FF'(for card 1) X'9F'(for card 2)
1, 2, 3	Unused (all zeros)
4, 5, 6, 7	Complement checksum of entire card (carry out of bit 0 is ignored in computing checksum)
8, 9	Unused (all zeros)
10, 11	Load address, minus one, for following data
12-119	Loader in absolute core image format

† 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.

The core image format of the Two-Card Loader is



The RBM Tables, SYSLOAD, and the RBM Symbol Table are output in the core image format



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.

APPENDIX A. RBM SYSTEM FLAGS AND POINTERS

Table A-1. RBM System Flags and Pointers

Name	Location	Description
K:SYSTEM	X'2B'	Monitor Identification (RBMIDENT) have the following meaning: Bits 0-7 System-identification (X'30' = RBM). Bits 8-11 Version (C=3, D=4, etc.). Bits 12-15 Update (1, 2, 3, etc.). Bits 16-23 Reserved. Bits 24-25 00 - Sigma 5. 01 - Sigma 6/7. 10 - Sigma 9. 11 - Reserved. Bit 26 Reserved. Bit 27 Reserved. Bit 28 Reserved. Bit 29 Real-Time Routines. Bit 30 Reserved. Bit 31 Reserved.
K:BACKBG	X'140'	Beginning address of background.
K:BCKEND	X'141'	Ending address of background.
K:FGDBG1	X'142'	Current beginning address of FGD.
K:FGDEND	X'143'	Ending address of FGD.
K:CCBUF	X'144'	Address of Control Card Buffer.
K:BPOOL	X'145'	Start address of BCKG Blocking Buffer Pool.
K:FGDBG2	X'146'	Beginning address of FGD set at SYSGEN.
K:FMBOX	X'147'	Start address of FGD Mailboxes.
K:FPOOL	X'148'	Start address of FGD Blocking Buffer Pool.
K:UNAVBG	X'149'	Start address of unavailable memory.
K:MASTD	X'14A'	Start address of Master Dictionary.
K:NUMDA	X'14B'	Highest valid DW index for MASTD.
K:VRSION	X'14C'	RBM 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 here.
K:JCP1	X'150'	JCP and Control Task. Bits have the following meaning: Bit 0 = 1, JCP is executing. Bit 1 = 1, Background is active. Bit 2 = 1, Background is checkpointed on the RAD. Bit 3 = 1, Background is being used by Foreground but was not checkpointed. Bit 4 = 1, Waiting for key-in response. Bit 5 = 1, Skip to next JOB card. Bit 6 = 1, Set by ABORT for CALEXIT. Bit 7 = 1, Set by CALEXIT for ABORT.

Table A-1. RBM System Flags and Pointers (cont.)

Name	Location	Description
K:JCP1 (cont.)		Bits 8-15, Previous assign. of C device (for TY key-in). Bits 16-21, Unused. Bit 22 = 1, System processor executing. Bit 23 = 1, Execute BKGD Debug. Bits 24-25, 0 means no PMD requested. 1 means conditional PMD. 2 means unconditional PMD. Bit 26, Flag for CKPT that alarm typed. Bit 27 = 1, RBM Initialize routine is running. Bit 28 = 1, FG key-in active. Bit 29 = 1, TY key-in active. Bit 30 = 1, Attend command was input. Bit 31 = 1, JOB command was input.
K:CTST	X'151'	Flags to execute Control Task subtask. Bits have the following meaning: Bit 0 = 1, Execute CHECKPOINT. Bit 1 = 1, Execute FGD Loader/Releaser. Bit 2 = 1, Execute Restart. Bit 3 = 1, Time to service all devices. Bit 4 = 1, Execute ABORT/EXIT. Bit 5 = 1, Execute key-in. Bit 6 = 1, Execute PMD. Bit 7 = 1, BCKG is IDLE. Bit 8 = 1, Execute BCKG load. Bit 9 = 1, Load JCP. Bit 10 = 1, Load BCKG (Program not JCP). Bit 11 = 1, Key-in required by higher priority subtask. Bit 12 = 1, Recycle FGL1/2 to FGL1 for possible RLS. Bit 13 = 1, Execute error logger. Bit 14 = 1, CKPT deferred during BCKG abort. Bit 15 = 1, BKG in WAIT following attended mode abort. Bit 26 = 1, KEY2 doing STDLB disk file OPEN/CLOSE. Bit 27 = 1, FGL1 called from FGL2. Bit 28 = 1, Control Task is operating. Bit 29 = 0, Execute ABORT part of ABORT/EXIT. Bit 29 = 1, Execute EXIT part of ABORT/EXIT. Bit 30 = 1, PMD from key-in request. 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 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'	Area to load BCK program from.
K:ASSIGN	X'159'	Address of ASSIGN table.
K:RUNF	X'15A'	Post run status here for FGD load.
K:HIINT	X'15B'	HW0 = Control task interrupt number. HW1 = Highest address used for interrupt.

Table A-1. RBM System Flags and Pointers (cont.)

Name	Location	Description
K:FGDBG3	X'15C'	Begin address of FGD from FMEM key-in.
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 Response Buffer (6 words).
K:FGDBG4	X'16F'	Byte 0 = FWA of FGD prior to CKPT (Page Bits 15-31 = K:BCKEND prior to CKPT).
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 Bits 0 - 8, 1 bit for each X1 file. Bit set to 1 means SAVE file. 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 keyins.
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. RBM 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 ABORT, I/O HUNG.
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'	RBM stack pointer.

Table A-1. RBM System Flags and Pointers (cont.)

Name	Location	Description
K:MDNAME	X'212'	Address of MDNAME table.
K:DCT1X	X'213'	Address of DCT1 table.
K:RBMEND	X'214'	LWA of resident RBM.
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.

APPENDIX B. PAPER TAPE STANDARD FORMAT

A binary record is signaled by an X'11' as the first character, and the two bytes following are the record sizes. The specified number of data bytes follow the count.

An EBCDIC record is one whose first character is not an X'11'. An EBCDIC record is terminated by an NL code (15_{16}), or a blank frame (00).

APPENDIX C. ERROR LOGGING

The detection of a system, device, or software error will cause RBM 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 RBM:

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

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

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

The I/O sequence is SIO, TDV.

DEVICE TIMEOUT

Field(s)				Reference
X'12'	Count = D	Model Number		DCTMODX
Milliseconds Since Midnight				
HIO Status		I/O Address		DCT12
(unused)	HIO CC	TDV CC	TIO CC	-, DCT19, DCT20, DCT20A
Subchannel Status	(unused)	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
(unused)				
Seek Address				IOQ12

UNEXPECTED INTERRUPT

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

DEVICE ERROR

X'15'	Count = D	Model Number		DCTMODX
Milliseconds Since Midnight				
AIO Status		I/O Address		DCT12
(unused)	AIO CC	TDV CC	TIO CC	-, DCT19, DCT20, DCT20A
Subchannel Status	(unused)	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
(unused)				
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	

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			

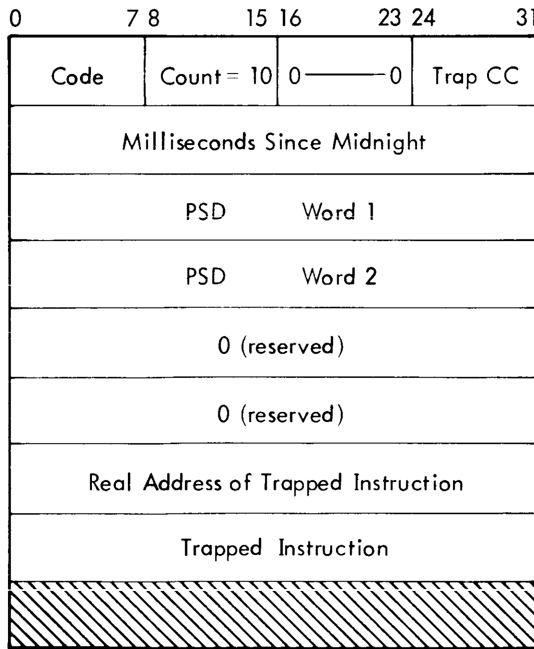
Generated by trap 4C.

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			

Generated by Trap 46.

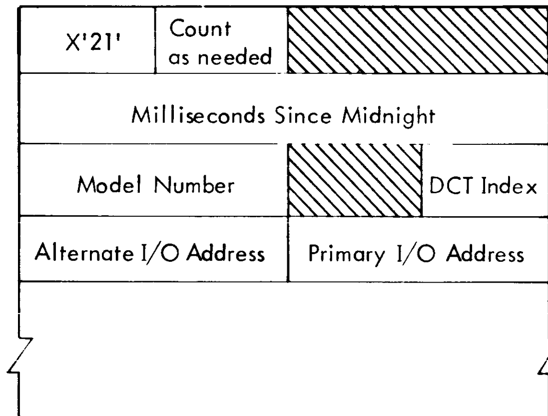
INSTRUCTION EXCEPTION



Generated by Trap 4D

CONFIGURATION RECORD

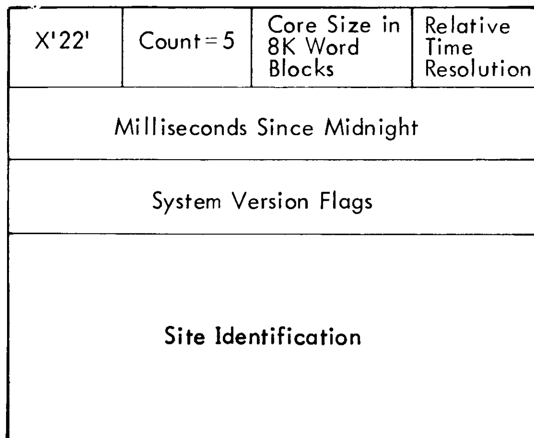
Entered at system STARTUP



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

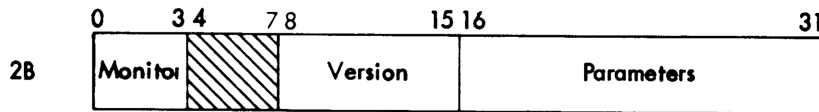
$n = 1$ 500 HZ

$n = 4$ 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 RBM 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	16 Bit RBM
3	32 Bit RBM
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

TIME STAMP

This record entered once each hour on the hour.

X'23'	Count = 3	(unused)
Milliseconds Since Midnight		
Year - 1900	Julian Day	

Binary integers

OPERATOR MESSAGE

X'27'	Count as Needed	(unused)
Milliseconds Since Midnight		
TEXTC Count		
TEXTC Message Max Size = 56 characters (CP-R)		

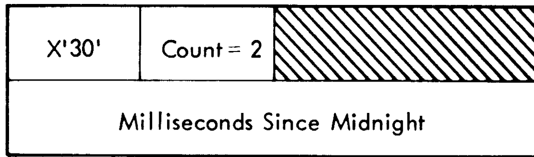
A facility is provided to inject messages from the computer operator (or diagnostic program) into the error log. The operator may enter these messages from the operator console via the ERRSEND key-in.

I/O ACTIVITY COUNT

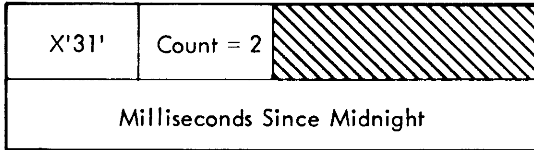
28	Count as needed	(unused)	DCT Index of First Device
Relative Time			
I/O $Address_1$		(unused)	DCT $Index_1$
I/O $Count_1$			
I/O $Address_2$		(unused)	DCT $Index_2$
I/O $Count_2$			
⋮			

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

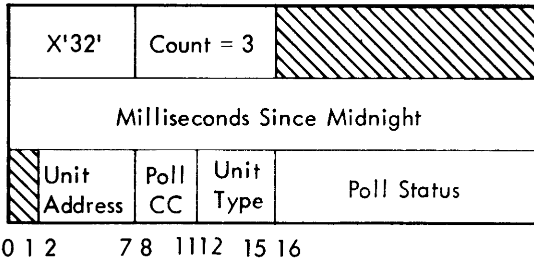
PFI PRIMARY RECORD



MFI PRIMARY RECORD

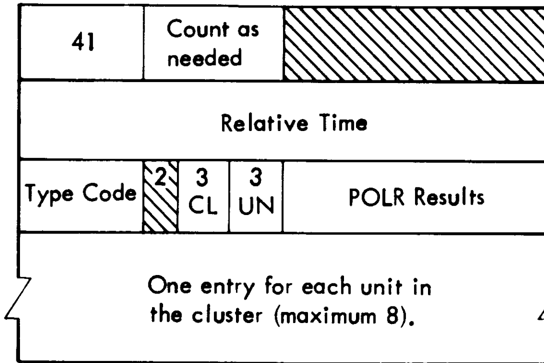


PROCESSOR POLL RECORD



One record produced per nonzero poll status received.

550 PROCESSOR CONFIGURATION

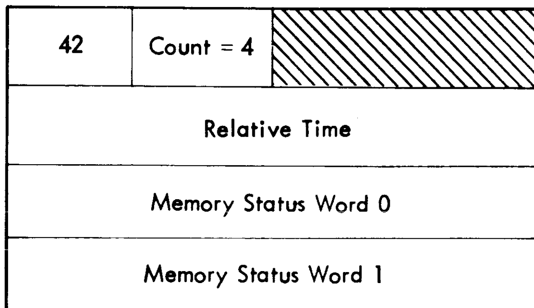


One record per cluster defined in SYSGEN.

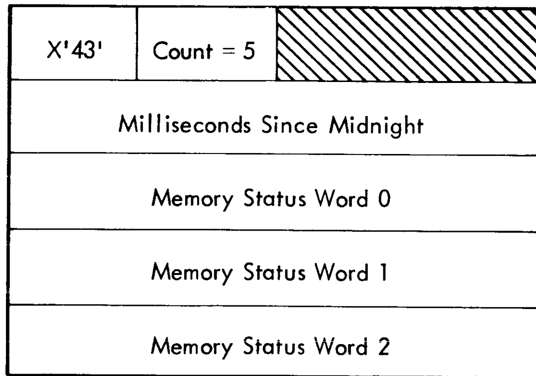
CL = cluster #
UN = unit #
TYPE = unit type

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

550 MEMORY PARITY SECONDARY RECORD



MEMORY POLL RECORD



APPENDIX D. XEROX STANDARD OBJECT LANGUAGE

INTRODUCTION

GENERAL

The Xerox standard object language provides a means of expressing the output of any Xerox 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 both cards and paper tape.

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 two-pass processor, such as the Xerox Meta-Symbol assembler, 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 CP-V, it is included in this manual because it applies to some of the processors operating under CP-V.

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 four 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

Sequence Number	
0	7

Byte 2

Checksum	
0	7

Byte 3

Record Size	
0	7

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 ($5 \leq$ record

size ≤ 108). The recordsize 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 a Symbol/Meta-Symbol 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. Each object module produced by an assembler is considered to consist of at least one control section. If no section is explicitly identified in the 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).

Declare Page Boundary 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

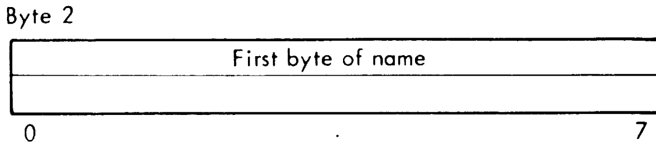
Control byte							
0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7

Byte 1

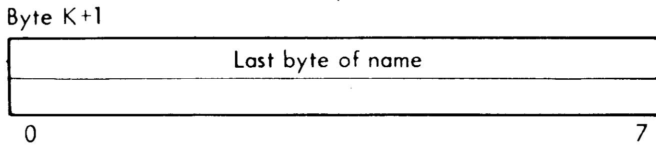
Name length, in bytes (K)	
0	7

[†] If the module has fewer than 256 previously assigned name numbers, this byte is absent.

Byte 2



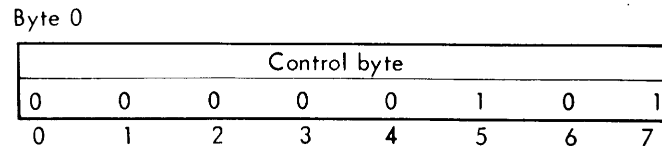
Byte K+1



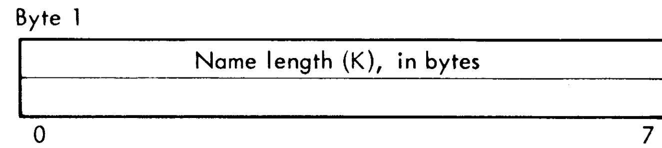
This item declares a label (in EBCDIC code) that is an external definition within the current object module. The name may not exceed 63 bytes in length.

Declare Primary External Reference Name

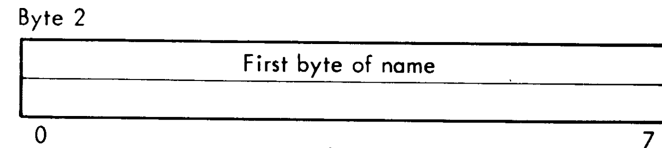
Byte 0



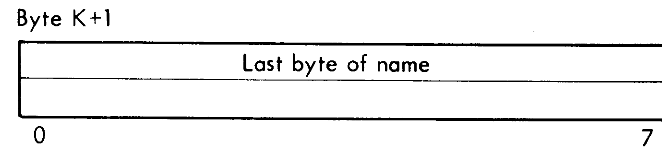
Byte 1



Byte 2



Byte K+1

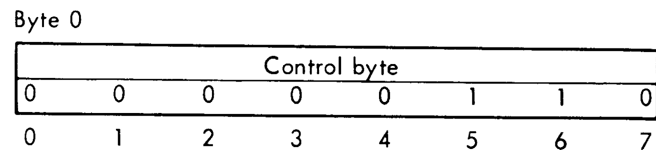


This item declares a symbol (in EBCDIC code) that is a primary external reference within the current object module. The name may not exceed 63 bytes in length.

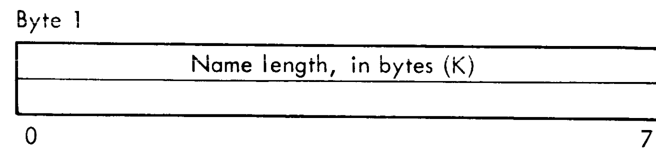
A primary external reference is capable of causing the loader to search the system library for a corresponding external definition. If a corresponding external definition is not found in another load module of the program or in the system library, a load error message is output and the job is errored.

Declare Secondary External Reference Name

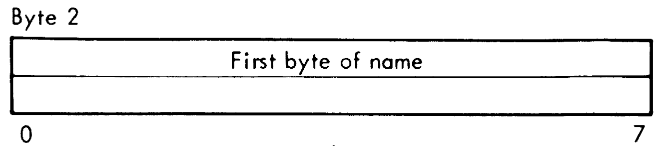
Byte 0



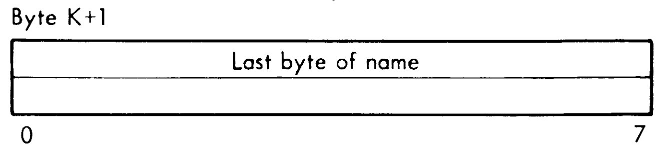
Byte 1



Byte 2



Byte K+1



This item declares a symbol (in EBCDIC code) that is a secondary 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.

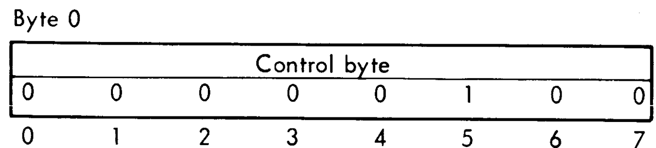
DEFINITIONS

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 "Expression Evaluation" in 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

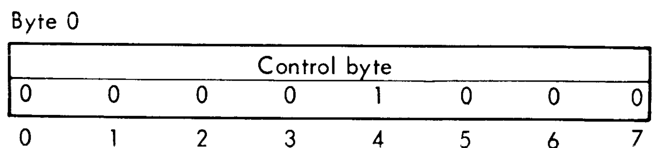
Byte 0



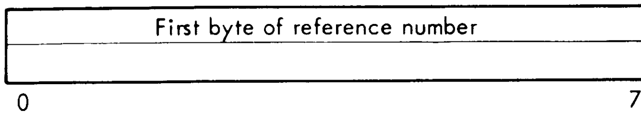
This item sets the loader's load-location counter to the value designated by the expression immediately following the origin control byte. This expression must not contain any elements that cannot be evaluated by the loader (see "Expression Evaluation" which follows).

Forward Reference Definition

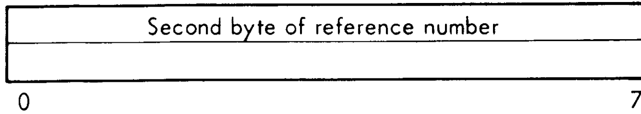
Byte 0



Byte 1



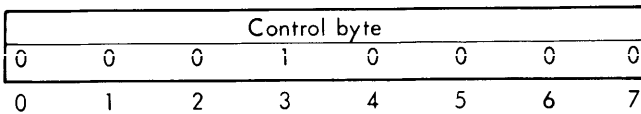
Byte 2



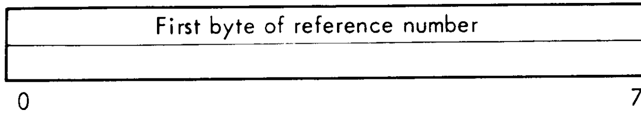
This item defines the value (expression) for a forward reference. The referenced expression is the one immediately following byte 2 of this load item, and must not contain any elements that cannot be evaluated by the loader (see "Expression Evaluation" which follows).

Forward Reference Definition and Hold

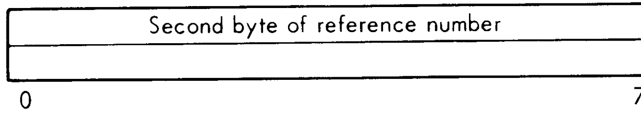
Byte 0



Byte 1



Byte 2

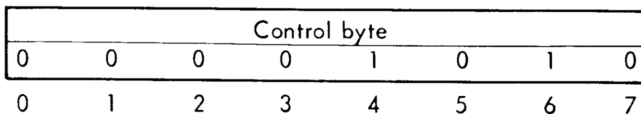


This item defines the value (expression) for a forward reference and notifies the loader that this value is to be retained in the loader's symbol table until the module end is encountered. The referenced expression is the one immediately following the name number. It may contain values that have not been defined previously, but all such values must be available to the loader prior to the module end.

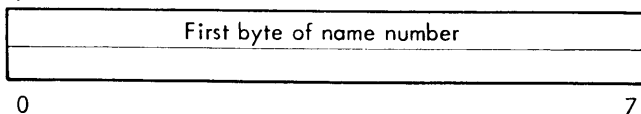
After generating this load item, the processor need not retain the value for the forward reference, since that responsibility is then assumed by the loader. However, the processor must retain the symbolic name and forward reference number assigned to the forward reference (until module end).

External Definition

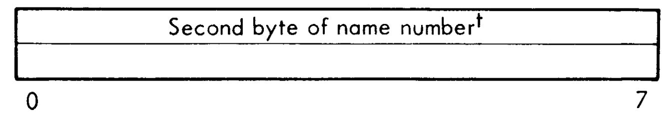
Byte 0



Byte 1



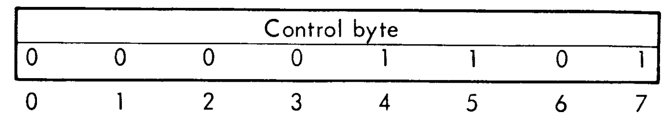
Byte 2



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.
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.

† If the module has fewer than 256 previously assigned name numbers, this byte is absent.

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 of 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 hava 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.

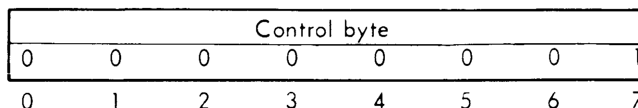
In the following diagrams of load item formats, RR refers to the address resolution code. The meaning of this code is given in the table below.

RR	Address Resolution
00	Byte
01	Halfword
10	Word
11	Doubleword

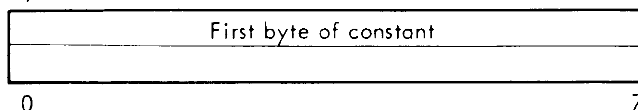
The load item discussed in this appendix, "Expression Evaluation", may appear only in expressions.

Add Constant

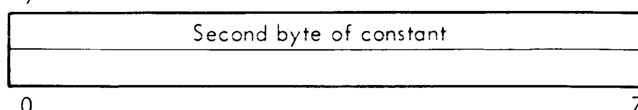
Byte 0



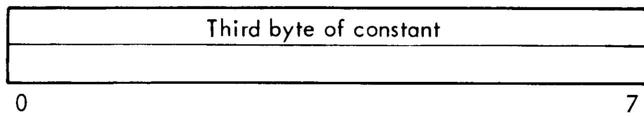
Byte 1



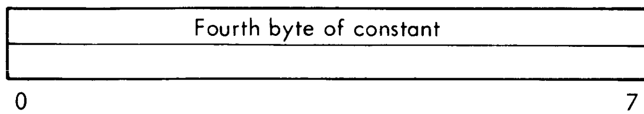
Byte 2



Byte 3



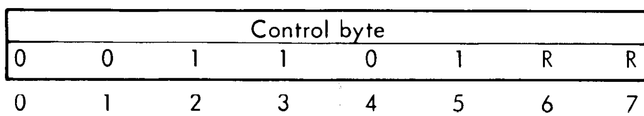
Byte 4



This item causes the specified four-byte constant to be added to the loader's expression accumulator. Negative constants are represented in two's complement form.

Add Absolute Section

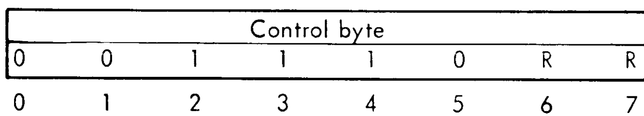
Byte 0



This item identifies the associated value (expression) as a positive absolute address. The address resolution code, RR, designates the desired resolution.

Subtract Absolute Section

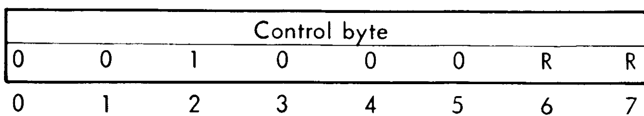
Byte 0



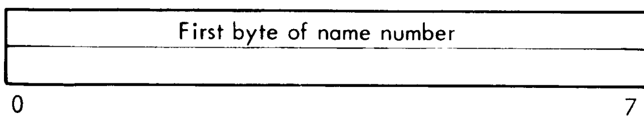
This item identifies the associated value (expression) as a negative absolute address. The address resolution code, RR, designates the desired resolution.

Add Value of Declaration

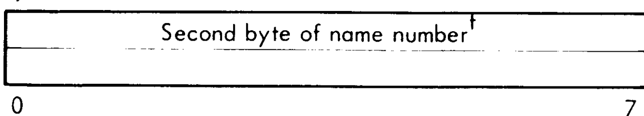
Byte 0



Byte 1



Byte 2



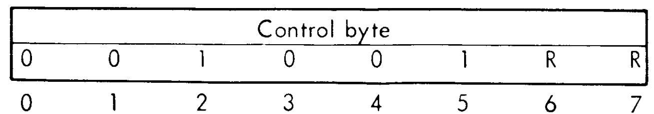
†If the module has fewer than 256 previously assigned name numbers, this byte is absent.

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.

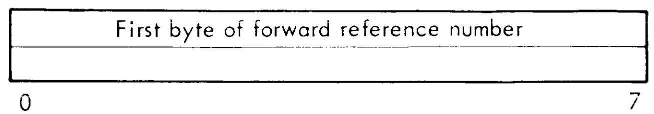
One such item must appear in each expression for a relocatable address occurring within a control section, adding the value of the specified control section declaration (i.e., adding the byte address of the first location of the control section).

Add Value of Forward Reference

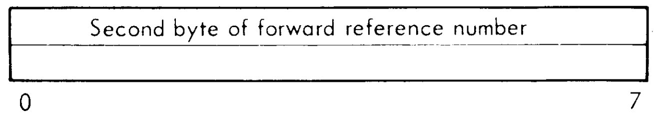
Byte 0



Byte 1



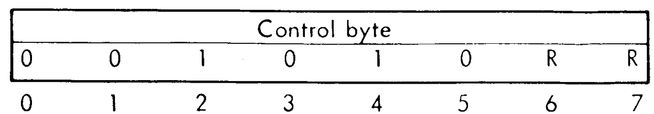
Byte 2



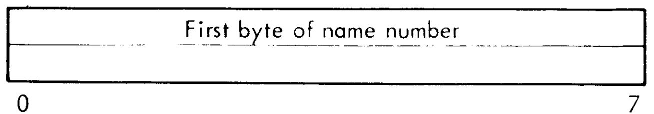
This item causes the value of the specified forward reference to be added to the loader's expression accumulator. The address resolution code, RR, designates the desired resolution, and the designated forward reference must not have been defined previously.

Subtract Value of Declaration

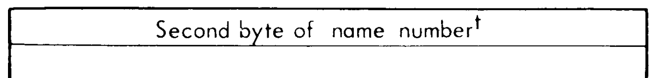
Byte 0



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.

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
100	Constant

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

Name number	
0	7

Byte 3 (if required)

Name number (continued)	
0	7

This item provides type information for external symbols. The Type and IR fields are defined above. The name 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	7

Byte 1

Type field (0–4)	IR field (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 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, n + 2

Two bytes 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 four-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 the module has fewer than 256 previously assigned name numbers, this byte is absent.

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).

Load Relocatable (Short Form)

Byte 0

Control byte							
1	C	D	D	D	D	D	D
0	1	2	3	4	5	6	7

This item causes a four-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

Control byte							
0	0	0	0	1	1	1	1
0	1	2	3	4	5	6	7

Byte 1

First byte of repeat count	
0	7

Byte 2

Second byte of repeat count	
0	7

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

Control byte							
0	0	0	0	0	1	1	1
0	1	2	3	4	5	6	7

Byte 1

Field location constant, in bits (K)	
0	7

Byte 2

Field length, in bits (L)	
0	7

This item defines a value (expression) to be added to a field in previously loaded information. The field is of length L ($1 \leq L \leq 255$) 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

Control byte							
0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

Padding bytes are ignored by the loader. The object language allows padding as a convenience for processors.

Module End

Byte 0

Control byte							
0	0	0	0	1	1	1	0
0	1	2	3	4	5	6	7

Byte 1

Severity level							
0	0	0	0	E	E	E	E
0	1	2	3	4	5	6	7

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 Meta-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	Object	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 ORGIN
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, AS
8				*			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	
	0302C1C1 (hexadecimal code comprising the load item)	Source Line 1
00000011	Declare external definition name (2 bytes) Name: AA Declaration number: 1	
	0302C2C2	
00000011	Declare external definition name (2 bytes) Name: BB Declaration number: 2	
	0302C3C3	
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 bytes) Name: RTN Declaration number: 5	

Begin Record	Record number: 0	
	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	
	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	

[†] 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.

Begin Record	Record number: 0	
	D269200000	
11010010	Load relocatable (short form). Relocate address field (word resolution)	Source Line 11
	Relative to forward reference number 18	
	The following 4 bytes: X'69200000'	
	4420000001	
01000100	Load absolute the following 4 bytes: X'20000001'	
	07EB0426000002	
00000111	Define field	
	Field location constant: 235 bits	Source Line 13
	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	
10000000	Load relocatable (short form). Relocate address field (word resolution)	Source Line 14
	Relative to declaration number 0	
	The following 4 bytes: X'680000CA'	
	8568000000	
10000101	Load relocatable (short form). Relocate address field (word resolution)	Source Line 15
	Relative to declaration number 5	
	The following 4 bytes: X'68000000'	
	08	
00001000	Define forward reference (continued in record 1)	Source Line 16

Begin Record	Record number: 1	
00011100	Record type: last, Mode: binary, Format: object language.	
00000001	Sequence number 1	Record Control Information
11101100	Checksum: 236	
01010001	Record size: 81	
	000C010000033C200002 (continued from record 0)	
	Number 12	
00000001	Add constant: 828 X'33C'	
00100000	Add value of declaration (byte resolution)	Source Line 16
	Number 0	
00000010	Expression end	
	42001	
01000010	Load absolute the following 2 bytes: X'0001'	
	080006010000000302	
00001000	Define forward reference	
	Number 6	Source Line 17
00000001	Add constant: 3 X'3'	
00000010	Expression end	
	080000010000000402	
00001000	Define forward reference	
	Number 0	Source Line 18
00000001	Add constant: 4 X'4'	
00000010	Expression end	

Begin Record	Record number: 1	
	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	
	Access code: Full access. Size 836 X'344'	
	0E00	
00001110	Module end	
	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

Object Code Control Byte	Type of Load Item
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 boundary 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 E. 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 F. SYSTEM OVERLAY ENTRY POINTS

ENTRY POINT NAME	OVERLAY NAME	DESCRIPTION
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
ARM	ARM	PROCESS CONNECT,ARM,DISCONNECT,DISARM CALS
BKLASSN	BKL1	DOES BACKGROUND DCB ASSIGNMENTS
BKL1	BKL1	PERFORM BACKGROUND LOADING FUNCTIONS
CALLQ	RWBFIL	SUB TO CALL QUEUE AND WAIT FOR I/O COMPLETION
CALLQP	RWBFIL	ENTRY TO CALLQ WITH PRESET PRIORITY
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
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
CKPT	CKPT	CHECKPOINT BACKGROUND (NOT IN MAPPED SYSTEM)
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
CORRES	DEVI	PROCESS CORRESPONDENCE CALS
CRD	CRD	CRASH DUMP FROM SE OP-LABEL
CRFIL4	CLOSEX	SUB. TO CLOSE A RAD FILE
CRS	CRS	CRASH SAVE TO SE OP-LABEL FROM CK AREA
CRS2	CRS2	CONTINUATION OF CRS
DBKG	ABEX	BACKGROUND DUMP DRIVER
DCBBUSY	READWR	SUB TO CHECK FOR AN I/O REQUEST TO A BUSY DCB
DEACTV	IOEX	PROCESS DEACTIVATE CALS
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
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)
DISC	DISC	DISC DEVICE HANDLERS
DISCCU		FIXED ARM DISC POST-HANDLER
DISCIO		FIXED ARM DISC PRE-HANDLER
DPAK		MOVABLE ARM DISC PRE-HANDLER
DPAKCU		MOVABLE ARM DISC POST-HANDLER
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	
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

ENTRY POINT NAME	OVERLAY NAME	DESCRIPTION
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
FGL1	FGL1	PRIMARY PROGRAM RELEASE
FGL2	FGL2	PRIMARY PROGRAM LOAD (INITIALIZE TABLES)
FGL2B04	FGL2	INTERNAL ENTRY TO FGL2
FGL2B30	FGL2	INTERNAL ENTRY TO FGL2
FGL3	FGL3	PRIMARY PROGRAM LOAD (READ IN ROOT AND PUBLIBS)
FGL3B81	FGL3	SEE IF SPACE IS AVAIL. FOR LMN OR PUBLIB LOAD
FINDBB	RWBFIL	GET A BLOCKING BUFFER
FINDDIR	OPENX	FIND THE SPECIFIED FILE ENTRY
FMBLDECB	GETNRT	BUILD AN I/O ECB
FMCHECK	CHECK	PROCESS I/O CHECK CALS
FMCKWP	READWR	CHECK FOR WRITE PROTECTION VIOLATIONS
FMCK1	CHECK	INTERNAL ENTRY TO FMCHECK
FMCK2	CHECK	INTERNAL ENTRY TO FMCHECK
FMCK3	CHECK	INTERNAL ENTRY TO FMCHECK
FMJCL	TTJOB	CLEAN UP RFT AND DCT ENTRIES AT JOB TERMINATION
FMMASTX	READWR	DETERMINE MASTD INDEX FOR AN AREA
FMOPL2AD	GETNRT	GET CALLER'S OPLBS2 TABLE ADDRESS
FPTBSY	READWR	CHECK FOR AN I/O REQUEST TO A BUSY FPT
GENCHARS	PRINT	PRINT EXPANDED TEXT FOR BREAK PAGES
GETDCBAD	GETNRT	GET DCB ADDRESS FROM FPT
GETDCTX	GETNRT	GET DEVICE INDEX FROM DCB
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
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
JMTENQ	TTJOB	CLEAN UP JOB LEVEL ENQS
JMTERM	TTJOB	DESTROY A JOB WHEN LAST TASK HAS TERMINATED
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
LOG	LOG	MOVE ERROR LOG RECORDS FROM LOG STACK TO ER OP LABEL
LP	LP	LINE PRINTER HANDLER PROLAY DUMMY ENTRY POINT
MODIFY	EXTM	SAME ENTRY AS STATUS
MTTYPE	REWIND	TEST FOR MAG TAPE
OPEN	READWR	PROCESS OPEN CALS
OPENDCB	READWR	ROUTINE TO OPEN A DCB
OPENX	OPENX	INTERNAL ENTRY TO OPENDCB
OUTSDBUF	SDBUF	OUTPUT SIDE BUFFERING LOGIC
PFIL	REWIND	PROCESS ALL PFIL CALS
PINIT	PINIT	PROCESS INIT CALS
PINTABNM	PINIT	SUB. TO PROCESS ABNORMAL ECB EXITS
PMD	ABEX	DISPATCH BKGD TO DUMP ITSELF
POLL	SIGNAL	PROCESS ALL POLL CALS
POLLABNM	SIGNAL	ROUTINE TO PROCESS POLL ECB ABRORMAL CONDITIONS
POLLCHK	SIGNAL	ROUTINE TO PROCESS CHECKS ON POLL SERVICES
PPOST	SIGNAL	PROCESS POST CALS

ENTRY POINT NAME	OVERLAY NAME	DESCRIPTION
PRECORD	REWIND	PROCESS PRECORD CALS
PRINT	PRINT	PROCESS PRINT CALS
PROMPT	DEVI	PROCESS SET PROMPT CHARACTER CALS
RBLOCK	RWBFIL	SUB TO READ A BLOCK INTO A BLOCKING BUFFER
READDIR	OPENX	SUB TO READ A DIRECTORY SECTOR
READWR	READWR	PROCESS READ/WRITE CALS
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
RWDEVF	RWDEVF	INTERNAL ENTRY TO READ/WRITE PROCESSING
RWRFILE	RWDEVF	READ/WRITE RANDOM RAD FILES
RWUFILE	RWDEVF	READ WRITE UNBLOCKED RAD FILE ROUTINE
SCAN	KEY3	COMMON SCAN ROUTINE FOR ALL KEY-IN ROUTINES
SDBUF	SDBUF	SIDE BUFFERING PROCESSOR PROLAY DUMMY ENTRY POINT
SEGLOAD	EXTM	PROCESS SEGLOAD CALS
SETNAME	SNAM	PROCESS SETNAME CALS
SETOVR	GETNRT	SUBR TO TEST/SET ABORT OVERRIDE IN I/O CALS
SETUP	REWIND	SUB TO OPEN A DCB AND GET ITS ASSIGNMENT
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
SJOB	SJOB	PROCESS SJOB CALS
SNAM	SNAM	PROCESS SETNAME CALS
SNAP	CRS	SNAP KEY-IN PROCESSING
START	SIGNAL	PROCESS START CALS
STATUS	EXTM	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
STPIO1	IOEX	PROCESS STOPIO/STARTIO CALS
STPIO2	IOEX	SAME ENTRY AS STPIO1(IOEX)
STRTIO1	IOEX	SAME ENTRY AS STPIO1(IOEX)
STRTIO2	IOEX	SAME ENTRY AS STPIO1(IOEX)
TAPE	TAPE	TAPE HANDLER PROLAY DUMMY ENTRY POINT
TERM	TERM	PROCESS TERM CALS
TEST	WAIT	PROCESS TEST CALS
TESTBUF	GETNRT	SUB TO TEST THE VALIDITY OF CALLER'S READ/WRITE BUFFER.
TESTWT4	GETNRT	ROUTINE TO TEST FOR DELETE-ON-POST I/O REQUEST
TEXIT	TRAPS	PROCESS TRAP EXIT CALS
TIME	WAIT	PROCESS TIME CALS
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 ITERM
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 ADDRESS
TMLM	TERM	SUBRTOUTINE 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
TMTRMT     TERM      SUB. TO TERMINATE A LOAD MODULE
TMTYC      TMTYC     SUB. TO SET FPT TYPE COMPLETION WORD PARAMETER
TMTYCB     TMTYC     SUB. TO SET FPT TYPE COMPLETION WORD BUSY
TMTYCS     TMTYC     SUBROUTINE 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
TRAP5      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
TTJOB      TTJOB     SUB. TO CLEAN JOB CONTROLS FOR TASK TERMINATION
TTPRIM     TT        SUB. TO DO MISC. TASK CLEANUP FOR PRIMARY TERMINATIONS
TYPE       PRINT     PROCESS ALL TYPE CALS
WAIT       WAIT      PROCESS  WAIT CALS
WAITALL    WAIT      PROCESS WAITALL CALS
WAITANY    WAIT      PROCESS  WAITANY CALS
WBLOCK     RWBFIL    SUB TO WRITE OUT A BLOCKING BUFFER
WEOF       REWIND    PROCESS  WEOF CALS
WLBLOCK    RWBFIL    SUB TO WRITE THE CURRENT BLOCK OF A RAD FILE
WRITDIR    OPENX     SUB TO WRITE A DIRECTORY SECTOR
```