digital

pdp11

disk operating system monitor

systems programmer's manual

digital pdp11
digital equipment corporation · maynard massachusetts
ADDRESS REGISTER
DATA
RUN
SWITCH REGISTER
SOURCE
DEST
14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
LOAD ADDR
EXAM
CONT
ENABLE
HALT
S-INST

DEC-11-OSPMA-A-D

PDP-11

DISK OPERATING SYSTEM MONITOR

(Version No. VØØ4A)

System Programmer's Manual

Software Support Category

The software described in this manual
is supported by DEC under Category I,
as defined on page iv of this manual.

This manual supersedes the Preliminary Edition,
issued under order no. DEC-11-MNDA-D.

For additional copies, order no. DEC-11-OSPMA-A-D from Digital Equipment Corporation, Software Distribution Center, Maynard, Massachusetts 01754

DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

Preliminary, November 1971
First Edition, May 1972

NOTE

Your attention is invited to the last two pages of this document. The "How To Obtain Software Information" page tells you how to keep up-to-date with DEC's software. The "Reader's Comments" page is beneficial to both you and DEC; all comments received are considered when documenting subsequent manuals.

Copyright © 1971, 1972 by Digital Equipment Corporation

Associated PDP-11 documents:

Disk Operating System Monitor, Programmer's Handbook
DEC-11-MWDC-D
Getting DOS on the Air
DEC-11-SYDD-D
PAL-11R Assembler, Programmer's Manual
DEC-11-ASDC-D
Edit-11 Text Editor, Programmer's Manual
DEC-11-EEDA-D
ODT-11R Debugging Program, Programmer's Manual
DEC-11-OODA-D
PIP File Utility Package, Programmer's Manual
DEC-11-PIDB-D
Link-11 Linker & Libr-11 Librarian, Programmer's Manual
DEC-11-ZLDC-D

NOTICE

This document is for information purposes, and is subject to change without notice.

Trademarks of Digital Equipment Corporation include:

DEC
DECtape
digital (logo)
COMTEX-11
PDP-11
RSTS-11
RSX-11
UNIBUS

P R E F A C E

This document explains the philosophy and structure of the PDP-11 Disk Operating System (DOS) Monitor. It is written for the PDP-11 programmer who needs to know the internal operation of the Monitor in more detail than is given in the PDP-11 Disk Operating System Monitor Programmer's Handbook (order no. DEC-11-MWDC-D), which is intended for the general user. Thus, this document should be of assistance to:

- Users who wish to adapt or extend the Monitor for their own special applications.
- Software Support Specialists.
- Programmers responsible for DOS Monitor maintenance or modification.

The document assumes familiarity with the contents of the Programmer's Handbook, and also with the DOS system programs as described in the documents listed on the back of the front page.

The source listings of the Monitor (available from DEC's Software Distribution Center) would be useful since this document is designed merely to supplement the detailed comments included in the source listings.

The document is divided into nine chapters:

Chapter 1 provides an overview of the concepts and organization of the DOS Monitor and gives general information on the form of the rest of the document.

Chapter 2 is devoted to the central executive section of the Monitor which is always resident in core memory.

Chapter 3 discusses the overall philosophy for the handling of I/O from any peripheral device, followed by a detailed examination of the routines providing program I/O services. This chapter also contains an introduction to device drivers.

Chapter 4 is particularly concerned with I/O upon bulk-storage devices for which a file-structure is defined. A general description of this structure is given, followed by an explanation of each of the special routines for file management.

Chapter 5 describes the modules which provide I/O services to load and unload the Monitor and certain general utilities.

Chapter 6 describes the methods adopted for the handling of operator commands at the console keyboard and the processing of each command. Also included is a discussion of a transient

Monitor section which occupies memory in the absence of a user program.

Chapter 7 covers the question of error diagnosis and outlines the routine providing a central printing service for messages from both the Monitor itself and the DOS system programs.

Chapter 8 shows how the Monitor is initially built and stored ready for use and can be later extended or modified.

Chapter 9 explains the format and contents of various system object and load modules.

Each chapter is concluded with its referenced illustrations.

NOTE

Appendix A, frequently referenced herein, was not available for this printing, thus it does not appear. In June 1972, the appendix will be available on request (free) from DEC's Software Distribution Center, Maynard, Mass. 01754.

SOFTWARE SUPPORT CATEGORIES

Digital Equipment Corporation (DEC) makes available four categories of software. These categories reflect the types of support a customer may expect from DEC for a specified software product. DEC reserves the right to change the category of a software product at any time. The four categories are as follows:

CATEGORY I
Software Products Supported at no Charge

This classification includes current versions of monitors, programming languages, and support programs provided by DEC. DEC will provide installation (when applicable), advisory, and remedial support at no charge. These services are limited to original purchasers of DEC computer systems who have the requisite DEC equipment and software products.

At the option of DEC, a software product may be recategorized from Category I to Category II for a particular customer if the software product has been modified by the customer or a third party.

CATEGORY II
Software Products that Receive Support for a Fee

This category includes prior versions of Category I programs and all other programs available from DEC for which support is given. Programming assistance (additional support), as available, will be provided on these DEC programs and non-DEC programs when used in conjunction with these DEC programs and equipment supplied by DEC.

CATEGORY III
Pre-Release Software

DEC may elect to release certain software products to customers in order to facilitate final testing and/or customer familiarization. In this event, DEC will limit the use of such pre-release software to internal, non-competitive applications. Category III software is only supported by DEC where this support is consistent with evaluation of the software product. While DEC will be grateful for the reporting of any criticism and suggestions pertaining to a pre-release, there exists no commitment to respond to these reports.

CATEGORY IV
Non-Supported Software

This category includes all programs for which <u>no support</u> is given.

C O N T E N T S

		Page
CHAPTER 1	INTRODUCTION	1-1
1.1	The Function of the DOS Monitor	1-2
1.2	Monitor Organization	1-3
1.3	Monitor Conventions	1-4
1.3.1	Calling Monitor Modules	1-5
1.3.2	Return State	1-5
1.3.3	Naming Conventions	1-6
1.3.4	Definitions	1-6
CHAPTER 2	RESIDENT MONITOR	2-1
2.1	Monitor Tables	2-2
2.1.1	System Vector Table	2-2
2.1.2	Monitor Residency Table	2-4
2.1.3	Device Driver List	2-5
2.1.4	Table Initialization	2-7
2.2	EMT Handler	2-9
2.2.1	Description of the Handler	2-9
2.2.2	General Comments	2-11
2.3	Swap Area Management	2-12
2.3.1	Swap Buffers	2-13
2.3.2	Calling SAM	2-14
2.3.3	Description of SAM	2-15
2.3.4	System Exit	2-16
2.3.5	The Swappable Routines	2-17
2.4	General Purpose Subroutines	2-20
2.4.1	Register Save/Restore & Stack Control	2-20
2.4.2	Free Core Management	2-22
2.4.2.1	Buffer Allocation	2-23
2.4.2.2	Buffer Release	2-24
2.4.2.3	Comments	2-24
2.5	Time Control	2-25
CHAPTER 3	GENERAL I/O PROCESSING	3-1
3.1	I/O Concepts and Control	3-2
3.1.1	General Strategy	3-2
3.1.1.1	Common Processing	3-2
3.1.1.2	Buffer Data	3-3
3.1.1.3	Dynamic Core Usage	3-3
3.1.1.4	I/O Levels	3-4
3.1.1.5	Device Assignment	3-5
3.1.2	I/O Controls	3-5
3.1.2.1	User Link Block	3-6
3.1.2.2	Device Assignment Table	3-7
3.1.2.3	Dataset Data Block	3-9
3.1.2.4	Driver Management	3-11
3.2	I/O Processing	3-15
3.2.1	Basic I/O Processing	3-15
3.2.1.1	Dataset Initialization	3-16
3.2.1.2	Basic Transfers	3-18
3.2.1.3	Dataset Release	3-20

Chapter 3, continued. Page

3.2.2	Normal I/O Processing	3-22
3.2.2.1	Dataset Open (OPN)	3-22
3.2.2.2	READ/WRITE Transfers (RWN)	3-26
3.2.2.3	Dataset Close (CLS)	3-33
3.2.3	Random Access I/O (BLO)	3-35
3.2.4	Special Operations	3-37
3.2.4.1	Special Functions (SPC)	3-38
3.2.4.2	Device Status (STT)	3-40
3.3	Device Drivers	3-41
3.3.1	Driver Interface Table	3-41
3.3.2	Driver Service Routines	3-45
3.3.3	Interrupt Servicing	3-45
3.3.4	System-Device Drivers	3-46
CHAPTER 4	FILE STRUCTURES	4-1
4.1	General Concepts	4-2
4.1.1	Files	4-2
4.1.1.1	Linked Files	4-2
4.1.1.2	Contiguous Files	4-3
4.1.2	Directories	4-4
4.1.2.1	Master File Directory (MFD)	4-5
4.1.2.2	User File Directory (UFD)	4-6
4.1.3	Bit Maps	4-8
4.1.4	File Protection	4-10
4.2	Application By Device	4-12
4.2.1	Fixed-Head Disks	4-12
4.2.2	Moving-Head Disks	4-13
4.2.3	DECtape	4-14
4.3	File Management	4-17
4.3.1	User File Block	4-19
4.3.2	File Information Block	4-20
4.4	General Purpose Routines	4-23
4.4.1	Directory Search (LUK)	4-23
4.4.2	Check Access, Set-Up and Release FIB (CKX)	4-25
4.4.2.1	Check Access Privilege	4-25
4.4.2.2	Set-Up File Information Block	4-27
4.4.2.3	Release File Information Block	4-28
4.4.3	Transfer Bit Map to and from Core (GMA)	4-29
4.4.3.1	Get Map (GETMAP)	4-29
4.4.4	Allocation of Blocks to Linked Files (LBA)	4-31
4.5	Normal File Processing	4-33
4.5.1	Opening Files	4-33
4.5.1.1	Open an Existing File (FOP)	4-34
4.5.1.2	Create a New File (FCR)	4-36
4.5.2	Processing a File	4-38
4.5.2.1	Next Block Determination (RWN)	4-39
4.5.2.2	Changing the Core Map Segment (GNM)	4-42
4.5.3	Closing Files (FCL)	4-45
4.6	Housekeeping Operations	4-48
4.6.1	Allocating Contiguous Files	4-48
4.6.1.1	Allocate Set-Up (ALO)	4-48
4.6.1.2	Contiguous Block Allocator (CBA)	4-50

Chapter 4, continued Page

4.6.2 Deleting Files 4-52
4.6.2.1 Deletion Set-Up (DEL) 4-52
4.6.2.2 Deletion of Contiguous Files (DCN) 4-54
4.6.2.3 Deletion of Linked Files (DLN) 4-55
4.6.3 Appending Files 4-56
4.6.3.1 Append General Routine (APP) 4-56
4.6.3.2 Special Append Operations on DECtape (AP2) 4-58
4.6.4 Renaming Files (REN) 4-59
4.6.5 Protecting Files (PRO) 4-60
4.6.6 Directory Status (DIR) 4-61
4.7 Magnetic Tape Structure 4-64
4.7.1 Opening Files on Magnetic Tape (MTO) 4-65
4.7.2 Special Operations 4-68

CHAPTER 5 OTHER PROGRAM SERVICES 5-1
5.1 Program Loading 5-2
5.1.1 Program Loader (LDR) 5-4
5.1.2 Monitor Module Loader (LD2) 5-7
5.2 General Utilities Package (GUT) 5-10
5.3 Conversion Utilities Package (CVT) 5-12
5.3.1 Conversion to Binary 5-12
5.3.1.1 Radix-50 Pack (code Ø) 5-14
5.3.1.2 Decimal ASCII to Binary (code 2) 5-15
5.3.1.3 Octal ASCII to Binary (code 4) 5-16
5.3.2 Conversions from Binary 5-16
5.3.2.1 Radix-50 Unpack (code 1) 5-17
5.3.2.2 Binary to Decimal ASCII (code 3) 5-19
5.3.2.3 Binary to Octal ASCII (code 5) 5-19
5.4 Command String Interpreter 5-20
5.4.1 Syntax Analyser (CSX) 5-21
5.4.2 Specification Decoder (CSM) 5-24
5.5 Program Exit (XIT) 5-28

CHAPTER 6 KEYBOARD SERVICES 6-1
6.1 General Philosophy 6-2
6.2 Organization 6-4
6.2.1 Command Acceptance 6-4
6.2.2 Command Decoding 6-6
6.2.3 Command Processing 6-7
6.2.4 Command Clean-Up 6-8
6.2.5 Command Conventions 6-9
6.2.5.1 Calling Parameters 6-9
6.2.5.2 Processor Stack 6-9
6.2.5.3 Call Techniques 6-10
6.2.5.4 Reentrancy 6-11
6.2.5.5 Residency 6-12
6.3 Common Command Processing 6-13
6.3.1 Console Keyboard Listener (RMON5) 6-13
6.3.2 The Special Console Driver (KBL) 6-18
6.3.3 Keyboard Commant Interpreter (KBI) 6-24
6.3.3.1 The WAIT Command 6-27
6.3.3.2 The CONTINUE Command 6-27
6.3.3.3 The STOP Command 6-28

Chapter 6, continued | Page

6.4 Run-Time Commands 6-28
6.4.1 The DATE Command (KBI.DA) 6-29
6.4.2 The SAVE Command (KBI.SA) 6-32
6.4.3 The ECHO, END & PRINT Commands (KBI,KB) 6-38
6.4.4 The ODT Command (KBI.OD) 6-41
6.4.5 The BEGIN and RESTART Commands (KBI.BE) 6-43
6.4.6 The KILL Command (KBI.KI) 6-50
6.4.7 The TIME Command (KBI.TI) 6-52
6.4.8 The MODIFY Command (KBI.MO) 6-54
6.4.9 The ASSIGN Command (KBI.AS) 6-59
6.4.10 The DUMP Command (KBI.DU) 6-65
6.5 Between-Program Services 6-71
6.5.1 The Transient Monitor (TMON) 6-71
6.5.2 The TMON Commands 6-79
6.5.2.1 The LOGIN Command 6-79
6.5.2.2 The FINISH Command 6-81
6.5.3 The RUN and GET Commands 6-83

CHAPTER 7 ERROR HANDLING 7-1
7.1 Types of Error 7-1
7.2 Calling Diagnostic Print 7-2
7.3 Diagnostic Print Routine 7-4

CHAPTER 8 MONITOR GENERATION and MODIFICATION 8-1
8.1 Monitor Module Preparation 8-1
8.1.1 Module Assembly 8-1
8.1.2 Module Linking 8-2
8.1.2.1 Resident Monitor 8-2
8.1.2.2 The Transient Monitor 8-3
8.1.2.3 Keyboard Language 8-4
8.1.2.4 Other Modules 8-4
8.2 System Building 8-4
8.2.1 Setting Up a System DECtape 8-5
8.2.2 Loading the System Device (SYSLOD) 8-7
8.2.2.1 Preparation of SYSLOD 8-7
8.2.2.2 SYSLOD Usage 8-8
8.2.2.3 SYSLOD Processing 8-8
8.2.2.4 Monitor Booting 8-11
8.3 Monitor Modification 8-12
8.3.1 Resident Monitor Modules 8-13
8.3.2 Program Services 8-14
8.3.3 Device Drivers 8-18
8.3.4 Keyboard Commands 8-18
8.3.4.1 New TMON Commands 8-19
8.3.4.2 Other Commands 8-19

CHAPTER 9 SYSTEM FILE FORMATS 9-1
9.1 Object Module Format 9-1
9.1.1 Object Module's Contents 9-2
9.1.1.1 Global Symbol Directory (GSD) 9-2
9.1.1.2 Text Block 9-5
9.1.1.3 Relocation Directory (RLD) 9-6
9.1.1.4 Internal Symbol Directory 9-6

9.1.2	Object Language	9-7
9.2	Load Module Format	9-13
9.2.1	Load Module Contents	9-14
9.2.1.1	COMD Contents	9-15
9.3	Object Module Library Format	9-17
9.3.1	Directory Contents	9-18
9.3.2	Object Module Contents	9-18
9.4	Load Module Library Format	9-19
9.4.1	Directory Contents	9-19
9.4.2	Load Module Contents	9-19

ILLUSTRATIONS

Figure	Title	Page
1-1	The Basic Resident Monitor in Memory	1-7
1-2	Monitor Library on the System Device	1-8
1-3	Memory After Program Load	1-9
1-4	Memory During Program Run	1-9
1-5	DOS-11 Monitor Module Structure	1-10
2-1	Usage of the Fixed Vector Locations	2-26
2-2	System Vector Table Content	2-27
2-3	Monitor Residency Table Format	2-28
2-4	Device Driver List Format	2-29
2-5	EMT Handler Operations	2-30
2-6	Stack Shuffle to Remove Call Arguments for Immediate Program Exit From the EMT Handler	2-31
2-7	Swap Buffer Format	2-32
2-8	SAM Operations	2-33
2-9	Memory Usage During I/O Set-Up	2-34
2-10	Buffer Allocation Management	2-35
2-11	Buffer Allocation Operations	2-36
2-12	Buffer Release Operations	2-37
3-1	User Link Block	3-48
3-2	Possible Device Assignment Table	3-49
3-3	Device Assignment Table Entry Format	3-50
3-4	Dataset Data Block (DDB)	3-51
3-5	Monitor DDB Chain	3-52
3-6	DDB Status Word	3-52
3-7	Driver Queue Management	3-53
3-8	A Driver Queue	3-54
3-9	Stack State In S.CDQ After Driver Completion Return	3-54
3-10	Flowchart For Initialization Module (INR)	3-55
3-11	Memory After Dataset Initialization	3-56
3-12	TRAN Block Format	3-57
3-13	Flowchart For Dataset Release Routine (RLS)	3-58
3-14	Flowchart For Dataset Open Routine (OPN)	3-59
3-15	Memory State After Dataset Open	3-60
3-16	User Line Format	3-61
3-17a	Flowchart For READ/WRITE Processing (mainstream)	3-62
3-17b	Subroutines to Handle Device Transfers During READ/WRITE	3-63

Illustrations (continued)

Figure	Title	Page
3-18	Unique Read Processing Sequences	3-64
3-19	Unique Write Processing Sequences	3-65
3-20	Flowchart For Dataset Close Routine (CLS)	3-66
3-21	The BLOCK Block	3-67
3-22	The Special Functions Block	3-68
3-23	Driver Interface Table	3-69
4-1	Linked File Format	4-70
4-2	Contiguous File Format	4-70
4-3	Master File Directory Block #1	4-71
4-4	Master File Directory Block #2	4-71
4-5	User File Directory Block	4-72
4-6	Bit Map Segment Format	4-73
4-7	Protection Code Format	4-73
4-8	Non-System Disk Format	4-74
4-9	System Disk Format	4-74
4-10	Possible Linked File on DECtape	4-75
4-11	DECtape Format	4-76
4-12	Monitor File Management Modules	4-77
4-13	Potential Stack State - Internal File Management Subroutine Call	4-78
4-14	Use of Swap Buffer in File-Handling Operations (First-Level Routine)	4-79
4-15	User File Block	4-80
4-16	File Information Block (FIB)	4-80
4-17	FIB Linkage to Bit Maps	4-81
5-1	Program Load Image as Produced by Linker (Link-11)	5-30
5-2a	Program Load Operations (Phase I, Program Load)	5-31
5-2b	Program Load Operations (Phase II, Monitor Module Load)	5-32
5-3	Operations of the General Utilities Routine	5-33
5-4	Conversion Routine Operations	5-34
5-5	Command String Input	5-35
5-6	Command String Syntax Rules	5-35
5-7	Command String Input Buffer	5-36
5-8	Command Buffer For CSI	5-36
5-9a	Command String Syntax Analysis (Mainstream)	5-37
5-9b	Command String Syntax Analysis (Subsidiary Routines)	5-38
5-10	CSI Command Block	5-39
5-11	Switch Data In the User Link Block	5-39
5-12	Command String Decoding	5-40
5-13	Exit Operations	5-41
6-1	Use of the Keyboard Swap Buffer in Command Processing	6-90
6-2	Stack States During Command Processing	6-91
6-3	Keyboard Command Buffers	6-92
6-4	Keyboard Listener Operations (RMON5)	6-93
6-5	Special Keyboard Driver Operations	6-94
6-6	Keyboard Interpreter Operations	6-95
6-7	Date Command Processing	6-96

Illustrations (continued)

Figure	Title	Page
6-8	Save Command Processing	6-97
6-9	Begin/Restart Commands Processing	6-98
6-10	Time Command Processing	6-99
6-11	Modify Command Processing	6-100
6-12a	Assign Command Processing (Overlay #1 & Mainstream,	6-101
	Overlay #2)	6-101
6-12b	Assign Command Processing (Overlay #3 & Subroutines, Overlay #2)	6-102
6-13	Stack State During 'Assign' Processing	6-103
6-14	Example of Line-Printer Dump	6-104
6-15	Dump Command Processing	6-105
6-16	EOM Adjustment by TMON	6-106
6-17	Transient Monitor, General Processing	6-107
6-18a	Run/Get Command Processing (Load File Check)	6-108
6-18b	Run/Get Command Processing (Preload)	6-109
7-1	Stack State On Entry to the Error Diagnostic Print Routine	7-7
8-1	Modules Present in DOS-11 Monitor (V4A)	8-23
9-1	Object Module Format	9-20
9-2	Blocking of the GSD	9-21
9-3	Contents of GSD	9-22
9-4	Format of a Text Block	9-23
9-5	Format of a Relocating Directory	9-24
9-6	Object Module Library Format	9-25
9-7	Blocking of Object Module Library	9-26
9-8	Load Module Library Format	9-27
9-9	Blocking of Load Module Library	9-28

CHAPTER 1

I N T R O D U C T I O N

Like other bulk media, generally, a disk is convenient for the storage and retrieval of system software such as Editors, Assemblers, Compilers, etc.; which assist the computer user towards a more rapid development of his application's programs and thereafter of those programs themselves and their data. More significantly, because of its relatively fast random access capabilities, a disk can be treated as a virtual extension of available core memory, thereby enabling the provision of a wide range of run-time services which can ease the burden of all programs, even in the smaller configurations.

The PDP-11 Disk Operating System (DOS) is designed to take advantage of these benefits in the single-user disk-based environment. Thus it consists basically of a Monitor which supplies the services mentioned and in particular supports a comprehensive set of program development software. In the Programmer's Handbook (DEC-11-MWDC-B) the Monitor is described for the user who wishes merely to obtain its services for his own purposes and to control the operation of his system. The usage of its associated programs, listed below, is explained in the relevant Programming Manual shown:

EDIT-11	Text Editor	DEC-11-EEDA-D
PAL-11R	Assembler	DEC-11-ASDC-D
LINK-11	Linker & LIBR-11 Librarian	DEC-11-ZLDC-D
ODT-11R	Debuggin Aid	DEC-11-OODA-D
PIP-11	File Utilities Package	DEC-11-PIDB-D
FORTRAN	FORTRAN IV Compiler	DEC-11-KFDB-D

This manual supplements the information in the Monitor Programmer's Handbook by presenting a more detailed discussion of the facilities provided by the Monitor and the methods used to achieve these. In general, the background concepts of each of its aspects are considered followed by an individual examination of the routines satisfying those concepts. The procedures by which the Monitor is prepared for usage are also illustrated to assist those who need to develop its amenities further, by modification, addition, or extension. As far as possible the manual is arranged to enable immediate reference to a particular topic; this then contains pointers to supporting information. This also should allow revisions to be readily included by replacement of the appropriate section.

The purpose of this Chapter especially is to provide a general overview of the Monitor. Section 1.1 discusses its principal aims and its organization to meet these aims is described in Section 1.2. Section 1.3 then introduces some established conventions.

1.1 The Function of the DOS Monitor

The DOS Monitor exists for two main reasons:

a. To make available to a running program a whole series of general utility routines to simplify its usage of the PDP-11 as a system, in particular for the handling of its I/O requirements.

b. To enable the user at the console keyboard to maintain control of the operations of his system.

As noted, the principal area of service to a program concerns I/O management. Chapter 3 will show that the user is given the opportunity to request as much or as little assistance as he needs. In many cases he need not consider the devices he will actually use when the program is under execution. If he wishes to utilize the facilities offered by bulk media for the storage of many independent sets of data or to share these facilities with other users, he is able to effect this within the framework of a Monitor-controlled file structure to be discussed in Chapter 4.

I/O aside, a running program may often need information about the particular system in which it is currently working or may require general help in manipulating its data. Moreover the processes by which it is loaded and unloaded are essential services. Chapter 5 covers the provision of these and other services.

In the normal way, the console operator needs to be able to dictate the sequence in which programs are executed and to access them accordingly. On occasion, he may be obliged to supply intermediate set-up information. Moreover it is possible, even in the best regulated system, that he must intervene to interrupt the normal sequence when adverse conditions arise. The method by which he can exercise these controls from the keyboard at all times is described in Chapter 6, and Chapter 7 shows how he may be kept informed of those adverse conditions.

1.2 Monitor Organization

The major feature of the Monitor's strategy to meet its commitments as outlined in the previous section is the fact that it has been setup as a series of normally independent modules, each of which satisfies one specific function. This modularity has the following advantages:

1. Ease of development
2. Flexibility in usage
3. Extensibility

The first of these implies that the Monitor has been brought to its current state by preparing each module separately, including any necessary modification (and in some cases complete replacement) and check-out, before finally incorporating it into the system. The third advantage effectively says the same thing for future development, either as a general provision or to meet a specific user's requirements (provided of course he himself does this).

The matter of flexibility is particularly relevant because it is this aspect which really permits the Monitor to gain most benefit from the availability of the disk within the system, as noted in the introduction to this chapter. Basically certain modules, which are discussed in Chapter 2, must remain within the computer memory at all times because they control the system generally. Figure 1-1 illustrates their normal location. The remaining modules which perform ephemeral tasks need only reside upon the disk which supports the system (called elsewhere the "system device") if the available computer memory is too limited. They can then be loaded when required and can later be removed when their purpose has been served. To this end, a Library area is reserved upon the systemdevice as shown in Figure 1-2. Its format is designed to allow simple set-up (see Section 8.2) and ready access (see Sections 2.1.2 and 2.1.4).

However, despite the fact that such loading can be accomplished fairly quickly from the disk, it still takes finite time and the user who has memory to spare may prefer to avoid the wastage. Modularity again helps, as follows (see Figure 13):

a. If a particular module is required so frequently by all users of a system, it can be added to those already included in the permanently resident part of the Monitor at system generation (currently by linking - see Section 8.1.2)

b. A module which is particularly appropriate to one application can be loaded with the program concerned just for the duration of a run. The user in this case indicates his requirement as a global reference in the program (see Programmer's Handbook for details) and, as shown in Section 5.1, the Program Loader obliges if the module is not already permanently resident. (No linking occurs, hence the same program can be run effectively in any system if the overall core capacity permits).

Thus modular flexibilty gives the user the choice between residency or swapping from the disk. In the latter case, it should be noted, the temporarily loaded routine occupies a reserved area within the Monitor (see Section 2.3.1) - it does not require that part of a program be swapped out first. This avoids two accesses and means that no restrictions need be placed upon the activities of a program as might be the case if part of its area were potentially removable.

A second feature of the Monitor is dynamic buffer allocation, which is most closely related to the I/O operations as discussed in Chapter 3. It will be shown that until a running program actually requires I/O services, no memory is allocated for the purpose but thereafter appropriate buffers are dynamically set-up or released from the then free space (see Section 2.4.2). Hence, the run-time core image is as illustrated in figure 1-4. (A similar layout is also included in the Programmer's Handbook.) In this respect, some of these buffers may be used to load the drivers for the devices providing I/O. These are stored within the system-device Library with other modules. Unlike the latter, however, the drivers cannot be loaded for the duration of a program for reasons given in Section 2.1.3. Nevertheless, they may be linked into the permanently resident Monitor, as above.

1.3 Monitor Conventions

The Monitor's modularity imposes a need for certain conventions in order to permit the necessary interface between its various components and the user program. This Section describes the more common ones.

1.3.1 Calling Monitor Modules

Both at the user level and internally within the Monitor, the standard method by which a Monitor module is accessed is through the EMT instruction (1). This has been chosen because its lower byte is not considered in the hardware decoding operation; it can therefore be used for a software code to identify the module required and avoids the use of a second word, e.g., (as shown in the Programmer's Handbook) a call for an I/O .READ is EMT 4 (or 104004). It also ensures that the necessary handler for this instruction has the opportunity to control all communication paths throughout the system, a particular advantage in the swapping situation. Hence Figure 1-5 illustrates the Monitor structure with this in mind.

In many cases, the call to a module requires a subsidiary transmission of data. At the user level, this is effected (as described in the Programmer's Handbook) by means of the processor stack in order to allow the program complete freedom in its use of registers. However, to keep the number of essential pushes to a minimum one of the arguments may be a pointer to a list of additional items. (This practice may mean that the user need exercise some caution if programming "re-entrantly" since any such list must be impure unless it also is developed on the stack). Inside the Monitor, data may be passed either on the stack or in registers. The module descriptions in the following chapters show the expected calling process.

1.3.2 Return State

The general principle applied is that unless data is being returned the processor stack is cleared of the necessary items pushed by the call. If data is returned, it is then the responsibility of the calling routine, be it the user program or another Monitor module, to remove such data as soon as it has served its purpose. The register state must remain as on entry. (Internally, this may not apply if the Registers are again used to return data : this is also indi-

----
1. The two principal exceptions to this general rule are:

   a. Requests for device driver services (see Section 3.3).

   b. Calls for Error Diagnostic Print (see Chapter 7)

cated in the module descriptions if appropriate.)

1.3.3 Naming Conventions

For search access to the internal modules, the following conventions for names have been adopted:

1. Potentially swappable processing routines - for these, three alphanumeric characters which can be radix 50 packed into one word. (see Section 5.3) They are used both as Assembler .TITLE designation for Library searching (see Section 8.2.1) and as global reference for linking (see Section 8.1.2). As noted in the Programmer's Handbook, users should avoid these names in their own globals.

2. Devices are normally identified by a two letter code, as listed in the Programmer's Handbook. A third letter may be added to distinguish between different controllers for the same device e.g. DTA and DTB might identify two TC11's. The code may be followed by an octal byte value to specify a particular unit.

1.3.4 Definitions

The Programmer's Handbook contains a list of the commonly used terms and a comprehensive glossary. This manual complies with that terminology.

In the figures which conclude this document, entire core is illustrated with word 0 at the bottom and upper words at the top; whereas, words in tables are illustrated with the smallest word at the top.

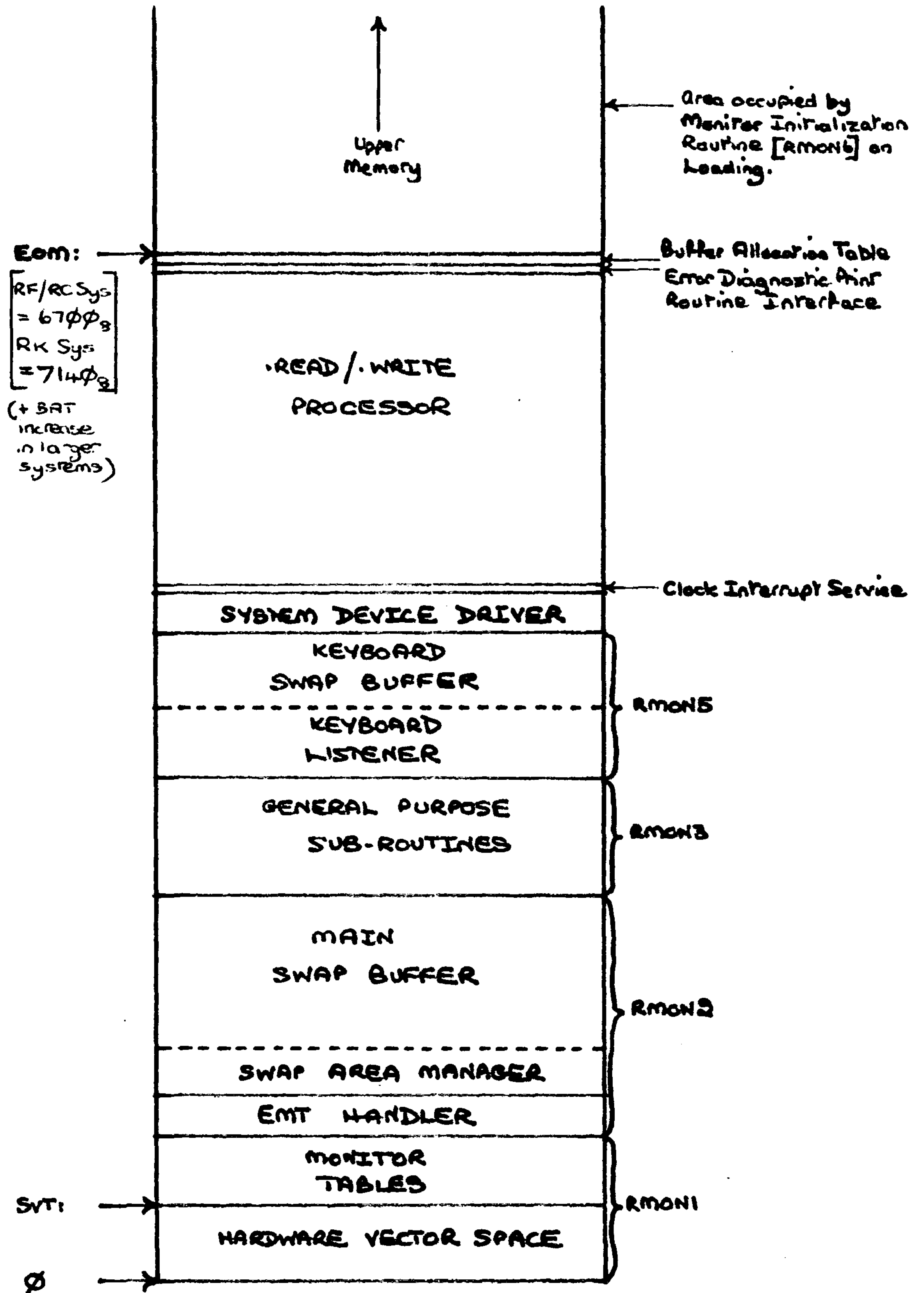


Figure: 1-1: THE BASIC RESIDENT MONITOR in MEMORY

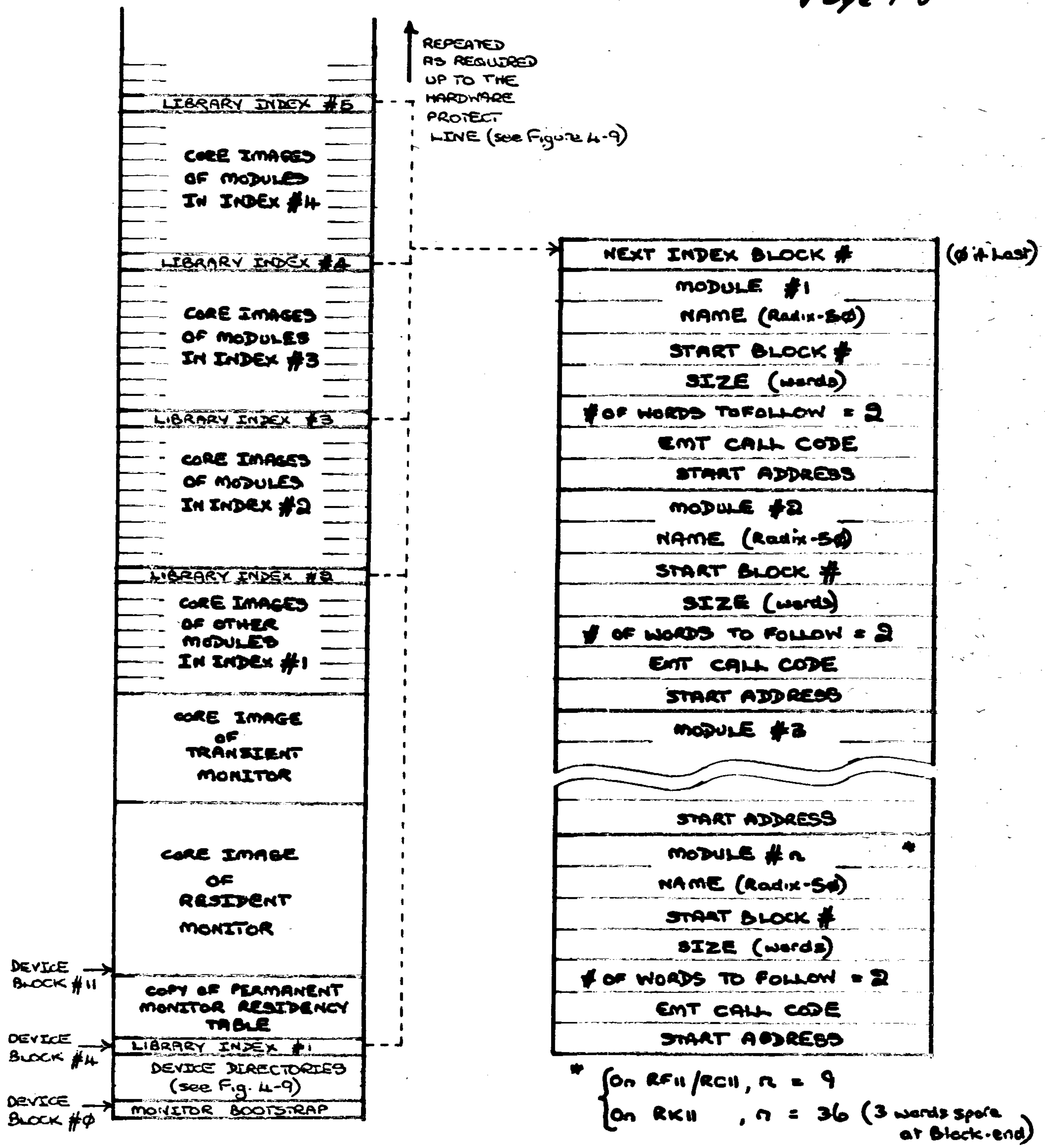


Figure 1-2: MONITOR LIBRARY on the SYSTEM DEVICE.

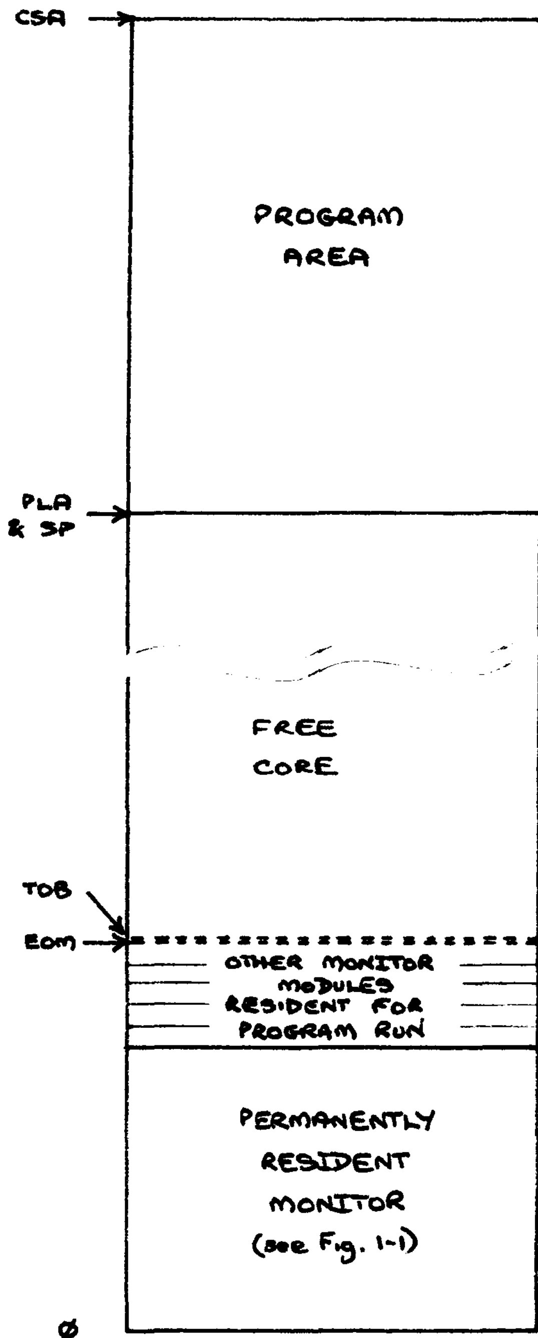


Figure 1-3: Memory after Program Load

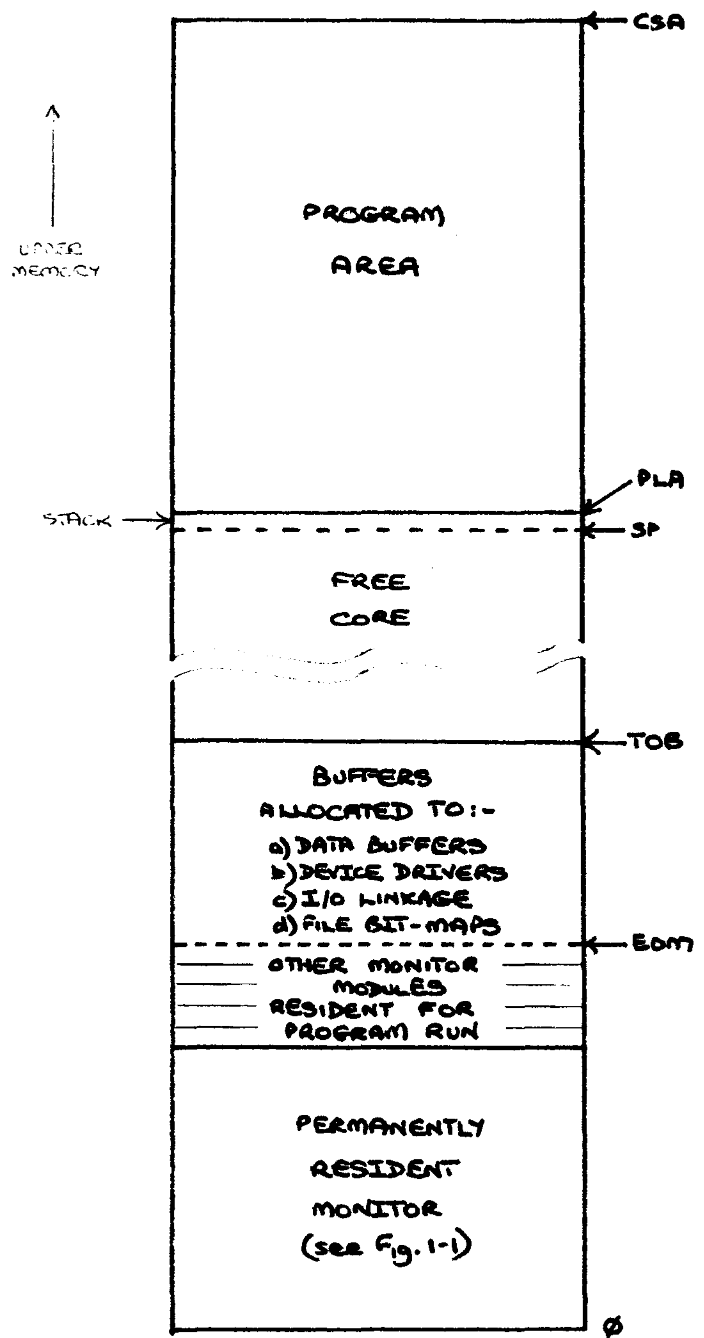


Figure 1-4: Memory during Program Run.

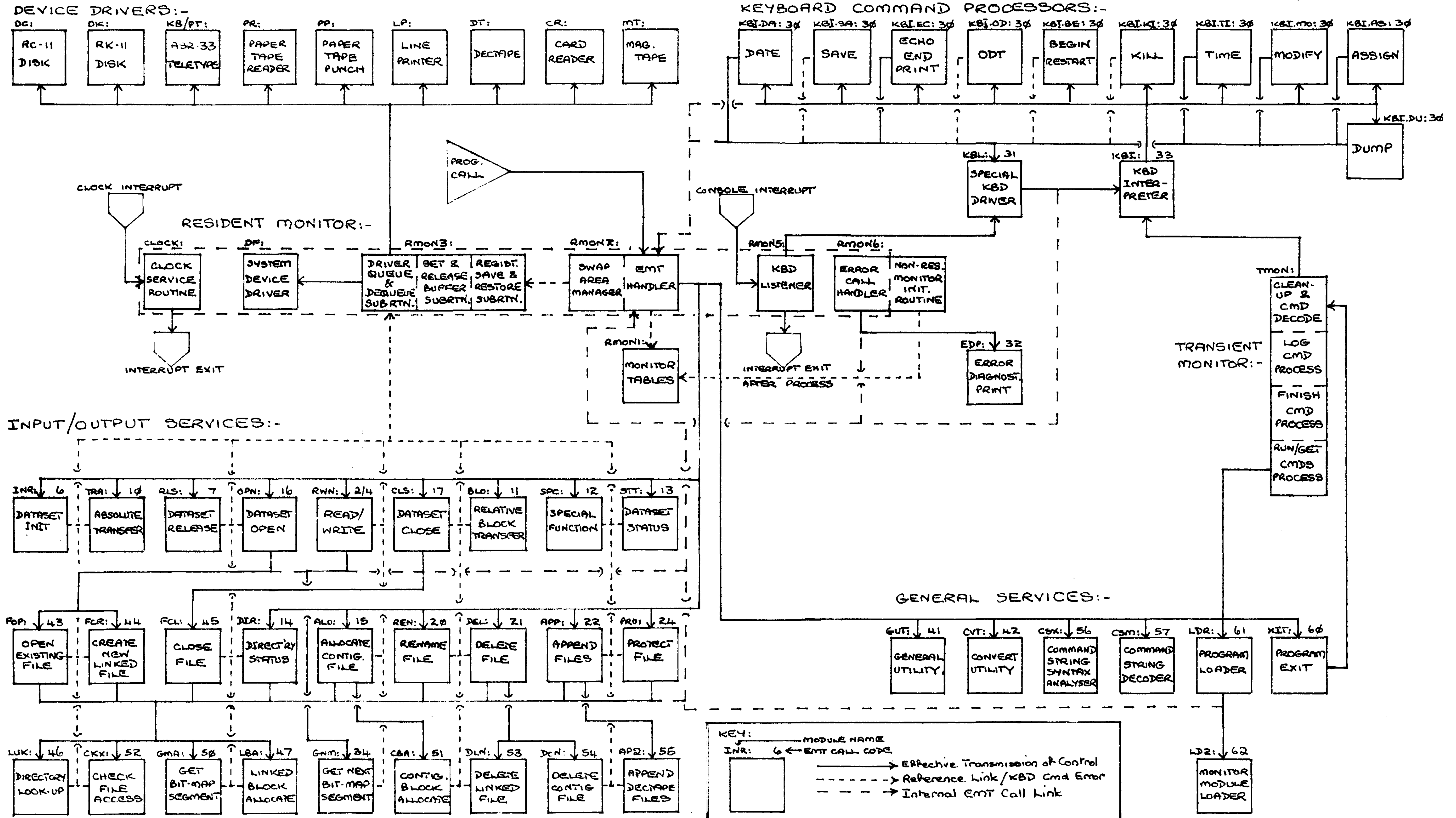


Figure 1-5

CHAPTER 2

RESIDENT MONITOR

It was shown in Chapter 1 that, in general, the user can himself determine the way in which he will use the modules forming the DOS Monitor either to save time by having them resident in memory or to save core by leaving them to be brought from the system-device only when required. However for certain of the modules there is no choice: they must be resident at all times for one of the following reasons:

a. They control the system generally including the loading of other Monitor routines from the system-device.

b. They are called with sufficient frequency that to bring them from the system device every time they are needed would be unrealistic.

The purpose of this chapter is to discuss the principal modules in this category. Section 2.1 contains descriptions of the tables used by the Monitor to maintain adequate control of the system and shows how they are set up initially. Section 2.2 is concerned with the real heart of the Monitor, the EMT handler, and Section 2.3 looks at the techniques for bringing modules from the system-device by the Swap Area Manager. Some general-purpose subroutines are covered in section 2.4 and finally Section 2.5 outlines the clock control routine.

Two other resident modules are intentionally omitted here in order that they may be discussed in context:

1. The system-device driver - see Section 3.3.

2. The minimal console typewriter driver needed to service keyboard command inputs - see Section 6.1.

Presently the .READ/.WRITE processor must also be resident. by reason of its size, it cannot be brought into the Swap Buffer in the usual manner. Nevertheless it is described with the other I/O processors in Section 3.2.

The layout of the modules within the permanently resident Monitor area is illustrated in figure 2-1.

2.1 Monitor Tables

The object of this Section is to introduce the various tables which are set up within the resident Monitor area, for intercommunication between all Monitor routines. The tables themselves are first discussed. This is followed by a description of the initialization process which is mainly provided to ensure that the tables are correctly set-up for use when the Monitor is loaded.

It should be noted at this point that the Monitor has been designed to be independent of the position it occupies in memory. This means that only those routines which form the basic resident Monitor can reference each other directly since the appropriate connections can be established during linkage by LINK-11. All other routines which can potentially be swapped-in only when required must use indirect methods.

To provide the most straightforward access to frequently required areas in the resident Monitor, DOS therefore utilizes the vector space in locations 40-57, generally reserved in all PDP-11 systems for system software use. These are in fact the only locations addressed absolutely within the Monitor(1) (device drivers controlling the external page of course excepted). Figure 2-1 shows their present usage. In general, a reference through them requires a level of indirection beyond that provided within the format of a single PDP-11 instruction. Hence the common technique for entry, say, into a general purpose subroutine with its start-point in one of the vectors is via the top of the processor stack as:

```
        MOV @#44,-(SP)      ;GET START ADDRESS OF
                            ;SUBR. 'S.RSAV'
        JSR PC,@(SP)+       ;ENTER ROUTINE & CLEAR STACK
```

2.1.1 System Vector Table (RMON1)

The System Vector Table (SVT) provides a common area for the storage of information on the state of the system at any time. In particular, it contains pointers to other parts of

----

1. The file structure routines currently assume the System Vector Table (SVT) starts at absolute location 400. This is a restriction that will be fixed in later releases of DOS.

the resident Monitor which are not referenced frequently enough to warrant the more immediate access discussed above or to areas within a running user program. It does not include detail for other routines which normally exist only within the Monitor Library stored on the system-device. These are covered by separate tables which will be discussed in the following sections.

The table occupies the first part of the resident Monitor area (the only real reason for this being the fact that its source also contains the contents of the fixed vectors in 40-57). Normally it starts immediately above the reserved interrupt vector space at location 400. Nevertheless, since it is always referenced through location 40, this address can be changed by reassembly, particularly if an installation needs extra vectors for special devices (because of the file structures' restriction indicated above, the SVT address of V004A version of DOS must not be changed) (1).

The items currently in the SVT are listed in figure 2-2. In the main, these are self-explanatory; however additional notes have been included where necessary. The table can be expanded as required by the current state of Monitor development but only by addition at the end. The presently defined internal structure cannot be changed since all Monitor routines depend upon fixed relative positions within the SVT in order to reference its content.

The SVT is basically established when the resident Monitor is first linked throughout LINK-11 as described in Section 8.1.2. The table source has been set up to include, in particular, the necessary global references to other Monitor addresses to accomplish this. Other table entries which reflect the constant state of the configuration in use are set up during the initialization process which follows Monitor loading (see Section 2.1.4). The variable entries, thereafter, are filled in or removed appropriately by several Monitor routines and these will be noted at the relevent point throughout the remainder of this Manual. In order that the user program may also reference particular entries deemed desirable, the General Utilities request based on EMT 41 is provided as the recommended means (see Section 5.2)

----

1. This facility may also allow a user who wishes to reserve an area of memory outside Monitor control to do so. However this usage is not considered a normal DOS provision. Thus correct Monitor operation is not guaranteed if its bottom is forced above 16K, the point at which positive addresses become negative.

2.1.2 Monitor Residency Table (RMON1)

The Monitor Residency Table (MRT) supplies two types of information, mainly for the EMT handler (see Section 2.2):

1. It shows which Monitor routines are resident in memory, either permanently or for the duration of a program run, and where they are loaded currently.

2. It acts as a directory to the remaining routines as stored within the Monitor Library on the system-device, to enable immediate access when one of these routines must be brought into a Swap Buffer.

The table is a set of one-word entries, each corresponding to an EMT code, and is ordered in the sequence of those codes starting at 0. Thus, for example, since the code for .INIT call is 6, the table entry for the .INIT routine is at MRT+14. The length of the table within any Monitor system naturally depends upon the range of the currently assigned codes, perhaps with spares, up to a maximum of 256 words.

The length of the MRT can be determined by subtracting the starting address of the MRT (SVT+46) from the starting address of the DDL (SVT+50) which immediately follows it.

The format of each word in the table shows the current location of the Monitor routine it represents, using the fact that for a valid PDP-11 address for execution access, bit 0 must be 0, i.e. a word boundary. The possible formats are illustrated at figure 2-3. It should be noted that two bits can sufficiently define the maximum number of 64-word segments (or fixed-head disk blocks) for any Monitor routine since the largest swap area available is 256 words (see Section 2.3). Moreover since the Monitor Library always comes first on the system device, it is not expected that any Monitor routine will be stored beyond block #17777.

The table-state needed for a user program run is reached in three stages:

1. The MRT source is set up to contain the global name of each existing Monitor routine in the appropriate position. During linkage of the permanently resident Monitor Section with LINK-11 (see Section 8.1.2) the core address for each routine included is automatically set correctly into the table. LINK-11 also zeroes the remaining entry words since their global references remain undefined.

2. Whenever a complete Monitor boot occurs, either from cold start via the ROM loader or following a

console FINISH command, the initialization routine (see Section 2.1.4) searches the Monitor Library (MONLIB) on the system device and resets all 0 entries in the MRT to the relevant disk information (format b, Figure 2-3) if the corresponding routine is found or with 1 otherwise. At this time, the MRT is in its permanent state. A copy is stored, within the Monitor Library on the system device for later use (see below).(1)

3. If the user, by a global reference in a program, indicates that a routine normally stored only on the system-device is to become resident while that program is running, the program loader (see Section 5.1) is so informed by LINK-11. The loader uses the disk information stored in the MRT to bring the routine into core and resets its MRT entry to the appropriate core address.

The MRT remains in the load-state throughout the program run; it cannot be changed for routines brought into a Swap Buffer, because the disk information relevant to them must remain available for several interleaved calls, e.g. .INIT followed by .OPEN, then back to .INIT again. The Swap Area Manager therefore uses other methods to control the "residency" of these routines (see Section 2.3). When the program is finally removed from core by .EXIT from the program or on a console KILL command, the MRT copy on the system device is read back into memory to restore the table to its permanent state. (see Section 6.5)

2.1.3 Device Driver List (RMON1)

Basically, the Device Driver List (DDL) provides the same information as the MRT concerning the current memory or system-device locations of device drivers. However, unlike the other Monitor routines, the drivers may be loaded into or removed from memory in accordance with the dynamic requirements of a running program and they do not use the Swap Buffer. Since they must be readily available to service program interrupts whenever their device is in use, they oc-

----

1. As stated, this write-out occurs under normal running conditions. Hence at present the Monitor Library on the system-device must not be write-protected.

cupy extra buffers from free core. Hence each entry must be longer to contain additional information.

The resulting format of the DDL is illustrated at figure 2-4. The significance of each item is as follows:

1. Device name - is the radix-50 form for the alphanumeric code assigned to each device. The only way an entry in the list can be found is by a search. This item therefore enables its identification.

2. Core load address - contains the start of the device driver when in core. Otherwise 0 indicates the driver's current non-residency.

3. Interrupt vector address - is the start address of the two-word vector assigned to the device within memory locations 0-377 (or as otherwise provided) and is needed to permit the linkage of the driver's interrupt service routine to the vector when loaded. (See Section 3.2.1) it is held within the DDL rather than the driver itself, because the user can physically reassign devices to different vectors. This obviates reassembly of the driver and also allows in-core modification, even though the driver at the time is available only in the external Monitor Library.

4. System-device start block - gives the actual device address for the driver within the Monitor Library. As for the MRT, no driver is expected to start beyond block #1777(8).

5. # of 16-word blocks - enables determination of the size of the driver for claiming and releasing the buffer it occupies while in core and for specifying of # of words to be read, in order to perform the necessary load (see Section 3.2.1). The six-bit capacity of the field allows a maximum driver length of 1K, which is more than ample in most cases.

6. DDL end - replaces the external address data in the system-device entry in the table, since the latter information is irrelevant when, by its purpose, the driver for such device can never leave memory. The provision of this item instead allows the DDL to be of variable size and hence contain only the drivers needed to support the particular configuration being used. However it is essential that the system device entry is alway the first in the DDL as shown in the diagram.

The DDL is established in the same way as was described for the MRT in the previous Section, with the following exceptions:

a. A driver automatically becomes resident for as long as its current usage is required; the user can gain nothing by extending its presence in memory for the duration of his program, (other than an occasional system-device access). Hence he is given no way of specifying this requirement.

b. Because each item in an entry serves a single purpose, there is no need for a re-initializing back-up copy of the DDL on the system-device.

As indicated above, whenever the program initializes or releases a dataset, the DDL is searched for the device associated with that dataset. Provided that such device exists within the system, the entry for the core address and the vector, through its stored address, are both set to reflect the presence or absence of the driver in memory Should the program be restarted at any time, perhaps following a failure, the DDL is restored to its initial state, thus effectively removing normally non-resident drivers from core (see Section 6.4)

2.1.4 Table Initialization (RMON6)

The permanently resident Section of the Monitor is loaded into its memory area either by a cold start through the ROM bootstrap as described in the Progammer's Handbook or following a console FINISH command (in order to prevent possible transmission of Monitor corruption between users). It is accompanied by a routine which performs the function of establishing the correct Monitor state for normal usage, especially with regard to the tables described in the previous sections. This routine, occupying core above the normal end of the resident Monitor, is entered immediately after loading. After serving its purpose it is subsequently over-written, leaving only a small Section which interfaces to the system error diagnostic print module (see Chapter 7) and a Buffer Allocation Table used to control operations within free core (see Section 2.3) (hence the object module containing it must come last in the resident Monitor linking process - see Section 8.1.2)

The tasks carried out by this special routine are as follows:

1. Determination of core size - this is accomplished by addressing core starting at 8K and then in steps of 4K until an error trap through location 4 indicates the address used is no longer legal for that particular PDP-11. The end address of the last valid 4K segment is stored as the top of core in the SVT.

2. Provision of Buffer Allocation Table - as will be shown in Section 2.3 free core is controlled by a bit map representation. Currently the size of this map provides for the potential use of half of the available memory as buffer space. The routine therefore prepares the appropriate map by clearing the requisite number of words at the end of the Monitor area. These are followed by two extra words reserved for Device Assignment Table linkage (see Section 3.1.2.2) The map then becomes part of the resident Monitor, so data concerning the map and the consequent new Monitor end are entered into the SVT.

3. Initialization of processor trap vectors - since the Monitor itself uses EMT & IOT, the routine stores the appropriate addresses and status values in their vectors in locations 30 and 20. The remaining vectors are set to cause error messages if trapping occurs, the status value in each case providing an identification code (within the Condition Code bits)

4. Initialization of device interrupt vectors - by searching the DDL, the routine determines which drivers were loaded with the resident system. It extracts information stored in the interface tables of these drivers (see Section 3.3) in order to set up the corresponding device interrupt vectors. As in step 3 all other vectors are initialized to cause error message output to protect against spurious interrupts before a non-resident driver, if such exists, is available.

5. MRT & DDL preparation - as indicated in the previous sections, the routine establishes the permanent state of these tables and stores a restoration copy of the MRT in the external Monitor Library. It also, from its own internal reference sheet, enters the EMT code used to call each Monitor module into the Library index on the system-device (see Chapter 1). This enables LINK-11 later to identify to the program loader those modules for which a user has specified program run-time residence (see Section 5.1). It should be noted that the whole Library is

searched: should there be more than one version of a Monitor module or driver, the last one seen will be catalogued in the appropriate table. Thus it is possible to make temporary changes to the Library merely by adding a new version at the end during system loading. (see Section 8.2.2)

6. Clock-initialization - if a clock-service routine is contained within the resident Monitor and a line-clock is determined to be included in the configuration (because no error occurs if it is addressed), the clock's interrupt is enabled to set it in motion.

On completion of these tasks, the initializing routine links the resident Keyboard Listener described in Section 6.3.1 to the console interrupt vectors, prints out a Monitor identification message and exits to await operator instructions (using .XIT - see Section 5.5)

2.2 EMT Handler (RMON2)

It was shown in chapter 1 that all calls for Monitor assistance are by way of an EMT instruction with its junior byte coded to indicate the particular service required. It is the primary function of the EMT Handler to decode the call and make the necessary connection to the routine providing the service. The Handler also ensures that I/O requests in particular do not proceed until it is currently safe for them to do so and contains its own routine to effect any consequently necessary wait.

2.2.1 Description of the Handler

An outline of the Handler's processes is given in figure 2-5. This shows that the first step is a move of registers 0-5 onto the processor stack to preserve them for the user program. This serves two purposes:

1. All the service routines can thereafter use the registers freely. However it is then their responsibility to replace the original contents before any exit to the user program.

2. It will be shown in Section 2.4.1 that the subroutine entered to effect the save also checks the state of the processor stack to protect the Monitor itself. Thus this check is imposed as frequently as the Monitor is called.

The Handler then verifies that the EMT code used lies within the established range (0-77). An invalid code should never occur if the calling program is working correctly. If it does, it is therefore treated as a fatal error: the program is stopped and an appropriate error message is output at the console (F002).

The Handler's next action depends upon the nature of the request in each case. Under DOS, in fact, the EMT codes are assigned on the following basis:

1. Codes from 0 through 27 signify requests for I/O services

2. All other requests have codes in the range 30-77

No further checking is needed for requests in the second category. The Handler therefore extracts the entry, corresponding to the code within the MRT described in Section 2.2.2. It was shown there that if bit 0 of the entry is 0, the entry itself is the core start address of an already resident routine. In this case, the Handler uses the entry to dispatch immediately to the routine. On the other hand, a 1 in bit 0 of the entry means that the routine must be brought into memory before it can be used. The Handler therefore calls the Swap Area Manager to effect this (see section 2.3) and passes on the MRT entry on the top of the processor stack to identify the routine required.

As noted earlier, I/O requests are not allowed to continue while it is unsafe. In their case, the Handler checks for two possible situations as follows:

1. The Programmer's Handbook shows that in all I/O operations, the user program must supply a Link-block for each dataset to be used and must pass its address as one of the parameters at each call. Once a dataset has been correctly initialized by a .INIT request (see Section 3.2.1.1), a link-word in the block, i.e. the one actually addressed, is set to a non-0 value. Using the given parameter as means of access, the EMT Handler examines this word and rejects all calls other than .INIT if it contains 0. This again implies program failure and results in a fatal error message (F000).

2. It will be shown in Section 3.1.2.3 that the value set into the link-word by .INIT is the address of a control block for the dataset within the Monitor area known as the DDB. The addressed word in the

DDB is a flag which is set to 0 only when there is no other I/O operation currently underway. If by looking at this word, the EMT Handler sees that the dataset is already busy, it sets up a loop sequence within itself until a later check shows otherwise. This looping does not happen, though, if the actual call is .WAITR. In this case, the processor stack is adjusted such that after restoration of the user program Registers and removal of the call parameters, an exit is taken at the busy return address given as one of these parameters (see illustration at figure 2-6)

If neither situation is encountered (or after the second has disappeared), the EMT Handler claims the dataset for the new request (unless it is .WAIT or .WAITR) by storing the address of the call in the DDB busy flag. It then uses the same process to dispatch to the relevant service routine as that described for non-I/O requests. At this point, of course, the purpose of .WAIT or. WAITR has already been satisfied. Hence no further processing is needed and the EMT Handler uses a similar sequence to that indicated for .WAITR in the busy state to take an immediate completion return to the user program.

2.2.2 General Comments

From the description given in the previous Section, it can be seen that the EMT Handler does not attempt to verify the validity of the parameters passed by the program at each call. In particular, it assumes that the link-block given with I/O requests does in fact contain a true DDB address. Thus it depends greatly upon the user respecting the integrity of data set into his program area by the Monitor. A check in this case could be simply accomplished from data in the DDB's held by the Monitor. However this would involve a linear search which could become lengthy. Since in the single-user system currently offered by DOS, the user can only corrupt himself, the check has therefore been omitted by intention. In the general case, of course, the EMT Handler has insufficient information to carry out the requisite checks. It is thus the responsibility of the called service module to take relevant action.

To provide each routine with as immediate an access as possible to the data it needs, the EMT Handler always passes the following appropriate Register contents:

R1 = stack address of the last call parameter pushed by the program
R2 = address of the relevent EMT call in the user program
R5 = address of processor Status Register (or -2)

In addition, for I/O calls other than .INIT, R0 remains at the address of the DDB; for all other calls, it is cleared.

The priority level at which the Handler operates merits explanation. In general, it uses the level of the calling program. However if this should be 4 or above, hardware interrupts needed perhaps to terminate an enforced .WAIT as described earlier could be locked out. On the other hand, at level 4, conflicts might arise if the user intervened at the console. Thus the general level is kept at 4 if the program is higher. On entry nevertheless, priority level 7 is set in order to ensure that the checking of the stack-state can be effected without interruption, but is dropped as soon as possible thereafter (taking approximately 70 usecs). Level 7 is again used while the Handler checks the busy state of a dataset. The check could be corrupted, if an interrupt were able to claim a dataset for another operation after conditions were already set to indicate its being available for the interrupted request. (in the present DOS of course, this cannot happen; however it was provided against a possible future expansion for real-time use.)

2.3 Swap Area Management (RMON2)

Two buffer areas are reserved within the resident Monitor to accommodate Monitor modules which are brought into core only upon program demand. It is the responsibility of a necessarily resident Swap Area Manager (SAM), to control the usage of these buffers. When asked by the EMT Handler to load a non-resident module, SAM checks the occupancy of the appropriate buffer: if the called routine is already loaded from a previous request and provided that it is idle or, if not, that it is reentrant, SAM passes control to it immediately. Otherwise, SAM loads the called routine from the system-device if and when the current occupant has completed its function. Until the routine is then able to take over, SAM keeps control within the Monitor by returning to the EMT Handler to loop and wait.

2.3.1 Swap Buffers

The format of a Swap Buffer has been designed to enable reasonably efficient machine utilization within the constraints of a single-user system. The main considerations are reflected in the items included in a preamble to the Swap Buffer as illustrated in figure 2-7:

1. Occupant - this basically allows SAM to re-use a copy of a called Monitor routine already loaded into the Swap Buffer to satisfy a previous request, rather than force an unnecessary wait for a new copy to be brought from the system-device. In order that the current occupant may always be identified SAM saves the unique system-device information passed on by the EMT Handler from the MRT (see Sections 2.1.2 and 2.2.1).

2. Usage Count - this protects the current occupant from premature "removal" before it has fully met its outstanding commitments. Many of the Monitor modules, especially those providing I/O services, are likely to be unable to complete a request immediately because they must wait upon the performance of some I/O operation. Any routine may in this case return to the user program to allow other processing to proceed instead of keeping the system idle. However the program may then make a new service request requiring use of the Swap Buffer for some other routine. It may even be a request upon the same routine which in many cases can be accepted before the first has been completed. Thus SAM expects to use the first byte of each routine it loads as a counter for the number of times control is passed into the routine and the routine itself counts back the number of times it completes a task (see Section 2.2.3). The buffer is then deemed free only if the counter has returned to its entry setting of 0.

3. Re-entrancy switch - this allows each routine to inform SAM of its re-entrancy capability. Some Monitor routines cannot accept a second request while another is in process, a principal reason being that the service they provide is too complex to be completed within the capacity of the Swap Buffer and they must therefore overlay themselves to cope. SAM therefore only recalls a busy routine if it is safe to do so as indicated by a 0 in the second byte of the routine.

4. Start address - because the first word of each routine is thus allocated as a control area, SAM enters the routine at this point(1).

Two Swap Buffers are in fact included in the resident Monitor area:

1. Monitor Swap Buffer (MSB) = 256_{10} words - this is generally used by all program-service modules.

2. Keyboard Swap Buffer (KSB) = 128_{10} words - this is mainly reserved for console-servicing, in order to allow overlapped processing between a running program and external operator action (see chapter 6). It is also used for emergency operations, such as error diagnostic printing (see chapter 7), especially when the request for these might easily originate within the MSB and obvious conflicts would then occur.

2.3.2 Calling SAM

As noted in Section 2.2.1, SAM is normally entered from the EMT Handler. At this point, certain Registers are set to values indicated in Section 2.2.2; these must be passed unchanged to the called routine as this may depend upon their content (just as if the routine were called directly from the EMT Handler, in other words). The user program Registers are on the stack with system-device information on the routine to be fetched by SAM, as shown in the MRT, above them.

There is a second way in which SAM can be entered: it can be called by its own reserved EMT codes (30 and 40). This special entry is provided mainly for internal Monitor usage as a means whereby a segmented routine can overlay itself in either buffer - hence the two codes, for reasons to be shown shortly. In this case however, the MRT cannot supply the data on the location and size of the overlay on the system device. Hence the calling routine must do so in the same format in R0. When SAM is then called, the data is on top

----

1. When, as indicated in Section 2.1.2, the Program Loader sets the MRT for routines made resident for a program run, it also uses this address rather than the actual start location (see Section 5.1.)

of the stack as saved by the standard Register-save process of the EMT Handler (see Section 2.2.1). It is duplicated on the stack at the special entry and thus conditions become set exactly as for the normal entry described in the previ-ous paragraph.

2.3.3 Description of SAM

Figure 2-8 gives a general outline of SAM's processing oper-ations.

The first main step taken is to check whether the routine requested does in fact exist. As shown in Section 2.1.2, the MRT entry contains a 1 in such cases. If this is seen, SAM uses the EMT Handler error sequence to produce an F002 diagnostic as shown in Section 2.2.1.

SAM must then determine which Swap Buffer is to be used. Routines which are loaded into the subsidiary KSB have in fact been assigned EMT codes in the range 30-37; all other codes imply MSB use. SAM sets a pointer to the appropriate buffer and also one to an internal DDB by which the system device will be controlled (see Section 3.1.2.4). The flag in this DDB is checked in case the driver is currently en-gaged in loading a routine. SAM continues only if this is not the case.

As described in the previous Section, SAM thereafter uses the identification for the occupant stored in front of the buffer to check whether the routine now required is already loaded and if so whether it is idle or re-entrant, on the basis of the setting of the first two bytes of the routine itself. If the correct conditions are seen, SAM transfers control into the routine, having incremented its Usage Count byte to indicate a fresh entry.

If the occupant of the buffer is some other routine, SAM ex-amines its Usage Count. If it is 0 signifying that the rou-tine is no longer needed, SAM stores the system-device in-formation for the called routine as occupant-identifier and then decodes it to set up the internal DDB appropriately (see Section 3.1.2.3). The system-device driver is called to perform the necessary transfer through the Driver Queue Manager, S.CDB (see Section 3.1.2.4).

While the transfer is in progress or if the previous checks show that the system-device driver is already engaged in lo-ading a Swap Buffer or that the required Buffer is not yet free for alternative use, SAM returns to the EMT Handler to loop and wait. This in fact repeats the whole process of decoding the call with the exception of user Register saving

(see Section 2.2.1). This serves three purposes:

a. Registers are reset to the required content for entry into the called routine. Thus SAM does not need to save their content in order to use them whil preparing the transfer.

b. SAM is reentered as originally even in the special case noted in the previous Section. By the subsequently normal procedure, transfer of control to the routine occurs naturally as soon as it has been loaded.

c. If may sometimes happen that as a result of an interrupt SAM is asked to load a different routine into the same buffer, even before the original request is satisfied, e.g. The DECtape driver might wish to print an error message because of hardware failure just when the operator has struck a console key and the Keyboard Listener is being loaded. The general procedure described will in this case honor the second request first. However conditions remain such that the servicing of the interrupted call can be resumed afterwards.

Nevertheless, this method creates a problem for I/O calls. By the time that SAM is first entered, the EMT Handler has already made the dataset busy upon the new call. If SAM merely returns, the EMT Handler will see conditions appropriate to an implied .WAIT as described in Section 2.2.1 and SAM will not regain the necessary control. Hence, together with the slight matter of clearing the data passed by the Handler from the stack, SAM removes the busy-state from the dataset (using the content of R0 to recognize I/O calls - see Section 2.2.2)

2.3.4 System Exit

It can be seen from the previous Section that SAM does not ask to be recalled by the system-device driver when the loading of the required routine has been accomplished. SAM merely requests the load and then decides when it can transfer control to the routine on the basis that the driver is no longer working on the system's behalf and the appropriate Swap Buffer is now occupied by the routine in an idle state.

Instead of the recall, SAM in fact, through a Completion Return Address set into the internal DDB directs the driver to a special sequence. This firstly frees the driver for alternative use through the Driver Dequeue Routine, S.CDQ to be described in Section 3.1.2.4. It then checks that the transfer has been satisfactorily carried out and if not sets up the ultimate fatal error - a System Halt - because it must now be assumed that the device can no longer support the system. If all is in order, the DDB busy-flag is cleared to inform SAM that the transfer is no longer underway. Finally, since the sequence is executed as part of an interrupt from the system-device (see Section 3.3), Registers saved by the driver are restored and an exit to the interrupted routine is taken (by RTI).

The last part of this sequence, Register-restore & exit, obviously complements the trap & Register-save sequence which begins the FMT service as described in Section 2.2. Thus it is the path which all Monitor routines must take to return to the calling program. Furthermore, it was shown in Section 2.3.1 that if one of these routines is in a Swap Buffer, it must also decrement the Usage Count in its first byte whenever it completes a task, in order to free the buffer for other use. It cannot do this while it is still in the buffer, in case this is re-allocated before it exits. The decrement is therefore included in the sequence discussed above (by-passed in SAM's case) and this is identified as the System Exit with an appropriate address set into fixed vector location 42 as the means of general access.(1) For the general case, this is:

```
S.XIT: DECB MSB.       ;ENTRY FOR U/C DECREMENT
       JSR  R5,S.RRES  ;GENERAL ENTRY
       RTI
```

2.3.5 The Swappable Routine.

The previous sections have shown that any routine which is a potential user of a Swap Buffer must be set up within some

----

1. The whole sequence including the device transfer validity check must not be changed - different routines enter it at various points on the basis of a fixed relative to position to the address stored in the vector location.

constraints because of this. Since all the Monitor routines, other than those which normally reside permanently in memory and the device-drivers, come under this heading, they all have common inherent features as a result. This section summarises these features. Its purpose its twofold:

a. It provides an introduction to the modules presently included within the Monitor.

b. It lays out the ground-rules which must be followed during the development of new modules (see also Section 8.3) (1)

Because SAM expects the first word of the Swap Buffer to be Usage Count & Re-entrancy Switches, as discussed in Section 2.3.1, all routines begin with a word in a format 0,0 or -1,0. The junior 0 byte then correctly initializes the Count; the high-byte setting indicates the routine's ability to service several requests together.

It should be noted of course that the protection afforded by the -1 setting of the Re-entrancy Switch does not mean that the routine can then break the rules which the term "re-entrant" normally implies. The switch only prevents SAM from allowing a second request to pass before a first has been completely satisfied. Moreover the fact that a routine is potentially swappable also signifies potential residency in which case there is no SAM to intervene. Hence as far as possible most routines try to observe in particular the need to avoid any form of self-modification, either in executable code or temporary data storage. That they are also postion-independent really goes without saying.

Because the first word of each routine is thus reserved, all executable sequences begin at the second word. As a result, the global reference identifying each routine for possible linkage into the resident Monitor (see Section 2.1.2) is made at this point in the Assembler Source for the routine rather than at the start.

In general, all routines end with a call to the System Exit via location 42. (There are exceptions: see in particular Section 4.3). Thus they meet the requirements for restora-

----

1. Console service modules, for other reasons, must be considered special cases in that some of the comments do not apply. Therefore the reader is referred to chapter 6 for further detail.

tion of user Register contents and release of the Swap Buffer as discussed in the previous Section. Potential residency, however, again raises a problem: a routine must obviously not decrement the Usage Count in the Swap Buffer if it is not actually occupying the buffer. Hence each routine must determine whether in fact this step must be taken. The fact that the count is only incremented in the first place when SAM gives control to the routine offers the solution. If the routine is resident and is accessed directly from the EMT handler as shown in Section 2.2.1, the increment does not occur. Thus a common form of exit as it appears within each routine is illustrated below, assuming that location 42 contains the address S.XIT illustrated in the previous section:

```
        MOV   @#42,R5           ;COLLECT SYSTEM EXIT ADDRESS
        TSTB  START             ;CHECK IF USAGE COUNT SET
        BNE   .+4               ;IF SO ROUTINE IS IN MSB
        CMP   (R5)+,(R5)+       ;OTHERWISE OMIT DECREMENT
                                ;(SECTION 2.3.4)
        JMP   @R5               ;GO TO SYSTEM EXIT
```

The limited size of the Swap Buffer naturally imposes restrictions on each routine. In most cases, this does not matter, since the service required can be simply accomplished within the available capacity, but not in all however. Thus some routines must overlay themselves to get the job done. If such overlaying can be done sequentially, i.e. by each segment completing a specific portion of the sequence and then passing to the next without a need for recall, the special SAM entry described in Section 2.3.2 may be used (see Chapter 6). However this raises other problems of storage within the system-device library - see especially the description of console service modules, which use this technique, in chapter 6. An alternative method, which also covers the non-sequential case, (but nevertheless uses up EMT codes) is discussed under File-structures in chapter 4.

2.4 General Purpose Sub-routines (RMON3)

This Section introduces a set of sub-routines providing services of a general nature for many other Monitor modules, including those in the permanently resident portion. They also are resident: they are moderately short and the frequency at which they are needed makes it impractical to load them from the system-device every time. As noted in Section 2.1, they are accessible by means of the system vectors in locations 44-57.

There are altogether six sub-routines in the set. However, they fall naturally into three pairs since, in each of the three cases, what one sub-routine does, the other undoes. Thus any call to the first must be complemented by a call to the second before any original state is regained. The three pairs are:

1. S.RSAV/S.RRES Register save & restore
2. S.GTB/S.RLB Allocate & release buffers
3. S.CDB/S.CDQ Queue & dequeue driver requests

Of these, only the first two will be described in the following sections. The third pair, being dedicated to I/O, will be discussed in more appropriate context in chapter 3.

2.4.1 Register Save/Restore & Stack Control

In the main, the DOS Monitor has been designed to supply its services to the user while still allowing him to take full advantage of the hardware capabilities of PDP-11. Hence, rather than impose unnecessary restrictions upon his use of Registers, the Monitor guarantees their contents by saving them on the processor stack whenever and for whatever reason control passes from a user's program and restores them before such control is returned. In the interim, the Monitor can then freely use the Registers for its own purposes.

For the general case, all the registers are likely to be needed. Thus the common subroutines S.RSAV and S.RRES with entry through locations 44 & 46 are provided. The first simply "pushes" R5 through R0; the second "pops" them in reverse. Both are called in the same manner as:

```
MOV @#44,-(SP)      ;FOR S.RSAV-OR '46' FOR S.RRES
JSR R5,@(SP)+       ;AUTO-SAVE R5 IN CALL
```

(Direct calling is of course possible from within the normally permanently resident Monitor modules, e.g. JSR R5,S.RSAV or JSR R5,S.RRES).

The processor stack itself is another consideration. Again the user is not asked to refrain from adjusting the stack pointer (SP or R6) to suit his needs. The Monitor merely provides its initial setting appropriately. However, the Monitor itself and any buffers it has established on the user's behalf are in a vulnerable position for corruption by the stack. Hence the Monitor must try to provide some protection similar to that afforded by the PDP-11 hardware for the vector space below location 400.

There are in fact three situations to be considered:

1. As a result of normal push operations or of adjustment to SP, the stack drops below a danger line and is still there when a check is made.

2. The stack is pushed below the line but is popped back again so that at any check, SP is seemingly in a harmless position.

3. By adjustment, SP is moved below the line, where it is used to store data, and is again returned out of harm's way before it is checked.

The Monitor can readily cope with the first two situations. At all times it maintains a pointer to its own dynamic end, namely that for its own permanently assigned memory area extended by modules loaded for a program run and by currently assigned buffer space with a small safety margin. This pointer is stored in the SVT as Top of Buffers (TOB) - see Section 2.1.1. A simple comparison of this against SP is a sufficient check for the first case. At the same time the Monitor sets a bit pattern into its end word as shown by TOB. It is highly probable that this pattern will be corrupted in the second situation and a simple check resolves this.

The checking operations must of course be carried out as often as possible to be effective. Since Register-saving through subroutine S.RSAV normally occurs as the first step whenever the Monitor gains control for any length of time, (see the EMT Handler in Section 2.2.1 and device drivers in Section 3.3), the check is included at this time. If a violation is seen, S.RSAV moves SP back to allow collection of the Error Diagnostic Print routine from the system-device and requests the output of a fatal F001 message (see Chapter 7). A problem can happen though, if the violation is by the system-device, especially when working on the Monitor's behalf. As a result, for this case, the system-device is al-

lowed to proceed to completion - the violation is picked up later.

It is obviously possible that the Monitor itself is the offender and not the user. It might therefore seem necessary that the check be made not only on entry but also on exit, i.e., in S.RRES as well. However in most cases, the Monitor's wholesale usage of the stack is confined either to saving Registers or, if not, is followed by such action. Thus any violation is seen almost immediately. The additional overhead of checking again every time, merely to catch the outside chance of this not always being the case, is too excessive and so the check is omitted from S.RRES.

Unfortunately, the Monitor has no way of providing for the third situation - even the hardware protection does not adequately cover this. It is only the user who can therefore prevent it. Thus in the Programmer's Handbook, he is asked to exercise care if he adjusts SP for any purpose. To assist him in knowing where the danger line is, the Monitor provides a means by which he can ask, within the General Utilities call (see Section 5.2). He is also advised to wait until he has completed I/O set up operations (i.e. .INIT & .OPEN) before making the adjustment as there is then less chance of running into trouble. Figure 2-9 is given to illustrate the advantage of doing this.

2.4.2 Free Core Management

As indicated in chapter 1, the Monitor uses free core space above its end as an area in which it sets up buffers in support of a user program's requests for I/O services. The area is quite dynamic in that as demand increases so does the buffer space provided and likewise when the demand goes away. Thus at any time the total area currently allocated is defined from the start of the first available buffer as stored as end of Monitor (EOM) in the SVT to the end of the last buffer still in use (with a 16-word safety margin) as shown by TOB in the SVT (see previous section).

In order to control the allocation of buffers and their later release, the Monitor maintains a bit-map for the free core area. This is merely a number of words in which each bit represents a 16-word buffer space. As noted above, the map is always originated at EOM; thus the first 16 words above this correspond to bit 0 of the first map word. Bits thereafter are numbered from right to left. The length of the table and hence the area it potentially controls is established at Monitor initialization (see Section 2.1.4). Presently half the available memory is set as the absolute limit. Bits are cleared to 0 while the buffers they repre-

sent are free; they are reset to 1 when the buffers are assigned. The end of the whole current buffer area denoted by TOB therefore is also marked by the last map bit set to 1. (see Figure 2-10).

2.4.2.1 Buffer Allocation

The buffer allocation is effected by a call to sub-routine S.GTB with access through vector location 54. The total space required must be supplied; this is done by pushing the number of 16-word units onto the processor stack as part of the call, e.g.

```
MOV #4,-(SP)         ;64-WORD BUFFER REQD.
MOV @#54,-(SP)       ;CALL S.GTB VIA VECTOR
JSR R5,@(SP)+
```

Provided that the necessary space is available, the subroutine returns its start address on the stack-top; 0 is used to show non-availability.

The allocation process functions as follows (see also Figure 2-11):

1. Save the calling routine Registers & set pointers to EOM & the bit map start (shown in SVT as BFS - see Section 2.1.1.)

2. Find the position in the map of the lowest 16-word buffer available on the basis of the first 0 bit. At same time maintain a pointer at the end of the buffer unit being checked.

3. Check if sufficient contiguous 16-word units follow to meet the specified number. If not, repeat step 2 and step 3 until the map-end stored in SVT as BFE is reached. If this occurs, exit with 0 on top of the stack. Similarily if at any time the end of area being considered moves into the processor stack area, do likewise.

4. If the total required area is available, update TOB and reset the stack-stop pattern word (see previous Section) as necessary.

5. Set appropriate map-bits to 1 to show the allocation, store the start address of the area on the stack, restore the saved Registers and exit.

2.4.2.2 Buffer Release

Buffer release is performed by a call to subroutine S.RLB which uses location 56 as its vector. In this case, two arguments must be passed - both the start address of the buffer area being relinquished and its size in 16-word units. This gives a calling sequence as:

```
MOV #ADDR,-(SP)         ;GIVE BUFFER ADDRESS...
MOV #N,-(SP)            ;...& SIZE
MOV @#56,-(SP)          ;RELEASE IT.
JSR R5,@(SP)+
```

Provided that the arguments are apparently valid, the appropriate map bits are cleared to 0 and TOB is adjusted to the end of the last buffer still allocated. The stack is cleared on exit.

The release is effected as follows (see also Figure 2-12):

1. Save the calling routine's Registers. Set relative pointers to the start and end of the area to be freed, based on the parameters supplied. Exit with the stack clear if the area is outside the range of the map or if no size is given.

2. Compute the bit position for last 16-word unit in the area specified and set a mask accordingly. Again exit if the supplied address does not coincide with a correct 16-word unit start point.

3. Clear appropriate bits in the map; at the same time track the start of each unit being released.

4. If the end of this buffer is also the end of the whole area presently allocated, continue tracking until either a 1 bit or the start of the map is reached. Reset TOB and its stack-stop accordingly.

5. Restore saved Registers, clean-up stack and exit.

2.4.2.3 Comments

It should be noted that once a buffer is assigned to some purpose, it remains so assigned, even though units before it may be released first. Reshuffling to prevent gaps is not considered worthwhile when this might mean the necessity of maintaining extensive data on the role of each buffer. Instead S.GTB attempts to fill up any gaps before claiming more free core (see step 1) and S.RLB ensures that not only

the specified buffer but also any preceding gap is returned to free core (see step 4). The user can also keep usage to a minimum by taking care in his sequence of requests for routines causing buffer allocation (.INIT's,.OPEN's). Thus this is advised in the Programmer's Handbook.

2.5 Time Control

Provided that the configuration includes a line-clock, DOS maintains the time of day (TOD) in two words within the SVT (see Section 2,1,1) as:

```
        TOD:    .WORD   0,0
```

where TOD represents the high-order 15 bits and TOD+2 the low-order 15 bits of time in clock ticks.

The sign in both cases is positive, i.e. 0. The value is initialized by means of the console TIME command (see Section 6.4.)

The clock itself is set in motion when the Monitor initialisation routine enables its interrupts, assuming that the requisite driver has been included within the permanently resident Monitor (see Section 2.1.4). Thereafter, at each interrupt, the driver merely performs a double precision increment of the stored value. It does not however attempt to make 24-hour adjustment or any form of calendar check, both of which would increase the driver's size. The operator must therefore reset the stored value daily, again by a console TIME entry.

LOCATION	CONTENT	CONTENT SYMBOL
40	START ADDRESS OF THE SYSTEM VECTOR TABLE.	SVT.
42	BASE ADDRESS FOR THE GENERAL SYSTEM EXIT.	S.XIT
44	START ADDRESS FOR THE REGISTER-SAVE SUBROUTINE	S.RSAV
46	START ADDRESS FOR THE REGISTER-RESTORE SUBRTN.	S.RRES
50	START ADDRESS FOR THE DRIVER QUEUE SUBROUTINE	S.CDB
52	START ADDRESS FOR THE DRIVER DEQUEUE SUBRTN.	S.CDQ
54	START ADDRESS FOR THE GET-BUFFER SUBROUTINE	S.GTB
56	START ADDRESS FOR THE RELEASE-BUFFER SUBRTN.	S.RLB

Figure 2-1:- USAGE OF THE FIXED VECTOR LOCATIONS.

	SYMBOL	PURPOSE	NOTES
0 SVT:	EOM	END OF MONITOR	Dynamic Origin for Free-core Buffer Space.
2	TOB	TOP OF BUFFERS	Dynamic End of Allocated Buffer Space
4	CSA	CORE SIZE AVAILABLE	Set on initialization to highest memory address (normal)
6	PLA	PROGRAM LOAD ADDRESS	Set only when a program is in core (lowest point loaded)
10	SCW	SYSTEM CONFIGURATION	Reserved for Bit Switches to indicate available facilities
12	BAT	BEGINNING of D.A.T.	Set only if a Device Assignment Table is established.
14	DCO	DDB CHAIN ORIGIN.	Indicates that program still has extant DDB's.
16	MUS	MONITOR / USER SWITCH	Low Byte: 1 = Program Loaded; −1 = Program Stopped. High Byte: 1 = Program Running; −1 = Program Waiting
20	OSW	OTHER SWITCH	Reserved for as yet unimplemented Batch-Type command.
22	PSA	PROGRAM START ADDRESS	Set if program in core to address in source .END [or] if none]
24	DSA	DEBUG PROGRAM START	Set if ODT-11R is linked to a loaded program
26	RSA	RESTART ADDRESS	Set by program for RESTART at console keyboard
30	WRA	WAIT RETURN ADDRESS	Saves PC for suspended program or is set to WTL (below)
32	DAT	DATE	Saves value entered by DATE command [1000 (year-70) + Day]
34	TOD	TIME OF DAY	Double-precision value [15 bits + Ø sign] as entered by
36			TIME command and then incremented by CLOCK
40	UIC	USER ID. CODE	Set at User LOGIN; cleared at FINISH
42	PGN	PROGRAM NAME	6-character value associated with source .TITLE
44			of a loaded program (packed in Radix-5Ø - single prec)
46	MRT	MRT START ADDRESS	Used for access to the Monitor Residency Table
50	DDL	DDL START ADDRESS	Likewise for the Device Driver List
52	SSP	SAVE STACK POINTER	Used during execution of DUMP command.
54	BFS	B.A.T. START	Start of Buffer Allocation Table (end of permanently resident Monitor)
56	BFE	B.A.T. END	Set on initialization to make BAT represent half of memory
60	WTL	SYSTEM WAIT LOOP	= BR . -effective while a loaded program is suspended
62	KBA	KEYBOARD DRIVER ADDR.	Set to DDL entry for Driver KB's core address (if in)
64	MSB	MSB START ADDRESS	Used for access to the Main Swap Buffer

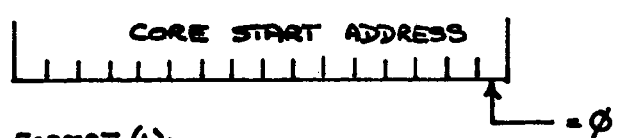
Figure 2-2:- System Vector Table Content.

MRT:

EMT CODE	
Ø	REQUESTS FOR I/O SERVICES
1	
2	
3	
4	
5	
6	
7	
1Ø	
11	
12	
13	
14	
15	
16	
17	
2Ø	
21	
22	
23	
24	
25	
26	
27	
3Ø	'EMERGENCY' SERVICES USING THE KEYBOARD SWAP BUFFER
31	
32	
33	
34	
35	
36	
37	
4Ø	OTHER PROGRAM SERVICES
41	
42	
43	
44	
45	

FORMAT (a):-

CORRESPONDING ROUTINE IS IN CORE:-

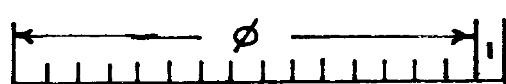


FORMAT (b):-

CORRESPONDING ROUTINE IS IN THE SYSTEM LIBRARY EXTERNALLY:-

FORMAT (c):-

NO CORRESPONDING ROUTINE EXISTS:-



(*) = # OF 64-WORD BLOCKS (ØØ:= 4)

Figure 2-3: MONITOR RESIDENCY TABLE FORMAT.

DDL.:

ENTRY FOR SYSTEM DEVICE

ENTRY FOR DEVICE #2

ENTRY FOR DEVICE #3

ENTRY FOR DEVICE #4

ENTRY FOR DEVICE #5

ENTRY FOR DEVICE #n-1

ENTRY FOR DEVICE #n

DDL.ND:

DEVICE NAME (Radix-50)
MEMORY START ADDRESS
INTERRUPT VECTOR ADDRESS
DDL END ADDRESS

DEVICE NAME (Radix-50)	
MEMORY START ADDRESS OR Ø	
INTERRUPT VECTOR ADDRESS	
SIZE IN 16-word UNITS	START BLOCK ON SYSTEM-DEVICE

Figure 2-4: DEVICE DRIVER LIST FORMAT.

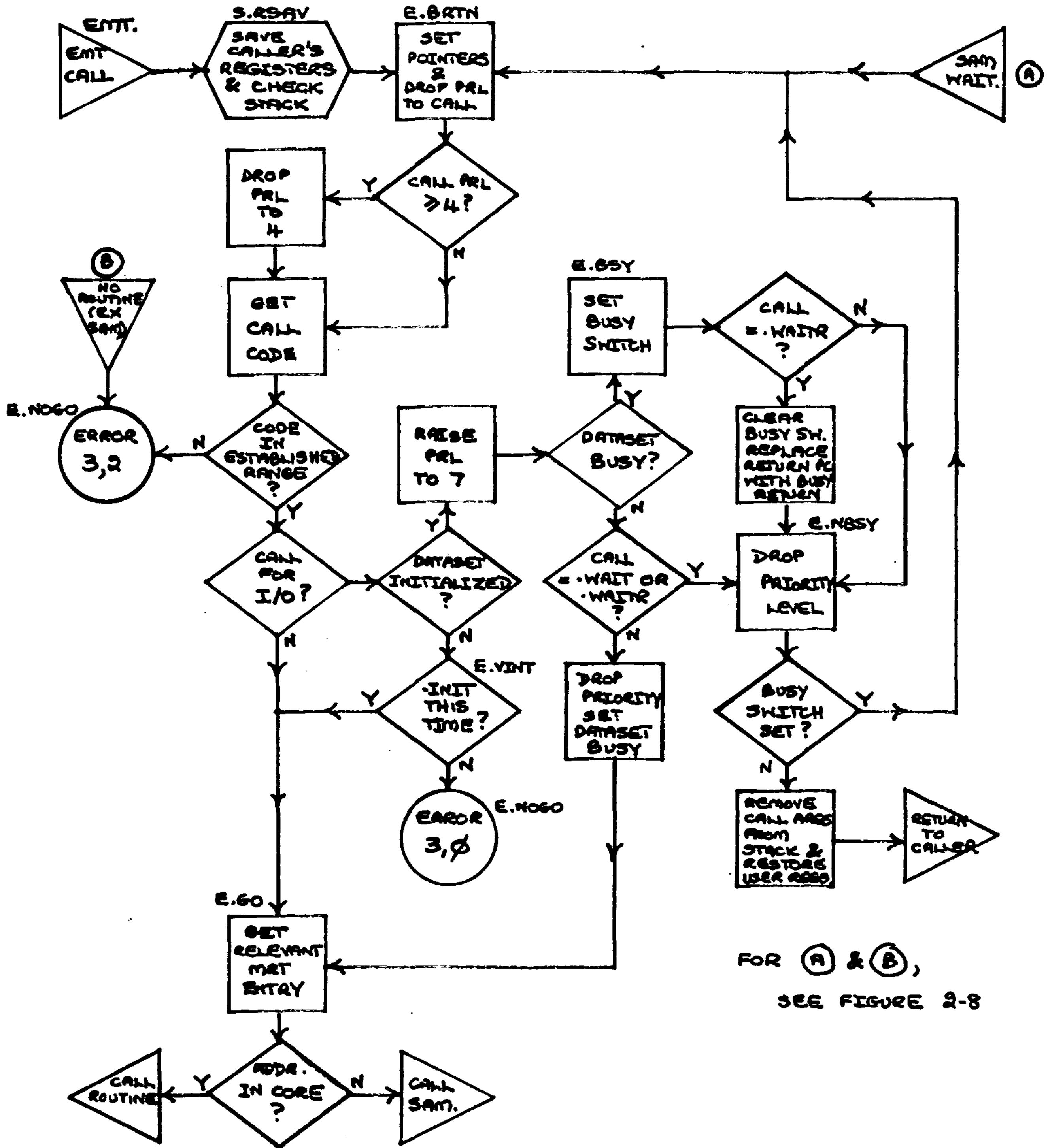
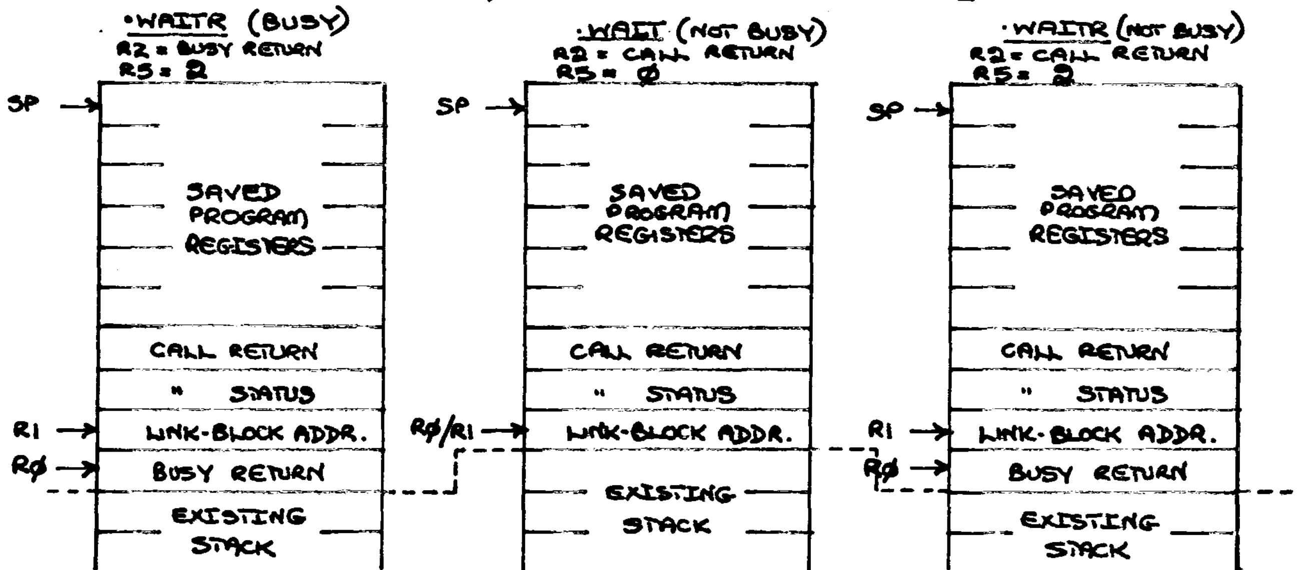
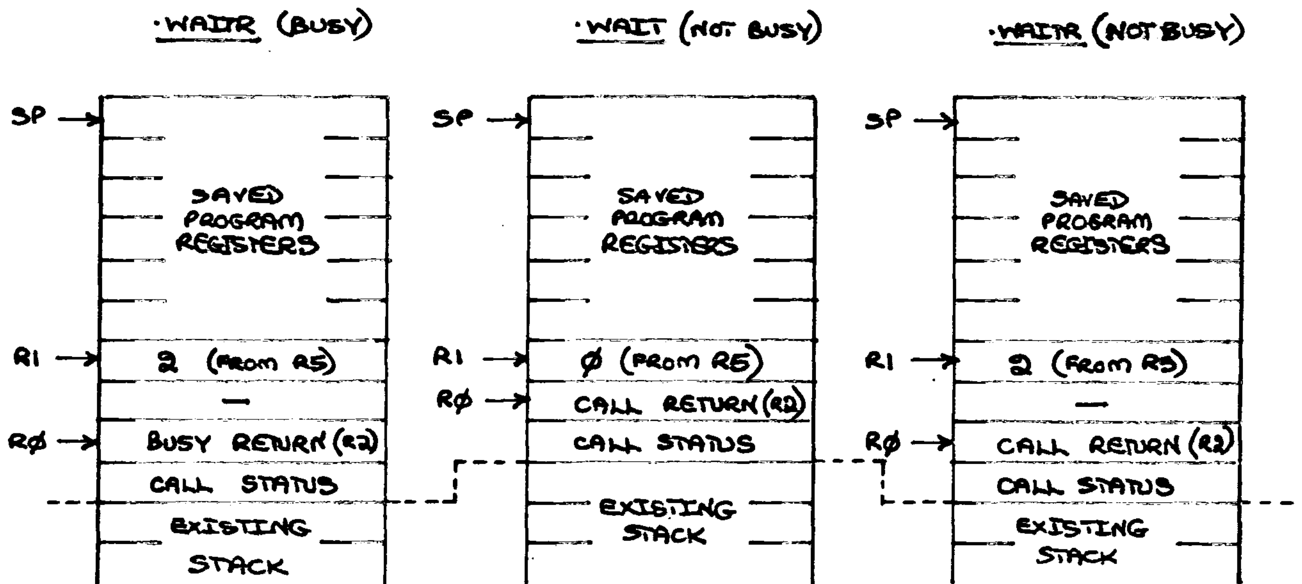


Figure 2-5: EMT HANDLER OPERATIONS.

STAGE 1: SETS POINTERS AS SHOWN BELOW; STACK AS SET AFTER REGISTER-SAVE; R2 & R5 SET AS INDICATED. [RØ = R1+R5]



STAGE 2: COMMON CODE SEQUENCE ADJUSTS STACK AS SHOWN BELOW:



STAGE 3: REGISTERS ARE RESTORED, THE STORED R5 INDEX VALUES ARE ADDED TO 'SP' -AFTER INCREMENT-2 'RTI' IS CALLED.

Figure 2-6: STACK SHUFFLE TO REMOVE CALL ARGUMENTS FOR IMMEDIATE PROGRAM EXIT FROM THE EMT HANDLER.

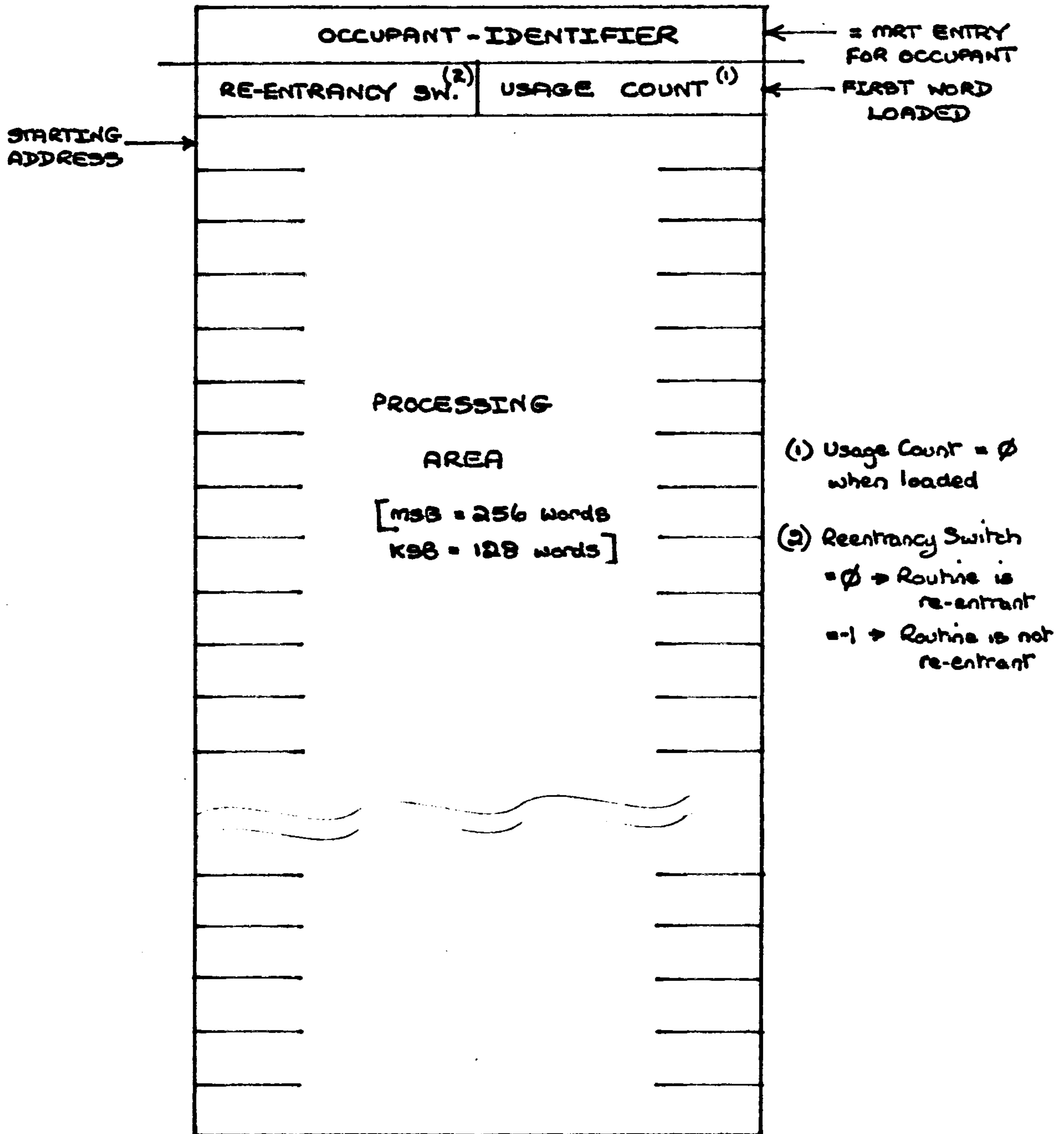


Figure 2-7: SWAP BUFFER FORMAT.

GENERAL SAM PROCESSING

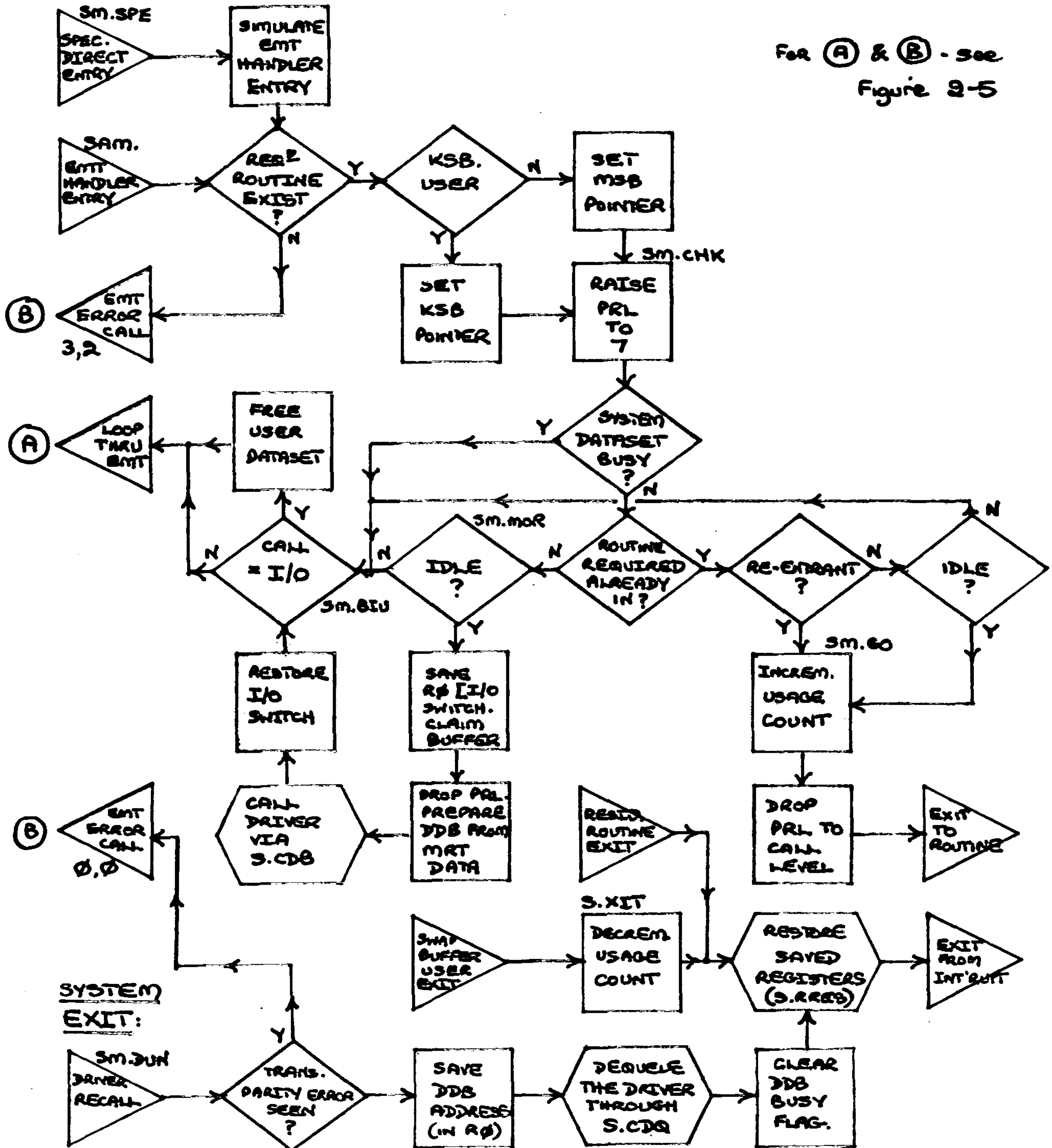


Figure 2-8: SAM OPERATIONS

ASSUMPTION: PROGRAM REQUIRES TWO FILES:- OUTPUT TO DECTAPE FROM DISK INPUT.

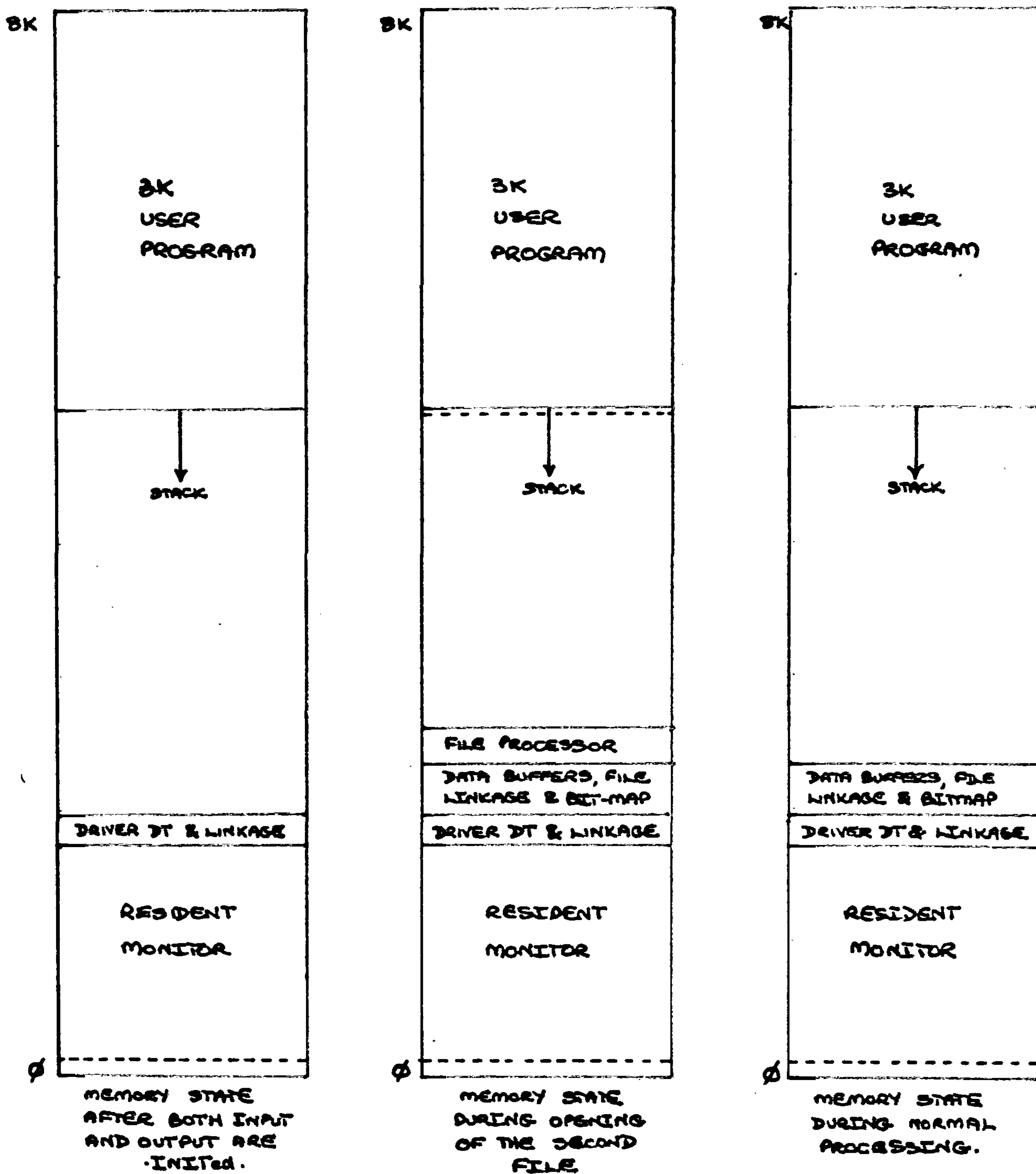


Figure 2-9: MEMORY USAGE DURING I/O SET-UP.

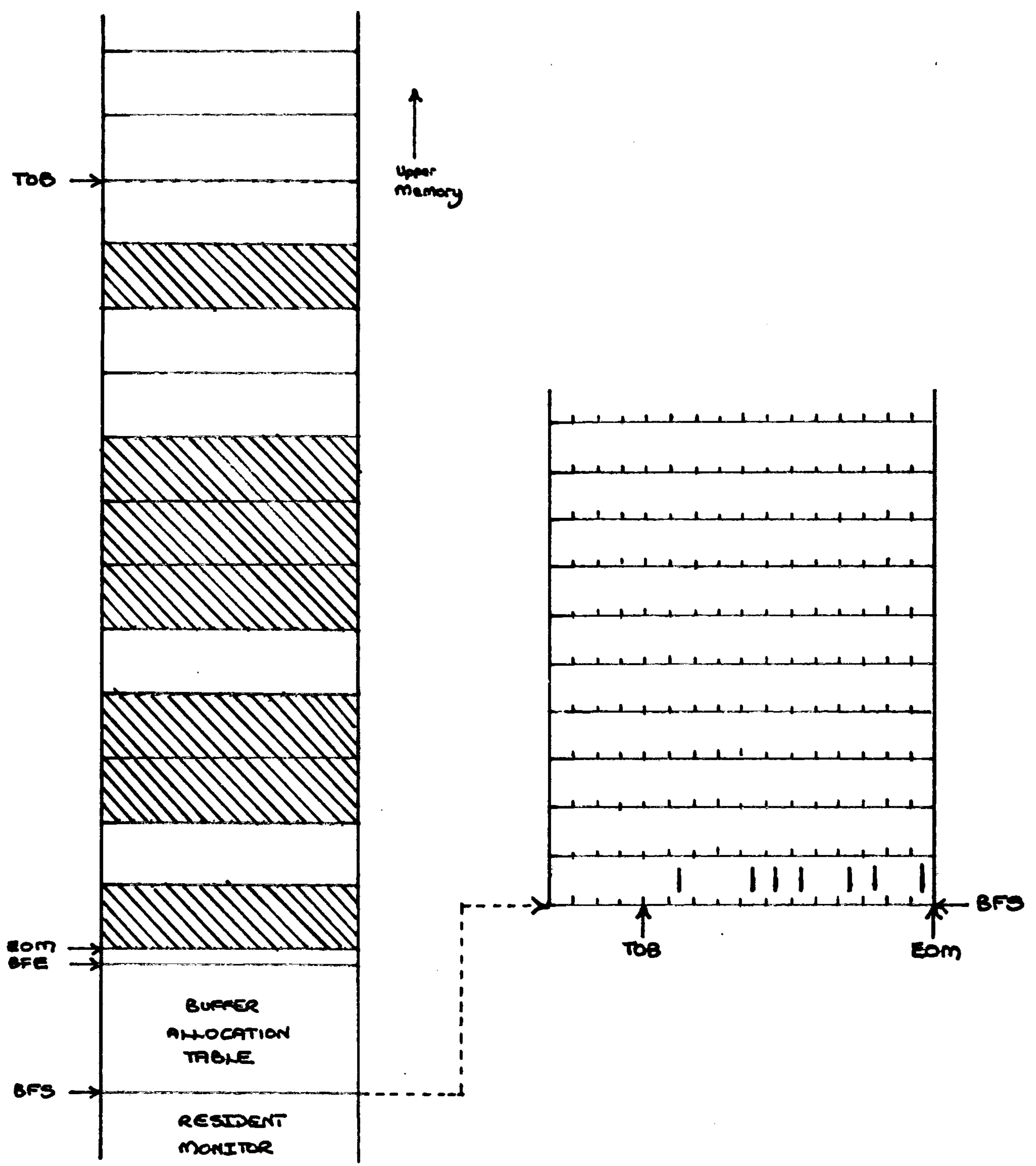


Figure 2-10: Buffer Allocation Management

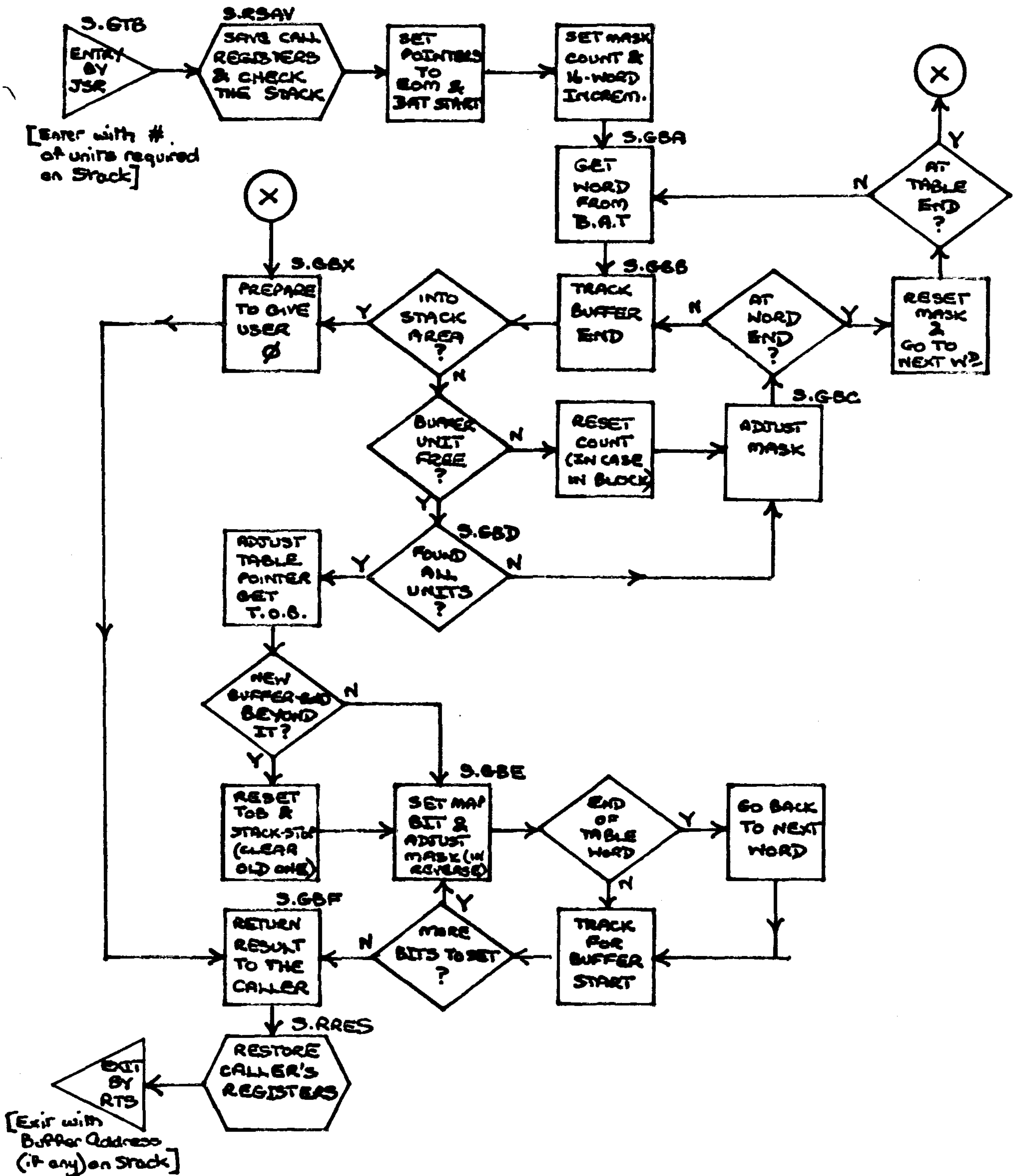


Figure 2-11: BUFFER ALLOCATION OPERATIONS

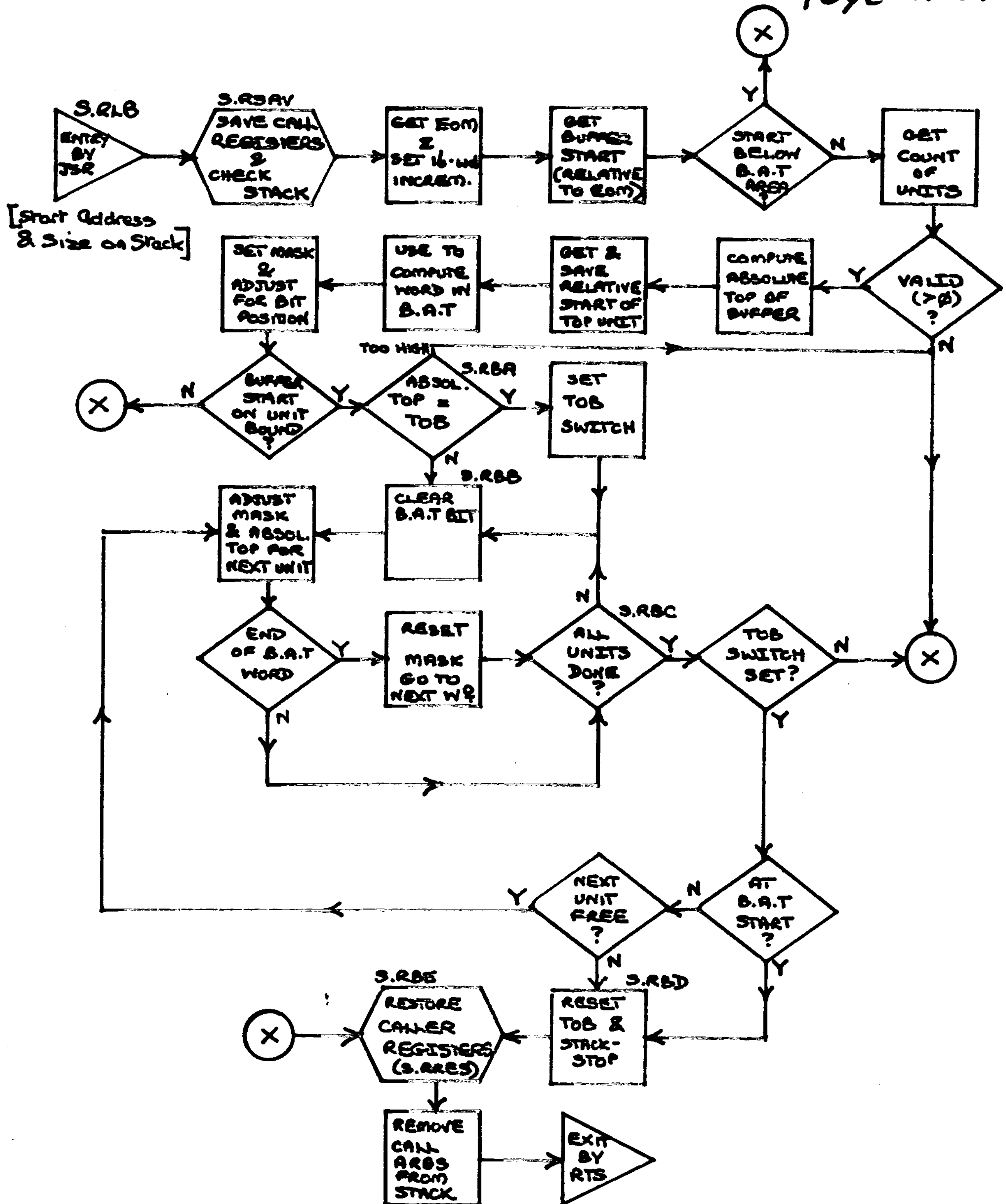


Figure 2-12: BUFFER RELEASE OPERATIONS

CHAPTER 3

GENERAL I/O PROCESSING

By far the major function of the DOS Monitor is to relieve a running program of the problems involved in moving its data between the PDP-11 and its peripheral I/O devices. Moreover while performing this function, the Monitor aims at meeting the following criteria:

1. Core economy - usage should be restricted to the minimum needed merely to service the I/O operations currently underway.

2. Machine efficiency - as far as possible the program should be allowed to continue alternative processing while waiting for normally relatively slow I/O to complete.

3. Device independence - if he so wishes, the user should be able to delay specification of the actual devices to be used by the program right up to the point of execution.

4. Shared usage of bulk media - the storage and retrieval of different sets of data upon bulk media should be comparatively simple for one user or several, with some measure of mutual protection.

The purpose of this chapter is to show how the first three of these goals have been applied for all devices, regardless of their nature. Section 3.1 further discusses the principles and explains the general control techniques employed. Section 3.2 describes in detail the routines which provide common processing services for all devices. Section 3.3 illustrates the general format specified for the drivers which control the devices on behalf of those routines.

The fourth objective is the principal subject of chapter 4. This will also show how provisions covered in this chapter are extended when they are applied to bulk-storage devices in particular.

3.1 I/O Concepts and Control

The purpose of this section is to show how in general the Monitor attempts to satisfy the three main objectives for all I/O defined in the introduction. The overall strategy is described firstly in section 3.1.1. Section 3.1.2 then illustrates the methods used to implement the strategy in order to maintain adequate Monitor control of I/O operations.

3.1.1 General Strategy

Five concepts are considered in the following paragraphs.

3.1.1.1 Common Processing

The basic concept is that, as far as is reasonable, all I/O processing is performed by common potentially reentrant routines which only call a device driver when some actual physical device action is needed. This cont has the following advantages when considered in the light of the stated goals:

1. The routines themselves, like other Monitor modules, can be brought into memory only when required. Because they use the already allocated Swap Buffer when they are, no extra core is needed for them. Moreover the drivers, being devoted solely to processes needed to control the device hardware they represent, are considerably shorter than they might be otherwise. Their core usage is therefore kept to a minimum.

2. The potential re-entrancy of the routines allows their use for several requests at a time. Thus no restriction need be placed upon the alternative processing a program may wish to perform while I/O is underway.

3. The use of common processors forces the definition of a standard interface for all drivers. This simplifies the implementation of device-independence.

3.1.1.2 Buffered Data

Because of the slow access rates of I/O devices, most programs make provision for the buffering of data being passed between the computer and its peripherals, particularly if any form of overlapped processing is to be effected. In general, these programs are developed with the devices to be used in mind. Their buffers are, as a result, set up to satisfy not only the requirements of the program but also device capabilities for handling data - singly or in blocks of varying size. In one sense, however, device-independence means that the program need consider solely its own processing needs for its buffers. The Monitor instead must take care of the varying device data-handling attributes.

For simplicity in the standard driver interface mentioned earlier, the Monitor therefore treats all devices as block-structured and establishes its own intermediate buffers accordingly. In each case the size of the block, and hence of the buffer provided, is fixed as convenient for each device, but can vary between devices. The drivers then transfer to or from these buffers as directed by the Monitor and as far as possible this is done in advance of the program's requirements, in the interest of machine efficiency. Thus when a program needs input, say, it can ask for a fixed amount of data and the Monitor may, depending upon the device, be able to satisfy the request from data already in core and perhaps have more for next time.

If should be noted, nevertheless, that the Monitor provides only a single buffer for each I/O task and the program may be forced to wait if more data must be transferred to meet any request. Greater machine efficiency might follow from double buffering. However this obviously uses more core. Furthermore it already exits to some extent (as Monitor buffer and user buffer) and so no further provsion is made. This does not of course prevent the user double-buffering within his program if he wishes to speed-up his I/O activity.

3.1.1.3 Dynamic Core Usage

Although the drivers may be relatively small as noted in section 3.1.1.1, they must still take up some core while they are in use - likewise the internal buffers discussed in the previous section. Moreover the need for their presence is not transient to the same extent as that for the normal Monitor routines: since it would be wholly unrealistic to force every device transfer to be associated with a second one from the system-device, once (a driver) is called to begin a series of I/O operations it must remain available in

memory until the series is complete - and its internal buffer with it. On the other hand, it is equally advisable that neither the driver nor buffer should occupy core until they are actually needed or again after they have served their purpose: that core might be put to some other use.

It is thus for cases like these that the dynamic allocation and release of buffer space within free core as described in section 2.4.2, is mainly provided. In addition certain of the Monitor's I/O processing modules to be discussed in section 3.2, primarily exist in order to allow the user to control the usage of core in this way.

The fact that a driver is loaded only when required and not with the program also furthers the cause of device independence in that the user can, under certain circumstances, rectify errors in device specification even after a program has been started. (see section 3.2.1.1)

3.1.1.4 I/O Levels

In section 3.1.1.2 it was shown that the Monitor can allow a user program to request its data transfers in quantities suited to its own needs rather than to meet device characteristics. The Monitor also enables the program to indicate how the move is to be made and whether any checking is to be carried out in the process. It is at this, the READ-WRITE level (see section 3.2.2), that most users will in fact wish to perform their I/O operation in the normal way.

For some applications however, the time taken both to move the data across two buffers and to accomplish the various processing forms may be prohibitive. Two other levels are therefore provided. At the very basic, or .TRAN level (see section 3.2.1.), the user sacrifices device-independence but then virtually has direct access to a driver to transfer data between his program area and the device without the Monitor's intermediate buffering. At the second, or BLOCK level (see section 3.2.3), the user is given access to the Monitor's buffer, instead of supplying his own, in order to carry out random-access I/O. At both levels, though, the user program must perform its own data-processing.

3.1.1.5 Device Assignment

As will be shown in section 3.1.2.1, the user can designate the device to service each dataset within a Link-block in the program. Similarly, if the device is file-structured, as described in chapter 4, he may also name the file he wishes to access in a program File-block (see section 4.3.1). On the other hand, he may not know this information when the program is written as perhaps in the case of a general copy program. Even in the former situation, there may be occasions when the original program specifications must be changed (a line-printer is temporarily out of action, for example), so a listing is to be output to a disk file or again, because a particular data-acquisition rate was too high for on-line processing, the data was dumped on the disk and now the processing program must access the latter rather than the data-aquisition device. The Monitor therefore provides two methods by which the user may specify the device (or file) at program run-time:

1. He can enter the run-time specification via the console ASSIGN command described in section 6.4. This will then over-ride the corresponding program detail.

2. He can structure the program to accept a command string and use the Command String Interpreter to perform the necessary decoding and set-up (see section 5.4). This method is especially appropriate if separate runs of the same loaded program use different devices and files, e.g. As in an Assembler.

3.1.2 I/O Controls

In order to maintain the concept of device-independence, the user is encouraged to program I/O in terms of the logical purpose served within the program by the data transferred rather than of the physical device involved. Thus, in the Programmer's Handbook, all I/O functions are shown to be performed not upon a device but upon a "dataset" - defined as "a logical collection of data which is treated as an entity by the program" (in other words, all the data which is processed in the same manner within the program). In practice, dataset and device may in fact mean the same thing. In a DECtape copy program, for instance, the tape being copied and the one being produced each constitute a dataset. On the other hand, a dataset may be less than all the data a device can supply, e.g., it may be just one of several files stored on the same DECtape (see chapter 4). At times, it may even stretch across several devices: as a typical exam-

ple, all the object modules being linked into one load module, whatever their source, form a single dataset to the linking program, LINK-11. Two points must however be noted:

1. A dataset can only support one source or destination at a time. Thus although two disk files being merged by a sort program both form its input, each must be associated with a different dataset.

2. It is conceivable that when a dataset is actually associated with a bi-directional device - the console typewriter perhaps, - the user might wish to use the dataset also bi-directionally. By definition this cannot be: logically input and output are processed differently. Nevertheless, usage of the same dataset linkage within the program (see section 3.1.2.1) is not completely out of the question if this produces economies, provided that such usage is sequential, i.e. the dataset can be re-initialized for output as long as it is released from any commitment to input.

Sections 3.1.2.1 to 3.1.2.3 following describe the channels set up by the user and the Monitor to control the operations upon each dataset. Section 3.1.4 discusses the mechanism by which the drivers servicing the datasets are managed.

3.1.2.1 User Link Block

For each dataset to be used concurrently, the user program provides a data-block of potentially variable length known as a Link-block, as illustrated in figure 3-1. This is described in detail in the Programmer's Handbook. Briefly, its purpose is to enable the user to identify the dataset both internally - by passing its address as a call parameter for each request for an I/O service - and externally by a logical name which can be used to associate the data-set with a physical device assigned to it at run-time (see next section). The Link-block also allows him to specify the device within the program or may provide space for additional information if he intends to obtain such specification through a command string (see section 5.4). Finally he can indicate what action the Monitor should take if an I/O request cannot be satisifed through lack of free core for the driver or buffers needed.

3.1.2.2 Device Assignment Table

In order to give the user as much flexibility as possible, Device/File specification by console assignment as noted in section 3.1.1.5 can occur at various times, even before the program is loaded. The Monitor must therefore store the data from each ASSIGN entry. For this purpose, a Device Assignment Table (DAT) is set up as required within the Monitor area (see the ASSIGN command processor in section 6.4.10) and its entries are then checked whenever the program under execution calls for dataset initialization (see section 3.1.2.1). When in existence, the table's start address is stored in the Beginning of Assignment Table (BAT) word in the SVT (see section 2.1.1).

Set up only by assign command?

As illustrated in figure 3-2, the table itself may consist of separate segments as follows, due to the varying memory loadings at the times when assignments are entered:

1. Prior to program loading - the permanently resident Monitor is virtually alone in core. The table may therefore begin immediately above its end. Entries made at this time need not as a result disappear when the program is removed from core on completion of its execution. Thus the user is given the facility of assignments which affect not just one program but perhaps several operating together in a suite. The fact that for this case the table starts where the Monitor Buffer Allocation Table ends is used to protect these entries. If this is the only DAT segment, the EOM stored in the SVT is set at its end (see section 2.1.1).

2. After program loading but before program execution - it is possible that the normally resident Monitor may be extended by routines loaded for the program duration. DAT entries here must come after these routines and, since the routines must later be removed with the program, the entries also go. (If in fact no routines are loaded, a word is left between the Buffer Table and DAT start as an indicator). EOM is adjusted also to the end of this segment.

3. After the start of program execution - the area from EOM is under the control of the Monitor buffer management routines described in section 2.4.2 and some buffers may already be allocated to other purposes. Thus further DAT entries must also use buffers similarly obtained. For simplicity in the ASSIGN command processor, each entry uses a new buffer even though this wastes space. Assignment at this time is therefore not recommended except as

a means of rectifying an error detected during dataset initialization (see section 3.2.1.1). There are other reasons too:

a. The buffer used by the entry cannot be released until the program terminates - this could lock out the buffer areas from free core even though they are no longer in use.

b. As part of the normal clean-up carried out during the processing of the console BEGIN command (see section 6.4), all free core buffers are released for a fresh start.

c. Unless the assignment is made before dataset initialization it will have no effect.

The segments are connected by a two word entry at the end of each. The first word is set to -1 to indicate that another segment follows, the second then shows the start address of that segment (1). (This linkage is also used to delete an entry amended by a later assignment). The end of the complete table is denoted by 0 in the first word.

The format of each entry is given in figure 3-3. The significance of the principal items is as follows:

1. Logical name - corresponds to the same item in the user Link-block discussed in the previous section and thus provides the connecting link between the entry and the dataset it affects.

2. Device name - shows the device to be associated with the dataset.

3. Device unit - stores relevant identification if the device controls several units.

4. Number of words to follow - allows restriction of the entry size to the minimum needed for any file-specification entered, e.g., if File-name only, this will be 2.

----

1. The second link word may be 0. This indicates the table end if a console BEGIN removes assignments in buffers (see Section 6.4.5).

The remaining items supply data otherwise given in the user File-block (see section 4.3.1). The logical name, of course must be supplied. The rest may be validly omitted as shown in section 6.4.10. (If their space must be included), the appropriate table-slots are then set to 0 e.g., if only UIC is entered, item 4 must be set to 4; File-name and Extension will be left blank.

3.1.2.3 Dataset Data Block

The Monitor has no knowledge of the program's I/O requirements until the program itself identifies these by requesting initialization of the datasets to be used. Hence, as stated in the Programmer's Handbook, .INIT must be the first function called in each case. In response to such call, the Monitor sets up its own control block for the dataset within a 16-word buffer unit claimed from free core and stores its address in the link-word of the user Link-block (see section 3.1.2.1). This block known as the Dataset Data Block (DDB), is used thereafter as the means by which different Monitor routines pass information to each other and also communicate with the driver servicing the dataset. It is retained in memory until the program releases the dataset from further I/O action (see section 3.2.1.3); its buffer space is then returned to free core.

Link pointer

The format of the DDB is illustrated in figure 3-4. The purpose of the items is as follows:

1. Monitor link - enables the Monitor to maintain a control chain of the DDB's currently established, as illustrated in figure 3-5. The chain originates at a DDB Chain Origin (DCO) word in the SVT (see section 2.1.1).

2. Q link - is used to chain the DDB's for datasets waiting upon the services of the same device as described in the next section.

3. Priority level - stores the level at which a queued call to the driver is made.

4. Driver routine index - is normally 0 unless a driver call is queued: a pointer to the driver routine required is then saved here instead.

5. Driver address - contains the address from which the driver associated with the dataset is loaded. This also serves to identify the driver when compared with entries in the DDL (see section 2.1.3).

6. Busy flag - is set by the EMT Handler (see section 2.2.1) to the address of the program I/O call when accepted. This forces any subsequent call to wait until the current request has been satisfied - when it is reset to 0. The address of this word is the one stored in the user Link-block as the DDB connection to its dataset.

7. User line address - in general is used to save the second parameter passed by most I/O calls (normally the address of a data-block in the program, e.g., Line Buffer in .READ/.WRITE, TRAN-block in .TRAN, etc.

8. Device block - points to the device address of a block on a bulk storage medium.

9. Buffer address - shows the start of a memory area for transfer - normally this is an internal Monitor buffer; thus this word is also used as an indication of the current assignment of such buffer (cleared to 0 if none).

10. Word count - gives the number of words to be transferred by the device as a two's complement value.

11. Status - is used to direct the driver on the type of transfer required, allows the driver to return error indicators and stores other control information on a bit-basis as shown in figure 3-6.

12. Completion return - shows the address at which a calling routine requires the driver to return when a requested service has been satisfied.

13. Driver word count - allows the driver to indicate (again as two's complement) how many words are not transferred because an end of data point is reached. This word is also used to store a variable pointer to the next byte to be processed in the internal Monitor buffer. (No device action can be underway at this time).

14. Byte count - is used during .READ/.WRITE processing to control bytes passed between the program line and the internal Monitor buffer.

15. Checksum - is mainly provided for the processing of formatted binary data (see section 3.2.2.2).

16. DAT pointer - is set during dataset initialization (see section 3.2.1.1) to the address of an entry in the Device Assignment Table corresponding to the

one does not exist for every dataset

dataset, if such exists (see previous section).

17. FIB link - connects the DDB to a 16-word buffer extension whenever file-structured operations are underway on the dataset as described in section 4.3.2. It is set to 0 otherwise.

3.1.2.4 Driver Management (RMON3 Continued)

It was noted in section 3.1.1.1 that the device driver should present a standard interface to the common I/O processing routines of the Monitor. This interface will be discussed in section 3.3. The routines in their turn, however, must also use standard calling sequences when they require the driver's services. This section will describe these.

Monitor Calls to the Driver:

Initially it is the calling routine's responsibility to ensure that the relevant data to control the device operations is set into the DDB described in the last section, i.e., Device Block, Buffer Address, Word Count, Status, Completion Return and 0 in the Driver Word Count. The routine also sets registers as follows:

R0 = Address of the DDB (Busy Flag)

R1 = Index into the Driver Interface Table, (see Section 3.3.1). This index points to a byte containing an offset from the driver start to the beginning of the routine providing the service required, e.g., Open is shown by 7, Transfer by 10, etc. address

The contents of the other Registers are immaterial; however drivers do not save them. It is thus up to the routine to do this if such contents are required after return.

The routines themselves do not actually call the driver. Normally a single device controller can only support one operation at a time, even though it may have several units. Thus the driver itself need contain only serially reusable code. However it must then be protected against recall while still performing any service. The Monitor therefore provides two sub-routines to control the use of drivers and since they must be called for every device action they are part of the resident section. Basically these subroutines provide for queued requests for a driver's services in order

that a calling program may proceed with alternative processing as far as possible.

The subroutines are S.CDB which checks the need for a queue and sets it up when necessary and S.CDQ which ensures that the driver is given jobs from the queue as it completes each one. Like the other resident sub-routines, they are accessed through vector locations (50 and 52) respectively. Since, as shown above, arguments are passed in Registers or the DDB, their calling sequences are simple:

```
MOV @#50,-(SP) ;CALL S.CDB(52 FOR S.CDQ)
JSR PC,@(SP)+
```

Driver Queueing:

Thus a routine needing a driver operation first calls S.CDB. The processing of such calls is illustrated diagrammatically at figure 3-7. It involves the following steps:

1. Extract the relevant driver start address from the DDB identified by R0 and check its first word - a busy flag in the standard interface table.

2. If this is 0, showing the driver to be currently idle, store R0 in the busy flag to claim the driver for the requested service and to provide the driver with a link to the DDB concerned.

3. Clear the driver routine index save location in the DDB to indicate direct entry and use R1 to build the address of the driver service routine required. Jump to this address, thus leaving the saved return address from the S.CDB call on top of the stack as the driver's means of immediate exit to the calling routine(1).

4. If the driver is already busy as shown by a non-0 flag at step 1, check the Q-link in the DDB addressed by the content of the flag, i.e., the one currently being serviced by the driver (from step 2). If this is 0, no queue yet exists; start one

----

1. As will be shown in Section 3.3, all drivers use the interrupt system. In general, a call to a driver merely results in the initiation of the hardware action and an exit to await the first interrupt. Hence the return from S.CDB is not the same as Completion Return stored in the DDB.

by storing the address of the DDB for the new request from R0 in this Q-link and save R1 and the priority level of this request in its own DDB. (see figure 3-8).

5. If a queue already exists, trace it via the Q-links in each DDB and insert the new request in its appropriate place either as the last at its priority level or at the end (shown by a 0 link), whichever comes first. Exit by normal sub-routine return.

On return, the calling routine is of course unaware of the fact that the driver may still not be called, since the exits at steps 3 or 5 are the same. However at this time, the difference is immaterial, the routine can do nothing but wait in either case until the driver satisfies its request. In general, the user program is recalled to continue its operations, until, as the result of an interrupt, the driver recalls the Monitor routine via the Completion Return in the DDB. It does this with Registers at the interrupt saved on the stack and R0 again set to the DDB address. Its busy flag however, is not cleared - and nothing has yet been done about further requests possibly still waiting in a queue.

Driver Dequeueing:

It is therefore the Monitor routine's responsibility to ensure that the driver is started upon its next task if any. For this a call to S.CDQ is used, normally preceded by a save of the contents of R0 since the routine needs the DDB address for its later processing and this can be lost in the possible driver recall. The operation of S.CDQ is also outlined in figure 3-7 and consists of the following steps:

1. Extract the driver address from the DDB given by R0 and clear its flag - this will be reset if another task is in line at step 4.

2. Check the Q-link in the DDB and if this is 0, exit.

3. Otherwise reset R0 to the Q-link content and clear the Q-link.

4. Using the new R0 value (pointing to the DDB to be serviced within the already established priority sequence), reset R1 to show the new driver service required from the previously saved value and return to S.CDB at step 2 to restart the driver.

   Again the saved return address, this time from the S.CDQ call, enables the driver to exit to the Monitor routine at the appropriate point from which it

can restore R0 and continue its service to the user. When it completes, the Register contents saved by the driver, as noted above, are restored as part of the normal program recall process. (see figure 3-9).

Comments:

It should be mentioned that both S.CDB and S.CDQ adjust priority levels. Normally both operate at the call level; hence S.CDQ particularly begins by dropping down from the device level since the Monitor routine is recalled from interrupt. However both perform flag checks and for safety, they raise the level to 7 temporarily while they do so - possibly an unnecessary feature in the current state of DOS development as a single-user single-task system; it is included nevertheless against possible future modifications.

3.2 I/O Processing

The purpose of this section is to examine each of the common I/O processing routines in detail within the context of the overall philosophy discussed in section 3.1. To illustrate the general effect upon the system, the routines called by a user programming basically at the .TRAN level will be handled as an introduction in section 3.2.1. Section 3.2.2 will then expand upon the basic pattern by describing the routines called for normal processing at the .READ/.WRITE level. The remaining general I/O modules for random access and special operations are covered in sections 3.3 & 3.4.

It should be noted that all the modules, being potentially non-resident (the .READ/.WRITE processor currently excepted), use techniques discussed in section 2.3 and as I/O modules incorporate concepts shown in section 3.1 (particularly driver calling illustrated in section 3.1.2.4). Hence cross-referencing to these sections by the reader will be assumed where not given.

All the routines may take advantage of register contents passed by the EMT handler as follows:

R0 = Address of the DDB for the dataset to be serviced
R1 = Stack address of the first call parameter, i.e., the address of the user Link-block.
R2 = Program call address
R5 = Processor Status Register address (or -2)

3.2.1 Basic I/O Processing

At the basic level of I/O, the user merely requires the Monitor to control the operations of a device-driver to perform direct transfers of data between his program area and the device. The program itself carries out all the processing of the data required. This is accomplished by a call to the .TRAN routine. However, before any I/O operation can be effected, the program must notify the Monitor of its requirements by calling .INIT to initialize the appropriate dataset and make the necessary device driver available in memory. Between transfer requests, the program may call .WAIT to ensure the transfer has been completed before proceeding. When all transfers have been done, the driver and dataset linkage are removed by a .RLSE call. The resulting program outline is given under .TRAN in the Programmer's Handbook.

The .WAIT processor is included in the EMT handler and was discussed in section 2.2.1. The other three routines are described in the following paragraphs.

3.2.1.1 Dataset Initialization (INR)

As noted above, the principal functions of the .INIT processor are to ensure that the driver for the device required by a user dataset is loaded into memory if necessary and to connect it to a program Link-block defining the dataset via a DDB within the Monitor. It also sets the device-interrupt vectors accordingly.

Calling Sequence:

The program calling sequence is as follows:

```
MOV #LNKBLK,-(SP) ;PASS LINK-BLOCK ADDRESS
EMT 6             ;CALL .INIT
```

Processing:

The sequence of operations carried out by the .INIT processor is outlined below and is further illustrated in figure 3-10. The state of memory on completion is given in figure 3-11.

1. Collect the Link-block address and clear it from the stack by moving the saved user Registers down. Set a pointer to the start of the Monitor DDB chain as stored in the SVT (see section 2.1.1)

2. If the link-word in the Link-block is non-0, check if its content points to a valid DDB by searching the DDB chain and ignore if not.

3. Otherwise this could be a re-init. Release any valid internally established buffers linked to the DDB, e.g., data buffer, File Information Block, Bit-map (see chapter 4), etc. Set up to use the same DDB again.

4. If no DDB is currently established, claim a buffer unit from free core, if available via S.GTB, and link it to the Monitor chain.

5. Clear the DDB completely and store its address in user link.

6. Using the dataset Logical Name from the user Link-block, search the Device Assignment Table for a valid entry (see section 3.1.4.2). If found, extract the name of any driver specified and store a pointer to the entry in the DDB.

7. Otherwise get the driver name from the Link-block. When none is given, call operator action (error message A003 - see chapter 7). If a return then occurs, a new console assignment may have been made, so return to step 6 and check again.

8. Search the DDL (see section 2.1.3) for the driver specified (1). Call operator action as in step 7 for a driver name which does not exist. Otherwise go to step 10 unless the DDL shows the driver to be already in memory.

9. From a switch set in its Interface Table (see section 3.1.1), verify the ability of a loaded driver to support more than one dataset at a time and omit the load process if satisfactory. Otherwise search the Monitor chain to ensure that no other DDB is associated with the same driver, calling for a new assignment as in step 7 if necessary.

10. Using the DDL information on the driver's location on the system device, claim a buffer of driver size from free core. If none is available for this, or for the DDB in step 6, take any user error exit supplied in the Link-block or, failing this, call a fatal error (F007).

11. Prepare the DDB for a read-transfer into the claimed buffer from the system-device based upon the DDL information and using the same routine as SAM (see section 2.3.4) for handling transfer completion. Also set the DDB busy flag.

12. Save significant Register contents and via S.CDB, call the System-device driver. Call .WAIT until done (as indicated by clearance of the DDB busy flag in the SAM sequence).

----

1. The driver for a console ASR-33 Teletype considers two devices KB and PT with different entry points. The DDL however only shows KB; hence the search for PT looks for this single entry.

The relevant entry point for device PT is computed from the entry point of device KB since the same driver contains both.

13. On completion, set the DDL to show the core location of the driver and using information from the driver interface table set up the device interrupt vectors (except for console typewriter - already connected to the resident Keyboard Listener (see section 6.3))

14. Clear the buffer address from the DDB to show that no data buffer has yet been attached and reset into the DDB "Driver Address". Collect and store the device unit from the DAT or user Link-block as appropriate.

15. Take the System Exit to release the Swap Buffer if necessary, restore program Registers saved by the EMT Handler and return.

Comments:

The re-entrancy of the .INIT processor merits further discussion. The Monitor S.GTB routine is called to obtain the buffers for the DDB and driver and this does not protect itself against interrupt (see section 2.4.2). As a result, .INIT is set to be not re-entrant when in the Swap Buffer (see section 2.3.1). However the protection does not prevent re-entry if .INIT is made resident. This is no problem under DOS currently since only single-tasking is permitted and the user program is not recalled (implied by step 12). Nevertheless care is necessary following any Monitor modification for real-time usage.

3.2.1.2 Basic Transfers (TRA)

The .TRAN Monitor call enables the user to perform direct transfers between a device and a memory area without intermediate Monitor buffering. The size of the transfer is limited only by the capacity of a single word-count (65K). The Monitor however, exercises its normal control over the device, i.e., allowing the transfer only if the driver is currently idle and queueing the request otherwise. It must be noted however that .TRAN allows absolute access to bulk-storage devices. Indiscriminate use in this respect can corrupt the file-structuring of such devices (see chapter 4).

Calling Sequence:

The .TRAN processor requires the program to supply control parameters for the transfer by means of a TRAN-block which

is illustrated for reference at figure 3-12. Briefly its purpose is to allow the user to detail the device and memory address, size and direction of the transfer and to allow the Monitor to return status information (for further detail, see the Programmer's Handbook). The address of the block is passed as an argument in the call sequence as follows:

```
MOV #TRABLK,-(SP) ;PASS TRAN-BLOCK ADDRESS
MOV #LNKBLK,-(SP) ;IDENTIFY DATASET
EMT 10            ;CALL .TRAN
```

Processing:

The sequence for .TRAN processing is relatively straightforward so no diagram is needed. Basically the steps are as shown below:

1. Extract the driver address from the DDB and store the TRAN-block address in the user Line Address word of the DDB. Clear the call arguments from the stack by moving saved program Registers down.

2. Move the transfer control parameters from the TRAN-block into the DDB (with word-count negated). At the same time check for invalid call as follows:

   a. Zero word-count

   b. Illegal function for device (e.g., read from line-printer)

   c. Invalid function

      For (a) and (c) ignore call by returning error status in TRAN-block and exit. For (b) call fatal error (F003)

3. Set the DDB Completion Return, R0 and R1 as required for a driver transfer call and go to S.CDB. While waiting, restore saved program Registers and recall the program.

4. When recalled by the driver, call S.CDQ to dequeue it.

5. Return a flag to the user in the TRAN-block for transfer of parity errors and similarly any incomplete Word-Count (as a positive value) if either of these appears in the DDB.

6. Clear the DDB Busy Flag and Buffer Address (not a Monitor buffer). Take the System Exit to return to

the calling program.

Comments:

.TRAN is reentrant at all times, regardless of the permanency of its core-residence. However because of its intermediate return to the user at step 3, a following .WAIT is advised before either the Tran-block or the data area is processed further.

3.2.1.3 Dataset Release (RLS)

The .RLSE processor basically performs the inverse function of .INIT: it releases the driver provided that no other dataset is still using it and that it is not permanently resident. It then returns the DDB buffer to free core and unlinks the user Link-block. Hence from this point, the Monitor forgets the existence of the dataset until perhaps a second .INIT restores it. .RLSE has no mechanism for handling still-open bulk-storage files; however it does perform basic .CLOSE operations on simpler devices.

Calling sequence:

.RLSE is called by:

```
        MOV #LNKBLK,-(SP) ;PASS LNK-BLOCK ADDR
        EMT 7             ;CALL .RLSE
```

Processing:

The sequence of operations for RLS is illustrated in figure 3-12. The basic steps are:

1. Save the Link-block address and remove it from stack. Extract the driver address from the DDB.

2. Check whether a Monitor buffer is still attached to the DDB indicating that no .CLOSE has been performed. Go to step 5 if not. Call fatal error (F005) if a flag in the driver interface table (see section 3.3.1) shows its device to be file-structured) otherwise.

3. Set the DDB Completion Return in case device action is needed, (using the saved returned address from a JSR call to cause execution of an embedded short form of the SAM driver completion sequence - see

section 2.3.4).

4. Determine whether the last operation on the dataset was output and then whether any valid data still remains in the Monitor buffer (based upon the value of the variable pointer stored in the DDB Driver Word Count). If so, save the DDB address and its Busy Flag content and call the driver to empty the buffer via S.CDB. Call .WAIT until finished.

5. Release the Monitor buffer, if appropriate, to free core via S.RLB, using the size shown as the driver standard in its interface table.

6. Search the DDL for the entry containing the same driver address as that in the DDB.

7. When found, trace the Monitor DDB chain for possible other users of the same driver (again allowing for the dual nature of the console ASR-33 driver - see footnotes in section 3.2.11). If any, ignore step 8.

8. When this dataset is the sole user, and the DDL entry shows that its driver is not permanently resident, reset the device interrupt vectors to trap on error, clear the driver's core address in the DDL and return the buffer it occupies to free core.

9. Unlink the DDB from the Monitor chain and release its buffer to free core. Take a normal System Exit to return to the calling program.

Comments:

Since .RLSE uses S.RLB with similar vulnerability to that of S.GTB, the remarks in the last paragraph of section 3.2.1.1 also apply here.

3.2.2 Normal I/O Processing

As discussed in section 3.1.1, the user who does not wish to consider physical devices and requires more service from the Monitor performs sequential I/O operations at the READ/WRITE level. In response to .READ or .WRITE calls, the Monitor transfers data, formatted to an extent specified by the user, between a user line of device-independent size and its own internal buffer which is filled or emptied by the device-driver as necessary. Prior to the transfers, a call to .OPEN establishes the necessary conditions for operations with any device. Similarly .CLOSE ensures proper clean-up after the transfers have been completed in all cases. The following paragraphs examine these functions in more detail.

3.2.2.1 Dataset Open (OPN)

In section 3.2.1.1 it was shown that the .INIT call merely enables the user to associate the dataset with the driver to service it. The .OPEN call is provided, in general, as the means whereby he can prime the driver and also request the establishment of the internal Monitor buffer in readiness for the transfers to come. In particular, .OPEN makes files on bulk-storage devices available for these transfers, as described later in section 4.5.1. Although the call is not strictly necessary except for the latter purpose, its use in all cases is recommended in the interest of device-independent programming.

Calling Sequence:

The calling sequence for .OPEN requires the user to provide a File-block within the program, which is illustrated at figure 4-15 (since its principal purpose is more appropriate to file-structured operations). For the general case, it enables the user to show the intended transfer direction and to supply an error exit if he wishes. The general form of the call is as follows:

```
MOV #CODE,FILBLK-2 ;SET HOW OPEN CODE
MOV #FILBLK,-(SP)  ;PASS FILE-BLOCK & ...
MOV #LNKBLK,-(SP)  ;...LINK BLOCK ADDRESSES
EMT 16             ;CALL .OPEN
```

"CODE" in this sequence in fact identifies different forms of .OPEN provided mainly again for flexibility in handling bulk-storage files(1). For the general case, their effect is as follows:

1. Open an existing file for update (OPENU) - implies both input and output. Since most simple devices (including magnetic tape because of inherent problems) cannot sustain both directions, this form of .OPEN is valid on file-structured devices only.

2. Create a new sequential file (OPENO) - normally precedes all .WRITE operations. It causes the setting up of an empty internal buffer. It generally results in a call to a non-file-oriented device for the output of some initializing data such as punch leader or printer form-feed.

3. Extend an existing sequential file (OPENE) - operates as OPENO on non-file-oriented devices.

4. Open an existing file for input (OPENI) - is used generally before all .READ requests. It usually results in a call to the driver for a non-file device to check its readiness. The internal buffer is established and by another driver call this buffer is filled in anticipation of the first .READ.

13. Open an existing contiguous file for output (OPENC) - is mainly provided to allow sequential .WRITE operations within a random-access area on a file-oriented device. On other devices, it acts as OPENO.

Processing:

The general .OPEN processor outline is shown at figure 3.13 and its affect on memory usage is illustrated at figure 3-14. Basically it follows the sequence given below:

1. Extract the driver address from the DDB. Save the call parameters temporarily in the DDB (User Line

----

1. The codes are allotted on the basis that bit 1 indicates a file-type dedicated to output and bit 2 similarly to input (see also Footnote to Section 3.3.2).

Address & Device Block slots) and clean-up the stack by moving saved program Registers down.

2. Check the validity of the code in the user File-block, also whether its implied direction is suitable for the attached driver (using an indicator in its standard interface table - see Section 3.3.1). For failure in either case, call a fatal error (F011).

3. Check the Open indicator in the DDB Status word. If already set to show a .OPEN has been called without a subsequent .CLOSE, exit to the user program with file error 0 as status or call a fatal error (F012) if no address for the return is supplied.

4. Determine the size of the internal data-buffer needed by reference to a standard for the device contained in the driver interface table and claim the necessary space from free core. If none is available, take the Link-block exit if given or otherwise call fatal error (F007).

5. Store the start address of the buffer in the DDB and clear the buffer (mainly for output).

6. From appropriate indicators in the driver interface table, check whether the device is file-structured or is sequential magnetic tape. (Further action in either case is described in the next paragraph)

7. For non-bulk storage devices, verify the existence of an OPEN routine in the driver and call it if included via S.CDB. Until completion, return to the calling program; then use S.CDQ to dequeue the driver.

8. Set the Open indicator in the DDB and check for input .OPEN. If so, recall the driver to fill the internal buffer, again returning to the program until done (1). Save a variable pointer to the data

---

1. This initial fill is omitted, however, for "terminal" type devices such as the console typewriter in order to make any input echo correspond to the appropriate .READ.

in the DDB Driver Word Count.

9. Clear the DDB Busy Flag and take the normal System Exit.

From the preceding paragraph it can be seen that the initial checking and setting-up of the internal buffer apply to all devices. However the .OPEN processor must then hand over control to the relevant file-management modules to complete their operations. It therefore prepares an appropriate interface. Thus for the normal file-oriented devices (Disk and DECtape), step 6 above is followed by:

7. Claim a 16-word DDB extension buffer known as a FIB (see section 4.3.2). Clear it and link it to the DDB.

8. Move the original call arguments to temporary storage in the last two words of the FIB and store the Open type code also in the FIB. Save a DECtape indicator in the DDB using driver interface table information.

9. Set Registers appropriately as follows:

```
R0 = Address of the DDB
R1 = Address of the call parameters
     as stored in the FIB
R2 = Address of the Open type code
     in the user File-block
R4 = Address of the FIB.
```

10. Prepare the appropriate EMT call to the file-management OPEN routine required and call it (see section 4.5.1). (1)

Similarly for sequential magnetictape, the following steps occur:

----

1. Since this may require Swap Buffer release the call cannot be made from within the Swap Buffer. Hence depending upon the current location of the .OPEN processor as shown by its Usage Count (see section 2.3.4), the following sequence is stored in the DDB (Byte Count & Checksum) - and is executed from there:

```
DECB MSB.   ;RLSE SWAP BUFFER IF NEC.
EMT   43    ;CALL FOP (44 FOR FCR)
```

7. Set Registers as follows:

   R0 = Address of the DDB
   R1 = Address of the call parameters (in DDB)
   R2 = Address of the user File-block
   R4 = Address of the driver

8. Prepare an EMT call to the magnetic tape OPEN routine (see section 4.7) in similar fashion to that shown in step 10 and make it.

Comments:

The file-management module called by the sequences described in the preceding paragraphs may be loaded into the Swap Buffer in the normal way. Moreover any .OPEN request is likely to result in the further call. Therefore if the general routine is itself using the Swap Buffer it cannot be classified as re-entrant. This problem does not exist if it is made resident, of course; nevertheless the other one noted under .INIT still remains - that of calls to the unprotected S.GTB subroutine for buffer allocation. The remarks made in the last paragraph of section 3.2.1.1 therefore apply also to .OPEN, especially since in its case intermediate program recalls can be made.

3.2.2.2 READ/WRITE Transfers (RWN)

By means of a .READ call, the user requests a line of data as ASCII characters or binary words, with or without formatting. As noted earlier, the line-size may be set up to satisfy the requirements of the program only. The .READ processor maintains an internal buffer which takes care of varying device provisions. At each .READ request, therefore, data from this buffer is processed and transferred to the user line as specified by the program (a byte at a time for simplicity in all modes). If the internal buffer is emptied, the driver for the device is called to perform the necessary refill and in general, this occurs immediately in readiness for the next request, even though the program's needs are currently satisfied. Similarly a .WRITE request enables the user to supply lines of data which are processed in like manner and stored internally until a full buffer can be transmitted to the device.

Calling Sequence:

The program must supply information on the line to be transferred: it therefore effectively sets up a Line-block for

this purpose. In normal usage, the block immediately procedes the line it controls as a header; this is not essential as a special format (Dump) permits its being detached. The format of the block is fully described in the Programmer's Handbook, together with details of the different types of processing available. For reference, it is illustrated at figure 3-15. Briefly, it allows the user to pass maximum and actual line-sizes and specify the type of processing required and enables the .READ/.WRITE processor to return status data when the transfer is complete. The address of the block is given as a parameter in the calling sequence as follows:

```
        MOV #LINE,-(SP)     ;PASS LINE &...
        MOV #LNKBLK,-(SP)   ;...LINK-BLOCK ADDRESSES
        EMT 2               ;CALL .WRITE (4 FOR .READ)
```

Processing:

Many of the operations needed to service either .READ or .WRITE are common to both. Furthermore concurrent input and output are highly likely. A single processor therefore handles both requests with its one entry point stored in the two corresponding slots in the MRT. The mainstream of this processor performs the common operation; it calls subsidiary routines to provide the functions unique to .READ or .WRITE. To do this and avoid the need for checking which routine is to be called every time, a co-routining technique is used as follows:

1. The function as shown by the EMT code (2 = .WRITE, 4 = .READ) upon initial entry (or the function adjusted by 4 on recall from driver action to allow for variations at this time) is the basis for the computation of an appropriate address in a branch table which is then called by a normal JSR PC,XXXX. (This provides the check for the unique routine required).

2. When the unique routine needs to return to the mainstream, it does so by JSR PC,@(SP)+. This performs as RTS PC in that processing continues from the address saved on the stack by the original JSR call and removes the address from the stack. However it is replaced by the address at which unique operations are to be resumed.

3. Similarly the mainstream effects the resumption by JSR PC,@(SP)+ and appropriate branches in both the mainstream and unique routines complete any looping necessary. Thus after the first check, the correct sequence is maintained naturally though the address

saved on the stack and no further checking is necessary. Departures from this sequence are accomplished, in general, by adjustment of the saved address

The mainstream process is illustrated in figure 3-16. It basically follows the outline given below:

1. Save the user Line and Link-block addresses in the DDB and remove from the stack

2. Perform the following checks on the request validity. For failure in either case call a fatal error (F010) - see chapter 7

   a. Function and mode acceptable for the device associated with the dataset, using data in the driver interface table (see section 3.3.1) (e.g. A punch cannot read, nor can a line printer handle binary data)

   b. A valid .OPEN call previously made for all file-structured devices.

3. Prepare the DDB for the transfer:

   a. Clear Byte Count & Checksum.

   b. Verify that an internal buffer is attached to the DDB. If none, because of no .OPEN on a non-file device, claim one now and link it to the DDB (start address & size). (if none is available, recall the program via the user Link-block error address or call a fatal error (F007) if this is 0).

   c. Collect the variable pointer to the buffer saved in the DDB Driver Word Count and replace it by the buffer end-address computed from start & size.

4. Set Registers (based on data passed by the program where necessary) to show:

   R0 = Address of the Byte Count in the DDB
   R1 = Address of the next Monitor buffer byte (0 if buffer empty)
   R2 = Address of the Byte Count in the user Line
   R3 = Mode & current status (may be accumulative during the transfer)
   R4 = Address of the next byte in the user Line

R5 = (Temporary work-space)

5. Call the unique .READ/.WRITE routines as indicated in the previous paragraph in order to complete initialization and collect the first data byte.

6. For all data modes, count the byte and accumulate a checksum in the DDB (the latter for simplicity, when it is only really necessary for formatted binary). For ASCII modes in particular, strip or generate bit 7 (the parity bit) within the byte as required; for formatted versions, set an End-of-Line (EOL) switch (by clearing the user Line Byte Count) if the byte is a line-delimiter. Recall the unique routine to complete the processing & storage of the byte.

7. If now at the end of the Monitor buffer, check if an End-of-Data (EOD) occurred at the previous device transfer. In this case, set appropriate status flags in the user Line-block and recall the program as in step 11. Also clear the variable pointer to force subsequent attempts to fill or empty the buffer to make this check again.

8. Otherwise save the current pointer values in the DDB or user Line-block and prepare to call the driver. In particular, for a file-structured device as defined in chapter 4, determine the next device block (the routine to do this is embedded; its description however has been included in context in section 4.5.2). Call the driver via S.CDB and return to the program until done.

9. On recall, dequeue the driver via S.CDQ. Check for device parity failure or EOD (as shown by an unexpired word count returned by the driver) and set DDB flags accordingly. (Again if a file-structured device is being used, some clean-up and checking is necessary; this is also covered in section 4.5.2). Restore saved pointers by returning to step 4. Force return after initialisation to the next step to omit further byte transfer at this stage

10. If device action is not yet required, or has been completed, check for the end of the user Line based upon either the EOL switch mentioned in step 6 or upon the Byte Count in the DDB being equal to that supplied by the program (unformatted modes only). If not seen, recall the unique routines for further checking and collection of the next byte as necessary, with a return at step 6.

11. When the previous step shows the transfer is complete, set status information and byte counts into the user Line-block. Save the buffer pointer in the DDB and clear its Busy Flag. Restore the saved program Registers from an otherwise cleared stack and recall the program.

From the above sequence it can be seen that the unique routines are called for four purposes and by definition these differ for .READ and .WRITE :

a. Initialisation on first entry and collection of the first byte

b. Completion of individual byte processing

c. Reinitialisation after device action

d. Further EOL checking and collection of subsequent bytes

For .READ, these operations are outlined below and are further detailed in figure 3-17:

1. For first initialisation, the following tasks are carried out:

   a. Set the user Line Byte Count to the maximum size; if none is given, return to the program with an invalid line error.

   b. If the internal buffer is empty (as shown by R1 = 0), return to step 8 in the mainstream to cause a fill by the driver (or to see an EOD set in a previous .READ)

   c. For formatted binary modes, look for the first non-0 word (with intermediate buffer refill if necessary). If this is not 1 as required by the format, recall the program with format error indication. Otherwise store the Byte Count from the data in the user Line-block.

   d. Collect the first data byte for processing.

2. No further byte-processing is required for binary data, so store the byte and recall the mainstream.

For ASCII data, however the following additional steps must be taken before this occurs(1):

a. In all modes, ignore nulls.

b. Ignore RUBOUTS in formatted modes and check parity, if necessary returning the appropriate error indication to the program.

3. On return from driver action, the mainstream can be recalled immediately for EOL checking unless the line has already been filled in normal formatted modes and excess data is being discarded. In this case, a line pointer reset on the basis of "line start + number of bytes read" would corrupt the area outside the line proper, hence reset the pointer to the last actual byte of the line before recall

4. If the Byte Count in the DDB and in the user Line are equal on return for additional EOL checks, the EOL switch is set for formatted binary operations. In this case, verify the Checksum and transmit an appropriate error to the program as necessary; return to the mainstream to recheck for buffer end. In all other cases of recall from the mainstream, it is possible in formatted modes that no further room exists in the user Line as shown by the maximum Byte Count. (because of no ASCII line delimiter or too long a binary line). Hence check for this and if there is no problem, return to step 1 to collect the next byte. Otherwise return an invalid line error to the user and:

a. In Normal formatted modes, set up to overlay the last line byte and continue from step 1.

b. In Special modes, recall the program (For formatted binary, the Byte Count is set to show the full line size to indicate how many more bytes must still be read)

For .WRITE, the unique operations are as follows (see also figure 3-18):

----

1. If an EOL is seen in formatted ASCII modes, the mainstream check on the end of the Monitor buffer is in fact omitted. If the device is a terminal the operator may not be ready to input more data as needed to refill the Monitor buffer. Hence in order to pass the already complete data to the program as quickly as possible, the normal practice of maintaining data in the buffer at all times may be overlooked in this case.

1. Initialization on first entry requires the following actions:

   a. If no Byte Count is provided by the program, return with an invalid line error indicated.

   b. If the buffer has just been claimed, zero it completely

   c. If an EOD was seen on a previous .WRITE, recall the program to show this.

   d. If a formatted binary transfer is requested, store an interline gap (8 nulls) followed by 1 (for binary mode) and the user Byte Count in the internal buffer, provided sufficient room still remains. If not, recall the mainstream at step 7 to empty it first. Also initialize a checksum to include the data just stored.

   e. Collect the byte and for formatted ASCII modes, remove any parity bit. Save the user Line Byte Count (see 2 below).

2. For all modes but formatted ASCII, no further byte-processing is needed; merely store the byte and return to mainstream. Otherwise the following checks must be carried out before this:

   a. If an EOL terminator was seen, check if the user line is empty. If not and the mode is Normal, restore the saved user Line Byte Count and continue.

   b. Also if at line-end, force output to a terminal device.

   c. Provided there is room in the internal buffer, follow horizontal TAB with RUBOUT and vertical TAB or form-feed with nulls. Otherwise force output first.

3. To reinitialize after a device transfer, clear the buffer unless an EOD was seen, in which case recall the program as in 1. For the formatted ASCII modes, perform the actions noted in 2(c). Otherwise return to the mainstream.

4. If a formatted binary line has been completed, store the Checksum in the buffer and return to check if the buffer is now full for output. Otherwise collect the next byte and repeat from step 1(e).

Comments:

The .READ/.WRITE processor returns to the program when held up for a device transfer as shown at step 8 of the mainstream sequence. The normal requirement that the program perform a .WAIT before accessing the line therefore applies. In this case it also is imperative that the user Line-block is not changed before the transfer is known to be complete because of the intermediate data stored in it. On completion, it is the program's responsibility to verify the accuracy of the transfer by examination of the status returned by the Monitor.

Because of its size, RWN cannot be brought into the Swap Buffer. Although this could be effected as for other routines by an overlay process, this would mean that re-entrancy would be sacrificed making it impossible to process more than one request at a time. In the case of .READ/.WRITE this is somewhat unrealistic. The nature of the processing involved too does not lend itself to simple overlaying as it is not sequential. An alternative method might be to use the Swap Buffer for, say, the mainstream and claim a separate buffer for use by the unique routines - a feasible solution since the co-routine technique can be simply adapted to provide it. However again size, particularly with the addition needed to control the extra operation, prevents the adoption of this method. Therefore for the present, the .READ/.WRITE processor must form part of the permanently resident Monitor - a not entirely unsatisfactory constraint when most programs will probably require its services too frequently for swapping. With regard to RWN re-entrancy, it must be noted that it may also call the unprotected S.GTB routine as discussed under .INIT (section 3.2.1.1) if the program does not call .OPEN first. Again this is no problem in a single-user system; for real-time operations, the standard use of .OPEN can obviate it.

3.2.2.3 Dataset Close (CLS)

The .CLOSE call enables the user to inform the Monitor that current transfer operations within a data-set are to be terminated. Its processing includes the output of any data still in the internal buffer, any necessary shut-down action on a non-file device (e.g. Punch trailer) or directory clean-up on a file-oriented device and finally the release of the internal buffer. The device driver and the DDB however remain in memory, linked to the user program. Thus operations can be resumed by a new call to .OPEN. Again, for the sake of device-independence, use of this call is recommended, even though it is only essential for file operations

(since a .RLSE call performs .CLOSE operations in other cases - see section 3.2.1.3)(1).

Calling Sequence:

The calling sequence for .CLOSE is simply:

```
        MOV #LNKBLK,-(SP)   ;PASS LINK-BLOCK ADDRESS
        EMT 17              ;CALL .CLOSE
```

Processing:

The .CLOSE processor is outlined at figure 3-19. As with .OPEN (see section 3.2.2.1), .CLOSE calls are handled initially by a general routine for all devices. This may then be followed by a call to the file-management module FCL if the device is file-oriented. The general routine uses the sequence given below:

1. Clear the call parameter from stack and get the driver address from the DDB.

2. Verify the existence of any valid data remaining in the internal buffer if the last operation performed was output (using the last value of the variable buffer pointer stored in the DDB Driver Word Count by the .READ/.WRITE processor (see last section)).

3. If such is the case and the device is file-oriented, check that no EOD has been seen - the last block has already been output in this case. For linked files (see section 4.1.1), set the last 0 link & the DECtape transfer direction.

4. Call the driver, via S.CDB to action this last output (1) and return to the calling program until done. Dequeue the driver by S.CDQ on recall.

5. Clear the Open indicator in the DDB Status and check for a file-structured device (see next para-

----

1. The last output in fact is a full buffer - the last valid data followed by nulls. On file-oriented devices, this may mean in the worst case that the last block of a linked file is all null. Although this is wasteful, it avoids the necessity of re-accessing the previous block to change its last link to 0.

graph if so)

6. For non-file devices, call the driver via S.CDB if it contains a CLOSE routine as indicated by a switch in its standard interface table (see section 3.3.1). Again return to the calling program until complete.

7. Using the standard buffer size stored in the driver's interface table, release the internal buffer to free core, via S.RLB.

8. Clear the DDB Busy Flag and take the usual system Exit.

As noted in the last paragraph, a .CLOSE call on a dataset using a file-oriented device requires a subsidiary call to a file-management module. By a similar method to that described for .OPEN in section 3.2.2.1, this is effected with appropriate data passed as follows:

R0 = Address of the DDB
R2 = Address of the FIB
R3 = Address of the driver
R4 = Original Open type code -2

In addition, bit 15 of the User Line Address in the DDB is set as a DECtape marker.

Comments:

Because Swap Buffer overlaying is again a possibility, the remarks in the last paragraph of section 3.2.2.1, also apply to the .CLOSE routine.

3.2.3 Random Access I/O (BLO)

It will be shown in chapter 4, that a user can reserve an area of physically adjacent blocks upon a bulk-storage medium by requesting the allocation of a contiguous file. The .BLOCK call is provided as a means whereby he can then access the area randomly. The user identifies a physical block by its relative position within the area. The .BLOCK processor transfers the required block in a specified direction to or from the normal Monitor buffer and then allows the user to process his data directly in the buffer, rather than cause a further unnecessary move to or from the user's own area. Before .BLOCK is used, the user must correctly OPEN the file in order that the Monitor can establish its buffer: .OPENU allows both read and write; .OPENI allows in-

put only. On completion .CLOSE must also be called.

Although it was seen in section 3.2.2, that the normal READ/WRITE level can be used for sequential processing of any type of file, the same does not apply to the use of .BLOCK for random access. The nature of sequential files on bulk storage devices, described in section 4.1.1, would make the use of .BLOCK too lengthy a process that for the present it has not been implemented. Random access, of course, on other devices is meaningless; while the ability to use .BLOCK perhaps as a means of sequentially processing data within the Monitor buffer might be useful, the same effect can be obtained by .TRAN with very little extra effort by the user. Hence the .BLOCK processor only accepts calls upon contiguous files and rejects all others as invalid.

Calling Sequence:

.BLOCK actually provides three functions. For input, it is sufficient that the user requests the block and the information on the block is returned so that he can process its content. For output, however, the user must supply the data before it can be transferred and to do this he must know the buffer location and size in advance. Hence in addition to READ and WRITE a block, a GET operation is provided for the purpose. To indicate the function and allow the Monitor to return information, the user supplies a BLOCK-block as illustrated in figure 3-20 and described in more detail in the Programmer's Handbook.

This then means a calling sequence as follows:

```
        MOV   #BLKBLK,-(SP)   ;PASS BLOCK-BLOCK &...
        MOV   #LNKBLK,-(SP)   ;...LINK-BLOCK ADDRESSES
        EMT   11              ;CALL .BLOCK
```

Processing:

The sequence of operations in the .BLOCK processor is quite straightforward; thus no illustration is necessary. Basically the following steps are taken:

1. Collect the address of the user BLOCK-block and clear both arguments from the stack by moving the saved program Registers down.

2. Move the address of the Monitor buffer into the BLOCK-block. If no buffer is set up, no .OPEN has been done. Return the appropriate error flag in the BLOCK-block status and exit as in step 7.

3. Extract the buffer size from the standard driver interface table (see section 3.3.1) and store it in the BLOCK-block. If the requested function is GET, processing is now complete, so exit as in step 7.

4. Reject the .BLOCK request by returning the appropriate error flag in the BLOCK-block status for any of the following reasons, again taking an exit as in step 7:

   a. The device is not file-structured, as shown by an indicator in the driver interface table.

   b. The function is not READ or WRITE.

   c. The file has been incorrectly opened, based upon the Open type code saved in the FIB (see section 4.3.2), i.e. .OPENU has not preceeded READ or WRITE, or .OPENI for READ only.

   d. The block requested is outside the range of the file as stored in the FIB.(where block 0 of the file is the first and the last is thus Length-1)

5. Compute the absolute block from the start stored in the FIB and set up the DDB for the transfer specified.

6. Call the driver to effect the transfer via S.CDB and return to the user until done. On return, dequeue the driver via S.CDQ and return any parity error in the BLOCK-block status.

7. Clear the DDB Busy Flag and take the normal System Exit.

Comments:

Because of the intermediate return, the user again must call .WAIT before attempting to process the buffer and it is also his responsibility to check the returned error status to ensure the transfer required has been satisfactorily executed. The .BLOCK processor is always reentrant, whatever its location.

3.2.4 Special Operations

Two further I/O functions are provided by the Monitor to cater for the fact that while device-independence is gener-

ally feasible for normal data-transfers, certain devices require special controls which the program itself must be allowed to manage if the user wishes. Thus the .SPEC call is a general means by which the user can exercise such controls and this is described in section 3.2.4.1. It can still be used in the device-independent environment because it is effectively a NOP if the device in use does not recognize the control specified. However, in order that the user can avoid unnecessary Monitor calls and for other situations where the program might need further information on the assigned device, the .STAT call, outlined in section 3.2.4.2, is available.

3.2.4.1 Special Functions (SPC)

As noted above, the .SPEC call is a means whereby the user can request a device-driver to perform a special function which does not generally require a data-transfer in the normal sense covered by the processes in the previous sections. Magnetic tape Rewind is a typical example. Potentially any driver may contain a Special Function routine, but this is only actually included when meaningful, either because the device itself can expect the function or it can be simulated, e.g. although it is not currently implemented, a Rewind on DECtape is not out of the question. Thus the .SPEC processor only calls the driver if the routine is seen to be present, as indicated in the standard interface table in the driver (see section 3.3.1,1). However the routine's presence does not necessarily mean that the driver can always satisfy the function; while the magnetic tape driver, for instance, can accept a Rewind call, it will not understand some special operation to control a display. The .SPEC processor in its general way cannot discriminate. It is therefore the responsibility of the driver to ignore functions it does not recognize (see section 3.3).

Calling Sequence:

The special function itself is defined by a code, assigned as the need arises from the range 0-255. This code is passed as a call paramenter. In some cases, the function alone defines the user's requirements; in others, additional information is needed to support the function or perhaps the Monitor must have the opportunity to return status data. There are therefore two possible calling sequences, the relevant one in each case being defined by the specific function:

1. If only the function needs to be passed, this forms the argument:

```
MOV #CODE,-(SP)     ;STATE THE FUNCTION
MOV #LNKBLK,-(SP)   ;PASS LINK-BLOCK ADDRESS
EMT 12              ;CALL .SPEC
```

2. If support information must also be supplied, the program sets up a Special Functions Block, illustrated at figure 3-21. This contains the code for the function basically and allows a variable length work-space as necessary. In this case, the address of this block is the argument:

```
MOV  #SPFBLK,-(SP)  ;PASS SPF-BLOCK &...
MOV  #LNKBLK,-(SP)  ;...LINK-BLOCK ADDRESSES
EMT  12             ;CALL .SPEC
```

Before the call is made, the dataset must of course be initialized by .INIT to load the driver into memory. In general, however, it should not follow a .OPEN unless this has been negated by .CLOSE.

Processing:

The .SPEC process needs no illustration: it merely follows the simple steps listed below:

1. Save the function argument in the DDB (User Line Address) and clear both arguments from the stack.

2. If the high-order byte of the function argument is 0, the user has passed only the code: this follows from the allotted code range and the fact that an address must be greater than 400. Hence check this byte. Move a code-only argument to Byte Count in DDB and replace it in the User Line Address word by the address of its new location.

3. Extract the driver address from the DDB and verify the presence of a Special Functions routine in the driver from the flag in its interface table. If none, exit as in step 5.

4. Otherwise call the driver via S.CDB and return to the program until done. On return, dequeue the driver via S.CDQ.

5. Clear the DDB Busy Flag and take the normal System Exit.

Comments:

If the driver is called, the operation at step 2 above means that the User Line Address in the DDB always contains an address pointing to a Special Functions Block as far as the driver is concerned - even if this merely consists of a single word elsewhere in the DDB. The driver can thus assume a standard procedure for extracting the code and checking both that this is appropriate to itself and that the required number of words follow (the high-byte 0 noted in step 2 again gives to correct result).

.SPEC is fully reentrant regardless of its location & usage. However device action may be needed; hence for safety the program should .WAIT before proceeding in many cases.

3.2.4.2 Device Status (STT)

As stated earlier, the .STAT call allows the user program to obtain information on the device a dataset is actually using for a particular run, after it has been initialized in the usual way by .INIT.

Calling Sequence:

The call is made by:

```
        MOV #LNKBLK,-(SP)   ;PASS LINK-BLOCK ADDR
        EMT 13              ;CALL .STAT
```

Processing:

The .STAT processor merely extracts the relevant data from the driver associated with the DDB by reference to its interface table (see Section 3.3.1). The data is returned on top of the stack. However, three items are supplied:

1. Driver Facilities Indicator (see Section 3.3.1 for detail)

2. The name of the device in packed radix-50 format

3. The size of the buffer deemed the device-standard

Since only one argument is passed, the .STAT routine contains an appropriate stack-shuffle operation. The Normal System Exit is taken to recall the program.

Comments:

The .STAT routine is also fully re-entrant. Moreover because no actual driver action is necessary, the required information is immediately available to the program when recalled. As in other cases where data is returned on the stack, it is the program's responsibility to clean-up after the data has served its purpose.

3.3 Device Drivers

Much has been said in the previous sections concerning the fact that in order to provide a device-independent environment, the drivers for all devices present a standard interface to the Monitor routine servicing a program I/O request. The purpose of this section is to summarize the features of the interface mainly for reference. Appendix G of the Programmer's Handbook provides a fuller description.

Sections 3.3.1 through 3.3.3 discuss the two main parts into which every driver can be divided: a standard interface table, which must come first, and a package of routines to service the different types of Monitor request and the interrupts occasioned by these. Section 3.3.4 then notes a problem peculiar to the driver for the system-device. Descriptions of the currently implemented drivers are available within the Device Driver Package document. The driver for the ASR-33 Teletype is described in Appendix A.

3.3.1 Driver Interface Table

The first part of every driver is a standard table which can be referenced by every Monitor routine, firstly in order to determine the capabilities of that driver and secondly to access the appropriate service routine via the Driver Queue Management subroutines, S.CDB or S.CDQ (see Section 3.1.2.4)

The format of the table is illustrated at figure 3-22. In effect it is in two parts, the first of which must appear in all drivers, the second is necessary only in drivers for file-structured devices as described in chapter 4. The significance of the entries is as follows:

1. Busy Flag - is a 0 when the driver is available for use and is reset by S.CDB to the DDB address for the dataset being serviced while the driver is busy (see section 3.1.2.4). It enables the driver to access the transfer control parameters passed by the calling routine. In general, the driver does

not clear this flag - this is done by S.CDQ (however see section 3.3.4)

2. General Facilities = is a series of bit-switches to ensure that the driver is only called upon to provide services of which it is capable, (bit set to 1 in each case):

bit 0 = multi-dataset support

this bit is checked by the .INIT routine to prevent the assignment of a single-purpose device like a paper-tape reader to more than one dataset at a time.

bit 1 = Output
bit 2 = Input

These two bits are used widely to verify transfer direction validity (1)

bit 3 = binary
bit 4 = ascii

These two bits enable rejection of data modes which are unacceptable, e.g. binary on a keyboard.

bit 5 = Special Function routine present
bit 6 = Close routine present
bit 7 = Open routine present

these bits are used by the relevant routine (.OPEN, .CLOSE, .SPEC) to ensure that the driver is not called if it is unable to supply the service (see also next section)

3. Special Facilities = again uses bits to identify particular facets of the driver:

---

1. The corresponding bits are used in all Monitor transfer functions for ease of checking, e.g. .Write=2, .READ=4; file type codes include them, etc. In many cases they also are similarly used for device hardware control.

bit 0 = terminal device

This bit shows that the driver controls a device like the console typewriter which allows manual data-entry and therefore raises special problems noted in Appendix G of the Programmer's Handbook. In particular, .WRITE forces output at EOL even though the Monitor buffer is not yet filled.

bit 2 = multi-unit system-device

As shown under 10 below, further table entries contain bit map pointers for each unit under 1 controller. If the driver is resident, as in the case of a system-device such as RK11, all these entries must be cleared for a fresh start after a program failure. this bit shows that there is more than one entry requiring such treatment.

bit 5 = sequential magnetic tape

This bit shows that the device, though not fully file-structured as defined in Chapter 4, has nevertheless a basic file format as described in section 4.7.

bit 6 = reversible medium

This bit classifies a device like DECtape which allows transfers in either direction of tape motion and thus needs special treatment (see section 4.2.3)

bit 7 = file-structured device

This shows the device has full file capabilities as in Chapter 4

4. Standard Buffer Size = shows the number of 16-word buffer units to be claimed via S.GTB (see Section 2.4.2) for the internal buffer used in .READ/.WRITE and BLOCK level I/O.

5. Offset to Interrupt Service Routine - enables the initialization routines to set the appropriate address into the device interrupt vector when the driver is in core(1).

6. Interrupt Priorty Level - like 5, is used to set the status into the interrupt vector. Normally it is the same as the level at which the hardware interrupt occurs.

7. Offsets to Routines - are referenced by their relative position in the table whenever a Monitor routines wishes to call the relevant service via S.CDB (see section 3.1.2.4). These bytes are set to 0 if the routine is not provided - also indicated by the corresponding general Facilities indicator (see 2 above).

8. Device Name - is the packed radix-50 value for the code allotted to each device.

9. MFD Start Block # - is the basis for the file structure on a bulk-storage device and is discussed in section 4.1.2.1

10. Bit-map Pointers - enable the chaining of several datasets sharing the same unit of a bulk-storage device for output and hence the same in-core block allocation bit map as described in Section 4.1.3. There may be either 1 or 8 entries depending upon whether the device has a single unit or several.

----

1. As shown in the Programmer's Handbook, the offset can be a positive increment of up to 256 words. In the larger drivers, this may be insufficient; in that case it may point to a JMP or BR to the real start of the routine.

3.3.2 Driver Service Routines

In general any driver may be called to provide one of four services and may therefore contain the necessary routines for the purpose:

a. Transfer

b. Open

c. Close

d. Special function

The first obviously is a requirement in all drivers; the other three are present only if some actual physical device action can be effected. For instance, a console typewriter output can be opened or closed with CR/LF output, whereas on a file-structured device as shown in Chapter 4, both of these functions are in fact performed by normal transfers. Hence the appropriate Monitor routines first check the General Facilities switch, described in the previous section, before calling the driver to perform one of these three functions (see Sections 3.2.2 and 3.2.4)

All drivers under DOS are handled on the interrupt system. The service routines therefore merely initiate the relevant device action, using the address stored in their first word (see previous section)

The routines extract control data supplied by the Monitor routine in the DDB for the dataset requiring the service and transmit it as necessary to the device control hardware. As noted in Section 3.1.2.4, they may use Registers freely for this process. They then enable the device interrupt facility and return to the Monitor routine, to await the first interrupt using the address saved on the stack by the JSR call to S.CDR.

The individual nature of each of the four routines is discussed in the Programmer's Handbook, Appendix G.

3.3.3 Interrupt Servicing

It follows from the previous section, that every driver must contain a routine to service the enabled interrupts. The primary purpose of this routine is to ensure that the data is transferred to or from the device in the form expected by the calling Monitor routine as satisfactorily as possible and to allow for the handling of any error conditions which may prevent this. The provisions of the device hardware of

course determine the amount of supplementary work each driver has to do in order to meet this purpose. Again Appendix G of the Programmer's Handbook discusses the possibilities in more detail.

Generally, though, at each interrupt the driver transfers data between memory and the device allowing for the fact that the Monitor can only recognize ASCII or binary. If the driver needs Registers, it is responsible for safeguarding their content on the interrupt entry. If further interrupts are needed to complete the transfer, the driver takes normal RTI exits to await them. When the whole transfer is done, the driver recalls the Monitor routine through the Completion Return set into the DDB, with any necessary status information placed in the DDB (parity failure flag in Status; incomplete transfer in Driver Word Count). As shown in Section 3.1.2.4, the Monitor routine expects R0 to contain the address of the DDB just serviced and the contents of all Registers at the final interrupt to be saved on the stack - the driver can use the Monitor's S.RSAV sub-routine to effect this (see Section 2.4.1). The driver may disable its interrupt facility before the recall; it does not however adjust the processor priority level or clear the Busy Flag in its interface table. The Monitor routine calls S.CDQ to perform these operations.

3.3.4 System-device Drivers

As noted in the introduction to Chapter 2, the driver for the system-device is part of the permanently resident Monitor. Basically this driver is no different in function from that servicing the same device when this is used as a subsidiary peripheral in a system based on another similar device. It is called in the same way with its information set into a DDB addressed by the content of its own Busy Flag, even though that DDB belongs to the System as noted in Section 2. Moreover system requests take their place in any driver queue like all the rest.

Nevertheless in its privileged usage it does have an extra responsibility. If a transfer cannot be accomplished because of some hardware failure, the ordinary driver in most cases calls for some diagnostic message. As Chapter 7 will show, this requires the loading of a Monitor routine to control the printing. In the interim the driver's Busy Flag remains set, as it is still effectively servicing the transfer request. For the driver of the system-device, this cannot be: if this is still seen to be busy, the request to load the Diagnostic Print routine will be queued to wait its turn. To circumvent this, the driver must itself clear its Busy Flag before requesting the error message. If necessa-

ry, where the error can also be rectified by operator action and resumption may then be requested, it must also save and restore the flag content as appropriate. (In practice, this special addition is included in the standard driver as a conditional assembly option - see section 8.1)

LNKBLK:

ERROR RETURN ADDRESS (no buffers)
Ø or Set to DDB ADDRESS by .INIT
DATASET LOGICAL NAME *
DEFAULT DEVICE UNIT # | # OF WORDS TO FOLLOW
Default DEVICE NAME *
EXTENSIBLE TO PROVIDE SWITCH SPACE IF THE COMMAND STRING INTERPRETER IS BEING USED (Size indicated in "# of Words to Follow")

* = Packed in Radix-5Ø format

Figure 3-1: USER LINK-BLOCK

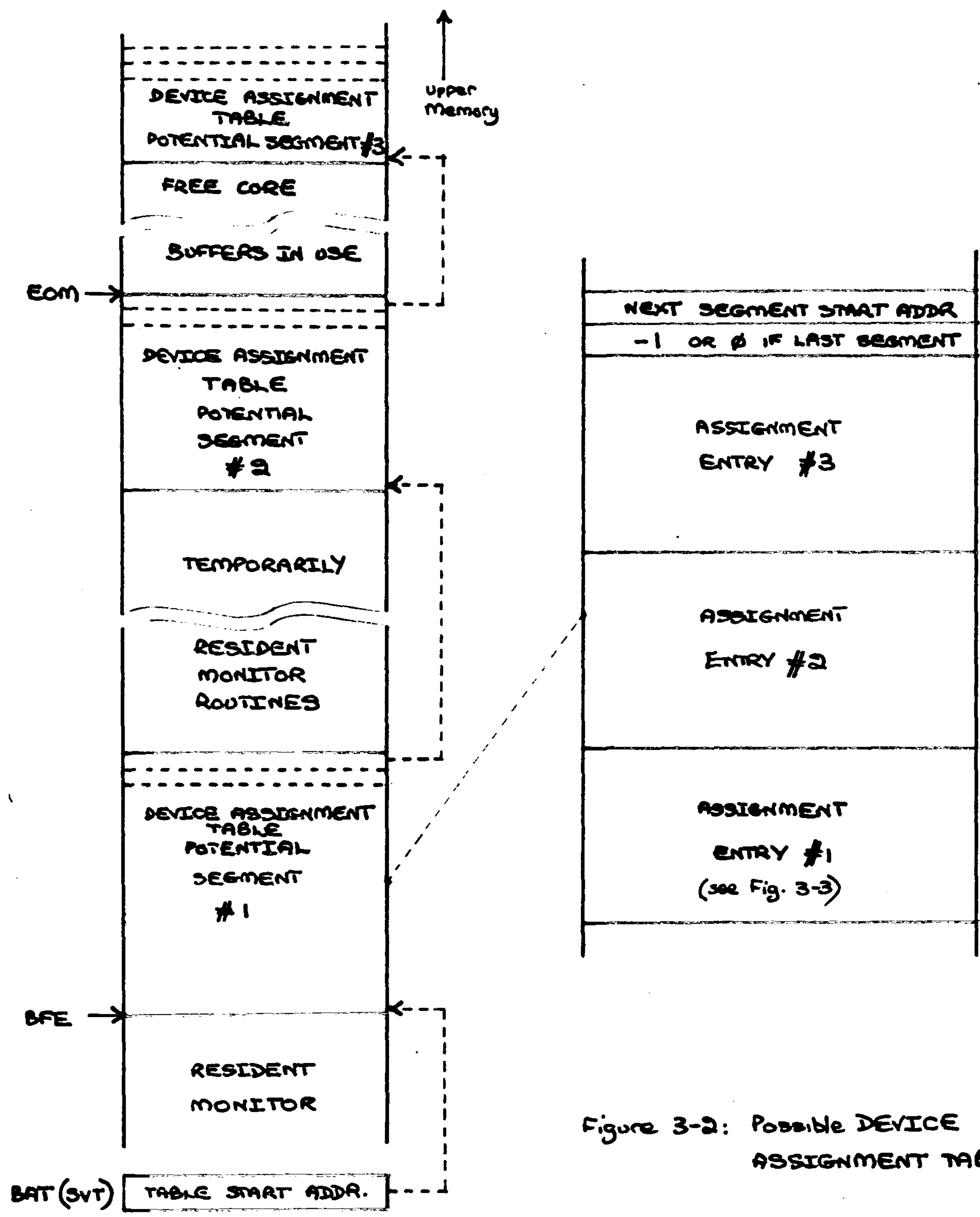


Figure 3-2: Possible DEVICE ASSIGNMENT TABLE

LOGICAL NAME of DATASET* concerned	
DEVICE - NAME*	
DEVICE UNIT #	# OF WORDS FOLLOWING
FILE -	
NAME*	
FILE EXTENSION*	
USER IDENTIFICATION CODE	

* = PACKED in Radix-50 Format

Figure 3-3: DEVICE ASSIGNMENT TABLE ENTRY FORMAT

DDB:

Field	Offset	Reference
MONITOR LINK		- see Fig. 3-5
DRIVER QUEUE LINK		- see Fig. 3-8
DRIVER ROUTINE INDEX / PRIORITY LEVEL IN Q		
Associated DRIVER ADDRESS		
BUSY FLAG (Ø = Idle)	0	
USER LINE ADDRESS	2	
DEVICE BLOCK #	4	
MEMORY BUFFER ADDRESS	6	
BUFFER WORD COUNT	10	
STATUS	12	- see Fig. 3-6
COMPLETION RETURN	14	
DRIVER WORD COUNT		
BYTE COUNT		
CHECKSUM STORE		
ADDRESS of ASSIGNMENT ENTRY		- see Fig. 3-3
F.I.B. LINK		- see Fig. 4-16

Figure 3-4: DATASET DATA-BLOCK (DDB)

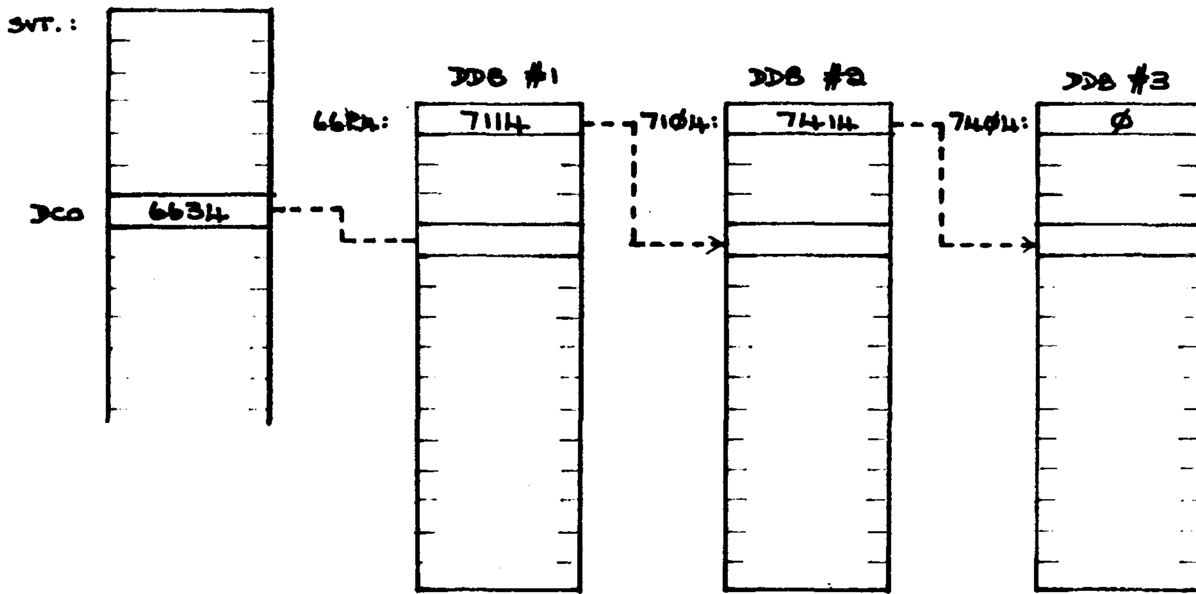


Figure 3-5: MONITOR DDB CHAIN

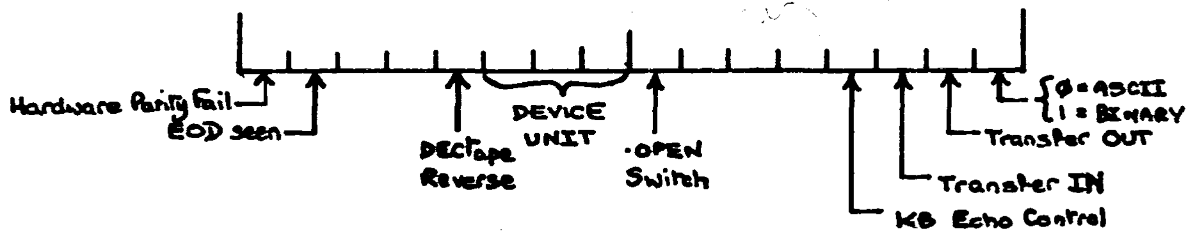


Figure 3-6: DDB STATUS WORD

DRIVER QUEUE ROUTINE:-

DRIVER DEQUEUE ROUTINE

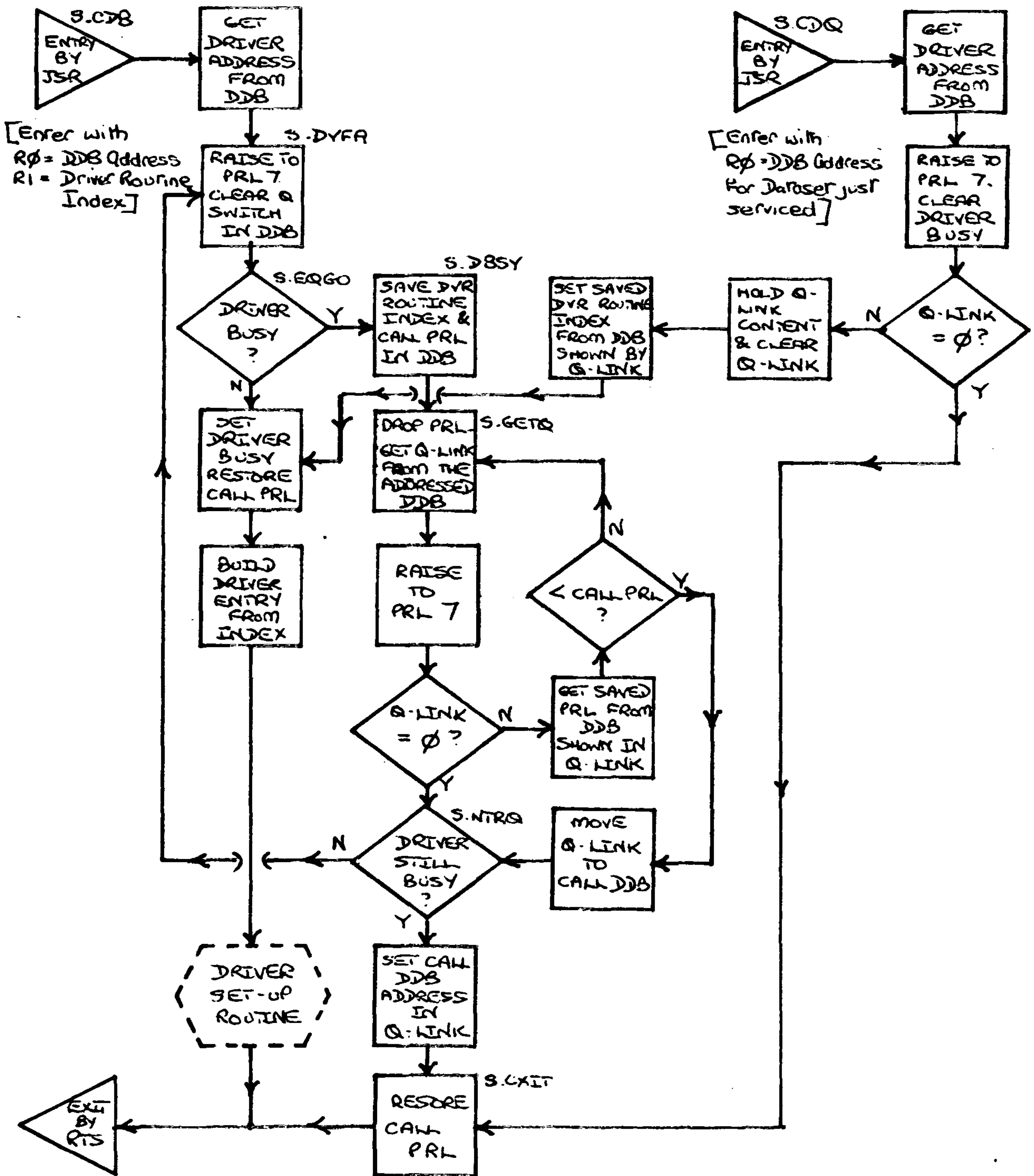


Fig 3-7: DRIVER QUEUE MANAGEMENT

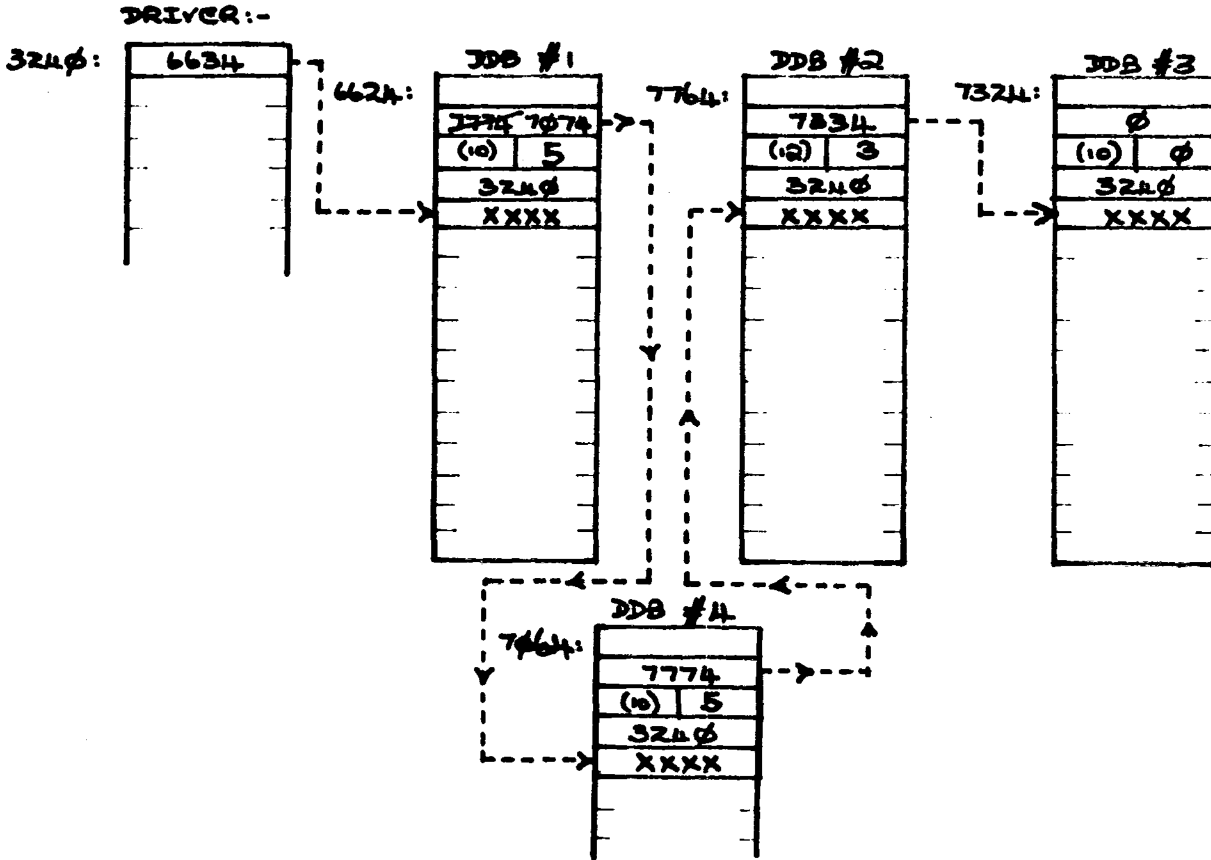


Figure 3-8: A DRIVER QUEUE

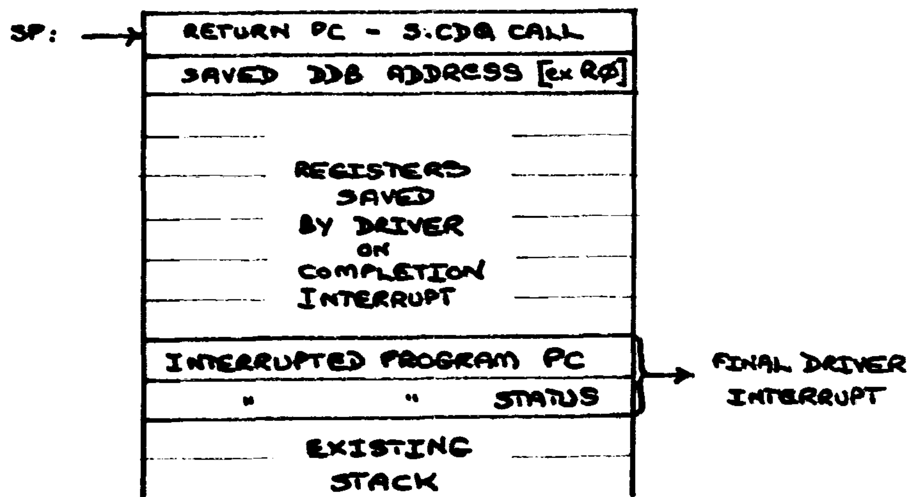


Figure 3-9: STACK STATE IN S.CDQ AFTER DRIVER COMPLETION RETURN

FLOW-CHART FOR INITIALIZATION MODULE (INR)

ENTRY FROM EMT 6 HANDLER

INR: GET LINK BLOCK & REMOVE FROM STACK

SAVE CALL ADDR. LEST ERROR

IN.REC: GET START OF DDB CHAIN (ex SVT)

DDB ALREADY LINKED ? (N / Y)

IN.CHC: DDB IN CHAIN ? (N / Y)

DRIVER ATTACHED ? (N / Y)

F.I.B ATTACHED ? (N / Y)

IN.RLM: DETACH FROM BIT-MAP CHAIN IF NEC.

IN.RLF: RELEASE FIB BUFFER

CLAIM BUFFER FOR DRIVER

GET DRIVER SIZE AND DISK LOCATION

IN.GO: CLAIM BUFFER FOR DDB

ANY AVAILABLE ? (N / Y)

LINK TO MONITOR DDB CHAIN

IN.CD: CLEAR DDB BUFFER

IN.RLL: RESET POINTERS

IN.RLD: RELEASE DATA BUFFER IF ANY

ANY AVAILABLE ? (Y / N)

DEVICE DRIVER ON DISK (Y / N)

IN.INR: PRINT A003 ERROR

SET UP AS EXIT ADDRESS

IN.NOA: ERROR ADDRESS IN LINK BLOCK ? (Y / N)

F7 ERR.

LINK TO LINK BLOCK

GET LOGICAL NAME FROM LINK-BLK

IN.GAT: GET D.A.T SEGMENT START

IN.LDG: PREPARE DDB FOR DRIVER LOAD

SAVE REGS & SET DUMMY LINK-BLK ON STACK

CALL SYSTEM DRIVER TO DO LOAD

IN.LDD: GET DDL ADDRESS ENTRY FOR DEV.

CLEAR DDB

CALL WAIT UNTIL DONE

IN.DVL: ALREADY ASSIGNED TO DDB IN CHAIN ? (Y / N)

SINGLE USER DEVICE ? (N / Y)

DVR. ALREADY IN CORE ? (N / Y)

IN.CDL: DEVICE IN DDL ? (N)

IN.DVS: SET DDL POINTER

IN.CHL: DEVICE IN LINK BLOCK ? (N / Y)

ANY ENTRY (Y / N)

DEVICE & CONSOLE ? (N / Y)

SET UP INTERRUPT VECTORS

IN.INC: ATTACH DRIVER TO DDB

STILL NO DRIVER (N / Y)

ADJUST IF 'PT,' STORE UNIT FROM DAT OR L.B.

IN.WNT: TAKE SYSTEM EXIT

DEVICE GIVEN IN DAT (N / Y)

END OF TABLE ? (N / Y)

IN.CAT: END OF DAT SEGMENT ? (N / Y)

IN.DFD: REMEMBER FOR LATER IN DDB.

ENTRY THIS DATASET ? (Y / N)

IN.CAN: MOVE TO NEXT ENTRY

Figure 3-10.

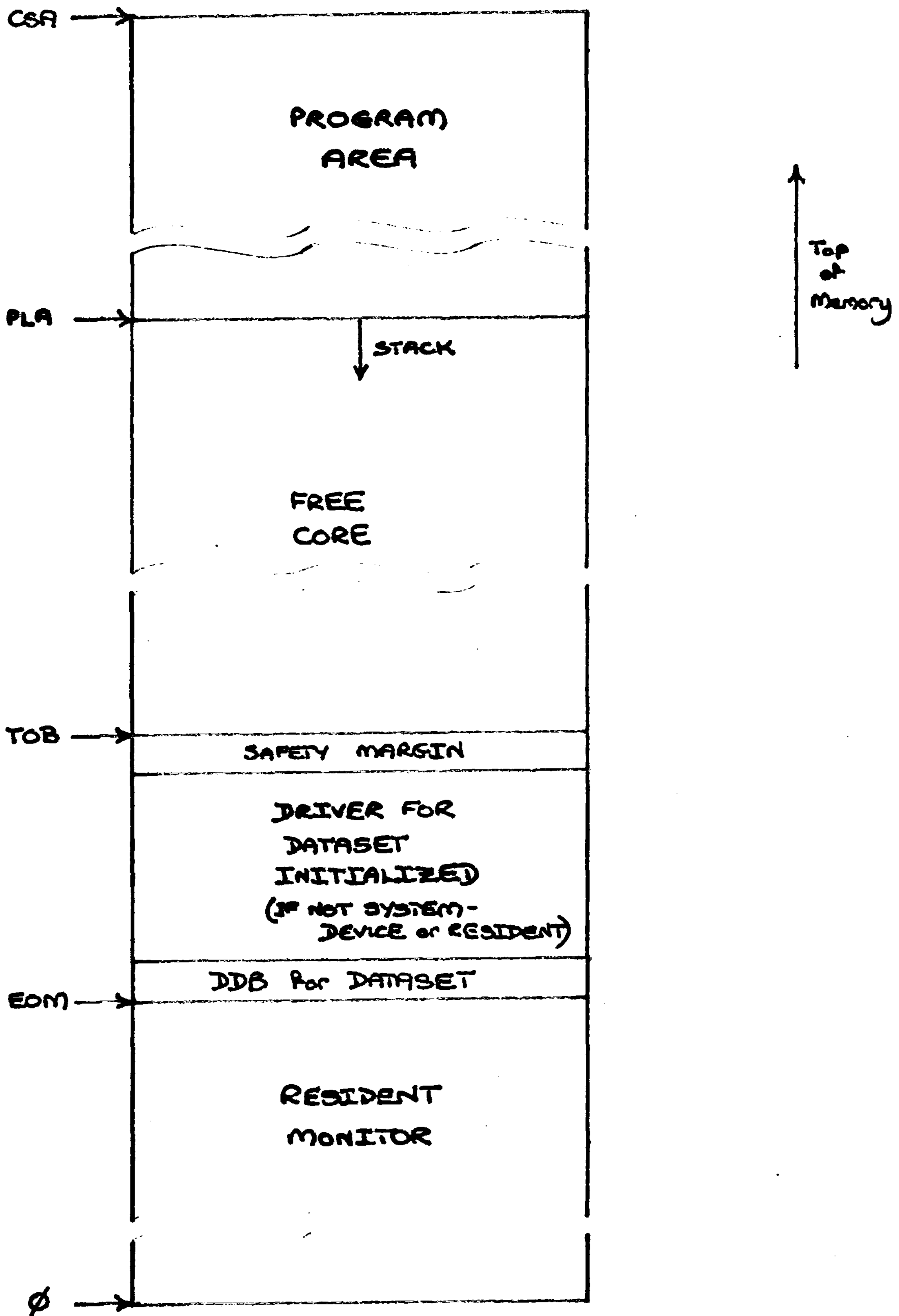


Figure 3-11: Memory after Dataset Initialization

TRABLK:

DEVICE BLOCK #
MEMORY START ADDRESS
WORD COUNT (positive)
FUNCTION / STATUS
OF WORDS NOT TRANSFERRED

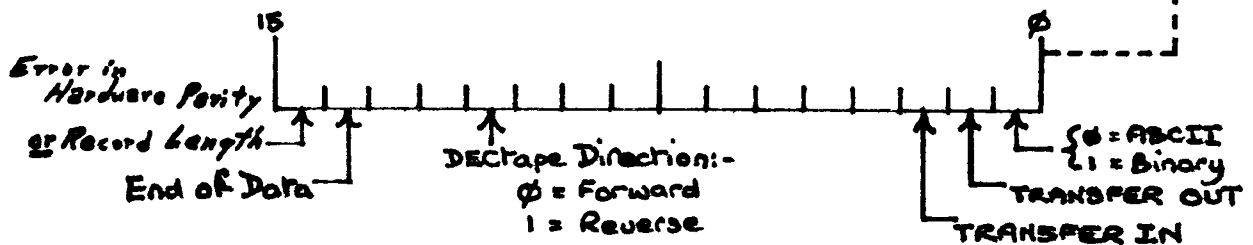
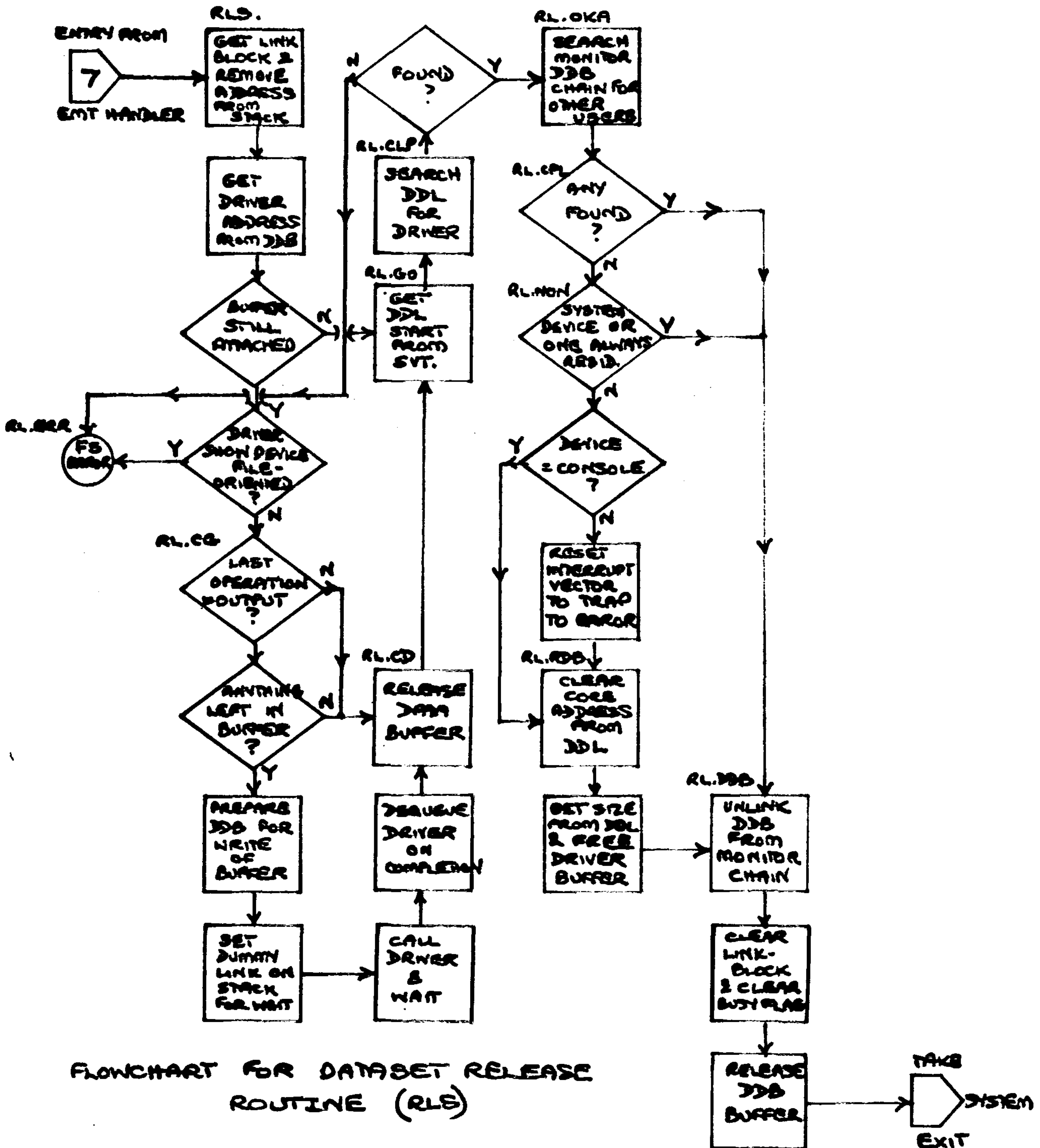


Figure 3-12: TRAN-block. Format.



FLOWCHART FOR DATASET RELEASE ROUTINE (RLS)

Figure 3-13.

ENTRY FROM 16 EMT HANDLER

OPN
SET PTRS TO DDB & DRIVER SAVE CALL ADDRESS FOR ERROR

SAVE CALL ARGUMENTS IN DDB & REMOVE FROM STACK

CLEAR ERROR BYTE IN USER FILE-BLOCK

OPEN CODE VALID ? (N → O.TER; Y)

O.TER
F11 ERROR

DEVICE FILE-ORIENTED ? (Y; N)

DIRECTION O.K. FOR DEVICE ? (N; Y)

OP.A
OPEN ALREADY DONE ? (Y; N)

O.OER
USER SUPPLY ERROR RETURN ? (N → F7/12 ERROR; Y)

F7/12 ERROR

SET UP AS EXIT ADDRESS

REMEMBER DIRECTION, GET SIZE FOR BUFFER FROM DRIVER

O.BER
SET UP ERROR MESSAGE

STORE IN DDB AS WORD CNT CLAIM DATA BUFFER

OP.B
ANY AVAILABLE ? (Y; N)

STORE ADDRESS IN DDB & CLEAR BUFFER AREA

STORE VARIABLE POINTER IN DDB

DEVICE FILE ORIENTED ? (Y; N)

DEVICE =MAGTAPE ? (Y; N)

OP.D
SET COMPLETION RETURN IN DDB

OPEN ROUTINE IN DRIVER ? (Y; N)

SET UP OPEN CALL

OP.XTX
OP.E
CALL DRIVER VIA S.CDB

RETURN TILL USER DONE

RE-ENTRY ON INTERRUPT

DEQUEUE DRIVER VIA S.CDQ

RESET POINTERS

OP.C
SET OPEN SWITCH IN DDB

OPEN FOR INPUT ? (Y; N)

OP.I
CLEAR VARIABLE BUFFER POINTER FROM DDB

SET UP INTERFACE FOR MTO CALL

OP.F
F.I.B ALREADY ATTACHED ? (Y; N)

CLAIM BUFFER FOR F.I.B.

ANY AVAILABLE ? (Y; N)

LINK FIB TO DDB & CLEAR

OP.G
SET UP INTERFACE FOR CALL TO FILE-STRUCTURED OPEN

OP.H
SET UP APPROP. CALL IN SPARE DDB AREA

CALL NEXT STAGE

BUFFER NOW FILLED ? (Y; N)

CLEAR DDB BUFFER POINTER

TERMINAL DEVICE ? (Y; N)

MORE TO COME ? (Y; N)

SET UP FOR READ TO FILL BUFFER

SET EOD IN DDB

OP.XT
CLEAR DDB BUSY FLAG

TAKE SYSTEM EXIT

FLOWCHART FOR DATASET OPEN ROUTINE (OPN)

Figure 3-14

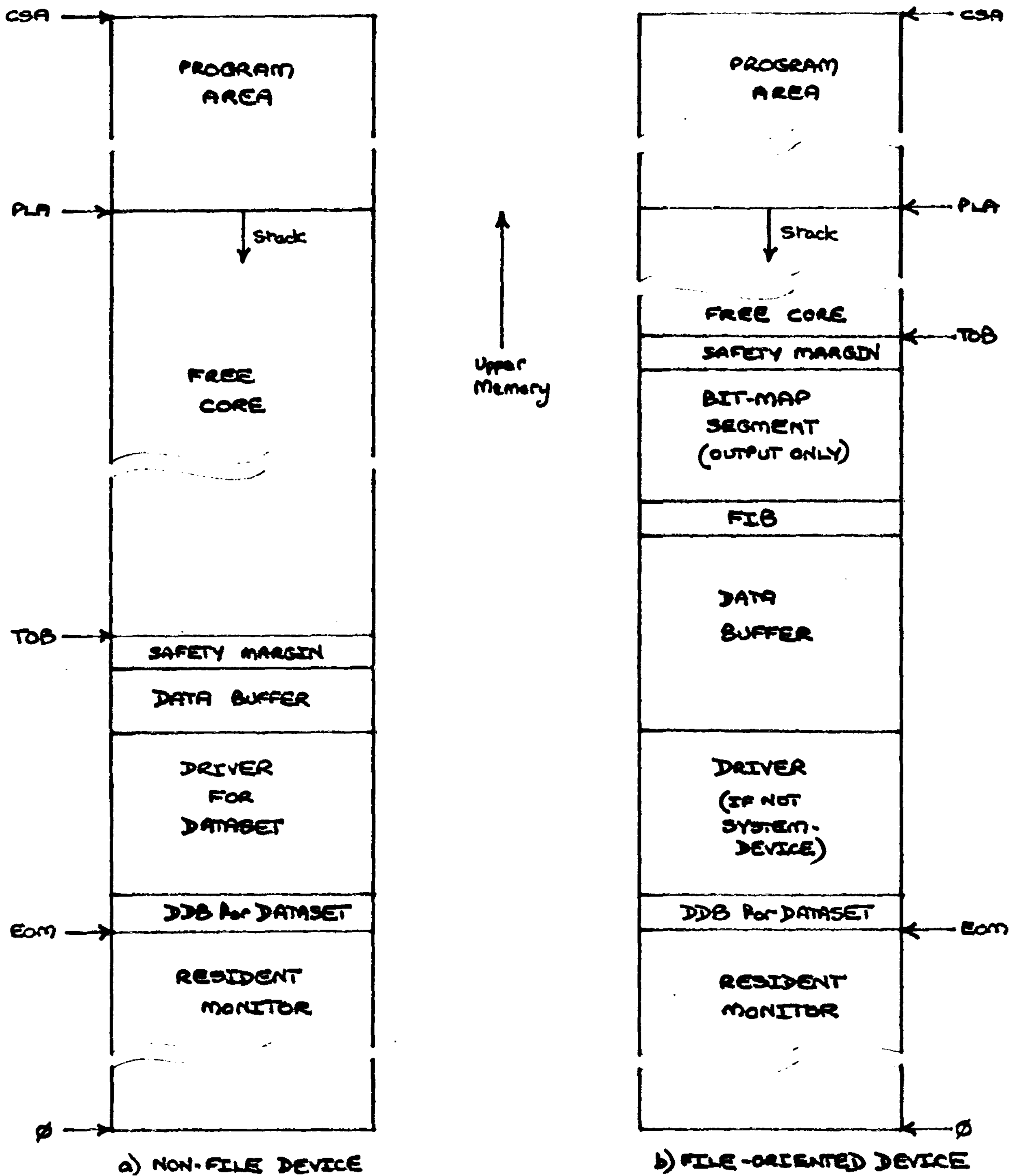
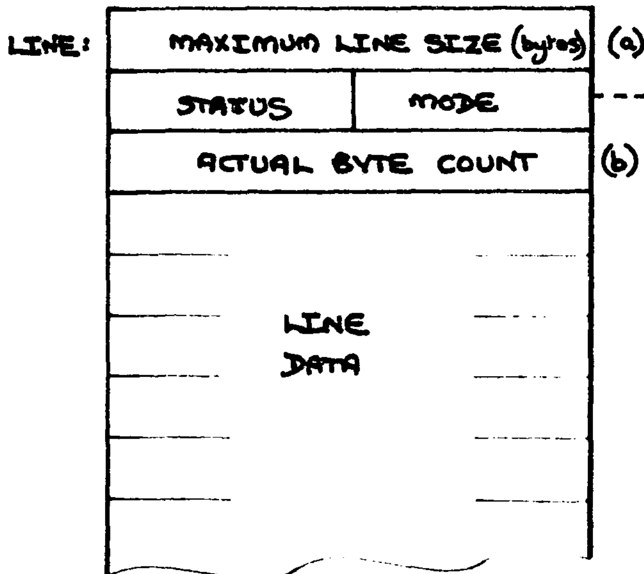
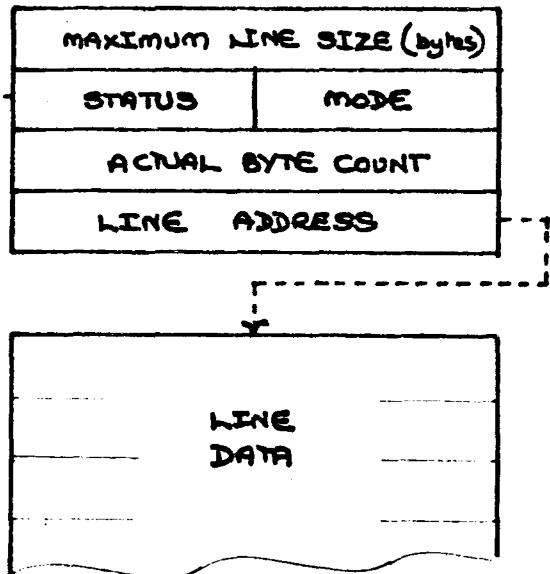


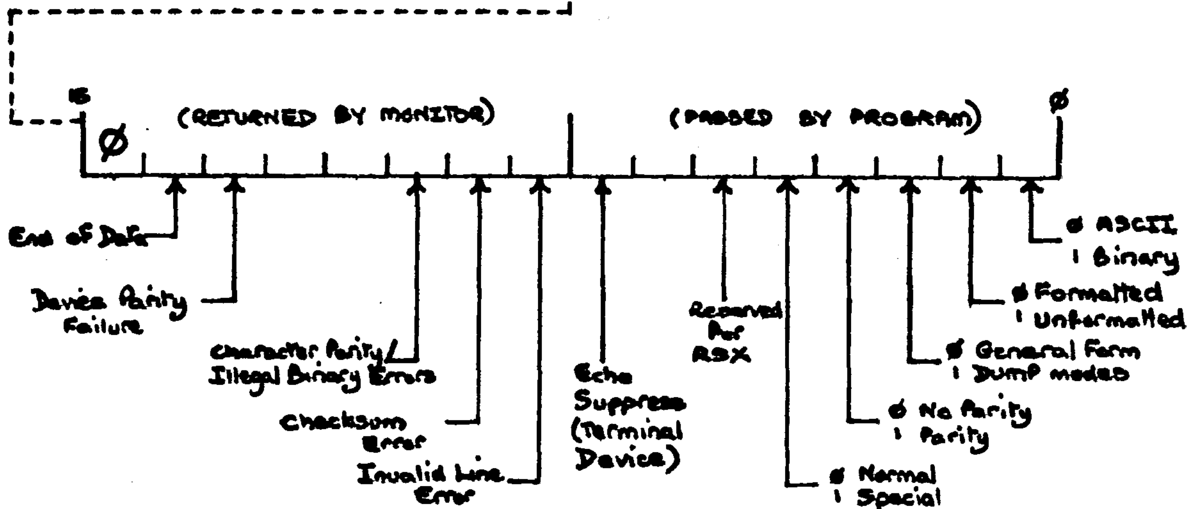
Figure 3-15: MEMORY STATE after DATASET OPEN

a) GENERAL FORM:-



b) DUMP MODES only:-





Notes:-

a) must be program-supplied for INPUT

b) must be program-supplied for OUTPUT

Valid Modes:-

Formatted ASCII Parity (Normal & Special)
Formatted ASCII Non-parity (Normal & Special)
Unformatted ASCII Parity/Non-parity (Normal)
Formatted Binary Non-parity (Normal & Special)
Unformatted Binary Non-parity (Normal)
(Dump valid in all above modes)

Figure 3-16: User LINE Format.

FLOWCHART FOR READ-WRITE PROCESSING (MAINSTREAM)

ENTER VIA 2/4 EMT HANDLER

RWN: GET CALL ARGUMENTS & CLEAR FROM STACK

RW.MC: SET PTRS TO DDB, DRIVER & USER LINE

GET GEN. MODE & TRANSFER DIRECTION

CLEAR USER LINE ERROR STATUS

MODE & DIRECTION O.K. FOR DRIVER?

RW.MER: F10 ERROR

DEVICE FILE-ORIENTED?

FILE OPENED?

OPEN DIRECTION O.K.?

RW.MCA: CLEAR CHECKSUM & BYTE COUNT IN DDB

GET BUFFER POINTER FROM DDB

STORE DIRECTION MODE & ECHO SUPPR SWITCH IN DDB

RW.MCB: ANY BUFFER ATTACHED?

CLEAR POINTER GET SIZE FROM DRIVER

SET DDB WORD COUNT. CLAIM BUFFER

ANY AVAILABLE?

RW.MCC: STORE START ADDRESS IN DDB

USER SUPPLY ERROR EXIT?

SET UP AS RETURN ADDRESS

RW.BER: F7 ERROR

RW.MCD: COMPUTE BUFFER END & STORE IN DDB

SET PTRS TO USER LINE HDR & NEXT DATA BYTE

SAVE TRANSFER MODE

BUILD ADDRESS FOR UNIQUE RD/WRT ROUTINE

RW.MCE: DO UNIQUE READ/WRITE INITIALIZATION GET NEXT TFR. BYTE (*)

RW.XF: ASCII TRANSFER?

8-BIT UNFORMATTED?

STRIP PARITY BIT

7-BIT UNFORMATTED?

BYTE = LINE TERMINATOR?

SET END OF LINE SWITCH

RW.XFA: PARITY REQUIRED?

RW.XFB: GENERATE PARITY BIT

RW.XFC: BUILD CHECKSUM & COUNT BYTES

RW.XFD: DO UNIQUE READ/WRITE CHECKING (*)

ADJUST RETURN TO UNIQUE RD/WRT ROUTINE

SET END OF LINE SWITCH

FORMATTED BINARY?

MODE UNFORMATTED?

BYTE COUNT = USER COUNT?

RW.EC: EOL SWITCH SET?

D

RW.DX: END OF BUFFER REACHED?

RW.DXA: EOD ALREADY SEEN?

B

RW.DXB: EMPTY BUFFER RETURN USER TILL DONE RESET PTRS (SEE 3-17b)

CLEAR BUFFER POINTER. SET USER LINE STATUS

USER LINE COMPLETED?

ANY TRANSFER DONE?

RW.EXT: SIGNAL SHORT LINE IN USER LINE STATUS

A

C

RW.XT: SAVE BUFF POINTER. RETURN BYTE COUNT & OTHER ERR. FLAGS

RW.DUN: CLEAR DDB BUSY FLAG

TAKE SYSTEM EXIT TO USER

Figure 3-17 (a)

(*) : SEE Figures 3-18 & 3-19

A = RETURN POINT FOR UNIQUE ROUTINE EXIT

SUB-ROUTINES TO HANDLE DEVICE TRANSFERS DURING READ/WRITE

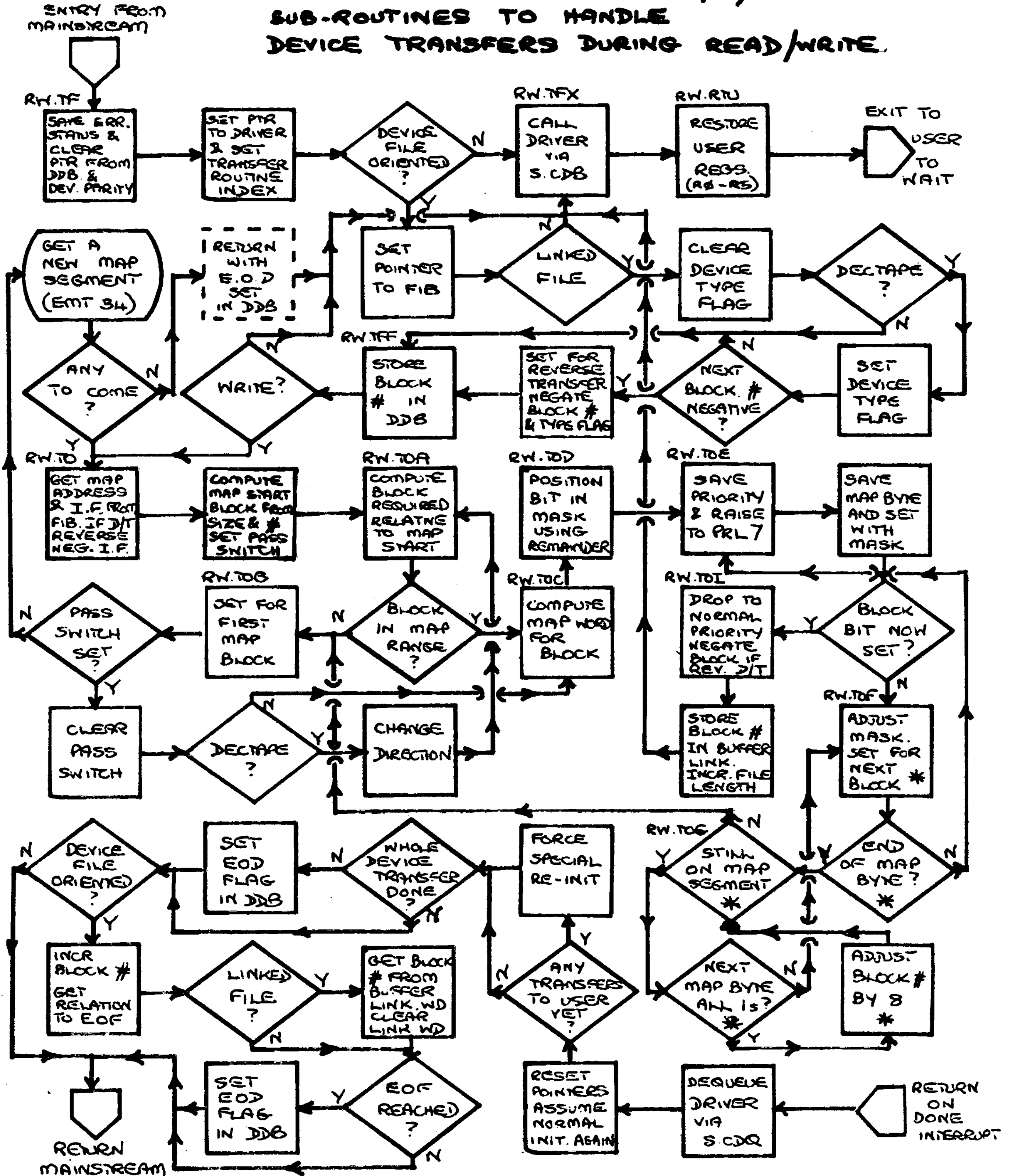


Figure 3-17 (b)

ENTRY FROM MAINSTREAM (INIT)

RD.IN
GET USER LINE BYTE COUNT

COUNT = Ø ?
EXIT TO A MAINSTREAM

ANY DATA IN BUFFER ?
EXIT TO B MAINSTREAM

FORMAT BINARY ?

ADJUST LINE PTR IN CASE SOME HDR BYTES READ

RD.IN1
READ BOTH HEADER WORDS ?

NEXT BUFFER BYTE = Ø ?

REACHED HEADER YET ?

STORE BYTE & COUNT

RD.IN2
AT BUFFER END ?

RD.IN3
B

RD.IN4
ADJUST HEADER COUNT & STORE IN USER LINE

COMPUTE HEADER CHECKSUM & STORE IN DDB

ANY DATA BYTES TO COME ?

FIRST HEADER WORD = 1 ?

RD.IN5
FLAG FORMAT ERROR IN LINE STATUS

EXIT TO MAINSTREAM C

NULL ?

ASCII MODE ?

STORE BYTE IN USER LINE

RETURN TO MAINSTREAM FOR GENERAL CHECKING

RD.A2
GET NEXT BYTE FROM BUFFER

RD.E
USER LINE FULL ?

SPECIAL MODES ?

SET COUNT TO SHOW FULL LINE SIZE

RD.XT
A
EXIT TO MAINSTREAM

RD.B1
FORMAT ASCII

PARITY O.K. ?

FLAG PARITY ERROR IN LINE STATUS

RD.C
CANCEL BYTE & DECR. COUNT

RUBOUT ?

RD.B2
AT LINE END ?

D
RETURN MAINSTREAM

RD.D
RETURN TO MAINSTREAM FOR END CHECKING

FLAG CHECKSUM ERROR IN LINE STATUS

COME BACK FOR F/B EOL ?

CHECKSUM O.K. ?

RD.A1
FLAG LINE ERROR KEEP PTR ON LAST BYTE

RE-ENTRY AFTER DRIVER TRANSFER
RN.IN

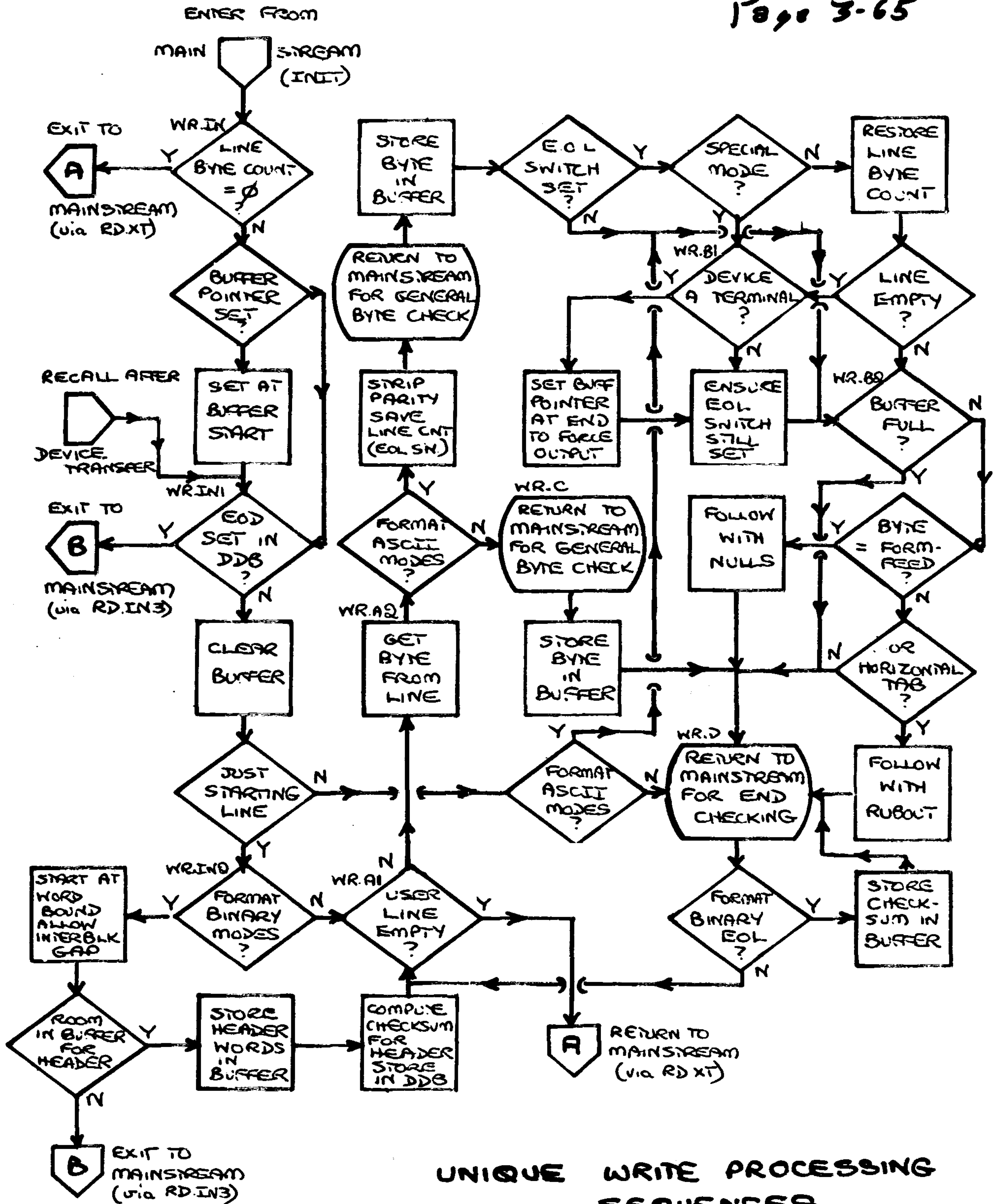
ADJUST LINE PTR TO SIT ON LAST BYTE

LINE OVERFLOW ALREADY ?

RD.IN6
AT LINE END ?

UNIQUE READ PROCESSING SEQUENCES

Figure 3-18



UNIQUE WRITE PROCESSING SEQUENCES

Figure 3-19

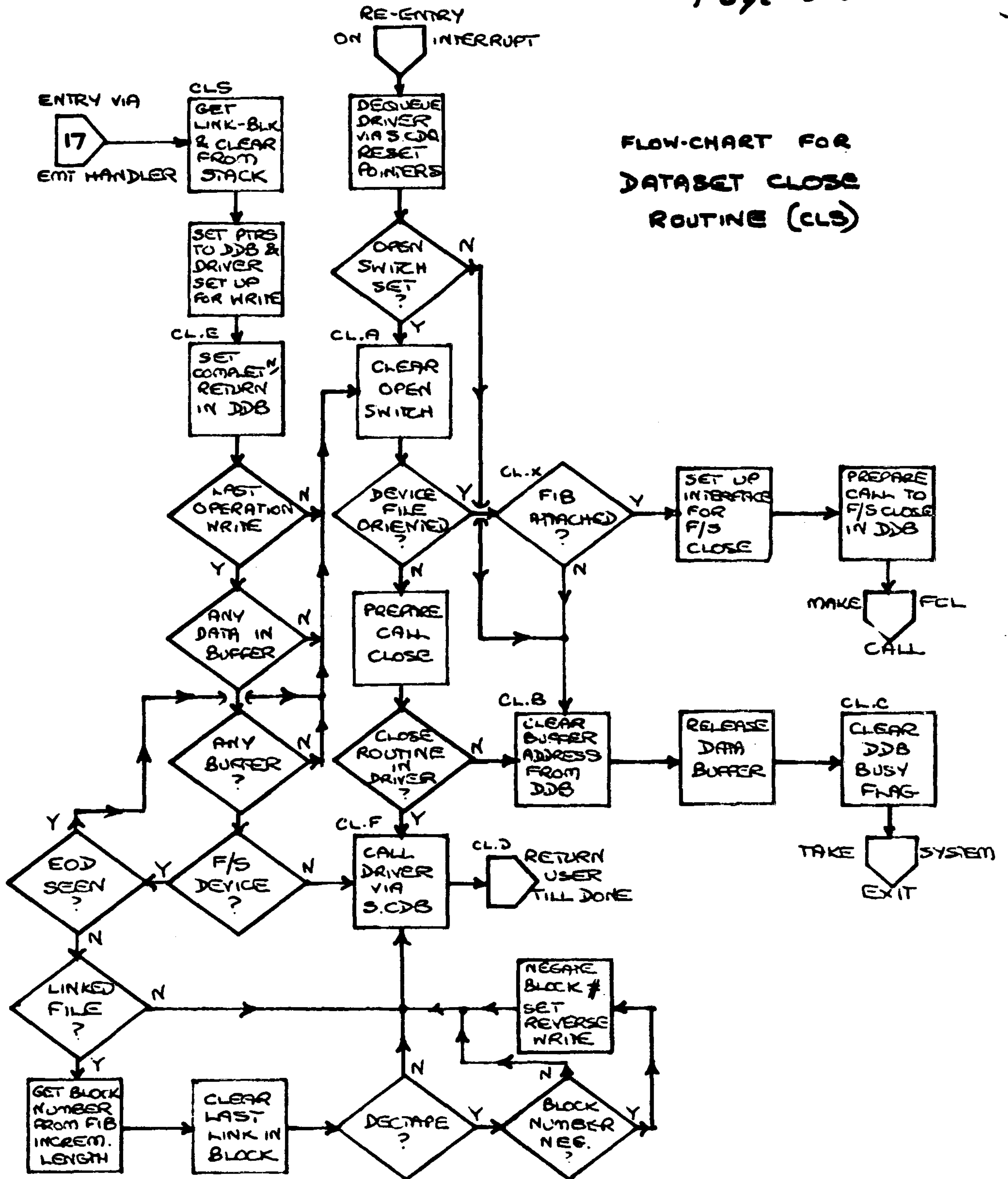


Figure 3-20

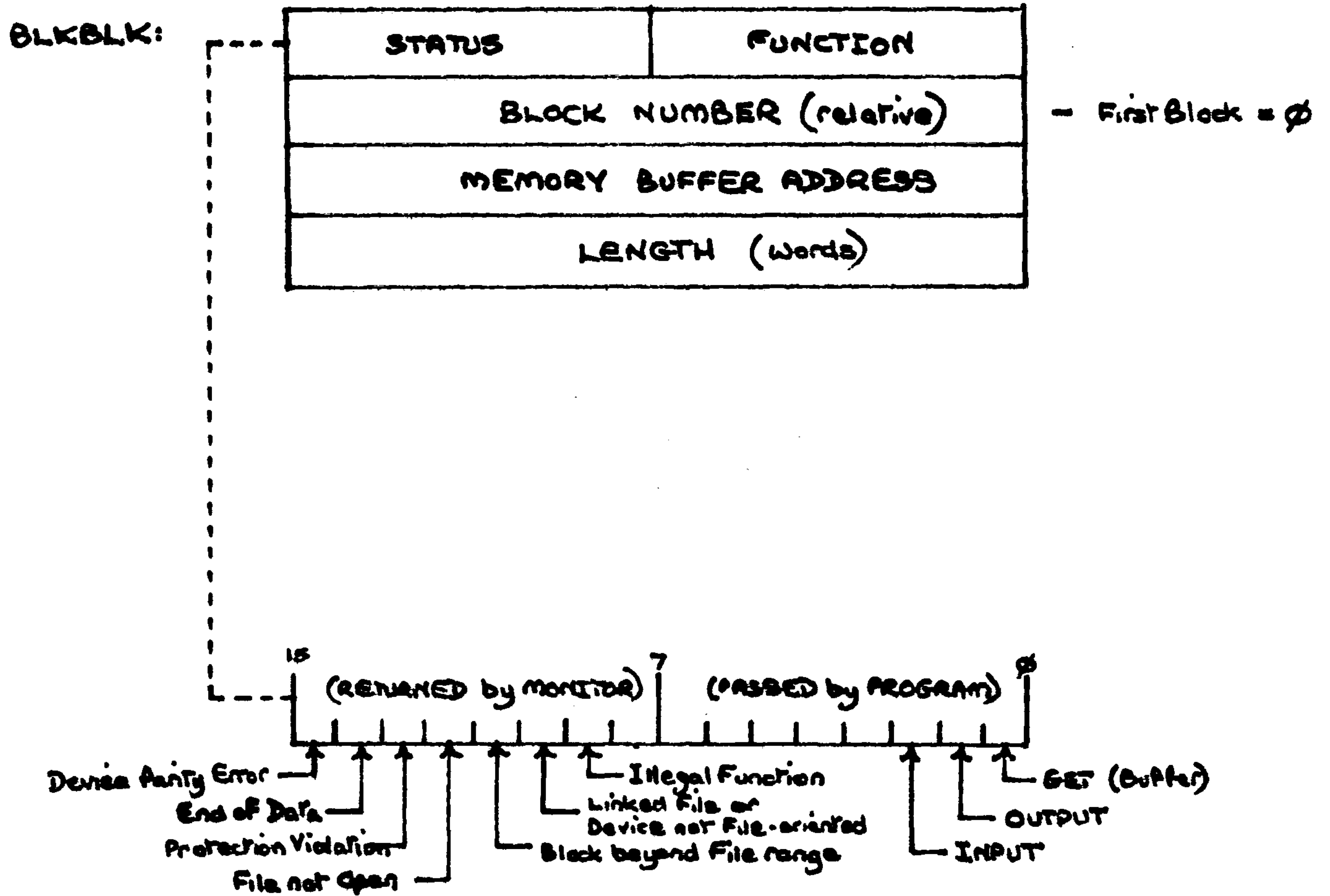


Figure 3-21: The BLOCK-block

SPFBLK:

# OF WORDS FOLLOWING	FUNCTION CODE
ADDITIONAL SUPPORT DATA FROM PROGRAM OR STATUS DATA FROM MONITOR AS REQUIRED BY FUNCTION	

Figure 3-30: The Special Functions Block

DRIVER START:

DRIVER BUSY FLAG [Ø or DDB Address]		
SPECIAL FACILITIES	GENERAL FACILITIES	Required for All Devices
OFFSET TO INTERRUPT SERVICE	STANDARD BUFFER SIZE (16-word units)	
OFFSET TO OPEN ROUTINE	PRIORITY LEVEL for INTERRUPT SERVICE	
OFFSET TO CLOSE ROUTINE	OFFSET TO TRANSFER ROUTINE	
(SPARE)	OFFSET TO SPECIAL FUNCTIONS R.	
DEVICE NAME (Radix-5Ø)		
START BLOCK OF M.F.D		Required only for File-oriented Devices
BIT-MAP POINTERS FOR INDIVIDUAL DEVICE UNITS (1 or 8)		
DRIVER SET-UP & INTERRUPT ROUTINES		

Figure 3-23: DRIVER INTERFACE TABLE

CHAPTER 4

FILE STRUCTURES

Bulk media such as disks or magnetic tapes are by their nature capable of holding many different sets of data. The sets may be wholly unrelated to one another and may, in fact, belong to different users sharing the same medium. In the normal way, none of these users is particularly concerned with how and where his data is held on the medium as long as its initial storage and later its retrieval and perhaps its modification or extension can be accomplished as simply as possible. Moreover each user expects some measure of protection against the corruption of his data either by himself or other users. On such media, therefore, the DOS Monitor supports a file structure which gives users these facilities and this is described in detail in this chapter.

In this context, a "File" is defined as any logically complete set of data stored on a bulk medium which can be accessed through this file-structure. In addition to the files, the medium is also used to store control information by which the Monitor can access the files on the user's behalf ("Directories") and can provide their storage-space ("Bit maps"). These general concepts are discussed in section 4.1.

As far as possible, the file-structure has been designed to be device-independent, in accordance with overall Monitor philosophy, at least in its usage on Disks and DECtape. However obvious differences in those devices, especially the faster disk access rate, have necessitated variations in detailed application. Section 4.2 is concerned with these variations.

Section 4.3 then describes the general manner in which the Monitor processes files and section 4.4 deals with some subroutines which are called by the file-management processes to perform special functions such as directory searching. Section 4.5 covers those processes which extend the standard procedures described in section 3.2.2 for opening, transferring and closing data-sets where these are specifically files. Finally section 4.6 examines routines especially provided to service user requests for housekeeping operations upon directories or the files in them.

It will be seen that the type of file-structure adopted is basically dependent upon the fact that the given medium readily allows updating in place, particularly of control information. There are inherently major difficulties in doing just this on industry-compatible magnetic tapes, a medium not previously mentioned. Because of these difficulties, a simpler structure has been implemented for such tapes and this is the subject of section 4.7

4.1 General Concepts

4.1.1 Files

As stated earlier, a file is any logically complete set of data stored on a bulk-medium within the framework of the special file-structure. It is identified both by the user and inside the system by a name which must start with a letter followed by any combination of letters and digits up to an overall maximum of 6 characters. The name may be further extended by up to 3 letters or digits to distinguish between individual members of the same file family. For example, extensions PAL, OBJ, and LDA to the name PROGRM might signify the assembler source, object and load files for the same program.

The way in which the file is stored upon the medium depends upon the way in which the user normally expects to create and later process it. Two methods are defined for the DOS Monitor and hence two different types of file are allowed:

1. Linked files for sequential access

2. Contiguous files for random access

4.1.1.1 Linked Files

When a user wishes to create a new file for data which he will build up and usually access sequentially by means of the .WRITE and .READ requests described in section 3.2.2, he is probably unable to predict with any accuracy how much storage space the file will use upon the medium. Moreover, because of the time taken to process each individual block, it is almost certain that the medium will have passed beyond the next adjacent block before another transfer is required. Repositioning the medium to use this block is obviously time-wasting.

Linked files are therefore self-expanding series of blocks which are not physically contiguous upon the medium. The first block for the file is allocated from free space on the medium in response to a user request for its creation by .OPENO. Further blocks are added one at a time automatically by the Monitor as the storage of the data requires this. At the same time, to provide for future access, the blocks are chained by a link-word embedded within each block. The first-word of the block is reserved for this purpose and within the file is set to the device address for the next block in the chain. When the file is finally closed the link-word of the last block used is set to 0 as a terminator and any unused portion of that block is cleared. (On all

devices, block 0 is reserved for system use; there can never therefore be any confusion concerning the terminal 0 link). The format described above is further illustrated in figure 4-1.

As previously noted, the blocks of a linked file are not, in general, adjacent to each other. Optimally they are separated by a gap which is deemed sufficient to allow the Monitor ample time to process one block before the next required reaches the read-write head of the device. The number of blocks forming this gap is dependent on device characteristics and is pre-determined for each device. This number is known as the "Interleave Factor" (IF). The gap may of course increase between individual blocks depending upon current availability.

The linked file format readily allows later extension of a file since more blocks can be added and linked in the same fashion as during its creation. Joining linked files together is also a relatively simple process when this consists merely of the replacement of the terminal 0 link in the last block of the first file by the device address for the first block of the second file. (1) The appropriate Monitor requests are therefore available to the user and are discussed under sections 4.5.1 and 4.6.3.

On the other hand, linked files are not designed for random access since the only means by which this can be effected is by perhaps a lengthy search along the links of the file for the required block. Currently the random-access request .BLOCK is illegal for linked files as noted in section 3.2.3.

4.1.1.2 Contiguous Files

Contiguous files are specifically intended for random access through .BLOCK. For this type of usage, the user is likely to know the sizes of his files. By definition, too, the current physical position of a medium is far less relevant to the order in which its data is processed. It is more important that the actual device block can be readily determined from a relative value supplied by the user and can be reached in minimal time. Both of these criteria are most simply satisfied by the use of an area of numerically contiguous blocks on the medium. The resolution of actual

----

1. No attempt is made to fill unused bytes in the last block of a file extended or the first file appended. While these bytes (all null) are no problem in ASCII files or Formatted Binary, they can be read as data in Unformatted Binary. Thus, extension or appendage of files in this mode is currently not recommended.

block is obviously a simple calculation provided that the start block of the area is known. It also follows that restricting operations on the file to a compact area on the medium affords optimal access, particularly on moving-head disks because of the matter of head-positioning and on DECtape because of its single linear track nature. A contiguous file format is shown at figure 4-2. It should be noted that no link-words are provided since they serve no useful purpose; all words including the first one are thus used for data storage.

Before a contiguous file can be used, its storage space must be preallocated through the .ALLOC request described in section 4.6.3. Because the same medium may be shared by both linked and contiguous files, some conflict in block availability must eventually occur, even though the medium may not actually be filled. To delay this as long as possible, linked files are set up on all media at the front, or low-address, end, and contiguous files are given space starting from the high-address end. If the conflict does occur, as evidenced by a failure of the .ALLOC processor to find sufficient space for a new contiguous file, the user must either delete some of his other files or transcribe them in order to utilize more fully smaller disconnected gaps between used blocks, possibly created by earlier deletions.

This simple approach to random access has its disadvantages in that, once the contiguous file area has been established, it cannot be extended since there can be no guarantee that the requisite adjacent blocks are available; nor can such files be joined together unless they are juxtaposed, an unlikely situation in most instances. A user request for either of these operations will therefore be rejected by the Monitor. Against this, however, the user is not limited to random access on contiguous files. Provided that he open the file by the appropriate .OPENI or .OPENC command, he can process his data sequentially by .READ or .WRITE. The Monitor also ensures that such operations are effected only within the preset file-bounds.

4.1.2 Directories

The Monitor employs a two-tier index structure which is stored on the medium with the files in order to maintain control information upon those files and on the users to whom they belong. At the first level, a Master File Directory (MFD) primarily identifies the authorized users of a medium and indicates the start point for the second-level User File Directory (UFD) associated with each of these users. The UFD catalogues that user's files.

Before a medium can be used for file-storage, a basic directory structure consisting of the first two blocks of the MFD must be established upon it. This may be effected in one of the following ways:

1. On the system device, or on unit 0 of the system device, if this has several units, the basic directory is automatically set up as part of the loading of the system by the SYSLOD program (see section 8.2.2).

2. Other devices or other units of the system device must be initialized by means of the /ZE switch provided by PIP-11 as described in the relevant manual (DEC-11-PIDA-D)

In essence, the directories at both levels are themselves linked files in that the individual device blocks allocated to them are chained through the first word of each block and further blocks may be added as and when their data-space is filled (except on DECtape for reasons given in section 4.2.2).

4.1.2.1 MFD

The MFD always starts at a block (MFD1) which is preset for each device. The device address for this block is stored within the relevant device driver interface table discussed in section 3.3.1. It is the only data not stored upon the medium itself and is the start-point from which all file operations are begun. However by design MFD1 falls within the area reserved for Monitor usage on a medium used as system device; it could therefore be write-protected in future Monitor revisions. As a result, the data it contains is set up during medium-initialization and thereafter is only read. Figure 4-3 shows that this currently includes general control data such as the fixed Interleave Factor for the device and pointers to blocks reserved for bit maps (described in the next section). It also of course gives the link to the second MFD block (MFD2) which starts the directory proper.

MFD2, in general, is allocated space at the high-address end of the medium (DECtape excluded), also during the initialization mentioned previously. Its format and that of subsequent blocks which may be added later (a feature however not currently implemented) provides for a series of four-word entries as shown in figure 4-4. The significance of each of these words is as follows:

1. User Identification Code (UIC) - is a two-byte value identifying the individual user and the group with whom he may be working, as described in the

2. UFD start block - points to the start of the particular user's UFD chain (initially set to 0)

3. Number of words per UFD entry - is provided against possible future changes in the file-structure which might lead to expansion of each entry (currently set at 9)

4. (Reserved for future expansion.)

After initialization, the only entries in the MFD on the system-device are the System itself (User 1,1) and a "general user" (200,200); on a non-system-device or unit there is just one - that for the user who requested the initialization. All other entries are zeroed. Before any other user can store files on the same medium, he must request the setting-up of his entry by means of the /EN switch in PIP-11. Until the user actually creates his first file, however, no UFD is established and his MFD link word remains at 0.

4.1.2.2 UFD

The format of each UFD block is depicted at figure 4-5. The significance of the items set up within each 9-word File Entry is as follows:

1. Filename - is the 6-character name for the file as described in section 4.1.1, stored across two words in left-justified Radix-50 format (see section 5.3).

2. Extension - is the 3-character suffix to the file-name also described in section 4.1.1 and similarly stored in Radix-50 in one word.

3. Date - is extracted from the low-order 12 bits of the value currently stored in the Monitor SVT by a keyboard DATE command (see section 6.4.1) at the time when the file was created.

4. Type - is presently used to discriminate between linked and contiguous files (0 & 1 respectively) - the remaining three bits of the same word are reserved against the possible future introduction of other file types.

5. Usage count - covering 6 bits, is set to 77 when the file is opened for creation and prevents the opening of the same file for any other purpose until it has been completed. The count is then reset to 0. The remaining capacity of the six bits

(1-76) is provided to enable the usage of the same file-structure in any future operating system allowing more than a single user. It would then be necessary that the Monitor for the system should know how many users were currently working upon a file in order to prevent its deletion or modification while still in use. (This feature is not implemented under DOS because it implies that every .CLOSE upon a file (even though this was opened originally for input only) requires directory clean-up action. This obviously takes time particularly on DECtape. It is not therefore deemed worthwhile since it merely protects the single user against himself).

6. Lock - using 2 bits, is set to 1 whenever the file is open for extension or update to prevent further opening for the same purpose until the first operation has been completed. The file may still be opened for input only during this period, however.

7. The remaining bits in this word of the entry were originally reserved for the storage of the data-mode of the file content (as described under .READ/.WRITE in section 3.2.2). This was not implemented in order to allow files of mixed mode data. These bits are now spare.

8. Start - shows the device address for the first block allocated to the file and for a linked file thus forms the start of the file block-chain.

9. Length - is the number of physical blocks covered by the file and basically enables control of random access of contiguous files.

10. End - shows device address of the last file-block, a necessary requirement for joining linked files.

11. Protection code - is the eight bit value by which file access can be denied to unauthorized users and which is described more fully in section 4.1.4. (The remaining byte of the same word is currently spare.)

As noted earlier, the first UFD block is allocated to a user and is linked to the appropriate word of his MFD entry when he creates his first file. The block itself is cleared and its first entry slot is then reserved for the new file. Subsequent files are associated with the first available empty slot within the UFD until no more remain. At this point a further block is added and linked to the UFD "file"

and is cleared for similar action. It should be noted that when a new linked file is to be created, an initial entry of Filename, Extension, Date and File start block is made when the file is opened. This both reserves the directory slot and prevents the opening of another file by the same name. The rest of the entry is completed when the file is closed. If a file is deleted, only the first three words of its directory slot, i.e. its name and extension, are removed as this is sufficient to free the slot for further use. No attempt is made, however, to return completely empty UFD blocks to the free space on the medium.

4.1.3 Bit Maps

The mechanism by which the Monitor controls the allocation of the individual blocks on a medium is a bit map in which each bit represents one block. The bit is set to 0 until the corresponding block has been assigned to some file when it is reset to 1. As with directories, a master copy of the bit map representing the whole available medium is stored upon the medium itself.

The size of the complete map for any particular medium is of course a function of the overall size of the medium itself and varies between devices. It follows, however, that on all but the small-capacity media more than one device block is needed to store the full map. Moreover it would obviously be unrealistic for the Monitor to have all of it in core for file-processing at one time. The map is therefore divided into segments, each of which occupies one device block and only one segment need be available in core at any instance. This multi-segment structure was selected, rather than another scheme in which the overall map size is constant for all devices and individual bits cover a group of blocks, not just one, because it offers the following advantages:

1. Wastage of the medium is kept to a minimum (half a block on average per file)

2. By restricting file operations to a compact area on a medium covered by the single segment allowed in core, greater efficiency is obtained, especially on moving-head disks

3. Control of allocation and of block interleaving is relatively simple.

For the sake of device-independence, the size of a map segment is constant for all devices and is based upon the capacity of the smallest block-size, namely 64 words. Figure

4-6 shows that the format of the resulting segment includes a four-word preamble, the purpose of which is indicated below:

1. Link to next block - implies that a complete medium bit-map is again a form of linked file and might as a result be stored anywhere on the medium. In practice, however, the individual blocks are contiguous and are generally set up during initialization at the high-address end for reasons to be given in section 4.2. Currently also the map-file cannot be extended to cover an increase in the capacity of the medium as it has been assumed that any such increase will normally be followed by a complete reinitialization which will rebuild the whole bit-map afresh. It should be noted, too, that because random-access capability is needed, particularly for file deletion, an index of blocks used for the bit map is stored in the first MFD block as indicated in the previous section. This index, rather than the links, may be used in some instances for sequential access as well.

2. Map number - is needed to allow computation of the range of the blocks represented by the map segment, using algorithms like the one following (based on numbering from 1):

   HIGHEST BLOCK = (MAP SIZE X 16 X MAP #) - 1

3. Map size - gives number of words actually used by the map since some devices (particularly DECtape at present) may not need the whole of the segment of constant size for their complete map. This information is therefore required for the above computation. It is constant for a given device - if all words in the last map segment are not required, they are all set to 177777.

4. Link to first block - is provided to enable the Monitor to apply the philosophy mentioned in the previous section - that of keeping linked files as far as possible at the front end of the medium. This implies that whenever the map segment currently in core shows that no more blocks are available within its area, the search for another free block for a linked file must always revert to the start of the map in case some earlier blocks have been released by some recent file-deletion. This link facilitates the process.

The remaining 60 words of the segment form the map itself, covering 960 Blocks on the medium. The numbering of the

bits within each word is from right to left. Thus the following simple algorithm suffices for the conversion of a relative block number into the map position of its representative bits (assuming Block 0 = Word 0, Bit 0)

1. Map word = Block/16

2. Bit position in the word = remainder

As mentioned above, the bit-map structure is established during medium-initialization. At this time the whole map is cleared to 0 with the exception of the bits which represent blocks already allocated for the MFD, the map itself and perhaps the System and its UFD. In the last map segment any bits for which there are no corresponding blocks are also set to 1. In general, thereafter, as a new file is created, the Monitor reads an appropriate segment into core, unless one is already there for some other purpose. The relevant bits for blocks assigned to the file are set to 1 in the core segment while the file is being built; when it is closed, the core segment is rewritten into the master copy on the medium. If the file is later deleted, its assigned bits are reset to 0.

4.1.4 File Protection

During the introduction to this chapter, it was stated that any file-owner expects to be protected against the corruption of his files either by himself or other users sharing the same medium. In fact he probably also wants to control the form of access those other users have to his files. Moreover he may naturally wish to give greater privileges to those with whom he is closely associated in his usage of the system than all others.

Figure 4-7 illustrates an eight-bit code by which the user can indicate his requirements for each of his files. The purpose of each of the three fields shown is as follows:

1. Owner: the user himself can, in general, always access his files in any manner he chooses (e.g. Read, Write, Delete, etc.). However he is afforded two safeguards represented by the bits in this field. By setting bit 6 to 1, the user can deny himself Write and Delete capabilities in order to prevent accidental corruption of his data. With bit 7 at 1, he can protect the file against automatic deletion by the system when he logs off the computer (a feature not yet implemented in DOS).

2. Group: this field can be set to a three-bit code by which the user indicates the level of access he is prepared to allow his "friends", i.e. those who

have the same UIC group number as himself. The codes presently assigned are:

7 = no access at all
5 = run access
3 = read access
1 = write access
0 = delete access

At any level, of course, the inclusion of higher levels is implied. For example, with protection set at code 1, the Monitor will allow a "friend" to write, read and run the file, but not delete it.

3. Others: this field can be similarly coded to show the level of access allowed to users outside the owner's group.

As indicated in section 4.1.2, the resultant byte is stored in the file entry in the UFD when the file is established. The Monitor always checks every request for access against the stored code (see section 4.4.2). The user can change the code at any time either directly from a program using a .RENAM or .KEEP request described in Section 4.6.4 or through the /RE or /PR switches available in PIP-11.

It should again be noted that the protection offered by this code can be valid only when every user sharing the medium is performing all his I/O operations within the file-structure framework, i.e. by .READ, .WRITE or .BLOCK. The Monitor cannot and will not deny access of any sort to a user requesting .TRAN (even to its own files).

4.2 Application By Device

The overall philosophy for the DOS file-structure, described in the previous section, provides for operations which can be device-independent to some considerable degree and this concept has been honoured as far as possible in its implementation on all disks and DECtape. Nevertheless, there are obvious problems in rigid adherence to it at every point because of the nature of the devices themselves. This section therefore illustrates the detailed application of the file-structure and in particular indicates variations from the norm.

4.2.1 Fixed-head Disks

Although the RF11 disk is basically word-oriented, DOS imposes a block structure based on 64 words to a block, with the driver providing the necessary address-conversion. The RC11 disk driver always assumes a similar block size and in response to each system request for a block, transfers the appropriate double 32-word sector accordingly. Furthermore the addition of extra platters to either controller merely extends the continuous surface available to the system (provided of course that the user has physically assigned all his platters in a strict ascending sequence from 0). To all intents and purposes, therefore, RF11 and RC11 systems differ only in their overall capacity and hence the number of bit-map blocks needed to control this. The following table shows the correspondence:

PLATTERS	BLOCKS	BIT MAPS
1 X RC11	1024	2
2 X RC11	2048	3
3 X RC11	3072	4
4 X RC11 or		
1 X RF11	4096	5
2 X RF11	8192	9
3 X RF11	1288	13
4 X RF11	16384	18

The basic format for either disk, when not in use as the system device, is illustrated at figure 4-9. This shows in particular that the requisite bit-map blocks are in a contiguous area at the high end as was noted in section 4.1.3. They are stored here for two reasons:

1. By their nature, they must be updated and should not be in any potentially write-locked area - the top of the disk is the least likely part to be protected.

2. The top of the disk on the other hand is the normal place for the storage of contiguous files - the bit map blocks are thus compacted to keep them out of the way.

The question of write-protection introduces another dissimilarity between the two types of disk when used as system-device. The hardware facility on both provides a set of switches with each switch covering a particular area of the disk surface. The area however is not the same. For reasons stated in section 2.1.2, DOS does not currently allow the user to apply any of these switches. Nevertheless to meet the future possibility that the Monitor itself might be write-protected, the top of the area reserved for its use has been set to coincide with an appropriate switch. On figure 4-9, which gives the layout of a system disk, a Hardware Protection Line has been shown corresponding to:

1. End of block #777 on RF11 (equivalent to word #77777)

2. End of block #437 on RC11 (equivalent to actual block #1077)

File-processing operations on both RF11 and RC11 follow the general outline given in the previous section. Linked file deletion merits further comment here since this is one operation handled differently on DECtape and disk. A part of the process is the adjustment of the master bit-map to reflect the blocks released by the deleted file. This can only be accomplished by a search of the file itself to determine the relevant blocks, since the general structure provides for no other record of this. Such search also implies that the link-words on the blocks themselves must be removed to safeguard against potential disk-corruption later; for, should a program or system failure occur, files might remain incomplete on the disk and the results of deleting them in this way could be unpredictable. This search and link-clear obviously takes some time but it is not deemed unreasonable, because of the relatively fast disk access rate and it avoids other complications to be discussed in section 4.2.3.

The interleave factor for linked files on both RF11 and RC11 is 5. This has been based upon the optimal time taken by the Monitor to complete the processing involved in loading a program from these disks.

4.2.2 Moving-head Disks

The only moving-head disk presently supported by DOS is the RK11, though this occurs in two versions - one high density,

structured into 4800 sectors of 256-words each, the other low density with sectors reduced to 128 words. For simplicity, both versions are handled by the Monitor in 256-word blocks, the RK11 driver again providing address-conversion where required.

Unlike the fixed-head disks, the addition of extra RK-11 cartridges cannot be treated as simple extensions of a continuous surface, since the user cannot be expected to retain the same set of individually changeable cartridges together as an entity. Instead each must remain an independent unit with its own structure - and, in fact, is a different medium in the sense of the definition of a file given in the introduction to this chapter. As a result, the number of bit maps to control the capacity of an RK11 disk is always constant, though of course there is a difference between the two versions:

HIGH DENSITY (4800 BLOCKS) = 5 MAPS
LOW DENSITY (2400 BLOCKS) = 3 MAPS

The layout of an RK11 disk, whether used as the system device or not, is otherwise the same as that illustrated for the fixed-head disks at figures 4-8 and 4-9. In particular, this means that the bit map blocks remain together at the top of the disk, because this again allows the user the most contiguous space for large random-access files. This is considered to be more important than the reduction of head-movement, which might follow if the map segments were dispersed across the disk surface so that each reside more closely to its area of control. (This was originally planned - hence the linked-file structure of the map.) It should also be noted that "hardware protection line" as such is meaningless for RK11, since this offers no selective write-lock feature - it is all or nothing. Nevertheless, the boundary it represents is the end of block #277 which gives the Monitor sufficient reserved area for its present needs and some room for expansion.

File-processing for RK11 again follows the outlines given in section 4.1 as qualified at the end of the previous section. The Interleave factor is also 5 on the same basis as that used for RF11 and RC11.

4.2.3 DECtape

Although, in many ways, DECtape can be equated to disks in mode of usage to the extent that most file-processing operations can be truly device-independent, it has one serious draw-back: it is single-tracked and it is linear. It is obviously slower therefore and tape-positioning assumes a much

greater significance. One minor effect of this is that Directory and bit map blocks, which are generally accessed in the same sequences should be kept reasonably close together rather than at opposite ends as on disk. This is readily feasible when the overall structure allows allocation of these blocks in any position and potential write-locking does not require consideration, since this is not selective.

A second consequence is a modification in the format of a linked file. If a search for the next free block reaches the end of the map segment currently in use, it is reasonable that this search be resumed from the start of the segment on disk - but not so on DECtape. Because of the physical tape position, it makes more sense to use the hardware facility for transferring in reverse. However, this facility only affects the access of the tape; at the core end, transfer is always forwards. This means that blocks written backwards must also be read backwards. Hence it becomes necessary to remember the tape transfer direction when the file is created. Since one DECtape contains only 576 blocks of 256-words each, it is convenient to use the link for this purpose:

Positive link = read next block forwards
Negative link = read next block backwards

The search algorithm actually applied to DECtape uses the fact that when the tape is stopped by the driver after transferring a block forward, it is appropriately positioned for immediate access to the preceding block (vice versa for a backward transfer). Thus this is the point from which the search is resumed. Hence the format of a DECtape linked file might appear as illustrated in figure 4-10.

Since the driver always stops the tape between transfers, tape position is also the basis for the interleave factor on DECtape, presently set at 4. Owing to the time actually taken to bring the tape to a standstill and later to bring it back up to normal speed for a transfer in the same direction, three blocks usually pass across the read-write head. Occasionally a fourth may also, especially if the braking-mechanism of the transport is incorrectly adjusted, and tape repositioning is then necessary. However, the frequency of such occurrence is not sufficiently high that it currently warrants an increase in the IF when this might unnecessarily impose some 50 milliseconds of extra access time for each block in almost all DECtape linked-file operations.

Linked-file deletion was specifically mentioned in section 4.2.1 because this is one major area in which DECtape is different. The disk method was shown to be a matter of searching the file for its associated blocks and at the same

time clearing the link-words. On DECtape this is out of the question: the search alone might be a lengthy operation; to access each block a second time to clear its link-word, with tape repositioning each time makes it impossible. The only alternative is the provision of a separate record to show the blocks in each file and the shortest form of this is another bit map. However, this then raises the problems of having several maps in core at one time (the master for determination of block availability and one for each file currently under creation) and of their storage upon the DECtape. Both questions have been solved by imposing the following restrictions on DECtape usage, which are reasonable in its case but could not be considered for disks:

1. By allowing only one file under creation at a time (not without sense for practical operating considerations) the need for any but the master bit map in core disappears. The file copy can be constructed from the before and after states of the master. Apart from saving core, this also requires less departure from the method of general processing - merely one extra operation while the file is closed.

2. By imposing a maximum of 56 files on a single tape, the individual file maps can be packed into a contiguous area of 8 blocks (since each block of 256-words can accommodate 7 maps each using 36 words). Moreover, this figure of 56 is also the capacity of 2 UFD blocks on the basis of 28 9-word entries per block. The map for any file is easily accessed by equating its relative position in the contiguous area to the relative position of the same file's entry in the UFD. For example, the 30th file in the directory has its map stored as the second entry in the fifth block of the map area. (It should be noted of course that the foregoing need not apply to contiguous files. If any of these are established on the tape, their corresponding file maps are left blank.)

The layout of a DECtape, allowing for the limitations described, is illustrated at figure 4-11. It will be seen that the full MFD/UFD structure is provided. Basically, this maintains compatibility with disk operations. It also enables the perhaps dubious facility of several users sharing the same tape. The UFD, of course, cannot be extended, in view of the restriction on the number of files. (Since DECtape is not currently a potential system device, there is no corresponding illustration.)

4.3 File Management

DOS file management is performed by a set of Monitor modules which are all called by EMT, each having its own unique code. However all of them have characteristics which make them different from the general I/O modules discussed during chapter 3. The purpose of this section is to examine these characteristics and then to discuss their common interface channels. The individual modules will be described in following Sections.

All the modules concerned are listed with their EMT codes in figure 4-12. They are shown to fall into one of four categories:

1. Modules called directly from a user program to perform file-housekeeping functions, e.g. .ALLOC, .DLETE, etc.

2. Modules called by other Monitor routines, performing general I/O operations such as .OPEN, in order to provide file-handling.

3. Modules called generally by other file-management modules to carry out common operations such as directory searching.

4. Modules specially associated with other modules in the file-management set.

This grouping illustrates one obvious difference from the general I/O modules: only those in the first category are immediately accessible by a user. These of course, being I/O modules themselves have EMT codes in the range 0-27 as the list shows. They have prescribed calling sequences, described in the Programmer's Handbook, which include the passing of information on the processor stack and this must be removed before the user program is recalled. On entry from the EMT handler, the stack top contains the contents of the program Registers as described in section 2.2; these must be restored on exit. Actual Register contents passed by the EMT handler are especially relevant to these modules, in particular:

R0 = address of the DDB for the
     dataset being serviced
R1 = address, on the stack, of the call
     parameters passed by the program

The modules in categories 2 through 4 on the other hand are invisible to the user as indicated by their EMT codes. In the Programmer's Handbook these are shown as reserved for system use. They are not in the I/O range since the mo-

dules, though still performing I/O, are called at a second or even third level within the Monitor and the requisite information has already been supplied at the first level. This fact also causes other changes in the call and exit interface as follows:

1. Calling sequences need follow no definite format. Transient information may thus be transmitted either through the stack or the Registers, whichever is more convenient to the modules being called.

2. If the Registers are used, the called routine still receives the data on the stack, as saved by the EMT Handler. (A typical stack following a call is illustrated at figure 4-13.)

3. Unless the interface requires it, the saved Register contents must be restored intact to the calling routines. This applies in particular to categories 3 & 4. Modules in category 2, in fact, exit directly to the user: they therefore actually remove the Register contents and the saved call PC and status in order to produce the stack state for such exit.

4. Contents of Registers on entry from the EMT handler in general have no significance.

A less apparent difference between the file-management modules and the others arises from the fact that they still provide program services. The user can therefore select between making them resident, either permanently or just for a program run, or leaving them to be swapped in when required. For this purpose, he is informed of the existence of all of them - even if they are shown only as subsidiary routines in Appendix C of the Programmer's Handbook. If he chooses residency, there is no problem. The swapping situation however is another matter. In reality, modules in categories 3 & 4 serve as subroutines providing special functions for the others but they cannot be brought into the Swap Buffer for this is still occupied by the calling modules in categories 1 and 2. The solution is that the latter obtain another 256-word buffer from free core, via the Monitor S.GTB routine described in section 2.4.2. and move themselves into it. This then leaves the Swap Buffer free for the called routines to use in the normal way when required, as described in Section 2.3.

The actual mechanics by which this operation is managed deserve further explanation. In the first place, the move-out must only occur if the category 1 or 2 module is actually in the Swap Buffer and not already resident outside it, for obvious reasons. The module must itself resolve this and it

uses the fact that the Usage-Count in its first byte can only be 0 if it has not received control through Swap Area Manager (in much the same way that the other modules described in chapter 3 determine their exit sequence). If the count is non-0, the extra buffer is obtained and the move is accomplished, as illustrated in figure 4-14. During the move, due regard is taken of the position-independent nature of both buffers. In addition, the first word of the routine when stored in the new buffer is used as a counter; this must start at 177401 (or -1,1) since, as an overlay, the routine cannot be re-entrant and hence is only once called; it ends at 1 after the move and thus still indicates non-residency. The routine then clears the first word of the Swap Buffer to free it for further use and continues from its new location. On exit, the extra buffer must be returned to free core; however, the routine cannot do this while it is itself still in the buffer. Therefore when the Usage Count (as noted above), shows such action to be necessary, the routine simulates a call to the Monitor S.RLB sub-routine by placing the system exit address (without Usage-Count decrement) on top of the stack and by then using JMP rather than JSR.

The following paragraphs describe the data areas especially established for the transmission of file management data.

4.3.1 User File Block

All user programs calling for file-management operations, or for general I/O which might lead to such operations if the device is right, are required to supply file information through a User File Block (UFB). This is described in detail in chapter 2 of the Programmer's Handbook. Briefly, as shown in figure 4-15, this allows the user to indicate his purpose and identify the file required, including its owner, (if not himself), and to establish its protection. It also provides room for the Monitor to return error information and an exit should such error occur.

The address of this File-block is passed as one of the program call parameters and the Monitor routine uses it to access its contents as required. However, as indicated in section 3.1.2.2, the user can supply different file information via a keyboard ASSIGN command. If he has done this, the .INIT routine saves the address of the relevent entry in the DAT pointer in the dataset DDB as noted in section 3.2.1.1. The file-management routine therefore checks such entry and uses its contents if valid (see LUK in section 4.4.1)

4.3.2 File Information Block

It has already been mentioned that the file-management modules can use the stack or Registers for transmitting transient data between one another. More permanent data which must be held across several requests requires different storage. Such data, being applicable only to the dataset being serviced, is impure and is therefore similar to that stored in the DDB for general I/O processing. As shown in Section 3.1.2, the size of the DDB itself is set to cater only for the general need; file-management requires additional space and this is provided in a File-Information Block (FIB).

Figure 4-16 shows the FIB format. Basically this is designed to follow a sequence corresponding to the file entry in the UFD. It will be noted, however, that the file name itself is not included. This is unnecessary for a file which already exists; for a new file, as noted in Section 4.1, the need is removed because the entry is stored on the medium as soon as the file is opened. Other items do appear to avoid unnecessary access to the medium. The purpose of each item is described below:

1. Next Block - is provided mainly for the creation or extension of a linked file, since the allocation of the next block must occur before the current one (or the UFD in the case of .OPENO) can be written.

2. How Open Code - is used to transmit the appropriate content of the UFB between program requests, since the latter is only accessible during .OPEN. (see Section 3.2.2.1).

3. Extension start block # - saves the beginning block following .OPENE. To protect the user against failure and also allow alternative use for input at the same time, a file being extended is left in its original state until the extension is completed by .CLOSE. This item therefore is needed to enable the "APPEND" operation which occurs at that time.

4. Type - corresponds to the similar entry in the UFD file entry and is discussed in Section 4.1.2.

5. Spare.

6. Start block # - always contains actual file start block even during extension (hence 3 above)

7. # of blocks - is used as a counter during file creation or extension (starting from the original length in the latter case) and eventually is stored in the UFD as "Length".

8. End block # - stores the present end of a file under extension - otherwise is normally unused until a new file is closed.

9. Index into Directory Block - shows the relative position of the file entry in the appropriate UFD Block (see next).

10. Directory block # - enables immediate access to the actual device block on which the UFD entry for a file under creation or extension appears and thus removes the need for a fresh directory search on .CLOSE.

11. Protection Code - shows the file's protection level.

12. Interleave factor - saves the content of the corresponding entry in MFD block #1 as described in Section 4.1.

13. Bit map pointer - contains the address of the buffer in core in which the appropriate medium bit-map segment is loaded (or is currently held - see next).

14. Bit map Q link - enables the opening of several new files (on different datasets]) on the same medium at the same time. For reasons stated in Section 4.1, only one bit map segment from a medium can be in core at any one instance. Thus several files must share the same segment. To allow this, the appropriate FIBs are chained (as shown in figure 4-17) starting from an allotted word in the device driver standard interface table (see Section 3.3.1). When the first file requiring a map is opened, the map segment is loaded into a buffer claimed from free core and the driver pointer to the concerned FIB is set. Subsequently FIBs are linked and unlinked as their associated files are opened and closed. When the last file is finished, the driver word becomes 0 again and this is used to signal the release of the map-buffer.

15. Temporary workspace - is used mainly during file creation or extension by .WRITE (see Section 4.5.2)

16. Temporary workspace - same as #15.

The FIB is set up in a buffer claimed from free core via the Monitor S.GTB routine only if file-management operations are called. For normal file-processing this is done by the general .OPEN routine (see Section 3.2.2.1). Housekeeping rou-

tines claim their own. (In fact, since the latter, in general, complete all their required functions in one user program request, the FIB is not really necessary. However for compatibility of processing especially by the common subroutines and to provide additional workspace it is still set up.) While the FIB is established, it is accessed through a pointer in the last word of the DDB (see figure 4-16 and section 3.1.2.3). It is released as soon as it has served its purpose, e.g. on .CLOSE. Its DDB pointer then is cleared and its buffer is returned to free core via S.RLB.

4.4 General Purpose Routines

The purpose of this section is to describe those file-management modules which act as sub-routines providing services generally required by the other modules, such as directory-searching, collection of a bit-map segment or allocation of a new block. (Category 3 in Section 4.3.) When non-resident, they operate from the general Swap Buffer.

4.4.1 Directory Search (LUK)

The function of this routine is to perform a directory search for a filename to determine the presence or absence of a particular file on disk or DECtape.

Calling Sequence:

```
MOV     #FILBLK,-(SP)
MOV     #LNKBLK,-(SP)
MOV     SP,R1
SUB     #14,SP
EMT     46
```

R0 = Address of the DDB
R1 = (ADDR-2) of Pointer to UFB

The caller must subtract 14 from the stack pointer for the routine's return arguments. Note, that this routine does not remove the arguments pushed onto the stack in the above calling sequence.

Return Arguments:

```
  (R6)=X        ;X DECODED BELOW
 2(R6)=FILE
 4(R6)=NAME
 6(R6)=EXT
10(R6)=UIC
12(R6)=PCODE  ;PROTECTION CODE
```

X=-1 If the UIC was not entered into the MFD DDB+4 will contain a zero upon return.

X=-2 If the file did not exist in the UFD, DDB+4 will contain a 0; DDB+2 will be 0 if no empty slots in the UFD or will be the UFD block number which has an empty slot; DDB+22 will be the index to the empty slot. If the named file was 00.0, A -2 will be returned also, in which case DDB+4 will be unchanged.

X=-3 If there was no UFD attached to the MFD entry. Upon return DDB+4 will contain the block number containing the MFD entry. DDB+2 will be set to the address of the UFD pointer within the MFD.

X=other than above - then it is the core address of the UFD entry. DDB+4 will contain the block number of the UFD entry; the block containing the UFD entry is in core.

Description:

This routine executes entirely in the Swap Buffer (MSB), unless it is made core resident. When this routine is entered, the state of the stack is as follows: Registers 0 through 5 are saved on the top of the stack, then the PC and PS, and then a 6-word work area for the return arguments. R0 and R1 are restored from their saved values on the stack. R1 is then set to point to the User File Block, and R2 is set to point to the 6-word work area on the stack. The user file block is then moved onto the stack as: file name, extension, UIC, and protection code. If there was an assignment made, the assigned name is moved in. If there was no UIC in the file name block, the UIC is taken from absolute location 440, which is the logged in UIC. If at this point the file name is 00.0, a -2 is put on the previous top of the stack and the routine returns to the caller through the common exit routine, S.EXIT, at absolute location 42.

Now the routine gets the block number of the MFD from the device driver. The first MFD block is then read into core. If the device is DECtape the first MFD block is skipped since this does not contain UIC's and would require an unnecessary time expense. The routine then scans the MFD block looking for the desired UIC. Reading in MFD blocks continues until none are left or the UIC is found. If the UIC is not found, a -1 is put on the previous top of the stack and the common exit is taken back to the caller. When the desired UIC is found in the MFD a test is made to see if a UFD exists for the UIC. If no UFD a -3 is put on the previous top of the stack and the common exit is taken back to the caller.

If a UFD does exist for this UIC the starting block number of the UFD is put into the DDB and the first UFD block is read into core. Now the routine begins scanning the UFD for the desired file name and extension. UFD blocks continue to be read in and scanned until the end of the UFD chain is reached or until the file and extension are found. If the end of the UFD is reached without a match, a -2 is put on previous top of the stack and the routine exits as previously described. If the scan is successful and a match is made,

the core address of the UFD entry is put on the previous top of the stack and the common exit is taken.

Note: the directory LOOKUP routine has an imbedded READ/WRITE routine and also uses a common exit routine through absolute location 42 which restores the stack to its state before the EMT request. This does not modify the return arguments on the stack.

4.4.2 Check Access, Set-up and Release FIB (CKX)

This routine is a three part utility program which performs the following functions:

1 - CHECK ACCESS PRIVILEGE
2 - SET UP FILE INFORMATION BLOCK
3 - RELEASE FILE INFORMATION BLOCK

The EMT codes for the three functions are the same, EMT 52, but to call the individual function a 0, +1, or -1 respectively is pushed onto the stack. The routine will then dispatch to the proper segment, within the file structure module. The three segments are individually described in the following discussion.

4.4.2.1 Check Access Privilege

The function of this routine is to determine the permitted READ/WRITE access privilege of the caller. The caller must request the level of access he desires. Then this requested access is compared with the file protection level. The requested access is then either granted or denied.

Calling Sequence:

```
        MOV   UIC,-(R6)
        MOV   PCODE,-(R6)
        MOV   #ACCESS,-(R6)
        CLR   -(R6)         ;ZERO INDICATES CHECK ACCESS CALL
        EMT   52

        ACCESS LEVELS:

          0 - Delete
          1 - Write
          3 - Read
          5 - Run
```

R0 = Address of DDB

Return Arguments:

(R6)=0	desired access permitted
(R6)=non 0	desired access denied

Description:

When the section of the routine is entered, the stack state is as follows: Saved Registers R0 through R5, PC and PS, Function indicator and calling arguments. R3 is set to the desired access, and R5 is set to the protection code, from the values passed on the stack.

The level of access to any file is determined by the owner when he sets its protection as described in Section 4.1.4. The check access routine determines whether the desired access by any user shall be permitted or not in the following manner. The routine first checks to see if the caller is the file's owner, by comparing the logged-in UIC against the UIC calling argument on the stack. If he is the owner then bit 6 of the protection code is tested; if it is set he cannot write on or delete the file, he protected his file against himself. Bit 6 being not set gives the owner complete access.

When the caller is not the owner a second test is made to determine if the caller is in the owner's group. They are in the same group if their project number is the same; this is the high order byte of UIC. If they are in the same group the access desired is compared against bits 3, 4, and 5 of the file's protection code. If not in the same group the access desired is compared with the low order three bits of the protection code. If the protection code is higher than the desired access requested then access is denied, and a -1 will be returned on top of the stack. If the protection code is equal to or lower than the requested access then access is granted, and a 0 will be returned on top of the stack. Then the routine returns to the caller through the common exit routine, and the top of the stack holds the returned value, 0 or -1.

Access Levels

Bit 6 Set - Owner can't write

Protect Code greater than 1, others can't write
Protect Code greater than 3, others can't read
Protect Code greater than 5, others can't run

4.4.2.2 Set-up File Information Block

When this routine is called a FIB may or may not be attached to the DDB. Therefore the routine has a two fold function. One function is to get a DATA BUFFER AND A FIB BUFFER from free core, when necessary. This function is desired when called by routines other than FOP (FILE OPEN) and FCR (FILE CREATE). The second function this routine performs is to set up the FIB and link it to the DDB. This routine also sets the address of the data buffer, in the DDB. It is necessary to set up the FIB when called by any file structures routine including FOP and FCR. The FIB is set up primarily by transferring data to it from the files UFD entry.

Calling Sequence:

```
        MOV     #1,-(R6)
        EMT     52

        R0 = Pointer to DDB

        R2 = Pointer to the File's UFD entry in
             core. This is necessary, when the
             routine is called, to set up the FIB
             and link it to the DDB.
```

Return Arguments:

DDB+26 contains a non-zero value if the FIB was successfully setup. The value is the core address of the FIB associated with this DDB. DDB+26 contains a zero when the FIB was not successfully set up.

Description:

When this section of the routine is initially entered, the state of the stack is the same as described under the "Check Access" section, with the exception of the call arguments. This section sets up R5 with the FIB Link Word from the DDB. Then, if there is a FIB already, R2 is restored as the pointer to the File's UFD entry.

When SETUPFIB is entered the FIB link word of the DDB is tested to determine if there is a FIB attached to this DDB or if it will be necessary to get a buffer for a FIB and link it to the DDB. A zero FIB link indicates no FIB; when this occurs the following procedure is followed. The driver address is determined from DDB -2; the Standard, Driver Buffer Size is gotten from the driver, as the number of 16 word units necessary. This value is saved and then this value is used to calculate a word count, which is put in the DDB. The number of 16 word units is incremented by one to

include a buffer for the FIB. A call is then made to the monitor, buffer allocate routine (GETBUF), requesting the number of 16 word units desired. The GETBUF routine is called through absolute location 54. The first 16 words of the returned buffer are linked to the DDB as the FIB buffer, and the 16 words are cleared. The Data Buffer is then linked to the DDB at DDB+6, transfer buffer address word. Now the routine loops back to the initial test upon entry to this routine and will now find a FIB attached to this DDB.

When a FIB is attached to the DDB the following occurs. Information concerning the file is transferred from the UFD entry to the FIB. The transferred information includes the Next Block Number which is also the First Block, the File Type, the File's Start Block Number, the File Length in blocks, and the File's Last Block Number. Then three more items are set up in the FIB: the index into the directory block, the directory block number and the protection code. This routine then exits back to the caller through the common exit routine, S.EXIT. The information used to set up the FIB is in core when this module is called.

4.4.2.3 Release File Information Block

The function of this routine is to unlink the FIB from the FIB chain and release the FIB Buffer, the Data Buffer and also the Bit Map Buffer if no more FIB's are attached to the DDB. The FIB chain must remain continuous after the FIB is unlinked. So therefore a FIB pointer must be reset.

Calling Sequence:

```
MOV     @PC,-(R6)
EMT     52
```

The instruction MOV @PC,-(R6) pushes a negative number onto the stack. A Negative number indicates a call for release FIB.

Return Arguments:

None

Although the FIB LINK in the DDB is cleared, and the associated bit map is released, if this is the only FIB in the chain.

Description:

Upon entry, the state of the stack is the same as for a "Setup FIB" call.

When the Release FIB routine is initially entered the status byte of the DDB is tested to see if the Dataset is currently busy, bit 7 set. If the Dataset is busy the routine returns immediately to the caller through the common exit routine, S.EXIT, with no action taken. If the Dataset was not currently busy an index into the device driver is calculated to point at the first FIB link, of the FIB chain for this device unit. If the first FIB link is zero, there is no FIB chain for this unit. In this case it is only necessary to release the Data Buffer and the FIB, which is done through a call to the monitor, buffer release routine. Then clear their pointers from the DDB. The routine then returns to the caller through S.EXIT. If the first FIB link in the driver is non-zero the FIB link in the DDB is then tested to see if there is a FIB attached to the DDB. If no FIB is attached to the DDB release the Data Buffer, unlink it from the DDB and exit back to the caller through S.EXIT. If the DDB has a FIB attached, it is tested to see if it has a bit map attached. If the FIB has a bit map attached to it and it is the last FIB in the chain, the bit map buffer is released through a call to the monitor buffer release routine. If the FIB has a bit map but is not the last FIB in the chain, the FIB is merely unlinked from the chain and the chain linkage reset. The Data Buffer and the FIB Buffer are then both released through the monitor and their linkages are removed from the DDB. The common exit is then taken back to the caller.

4.4.3 Transfer Bit-map to and from Core (GMA)

4.4.3.1 GET MAP

This routine performs two functions: it can be called to get a buffer for a bit map, read the bit map into core and link this FIB to the end of the FIB chain. The second function of the routine is to write the bit map from core to the device. The first function is called GETMAP, the second WRITMAP. Both are called with the same EMT, with the value of R3 determining the specific function.

Calling Sequence:

a. To read in a bit map

```
R0=pointer to DDB
R3=pointer to DDB
EMT     50
```

b. To write out a bit map

```
R0=pointer to DDB
R3=0
EMT     50
```

Return Arguments:

FIB+0 contains X

```
X=0     if map setup sucessfully
X=+1    if no buffer available
X=-1    if file already open on DECtape.
```

Description:

This routine executes entirely within the Swap Buffer (MSB) unless it is made core resident. On entry registers R0 through R5, the PC and PS are respectively saved on the top of the stack. Upon entry, R0 is restored from its saved value on the stack, to point to the DDB. R2 is set to point to the FIB linked to this DDB; then R1 is set to the value of the core bit map pointer in the FIB. The word count in the DDB is saved, and a -64 decimal, the bit map word count is put into the DDB. Then the saved value of R3 on the stack is tested to determine whether this is a GETMAP or WRITMAP call.

If WRITMAP is called for, the DDB transfer block number and buffer address are saved. Then the buffer address is set equal to the core bit map address, the transfer block number is set up and the bit map is written out. The original buffer address, the block number and the original word count are then restored; and an exit is taken through the common routine, S.EXIT.

If GETMAP is called, a test is first made to see if there is already a bit map attached to this FIB. If the FIB bit map pointer is non-zero there is a map attached, so the routine restores the original word count and returns to the caller through the common exit routine with a zero in FIB+0 to indicate successful completion. If no bit map attached the routine clears the FIB link of this FIB to indicate end of chain and then gets the pointer to the beginning of the FIB chain for this unit from the driver. If the pointer to the beginning of the FIB chain is non-zero there is already a bit map in core for this unit. A test is then made to determine if this is DECtape. If it is DECtape then you can-

not open a second output file on the same unit. If not DECtape and a file is open on this unit then link this FIB at the end of the FIB chain and insert core bit map pointer and interleave factor into this FIB, and then exit back to the calling module, after restoring the original word count in the DDB.

When the bit map is not already in core it is necessary to get a 64 word buffer for the map. This is done by a call to the monitor GETBUF routine through absolute location 54. The GETBUF routine allocates buffers from free core. The buffer address is then set in the FIB as the core bit map pointer. The MFD pointer in the driver is moved into the DDB and MFD #1 is read in. Then the routine extracts from it the interleave factor and the bit map block number. Then the bit map is read in and it is linked to the device driver. Restore the original DDB and clear FIB+0 to indicate a successful call. Then leave through common exit routine. The bit map transfer routine also has an imbedded read/write routine in it.

4.4.4 Allocation of Blocks to Linked Files (LBA)

The function of this routine is to allocate one block of a device. This routine is called under the following three conditions:

1. .ALLOC (Create a Contiguous File) to allocate a block for the UFD or to add a block to the UFD.

2. .OPENO (Opening A File For Output) to allocate a block for the reasons under .ALLOC and also to allocate the first block of the file.

3. .OPENE (Opening A File For Extension) to allocate the first block of the extension.

The READ/WRITE processor uses a different routine to do block allocations while a READ/WRITE is in progress (see Section 4.5.2).

Calling Sequence:

```
MOV     PC,R3
EMT     50              ;GETMAP
EMT     47              ;LINK BLOCK ALLOCATE
R0=pointer to DDB
```

Set R3 equal to non-zero value, then make sure there is a bit map in core, by an

EMT call to .GETMAP.

Return Arguments:

Sets FIB+0 to the block number of the allocated block. If FIB+0 was set to 0 then the device is full.

Description:

This routine is re-entrant and executes entirely within the Swap Buffer (MSB) unless it is made core resident. Upon entry, the Registers R0 through R5, the PC and the Processor Status are saved respectively on the top of the stack. R0 is restored from its saved value on the stack, to point to the DDB. The Block Number, Buffer Address and Word Count in the DDB are then saved on the stack. The DDB word count is then set to -64 decimal, for transferring of bit maps. The core bit map pointer is taken from the FIB and put into the DDB as the Buffer Address for transfers, and the Block Number of this bit map is calculated and also put into the DDB. The routine then computes the lowest block number described by the current core bit map. The bit map is then scanned on a word by word basis looking for a word that has a free bit. If no free blocks are found in the current map, it is written out and the block number of the first map in the chain is gotten. The first map is read in and a search is begun of the entire bit map chain for a word containing an unset bit. If none are found the device is full. For a full device the following occurs: The Data Buffer is released by the Monitor, Buffer Release Routine, S.RBUF, and its link is cleared from the DDB. The Open Indicator in the DDB is cleared. The FIB chain is searched for the proper FIB and when found it is released by a call to S.RBUF and it is also unlinked from the FIB chain. The Monitor Diagnostic Print routine then prints an error code indicating a full device.

When a word is found with a free bit a mask is set up to determine which bit of the word indicates the first free block available. The first free bit of the word is set when found, and the block number of the allocated block is set in DDB+26. This block number was being kept and updated during both the word and bit searches. The word count buffer address and block number initially saved are now restored in the DDB, and the common exit routine S.EXIT is taken back to the caller.

4.5 Normal File Processing

4.5.1 Opening Files

Section 3.2.2.1 showed that all .OPEN calls are processed initially by a general routine regardless of the device in use. However, having executed some common operations such as checking the validity of the call and obtaining the necessary data buffer, this routine then checks the device: if it is seen to be file-structured, a FIB is attached to the DDB, Registers are set to provide relevant data and one of two file-management routines is called to complete the processing. The purpose of this section is to describe these routines:

1. FOP - Open an existing file, called therefore for .OPENU .OPENE, .OPENI and .OPENC - Section 4.5.1.1

2. FCR - Create a new file (=.OPENO) - Section 4.5.1.2

Both routines use the technique described in Section 4.3 for moving themselves out of the Swap Buffer if they are non-resident. On completion, they return directly to the user program and at this time, they are responsible for leaving a machine-state exactly as that established by the general processor for non-file devices, i.e.:

1. OPEN switch set in the DDB Status (bit 7, byte 12)

2. Data-Buffer cleared for expected .WRITE following, or

3. Data-Buffer filled for expected .READ following

4. Buffer Pointer set in DDB Driver Word Count, if 2 or 3 effected:

   a. Linked files: START+2
   b. Contiguous files: START

   In addition, the first file Block must be correctly set if .READ or .WRITE may follow, i.e.:

   a. Linked files: in "Next Block" in the FIB
   b. Contiguous files: in "Device Block" in the DDB

4.5.1.1 Open an Existing File (FOP)

The function of the File Open Module (FOP) is to open a file structured device for linked file extension (OPENE), contiguous file input (OPENI), or contiguous file output (OPENC). The routine performs the following operations to accomplish its function. It first makes a directory search to determine if the file exists, since a call to file Open (FOP) requires the file to already exist. The routine then checks if the caller has the access privilege he desires. For an OPENE it allocates the first block of the extension. For an OPENI the first block of the file is read into core.

Calling Sequence:

Registers are expected to be set as follows when this routine is called.

R0 - Pointer to Dataset Data Block (DDB)
R1 - Pointer to insert call arguments

The File Open Module is called by the Common Open Processor (OPN) through execution of code in the DDB. Set up and call sequence is as follows:

```
        MOV     #104043,-(SP)   ;EMT 43 PUT ON STACK
        CMP     (R2),#2         ;SEE IF AN OPENO
        BNE     EX2             ;BRANCH IF OPENE, U, C, OR I
        INC     (SP)
EX2:    MOV     (SP)+,-(R3)     ;EMT PUT INTO DDB
        TSTB    (R5)            ;TEST IF ROUTINE RESIDENT
        BEQ     EX3             ;BRANCH IF RESIDENT
        MOV     (PC)+,-(R3)     ;MUST FREE SWAP BUFFER
        DECB    (R5)            ;THIS INSTRUCTION PUT IN DDB
EX3:    MOV     R3,PC           ;BEGIN EXECUTING CODE IN DDB
```

EMT 43 Calls File Open
EMT 44 Calls File Create

Return Arguments:

None

Description:

When the File Open (FOP) Module is initially entered it tests to see if it is in the Swap Buffer or is permanently core resident. If in the Swap Buffer it calls the Monitor's, Get Buffer routine (GETBUF), to allocate a buffer in free core. It then moves itself into the allocated buffer. The registers R0 and R1 are then restored from the stack. This leaves six words R2-R5, PC and PS on the stack

which will be used for EMT .LOOKUP (Directory Search) return argument.

The File Open Module then issues the EMT .LOOKUP, to see if the file exists. If the file does not exist, the Open Switch in the DDB status word is cleared and an EMT .RLSFIB (Release FIB) is issued to release the FIB Buffer and the Data Buffer. The routine frees the Dataset by clearing DDB+0 and then it must release the free core buffer it occupies. The manner in which this is done is described in the last paragraph of this section; but the common exit routine goes to the user error return. If no error return was set, the Monitor Diagnostic Print routine is called to print an appropriate error code.

When the file exists, the next check is of the Usage Count in the file's UFD entry; if Usage Count is 76 or 77 then the previous error procedure is followed, the file is open. Several more tests are now made. If an OPENC, is it to a linked file, this is illegal if it is. Then for OPENC, OPENU and OPENE requests the UFD entry is looked at to see if the file is locked. If locked the file can not be opened, error handled as before. If not locked, the file is locked then. Then for an OPENE a check is made to see if it is to a contiguous file, error if it is.

All open requests that use the File Open Module (FOP), execute this next sequence. An EMT .SETUPFIB is issued to get a FIB and a Data Buffer. The UIC, the protection code, and the access level necessary for the open request are set up and an EMT .CKACSP is issued to determine if the caller is permitted the access to the file that he desires, (Read or Write). If access is denied the error procedure previously described is performed.

If an OPENC or OPENU the directory entry is written out. If an OPENE an EMT .GETMAP is issued to make sure a bit map is in core, then an EMT .BALLOC is issued to allocate the first block of the file extension. The block number is saved in FIB+4. Then the directory block is written out. Now for an OPENI it is necessary to read the first block of the file into the Data Buffer. The (EOF) end of file switch is set, if this is also the last block of the file.

All the OPEN requests now leave in the same manner. The Data Buffer pointer is set in the DDB; the open switch is set in the status word of the DDB and the Dataset is set to the free state by clearing DDB+0. The routine then tests if it is permanently core resident; if it is it leaves directly through the common exit routine, S.EXIT. If in a free core buffer the routine must release the buffer it occupies before returning to the user. This is done by simulating a JSR to the Monitor's Buffer Release routine by the instruc-

tion, MOV @#S.RBUB,PC. This forces a jump to the Buffer Release routine does not put a return address on the stack, so prior to executing the command the address of the common exit routine is put on the stack. The Buffer Release routine will, when done, take that return and go directly to the common exit routine, to return to the user.

4.5.1.2 Create a New File (FCR)

The function of the File Create Module (FCR) is to create a linked file and open it for output; this routine is called from the Common Open Processor as the result of an OPENO (Open Linked File for Output) request. This routine performs the following operations to accomplish its function. It tests if the file exists, since for an OPENO request it is illegal for the file to already exist. If not, the routine creates a UFD entry for this file. The first block of the file is allocated and the available file information is put into its UFD entry. This routine also performs some setup of the FIB.

Calling Sequence:

Registers are expected to be set as follows when this routine is called.

R0 - Pointer to Dataset Data Block (DDB)
R1 - Pointer to User Call Arguments

The File Create Module is called by the Common Open Processor through execution of code in the DDB. Set up and call sequence is as follows:

```
        MOV     #104043,-(SP)   ;EMT 43 PUT ON STACK
        CMP     (R2),#2         ;SEE IF AN OPENO
        BNE     EX2
        INC     (SP)            ;IF OPENO, EMT 43 BECOMES EMT 44
EX2:    MOV     (SP)+,-(R3)     ;EMT PUT INTO DDB
        TSTB    (R5)            ;TEST IF ROUTINE RESIDENT
        BEQ     EX3             ;BRANCH IF RESIDENT
        MOV     (PC)+,-(R3)     ;MUST FREE SWAP BUFFER
        DECB    (R5)            ;THIS INSTRUCTION PUT INTO DDB
EX3:    MOV     R3,PC           ;BEGIN EXECUTING CODE IN DDB
```

Emt 43 - Calls File Open
EMT 44 - Calls File Create

Return arguments:

None

Description:

When the File Create routine is entered the stack is adjusted to create a six word work area for transferring information between modules, (i.e. Directory Search to File Create). Then the routine tests if it is in the Swap Buffer or is permanently core resident. If in the Swap Buffer it must move itself out. This is done by first calling the Monitor's Get Buffer routine, (GETBUF) to allocate a temporary buffer in free core. The File Create routine then moves itself to the temporary buffer just allocated, and continues execution.

Now the File Create Routine issues an EMT .LOOKUP (Directory Search) to determine if the file to be opened already exists. The Directory Search routine (LUK) returns arguments in the stack work area. If this was an illegal file name, if there was no UIC or if the file existed an error has occurred and file creation can not continue. If an error occurred the open switch in the DDB status word is cleared and an EMT .RLSFIB is issued to release the FIB Buffer and the Data Buffer, and then their links are cleared from the DDB. The Open Switch in the status byte of the DDB is also cleared. If an error return was set up it is taken after releasing to free core the temporary buffer occupied by File Create. The manner in which the free core buffer occupied by the module is released is described in the last paragraph of this section. But since this is an error exit the return is to the users error return address. When there is no error return set the Monitor's Diagnostic Print routine is called to print the appropriate error code.

If none of the above conditions occurred File Create continues by setting up a UFD if there was none when the Directory Search was attempted. This is done by issuing an EMT .GETMAP to get a bit map buffer if necessary; the bit map may already be in core. An EMT .BALLOC (Link Block Allocate) is then issued to allocate a block for the UFD. The bit map buffer is then written out by issuing an EMT .WRITEMAP. The allocated UFD block is linked to the MFD and the MFD is written out. The UFD block is cleared and then written out. The routine then continues as if the file was not found and there was a UFD block which contained an empty directory entry slot. The routine now has the UFD block and a pointer to the free slot.

If the file did not exist in the UFD the routine looks at DDB+2 to see if the Directory Search returned a UFD block number, containing an empty directory slot. If there wasn't an empty slot in the UFD then it is necessary to allocate a block and link it to the UFD. The same procedure is used to

do this as when allocating the first UFD block. If an empty slot existed or when a UFD block is allocated and linked, the following sequence occurs. The directory block containing the empty slot is read into core. The index to the free slot was put in DDB+22, by the Directory Search routine, previously. An EMT .GETMAP (Get a Bit Map) is issued to make sure there is a bit map in core. A test is then made to see if this is a second open on DECtape, which is not permitted. An EMT .BALLOC (Linked Block Allocator) is issued to get the first block of the file. The directory entry is now set up with File Name, Extension, Date, File Type, Usage Count of 77 indicating open for Output, and file protection. File length and Last Block Number are cleared. An EMT .SETUPFIB is then issued to set up file information in the FIB. The UFD block is then written out, and the open switch is set in the DDB Status Byte. If permanently core resident the routine returns to the user directly through the common exit routine. When this routine is executing from a temporary buffer in free core, which will generally be the case, the return is slightly more complex, because the buffer it is executing in must be released. The return to the user is performed as follows: the address of the common exit routine is put on the stack, then the address of the Monitor's Buffer Release routine is moved into the PC. This simulates a JSR to the Buffer Release routine but when it goes to the stack for its return address it will find the address of the common exit, and will return directly to the common exit routine which then returns to the user.

4.5.2 Processing a File

Once a file has been opened for normal input or output as described in section 4.5.1, the user is able to process its data in just the same way as for any other non-file devices, by means of .READ or .WRITE. While the Monitor can operate upon the data within its own buffer, it also need take no special action. However, when a device transfer is necessary to fill or empty the buffer, the Monitor must now determine which actual device block is to be used according to the type of file opened. The purpose of this section is to examine the procedure by which this is done.

As noted in section 3.2.2.2, the principal routine concerned is embedded within the .READ/.WRITE processor and this is discussed first in section 4.5.2.1.

It is shown that during the Creation and Extension of a linked file, this routine can make use of the bit-map segment brought into memory during the .OPEN process, until this indicates that no further blocks remain unassigned. In

this case, the core segment must be changed and section 4.5.2.2 describes the subsidiary routine called, usually from the system-device, to effect the switch.

4.5.2.1 Next Block Determination (RWN cont)

This section in fact covers two sequences in the .READ/.WRITE processor which are executed when examination of the Facilities Indicator in the driver Interface Table shows that the device servicing a dataset is file-structured (see Section 3.3.1). The first sequence is responsible for ensuring that the DDB Device Block is set up correctly for an ensuing transfer. It must also find out in advance the next block to be used in the output of a linked file in order to chain it to the one now being written. The second sequence forms part of the reinitialization following the transfer, in particular checking for the end of a file and at the same time beginning the set-up for the next transfer.

Calling and Exit Sequences:

As an integral part at the .READ/.WRITE processor, there is no specific call or exit and the Register state follows on naturally within the general operation. However the pre-transfer sequence assumes that for a linked file the device block # needed is held in "Next Block #" in the FIB whereas it is expected to be already correctly stored in the DDB when the file is contiguous. The appropriate file-management routine for .OPEN sets up the necessary state in readiness for the first transfer (see section 4.5.1); the clean-up sequence maintains it (see below). A properly established FIB is also essential, though automatic since .READ or .WRITE must be preceded by .OPEN or the request is rejected (see section 3.2.2.2)

Processing:

It follows from the previous paragraph that no further action is required before a transfer if "Type" in the FIB shows the file to be contiguous. Otherwise, the content of "Next Block #" is moved into the DDB. However for DECtape, this could be negative, signifying that the block is to be transferred with the tape moving backwards (see section 4.3.2). Hence the necessary check is made and the Block # and Tape Direction switch in the DDB Status are adjusted accordingly. Once done, preparation for input is also complete.

For output, on the other hand, a search of the core bit map segment attached to the FIB by .OPEN must be made in order

to ascertain the next block to be written and link it to the current block. Thus the map segment origin and range are computed from data in the segment pre-amble (see section 4.1.3). The start block for the search is determined on the basis of the current block number incremented by the IF field in the FIB (decremented if the current block is a DECtape reverse WRITE). The result is converted to a value relative to the segment origin and if it is seen to be either above or below the segment range, it is reset to 0 to correspond to the origin. At the same time, a switch is set to restrict the search, since only two passes over the segment are needed, one from the optimal block to the end and the other from the opposite end up to the optimal block.

The value of the optimal block is used to formulate the address of its representative bit in the map as shown in section 4.1.3. If this bit is 0, it is reset to 1 to claim the block (1). The block number, absolutely reconstituted by addition of the origin, is stored as the file-link in the first word of the data buffer (negated for a reversed current block on DECtape) and output preparation is done. Otherwise the map is examined for the first available block from the optimum, with the current DECtape block direction determining that for the search, on the following basis:

1. The adjacent bits in ascending (or descending) order within the same byte are checked while the block value is incremented or decremented.

2. Successive bytes in the appropriate direction are tested, with the block number adjusted by 8, until one not containing 377 is found.

3. The individual bits of this byte are again checked as in step 1 with corresponding modification of the block number.

The search is terminated as soon as a 0 bit is seen and the result is passed on as shown in the last paragraph. If however, the relevant end of the segment is reached without success, the pass switch is reversed and the second pass of the search is set up as follows:

1. On disks, the search block value is reset to the segment origin.

----

1. Because of the potential re-entrancy of the .READ/.WRITE processor, the actual bit-checking sequence (approx 30 msecs) is carried out at level 7 priority to prevent corruption through interrupt.

2. For DECtape, the direction control is switched to reverse all the search operations and the current block becomes the search block (for reasons given in section 4.2.3.)

The search is completely restarted with the translation of the new potential block into bit-position. Should the second pass also fail to produce a vacant block, the current map segment must be replaced in memory by another. This is effected by a call to the special routine described in the next section. On return, the search is completely reinitialized - in this case always being resumed at the origin and the process is repeated until either a block is found or the device is seen to be full. (This case is handled by the special routine.)

Once the appropriate action has been satisfactorily accomplished, the set-up sequence rejoins the main routine for all devices, to carry out the transfer. On return, the clean-up sequence is entered after the necessary pointers have been reset. This simply increments "Device Block #" in the DDB if the file is contiguous. However, the new value is checked against "End Block" in the FIB and if greater, the EOD marker is set in the DDB. For linked files, the first word of the data-buffer, i.e. the link-word just read in or the one written out after the search, is moved into "Next Block" in the FIB, as required for the subsequent transfer. The EOD marker is again set if such link-word is 0 signifying the end of an input file or one forced upon an output file when no more blocks are available. Normal .READ/.WRITE processing then follows as described in section 3.2.2.2.

Comment:

The process for allocating the next output block described above of course performs in much the same way as the module LBA discussed in section 4.4.4. The two routines differ only, in fact, over their handling of DECtape: LBA is not concerned with the problem of blocks written in reverse since directory extensions or the first file-blocks are always forwards. The appropriate code could be added and the embedded sequence would then be unnecessary. However this is not done presently for two reasons:

a. Although the .READ/.WRITE processor must be resident for the time being, this need not always be the case. The Swap Buffer might not then be available for LBA's use.

b. Even if the Swap Buffer is free, the operation to fetch LBA must potentially occur for each file-block written, i.e. two device transfers

for one (although the chances are that once loaded, LBA in many instances could be used several times over). Moreover there is always the possibility that LBA might be needed at an interrupt level while the program is being serviced perhaps by a Utilities routine. This would be unable to complete and release the MSB and LBA could not be brought in until it did, thus hanging the system.

4.5.2.2 Changing The Core Map Segment (GNM)

The module called by the block allocation routine of the .READ/.WRITE processor described in the last section is responsible for saving the latest state of the in-core segment in the master bit map stored on the device and for replacing it by another segment in accordance with the avowed monitor policy for keeping linked files as far as possible at the front end of the medium. (See section 4.1.3.) It must also provide for the probability that no further blocks remain free for assignment. However, the module does not itself attempt to examine the segment it may load; this is left to the calling routine.

Calling Sequence:

The module depends upon the Register state of the .READ/.WRITE processor at the call, as saved on the stack by the EMT handler:

- R0 = Address of the DDB at "Buffer Address" (DDB+6)
- R1 = 10 (set ready for access to the driver transfer routine - see Section 3.1.2.4; also used as block value increment during map-search)
- R2 = Address of the FIB at the first temporary workspace (FIB+32)
- R4 = Device switch (non-0 for DECtape)

In addition, it expects "Next Block #" in the FIB to be non-0 when first called (1) and 0 if recalled during the same search operation (i.e. because the new segment is also full). Also two unwanted items are on top of the stack.

----

1. This follows from the fact that its normal content cannot be 0, since this as a block is never available for program access (see figures 4-8 & 4-11).

The problem of conflict over MSB usage discussed under "Comments" in the last section also applies to this routine - hence module GMA (see section 4.4.3) is not used. However this problem cannot be solved in the same way, when the operation of changing maps is required much too seldomly: the inclusion of the routine in RWN is thus unrealistic. Nevertheless this infrequency makes it feasible to risk the same conflict for the subsidiary KSB, since this can only occur if the operator happens to use the keyboard at the same time or some parallel device transfer produces a hardware failure. Hence GNM is called by EMT 34, i.e., using a code forcing SAM to load it into KSB (see section 2.3.3).

As noted in the introduction, this routine must handle "Device Full" and it will be shown below that it merely sets the EOD marker in the DDB and exits directly to execute the device transfer. To enable it thus to omit a further search, it expects the instruction before the call to be a branch to the appropriate point. Hence the full call sequence is:

```
BR  RW.TFX ;CALL TFR IF DEVICE FULL
EMT 34     ;CALL GNM
           ;RETURN IF NEW SEGMENT LOADED
```

When entered, GNM uses the actual Register state set by the EMT handler (see section 2.2.1) as follows:

R1 = Stack address of the first word after the call return parameters
R5 = Address of the processor Status Register

Exit State:

On completion, GNM recalls the .READ/.WRITE processor with the original entry state restored, except for the following adjustments to enable re-initialization of the search as noted in the last section:

a. R0 = Address of the DDB at "Word Count" (DDB+10)
   R2 = Address of the FIB at "Next Block" (FIB+0)

b. "Next Block" itself is cleared to remember the entry (see previous paragraph)

c. The unwanted items on the stack are removed and are replaced by a single entry of 10000 (used as a rotation counter in the computation of the segment origin).

Processing:

To reduce the probability of keyboard interruption, GNM first raises the priority level to 4 (cleared by RTI on exit) and then extracts and adjusts the saved Register data as noted above. At the same time the link-word in the data buffer is cleared so that if "Device Full" is seen, the last output transfer correctly terminates the file-chain.

If GNM is called and the device is DECtape, the loading of a new segment is impossible when there is only one (see section 4.3.2). However the call is made, because the two-pass search described in section 4.5.2.1 could, in fact, miss checking the blocks skipped by the application of the IF (unnecessarily perhaps - but it obviates special handling in RWN for a "once in a blue moon" exercise!). Hence for a first entry, GNM just exits immediately with "Next Block" in the FIB cleared, thus forcing a second complete search of the map segment already in core and causing a re-entry to result as "Device Full" (see below).

For disk operations, rather than save the existing content of the program's DDB, GNM uses its own internal version, already set to transfer always 64 words (see section 4.1.3). This is prepared to show the same Driver Address, Device Unit and Busy Flag content (in case of error as under). "Buffer Address" is entered from "Bit Map Start" in the FIB and "Device Block" is computed from the data in the map segment preamble. "Completion Return" is set to cause the driver to call a similar sequence to that used by SAM for dequeuing the driver, checking for transfer failure, clearing the busy flag and taking the System Exit (see section 2.3.4).

If this is the first call to GNM for this particular search, the current map segment must be written out before a new one is loaded. Hence if "Next Block" in the FIB is non-0, the driver is called for output via S.CDB, with Registers and Busy Flag contents saved (see section 2.4.2). .WAIT follows - with a link-block simulated on the stack. On satisfactory completion, the DDB Busy Flag is reset and the block number for map segment #1, as stored in the map preamble, is entered - thus always resuming the search from the front of the complete map as required. When this has been brought in, "Next Block" in the FIB is cleared to show the GNM entry and RWN is recalled. For subsequent calls to GNM within the same search operation, the write-out is unnecessary as it can be assumed that the segment loaded was already full. Therefore it is omitted and reading of successive map segments only occurs.

The form of the exit differs from that previously described. Because of the use of the KSB. GNM, like the other rou-

tines, checks its residency from the first byte (Usage Count); however it cannot call the normal System Exit, when this will decrement the MSB count as described in section 2,3,4. Instead therefore, when GNM is in the KSB it uses a similar form of exit specifically provided for the keyboard language modules, (see Section 6.2.5.3.). It requires that GNM store "RTI" in the third word below the KSB and also that it restore itself the saved Registers for the calling program, The same situation arises, if a device transfer fails; in this case, GNM frees the KSB while executing "IOT" in lieu of "RTI" (fatal error F014) (see chapter 7).

Comments:

Because of the internal DDB, GNM currently cannot be re-entrant. As shown for .INIT in section 3.2.1.1., this is no problem as long as SAM can provide the protection afforded by the re-entrancy switch in the second byte of GNM (see section 2.3.1). In the unlikely event that a user requires its residency, there could be problems (even though this is implied by the check mentioned in the last paragraph).

4.5.3 Closing Files (FCL)

As shown in Section 3.2.2.3, all .CLOSE calls, like those for .OPEN, are initially processed by a general routine. This ensures that the last block of any file open for output is dispatched to the device. It then calls a file-management routine to perform directory operations if the device in use is seen to be file-structured. The purpose of this section is to examine the routine so called. Its prime functions are:

1. Update the master bit map on the medium for a file under creation or extension, using the latest state of the segment in core

2. Update a file Bit Map for a new or extended file on DECtape

3. Complete the UFD entry for the file to show its latest state

4. Release the FIB and Data Buffers to free core; also the Bit Map Buffer if no other files are still using it.

Calling Sequence:

A call to FCL requires registers set as follows:

```
R0 = Pointer to DDB
R2 = Pointer to FIB
R3 = Driver Address
R4 = Modified How Open Code (Code =2)
```

The calling sequence from the Common Open Porcessor is as follows:

```
MOV     #104045,@R1
TSTB    @R5
BEQ     .+6
MOV     (PC)+,-(R1)
DECB    @R5
MOV     R1,PC
```

An EMT 45 is put into the DDB. A test is made to see if the Common Close Processor is in the Swap Buffer: if it is the Monitor Swap Buffer must be free prior to calling the File Close routine. This is done by putting a second instruction which frees the buffer into the DDB and then putting into the PC the DDB address where the next instruction to be executed is stored.

Return Arguments:

None

Description:

When this routine is entered it first calls the Monitor Register Restore routine which takes the saved contents of the general registers from the stack and restores them. The modified How Open Code for the File to be closed is then decoded and appropriate branches are taken.

A linked file opened for output (OPENO), proceeds to close as follows: the block number of the last block written is put into the FIB at FIB + 16, LAST BLOCK WORD. A pointer to the driver location which contains the desired bit map address is calculated and saved. Then a test is made to determine if the device is DECtape: if it is DECtape, it is necessary to update both the permanent bit map and file bit map. For DECtape the permanent bit map is read into core. The core bit map, which describes the old permanent bit map plus the block being used by the new file, is written out. Then the core bit map is bit cleared with the permanent bit map, so that the remaining set bits describe the file bit map. The file bit map is read in, and bit set with the modified core bit map and then it is written out. For Disk it

is necessary to search the bit map chain to get the correct permanent bit map block. The core bit map is then written out. If there are no FIBs left on the FIB chain then the Core Bit Map Buffer is released, by the Monitor Buffer Release Routine. Now the file's Directory Entry is read in and the File information is updated with a length and a last block of file. The block containing the directory entry is then written out. The FIB and Data Buffers are then released and the common exit routine is taken, for returning to the user.

A contiguous file opened for update or output, (OPENU or OPENC) proceeds to close in a common manner as follows.

First the block containing the files UFD entry is read into core. The file Lock bit is cleared and the usage count is decremented. The new file length and the file's last block number are put into the directory entry. The block containing the file's directory entry is then written out, the necessary buffers are released and the routine returns to the user through the common exit routine.

A linked or contiguous file opened for input (OPENI) is closed in the following relatively simple manner. The FIB is released by the Monitor buffer release routine. Then the Data Buffer is released by the same routine. Control is then returned to the user through a return via the common exit routine.

A linked File opened for extension (OPENE) is closed in the following manner. The last block number of the original file, which is stored in the FIB, is tested to see if it is negative. A negative block number means the last block was written backwards, which can only happen on DECtape. If it is negative the reverse bit is set in the DDB and the block number is negated. Then the original last block is read in, and the block number of the beginning of the extension is put into the link word of the block. Then the block is written out, and the reverse bit in the DDB is cleared. The closing procedure now proceeds the same as closing an OPENO file, except when the directory entry is updated the lock bit must be cleared for an OPENE.

4.6 Housekeeping Operations

A program needs facilities other than the mere opening, processing, and closing of files. Quite commonly, these files may be only temporary - as a way of utilizing the bulk-storage medium as an extension of available core. At later stages, these files must be deleted or be made permanent under a new name or different protection. Contiguous files, also, can only be processed if they are already in existence; the means of creating them initially must be accessible. Futhermore, it has been shown that opening a file is only permissible in specified cases, e.g. OPENU is legal only for contiguous files, as .OPENO implies that the file does not exist already. The program may therefore wish to examine the device directory in advance to protect itself against error. Hence the appropriate program requests are provided for these operations and the object of this section is to describe the manner in which they are processed. In general, these requests are valid only when a file is not already opened on the dataset concerned and although the processing module is always called, it ignores the call if the device is not file-structured.

4.6.1 Allocating Contiguous Files

4.6.1.1 Allocate Set-up (ALO)

The function of the module is to create a contiguous file by searching a device's permanent bit map from the end, looking for the appropriate number of unoccupied contiguous blocks. When found the corresponding bits are set in the permanent bit map, and an entry is made in the UFD for the file.

Calling Sequence:

```
MOV     #NUM,-(R6)
MOV     #FILBLK,-(R6)
MOV     #LNKBLK,-(R6)
EMT     15
```

NUM - Number of 64 word units desired

Return Arguments:

A value of -1 returned on top of the stack indicates a successful allocation.

A value of X, not equal to -1, is returned on the top of the stack when allocation was unsuccessful. The value returned, X, indicates the size of the largest allocatable segment available, in 64-word units.

Description:

The module first checks to see if it is in the Swap Buffer. If it is it requests a buffer from free core through the Monitor's Get Buffer routine (GETBUF) and then moves itself into it. If a free core buffer is not available and an error return was not set up, the Monitor's Diagnostic Print routine is called to print an appropriate error code. If an error return was set up it is taken by the common exit routine after releasing the necessary buffers.

The routine tests to see if this is a file structured device, and if not, the routine exits through the common exit routine after freeing the Dataset and cleaning up the stack. The routine also releases the free core buffer it occupies through the Monitor's Buffer Release routine S.RBUF which returns directly to the common exit routine S.EXIT. The success indicator (-1) is returned on the stack.

The EMT .SETUPFIB is issued at this point to get a Data Buffer and a FIB. If no buffer is available, use the previous error exit for no buffer. Now an EMT .LOOKUP is issued. Tests are made for an illegal file name, no UIC, and if the file already exists. If any of these errors are detected, the error return in the file block is taken, if there is one; otherwise, the Monitor requests Error Diagnostic Print.

If there is no UFD it gets the first UFD block by getting a bit map into core, allocating a linked block and writing the bit map out. The UFD is then linked to the MFD. Then the MFD is written out.

If there was a UFD see if the EMT .LOOKUP found an empty slot for the file entry. If no empty slot add a block to the UFD. A new UFD block or the first UFD block is gotten in the following manner. An EMT .GETMAP is issued to make sure a bit map is in core. An EMT .BALLOC is issued to allocate a block for the UFD and then an EMT .WRITMAP is issued to write out the bit map. The block to which this one will be linked is read in, and the linking takes place. If the device was a DECtape and there were no empty slots an error is detected and handled in the previous manner. The new block is linked to the last and the last is written out.

Now move the file name and extension which were put on the stack by the EMT .LOOKUP, into the UFD entry. Clear the usage count, the file start, length, and end. Also put pro-

tection code from the stack into the UFD entry. Then write out the UFD block.

The routine now sets up to call the Contiguous Block Allocator (CBA). The addresses of the Read routine and the Write routine in the ALO module are passed to the Contiguous Block Allocator through registers R4 and R5. The EMT .GETCONTIG (EMT 51), is issued. This allocates the requested number of contiguous blocks if available. If the requested number of blocks were allocated the success indicator (-1) is moved into the return argument position. If the requested number of blocks is not available the largest number of available contiguous blocks is moved into the return argument. An EMT .RLSFIB is issued to release the FIB and the Buffer. Then the routine returns to the caller through the common exit routine. Before returning to the caller it is necessary to release the free core buffer occupied by the Allocate Routine (ALO). This is done by simulating a JSR to the Monitor's, Buffer Release routine, S.RBUF but setting the stack so S.RBUF returns directly to the common exit routine which frees the Swap Buffer and returns to the user.

4.6.1.2 Contiguous Block Allocator (CBA)

This module is called only by Allocate (ALO) and its function is to allocate the requested number of contiguous 64 word units. The function is performed by searching the devices bit maps from the highest number bit map's last word and working towards the beginning of the device's bit maps.

Calling Sequence:

```
R0=ADDRESS OF DDB
R4=ADDRESS OF WRITE ROUTINE IN ALO
R5=ADDRESS OF READ ROUTINE IN ALO

FIB+0=ORIGINAL R1, POINTER TO CALL ARGUMENTS
FIB+2=DIRECTORY BLOCK
FIB+4=DIRECTORY INDEX

EMT 51
```

Return Arguments:

```
R1=Number requested
R2=Largest number available
```

Description:

When this routine is initially entered the Monitor's Register Restore routine is called to reset the registers. Then the addresses transferred through registers R4 and R5 are set up for use of the READ/WRITE routines belonging to the calling module (ALO). Pointers are set to the driver and to the FIB; and the drivers standard buffer size is picked up for the next computation. Then the number of 64 word units desired is converted into blocks. R4 will be the number of blocks.

Read in the MFD block, Then determine if there is a bit map in core for this device and unit number. If there is a map in core, look down the list of maps to find the current one, save its core address and block number then write it out. This is done so it can be restored before returning to Allocate (ALO). We now proceed as if there was no map in core. Read in the last, highest numbered map. Then set up as follows, for bit map search.

4(R6) = Cleared, highest count to date
2(R6) = Number of blocks needed
(R6) = (Address -2) of last map word.

R1 = Address of first word of map
R2 = Address of last word of map
R3 = High block number in map
R4 = Cleared, for counter
R5 = Mask

This section of the contiguous block allocation module (CBA) searches the bit maps to find the number of contiguous free blocks requested. Shift a mask through each word of the map looking for the required number of successive "off" bits. If an "on" bit is reached before the required count is satisfied, update 4(R6) with the highest count to date. Then continue mask search. If the bottom of a map is reached read in the next map and continue. If no more maps to be read, get the directory entry block, clear this entry and write the directory entry block out. Restore original map in core if there was one and then exit back to the Allocate Routine, through the common exit routine, S.EXIT. Upon exit R0 will be restored, R1 will contain the number of units needed and R2 will contain the highest number found.

If the required number of bits have been found begin setting them successively in the bit map. If the end of the map is reached before setting all the necessary bits, write this map out and read in the next. Then continue setting the necessary bits, upon completion write out this block. If there was originally a bit map in core, then read it back

in. Read the directory entry, set last block, length, and start block; then write the directory entry out and exit as described above. If R1=R2 allocation was successful.

4.6.2 Deleting Files

4.6.2.1 Deletion Set-up (DEL)

The function of the module is to delete a file on a file structured device. Delete determines if the caller has the necessary access privilege to delete the file. Delete is divided functionally into deletion of three types of file deletes.

1. DECtape delete -
   Bit clearing of the file bit map upon the permanent bit map.

2. Disk linked file delete -
   Calls the delete linked file module (DLN) to follow the file's chain of links to determine which bits in the permanent bit map to clear.

3. Disk contiguous file delete -
   Calls the delete contiguous file module (DCN) to zero the link words from start through the length of the file. Then reads the bit map, and clears length consecutive bits beginning at start.

   ```
   START  = FIRST BLOCK OF THE FILE
   LENGTH = NUMBER OF BLOCK IN THE FILE
   ```

Calling Sequence:

```
MOV #FILBLK,-(R6)
MOV #LNKBLK,-(R6)
EMT 21
```

Return Arguments:

None

Description:

The delete module first determines if it is in the Swap Buffer. If it is it gets a buffer from free core through the Monitor's Get Buffer routine (GETBUF) and moves itself into it. If a buffer is not available the module releases

the necessary buffers, frees the Dataset and leaves through the common exit routine, S.EXIT which takes the error return address. If there is no error return address, a message is printed by the Monitor Diagnostic Print routine identifying the error.

The routine tests to see if the dataset is busy, and if it is the same error procedure as above is used. Then the device is checked for being file structured. If non file structured an immediate exit is taken back to the call through the common exit routine.

Delete now issues an EMT .SETUPFIB, to get a Data Buffer and set up a FIB. Then an EMT .LOOKUP is issued to determine if the file exists. If the file does not exist it is an error and is handled in the previous manner. The caller's access privilege is then checked (via EMT .CKACSP) to determine if deletion of the file is permitted. If access privilege is denied the previous error handling procedure is followed. The routine then tests if the file is open and if so the previous error procedure is followed.

If no errors to this point the routine calls the Monitor's Get Buffer routine (GETBUF) to allocate a 256 word buffer. Then its address is stored in R2. The R2 Buffer will be used for reading in the device's MFD blocks and bit maps. The Delete routine then determines if the device is disk or DECtape.

If DECtape, the routine determines the block number and index of the directory entry, and then clears the directory entry from the UFD and writes the UFD block out. Note the block containing the directory entry is still in core from the EMT .LOOKUP call. Now calculate the block number of the file bit map. Read the permanent bit map into the R2 buffer. If the file is contiguous an EMT .DELCONTIG (delete contiguous file) is issued which updates the permanent bit map; upon return the permanent bit map is written out. If the file is linked, read in the FBM and bit clear the permanent bit map with the file bit map, and also clear the file bit map. Then write the file bit map and the permanent bit map out. For both linked and contiguous DECtape files the same action is now taken. The 256 word R2 buffer is released and the routine exits as described in the last paragraph of this Section.

If the device is disk an EMT .DELCONTIG (delete contiguous file) or an EMT .DELNK (delete linked file) is issued. Restore the user file directory buffer address and block number, clear the file entry and write the UFD out.

Now for both disk and DECtape files the following exit is taken. An EMT .RLSFIB is issued to release the FIB and the

Data Buffer. If this module is not permanently core resident it is necessary to release the free core buffer it occupies. This is done by setting the address of the common exit routine as the return address for the Monitor's Buffer Release routine; then simulating a JSR to the Monitor's Buffer Release routine, S.RBUF. The common exit routine returns to the user.

4.6.2.2 Deletion of Contiguous Files (DCN)

The function of this module is to assist in deletion of a contiguous file from disk or DECtape. This module performs the File Management operations which include updating the device's permanent bit maps, and clearing the File's block links from start through length. This module is called only from the file structures module delete (DEL).

Calling Sequence:

```
            R0 - Pointer to DDB
            R2 - Pointer to 256 word auxiliary buffer

            STACK -Holds return information from directory
                  search (EMT .LOOKUP).
                  EMT    54
```

Return Arguments:

None

Description:

When the Delete Contiguous Files Module (DCN) is initially entered a test is made to determine if the device is disk or DECtape. Disk File deletes and DECtape File deletes are handled in two separate sections of this module. First Disk deletes will be discussed and then DECtape deletes.

For a contiguous file delete from disk the module first clears the bit map link word and sets the bit map buffer address in DDB+6 (BUFADR.) saves the original word count and sets the DDB word count so that only one word, the link word, will be transferred. The Transfer Block Number in the DDB is then set to the start block of the file. Then the routine loops through the file clearing the links by transferring out one word, the cleared link word until length (file length) has been satisfied. Then, it restores the original word count.

Now the first MFD block is read into core. If a bit map is in core, its address and block number are saved and it is written out. When this is complete, or no bit map was in core, the following occurs. The bit map number correspond-ing to the starting block of the file is computed. The bit map is now read in. The starting block number is converted to a particular bit position in the map. The bits corres-ponding to this file are then cleared. If the end of the bit map is reached before completion, this map is written out and the next map is read in, and the bit clearing con-tinues. Upon completion of the bit clearing, the current map is written out. The necessary buffers are released and the routine exits through the common exit routine. Control is returned to the Delete Module (DEL), from which Delete Contiguous Files (DCN) was called.

For a contiguous file delete from DECtape, a much simpler procedure is required. The file's starting block and length are stored in two registers. Then the starting block of the file is converted to the corresponding bit in the DECtape permanent bit map which is in core when this routine is called. The bits in the permanent bit map corresponding to the file blocks are cleared beginning at start and going through length bits. The common exit routine (S.EXIT) is then used to return to the Delete Module (DEL), from which Delete Contiguous Files (DCN) was called.

4.6.2.3 Deletion of Linked Files (DLN)

The function of this module is to assist in deletion of a linked file from disk. This module accomplishes its func-tion through performance of File Management operations; which include updating the devices permanent bit maps and clear the link word of the file's blocks. This module is called only from the file structures module delete (DEL).

Calling Sequence:

```
        EMT     53
```

Return Arguments:

None

Description:

For a linked file delete from disk the Delete Linked Files (DLN) module first reads in MFD #1. The present word count is saved and the word count is set for 64 word transfers. If a bit map is in core it is written out and its buffer ad-

dress and block number are saved. Set up for one word transfers and determine the block number of the first bit map, and read it in. The routine reads in the link word of the first 6 blocks of the file, then writes out a zero link into the first six blocks, unless the file occupies less than 6 blocks the last link is saved. See if the bit map in core covers the modified file blocks. If not write it out and read in the proper bit map. Now clear the corresponding bits in the bit map. The routine continues in this loop till the file end. The saved link is now the next block of the file to be read in. The module continues through the file in the above manner saving links, clearing link word and clearing bits in the appropriate bit map. When a zero link is detected, the end of the file has been reached. Then write out the current bit map. If a bit map was originally in core read it back in, release the necessary buffers and exit back to the delete (DEL) module, through the common exit routine.

4.6.3 Appending Files

4.6.3.1 Append General Routine (APP)

The function of this module is to append two linked files together. Appending involves linking FILEA to FILEB and adjusting the file entry for FILEB. If it happens that the device is DECtape module AP2 is also called to modify the DECtape bit maps. FILEA ceases to exist as a separate file and is now part of FILEB.

Calling Sequence:

```
        MOV     #FILEA,-(R6)
        MOV     #FILEB,-(R6)
        MOV     #LNKBLK,-(R6)
        EMT     22

        FILEA - Address of the UFB for File A
        FILEB - Address of the UFB for File B
```

Return Arguments:

None

Description:

This module first checks to see if it is in the Swap Buffer and if it is it gets a buffer from free core through the

Monitor's Get Buffer routine and then moves itself into it. It then tests to see if an assignment has been made or if the dataset is busy. If either condition occurs an error is detected and handled as follows. The stack is adjusted for an exit; if an error return was set up the necessary buffers are released and the error return is taken through the common exit routine. If an error return was not set up an appropriate error code is printed by the Monitor Diagnostic Print routine.

If the above errors were not detected an EMT .SETUPFIB is issued to get a Data Buffer and a FIB. The routine also makes a call to the Monitor's Get Buffer routine (GETBUF) to allocate in free core an auxiliary buffer of 256 words; its address is kept on the stack. A test is made for a file structured device, and if not file structured the necessary buffers are released and the common exit is taken back to the caller.

If file structured an EMT .LOOKUP is issued to see if FILEB exists. If the file doesn't exist or if it is open, the previous error handling for a busy dataset (B file block) is performed. A test is now made to see if the file is contiguous; if it is this is an error and the error procedure is followed. If no errors detected an EMT .CKACSP is issued to check for protection violation, i.e. does the caller have write access. If a protection violation occurs the previous error handling procedure is used. The same procedure as above is used for FILEA. If FILEA is the same as FILEB release the necessary buffers and exit back to the caller. If not the same, the file name and extension are cleared from the directory entry for FILEA, and this directory block is written out. The routine saves in the FIB the following information for FILEA: block number, directory address, the start, length and end of the file. The directory entry for FILEB is now read in. The new file length is computed and set in the directory entry along with the new file end. The updated directory entry for FILEB is then written out.

Now the device is checked as to whether it is disk or DECtape. If the device is DECtape and the block number of the start block of FILEA is negative, make it positive and set the DECtape reverse bit in the status word of DDB. The reason for the above is that the first block of a DECtape file is always written in the forward direction. If the link to a block is negative this indicates it was written in the reverse direction. Thus the link to a block written in the forward direction must be positive.

Now for both disk and DECtape move the start block of FILEA into the FIB. Then read the end block of FILEB into core. Move the link to FILEA start block into FILEB end block. Then write this block out. This links FILEA to FILEB.

Now if the device is DECtape an EMT .APNDP2 (Append Part 2), is issued to set the bits in the FILEB bit map that were set in the FILEA bit map, and clear the set bits in the FILEA bit map, then write out the FILEB bit maps. Upon return from the Append Part 2 Module or if the device was disk the following procedure is followed. DDB+0 is cleared to free the Dataset, an EMT .RLSFIB is issued to release the FIB and the Data Buffer. The 256 word auxiliary buffer is released through a call to the Monitor's Buffer Release routine. Finally the address of the common exit routine, S.EXIT is put on the stack. Then a JSR is simulated to the Monitor's Buffer Release routine to release the free core buffer occupied by the Append Module. Upon completion the Buffer Release routine returns directly to the common exit routine which returns to the user.

4.6.3.2 Special Append Operations on DECtape (AP2)

The function of this module is to modify the file bit maps when FILEA is appended to FILEB. This routine is called only by the module Append (APP), and only when the device is DECtape.

Calling Sequence:

```
EMT     55
```

Registers are set as follows when this routine is called; also FIB+0 and FIB+2 contain the following pertinent information.

R2=FILE B Block Number
R3=FILE B Directory Address

FIB+0=FILE A Block Number
FIB+2=FILE A Directory Address

Return Arguments:

None

Description:

This routine first reads in the file bit map for FILEA and then the file bit map for FILEB. The routine then sets the corresponding bits in FILEB bit map that are set in the FILEA bit map and clears the set bit in FILEA bit map. Now it writes out FILEA and FILEB file bit maps. Now exit back to the append (APP) module through the common exit routine.

(Before reading in the file bit maps a subroutine is called to calculate the block containing the desired file bit map and the index into the file bit map.)

Subroutine Inputs:

    2(R6)=Directory Block
    4(R6)=Index into Directory

Subroutine Outputs:

    2(R6)=File Bit-map Block
    4(R6)=Index into File Bit-map

4.6.4 Renaming Files (REN)

The function of this module is to change the name and protection code of an existing file. This is done by reading in the OLDNAM UFD entry, checking the caller's access, then moving the NEWNAM and protection into the OLDNAM UFD entry and writing out the UFD block. This routine is called directly by the user.

Calling Sequence:

```
MOV     #NEWNAM,
MOV     #OLDNAM,
MOV     #LNKBLK,
EMT     20
```

OLDNAM-is the address of the existing file's filename block.

NEWNAM-is the address of the filename block containing the new information.

Return Arguments:

None

Description:

The module first checks if it is in the Swap Buffer and if it is it gets a buffer from free core, through the Monitor's Get Buffer routine (GETBUF) and then moves itself out of the Swap Buffer into the free core buffer. The routine then checks if the dataset is busy. If it is busy the Rename call was illegal. If an error return address in the OLDNAM file block was set up, this is taken by the common exit routine after releasing the free core buffer that Rename occu-

pies. If no error return, control goes to the Monitor Diagnostic Print routine which identifies the error.

For the case of a non-busy data set an EMT .SETUPFIB is issued to get a Data Buffer and a FIB. A check is then made to see if the device is file structured. If not file structured an immediate exit is taken after releasing the necessary buffers.

If the device is file structured an EMT .LOOKUP is issued to determine if the old named file exists. If the file doesn't exist, if the caller is not the owner of the files or if the file is in use an error is detected and handled as follows. If an error return had been set up in the OLDNAM file block, it is taken by the common exit routine after releasing the necessary buffers, through the Monitor's Buffer Release routine which returns directly to the common exit routine. For the case of no error return address the Monitor Diagnostic Print routine, prints an identifying error code.

If the file does exist and the caller is the owner, a second EMT .LOOKUP is issued to see if the new file name is in use. If the file was in use or a .LOOKUP error occurred the previous error handling procedure is used. If the file name is available and no errors were detected the block containing the directory entry of the old file is read in. The new file name, extension, and protection code are moved into the file entry and the block is written out. The FIB and Data Buffers are released by issuing an EMT .RLSFIB. It is now necessary to release the free core buffer that Rename occupies. This is done by putting the address of the common exit routine on the stack; then a JSR to the Monitor's Buffer Release routine is simulated but no return is put on the stack. When the Buffer Release routine returns it goes directly to the common exit routine which returns to the caller.

4.6.5 Protecting Files (PRO)

The function of this module is to protect a file from automatic deletion upon logout (finish command). This routine is called directly by the user and protects his named file by setting bit 7 of the protect byte in the file UFD entry.

Calling Sequence:

```
        MOV     #FILBLK,-(R6)
        MOV     #LNKBLK,-(R6)
        EMT     24
```

Return Arguments:

None

Description:

The module first checks to see if it is in the Swap Buffer or is permanently core resident. If in the Swap Buffer the module gets a buffer from free core, through a call to the Monitor's Get Buffer routine (GETBUF), and then moves itself into it. The routine now tests the status byte in the DDB to see if the Dataset is busy. If the Dataset is busy the module cleans up the stack, clears DDB+0 to free the Dataset, and sees if an error return address was set by the user; then releases the free core buffer it occupies through the Monitor's Buffer Release routine which returns directly to the common exit routine which does the user's error return. If no error return was set an error identifying message is printed, through the Monitor's Diagnostic Print routine.

If the Dataset was not busy the module issues an EMT .SETUP-FIB to get a Data Buffer and File Information Block. The device is then checked for being non-file structured. If non-file structured an immediate exit is taken, after releasing necessary buffers, similar to error exit above except common exit routine goes to the user's call instead of error return.

An EMT .LOOKUP is now issued to see if the file exists and is not open. If the file does not exist or is open the previous error procedure is followed. When the file exists and is not open the protect bit can now be set. Bit 7 of the protection byte of the file's UFD entry is then set, and the UFD block is written out. The UFD block was in core from the EMT .LOOKUP call. The FIB and Data Buffer are released by issuing an EMT .RLSBUF. Then the routine releases the buffer it occupies by a call to the Monitor's Buffer Release routine which returns directly to the common exit routine, which returns to the user.

4.6.6 Directory Status (DIR)

The function of this module is to determine if a particular file exists within a particular UFD. The routine also determines the permitted methods of access for a file and optionally determines the starting block of a file.

Call arguments

1. Optional function not desired.

```
MOV     #FILBLK,-(R6)
MOV     #LNKBLK,-(R6)
EMT     14
```

2. Optional function desired.

```
MOV     #FILBLK,-(R6)
CLR     -(R6)
MOV     #LNKBLK,-(R6)
EMT     14
```

Return Arguments:

1. Optional function not desired

   (R6) = number of blocks in the file, in binary
   2(R6)=file indicator word

2. Optional function desired

   (R6) = starting block of the file
   2(R6) = number of blocks in the file, in binary
   4(R6) = file indicator word

FILE INDICATOR WORD

BIT 0=1	.OPENC allowed
BIT 1=1	.OPENI allowed
BIT 2=1	.OPENE allowed
BIT 3=1	.OPENU allowed
BIT 4=0	File not in use
BIT 4=1	File in use by another dataset
BIT 5=1	Dataset already has a file open
BIT 6=0	File is Linked
BIT 6=1	File is Contiguous
BIT 7=0	.OPENO allowed(file does not exist)
BIT 7=1	.OPENC not allowed (file exists)
BITS 8-15	Protection Code

Note: If a file is protected against READ access it will be signaled as non-existent to a caller other than the owner.

Description:

The module first tests to see if it is in the Swap Buffer; if it is it gets a buffer from free core through the Monitor Get Buffer routine (GETBUF), and then moves itself into it. A test is then made to see if the Dataset is busy. If the

Dataset is busy the return information is put on the stack, (bit 5=1) the buffer is released and the routine exits back to the user. If the dataset was not busy an EMT .SETUPFIB is issued to get a Data Buffer and a FIB. If no buffers are available an error is indicated either by an error diagnostic being printed or return through the error return address in the link block, if present.

If the device is not file structured, the file exists bit is set in the indicator word. Then if the "output allowed" bit is set in the driver the OPENC permitted bit is set in the indicator word and the file exists bit is cleared. If the input bit is set in the driver the OPENE bit is set in the indicator word and the file exists bit remains set. The return arguments are now put on the stack. Zero is put on as the file length, and as the starting block if this is requested. Then the file indicator is put on. The necessary buffers are released and the routine exits back to the caller.

If the device is file structured an EMT .LOOKUP is issued to determine if the file exists and its characteristics. If an illegal file name is detected the routine takes the error exit in the file block, or calls Error Diagnostic Print, if there is none. If the file is not found by the .LOOKUP, the return information is set on the stack, the necessary buffers are released and the routine returns to the caller. If the file exists and no errors have been detected an EMT .CKACSP is issued to check if read access is permitted. If read access is not permitted the same return is taken as if file not found. The available information is now used to set the appropriate bits in the file indicator word. A second EMT .CKACSP is issued to see if write access is permitted. On the return appropiate conditions bits are set in the file indicator word, based on a sequence of tests, if the caller has write access. If the file is locked it can not be written on. If the file is linked OPENU and OPENC are allowed. If the file is contiguous, OPENE is allowed. The starting block of the file is put on the stack if it was requested, and then the length and file indicator word are put on the stack. The Data Buffer and FIB are released by issuing an EMT .RLSFIB. DDB+0 is cleared to free the Dataset; if not permanently core resident the free core buffer that the Directory Status Routine occupies must be released. This is done by simulating a JSR to the Monitor's Buffer Release routine and then having it return directly to the common exit routine which returns to the user.

4.7 Magnetic Tape Structure

Magnetic Tape is being treated individually because of its specialized structure. Magnetic Tape Structure is File oriented but has no directory structure as previously defined. Its file structure handles contiguous record Files in sequential ordering. The files are identified by the use of labels making up the first record of each file.

A Magnetic tape file is a collection of sequential records bounded by end of file records or by bottom of tape marker and an end of File record. For "TRAN" non-File structured processing the records of a File may be from 2 to 32767 words long. For File structured "OPEN/CLOSE" processing, each record of a File is 256 words long except for the First record, which is the file label and which is 7 words long.

In order to perform label searching to support multiple files on magnetic tape it is necessary to know when the last file on a Tape has been passed. This is accomplished by having CLOSE, for an Output File, write a logical end of tape (LEOT), which is a null-File. A null-File is 3 end of File records with no intervening data records. New Files which are added to the tape write over the old LEOT and write a new one after their last record.

Each File created by OPEN Processing has as its LABEL a seven word first record of the following form:

```
LABEL+0   FILE
LABEL+2   NAME
LABEL+4   EXTENSION
LABEL+6   UIC
LABEL+10  PROTECT CODE (BYTE)
LABEL+11  UNUSED (BYTE)
LABEL+12  DATE CREATED
LABEL+14  UNUSED
```

When special Functions involving the operation of magnetic tape are requested of the mag tape driver a Special Function Block is used for information transfer. A pointer to the special functions block is passed to the driver in DDB+2. The Special Function Block has the following form:

```
SFBLK+0   Special Function Code          Byte
            1.  Rewind Tape And Unload
            2.  Write End of File
            3.  Rewind Tape
            4.  Skip Records
            5.  Backspace Records
            6.  Set Density And Parity
            7.  Tape Unit Status
```

SFBLK+1	Words to Follow (3 or larger)	Byte
SFBLK+2	Tape Unit Status	Word
SFBLK+4	User Specified Count or Control Information	Word
SFBLK+6	Residue Count	Word

The allowable functions involving operation of Magnetic Tape are listed below and are described in detail in the following two sections.

Standard Monitor Functions:

.OPENI
.OPENE
.OPEND
.OPENC
.CLOSE
.READ
.WRITE
.TRAN

Special Functions

REWIND AND UNLOAD
WRITE END OF FILE
REWIND
SKIP RECORDS
BACKSPACE RECORDS
SET DENSITY AND PARITY
READ TAPE UNIT STATUS

4.7.1 Opening Files on Magnetic Tape (MTO)

This routine is called from the Common Open Processor only; when an OPENI, OPENO, OPENC or OPENE is requested for magnetic tape. OPENU to magnetic tape is illegal because it is inconsistent with magtape structure; if this is attempted it is rejected by the Common Open Processor. The magnetic tape Open module is called from code executed in the DDB, put there by the Common Open Processor. This is to avoid executing an EMT in the Swap Buffer.

The function of this module is to perform special OPEN features for Magnetic Tape. The Open Functions include rewinding the tape, and checking if the device is already open. If the OPEN is for output processing it checks the Write Lock Bit. If the Write Lock Bit is on when opening for out-

put, an action monitor request will be issued to insert the File protect ring before continuing. The routine then looks at the first record of each file, comparing the file name, extension and UIC of the label with the OPEN request information, until a match is made or the logical end of tape is reached.

OPENI - This open requires that the File be found. If not found, this is an error. Exit is taken through the address in the file name block, if there; or through the Error Diagnostic Print, otherwise.

OPENE - If the File is found the tape will skip to the end of the File. If the File is not found the File label is written over the logical end of tape (LEOT).

OPENO - If the File is found, an action diagnostic is issued at this point. A new tape may be mounted and the search re-occurs or a continue is given without replacing the tape and the OPEN behaves as if it just wrote the File label. If the File is not found the file label is written over the logical end of tape (LEOT).

OPENC - Same as OPENE except if the file is found it does not skip to the end of file.

OPENU - Illegal

Calling Sequence:

Registers are set as follows when this routine is called.

R0 - DDB Address
R2 - File Name Block Address
R4 - Device Driver Address

The following is the instruction sequence to call the MTO routine from the Common Open Processor, by execution of code in the DDB. R3 points to DDB+30. R5 points to the Swap Buffer "in use" byte.

```
MOV  EMT+63,-(R3)
TSTB @R5
BEQ  .+6
MOV  (PC)+,-(R3)
DECB @R5
MOV  R3,PC
```

The above sequence of code proceeds as follows:
An EMT 63 is put into DDB+26. Then a residency check is

made to determine if the Common Open Processor is in the Swap Buffer. If it is in the Swap Buffer a second instruction is put in the DDB at DDB+24. The second instruction frees the Swap Buffer before the EMT is issued. Now the DDB address where the code is to be executed is put into the PC. Then the code in the DDB is executed.

Return Arguments:

None

Description:

When this routine is first entered the Monitor Register Restore Routine is called to perform its function of restoring the general registers. The word count for File Labels is set in the DDB. The Mag Tape Open Routine then builds a Special Function Block and puts its address in DDB+2. This is to enable the Open routine to request special functions directly from the magnetic tape driver. A test is then made to see if the device is OPEN. If the device is OPEN, the error return address is taken if one was set up; otherwise a fatal error diagnostic is printed by the monitor. If the device was not open then a special function call is issued directly to the device handler, to rewind the tape to the beginning of tape marker (BOT). A test is made to see if the request was an OPEN for output. If OPEN for Output a test is made if Write Lock is on; if not on, continue; if Write Lock on then an action diagnostic is printed. When the program continues after action message the check is made again.

Now a Label Block is built from information in the File Name Block: this includes Filename, extension, UIC and protect code. The creation date is then set in the Label Block. A check is made to see if there was an assignment, and if an assignment was made this over-rides the file name block information. The over-riding information is then put into the Label Block. A search is then made of the existing files on the tape by comparing the Open request file name, extension and UIC with the Label Records on the tape.

If a match is not made on the First Label Record, the file is skipped by a special function call to the driver, and the next label record is read and a match attempted. If the logical end of Tape is detected before a match is made, the search is terminated. A test is made to see if the logical end of tape occurred after the physical end of tape, and if it did an error occurred. The error return address is taken if one was set up, otherwise a Fatal error diagnostic is printed by the Monitor. If LEOT occurred before physical end of tape the tape is back spaced by a special functions call and a test is made if the OPEN request was an OPENI.

If the request was an OPENI the file must be found, so an error is detected if not found. If an error the error return address is taken if set up; otherwise a fatal error diagnostic is printed. If the request was an OPENE, OPENO or OPENC the Label Record built in core is written over the logical end of tape. The Open Flags are set in the Driver and the DDB. The stack is cleaned up and the common exit routine is taken back to the user.

If a LABEL Record is found to match the Open request information, then the type of Open is decoded. For an OPENO if a match was made an action diagnostic is printed to allow a new tape to be put on and the search procedure to re-occur. If a new tape is not put on and continuation is requested OPEN behaves as if it just wrote the file label, the open flags are set and the routine returns to the user through the common exit routine. For an OPENI or OPENC, the Open Flags are set, the stack is cleaned up, and the routine exits back to the user. For an OPENE, a special function is issued to skip records to the end of file marker, and then another special function is issued to backspace over the end of file marker. The Open Flags are set and the routine returns to the user via the common exit routine.

4.7.2 Special Operations

The special functions appropriate to Magnetic Tape described in the introduction to Section 4.7 are called by means of the general routine SPC described in Section 3.4.1. In this case, the call sequence uses the address of a Special Functions Block as its second argument, where the Block itself is set up as noted, i.e.

```
        MOV #SPFBLK,-(SP)       ;PASS SPF BLOCK...
        MOV #LNKBLK,-(SP)       ;... LINK-BLOCK
        EMT 12                  ;CALL SPC
```

The processing carried out by the Magnetic Tape Driver is examined in its description the TM11/TU10 Magtape Driver document (DEC-11-RIMB-D), assuming that a valid code is given in the Block. On completion, control returns to the calling program, through the Special Functions general routine, with data set into the Block as follows:

1. Tape Unit Status (SPFBLK+2)

BITS	CONTENT
0-2	Last Command 0 = Offline 1 = Read 2 = Write 3 = Write EOF 4 = Rewind 5 = Skip Record 6 = Backspace Record
3-6	unused
7	1 = Tape after EOF (before EOF if last command was BSP)
8	1 = Tape at BOT marker
9	1 = Tape after EOT marker
10	1 = Write Lock on
11	Parity 0 = odd, 1 = even (default = odd)
12	0 = 9 track; 1 = 7 track
13-14	Density 0 = 200 BPI 1 = 556 BPI 2 = 800 BPI 3 = 800 BPI dump mode

Note: Tape unit status is returned in SFBLK+2 for all special functions.

2. Residue Count

When special function requests are issued to skip records or backspace records and the skip or back space count is not satisfied before termination, the residue is returned to the "Residue Count" word (SFBLK+6). Refer to description of SKIP RECORDS and BACKSPACE RECORDS.

Page 4-70

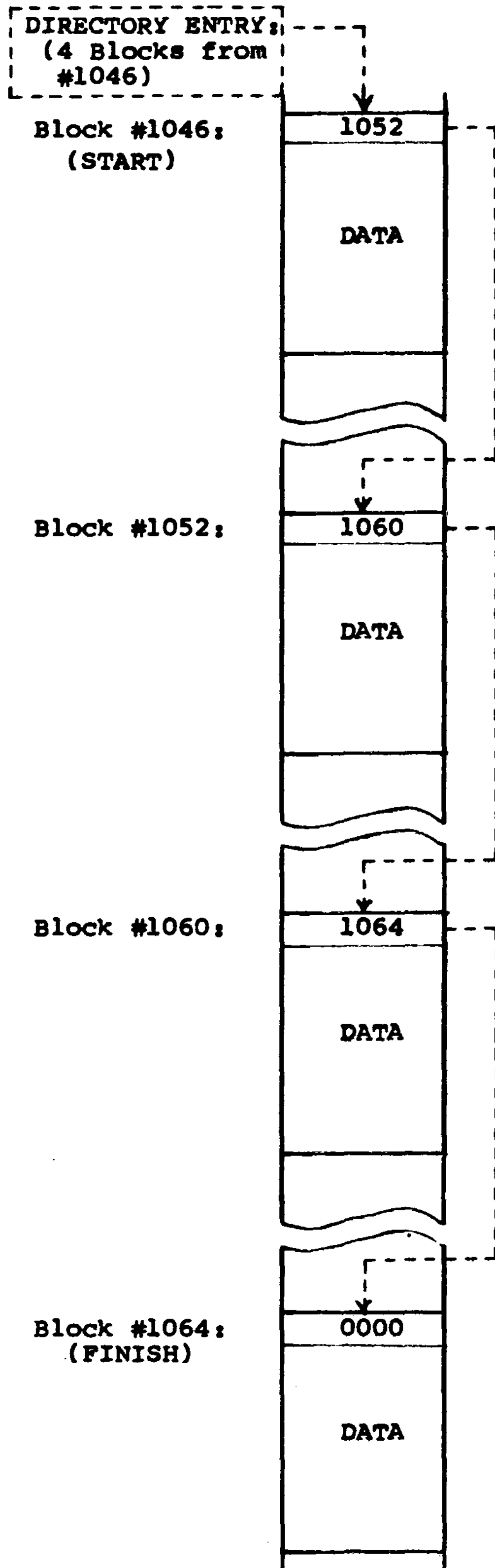


Fig.4-1: Linked File Format.

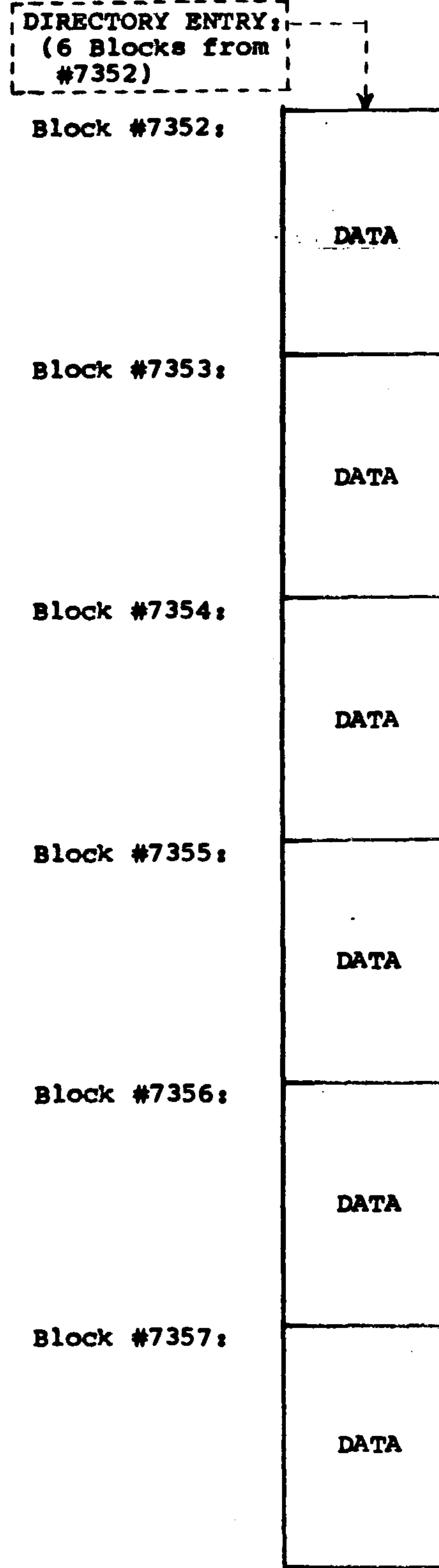


Fig.4-2: Contiguous File Format.

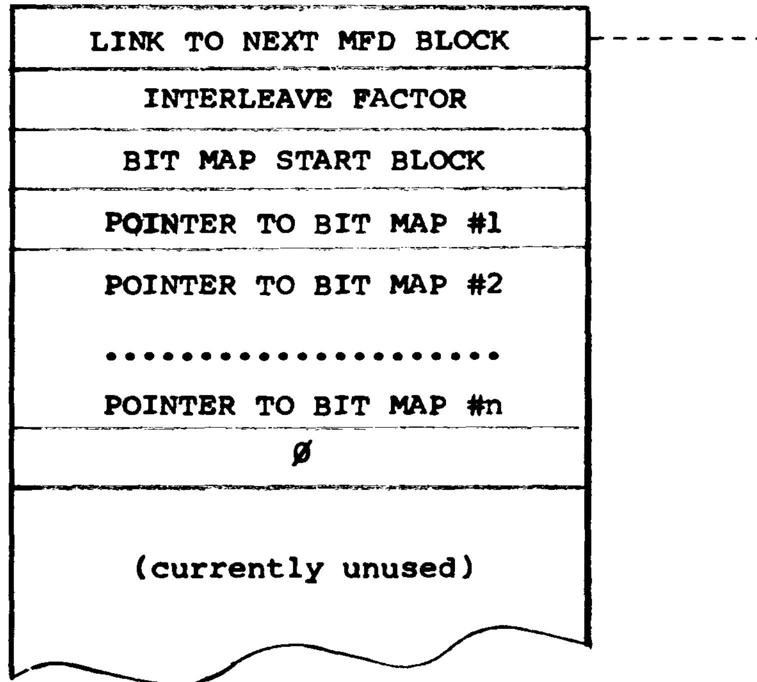


Fig.4-3: Master File Directory Block #1

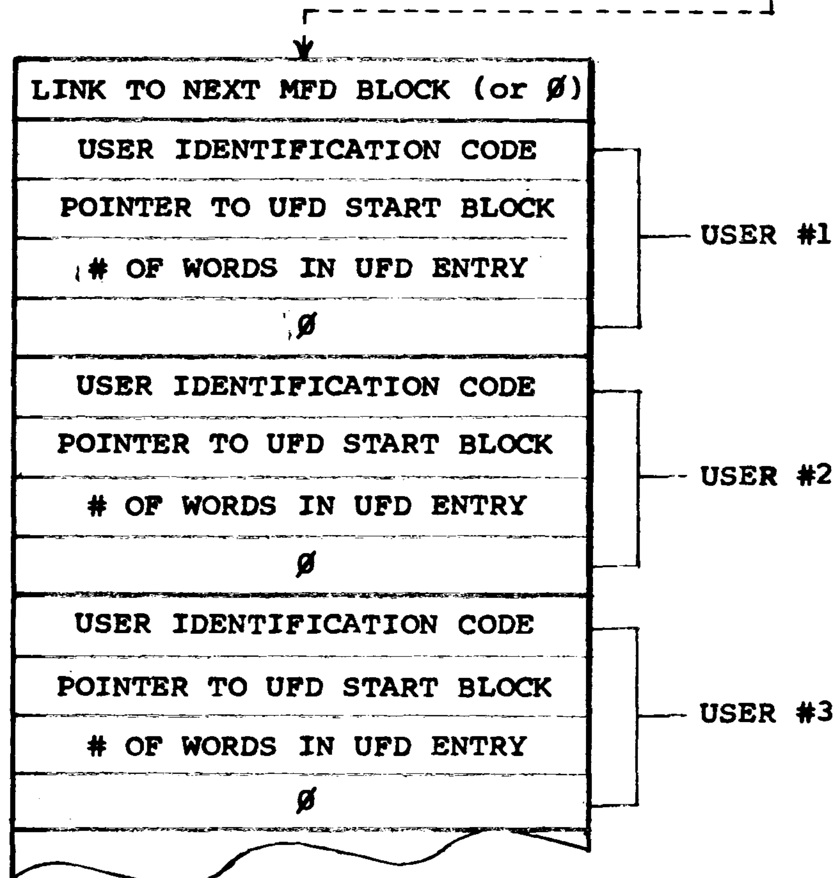


Fig.4-4: Master File Directory Block #2

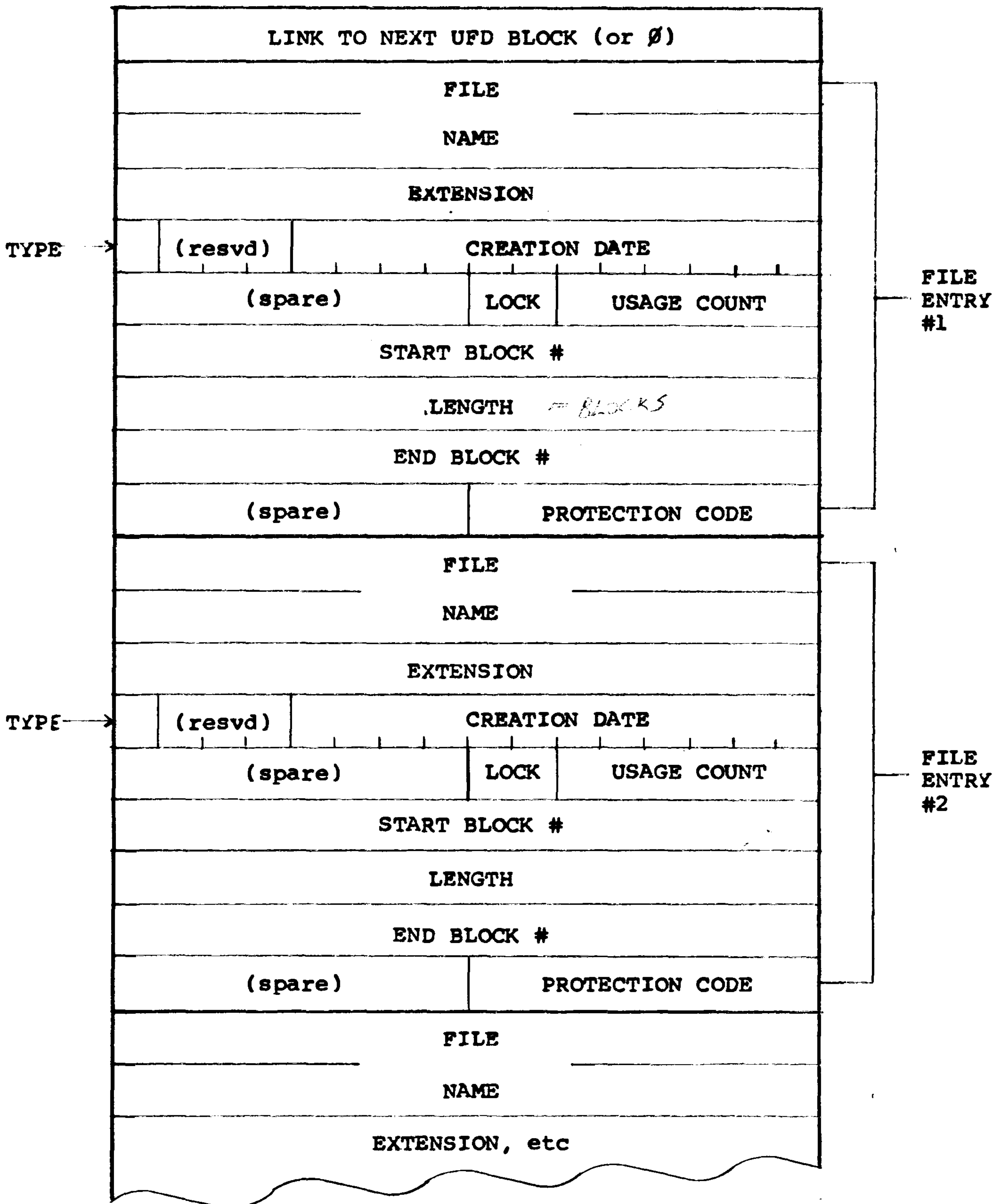


Fig.4-5: User File Directory Block

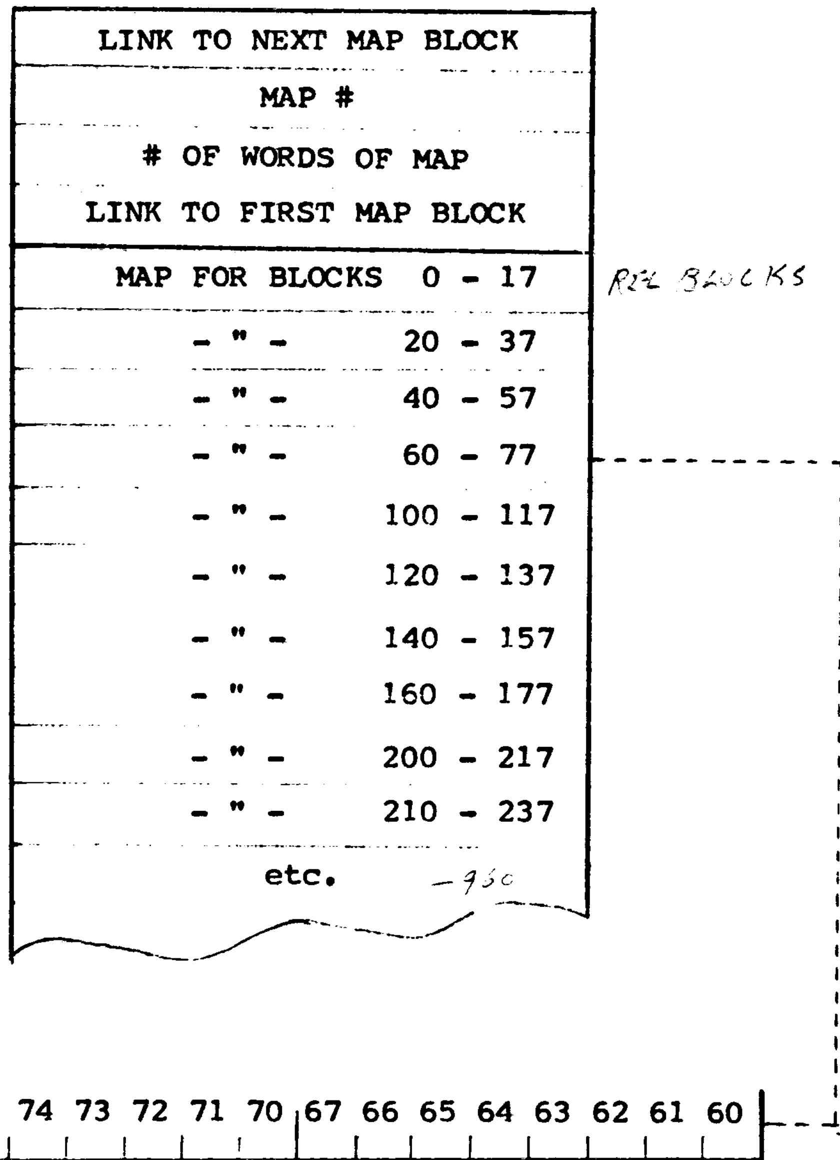


Fig.4-6: Bit-map Segment Format

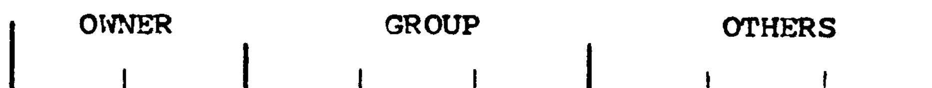


Fig.4-7: Protection Code Format

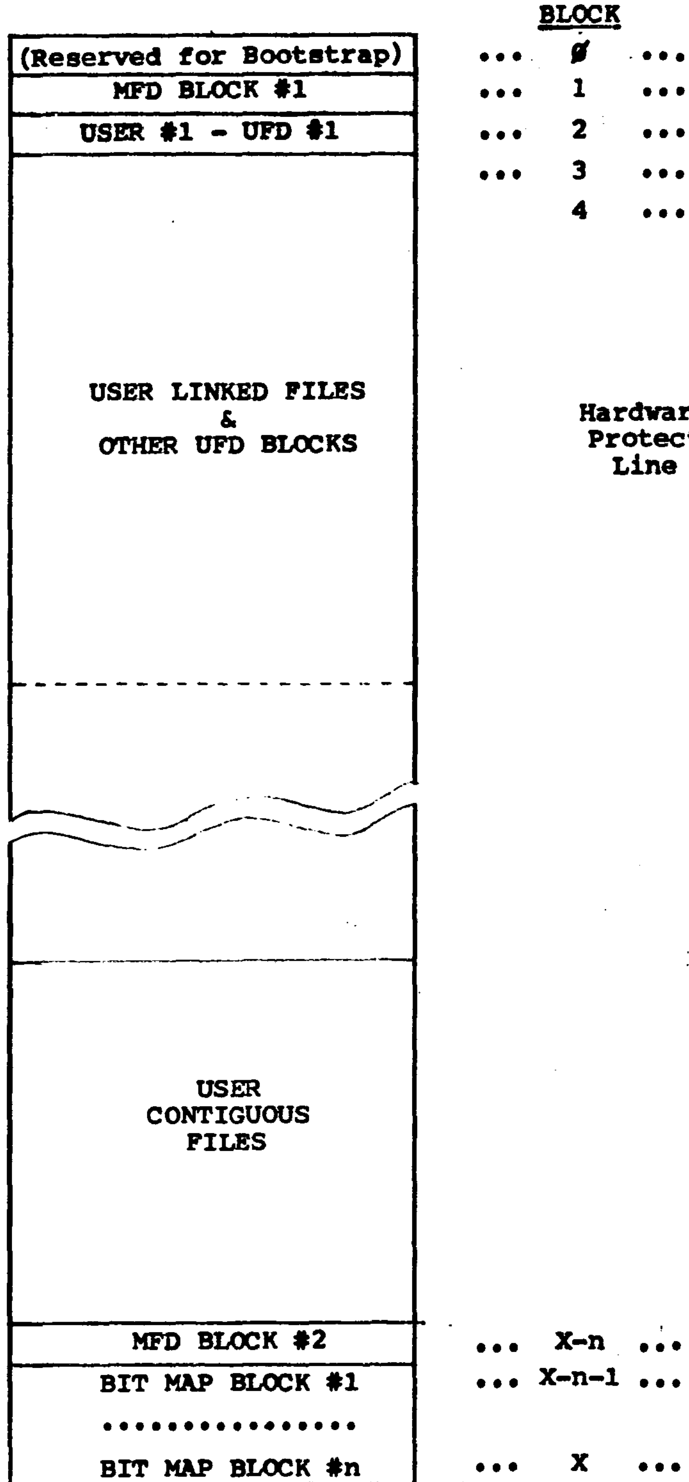


Fig.4-8: Non-system Disk Format

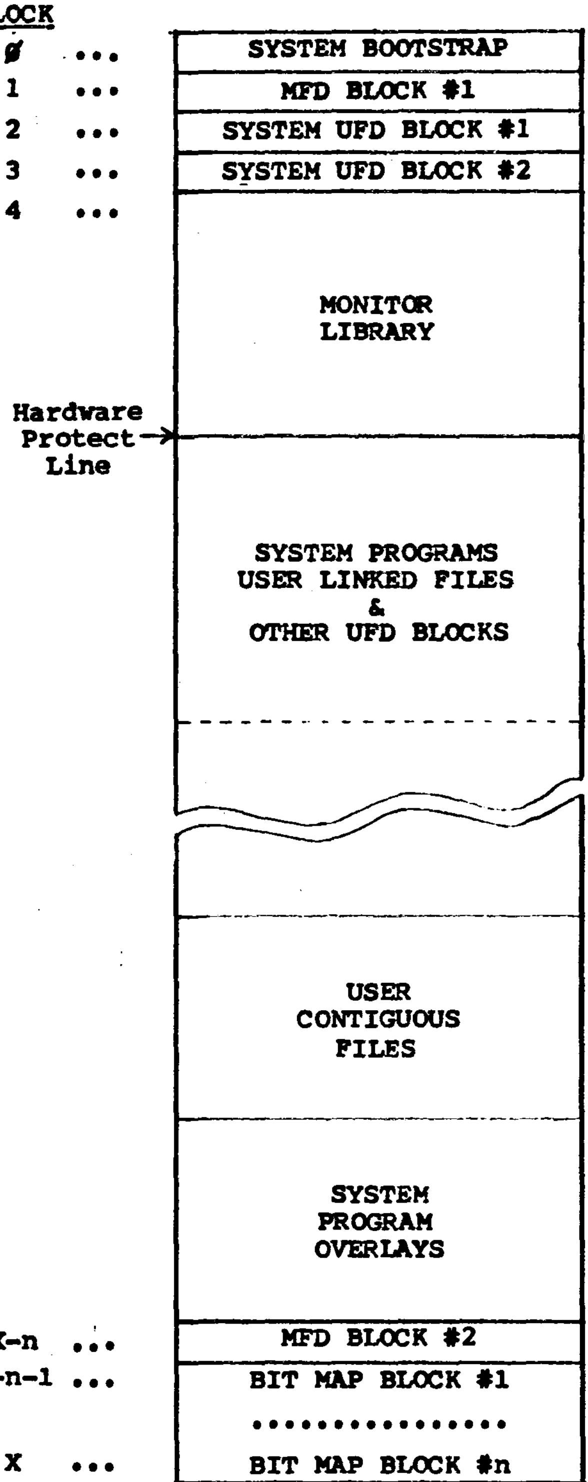


Fig.4-9: System Disk Format

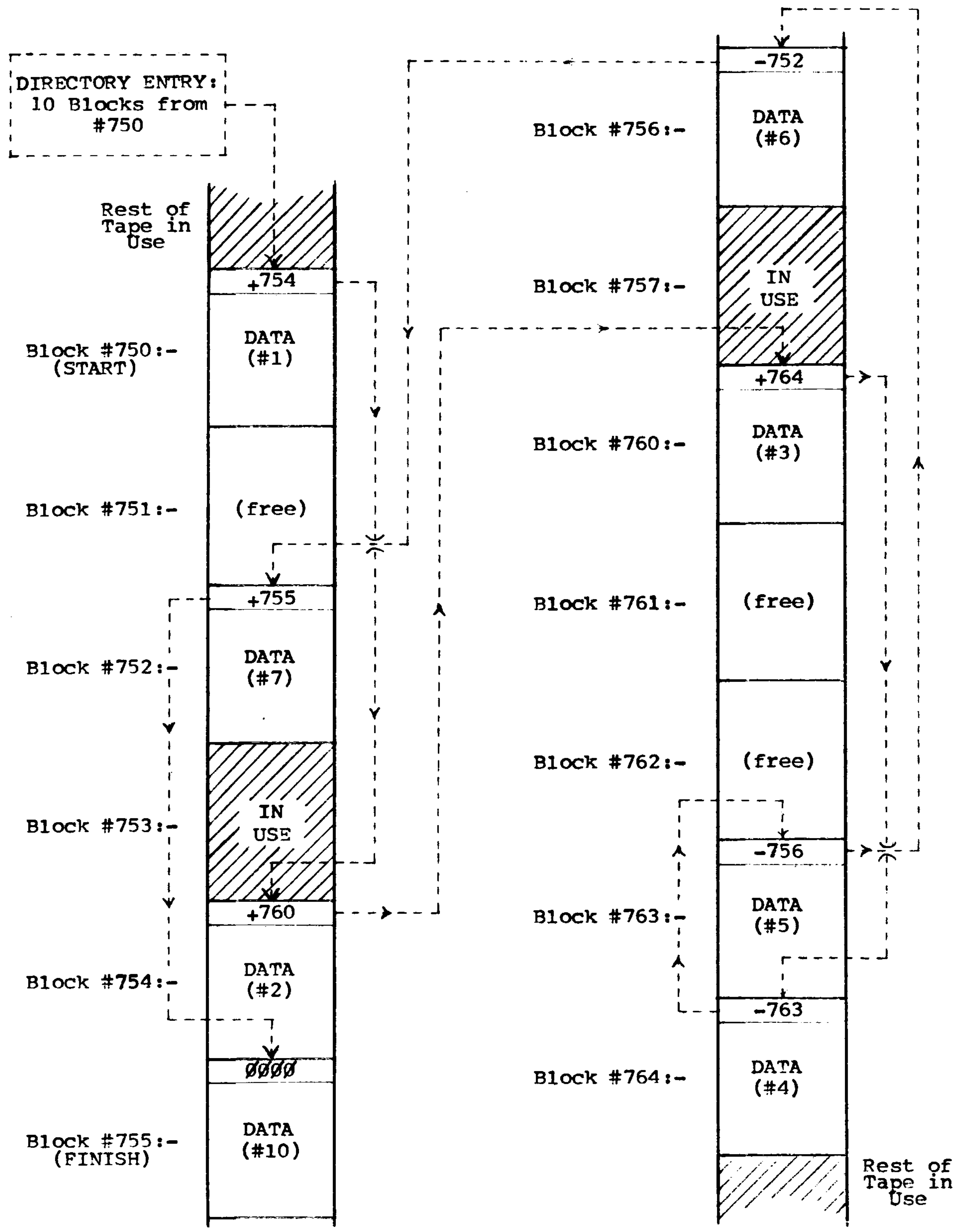


Fig.4-10: Possible Linked File on DECtape

BLOCK	
Ø	(Reserved for Bootstrap)
1	USER LINKED FILES
	USER LINKED FILES
7Ø	FILE MAPS: 1 - 7
71	- " - 8 - 14
72	- " - 15 - 21
73	- " - 22 - 28
74	- " - 29 - 35
75	- " - 36 - 42
76	- " - 43 - 49
77	- " - 50 - 56
1ØØ	MFD BLOCK #1
1Ø1	MFD BLOCK #2
1Ø2	UFD BLOCK #1
1Ø3	UFD BLOCK #2
1Ø4	MASTER BIT MAP
1Ø5	USER LINKED FILES
	USER CONTIGUOUS FILES
1Ø77	

Block #7Ø:- (36-word entries)

- FILE #1 BIT-MAP
- FILE #2 BIT-MAP
- FILE #7 BIT-MAP
- SPARE

Block #1Ø2:- (9-word entries)

(Link to #1Ø3)

- FILE #1 ENTRY
- FILE #2 ENTRY
- FILE #3 ENTRY
- FILE #4 ENTRY
- FILE #5 ENTRY
- FILE #6 ENTRY
- FILE #7 ENTRY
- FILE #8 ENTRY
- FILE #25 ENTRY
- FILE #26 ENTRY
- FILE #27 ENTRY
- FILE #28 ENTRY

(Spare)

Fig.4-11: DECtape Format

MONITOR FILE-MANAGEMENT MODULES

NAME	PURPOSE	EMT CODE	CATEGORY
DIR	Check Directory Status	14	1
ALO	Allocate Contiguous File	15	1
REN	Rename File	2Ø	1
DEL	Delete File	21	1
APP	Append File 'A' to File 'B'	22	1
PRO	Keep File on Log-out	24	1
FOP	Open File (I, E, U & C)	43	2
FCR	Create Linked File	44	2
FCL	Close File	45	2
LUK	Search Directory	46	3
LBA	Allocate Linked Block	47	3
GMA	Get Bit-map into Core	5Ø	3
CBA	Get contigous blocks	51	4
CKX	Check Access Privileges	52	3
DLN	Complete Linked file delete	53	4
DCN	Complete Contiguous file delete	54	4
AP2	Complete DECtape Append	55	4

Fig.4-12

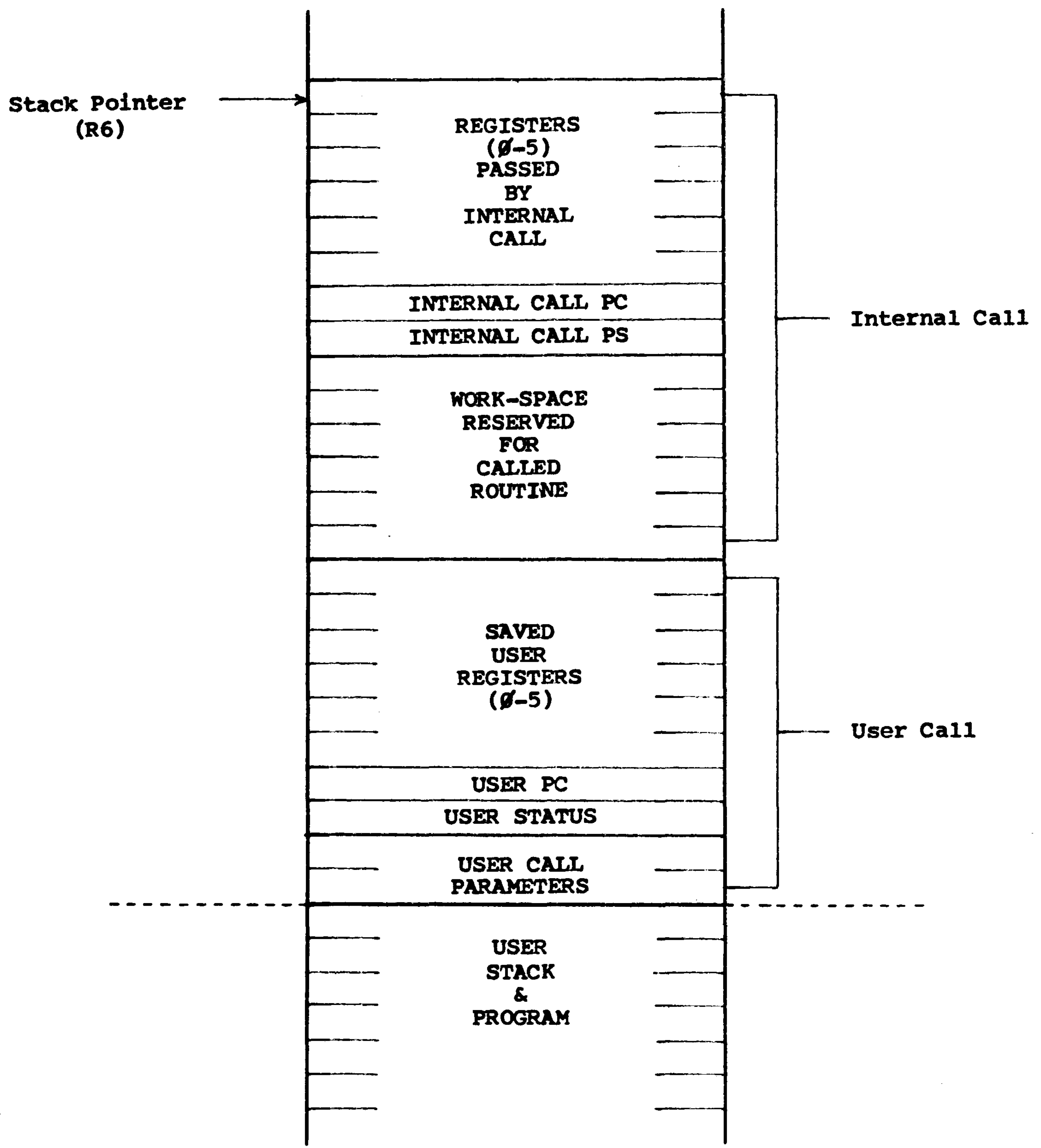


Fig.4-13: Potential Stack State - Internal File-management Subroutine Call

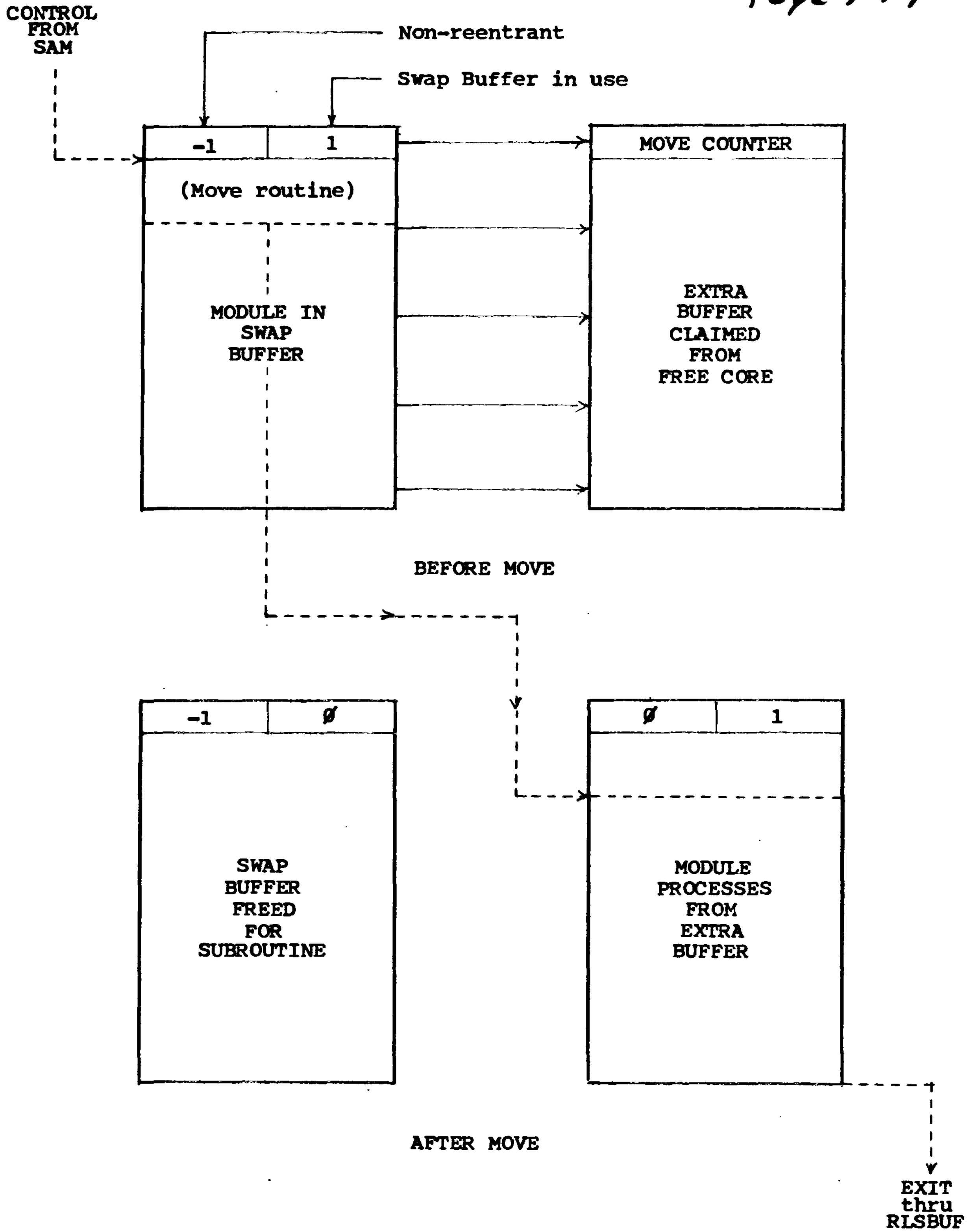


Fig.4-14: Use of Swap Buffer in File-handling Operations
(First-level Routine)

FILBLK:

ERROR RETURN ADDRESS	
ERROR STATUS	HOW OPEN CODE
FILE-NAME (in Radix-5Ø)	
EXTENSION (in Radix-5Ø)	
USER IDENTIFICATION CODE	
(spare)	PROTECTION CODE

Fig.4-15: User File Block

Offset		
	(DDB)	
DDB + 24	DAT ENTRY ADDRESS	
26	POINTER TO FIB	

Offset		
FIB + Ø	NEXT BLOCK #	
2	Ø	HOW OPEN CODE
4	EXTENSION START BLOCK #	
6	TYPE	(spare)
1Ø	(spare)	
12	START BLOCK #	
14	# of BLOCKS	
16	LAST BLOCK #	
2Ø	INDEX INTO DIRECTORY BLOCK	
22	DIRECTORY BLOCK #	
24	(spare)	PROTECTION CODE
26	INTERLEAVE FACTOR	
3Ø	BIT-MAP POINTER	
32	BIT-MAP Q LINK	
34	TEMPORARY WORK-SPACE	
36		

Fig.4-16: File Information Block

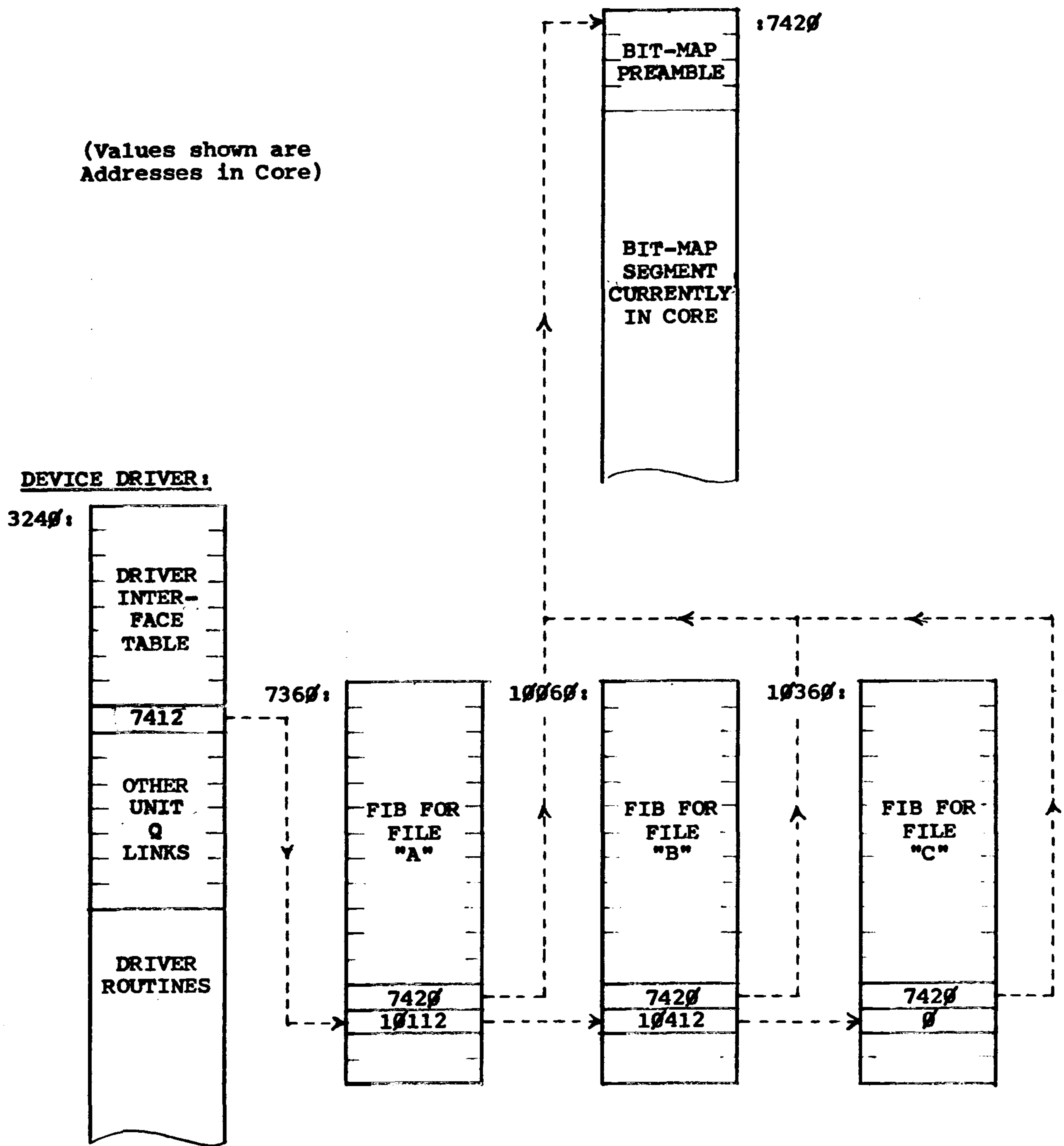


Fig.4-17: FIB Linkage to Bit-maps

CHAPTER 5

OTHER PROGRAM SERVICES

As well as assisting the user in handling I/O as described in the last two chapters, the DOS Monitor offers a range of other program services such as loading and unloading the program itself and some general utilities. As with I/O services, each operation is handled by a particular module, which is normally brought from the system-device when required or can be resident in memory. The purpose of this chapter is to describe each of these modules.

Section 5.1 illustrates the process by which a program is brought into core following a console RUN or GET command. Section 5.2 and 5.3 discuss the utility packages allowing the user to obtain SVT information and for some common radix conversions. The Command String Interpreter which provides a further means for device-assignment while the program is actually under execution, is the subject of section 5.4. Finally the process of unloading the program upon completion is covered in section 5.5.

Since all the modules in the category are potential users of the Swap Buffer, they use the techniques noted in section 2.3.5. Unless these techniques are particularly relevant, no further reference to them will be made.

5.1 Program Loading

All programs which run under DOS, including the Systems Programs themselves, are stored, regardless of the device, in the load format into which they are converted by LINK-11. This is done for two main reasons:

a. It follows the concept of device-independence discussed in chapter 3 in that loading from any source is possible. While in most cases users are expected to hold their programs on some form of bulk-storage medium, for which a core-image would produce a faster load, it is still possible that paper-tape or cards might be used. Unlike the others, neither reader of these latter media provides any form of guarantee on the accuracy of the data, a particularly important requirement when that data is an executable program. Instead, checking must be done by software and the formatted binary mode used for the load image amply enables this. For simplicity therefore all devices are handled similarly.

b. Even on the bulk-storage media, there are advantages in being able to maintain the programs as linked files rather than contiguous as implied by core-image format. They can be stored on any part of the medium rather than rigidly at one end (especially significant if the medium is DECtape); they can be extended without the probability of wastage of the medium because the old area is no longer large enough and a new one must be provided, leaving the other possibly locked out of further effective use. Also of course, a further step in program preparation is avoided, admittedly at the expense of a slightly longer load-time.

Before the Loader itself is discussed it is worthwhile therefore to examine the load-format as it is currently defined. The illustration at figure 5-1 shows the structure of a typical executable program module. Basically it consists of a series of formatted binary blocks, each starting with the two words of header information - mode and size - and ending with the checksum the format implies (see section 3.2.2.1). The first data word of each block is a load address for the words which follow. The first block of the module is a Communication Directory (COMD). This contains general information including the program name, its load and start addresses and size; this information may be followed by a list of EMT codes taken from the Monitor Library on the

system-device by LINK11 because they correspond to Monitor modules for which the program has specified core-residency by global reference (see section 2.1.4). (If this list is too long for the 128-byte maximum buffer size of LINK-11 a second block may be used). The program itself uses the subsequent blocks and finally a single-word Transfer Block provides a possible automatic start address (if included in the .END statement in the program source - or 1 by default) (1)

Loading actually begins in the transient Monitor section which occupies memory in the absence of any user program (see section 6.5). This routine processes the console RUN and GET commands requesting the load and performs the following initial operations:

1. Verify the existence of the specified program module and if present, read its first part into a buffer of standard size for the device, claimed from free core.

2. Transfer appropriate data from the COMD general information section into the SVT (see section 2.1.1)

3. Move onto the stack:

   a. Program start address or 0 to signify RUN or GET

   b. The list of the Monitor modules to be loaded (if any given in the COMD)

   c. Count of the items in the list - or 0

   d. Program load address

4. Set registers as follows:

   R0 = Address of a DDB established in free core to service the load module dataset
   R1 = 0 for linked files; 1 for contiguous file or a non-file-structured device; sign bit negative if device is DECtape
   R2 = Start address of the allocated buffer

---

1. The format described in fact is exactly the same as that prescribed for the Absolute Loader in the Paper-tape System on PDP-11 (see DEC-11-GGPC-D). Hence, LINK-11 output can also be used outside the DOS environment assuming of course that no Monitor modules are specified. In this case, the COMD is also irrelevant; however, because this is given the same load point as the the program it is immediately overwritten and in all but very few instances is no problem.

R3 = End address of the buffer
R4 = Address of next byte to be processed (after COMD blocks)
R5 = (Currently irrelevant - see Section 6.6)

The routine then calls the Loader proper to continue the transfer of the program itself into memory as shown below in section 5.1.1. This in its turn calls a further module which takes care of the loading of the specified Monitor modules and of any general clean-up in readiness for program execution. This is described in section 5.1.2.

5.1.1 Program Loader (LDR)

The Program Loader routine is responsible for transferring the program blocks from the load module and storing their data in the correct memory locations until the terminal transfer block is detected. To avoid an extra buffering stage and thereby save time, the transfers are carried out by use of .TRAN rather than .READ. The Loader itself performs its own checks upon the formatted binary mode of the data before it is stored. On completion, the Monitor-module Loader is called to complete the load operation.

In most cases, the Loader is expected to be non-resident and in this case, like other Monitor modules, it is brought into the Swap Buffer when required. The same also applies however to the .TRAN processor which the Loader calls as noted in the last paragraph. The Loader solves the possible conflict in a manner similar to that used by the file-management modules in a like situation (see Section 4.3.2.): If, before it begins its processing, it detects that it is in the Swap Buffer (by examination of the Usage Count in its first byte), it moves itself into another buffer claimed from free core and releases the Swap Buffer for the use of .TRAN.

Calling Sequence:

The Loader expects the Register and Stack-state detailed above. Otherwise its call is merely:

```
EMT 61
```

Processing:

The processing sequence followed by the Program Loader is illustrated at figure 5.2. Basically the following operations are performed:

1. Reset Registers to the values passed by the transient Monitor. Remove the return PC and Status from the stack.

2. Link the established DDB, as shown by the address in RØ, to an internal Link-block in order to process the program load module as a dataset in the normal manner (the Link-block used by the transient Monitor may of course disappear underneath the program as it is stored)

3. From the data in R2,R3 and R4 build an appropriate TRAN-block - also include an EOD flag if this is already set in the DDB (see section 3.2.1.3)

4. The current stack may also be over-written by the loaded program, so move the data passed by the transient Monitor onto a new one set immediately below the Program Load Address. (This requires two moves since there is always the possibility the old & new areas might overlap one another).

5. Clear the whole of memory between the Program Load Address and the start of the area reserved for the Paper-tape System Loaders.

6. Process the data in the buffer, a byte at a time - since there is no constraint that binary blocks must be complete words - as follows:

   a. Look for the first non-Ø byte. If this is not 1 and the next not Ø the format is incorrect; so reject the load with a fatal Format error message (FØ22)

   b. From the next two bytes, build the Byte Count for the number of data bytes to be read and initialize a checksum accumulator.

   c. Form the Load Point for the data from the next two bytes and set a memory pointer accordingly. Adjust the Byte Count for the six bytes just read. If this is now Ø, the block just processed must be the terminal Transfer Block; hence release the dataset linkage and call the Monitor-module Loader (see next section).

   d. Otherwise store bytes in memory via the pointer set in step (c) until the Byte Count goes to Ø.

   e. During steps (c) and (d) add each byte into the checksum accumulator. Then add the data checksum byte. If the result is Ø, return to step

(a) to look for the next data block. Otherwise, stop the load with a fatal Checksum error (F021)

7. If, after any byte has been processed in the previous operation, the data buffer is seen to be empty, call the device to refill it as follows:

   a. Check for an EOD seen at the last device transfer by examination of the TRAN-block flag. Since a terminal Transfer Block should come first, this must be an error, so reject the load on the grounds of incorrect format as in step 6(a) above.

   b. Using the flag set into R1 when called, check if the load module is a linked file, as described in section 4.1.1.1. If not, increment the device block number in the TRAN-block. Otherwise extract and store the first word of the data-buffer as the device block. If this is negative and the device is DECtape, (see section 4.3.2), turn it positive but set the TRAN-block flag to force reverse tape motion.

   c. Call .TRAN, followed by .WAIT. On completion, stop the load as in step 6(a) above, if any device parity failure has been detected as shown by the TRAN-block flag. Otherwise adjust the buffer and pointer for an incomplete block transfer if this is signalled in the TRAN-block. Reset the byte-extraction pointer, skipping the first word if again the input is a linked file, and continue.

Exit State:

When the module performing the final load phase is called, the contents of Registers are of no consequence; the stack remains as on entry except for the Program Load Address entry which is removed during the processing.

Comments:

It will be noted, perhaps, that the Loader module does not, at step 6(c), attempt to release the buffer it may itself be occupying. At this time any buffer allocation ceases to be meaningful. As was shown in section 2.4.2, the memory area controlled by the Monitor Buffer Allocation Table is origined always at the current end of the resident Monitor. During the next load phase, this end is likely to move up

memory to include further modules requested by the program and the free core area is then different. Hence, the Buffer Allocation Table is cleared by the next module and this automatically reclaims outstanding buffers. In fact, the call to release the input dataset is made purely as a simple way of cleaning the Monitor DDB chain (see section 3.1.2.3) and the DDL entry for the input device (see section 2.1.3).

This module cannot be re-entrant, because of the Link- and TRAN-blocks it must set up internally. However by its nature, such requirement is irrelevant. Likewise, there is no restriction on its residency, for its current purpose, (though it is perhaps inconceivable that there is any point in its being in core other than when required).

5.1.2 Monitor Module Loader (LD2)

The load process performs two main functions:

a. It extends the resident Monitor to include the additional modules specified by the program and currently identified by their EMT codes in the list on the stack passed by the transient Monitor.

b. It ensures that the Monitor is correctly primed for the program execution run and if required automatically begins the run.

In this particular case, any transfer from the system-device in order to load a required Monitor routine is controlled by the Loader module, using its own internal DDB. Since the loaded routine would overlay any buffer allocated from the then free core, the module cannot repeat the technique adopted by its predecessor to use .TRAN, as shown in the last section. The whole operation must be effected within the Swap Buffer. This of course means that re-entrancy is again out of the question. The remarks on the last paragraph of section 5.1.1 still apply.

Calling Sequence:

As noted in the previous section, this module expects a stack-state as passed by the transient Monitor section, starting with the count of Monitor modules to be loaded. No additional information is needed. Hence its call is:

```
EMT 62
```

Processing:

The processing sequence in this case is relatively simple. No illustration is therefore included. Basically the following steps are taken:

1. Remove the data pushed by the EMT call from the stack, i.e. saved register contents and return PC and Status.

2. Clear the Monitor Buffer Allocation Table completely (see section 2.4.2) and reset the TOB entry in the SVT and corresponding stack-stop (Lowest allowable address for stack expansion) to remove any buffers still allocated (see section 2.1.1 and "Comments" in the previous section).

3. Collect from the stack the count of Monitor routines to be loaded and if zero, proceed to step 9. Otherwise determine the address of the system-device driver via the DDL start stored in the SVT and put it into an internal DDB (see Section 3.1.2.3). Also set the DDB Completion Return to use the same sequence as SAM for dequeuing the driver via S.CDQ, checking the transfer validity and clearing the DDB busy state (see Section 2.3.4).

4. Using the EMT code passed on the stack, extract the corresponding MRT entry for a required routine (see section 2.1.2). If the value collected is even, the routine is already resident, so ignore the request and go for the next.

5. Otherwise compute the size of the routine from the MRT data and adjust the SVT entries for EOM & TOB (and its stack-stop) accordingly.

6. Prepare the DDB for the transfer, setting its Busy Flag and using the MRT data to store Device Block # and Word Count and the old EOM for Buffer Address. Also make the MRT entry show the routine's start address now that it will be resident. (Because of the Usage Count & Reentrancy Switches in the first word of the routine, this must be old EOM+2. (see section 2.3.2)) .

7. Save current Registers and call the system-device driver to effect the transfer, via S.CDB (see section 3.1.2.4). Request .WAIT until done (as indicated by the DDB becoming idle because of step 3).

8. Restore the saved Registers and if the count shows that more Monitor routines remain to be loaded, return to step 4.

9. Make a final adjustment to EOM, TOB and stack-stop to reserve two words for Device Assignment Table linkage (see sections 3.2.1.2 and 6.4.10). Also ensure that the SVT entry for WRA points to the System Wait Loop (see section 2.1.1).

10. Simulate a normal System Exit for a routine using the Swap Buffer (see section 2.3.4) and allowing for the user's specification for RUN or GET as follows:

    a. Return Status - move the RUN/GET switch passed by the transient Monitor up the stack and insert 0 to force no priority level.

    b. Return PC - if the RUN-GET switch is non-0, the value is the automatic start address, hence the correct exit point (1). Otherwise replace the switch by the address of the System Wait Loop to force the necessary GET state.

    c. Saved Registers - clear the next 6 words on the stack to represent R0 through R5.

11. Set the Monitor/User switch (MUS) in the SVT to show correct state (see Section 2.1.1):

    a. Program in and Running (1,1) for RUN
    b. Program in but Waiting (-1,1) for GET

12. Also for GET, force acceptance of keyboard command input by setting the Listener input underway switch (see section 6.3.1) and output '$' to inform the operator that such input is expected.

13. Take the System Exit to release the Swap Buffer and continue as set in step 10 with Registers cleared and the stack starting immediately below the loaded program.

----

1. If the .END statement in the program source did not, in fact, supply a start-point, this is set to 1; hence the automatic start results in a fatal error (F342) because this is an illegal address. The user can recoup by means of a keyboard BEGIN with the correct address supplied.

5.2 General Utilities Package (GUT)

As noted in section 2.1.1, the Monitor retains system-state information in the SVT. In addition, the TRAP instruction has been reserved for the user; its vector is also within the resident Monitor. In order to allow the user to access this area, a General Utilities package is provided. While this is perhaps unnecessary under the present DOS because the user cannot be denied such access if he chooses to go more directly, later systems developed for a PDP-11 with possible hardware protection might prevent this. Moreover it cannot be guaranteed that those systems will necessarily structure their information tables in the same way. Thus in the interest of upward compatibility, access through the Utilities package is strongly recommended.

Calling Sequence:

The package is called by a single EMT (41); to identify a particular function an identifier code is passed as a call argument. Basically the functions can be divided into two main groups and the codes reflect this. Each group also has a different calling sequence:

a. PUT functions (1-77) - in which the user is supplying information for appropriate storage by the Monitor. For these, the user pushes both the information and the code as call arguments; the Monitor returns control to the program on completion with the stack clear, e.g.:

```
MOV   #DATA,-(SP)      ;PASS INFO FOR STORAGE...
MOV   #CODE,-(SP)      ;... & IDENTIFIER
EMT   41               ;CALL GUT
```

b. GET functions (100-177) - by which the user asks for information already stored. In their case, the user provides only the identifier in the call; the Monitor returns the required information on the stack with the user then responsible for its removal:

```
MOV   #CODE,-(SP)      ;PASS IDENTIFIER
EMT   41               ;CALL GUT
```

The currently assigned codes and their calling sequences are shown in the Programmer's Handbook.

Processing:

The processing sequence for GUT is illustrated at figure 5-3. The basic steps are as follows:

1. For code 1. (Set trap vector) - move the supplied PC and Status into the vector in locations 34 & 36, clear three arguments from the stack and take a normal System Exit to free the Swap Buffer if necessary and restore user Registers.

2. For code 2. (Set RESTART address) - move the supplied address into the SVT for possible later use by the console RESTART command (see section 6.4.5), clear two arguments from the stack and take a normal System Exit.

3. For all GET codes - set a pointer to the SVT start from the content of its vector in location 40 and reduce the identifier code to an index. Check its range and if it represents an unassigned value, call a fatal error (F002).

4. Otherwise use the index to collect from a table of offsets the one relevant to the SVT entry required by the user. Replace the call code on the stack by the information to be returned and again take a normal System Exit.

5. Presently two GET functions require special treatment:

   a. Code 104 (TOD request) - two words must be returned, hence move the saved Registers and return parameters up the stack to provide necessary space.

   b. Code 106 (System-device name) - this information is in the DDL, hence use the SVT pointer to collect it from the first DDL word (see section 2.1.3).

Comments:

The General Utilities package is completely re-entrant and may be resident without restriction.

5.3 Conversion Utilities Package (CVT)

Commonly-used sub-routines for converting an appropriate string of ASCII characters or similar non-binary values into a binary quantity - or vice versa - are provided as a package within one Monitor module. Currently this package allows for single-precision binary conversions only. This automatically pre-determines the length of each string, as shown by the following list of the routines available:

a. Radix-50 pack & unpack - 3 bytes

b. Decimal ASCII digits to/from binary - 5 bytes

c. Octal ASCII digits to/from binary - 6 bytes

However on input this length is only deemed a maximum: the format of the string can cause the conversion to stop with the result at that point meaningful to the user. An output string nevertheless always produces the fixed number of bytes. In general, error conditions detected during a conversion are reported to the user in the processor condition codes: it is the user's responsibility to check these (using BCS,BVC etc) when the program is recalled, e.g.

C bit:invalid byte entered as input, or produced as output
V bit:input too large for single-word storage

As with the General Utilities package discussed in the previous section, the conversion routines are called by a single EMT code (42) with an identifier passed as one of the call arguments. Similarly there are two basic groups each with its own calling sequence and program recall state. These are described in sections 5.3.1 and 5.3.2.

The whole package is reentrant under all conditions and may be freely used residence-wise.

5.3.1 Conversions To Binary

Calling Sequence:

The three routines which convert from ASCII string to binary word, are all assigned even codes. They require only that the user supply the start address of the string since the length is fixed for the conversion as noted in the introduction. Hence their calling sequence is as follows:

```
MOV   #ADDR,-(SP)    ;PASS STRING START...
MOV   #CODE,-(SP)    ;...& IDENTIFIER
EMT   42             ;CALL CVT
```

Exit State:

When the program is recalled, the conversion replaces the code on top of the stack and the address is updated to point to the byte following the last one included in the conversion. The returned value is based upon all the bytes seen up to an invalid one or until overflow, subject to the maximum for the particular conversion. It is always correct up to the point of stoppage. For example, if the string '61,60,60,15' (i.e.100<CR>) is converted from decimal ASCII, the binary result is 000144 with the address returned on the stack being that for the <CR> byte. As mentioned earlier, however, the C bit is also set in this case to indicate the invalid input.

Processing:

Certain operations are common to all three conversions; therefore a co-routining technique is used. This is similar to that described for the .READ/.WRITE processor in section 3.2.2.2. The sequence followed by the mainstream is described below; the unique sections for each conversion are shown in sections 5.3.1.1 through 5.3.1.3. The whole module is further illustrated at figure 5-4.

1. Use a dispatch sequence common to all the conversion routines in order to:

   a. Clear the condition codes from the return status in anticipation of no errors

   b. Extract two call arguments and leave a pointer to their stack position

   c. Build the address of the first return point to the mainstream on top of the stack.

   d. Check the validity of the identifier code - if not assigned call fatal error (F034)

   e. Using the code as an index into a JMP-table, go to the unique routine to set a byte counter for the appropriate maximum string-length.

2. On return, clear the code word on the stack as the store for the accumulated result.

3. Collect a byte from the string, remove parity (bit 7) and reduce to numeric digit range. Return to the unique routine for further checking and perhaps adjustment.

4. Using a common sequence with variable return points, multiply the present result by a factor appropriate to each conversion (i.e. 50 (octal) for radix-packing, 10 for decimal and 4 for octal (1)). Add in the new byte.

5. If a check on the count shows further bytes remain to be processed, return to step 3. Otherwise update the string pointer on the stack to show the next byte to be processed. Remove the co-routine link from the stack and take a normal System Exit to free the Swap Buffer if necessary and recall the program.

5.3.1.1 Radix-50 Pack (code 0)

This form of conversion allows the restricted set of characters normally used for symbolic names to be stored in one word, three at a time, using the algorithm:

50(50A+B)+C

where 50 is octally based and A, B & C are the three characters coded in accordance with the following table:

0	=	SPACE
1-32	=	A-Z
33	=	$
34	=	.
35		(unused)
36-47	=	0-9

This conversion is stopped therefore after three bytes or upon recognition of a character not included in this set. In the latter case, however, the result must be left-justified, i.e. 'A' is equivalent to 'A '. Hence the

----
1. The octal conversion unique routine already has doubled to present result (see section 5.3.1.3).

unique routine called by the mainstream listed above performs the following operations:

1. On entry set the count for three bytes and recall the mainstream for the first byte.

2. Convert a valid byte in accordance with the table and return to the mainstream for multiplication of the current result by octal 50 and addition of the new value. Repeat if recalled with another byte.

3. Hold the string pointer at an invalid byte and set the C bit in the program Status saved on the stack to signal the error. Replace the byte with 0 and continue from step 2 to force the required left justification.

5.3.1.2 Decimal ASCII To Binary (code 2)

The conversion of decimal digits uses the standard algorithm in which a previously computed result is multiplied by decimal 10 and is then augmented by a new digit, always leaving a correct value if stopped at any time, e.g.:

```
Result #1 = A
Result #2 = 10A+B
Result #3 = 10(10A+B)+C
etc.
```

The maximum decimal value that can be stored in a single word is of course 65535. Hence the unique routine must ensure that this is not exceeded as follows:

1. Set the counter for five bytes and recall the mainstream for the first byte.

2. Verify that the byte is a valid decimal digit. If not, hold the string pointer at the incorrect byte and signal the error by setting the C bit in the program Status saved on the stack. Exit through step 5 in the mainstream.

3. Check if the currently accumulated value is 6553. If less, or if equal with a new digit not greater than 5, return to the mainstream for multiplication of the present result and addition of the new digit. Repeat from step 2 if recalled with another byte.

4. If the overflow condition is seen in step 3, hold the string pointer at the current byte, set the V

bit in the saved program Status and exit through step 5 in the mainstream.

5.3.1.3 Octal ASCII To Binary (code 4)

The conversion of octal values uses the same algorithm as in the previous section except that the multiplication factor is octal 10 rather than decimal. If the computation is stopped at any time, the last result is again valid. An overflow problem also exists; the unique routine in this case therefore checks that the result does not become greater than octal 177777 as follows:

1. Set the counter for six digits and recall the mainstream for the first byte.

2. Verify that the byte is a valid octal digit. If not, exit through step 2 of the decimal routine in the previous section to signal the error.

3. Check if the current result is greater than octal 17777. If so, exit through step 4 of the decimal routine to flag the overflow. Otherwise, double the result and return to the mainstream for its further multiplication by 4 and addition of the new digit. If recalled with another byte, repeat from step 2.

5.3.2 Conversions From Binary

Calling Sequence:

The corresponding routines to convert from binary to ASCII string use odd identifier codes. In their case not only must the user supply the start of a buffer for the storage of the converted bytes but also the binary word to be converted. The calling sequence therefore passes three arguments as follows:

```
MOV   #WORD,-(SP)    ;PASS DATA FOR CONVERSION...
MOV   #ADDR,-(SP)    ;...BUFFER START...
MOV   #CODE,-(SP)    ;...& IDENTIFIER
EMT   42             ;CALL CVT
```

Exit State:

On return to the program, the required string is stored as 7-bit ASCII characters in the buffer supplied. Its size is the fixed number of bytes associated with each conversion as noted in the introduction, with appropriate zero padding. Hence there is no need to update the buffer pointer returned to the user - he can compute its new position if required. Instead, the stack is completely cleared on recall.

Processing:

These conversions do not have the same compatibility as their counterparts discussed in section 5.3.1. They are therefore effected in separate subroutines discussed in sections 5.3.2.1 through 5.3.2.3. However they use the same common dispatch sequence as that previously described under step 1 in section 5.3.1 and share a common exit to remove the three call arguments from the stack and recall the program through the normal System Exit.

5.3.2.1 Radix-50 Unpack (code 1)

To reproduce the original characters packed into one word in accordance with the algorithm given in section 5.3.1.1, two operations are needed:

a. Successive division of the binary value by octal 50 in order to extract the three coded bytes.

b. Conversion of the bytes into their corresponding ASCII equivalents.

The algorithm used to effect the first of these needs explanation since it is not a generally standard one. It is based upon the fact that 3/16 of a quantity is almost equal to but is always less than 1/5 of that quantity. This follows from the basic algebraic tenet:

(x-1)(x+1)=x.x

This fraction therefore provides a reasonable first approximation for division by 5; re-iteration of the same process with the remainder (1/16 of the previous value) then enables a fairly rapid accumulation of the correct result. The number of iterations is certainly less than those required for the more normal division technique of divisor or dividend rotation with trial subtraction and, of course, this number reduces with the original quantity unlike in the oth-

er system. Moreover the core usage in both methods is roughly the same. Such usage is obviously greater than in the very basic successive subtraction technique but the time taken for this method does not stand comparison. Hence the reduction algorithm outlined is adopted because it is faster and currently the overall size of the CVT module is no problem.

The algorithm above can be applied in fact to any number which is one greater than a binary-power, e.g., 3,9,17, etc. and naturally becomes more appropriate the larger the number.

The second operation merely involves comparison of each byte with the corresponding ranges of the table given in section 5.3.1.1. to determine the ASCII equivalent. Two points however should be noted:

a. The maximum acceptable value for a valid result is that corresponding to 999, i.e. 174777 (octal). If a quantity greater than this is passed for conversion, the first byte with a coded value of 50 becomes (:) by the translation method used.

b. Any of the bytes can result in the unused code 35 which is translated into (/)

In either case the invalid character is returned to the calling program in order to complete the conversion. The error is however flagged in the C bit.

The bytes produced by the division are in the reverse order to that required by the final string. Thus the stack is used to effect the necessary switch as indicated in the following sequence for the conversion:

1. Set a stack marker at its present position (negative value)

2. Store the value on the stack and compute 3/16 (=(x-x/4)/4). In the same process, reduce the value on the stack to 1/16 (=(x-(x-x/4)-((x-x/4)4). Sum the quotients as this step is repeated until the remainder on the stack falls to 47 or below.

3. Step 2 produces division by 5, so divide the quotient obtained by octal 10 and if the second byte remains to be processed, repeat step 2 with the result. Otherwise check the result - now the first byte - and set the C bit in the program Status saved on the stack if greater than 47.

4. Check if the byte is the unused code 35. If so similarly signal the error by the return C bit. Convert the byte regardless of any error into its ASCII equivalent and store in the user-specified buffer. Collect the next byte from the stack and repeat this step until the marker set in step 1 is reached.

5. Take the common exit to clean-up the stack and re-call the program.

5.3.2.2 Binary to Decimal ASCII (code 3)

The conversion to decimal ASCII uses a standard simple algorithm which needs no explanation. Moreover no input value can produce erroneous results. Thus the sequence for the conversion is merely as follows:

1. Set a pointer to a table of 10-power values (10000-10)

2. Starting from octal 60 as the necessary ASCII base, count the number of times each 10-power value can be successfully subtracted from the input quantity and pass the result to the user's buffer. Repeat until 10 itself has been used.

3. The remainder is the digits byte - add 60 and pass to the user. Take the common exit to clear the stack and recall the program.

5.3.2.3 Binary to Octal ASCII (code 5)

Conversion to octal ASCII is effected by simple rotation of the input value to extract a requisite number of bits (1 for the first byte; 3 for the rest); to these octal 60 is then added as the necessary ASCII base for each of the six digits. Again no errors are possible. The particular algorithm uses C-bit detection of marker bits set into the rotated result-store in order to count the number of bits for each digit and also the number of digits as follows:

1. Set a marker bit and ASCII-base equivalent for one rotation in the junior byte of a result-store. In the high-byte set a marker for 6 rotations.

2. Rotate a bit from the input word into the junior byte of the result until the marker bit moves to the C-bit. Store the resulting byte in the user's

buffer.

3. Reset the junior byte as in step 1 for four rotations. Rotate the whole result word and repeat from step 2 until the high-byte marker reaches the C-bit. Then take the common exit to clean-up the stack and recall the program.

5.4 Command String Interpreter

In chapters 3 & 4, it was shown that the user has two normal methods of indicating the devices and if necessary the files he wishes to associate with the datasets providing I/O for his programs:

a. He can pre-set the necessary information into Link-blocks and File-blocks within each program source.

b. He can supply this information or can override that already in the program at load time by means of the console ASSIGN command.

Both methods presume that once the program has begun execution, it will need the same devices and files for each run while it remains loaded in memory. Neither method is really suitable, however, if the program needs the ability to use different I/O media for different runs, particularly if the new dataset specifications are at the discretion of the operator at the console keyboard. For the simple case, this might be managed by the program setting an appropriate RESTART address (see section 5.2), printing some operator signal and then waiting for the input of console ASSIGN and RESTART commands to start a new run. For the general case, however, this is clumsy and wasteful: each specification needs a separate ASSIGN and uses an individual 16-word free core buffer (see section 3.2.1.2)

The Monitor therefore contains a Command String Interpreter (CSI) as a means whereby dataset specifications, requested and entered as a string, can be translated and actually stored within the program Link-blocks and File-blocks in readiness for each run. Furthermore, most System Programs typically need this facility to change devices and files. Their use of CSI provides the added benefit that they all now present a standard interface to the operator at the keyboard.

This interface and the method by which user programs may also call for the services of CSI are both fully described in the Programmer's Handbook. This section thus contains only

sufficient outline for immediate reference. Briefly, the program performs the following operations:

1. Request command string input by printing '#' at the console typewriter; call .READ and .WAIT for the operator's reply.

2. Call a general Syntax Analyser, discussed in section 5.4.1, to check the overall accuracy of the input string and set up the decoding phase.

3. Call an Individual Dataset-specification Decoder, described in Section 5.4.2, to set up the Link-block and File-block for each dataset concerned.

5.4.1 Syntax Analyser (CSX)

The format specified for a command string to be processed by CSI is illustrated at figure 5-5. In particular, this shows that the specification for each dataset in the string can consist of a number of elements; however not all need be given, but those that are must comply with a set of rules, tabulated at figure 5-6. It is the principal function of the Syntax Analyser to ensure that these rules are not broken and to inform the program of infringments if detected. In addition, the Analyser cleans-up the string by removing extraneous characters (spaces, TABS, etc.) and fixes the position of pointers needed in the decoding operations which follow.

Calling Sequence:

In order to collect the command string, the program requests .READ from the dataset providing the input - usually one assigned to the console keyboard. As a standard requirement of this I/O function (see section 3.2.2.2), the program supplies a line buffer for the data, normally preceded immediately by a 3-word line-block of control information. For the use of CSI, the line buffer with its header for the input of a command string is further preceded by a 7-word work-space as shown in figure 5-7. (CSI assumes that this format is followed exactly, and in particular does not expect the Dump mode described in section 3.2.2.2.) Hence it requires only that the program pass the start address of the workspace (CMDBUF) as part of its call in order to access both the workspace and the data in the line. Moreover at this stage is is not concerned with the program dataset linkages. Thus the calling sequence is simply:

```
        MOV   #CMDBUF    ;PASS W/SPACE ADDRESS
        EMT   56         ;CALL CSX
```

General Description:

CSX initially prepares the workspace for later use as depicted in figure 5-8. The significance of the entries is as follows:

1. Input/Output Code - enables the program to identify which side of the string is to be used at the next request for the decoding of a dataset specification - see next section.

2. Current Input Address - shows the start address of the next input dataset specification to be processed at any time. It is initially set at the address of the byte following '<' if detected, or 0. It is updated as each specification is decoded.

3. Current Output Address - performs the same function as 2 for output. Its initial setting is the string start address.

4. Current Input Default Device - provides a device name when none is entered as part of a specification. The start assumption is the system-device; any new device entry overrides the previous default (on the input side only - defaults do not cross the '<' boundary)

5. Current Output Default Device - is the same as 4 for output.

6. Current Input Default Unit - gives the unit number if the device can control several. Its initial content is 0 and is updated as in 4.

7. Current Output Default Unit - equates to 6 for output.

Having collapsed the string to show only recognizable characters, CSX performs its necessary checks on the basis of a set of states corresponding to the elements being scanned. These are as follows (with their initiating characters):

0 = Specification Start (,)
2 = Device (letter)
4 = Filename (letter)
6 = Extension (.)
8 = UIC ([)
10 = Switch (/)
12 = Input Side Start (<)
14 = String End (LF,FF,VT)

Whenever a punctuation character in the string denotes transition between states, a table corresponding to the prior state is examined. This table gives a list of current states which cannot be accepted in accordance with the rules previously illustrated in figure 5-6. Upon detection of an invalid state or of a character which is illegal generally or within a state, CSX recalls the program with the address of the byte terminating the scan on top of the stack. Otherwise 0 is returned. It is then the responsibility of the program to check the indicator (and remove it from the stack) and to take action accordingly.

Detailed Processing:

The processing operations of CSX are illustrated at figure 5-9. The sequence of these operations is basically as follows:

1. Collect the start of the workspace CMDBUF and initialize it as noted above - using the DDL entry in the SVT to determine the system device (see section 2.1.1 and 2.1.3). Set pointers and switches

2. Scan the string and remove spaces, TAB's and rubouts until a valid line terminator is seen.

3. Ignore ',' and remain at state 0 until some other character is detected. Assume state 4 and if the character is a letter, scan the string for the first non-alphanumeric character, counting the number of leading letters and setting a flag if digits occur between letters. If the new character is not ':' go to step 5 to check the acceptability of Filename at this time. Otherwise set state 2, provided that up to three letters perhaps followed only by digits have been seen. Again go to step 5 for similar check for Device.

4. If the first character not ',' in step 3 is '*', accept it in lieu of Filename, as long as it is not preceded or followed by letters or digits. Reset the state and action accordingly if one of the following characters is detected. Go to step 5 on completion for the relevant check.

   a. '.' (state 6) - move past letters and digits (or a single '*').

   b. '[' (state 8) - scan beyond the only valid sequence '[(digits),(digits)]' (or '*' in lieu of (digits) in either case).

   c. '/' (state 10) - ignore letters or digits followed by repetitive sequences of ': (letters, digits, '.' or '$')'.

   d. '<' (state 12) - accept this character only if not previously seen and reset 'Current Input Address' in CMDBUF.

   e. CR - ignore as long as it is followed by a terminator

   f. LF, FF, VT - accept

   g. All others (including digits) - reject

5. In the relevant state table check that the state just terminated can follow the one previously seen. Remember the new state and if not 14, return to step 3. Otherwise take a normal System Exit (see section 2.3.4) with 0 on top of the stack.

6. For invalid characters or states, replace the 0 on the stack with the latest value of the pointer used to access the string before taking the System Exit.

Comments:

CSX is fully reentrant at all times and can be used freely whether resident or not. It may also be recalled to re-initialise the workspace for the string if required.

5.4.2 Specification Decoder (CSM)

The purpose of the Decoder routine is to set up the Link-block and File-block for a dataset and transfer direc-

tion specified by the program, by extracting the next specification from the relevant side of a command string already checked by the Syntax Analyser discussed in the previous section and by then translating its content into the format required. The program can thereafter perform its I/O operations upon that dataset in the normal way described in chapters 3 & 4, as qualified by the requirements of any switches entered with the specification. The program itself, however is responsible for determining the validity of those switches for its purpose.

Calling Sequence:

The Decoder expects the workspace CMDBUF associated with the line buffer containing the string to be already initialised correctly and performs no syntax checks. A call to the Syntax Analyser must therefore precede the calls to the decoder. If the Analyzer is recalled, the workspace is reinitialised and decoding can thus be started afresh if required (affecting both sides however).

As noted in the previous section, the first word of CMDBUF is reserved for the program to indicate which specification is to be used next. The program sets this word to 0 to signify the input side of the string; 2 the output side. The appropriate Link-block and File-block must also be correctly established. The format of the latter is exactly as described in section 4.3.2. The Link-block, on the other hand, must include space for the device name as shown in section 3.1.2.1 and may be further extended to allow a buffer in which the Decoder can return Switch information. The program therefore allows sufficient room for all the information expected and indicates the size in the Link-block byte showing "Number of words to follow". Finally, the decoder needs the three relevant addresses. These are supplied by the program in a table as shown at figure 5-10 and the address of this table is passed as the only argument in the calling sequence, e.g.

```
        MOV #CSIBLK,-(SP)       ;PASS TABLE ADDRESS
        EMT 57                  ;CALL CSM
```

Exit State:

When the program is recalled, the device name and unit number from the specification or from the default stored in CMDBUF are set into the Link-block, unless no specification at all was entered for the dataset. In this case, the device name in the Link-block is set to 0. Data for the File-block is correctly stored or is replaced by 0 in the

elements not supplied, '*' entries are indicated by octal 52 in Filename or Extension or by octal 377 in either byte of the UIC. Switch data is listed in the Link-block extension as depicted at figure 5-11. In addition, a status code is returned on the stack to notify the program as follows (bit=1):

Bit 0: No more specifications to come on this side
Bit 1: Too much switch data for the space allowed

It is the program's responsibility to check this (and clear it from the stack) and act accordingly. Internally the relevant entries in the CMDBUF - current address and, if any entered, current device and unit - are updated in readiness for the next call for decoding on the same side.

Processing:

The processing operations to accomplish the Decoder function are illustrated at figure 5-12 and basically follow the outline given below:

1. Collect the call parameter and use it to clear device name and unit in the Link-block, the File-name block and if Switch space is provided, a switch stop in its first word. Also collect the relevant string pointer from CMDBUF for the side requested - if 0, exit as for '<' or CR below.

2. Check the next string byte and action accordingly as follows:

   a. ',' - go to exit at step 7 with 0 flag to show more to come.

   b. '<' or CR - go to exit at step 7 with 1 flag to show no more.

   c. '.' - pack in radix-50 up to the first three alphanumeric characters and store as Extension in the File-block. Discard all subsequent alphanumerics. Repeat the check. (If '*', store octal 52 instead).

   d. '[' - convert octal ASCII digits up to the dividing ',' and then those following into two binary bytes and store in the UIC slot in the File-block. Repeat the check.

   e. Letter - scan for the first non-alphanumeric. It this is ':', pack leading letters into radix-50 and convert the digits, if any, from

octal ASCII into a binary byte. Store both in the Link-block as Device name and unit. Otherwise perform single-precision radix-50 packing of up to the first six alphanumeric characters and store as Filename in the File-block. Ignore further subsequent alphanumerics. Repeat the check.

f. Digit - treat as start of File-name as shown in (e) above.

g. '*' - store octal 52 in Filename and repeat the check.

3. If at the check in step 2, none of the listed characters is seen, the next element can only be a Switch, so store up to the first two characters following as bytes on top of the stack and discard all subsequent alphanumerics.

4. If the next byte is ':', stack the address of the one following it. Discard all subsequent alphanumerics, '$' or '.'. Repeat this step for any further values included in the Switch, counting the number of addresses stacked.

5. Otherwise check if sufficient room exists in the Link-block Switch area for the new data. If so, move the items stacked during steps 3 & 4 into the Link-block preceded by a count of the words moved and followed by 0 (if still room) as a switch-stop. Repeat from step 2.

6. If the Link-block area cannot take the whole of the data for the particular switch, remove such data from the stack and set a switch overflow indicator. Repeat from step 2 to move to the specification end.

7. Store the marker results from step 2 (a & d) and step 6 on the stack for return to the user. Save the string pointer, now showing the start of the next specification, in the relevant "Current Address" slot in CMDBUF. If any elements have been processed at all, save any new default device name and unit in CMDBUF; otherwise move the present default values into the Link-block. Take a normal System Exit.

Comments:

Like its counterpart in the CSI, this module is fully re-entrant and its residence is without restriction. It should be noted of course that CSM contains its own routines for radix-50 packing and octal ASCII to binary conversion routines: it cannot expect to be able to use the Conversion Utilities described in section 5.3 since these may also need the Swap Buffer.

5.6 Program Exit (XIT)

The Monitor offers two methods by which a program can be unloaded from memory when it has served its purpose:

a. The console keyboard KILL command

b. The program service request .EXIT (EMT 60)

Both methods in fact effectively result in a call to the same Monitor module discussed in this section. Its purpose is merely to act as a loader for a transient Monitor section which occupies memory whenever there is no loaded user program - already mentioned in section 5.1 and to be described in detail in section 6.5.

Calling Sequence:

The calling sequence for the XIT module requires no parameters; the EMT 60 instruction alone suffices.

General Description:

Using its own internal DDB and an area of once-only code in itself as a buffer, the routine reads the first two entries from the directory of the Monitor Library on the system-device. As shown in section 8.2, the first entry relates to the permanently resident Monitor itself and is used for re-loading. The second entry records the system-device location of the transient Monitor section. The form of the entry was shown earlier in figure 1***

The XIT module resets its internal DDB from the data in this entry in order to read in the transient section. The memory area this uses is relocatable to the upper end of availabe core as shown by CSA in the SVT (see section 2.1.1) provided that it is correctly linked to a top of 17400 (see section 8.1). XIT assumes this and computes both the load point and

later the start point on this basis.

Before requesting the read, however, the routine waits for any current I/O to complete since there is no restriction that the user must close and release all datasets before the .XIT. The wait is accomplished by a repeated search along the DDB chain from its origin at DCO in the SVT (see section 3.1.2.3) until no busy states are seen. The read is then called. Any failure in this or in the earlier library directory read results in the ultimate error - the System Halt.

Exit State:

On satisfactory completion, XIT repositions the stack immediately below the loaded transient Monitor section and transfers control to this section with the Swap Buffer released for its use. The transient section is then responsible for cleaning-up after the program, in particular ensuring that any datasets still open are correctly closed with all buffers released and that the normal Monitor states are restored. To assist in these operations, XIT transmits the following valid register contents:

R0 = Address of processor Status Register (or -2)
R5 = Address of DCO in the SVT (see Section 2.1.1)

Processing:

The foregoing discussion gives sufficient outline of the operations performed by XIT. Figure 5-13 shows these in more detail.

Comments:

By its nature, this module need not be re-entrant. It is not, of course, because of the use of internal workspace. In theory, it can be made resident, though since it is used relatively seldomly, there seems little point in its being made so.

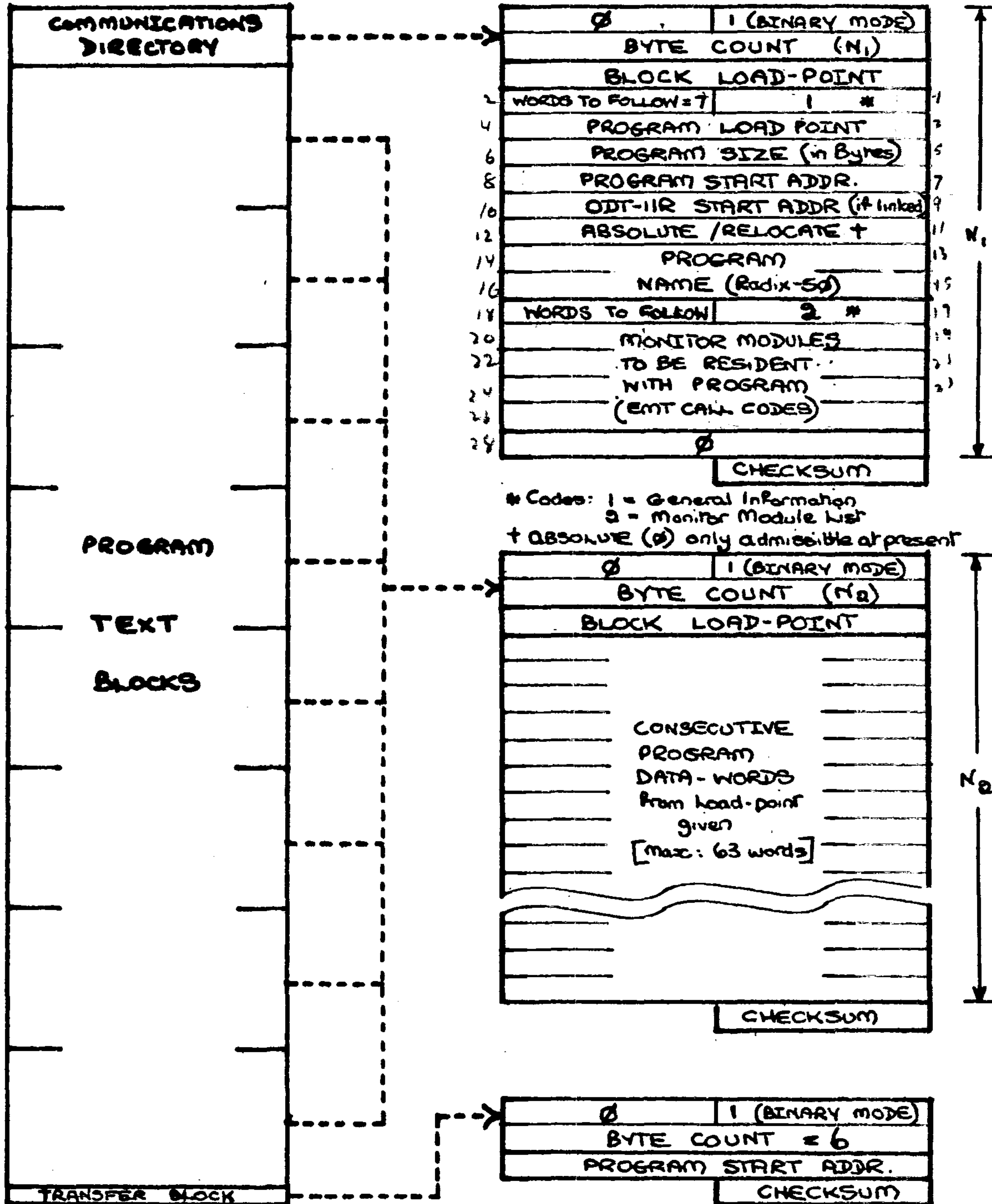


Figure 5-1: PROGRAM LOAD IMAGE as produced by Linker (LINK-11)

PROGRAM LOAD OPERATIONS
(after Initial Action in TMON [RUN/GET])

ENTRY VIA EMT 61 HANDLER

LDR. THIS ROUTINE IN RESMON? Y / N

CLAIM BUFFER & TRANSFER SELF INTO IT

PROCEED FROM NEW AREA FREE UP SWAP BUFFER

LD.LDR GET BACK REGISTERS PASSED ON CALL

ATTACH DDB PASSED BY CALL TO LINK-BLOCK

GET PROG. LOAD LINK & TRAN-BLOCK ADDRESSES

SET UP TRAN-BLOCK FROM DATA PASSED BY CALL

LD.TSK DOUBLE MOVE STACK TO BELOW LOAD POINT VIA AREA ABOVE TOB

SAVE FILE TYPE. CLEAR PROGRAM AREA

LD.LBB IF DATA BUFFER EMPTY REFILL.

LD.OK1 NEXT BYTE = Ø? Y / N

=1? Y / N

LD.FER F22 ERR

REFILL BUFFER IF EMPTY

LD.OK0 NEXT BYTE = Ø? Y / N

REFILL BUFFER IF EMPTY

LD.OK3 STORE NEXT BYTE AS LOW PART OF COUNT START CHECKSUM

REFILL BUFFER IF EMPTY

LD.OK4 COMPLETE COUNT WORD ADD BYTE TO CHECK

REFILL BUFFER IF EMPTY

LD.OK5 USE NEXT BYTE AS LOW PART OF LOAD POINT. CHECKSUM

REFILL BUFFER IF EMPTY

LD.OK6 COMPLETE WORD FOR LOAD POINT. CHECKSUM

REMOVE HEADER BYTES FROM COUNT

ANY COUNT LEFT? N

LD.UND RELEASE LOAD DATASET

CALL LOADER PT. II

LD.MYL REFILL BUFFER IF EMPTY

LD.OK7 MOVE IN NEXT BYTE STARTING AT INPUT LOAD PT.

WHOLE BLOCK READ? Y / N

REFILL BUFFER IF EMPTY

LD.OK8 CHECK-SUM O.K.? Y / N

LD.RER F21 ERR

LD.RNB FILL BUFFER SUB-ROUTINE

GET TRAN-BLOCK ADDRESS

E.O.D SEEN LAST TIME? Y / N

LINKED FILE? Y / N

GET FILE LINK FROM BUFFER

Ø? Y / N

LD.CNF INCREMENT BLOCK #

SET POINTER AT BUFFER START

LINK NEGATIVE? Y / N

LD.CDT DECTAPE? Y / N

SET REVERSE TRAVEL FLAG. NEGATE BLOCK #

LD.DTF CLEAR REVERSE TRAVEL FLAG IN FUNCTION

LD.GTI RESET POINTER AT BUFF START + 2 (OMIT LINK)

LD.ATI CALL .TRAN & .WAIT

ERROR IN TRANSFER? Y / N

F21 ERR

ADJUST BUFFER END POINTER IF SHORT BUFF. FILL

EXIT TO CALLER

Phase I (PROGRAM LOAD)

Figure 5-2(a)

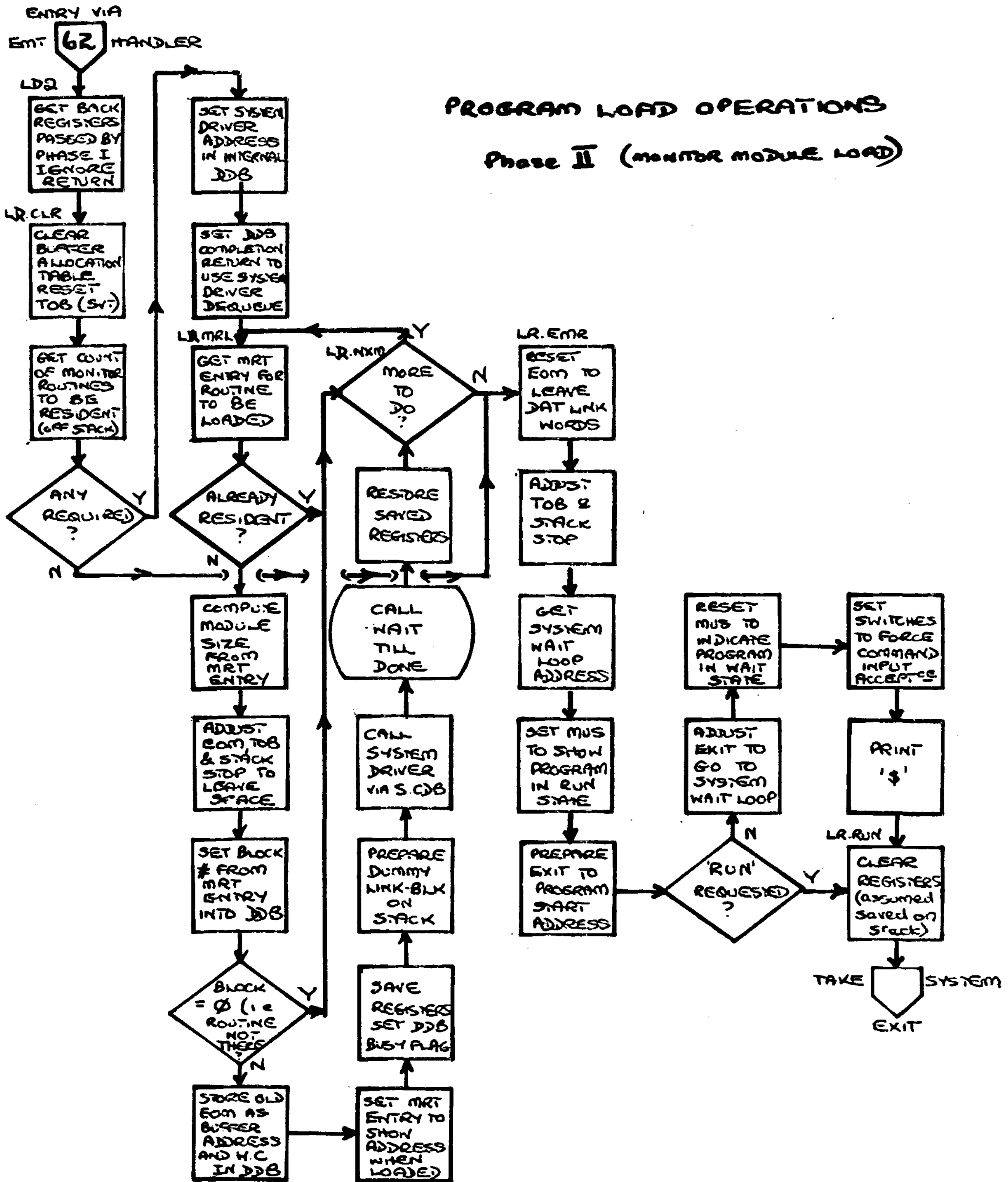


Figure 5-2 (b)

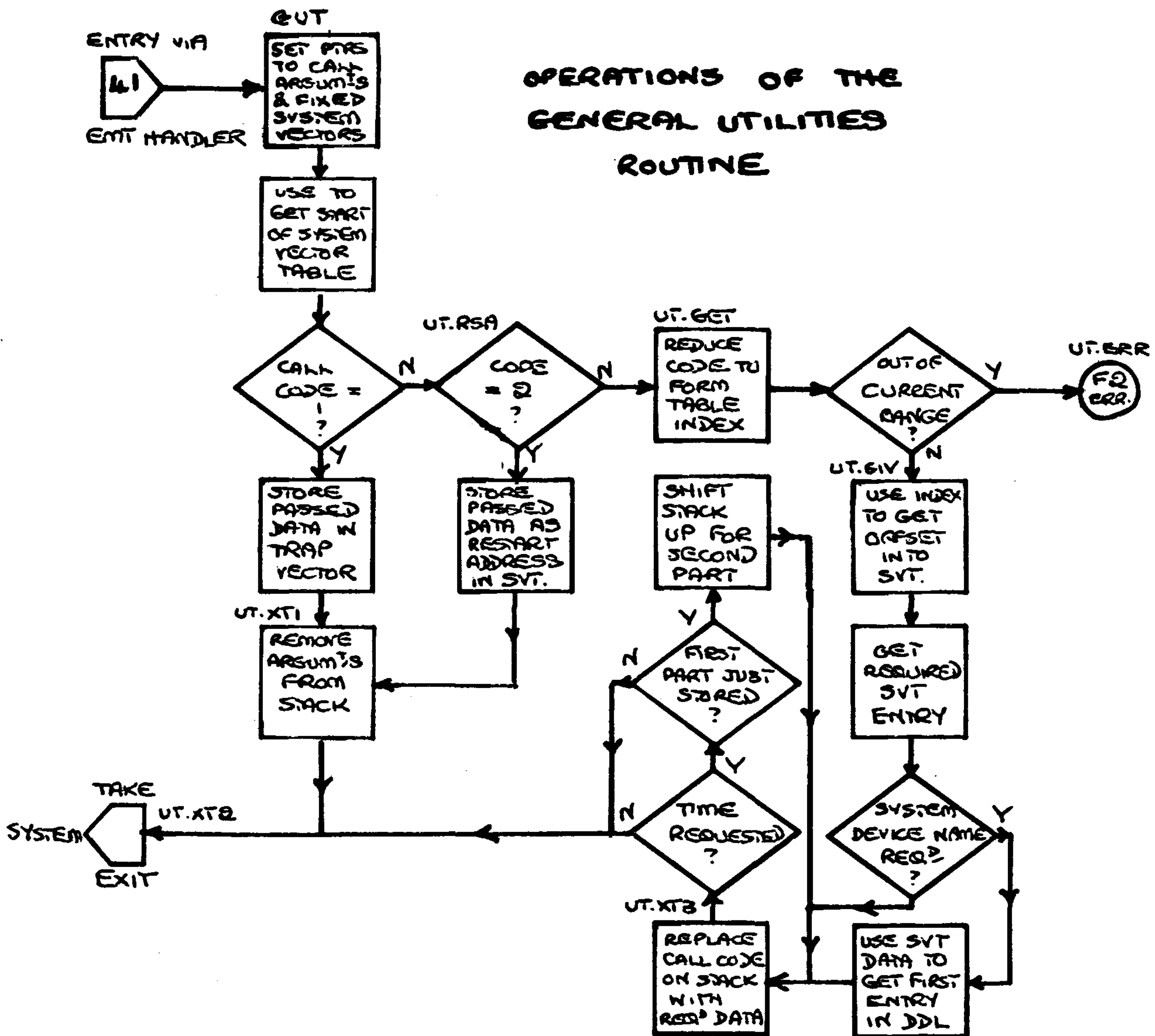


Figure 5-3

CONVERSION ROUTINE OPERATIONS

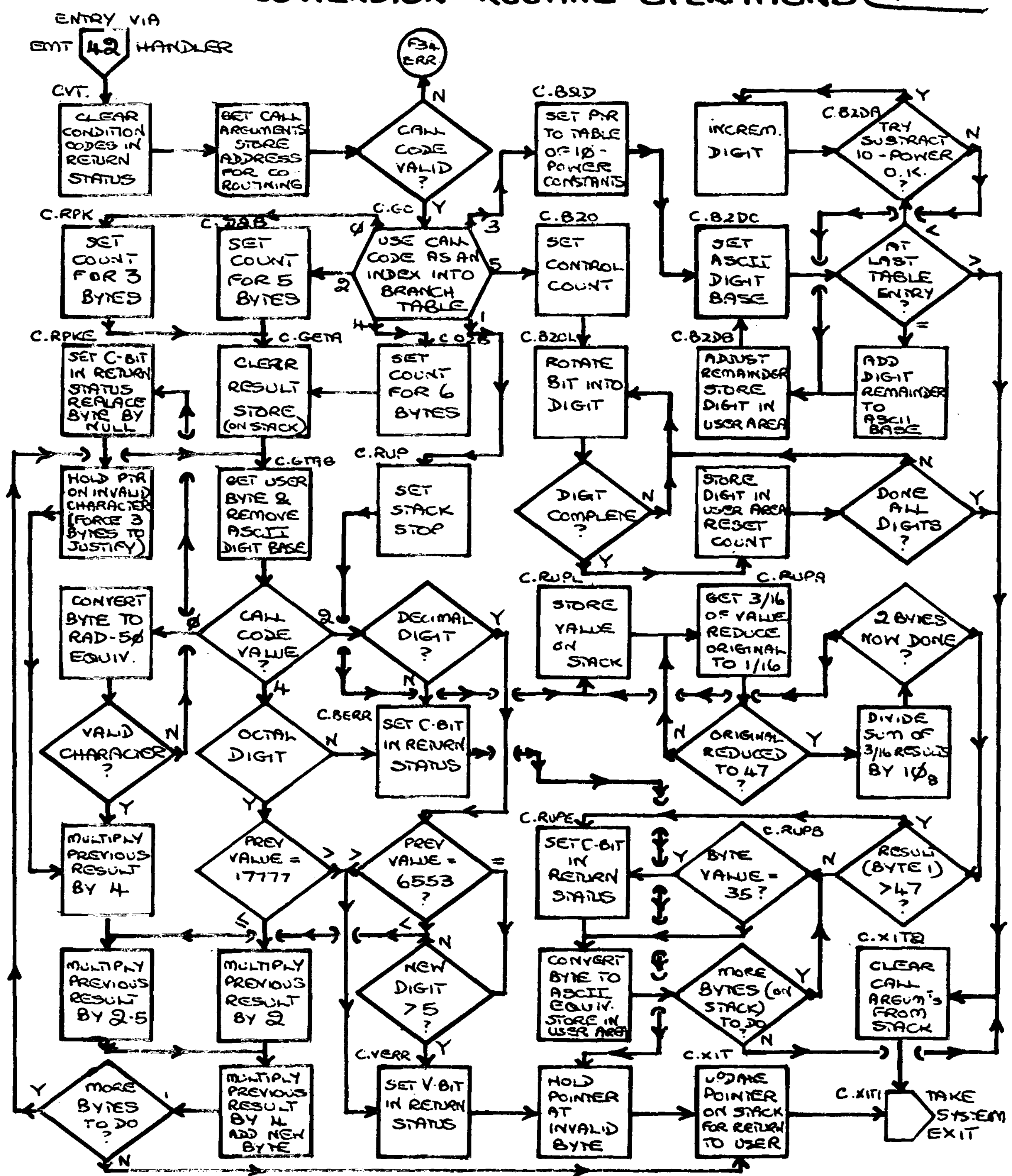


Figure 5-4

COMMAND STRING FORMAT :-

Output Specification String< Input Specification String <CR>

where either string is a series of one or more dataset specifications separated by commas and of the general form :-

DEVICE NAME: FILENAME. EXTENSION [UIC] /SWITCH : VALUE1:... VALUEn

Figure 5-5. Command String Input

Item Which Last Appeared	Item Immediately Following							
	,	DEV:	FILNAM	.EXT	UIC	/SWITCH	<	Terminator
blank[1]	*	*	*	E	*	*	*	*
,	*	*	*	E	*	*	*	*
DEV:	*	E	*	E	*	*	*	*
FILNAM	*	E	E	*	*	*	*	*
.EXT	*	E	E	E	*	*	*	*
UIC	*	E	E	E	E	*	*	*
/SWITCH	*	E	E	E	E	*	*	*
<	*	*	*	E	*	*	E	E

Legend: E indicates error. * indicates legal.

Note: [1] The next item encountered is the first item in the command string.

Figure 5-6. Command String Syntax Rules.

Figure 5-7: COMMAND STRING INPUT BUFFER

CMDBUF:	INPUT / OUTPUT CODE	
	CURRENT INPUT ADDRESS	
	CURRENT OUTPUT ADDRESS	
	CURRENT INPUT DEFAULT DEVICE	
	CURRENT OUTPUT DEFAULT DEVICE	
	CURRENT INPUT DEFAULT UNIT	
	CURRENT OUTPUT DEFAULT UNIT	
LINE:	MAXIMUM LINE SIZE	
	STATUS	MODE

Figure 5-8: COMMAND BUFFER FOR CSI

MAINSTREAM:

ENTRY VIA 56 EMT HANDLER

CSX
SET PTR TO CMD BUFFER PROVIDED CLEAR OK SWITCH

CLEAR INPUT ADDR. IN CMDBUF OUTPUT = STRG STRT

SET SYS. DEVICE (UNIT Ø) AS DEFAULT (INPUT & OUTPUT)

CLEAR INPUT SWITCH. SET OLD STATE = Ø

CSINØØ
COLLAPSE STRG. TO REMOVE UNWANTED FORMAT CHARS.

CSTXØØ
SET NEW STATE = Ø

NEXT CHARACTER = ';' ? (Y / N)

SET NEW STATE = 4

NEXT CHARACTER ? (LETTER / SPECIAL / DIGIT)

A

CSCINØØ
MOVE POINTER TO NEXT CHARACTER

H

CSLTIØ
NEXT CHARAC. ? (LETTER / SPECIAL / DIGIT)

CSLTIB
SET CONSEC. LETTER FLAG

CSLTØØ
CLEAR CONSEC. LETTER FLAGS

CSLTØ5
COUNT CONSEC LETTERS

CONSEC LETTER FLAG SET ? (Y / N)

SET NON-CONSEC LETTER FLAG

CSLJ2Ø
NEXT CHARACTER = ':' ? (Y / N)

CSDVØØ
>3 CONSEC LETTERS ? (Y / N)

ANY NON-CONSEC LETTERS ? (Y / N)

SET NEW STATE = 2

A

CSTXØ3
NEXT CHARACTER = '#' ? (Y / N)

DO ASTERISK CHECK

CSTXØ5
SET NEW STATE = 6

NEXT CHARACTER = ',' ? (Y / N) → B

SET NEW STATE = 8

NEXT CHARACTER = '[' ? (Y / N) → C

SET NEW STATE = 1Ø

NEXT CHARACTER = '/' ? (Y / N) → D

SET NEW STATE = 12

NEXT CHARACTER = '<' ? (Y / N) → E

SET NEW STATE = 14

NEXT CHARACTER = 2 ? (Y / N)

MOVE POINTER TO NEXT CHARACTER

CSTX1Ø
NEXT CHARACTER = TERMIN. ? (Y / N)

CSSTCK
SET STATE TABLE AS APPROP TO OLD STATE

G

CSCKØØ
NEW STATE AFTER OLD VALID ? (Y / N)

MOVE NEW TO OLD STATE

ERROR
SET OK SWITCH WITH PTR TO INVALID CHAR FOR USER

OLD STATE = 14? (Y / N)

EXIT
TAKE SYSTEM EXIT

Figure 5-9 (a)

COMMAND STRING SYNTAX ANALYSIS

SUBSIDIARY ROUTINES:

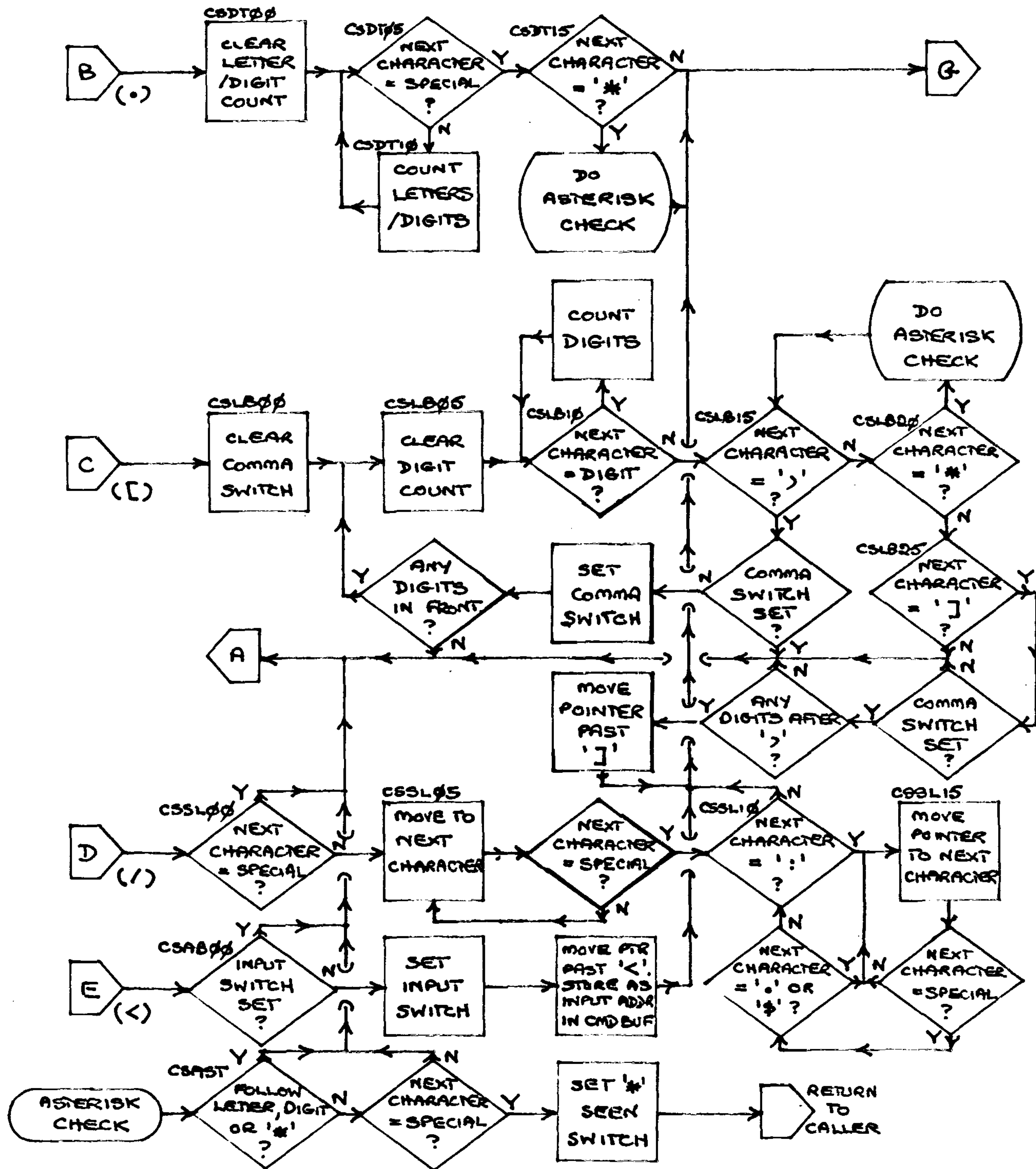


Figure 5-9 (b)

CSIBLK:

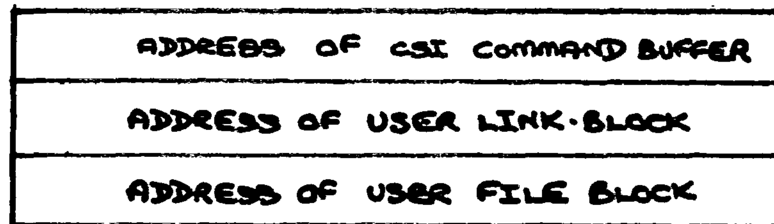
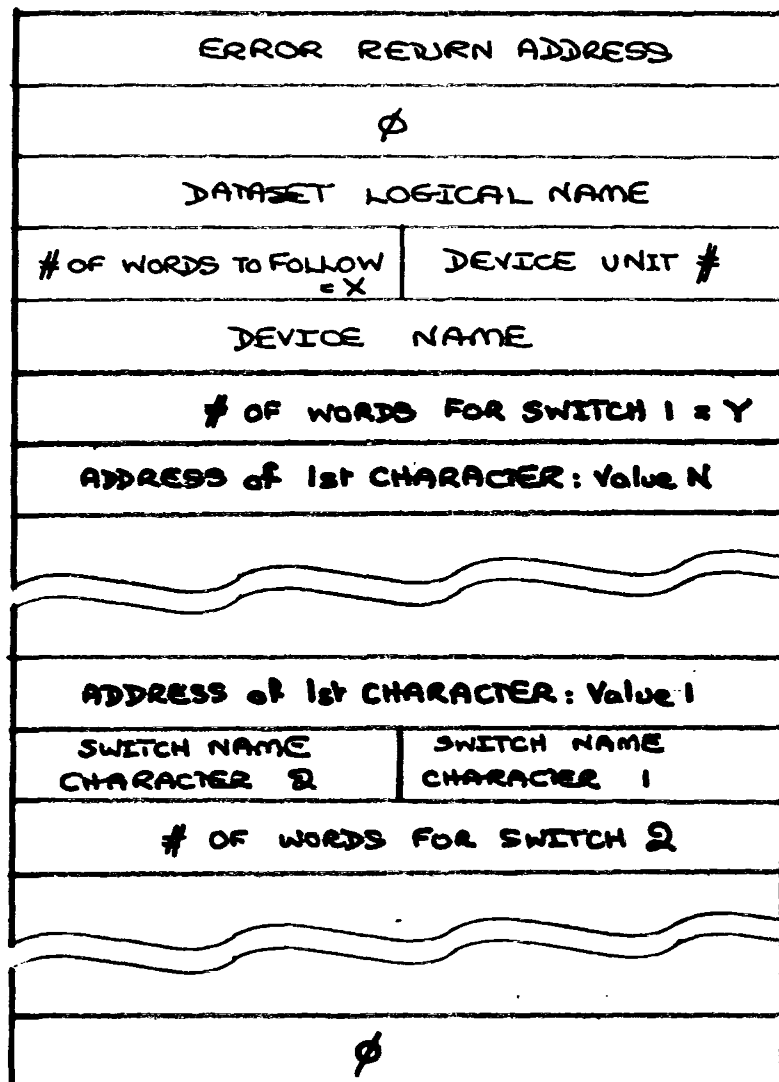


Figure 5-1Ø: CSI COMMAND BLOCK

LNKBLK:



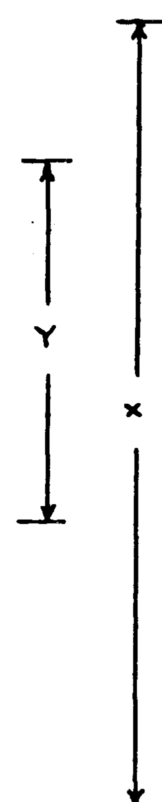


Figure 5-11: SWITCH DATA IN THE USER LINK-BLOCK

COMMAND STRING DECODING.

ENTRY VIA EMT 57 HANDLER

CSM

CLEAR ITEM SWITCH & RETURN STATE

GET CALL ARGUMENTS

SET PTR TO STRING SIDE AS GIVEN BY CALL CODE

POINTER = Ø ?

SET PTR TO SWITCH AREA IN LINK-BLOCK

CLEAR DEVICE & UNIT FROM LINK-BLK

ANY SWITCH AREA GIVEN

CLEAR FIRST ENTRY AS A STOPPER

CSMIØØ

CLEAR USER FILE-BLOCK

CSMØØØ

NEXT CHARACTER = ','

CMCMØØ

MOVE POINTER TO NEXT CHARACTER

CMCONT

MOVE STATE TO STACK FOR USER

STORE POINTER IN CMDBUF FOR NEXT ACCESS

ITEM SWITCH SET ?

DEVICE IN ITEM

STORE NEW DEVICE & UNIT AS DEFAULT IN CMDBUF

CSMXØ5

GIVE USER DEFAULT DEVICE & UNIT

TAKE SYSTEM

CSMX10

EXIT

NEXT CHARACTER = '<' OR '=' ?

CMLNØØ CMABØØ

SET STATE TO SHOW NO MORE THIS SIDE

SET ITEM SWITCH

NEXT CHARACTER = '.' ?

CMDTØØ

MOVE POINTER TO NEXT CHARACTER

PACK 3 LETTERS/DIGITS (OR ASTERISK) INTO RAD-50

STORE AS FILE EXTENSION

CMDTØ5

DISCARD FURTHER LETTERS /DIGITS

CMSL27

IF MORE SPACE IN AREA, SET NEXT = Ø AS STOPPER

NEXT CHARACTER = '['

CMLBØØ

MOVE POINTER TO NEXT CHARACTER

CMLBØ5

CONVERT DIGITS FROM ASCII TO OCTAL

STORE IN APPROP SECTION OF UIC

CMLB10

NEXT CHARACTER = ']'

ANY DIGITS AFTER ?

CONVERT DIGITS FROM ASCII TO OCTAL

CMSL20

ROOM IN SWITCH AREA FOR ENTRY ?

CMSL25

MOVE ENTRY INTO SWITCH AREA

DIGIT

NEXT CHARACTER ?

LETTER

CMLTØØ

SEARCH FOR NEXT SPECIAL CHARACTER

FOUND ':' ?

CMDVØØ

SEARCH FOR DIGIT. IF ONE FOUND, MASK IT

CMDV10

PACK LETTERS INTO RADIX-50 FORMAT

STORE AS DEVICE

STORE AS UNIT

CMSL15

NEXT CHARACTER = '.' OR '$' ?

CMSL3Ø

REMOVE ENTRY FROM THE STACK & SET STATE TO SW. ERR

NEXT CHARACTER = '*'

CMFLØ5

RADIX-50 PACK 6 LETTERS/DIGITS OR '*'

STORE AS FILE-NAME

CMSLØØ

MOVE PTR TO NEXT CHARACTER SET COUNT STACK 1ST CHARACTER

NEXT CHARACTER = SPECIAL ?

COMBINE NEXT CHARACTER WITH ONE STACKED

CMSLØ5

NEXT CHARACTER = SPECIAL ?

CMSL10

NEXT CHARACTER = '.' ?

STACK CURRENT POINTER VALUE & COUNT

Figure 5-12

EXIT OPERATIONS

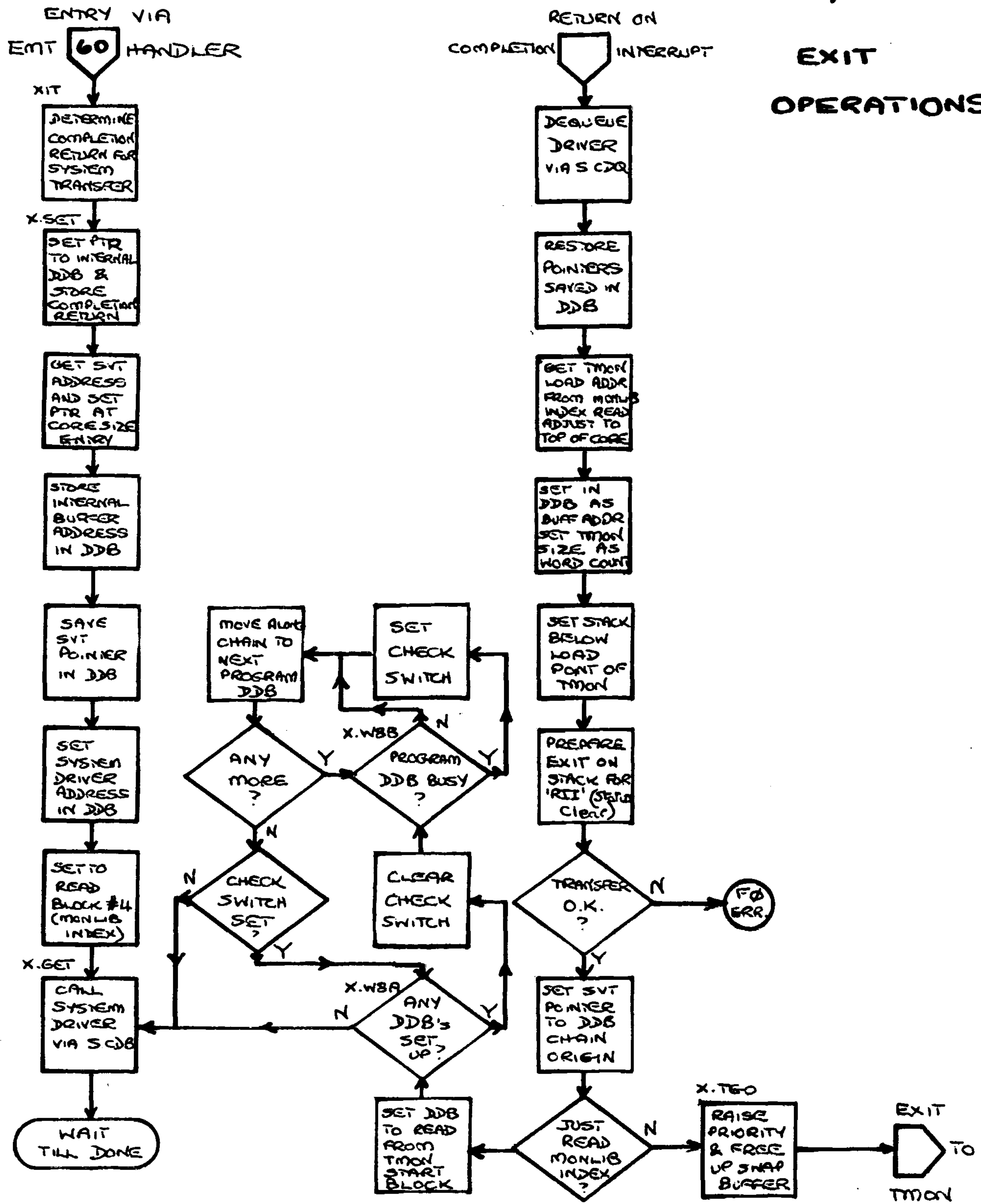


Figure 5-13

CHAPTER 6

K E Y B O A R D   S E R V I C E S

As its second major function, the DOS Monitor exists to enable the user to control the operations of the system through the console keyboard. Thus Chapter 3 of the Programmers Handbook describes various commands which can be entered whenever the Monitor is in a listening state, because either no program is under execution or such state has been forced by a prior entry of CTRL/C. The commands permit the loading, starting, stopping and unloading of programs, the allocation of resources and the exchange of information between the operator and the system. The object of this chapter is to illustrate the processes by which these commands become effective.

Section 6.1 reviews the general philosophy of the Command Language and Section 6.2 describes the organization of the modules which go to form it. It will be shown that most commands pass initially through a common Interpreter which is made up of three modules discussed in Section 6.3; these commands are then individually processed by routines considered in some detail in Section 6.4. Section 6.5 covers the remaining commands which are handled by a special transient Monitor section which occupies core whenever no program is actually loaded.

The external operator interface is already fully explained in the Handbook. In this chapter therefore, this information appears again only to the extent that it is necessary for the understanding of the internal workings. In particular the conventions for command formats are listed at the start of Chapter 3. They are used here without further reference. It is also assumed that the reader is familiar with the special significance afforded certain keyboard characters within the language.

6.1 General Philosophy

The purpose of this section is to summarize the structure and usage of the Keyboard Command Language and to comment upon the underlying principles involved.

As noted in the introduction, a command can be input whenever the Monitor is in a position to accept such input. This is indicated to the operator by the output of one of the following characters at the console typewriter:

1. $ - this signifies that the Monitor already has overall control of the system and requires operator instructions for its next action. This implies one of the following situations:

   a. No program is currently in memory.

   b. A loaded program has not yet been started.

   c. A running program has been suspended, either because of some operator request or following the detection of some error condition which needs operator intervention. (see Chapter 7)

2. . - this notifies the operator of the acceptance of his entry of CTRL/C as the means of forcing the Monitor to listen, even though the system is under program control.

The general format for all commands is:

COMMAND [Argument String]CR

where "COMMAND" is a single word identifier of which only the first two letters need be entered and "Argument String", if any, may consist of one or more arguments separated by spaces (or commas) depending upon the nature of the particular command. Apart from the obvious advantage that this format is standard, it was also deemed more advisable than one using special CTRL characters for some commands, for the following reasons:

a. The unique identification of a command on the basis of two letters offers a wider range of variations and hence is more likely to be mnemonic.

b. It is envisaged that devices under development as console terminals for PDP-11 may not all have the full CTRL character facility.

c. It is readily possible to input a single character incorrectly with perhaps disastrous results (despite the need to depress the CTRL key, since this can be done mistakenly for SHIFT).

On the other hand, of course, the command being longer takes more time to key in. It is unrealistic that a running program should be suspended during the whole of this period as well as that needed to effect the desired result. Therefore no command is examined until the terminal carraige return (CR) is detected. Prior to this, the operator can change his mind as often as is necessary or can correct his mistakes, and in the meantime, any program can continue its normal execution. However, once the command has been fully entered, no further input is accepted until it has been completely processed.

Finally it is the intention that any command, which does not itself imply some restriction upon its usage (such as KILL a program when there is none in core to be killed), should as far as possible be available to the user at all times regardless of the general status of the system. This raises two problems, the solution to which will be discussed in the next section:

1. The Monitor must have access to a driver for the console terminal even though the program in core does not expect I/O from this as a device.

2. The processing of a command cannot be carried out in the normal Swap Buffer when this might be already in use by some other Monitor routine servicing a running program request.

Moreover, unless the command itself changes the status, any running program must continue after its execution as if nothing had happened.

6.2 Organization

Before the command processors themselves are examined, the overall internal structure by which they are controlled requires explanation. As noted in the last section, the Monitor must, in effect, be able to accept a command at any time and the operation requested must be carried out tranparently to the program currently occupying memory. Sections 6.2.1 through 6.2.4 show how this is accomplished under the four main headings:

a. Acceptance

b. Decoding

c. Processing

c. Clean-up

Section 6.2.5 then discusses conventions used throughout the normal processing of the language.

It should be noted, however, that the processes described in this Section do not apply to commands which by their nature do not have the problem of a running program to consider, because none can be loaded when they are used, i.e., RUN, GET, and FINISH. These commands are handled by the transient Monitor section and as mentioned in the introduction are described in Section 6.5.

6.2.1 Command Acceptance

Basically, to the Monitor expecting a command, the console terminal is an input device with a purely subsidiary output facility as a means of echoing the characters entered. To a running program, however, the console terminal is a device with mutually independent input and output facilities, but with the echo requirement an added complication. Furthermore, if the particular console device is the currently standard ASR-33 Teletype, there is also the paper-tape equipment which again in its usage need have no relationship to the keyboard/typewriter. As a result, the normal driver which caters for the full range of the console terminals facilities is necessarily somewhat large and it is especially important that no program should pay the penalty for its presence in core unless it is actually required. In other words, this driver should be no different from those for other devices which are loaded in response to .INIT and unloaded by .RLSE as described in Section 3.2.1.

Nevertheless, the Monitor must have access to a driver of some sort at all times, despite the programs needs. Such driver can be more limited in its scope and need be interrupt-driven only. However, since any keyboard input must initially be considered a potential command, this driver must examine every character entered in case it should happen to be the CTRL/C needed to make the Monitor listen. Moreover, any character output might be the echo of a command input. The first part of this driver is therefore a module within the permanently resident monitor, as noted in the introduction to Chapter 2. This module intercepts all console terminal interrupts and then connects these appropriately either to the remainder of the Special Driver or to the full driver if this is in core at the service of the loaded program. That this module need be resident really goes without saying in view of the excessive overhead otherwise incurred at each console interrupt.

On the other hand, the rest of the Monitors driver only needs to be in core for the time taken by the operator to enter a command. Unfortunately, though, it cannot be loaded by the usual method for drivers for the following reasons:

a. The normally non-resident .INIT is needed and this is not necessarily available since the Swap Buffer might already be occupied by some other module servicing a program request, as noted at the end of Section 6.1.

b. The driver itself would occupy a buffer claimed from free core through the Monitor sub-routine S.GTB which was shown in Section 2.4.3 to be not re-entrant and which also might be in use on the programs behalf.

An alternative method has been adopted therefore; the Monitor driver is in fact a module called by its own EMT (31). Hence it is loaded by the normal processes of SAM into the subsidiary Swap buffer reserved for the keyboard command usage as outlined in Section 2.3.1.

In order to allow a running program to proceed as stipulated in Section 6.1, the command itself must be buffered until it is completely input and echoed. The necessary area for this cannot be provided within the KSB for, as will be shown in the following sections, this is used by the other command modules. Moreover for the same reason as that given above, the use of free core is out of the question. Hence the command is stored by the driver within the area allocated to its permanently resident interceptor portion discussed earlier. The resulting use of memory during command acceptance is illustrated at figure 6-1(a). It should be noted that in the interest of keeping the resident buffer area to a mini-

mum, it is not possible to allow more than one command input at a time. Hence as soon as a complete command has been entered, the keyboard interrupt facility is disabled until the execution of that command has been completely effected.

6.2.2 Command Decoding

Once the Special Driver has served its purpose of collecting the command, an appropriate routine must be called to service it. This routine must also use the subsidiary KSB. However, in most cases, each of the currently implemented commands requires a measure of processing for which the whole of the 128-word space allocated to the KSB is taken up as a minimum. Thus each command needs its own individual processing module, and which module this is must be determined from the command code entered. Hence, the driver calls an Interpreter which satisfies this function and also then provides for the necessary call to the module required.

The first of these operations is relatively simple to accomplish and naturally includes a check on the validity of the code. However, the subsequent call raises a problem. The Interpreter itself, for simplicity, can be and is allocated its own EMT code and the driver uses this as its means of transferring control. However, it is not feasible that every single command routine should absorb further unique EMT codes, particularly when additional commands may become necessary as the Monitor is developed further. The solution is the use of the special entry into SAM described in Section 2.3.2. This was shown to allow the calling of several modules by a single code through the following sequence:

1. Set R0 to show the system-device address and size of the individual module in MRT format (see Section 2.1.2).

2. Call EMT 30 (for KSB usage).

This means, of course, that the Interpreter must be able to supply the necessary information for R0 on the basis of the command code. For this purpose, a directly accessible index to the locations of the modules within the system Library must be available, for a complete search of the whole Library at each command is out of the question. However, the establishment of such index is a further problem, when in the normal way, Monitor modules can be stored anywhere within the Library and in addition the varying block sizes of the potential system-devices noted in Section 4.2 must be taken into account. Moreover, as implied above, several of the command processors themselves have the same problem because they need to overlay themselves in order to accomplish their

function. Currently, therefore, the index and the corresponding module storage on the system device is prepared as part of the assembly and linking of the Interpreter and the supporting command modules as follows:

1. Each module is conditionally assembled with a secondary parameter input which is based upon the intended system-device (see Section 8.1.1).

2. All the modules are linked together into one load module in a sequence fixed by that of the index. This then provides the correct detail of the relative position of each command processor within the complete block (see Section 8.1.2).

The index itself can potentially be too large for inclusion within the interpreter module. To overcome this, the index can be segmented across several device blocks immediately following the Interpreter in the Library. The general operations carried out by the Interpreter can be summarized as follows (see Section 6.3.3 for further detail):

1. Check whether the requested command is one which is processed by an embedded routine - because a fairly immediate response is needed (i.e. CONTINUE, WAIT, STOP). If so go to it.

2. Otherwise use the area occupied by these embedded routines as a buffer into which the appropriate block of the index can be read, then from the entry corresponding to the required command, set R0 and call the relevant routine by EMT 30.

The effect of these operations on KSB usage is illustrated at Figure 6-1(b).

6.2.3 Command Processing

It was shown in the previous section that, apart from the three commands having routines embedded within the Interpreter and those especially handled by the transient Monitor, all other commands have corresponding unique modules. It is their responsibility firstly to verify and process the argument string within the command and then to carry out the operators requirements and provide the appropriate return to the running program if need be.

In general, the arguments are entered in some format understandable to the operator. This means that conversion into machine code is necessary in all cases. Once again, the processing routine must presume that the normal Swap Buffer

is not available for access into the Monitor Utility Facilities described in Chapter 5 unless this is specifically allowed by the system-state at which the command can be given. Hence most routines contain their own embedded conversion functions.

The amount of processing needed to execute the command obviously determines the form of each module. In some cases, the whole operation can be completed within a single KSB frame, in others however this is impossible. As a result, overlaying is necessary and the processing is naturally ordered to allow a simple sequencing of the overlays, as shown in Figure 6-1(c), rather than partial substitution. Hence each overlay can be called by the same special SAM entry technique used to fetch the module initially. In each case, the system-device location of the next overlay is determined on the assumption that the complete set occupies numerically contiguous blocks as provided by the assembly/linkage process noted in the previous section.

6.2.4 Command Clean-up

The form of the exit taken by the command processor depends upon the command itself. Some commands, of course, imply that any program possibly running when the operator intervened is not to be recalled in the normal way, e.g. BEGIN, RESTART, KILL, ODT. In their case, the processors are responsible for establishing the correct machine-state before they leave. In particular, the processor-stack and Registers must be correctly set for the operations to follow.

In all other cases, however, the original state must be restored. Moreover, for any command entered there is always the possibility that, due to some error in syntax, time of entry or argument invalidity, the operators request must be rejected. To provide a common return to normal, the Special Driver module outlined in Section 6.2.1 is recalled to the KSB with appropriate switches set, as shown in the next section. This is then responsible for printing any error messages required and, if necessary, the $ needed to show that the system continues in a Monitor listening state.

The driver module performs the terminal printing on interrupt. In the meantime it exits through its resident section and in the process the stack and Registers are reset to their state prior to entry into the command processing operations. After the printing has been completed, the resident Listener carries out the final clean-up of its own internal switches. It then restores the keyboard for acceptance of a new command and the typewriter to any program output interrupted.

This procedure also leaves the Special Driver module in the KSB as shown in Figure 6-1(d). In general, therefore, it is immediately available for the entry of a new command without further system-device access.

6.2.5 Command Conventions

The general processing scheme outlined in the previous sections obviously means that certain conventions are established to permit the standard interface needed. These are summarized for reference in the following paragraphs:

6.2.5.1 Calling Parameters

In order to pass data between the various modules, the Registers are used and it is the responsibility of the calling module to set them accordingly (the receiving module of course collects their content from the stack as saved by the normal EMT process described in Section 2.2.1). In particular, the common interfaces to and from the individual command processors are as follows:

1. Entry from the Interpreter:

   R1 = Address of the last byte (i.e.CR) in the command input
   R2 = Command code (1 character per byte)
   R3 = Address of the Start of the SVT
   R4 = Address of the typewriter hardware buffer
   R5 = Address of the start of the command argument string (or if none = R1)

2. Exit via the Special Driver:

   R1 = Error flag (a negative code identifies an error detected - positive (any value) if no error)
   R4 = Address of the typewriter hardware buffer
   R5 = 0 to indicate processing completed

The remaining interfaces are detailed within the appropriate module descriptions in the following sections.

6.2.5.2 Processor Stack

Although the stack is not used to transmit data, each module must ensure that the correct state is passed to its succes-

sor. In general, this means that the contents of Registers saved at each internal EMT call must be removed. In addition, the only return taken in the normal way is that to recall the resident Listener on completion. Hence the return PC and status saved by the EMT must also be cleared in all other cases (see Figure 6-2)

6.2.5.3 Call Techniques

Since each module calls the next stage by means of an EMT, the normal SAM requirements described in Section 2.3 must be observed. In other words, the calling routine must ensure that the Swap Buffer it is currently occupying is released for the following user by decrementation of its Usage Count. Again, there arises the problem of doing this from within the buffer while the routine is still present. Other Monitor modules, it was shown in Section 2.3.4, solve the problem by calling a common System Exit in the resident Monitor. However this sequence is of no use to the command language modules, since they do not restore Registers or return. Instead therefore a special outlet is provided immediately following the KSB which they can access by "dropping through".

The first instruction in this sequence performs the necessary reduction in the Usage Count byte. The second is set by the routine to cause the requisite next action. Thus in the general case following:

```
        DECB    KSB     ;FREE SWAP BUFFER
        XXX
```

XXX is replaced by one of the following:

a. EMT 33 for the call from Special Driver to Interpreter

b. EMT 30 between interpeter and individual processor module (or between overlays of that module)

c. EMT 31 for the Special Driver recall from a processor module

d. RTI as the means of exit from the driver to its resident Listener.

It was shown in Sections 6.2.3 and 6.2.4 that, because R0 must be correctly set when the call is EMT 30, the modules and the Interpreter index are set up at assembly/link time to provide a fixed relative system-device location. This

means that the absolute position of the module currently in the KSR must also be known. This is readily available from either of the following sources:

a. R0 on entry (as stored on the stack by the EMT handler) - if also called by EMT 30

b. The occupant-identifier word in front of the KSR as set by SAM (see Section 2.1.3)

(The relative increment, of course, must allow for MRT format, i.e. 2*Relative Block #)

6.2.5.4 Reentrancy

It has been shown, in fact, that the keyboard language is executed by an overlaying scheme. This in itself automatically removes any possibility of re-entrancy from each of the modules concerned. No definite attempt has therefore been made to ensure that the modules do not break the rules in this respect (though in most cases, natural usage of the stack rather than of ordinary in-core locations tends to prevent this). In particular, the call technique described in the previous section is a deliberate violation.

Due to the lack of reentrancy capability, two features have been implemented and should be noted:

a. Any interrupt from the console recalls the Special Driver module. Although in reality, such interrupt should never happen, since both input and output sides are disabled while a command is processed, for absolute safety the priority level is maintained at 4 as set by the interrupt call initiating the execution phase. This also has the advantage that the command operations are completed as quickly as possible without stoppage to service the less critical devices using this priority level, without preventing the more important interrupts at higher levels(1).

----

1. There can be problems, however, if errors occur in the devices using the higher levels - see Chapter 7.

b. Because the tyewriter interrupt facility cannot thus be used to control the printing of information requested by the operator, the processors of commands for this purpose contain their own routines to access the hardware registers directly and wait between characters.

6.2.5.5 Residency

Elsewhere in this manual it has been shown that without true reentrancy, the residency of any module unprotected by SAM can also be dubious in certain situations. To some extent, this applies also in the case of the keyboard language modules. However two more important aspects must be considered:

1. The use of the special entry to SAM by EMT 30 as the means of access to the indivdual processors intrinsically presupposes that these will always be brought from the system-device when required. Moreover, the very reason for this practice - the avoidance of excessive allocation of EMT codes and hence of a longer MRT to cover them - means that the sole link to their location is through the Interpreter index based on system-device Library data only.

2. As well as preventing re-entrancy, the call technique described in Section 6.2.5.3 also implies non-residency since it can only work if the routine using it is within the KSB. As shown, this applies to all the language components not just the individual processors.

All in all, therefore the residency of any part of the keyboard command structure is out of the question and on this basis no other provisions for such a possibility have been made. In particular, none of the routines attempts to check its location in the same way that the other Monitor modules do, using the procedure described in Section 2.3.5. This means of course a departure from overall Monitor philosophy. However it is currently felt that strict adherence to the principle does not warrant the extra code, probably resident, taken up to implement it in every system - an especially important consideration for the user with the smaller configuration. Furthermore, even for a user with core to spare, it is expected that the keyboard operations will be very low in his list of priorites for modules to be made resident, firstly because of the predictably low relative frequency of usage and secondly because, even with system-device accesses, the response time for any command is not unreasonable in most cases - at least by comparison with the inherently slow nature of the console terminal device.

6.3 Common Command Processing

The object of this section is to discuss in detail the key-board language modules which provide for common acceptance, decoding, and clean-up for all commands. Section 6.2 indicated that three modules are in fact involved:

a. A permanently resident routine which monitors all console terminal interrupts and calls a special Special Driver when a command is under-way - see section 6.3.1

b. A non-resident driver which accepts, buffers, and echoes a command input and later provides a common exit for terminal printing on completion or error - see Section 6.3.2

c. An Interpreter which dispatches to the appropriate routine to process the command on the basis of its two-letter code - see Section 6.3.3.

6.3.1 Console Keyboard Listener (RMON5)

In Section 6.2.1 it was shown that the permanently resident Monitor must contain a routine capable of intercepting all console terminal interrupts since any one of these, whether input or output, is potentially appropriate to the entry of a keyboard command. It is then the responsibility of this routine to transmit each interrupt to the correct driver, whether this be the one especially provided to handle command input described in the next Section or the full driver if this is in core on a programs behalf. This routine also includes the storage space in which a command is buffered and echoed.

Call Sequence:

The routine is called by direct connection through the console terminal interrupt vectors (standardly locations 60-66). As shown in Section 2.1.4, these are set as part of the initialization sequence when the permanent Monitor is first booted into memory. A single entry for both input and output is used to enable a common sequence for the saving of the interrupted programs Registers and the setting of pointers. The processor Status Register however is given a different C-bit on the interrupt as the means of determining its source when unique operations must be performed, i.e. 0 = input and 1 = output.

General Description:

Basically both forms of interrupt lead to a check on whether or not command processing is required. If it is, the Special Driver is called into the KSR as indicated in Section 6.2.2. If the driver then returns control, either more input is expected or a command allowing normal program resumption has been executed. In both cases, the latest state of pointers - assumed correctly set - is saved and the interrupted program is recalled. When command processing is not underway, the routine verifies the presence in memory of the full console driver, using the content of KBA in the SVT (see Section 2.1.1) to access and examine the core-address entry for device KB in the DDL (see Section 2.1.3). If this is non-0, control is passed to the full drivers interrupt service routine to take the appropriate action. Otherwise the interrupt is ignored and the suspended program is recalled.

The internal buffer space is structured as illustrated at Figure 6-3. It provides in effect three main areas as follows:

1. Pointer Store - used to save pointers between interrupts and temporary data during processing as under:

   a. Command Buffer pointer - shows the next byte for input storage when a command is underway. At all other times it is cleared to 0 and thus also acts as a switch to show whether or not the Monitor is in a listening state, hence the Program Loader (see Section 5.1) and the Error Diagnostic print routine (see Section 7.2) can force such state - $ type- by merely setting the pointer. (its address is stored at the start of RMON5 to give these routine access through the console interrupt vector)

   b. Echo Buffer input pointer - indicates the last byte of echo stored ready for printing.

   c. Echo Buffer output pointer - is held at the last echo character despatched to the printer. This must always trail the input pointer so it is initialized one byte behind.

   d. Hardware Register pointer - is used to access the external page registers for the console terminal.

   e. I/O Switch - is set to the C-bit content on entry to the routine (see "Call" above)

f. Input Byte Store - holds an input character for checking or is 0 for an output interrupt. (It can therefore be used as an alternative I/O Switch - see next section)

2. Command Buffer - allowing for a full typewriter line of 72 characters since the command language makes provision for "off-line" comment preceeded by ; - see Programmers Handbook.

3. Echo Buffer - necessary because there may be more than one character to a single input, e.g. CTRL/C becomes ^, C, CR. (LF and . are output specifically by the driver - see next section). It begins with a single switch byte, which is used to store the console printer interrupt status appertaining to a suspended program (or 1) while command input is underway and is 0 otherwise; it ends with a stop-byte containing -1. Between these two bytes, a minimum area is provided on the basis of the following principles - (see next Section):

   a. The printer is always considered the slave of the keyboard when a command is being entered, any other operation by the program being suspended to accomplish this. Thus in most cases the echo can keep pace with its input.

   b. The buffer pointers are re-initialized as soon as the last stored character has been printed.

   c. For the longer echo sequences, such as CTRL/C and CTRL/U, the keyboard interrupt is temporarily disabled to prevent the input from getting too far ahead.

For the sake of completeness, the KSB is also included within the RMON5 module. As noted in Section 2.3.1, this is preceded by the word in which SAM saves the system-device data from the MRT to identify the current occupant . In front of this word, two additional data items are stored for easy access by the Special Driver, when this is in the KSB on recall after command execution:

   a. Dummy buffer - (using the high-order byte of the word containing the Echo Buffer stop byte) allows the driver to store $ for continuation of a Monitor listening state without the need to re-initialize pointers completely.

   b. Echo Buffer start - enables reinitialization of pointers when no continuation of listening state follows:

The KSB is followed by the three-word exit sequence described in Section 6.2.5.3

Exit state:

As noted in Section 6.2.5.1, data is transmitted through Registers on transfer of control to the drivers. This is as follows:

1. Call to the Special Driver:

   R0 = Start address of the Echo Buffer
   R1 = Address of next byte for input storage (or 0 if CTRL/C just entered)
   R2 = Address of the next character to be stored in the Echo Buffer
   R3 = Address of the last character printed from the Echo Buffer
   R4 = Address of the appropriate hardware buffer in the external page
   R5 = Address of the Input Byte Store

2. Call to the full driver:

   R0 = Input character or 0 if output
   R4 = Address of the appropriate hardware control status Register
   (C bit is set as on entry to show interrupt type, i.e. 0=input and 1=output)

In both cases, the interrupted programs Register contents are saved on top of the stack. For the Special Driver call, the address to the Pointer Store is above them and must be returned intact on recall (see Figure 6-2)

Detailed Description:

The outline of operations performed by RMON5 follows. Figure 6-4 illustrates the sequence in more detail.

1. Save the entry C-bit in the I/O switch byte in the Pointer Store (see above) and then the interrupted programs Register contents on the stack. Reset the Registers from the values held in the Pointer Store and remember the Store address on the stack.

2. If the I/O switch shows the interrupt was caused by output, go to step 8. For input, read the character entered and save a copy of it, stripped of bit 7, i.e. parity, in the Input Byte Store for

later checking. (ignore nulls within commands)

3. Test the Command Buffer Pointer. If it is non-0, a command input is already underway, hence go to step 7 to call the Special Driver. Otherwise check whether a switch was set at step 5 because the last character input was ESC or ALTMODE, either of which prevents special treatment for the character input this time (including CTRL/C). If so, clear the switch and go to step 6 provided that the full driver is in core (see "General Description" above)

4. If the input character is not CTRL/C, and the full driver is not in core, collect the Pointer store address from the stack and save the appropriate Register contents for next time. Go to the System Exit (see Section 2.3.4) to restore the programs Registers and return, thus ignoring the entry.

5. If the full driver is in core, look for ESC or ALT-MODE (i.e. ASCII codes 33, 175 or 176). Set a switch for next time when found (see step 3)

6. Determine the entry point for the full drivers interrupt service routine from the offset in its interface table (see Section 3.3.1), reset the C-bit from the I/O switch byte and go to it.

7. If the input character is CTRL/C, set a pointer to the start of the Echo Buffer and call the Special Driver by EMT 31. Should a return occur, because either more input is expected or the command has been executed, recall the interrupted program through step 4.

8. For an output interrupt, clear the input Byte Store as a switch for the Special Driver (see next Section) and disable the printer interrupt. If the Command Buffer Pointer is non-0 (command input underway) or the Echo Buffer output pointer is not at the buffer start (clean-up print still incomplete), go to step 7 to call the driver.

9. Otherwise examine the first byte of the Echo Buffer (see above). If it is a 0, the interrupt can only be program-initiated, hence go to step 5 to call the full driver provided that it is in core. A non-0 content, however, indicates that a command has been finally completed. Use such content to

reset the console printer interrupt status to its state prior to the command entry and clear the byte (1). Exit through step 4.

6.3.2 The Special Console Driver (KBL)

The purpose of the special command driver, as already indicated in Sections 6.2.1 and 6.2.4, is twofold:

1. It processes all console terminal interrupts arising during the input of a command, as passed on by the Listener discussed in the previous section, until a complete command has been stored in the buffer within the Listener. It then calls the Interpreter described in the next section to process the command as required.

2. It provides a common exit sequence for all command processing modules returning control normally to the program interrupted by the command, taking care of any error conditions detected.

Call Sequence:

For reasons given in Section 6.2.1, the Special Driver is called by its own EMT thus:

```
        EMT     31              ;CALL KBL
```

No arguments are passed on the stack; however it does expect the stack-top to contain Register contents saved by the EMT Handler (see Section 2.2.1), as listed either in the previous section when called by the Listener during the input phase or in Section 6.2.5.1 for the common exit recall.

General Description:

The content of the saved R5 on entry is non-0 for the Listener call. In this case, the driver is responsible for storing each valid input character in the Command Buffer within the Listener and for producing the appropriate echo.

----

1. If this reenables the printer interrupt, the Listener is immediately recalled after exit. By the check in step 9, the new interrupt is then passed on to the full driver, thus allowing program output suspended by the entry of the command to proceed.

Each character of input or output is handled at one interrupt and the pointers passed in the Registers are used for buffer access. The driver must also provide the special treatment afforded to certain characters detailed in the Programmers Handbook. When it recognizes that the echo for the CR terminating an input line has been output, the driver removes any comment content proceeded by ; and if any data remains, it calls the Interpreter. Otherwise it exits to the suspended program, perhaps with $ output if a Monitor listening state is to be maintained. The return in this case and from the intermediate interrupts during input acceptance is by way of the Listener.

The common exit operation follows detection of 0 in R5 on entry. If R1 is negative, the Interpreter or an individual command processor has uncovered an error in the input command. In this case, an overlay is called and this converts the implied error code in R1 into a plain language error message at the printer and the command is rejected. (such errors must be handled in this way because the normal Diagnostic Print routine also uses the KSB (see Chapter 7) and this of course is still occuped by the driver). For positive R1, indicating command processing has been satisfactorily completed - or after recall when the error message has been output, the driver cleans up and again returns to the Listener to exit to the interrupted program.

Exit State:

The Register contents passed on by the driver are as follows:

1. Interpreter call:

   R1 = Address of the byte containing the terminating CR
   R4 = Address of the console printer hardware buffer
   R5 = Address of the first byte of the command

.

2. Listener recall:

   R1 = Address of the next byte to be filled in the command buffer (or 0 if no further input is expected)
   R2 = Address of next byte to be filled in the Echo Buffer
   R3 = Address of the last byte printed from the Echo Buffer (or initialized as noted earlier one byte behind R2).

On the Interpreter call, the stack is passed with the Listener recall parameters on top, ready for eventual exit when the command has been processed. The Listener is recalled with the stack set as on entry from it. (This requires some clearance when the driver is used in its common exit role)

Detailed Processing:

Figure 6-5 illustrates the processing procedure of the Special Driver; an outline of the steps taken follows:

1. Restore Register contents passed by the calling routine. 0 in R5 identifies a call for the common exit operation of the driver; if found go to step 12. Otherwise examine the Listeners Input Byte Store through R5. When this again is 0, a printer interrupt has occurred, hence go to step 9.

2. Test the input byte for CTRL/C or CTRL/U. In either case, move ^ into the Echo Buffer followed by the corresponding letter and reset the input byte store to CR. Set the Command Buffer pointer for a new command input - and thus also the underway switch. Save the current printer interrupt Status, forced non-0 by bit 0 = 1, in the first byte of the Echo Buffer (unless this is already other than 0, indicating a fresh attempt at the command string). Disable the keyboard interrupt to allow the echo print to keep in pace and go to step 7 for exit as required.

3. If the input byte is RUBOUT, check the next Command Buffer byte for 0, signifying that this is the first entered (see below), in this case move ◊ into the Echo Buffer(1).

----

1. The check uses an algorithm based on the fact that if any 7-bit ASCII character is incremented, RUBOUT is the only one which becomes negative, i.e., 200 (a value with no positive complement). As indicated in the text, the next byte in the Command Buffer is set to 0, unless RUBOUT is seen when 200 is stored. Hence a negative comparison between the new input and this byte signifies ◊ print, as shown by the following table:

	INPUT BYTE	BUFFER BYTE	RESULT
NO R/O	<200	0	<200(+)
FIRST R/O	200	0	200(-)
SUCCESSIVE R/O	200	200	0(+)
FOLL. NON-R/O	<200	200	>200(-)

Move the last character from the Command Buffer in the Echo Buffer, if any remain (1), and replace it by 200 as a flag that a RUBOUT sequence is underway. Go to step 8 to exit.

4. For any other input, also perform the check on the next buffer byte and if negative as set by step 3 at the last input, move a terminating ◊ into the Echo Buffer (hence producing required print sequence: ◊ (deleted characters)◊)

5. Check for CR. When detected, store; in the Command Buffer as a stop and then scan the buffer from the beginning for the first occurrence of ; as the possible start of comment. Replace this with the CR entered to remove the comment. If no valid input remains after this, because either the whole line was comment or there was no line at all, clear the Command Buffer pointer and hence the underway state. Again disable keyboard interrupts to avoid Echo Buffer overflow (and prevent a new command input if one is now in). Go to step 7.

6. Ignore any other input if no more space exists within the Command Buffer. Otherwise store the new character.

7. Clear the next buffer byte as the "No RUBOUT" indicator and move the input byte into the Echo Buffer (or its replacement - see step 2)

8. Should no further room remain in the Echo Buffer, stop keyboard interrupts to allow the print operation to catch up. If then the printer is currently busy , recall the Listener to return to the interrupted program. (the new echo will be handled subsequently to the next printer interrupt) (2)

----
1. Because the input Byte Store immediately precedes the Command Buffer - see Figure 6-3, it follows from the previous note that if an attempt to collect the last character entered shows a negative value, it is in fact the new RUBOUT in the store. Hence no further deletions can be made on this basis.

----
2. If the command is just being started, the printer is working on the programs behalf in this case. Since the underway switch is now set, the Listener will pass the interrupt expected by the full driver into this routine. Hence the echo-print will then be initiated and program output will be suspended until the command has been executed.

9. When the last character dispatched from the Echo Buffer is not CR, check whether further echo printing is needed (mainly for direct entry from step 1). If so, send the next character to the printer with its interrupt enabled. Otherwise reset the Echo Buffer pointers and if necessary re-enable keyboard but not printer interrupts (see step 8). In either case, return to the interrupted program via the Listener to await a fresh call.

10. Follow a CR print with the despatch of LF to the console. If a valid command line now exists, call the Interpreter to process it.(1) For a command just started or restarted by CTRL/U or a second CTRL/C store . in the Echo Buffer, overwriting the stored CR to ensure its being within the available space. Exit through step 9 with the printer interrupt enabled; however, wait for next time to reenable the keyboard.

----

1. It should be noted that the LF print is still in progress during the operations carried out by the Interpreter and perhaps even after the command has been fully processed. Hence in step 11, $ is not immediately printed on completion recall - it is stored for output at the next interrupt.

11. If a new command must now be initiated because the present one either consisted of comment only (by step 5) or has been processed (by step 12), check the program exit. If this is to the System Wait Loop in the SVT, indicating that the system is in fact normally under the control of the Monitor in a listening state, initialize the Command Buffer pointer and exit as in step 10 with $ replacing . in the Echo Buffer. Otherwise clear the Command Buffer pointer to force new CTRL/C and exit through step 9 to reset the Echo Buffer pointers and re-enable input keyboard interrupts (1).

12. At the common exit recall, remove the return parameters stored on the stack and check R1. If this is positive, no errors were detected during the execution of the command just finished, so collect the start address for the Echo Buffer from the KSB preamble (see previous section) and exit through step 11.

13. For negative R1 signifying that error print is required, set up an EMT 30 call to an overlay assumed to start at the next adjacent system-device block in the system Library and exit to it. (2)

14. When the overlay is called, restore Registers passed by the driver and clear the return parameters from the stack. Use the value in R1 returned by the command processor to access a table of messages, each of which ends in a character with its parity bit set (bit 7). Output characters (in non-interrupt mode), until this character is detected and follow with CR/LF. Recall the driver proper by EMT 31 to complete the clean up as shown from step 12. (R1 now positive).

----

1. As shown in the previous section, the interrupt is reenabled to resume program printing as part of the final clean-up by the Listener - otherwise the printer remains disabled until pressed into some new service.

----

2. KBL must also therefore be assembled with a relevant parameter tape as noted in Section 6.2.2 for the other command modules. However, it must be linked alone as it has its own EMT.

6.3.3 Keyboard Command Interpreter (KBI)

The Interpreter exists, as discussed in Section 6.2.2, in order to ensure that the two-letter code entered as a command is valid and then to pass control to the appropriate routine to process the command, if necessary arranging for its loading from the system-device. If the code, however, is currently unassigned or corresponds to a command handled only by the transient Monitor (see Section 6.6), the Interpreter must so inform the operator.

Call Sequence:

The Interpreter is principally called by the Special Driver and therefore expects Register contents as listed for that modules exit state in the previous section. It also assumes that the driver may be recalled for the common exit sequence - hence the stack-state must be set accordingly (see Figure 6-2). No other arguments are needed, hence the simple call sequence is:

```
EMT 33          ;CALL KBI
```

General Description:

The Interpreter extracts the first two bytes from the command line passed by the calling routine and then moves the pointer beyond the first space (or comma) to the start of the argument string or to the terminal CR if none. For the WAIT, CONTINUE and STOP commands (see sections 6.3.3.1 through 6.3.3.3), the current system state is checked and where correct, the Interpreter modifies the program exit on the stack to reflect the change of state required by the command and takes such exit through a KBL recall.

All other command codes are covered by an index showing the system device locations of the corresponding routines as discussed in Section 6.2.2. Since the index proper is also stored on the system-device in segments, the Interpreter examines an embedded segment table to determine the block required. It then uses its own DDB to read that block on top of the WAIT, CONTINUE and STOP processors. If the command is valid, the appropriate entry in the index is used to build the system device data for the processing routine and this is then called by EMT 30.

For a command code not included in its expected index segment (by implication this also eliminates a single-letter command), or for an incorrect machine state for the embedded commands, the Special Driver is recalled to print the relevant error message - see below.

Exit State:

Data passed in Registers from the Interpreter to the individual command processor or on Driver recall is as listed in Section 6.2.5.1. In the recall case, R1 in particular may have the following content:

-1 = Command does not exist and is therefore Illegal
-2 = Command is Invalid at this time
+X = An input WAIT, STOP or CONTINUE has been executed

In all cases, the stack is left as at the point of call from the Special Driver (see Figure 6-2).

Detailed Processing:

The processing operations of the Interpreter are outlined below and are further illustrated at Figure 6-6.

1. Restore register contents passed by the Special Driver and clear its return parameters from the stack. Extract the first two characters from the command input and save them for later. If there is only one available before the terminal CR, go to step 9 to set up and call for error print.

2. Scan the string for the first occurence of space or comma and leave the buffer pointer at the next character. If the terminal CR is seen before this, hold the pointer on it.

3. If the input code represents a command processed within the interpreter, namely WAIT, CONTINUE and STOP, go to the appropriate routines as described in sections 6.3.3.1 through 6.3.3.3.

4. Set pointers to an internal DDB - see Section 3.1.2.3 - and to a table identifying the different segments of the index for the processing routines stored on the system device.(1)

5. Determine the index segment for the range in which the input code falls and compute the corresponding system-device block number for the segment on the basis of its relative position from the start of KBI itself, as indicated by the MRT data stored in front of the KSB by SAM (see Section 2.3.1).

6. Prepare the internal DDB for the reading of this block into the area in the KSB presently occupied by this set-up routine, the index-segment table and the embedded command processors (62 words currently). Use the DDL entry in the SVT (see Section 2.1.1) to obtain the address of the system-device driver and force the completion return to call the driver dequeue, validity check and DDB clear sequence set up for SAM (see Section 2.3.4). Set the DDB busy flag.

7. Save the current Register contents on the stack using the Monitor S.RSAV routine (see Section 2.4.1) and call the system-device driver via S.CDB (see Section 3.1.2.4). Call .WAIT until the read-transfer is complete. (This requires a user Link-block simulation, i.e., the DDB address is stored on the stack followed by a pointer to its stack position).

8. When recalled, restore the Register contents using S.RRES. Search the index segment for the command code and at the same time, build the system-device data for the corresponding processing routine, starting from the MRT entry for the Interpreter and accumulating the sizes of intervening routines. If the code is found, call its routine by EMT 37.

9. If the input command code is not in the index segment loaded, recall the Special Driver to print an "Illegal Command" message and return to the inter-

----

1. Currently one segment only is needed to cover the whole range of the assigned codes. However, for expansion, three segments are possible and if needed the table is a list of the last codes covered by a segment in each case - ascending order assumed - followed by -1 as a stop value.

rupted program, thus ignoring the operators request.

6.3.3.1 The WAIT Command

The WAIT command enables the operator to suspend a program temporarily while still allowing any I/O operations currently underway to complete. It is obviously valid only if a program is loaded and running, hence it can only be used after the input of CTRL/C and its format is thus:

```
^C
.WA[IT]<CR>
```

When given control by the Interpreter, the WAIT processing routine checks the MUS in the SVT (see Section 2.1.1). If this is not 1,1 indicating a program in and running, the command is rejected by a recall of the Special Driver to print an "Invalid Command" message and exit. Otherwise the upper byte of the MUS is complemented to give a content of -1,1 signifying that the loaded program is now suspended. In order to produce the desired wait state, the program return address saved on the stack is switched with the content of WRA in the SVT which is always initialized to the address of the System Wait Loop. Thus it is to this loop that the return is made, when the Special Driver is recalled to take the program exit, and the program resumption address is saved in case the operator later enters a CONTINUE command. By the normal processes of the Driver described in Section 6.3.2, the fact that the Monitor is now in a listening state is signalled to the operator by output of $. During the wait, the Registers are restored to their program content prior to the interruption caused by the command.

6.3.3.2 The CONTINUE Command

The CONTINUE command is the natural complement of WAIT. However it may also follow a suspension forced by the detection of some "Operator Action" error to be discussed in Chapter 7. Since it can only be meaningful if a program is loaded and suspended, the Monitor must be in a listening state. Hence the normal usage of the command is:

```
$CO[NTINUE]<CR>
```

Although a different mask is used to ensure that the MUS in the SVT is currently set to show the correct system state noted above, i.e. -1,1, the Interpreter calls the same routine to process the CONTINUE as was described for WAIT in

the previous Section for:

a. The MUS complement operation aptly produces the required new value of 1,1 - program loaded and running.

b. The address switch reinitializes WRA to the address of the System Wait Loop and the eventual exit through the Special driver resumes the program from the point at which it was suspended with Registers again appropriately restored.

6.3.3.3 The STOP Command

STOP is provided as an emergency command to halt everything - including current I/O. Its usage is similar to the WAIT command outlined in section 6.3.3.1, i.e.,:

```
^C
.ST[OP]<CR>
```

It is also processed by the same routine as WAIT, except that:

a. The MUS content is further complemented to give -1,-1 to signify that the program is not only suspended, it is also stopped. (Hence it cannot be followed by a CONTINUE to resume the programs operations - a necessary restriction, because of the potentially incomplete I/O).

b. Current I/O is killed by the use of a RESET instruction (the real-time clock also if this is present in the configuration).

6.4 Run-time Commands

In Section 6.2 it was shown that most of the commands available to the operator at the console keyboard can be given even though a user program is loaded. They therefore are processed by individual routines operating from the KSB. The purpose of this section is to consider each of these commands and the corresponding processing operations from the point at which control is received through the command Interpreter discussed in Section 6.3.3. It should again be noted however, that some commands are not included here because they are illegal unless the system is occupied by the Monitor alone. These are described under the transient Mon-

itor in Section 6.5.

The routines are discussed in the order in which they appear within the combined keyboard language load module mentioned in Section 6.2.3. It will be assumed that the reader is now familiar with the common calling and exit sequences and the corresponding state of the Registers and stack, as detailed in sections 6.2.4 and 6.2.5. The remarks on internal conversions also apply.

6.4.1 The DATE Command (KBI.DA)

In order that the current date can be available for inclusion within the identification of files held on bulk media (see Chapter 4) and perhaps for output on program listings, a word is reserved within the SVT (see Section 2.1.1) for its storage. The DATE command is provided as the means whereby the operator can initially enter and later examine the content of this word, with an external form in plain language. It can be used at any time.

Command Format:

For the entry of a new date, the DATE command has the form:

DA[TE] DD-MMM-YY<CR>

Where DD can be up to two decimal digits for "day", MMM represents the first three letters of "month" and YY shows the last two digits of "year", e.g. 1-SEP-71.

By omitting the argument, the operator requests printing of the currently stored date in the same "DD-MMM-YY" format.

General Description:

The basic operation of the DATE processor merely requires the appropriate conversion between the plain language "DD-MMM-YY" and the single-word Julian format in which the date is stored in the SVT. I.e.:

1000(Year-70)+Day of the Year

For either input or output, the computation of day of the year involves a search through a table of months, identified by their three letter code, and the corresponding days in each. Year and actual day are simple decimal ASCII/binary conversions - by embedded routines for reasons given earlier(1).

However both forms of operator request cannot be handled within a single module. Therefore on entry from the Interpreter, an immediate check is made for the presence of an argument and if none has been entered, indicating output required, an overlay is called to process it using EMT 30 as described in Section 6.2.

The command is rejected on the grounds of "Illegal Syntax" if a date being entered has incorrect format. An apparently invalid date already stored is printed as:

00-xxx-(Year)

Exit State:

Both the input and output sections of the DATE processor recall the Special Driver described in Section 6.3.2 as their means of return to the interrupted program. Hence Register and stack are as detailed in section 6.2.5. In particular R1, is set to -3 if errors in the syntax of the argument are detected.

----

1. Due allowance is made for February 29th in any four-yearly leap year. The century adjustment is however ignored (2100 is still a little way off)!

Detailed Processing:

The form of the DATE processor is illustrated at Figure 6-7. The main steps in the sequence of operations follow:

1. Restore Register contents passed by the Interpreter and remove its return parameters from the stack. If there is no data in the Command Buffer for processing or the first character of such data is not a decimal digit, compute the device address of the start of the next module in the system Library (in MRT format) and call it by EMT 30. Continue from step 6.

2. Save the current content of the SVT Date store and clear it as a buffer for the conversion of the input argument. Convert valid decimal digits into binary by the standard "multiply by 10 and add" algorithm. If the conversion is stopped by recognition of - and the result is not greater than 31, store it in the SVT. Otherwise go to step 5 to report the error.

3. Set a pointer to a list of months and their corresponding days. Search the list for the three-letter combination equivalent to that in the argument, at the same time building "day of the year" in the SVT store. Again go to step 5 if the required combination cannot be found or the subsequent input character is not -.

4. Return to step 2 to convert the decimal digits for "year". On return (forced by the fact that the SVT entry is no longer 0 when the conversion ends), look for a Leap Year. If found and the day of the year is beyond February 28th, increment it. Reduce the year value to 70, allowing that 00 entered now becomes 30, and set the result into the high order 7-bits of the SVT store. However reject the command through step 5 for any year seemingly below 1971 or above 2069. On completion, recall the Special Driver.

5. For any error noted in the previous process, set R1 for "Illegal Syntax" and restore the original SVT date before Driver recall.

6. When the output overlay has been loaded into the KSB, restore Registers saved on the stack by the call and remove the return parameters. Set a pointer to a list of month identifiers and corresponding days.

7. Collect the current Date from the SVT and extract the year by counting successive successful subtractions of 1000, starting from 69. If the result represents a Leap Year adjust the day value for February in the list.

8. Reduce the remaining day of the year by the day counts stored in the list as far as possible and hold the pointer at the relevant month identifier (1). If the list end is reached before the reduction is complete, reset the pointer for a month XXX and the remaining day to 0.

9. Convert the day into 2 decimal digits by the simple "Subtract -10" algorithm and send to the console pointer followed by -.

10. Print the appropriate month identifier from the list terminated by -. Return to step 9 to convert the year value, again providing for the output of 100 as 00, etc. On completion, restore the printer carriage by CR/LF and recall the Special Driver.

6.4.2 The SAVE Command (KBI.SA)

The SAVE command is provided to enable the user to produce external modules from a current core image upon any appropriate device and in a form acceptable to the Program Loader for later reentry (see Section 5.1). In order that the SAVE processor may have full access to all Monitor facilities, especially the File-management routines, the normal Swap Buffer must be readily available. Hence to avoid problems of conflict over MSB usage, as generally discussed in Section 6.2, the SAVE command may only be given at times when the loaded program is unable to perform, i.e. either before it is started after a GET command or after it is permanently suspended by a STOP command or some fatal error.

----

1. After step 7, the remainder is negative. The reduction is in fact therefore accomplished by addition until a positive value is detected starting with an increment to make December 31st = day 365 and then working backwards through the year.

Command Format:

Basically, the SAVE command may have one argument as follows:

$SA[VE] [Dataset specifier]<CR>

Where "Dataset specifier" must comply with the format for the Command String Interpreter (see Section 5.4),i.e.

[Device:][Filename[.Extension]][UIC][/RA:LOW:HIGH]

(RA being a switch identifying the range of the SAVE image between the octal addresses LOW and HIGH)

As indicated, elements of the specifier may be omitted in which case the following default assumptions are provided:

a. Device - the system-device as stored in the first entry of the DDL (see Section 2.1.3)

b. Filename and Extension - SAVE.LDA (also causes auto-deletion if such file already exists)

c. UIC - the identification code for the logged-in user as stored in the SVT (see Section 2.1.1)

d. LOW - the Program Load Address in the SVT (PLA)

e. HIGH - the highest address in available memory from the SVT (GSA)

General Description:

Because the actual SAVE operation can be a lengthy process and is likely to produce normal errors for which the KSB may be needed by the Diagnostic Print routine (see Chapter 7) the SAVE processor is divided into two main sections:

1. An initialization routine, called by the Interpreter into the KSB to check and decode the command argument.

2. The SAVE routine proper, brought into a 256-word buffer claimed from free core by the initialization routine to produce the required load image by normal I/O operations, with the KSB then freed for other usage.

The initialization process in fact uses the Command String Interpreter (see Section 5.4) for its operations. This me-

ans that the necessary buffers and control blocks must be provided. Owing to the limited capacity of the KSB and to avoid the use of core outside it until the input specification is completely verified, the routine is organized as straight-line code which successively overlays itself with the CMDBUF and Line buffer containing the input argument needed by module CSX and then the Command-, Link- and File-blocks expected by CSM. Finally an internal DDB (see Section 3.1.2.4) is similarly prepared for the transfer of the SAVE processor into its free-core buffer from the system-device. However if any error is detected, the command is rejected by a recall to the Special Driver with appropriate diagnosis (see below)

The SAVE processor itself must of course move the data established by CSI out of the KSB into its own Link- and File-blocks with default values supplied as relevant. Once done, however, the KSB is released and the Listeners Pointer Store is reinitialized (see Section 6.3.1). With keyboard interrupts then reenabled and processor priority reduced to Ø, the actual SAVE routine becomes just another running user program. It uses the standard .INIT, .OPEN, .WRITE,.CLOSE,.RLSE sequence to output the formatted binary data as illustrated under Section 5.1. During the .WRITE, a double-buffering scheme is adopted: this again utilizes a once-only code area within the SAVE processor to obviate the possibility of insufficent free core outside it.

The SAVE processor signals its completion by restoration of the implied Monitor listening state from which it was originally entered ($ output and forced input acceptance). It must then release the buffer it occupies. To accomplish this, it uses the same technique as the File-management modules in the same predicament (see Section 4.3), in which the Monitor S.RLB routine is called and returns through the normal System Exit described in Section 2.3.4. This restores the Registers which were originally saved on the stack by the Listener before the command was processed and which are still there . The final exit returns to the System Wait Loop.

Any errors arising during the actual SAVE cannot of course be handled through the Special Driver once this is no longer at the processors disposal. Hence the appropriate messages are printed by the processor itself before it returns to the System Wait Loop as described in the previous paragraph. I.e.

a. File error XXX - where XXX is the error code returned in the File-block Status (see Programmers Handbook)

b. Device Full

Exit State:

During the initialization process, the Special Driver may be recalled to handle error messages, hence the Register and stack state must be as prescribed in Section 6.2.5 with R1 in particular set as follows:

-2 = A user program is not loaded or is not stopped hence the command is Invalid
-3 = Faulty Syntax
-6 = Illegal limit addresses, i.e. LOW >HIGH, or outside core
-7 = No buffer space available for the actual processor.

As noted above, the actual SAVE routine does not return through the common keyboard language exit. Thus it ensures itself that the Registers and stack are returned to their state prior to the command interrupt.

Detailed Processing:

As indicated under "General Description" above, the SAVE operation is performed in two parts. These are described in the following paragraphs. The initialization process, being mainly straight-line code needs no illustration. It is outlined below:

1. Restore Register contents passed by the Interpreter and remove its return parameters from the stack. Compute the system-device block for the SAVE processor proper on the basis that it follows the initializing routine in the system Library and save for later. Set a pointer to the start of the routine as the beginning of a CSI CMDBUF with a Line buffer following. (see Section 5.4)

2. Check the MUS in the SVT (see Section 2.1.1). If not set to either 1,0 (program loaded but not started) or -1,-1 (program suspended and stopped), recall the Special Driver to print an "Invalid Command" message & ignore the command. Ensure the MSB is free by clearing any current Usage Count (see Section 2.3.1)

3. Move the command argument string unto the "Line Buffer" section relative to the pointer set in step 1 and ensure that a correct line terminator fol-

lows. Call CSX (Section 5.4.1) to check the overall syntax. Recall the Special Driver to print an "Illegal Syntax" message if in error and reject the command.

4. Prepare dummy Link-, File- and CSI Command-blocks. (Error return addresses & Status words for the first two are unnecessary at this stage - however the Link-block must allow room for a four-word switch - see Section 5.4.2). Indicate output specification required in the CSI CMDBUF and call CSM to decode the argument, with error action as described in step 2.

5. If the RA switch is present in the command string, call the Conversion Utilities routine (see Section 5.3) to convert the limits given to binary values and store them in the Link-block instead of their start pointers as returned by CSM. If LOW is not less than HIGH or HIGH is above available core, recall the Special Driver to tell the operator and again reject the command.

6. Via the Monitor S.GTB routine (see Section 2.4.2), claim a 256-word buffer from free core. If none available, inform the user through Special Driver recall

7. Prepare an internal DDB (see Section 3.1.2.3) to read the 256-word SAVE processor into the new buffer from the start block computed earlier. Extract the system-device driver address from the DDL (see Section 2.1.3) via its pointer in the SVT and force the completion return to use the SAM sequence described in Section 2.3.4. Use spare DDB locations to save relevant Registers (and set the DDB busy flag)

8. Call the system-device driver to effect the transfer via the Monitor S.CDB routine (see Section 3.1.2.4) and call .WAIT till done (like step 7 in the Interpreter in Section 6.3.3, this requires a Link-block simulation). When recalled, restore registers saved in the previous step and jump to the loaded routine.

The SAVE routine proper is illustrated at Figure 6-8. Its basic steps are as follows:

1. Give the SAVE routine now the appearance of a loaded running program by resetting MUS in the SVT to 1,1 to allow normal usage of the keyboard language during the output phase (after saving the original

MUS content)

2. Move into an internal full Link-block the device name and unit stored in the KSB dummy by CSM or, if none, the name of the system-device from the DDL with unit set to 0(1)

3. If the KSB dummy File-block contains a file-specification, move the relevant data into an internal full block to overwrite the default assumption SAVE.LDA already included and set a switch to prevent automatic deletion if the file is seen to exist later (see step 6). Also store any UIC specification.

4. Set pointers to use the area occupied by the processor up to and including the call for .OPEN in step 7 as a double-buffer. In the first of these buffers, prepare a standard COMD block for the load image as described in Section 5.1, using the Range limits, if any have been supplied, or the Program Load Point and the top of available memory if not. Fill in the remaining items, e.g. Program and ODT start addresses and program name, from the existing entries in the SVT.

5. Remove the keyboard Listener recall addresses from the stack and use its saved pointer to reset its Pointer Store (see Section 6.3.1). Simulate an interrupt return on the stack with 0 priority level and use the KSB release sequence stored below it (see Section 6.5.3) with RTI as the final instruction. On recall, reenable keyboard interrupts.

6. Request .INIT upon the output dataset and if the specified file is the default SAVE.LDA, try to delete it. (The File-block error return is set to come back at the same point if it does not exist).

7. Reset the File-block error return to call a sequence which prints a "File Error" message and exits through step 11. Call .OPEN. Amend the buffer

----

1. The error return in the Link-block is set to 0; thus if the requisite buffer space for normal I/O operations is not available, a SAVE command can result in a standard fatal error (F007) - see Section 3.2.1.1.

pointers for filling the second buffer and initiate a formatted binary .WRITE for the COMD block now in the first.

8. Store the current load point represented by the start of the data area now being output followed by the actual data words until the buffer is filled or until the high limit is reached. Compute the number of bytes moved and store in the appropriate third word of the Line header (see section 3.2.2). Call .WRITE to output the buffer now filled (in formatted binary)

9. When recalled, check whether EOD occurred at the last .WRITE. If so, print a "Device Full" message and go to step 11 to exit. Otherwise switch the buffer pointers - the other buffer must now be empty. Go back to step 8 if more data remains to be output.

10. When the high limit is reached, prepare and output a terminal Transfer Block using the address stored at PSA in the SVT (see Section 2.1.1). Then call .CLOSE and .RLSE on the dataset.

11. Restore the original system-state saved at step 9 to the MUS in the SVT and ensure that the System Wait Loop is recalled on exit. Output $ and force command input acceptance by resetting the keyboard Command Buffer pointer (see Section 6.3.1.). Store the system exit address on the stack and jump to the Monitor S.RLB routine (as noted under "General Description" above.)

6.4.3 The ECHO, END & PRINT Commands (KBI,KB)

This Section covers three commands which are primarily intended to enable the user to control the console terminals operations on behalf of a running program. Because their processing is similar, they are included within the same module; in particular they are all invalid unless a user program is loaded and the full driver is in core. The commands are:

a. ECHO - which allows flip-flop control of the echo printing associated with keyboard but does not affect normal output.

b. PRINT - which provides the same facility for normal output but not echo-print.

c. END - which enables the user to signal the end of a set of input data.

Command Format:

The basic format acceptable for all three commands is the same:

```
EC[HO]   [Device]<CR>
PR[INT]  [Device]<CR>
EN[D     [Device]<CR>
```

where "device" is the conventional two/three character code for the device concerned (without :) and is by default KB

However the device in the case of ECHO and PRINT must be a terminal, the only one currently implemented being KB - hence it is omitted in the present description in the Programmers Handbook. Nevertheless it can be specified for other terminal devices, provided that the associated driver is set up to use the results of the processing described below. On the other hand, END does not have the same restriction but again the only driver which recognizes its operations is that for the standard ASR-33 Teletype. Hence KB and PT are the only meaningful specifications.

General Description:

Provided that all the requirements noted in the previous paragraphs are satisfied, the processing for the three commands requires access to the byte in the device driver Interface Table which contains the priority level used to set the interrupt vector (see Section 3.3.1). Since this information is only required when the driver is first loaded, the byte can now be used as a set of switches as follows (assuming an initial value of 200, i.e. PRL4):

```
Bit 7=0 - EOD
Bit 4=1 - Suppress echo (operator-control)
Bit 3=1 - Suppress echo (program-control)
Bit 1=1 - Suppress print (operator-control)
Bit 0=1 - Suppress print (program-control)
```

The ECHO and PRINT commands merely complement the relevant bits (4 and 1) respectively; END clears bit 7. It is then the responsibility of the driver to check the appropriate bit as required and to act accordingly.

It should be noted that because no attempt is made to call the driver, the command can only be effective when the driver again has control. If in fact, the program has alre-

ady requested .READ and the driver has then returned to await an interrupt, an END request particularly must be followed by a new program input to force such interrupt. Hence it is recommended that a double CR be used to terminate the command. The user is also warned that once the EOD switch is set, it is cleared only by an OPEN call to the driver or by a reload on .INIT after .RLSF. Thus if more than one dataset is sharing the driver, the EOD applies to all - a potential problem if say both command strings and data are being entered from the same keyboard.

Exit State:

All three commands result in Special Driver recall on completion or error. Hence Registers and stack are set as shown in Section 6.2.5. R1 in particular may indicate the following error conditions:

-2 = No program loaded, so the command is Invalid
-3 = Faulty Syntax (i.e. non-alphabetic characters in device name)
-4 = Illegal Device (which does not exist, is not in memory, or is not a terminal for ECHO or PRINT)

Detailed Processing:

The sequence of operations needed to process the three commands is outlined below. It is straightforward and thus needs no supporting illustration:

1. Restore Registers passed by the Interpreter & clear its return parameters from the stack. If the low-order byte of MUS in the SVT is not set to 1 to show a program is in core (see Section 2.1.1), reject the command by recalling the Special Driver to print an "Invalid Command" message.

2. Convert up to the first three letters of any argument string entered into left-justified radix-50 format (see Section 5.3). (Any non-alphabetic character other than space, comma or CR is "Illegal Syntax"). If no argument is given, substitute KB.

3. Using its SVT pointer, access the DDL (see Section 2.1.3) and look for the device (KB acting also for PT since both devices share the same driver - see Appendix A). Reject the command on the grounds of "Illegal Device" through the Special Driver if either the device is unknown or its driver is not in

core.

4. For ECHO and PRINT, check the nature of the device by examination of the Facilities Indicator in the driver Interface Table (see Section 3.3.1). Again the device is illegal unless it is a terminal.

5. Set an appropriate mask for the command required and perform the relevant bit operation as noted under "General Description" above (complement = ECHO/PRINT; clear = END). Return to the interrupted program through the Special Driver.

6.4.4 The ODT Command (KBI.OD)

LINK-11 gives a user the opportunity to identify ODT-11R as a module within a program load-image by its /OD switch. (see DEC-11-ZLDC-D). The start address of the module is then stored within the COMD block in front of the load module (see Section 5.1) and during the load process it is transferred to the SVT (see Section 2.1.1). The ODT command then gives the user the means,whereby the debugging routine can be called or recalled regardless of whether a running program is in control or has lost it as a result of some inherent failure.

Command Format:

As shown in the relevant manual (DEC-11-OODA-D), three entry points into ODT-11R are possible. Hence the command has the form:

OD[T] [Code letter]<CR>

Where "code letter" may be as follows:

a. (null) = completely initialize ODT-11R (ODT+0)

b. R = Remove breakpoints currently set (ODT+2)

c. K = Keep current breakpoints (ODT+4)

General Description:

By implication, a program must be loaded (even if it is a linked version of ODT-11R alone) and ODT-11R must be included within it. The command is rejected as invalid by Special Driver recall otherwise.

The command argument is used to determine the ODT-11R entry, provided that it is a recognized value. However before control is transferred, the ODT command processor must assume that the operator will expect conditions appropriate to his program and may later resume the program directly from the debugging routine. Therefore, the system is first placed in a program-run mode, all evidence of a command being underway is removed and the exit is taken with the Registers and stack restored to their state prior to the command interrupt.

Two points should be noted:

1. The Monitor does not attempt to initialize ODT-11R either on loading or after an ODT command. This is left to ODT-11R itself, if so called by the user. In particular, the debugging trap location at location 14 must be correctly linked. Thus the first ODT command should have no argument and should be entered as soon as possible after loading.

2. ODT-11R always restores breakpoint locations temporarily while it is in control, to allow the user to examine and modify these like any other; it only replaces them when the user calls for program continuation by its own commands for the purpose. If the user instead enters CTRL/C to recall the Monitor, starts the program by a Monitor command and then later uses the ODT command with an argument, ODT-11R attempts its restoration with its initializing value 000003 thus destroying the original content. The user is therefore advised to exercise some caution (see "Getting DOS on the Air" - DEC-11-SYDD-D)

Exit State:

If the Special Driver is recalled through error, Registers and stack are set as shown in Section 6.2.5. In particular R1 contains one of the following values:

-2 = ODT-11R is not in core, so the command is Invalid
-3 = Code letter is unknown, hence Faulty syntax

As noted above, commands acceptance results in an ODT-11R call with program state restored.

Detailed Processing:

The simple sequence is as follows and requires no illustration:

1. Restore Registers passed by the Interpreter and remove its return paramters from the stack. Reject the command by recalling the Special Driver to print an "Invalid Command" message if either the MUS in the SVT does not contain a low-order byte of 1, signifying that a program has been loaded (see Section 2.1.1), or DSA in the SVT is set to 0 (ODT-11R not present).

2. Modify the content of DSA by 2 for an argument R or 4 for K. Similarly reject the command for any other input on the grounds of "Illegal Syntax". Store the resulting address on the stack in place of the return to the interrupted program.

3. Clear the return to the Special Driver from the stack. Using its saved address, reinitialize the Listeners Pointer Store (see Section 6.3.1). Wait until the console printer has completed the command LF terminator (see Section 6.3.2).

4. Set the MUS in the SVT to 1,1 to signify "program in and running" and ensure that WRA is initialized to the address of the System Wait Loop (in case the command followed a fatal error - see Chapter 7)

5. Restore the Registers saved at the command interrupt - and thus clean-up the stack and exit to ODT-11R at the required entry point.

6.4.5 The BEGIN and RESTART Commands (KBI.BE)

The BEGIN command is provided as the means whereby the user can initially start the execution of a program already loaded by a GET command or can later start over with a clean slate, perhaps after a fatal error. RESTART allows the user to transfer control to a different point in a running program, possibly as previously established by the program itself using the "Set Restart Address" facility within the General Utilities package (see Section 5.2). In this case, current I/O is stopped but all linkages remain. Both commands share a considerable amount of common code; they are therefore initially processed by the same module.

Command Format:

Basically, the command format for both BEGIN and RESTART is the same i.e.:

BE[GIN] [Octal address]<CR> (1)
RE[START] [Octal address]<CR>

An address argument supplied in either case becomes the start point for program execution. If omitted, the following default values are used:

a. BEGIN - the automatic start address included in the program source .END statement and as a result of assembly, linking and loading, now stored in PSA in the SVT (see Sections 2.1.1 and 5.1). (Maybe 1 - an illegal address - if

----

1. A reader who has the listing of the KBI.BE module may have noticed that a variation in the BEGIN format is possible, i.e.:

BE[GIN] [Octal address] [S]<CR>

where the optional argument S, meaning SAVE, prevents the normally automatic deletion of files currently open for output. This was provided originally for the benefit of a user who had already amassed a lengthy file, perhaps of irrecoverable data, by the time a program failure occurred and would therefore not want to lose it. However, there are two major problems inherent in the use of this argument:

a. The SAVE operation uses normal File-management procedures to close the file: in particular this means writing out data currently in memory to the device. Since, in general, BEGIN in this case is requested only because of program failure, there is a high probability that this is corrupted data, thus spreading the effect of the failure to the device. With the program then begun again, the eventual result might be a disaster.

b. Since the file would now exist (or files, since the process is non-selective), the program must be set up to use a different File-names for the rerun or a second failure occurs.

Therefore this option has not been advertized; instead such user is advised to KILL the program for, although this also performs a normal .CLOSE, he is then in a position to use PIP-11 to pick up the pieces before it is too late. References to this feature are omitted in later discussions.

no source address is supplied)

b. RESTART - the latest address stored at RSA in the SVT by a program request through GUT.

Both commands require only that a program be loaded; its run state is immaterial. However because RESTART must be given an address and performs less initialization, its use before the program has been started seems pointless.

General Description:

As noted above, the processing for both BEGIN and RESTART commences at the same module, firstly to verify the presence of a loaded program and secondly to check the validity of the start address whether this be a conversion from an argument supplied or the relevant default. Failure in either case results in rejection of the command by Special Driver recall.

The two commands also share a common sequence for general clean-up such as killing current I/O by hardware RESET, re-initializing the stack, the keyboard Listeners pointers and various SVT entries and ensuring the vacancy of the MSB. Apart from removing any I/O Busy Flags, RESTART processing is then complete. A direct exit is therefore taken to the user program at the start point specified, with Registers restored to their content prior to the command interrupt.

BEGIN however requires further action and this is effected in two overlays called consecutively into the KSB by EMT 30 (see Sections 6.2.3 and 6.2.4). In the first overlay, all the I/O linkage currently established is examined in detail via the Monitor DDB chain. Again any busy flags are removed, but in addition each dataset is checked for a file remaining fully opened (see Chapter 4). If such file is contiguous or is linked and in use only for input, it is merely closed. A linked file being created on the other hand must be deleted to allow the program to reopen it by the same name without a "File Exists" error. On a DECtape, this is effected by removing the name from its directory; on disks it also means that the blocks already allocated must be unlinked - an operation carried out internally using .TRAN. This avoids the necessity for closing the file first, a potential cause of disk corruption (see Footnote to "Command Format"). An Extension to an existing disk file is similarly removed.

The second overlay is called when the I/O checking is complete. This then unlinks any buffers still allocated from free core, i.e. Device Assignment Table extensions made after the program start (see Section 3.1.2.2), the DDB chain

and device drivers normally non-resident, and effectively returns those buffers to free core by clearing the Buffer Allocation Table (see Section 2.4.2). It also resets the interrupt vector tied to the drivers thus removed and ensures that the resident drivers are unlinked from any FIB chain (see Section 4.1.3). Finally it calls the program as specified with all Registers cleared.

Exit State:

If the Special Driver is recalled on error, the Registers and stack state are as shown in Section 6.2.5 with R1 set as follows:

-2 = Program not in core, hence the command is Invalid.
-3 = Address argument contains other than octal digits or is too big, thus Faulty Syntax.
-6 = Address to be used (including a default) is Illegal because it does not fall on a word-boundary or is outside the potential program area, i.e. above the resident Monitor and below the top of available core.

As noted under "General Processing", an accepted command restores the system to normal program run-state with the stack cleared to Program Load Point and Registers reset to their content prior to a RESTART or 0 for BEGIN.

Detailed Processing:

The common processing for BEGIN and RESTART is straightforward and needs no illustration. Its basic steps are as follows:

1. Restore Registers passed by the Interpreter and clear its return from the stack. Reject the command as invalid through the Special Driver if the MUS in SVT contains a low-byte of 0 (program not loaded - see Section 2.1.1)

2. If any argument is supplied, convert valid octal digits into their binary equivalent until either a recognized delimiter (space, comma, or CR) or single-word overflow. (Again reject the command for Faulty Syntax for illegal characters or such overflow). Substitute the appropriate default from the SVT for no argument.

3. Verify the legality of the start address, with similar rejection if it is not on a word-boundary or within the program area as defined by the top of the resident Monitor and that of available core.

4. When the console printer has completed the command LF output (see section 6.5.2), kill all I/O by RESET, but restart the line-clock if present (as shown by its vector pointing elsewhere than the Trap error routine (see chapter 7)

5. Prepare a new stack immediately below Program Load Point as saved in the SVT, with its first entries being 0 for program-execution priority level and the specified start point (1)

6. Remove from the present stack the recall parameters for the Special Driver and use the address for the Listeners Pointer Store saved below them to restore normal run-time keyboard control (see Section 6.3.1).

7. Release the MSB by clearing any outstanding Usage Count and its Occupant-Identifier (see Section 2.3.1). Also ensure that WRA in the SVT contains the address of the System Wait Loop (see Section 2.1.1).

8. Call the first overlay from the next system-device block if the command is BEGIN. For RESTART, trace the Monitor DDB chain from its start in the SVT (see Section 3.1.2.3) and remove Busy Flags from the DDBs and their associated device drivers together with any 0 linkage (see Section 3.1.2.4)

9. Set the MUS in the SVT to show "program loaded and running" 1,1. Restore the original program registers, reset SP to the new stack prepared at step 5 and exit by RTI.

The overlay processing now performed for BEGIN is illustrated at Figure 6-9. Again the following are the basic operations:

1. Restore the Registers to their state prior to the overlay call and clean up the stack. Get the start

----

1. If the stack was empty before the command interrupt these items overwrite the interrupt return and hence cannot corrupt any other valuable data (see Figure 6-2).

of the Monitor DDB chain from the SVT and if there is no linkage currently go to step 7 to call the next overlay.

2. Remove the SVT DDB-link. For each DDB in the chain (again see Section 3.1.2,3), clear its Busy Flag, that of its associated device driver and remove any Q linkage. Go to the next DDB in the chain if either the Driver Interface Table). (see Section 3.3.1) shows the device to be non-file-oriented or though the device is file-structured, there is currently no file open, (i.e. Open-flag is clear and no FIB is attached - see Section 4.3.2)

3. Otherwise ensure buffer availibility by preparing one on the stack of a size set as the standard for the device (as shown in the Driver Interface Table). Link this buffer to the DDB.

4. Using a dummy Link-block overlaying the initial once-only code of this routine, call .CLOSE if data in the FIB indicates that the file is contiguous or is open only for input if linked. Lose the stack buffer and return to step 2 when done.

5. For a linked file open for creation or Extension, prepare a dummy TRAN-block (see Section 3.2.1.1) - again on top of the start of the routine - and use it to read in the appropriate directory block from the device. (The FIB contains its identification - if none the .OPEN cannot be complete - so return to step 2). Clear the File-name from the relevant entry for a new file or unlock an old one being extended and write the block back. (The old file still retains its former identity in the latter case - see FIB description).

6. No further action is required for DECtape, hence check and return to step 2 if the driver Facility Indicator shows this is the device in use. Otherwise, again by .TRAN, trace the blocks of the file, starting from the FIB entry for "First Block" and finishing when a 0 link is encountered, clearing all link-words in the process. On completion, return to step 2, again with the stack-buffer removed (1).

7. Call the second overlay from the next system-device block and when in, restore Register contents as passed.

8. Collect the start address for any Device Assignment Table from BAT in the SVT. If non-0, trace the table linkage (see Section 3.1.2.2) and unlink all segments above the top of the resident Monitor as shown at EOM in the SVT - since these must occupy free core buffers (see Section 6.4.9 also).

9. Clear all words in the Buffer Allocation Table between the limits stored at BFS and BFE in the SVT. Reset TOP and its stack-stop (see Section 2.4.1) from EOM with the standard 40-byte safety margin.

10. Search the DDL (see Section 2.1.3): for permanently resident drivers (signified by no system-device data) which are file-structured, clear bit map pointers; for non-resident drivers, clear their core addresses in the DDL and reset the device interrupt vectors to trap to the error routine.

----

1. It should be noted that no attempt is made to modify the bit-map segments stored on the device, again to avoid possible corruption. As shown in Section 4.1.3, the block allocation scheme uses an in-core segment during file-creation. Hence it follows that if this is not written to the device, the blocks given to the deleted file within its range remain free for further use. However, there may still be some blocks which were allocated from a map segment which became full and was therefore replaced in memory. These are not reclaimed. It is possible, as a result, that a "Device Full" error can occur later even though other evidence contradicts this. Thus the user is advised to avoid excessive use of BEGIN as a means of resumption after program failure, particularly if longish files are involved. (see "Getting DOS on the Air" - DEC-11-SYDD-D).

11. Clear all Registers and exit to the program with the MUS in the SVT again set to 1,1 and the stack removed to program load point.

6.4.6 The KILL Command (KBI.KI)

The KILL command enables the operator to request the unloading of a program presently in core. If necessary it stops the programs operations including current I/O. However it establishes conditions which permit the transient Monitor now brought into core to close any files left open and release all datasets still initialized.

Command Format:

No arguments are needed, so the command is simply:

KI[LL]<CR>

General Description:

If a program is seen to be loaded, all I/O is killed by a hardware RESET and, by a search of the Monitor DDB chain, all I/O_busy states are removed. At the same time, if any linked file is still open for output, a final block with a 0 link is prepared for later dispatch when the transient Monitor calls CLOSE. (see Section 4.1)

The actual KILL is effected by a .EXIT call (see Section 5.5) and, of course, the Special Driver is not recalled. Hence the command processor ensures that the MSB needed by XIT is free and reinitializes the keyboard Listeners pointers. Finally to allow XIT ample room and prevent possible failure, the stack is set at the top of available memory before the EMT 60 call is made.

Exit State:

If the KILL command is accepted, current Register and stack states cease to be relevant. However if no program is loaded, the Special Driver is recalled to reject the command. In this case, the requirements noted in Section 6.2.5 apply and in particular R1 is set to -2 to request an "Invalid Command" message.

Detailed Processing:

KILL processing takes the following sequence; no illustration is needed.

1. Restore Registers passed by the interpeter and remove its return parameters from the stack. Reject the command as invalid through the Special Driver if the MUS in the SVT (see Section 2.1.1) contains a low byte of 0 = program not loaded.

2. Wait for the console printer to complete its output of the command LF (see Section 6.3.2). Kill all I/O by RESET, but restart the line-clock if present (as shown by its interrupt vector pointing elsewhere than the trap error routine (see Chapter 7)).

3. Get the start of the Monitor DDB chain from the SVT (see Section 3.1.2.3). Go to step 5 if no DDBs are currently established; otherwise for each DDB in the chain, clear its Busy Flag and that for any driver asigned to it and remove any Q linkage (see Section 3.1.2.4)

4. If, after examination of its Facility Indicator (see Section 3.3.1), any driver is seen to be file-structured, check for a file still open for output. If such file is linked and apparently correctly underway, clear the link-word in the data buffer attached to the relevant DDB and set a variable pointer in the DDB Driver Word Count to indicate that data still remains for output when later closed.

5. After all the DDBs have been processed, remove the Usage-Count and Occupant-Identifier from the start of the MSB to release it for further use (see Section 2.3.1)

6. Remove the Special Driver recall parameters from the stack and use the saved Listener Pointer Store address now on top (see section 6.3.1) to reinitialize the Listener.

7. Move the stack pointer to the top of core (below the possible Paper-tape System Loaders) and call .EXIT to fetch the transient Monitor to perform post-program clean-up.

6.4.7 The TIME Command. (KBI.TI)

In Section 2.1.4, it was shown that if a line-clock is present in a DOS configuration, the Monitor initialization routine automatically sets it into operation by enabling its interrupt facility; Section 2.5 then outlined the resident Monitor module which uses each interrupt to increment a double-precision counter stored as TOD in the SVT. The TIME command exists so that the operator can enter a daily initial setting for TOD and can later interrogate its content.

Command Format:

The general form for the command is as follows:

TI[ME] [HH:MM:SS]<CR>

Where HH, MM and SS each represent up to two decimal digits for hours, minutes and seconds.

As with the DATE command, discussed in Section 6.4.1, the presence of the argument identifies the entry of a new time, its absence a request for print-out in the same format.

General Description:

Provided that the first character of any argument in the command is a valid decimal digit, the TIME processor assumes TOD input and converts each decimal ASCII element into its binary equivalent. If each value is then within its expected limit, it is reduced to clock-ticks after effective multiplication by an appropriate factor. The sum of the three results is stored in the SVT Time slot in the defined double-precision format, i.e. with each word consisting of a positive (0) sign followed by 15 numeric bits. If any element is omitted, a default value of 0 is substituted (e.g. "TI 12" is acceptable in lieu of "TI 12:0:0"); however any other departure from the specified format causes rejection of the command on the grounds of "Illegal Syntax".

TOD print-out is supplied as a result of there being no command argument or of one beginning with a non-digit. The current content of the SVT Time counter is successively divided by an appropriate value to produce the required ASCII digits which are then dispatched to the console printer with separating colons and a terminating CR/LF.

Exit State:

Regardless of the processing performed, the TIME routine returns to the interrupted program by Special Driver recall. Hence the Register and stack state are as detailed in Section 6.2.5, if in fact the command is rejected because of Faulty Syntax, R1 contains -3.

Detailed Processing:

Both the input and output functions use similar successive addition and subtraction algorithms for multiplication and division. The processing sequence therefore is as illustrated at Figure 6-10 and is basically as follows:

1. Restore Registers passed by the Interpreter and clear return parameters from the stack.. If the first character of any argument is not a decimal digit, assume output is required and go to step 7.

2. Clear a double-word store on the stack and set a pointer to a table of alternate element limits (i.e. 24 hours, 60 minutes, 60 seconds) and corresponding multiplication factors.

3. Using the standard "Multiply previous result by 10 and add new digit" technique, convert consecutive decimal digits to binary (single-precision only needed), stopping at the first non-digit. If the resulting value exceeds the expected limit, recall the Special Driver to reject the command with "Faulty Syntax" print.

4. Use a non-0 input value as a counter while adding the current multiplication factor into the store on the stack (allowing for the 0 sign in the low-order word).

5. If the "Seconds" factor has just been used or the end of the argument has been reached, go to step 6 to store the result. Otherwise check the character which stopped the conversion in step 3. Treat space as the end of input and again go to step 6. For colon move to next table entry and repeat from step 3. In all other cases, reject the command for "Faulty Syntax".

6. Move the final result from the stack into TOD in the SVT and recall the Special Driver.

7. For print-out, collect the current TOD content and adjust it to remove the low-order sign. Set a po-

inter to a table of reduction factors for each decimal digit of the output.

8. Starting from the ASCII digit base(60), count the number of successful subtractions of the current table factor. Restore the remainder when negative and move to next factor.

9. Send the resulting character to the console printer. If the first digit of a pair, repeat from step 8. If the second but not that for the "Seconds" element, print colon and then repeat from step 8.

10. Otherwise restore the printer carriage with CR/LF and recall the Special Driver.

6.4.8 The MODIFY Command (KBT.MO)

The MODIFY command permits keyboard simulation of the Examine and Deposit panel switches for making minor changes to locations in normal memory at any time. At each request, the content of a specified address is output at the console printer. The user may then merely close the location by entry of CR alone or he can type in some new content first. Replacing the CR by LF results in the Automatic output of the content of the next location for similar action and thus provides for sequential examination and modification.

This of course is the same technique as that used in ODT-11R. However it is not intended that the MODIFY command be a substitute for that program in normal debugging; it is just a means for making known patches to an already tested program as a temporary measure, until the original source can be corrected and reassembled. The commands provisions do not therefore extend to operations upon the contents of Registers or the external page. For the latter case especially, the commands processor has no way of checking address-legality; it could only try to access a required location and trust that no hardware trap followed. As shown in Chapter 7, such trap would result in a call for Error Diagnostic printing which also needs the KSB; since this is not free and the MODIFY processing cannot be continued to make it so, the system would hang. Hence the command is rejected unless the address can be verified as being within normal bounds (1).

Command Format:

The MODIFY command must include an argument; thus its format initially is:

MO[DIFY] Octal address <CR>

where the address lies on a word-boundary within available memory (leading zeros not required).

General Description:

Provided that the address is valid as noted above, the MODIFY processor internally converts its octal ASCII digits into a binary value which is then used to set a pointer. Thereafter the operation is a dialogue consisting of one or several lines, for each of which the processor first prints the following, again using its own octal translator:

Address/Content:

----

1. The hang-up can nevertheless occur if, by the LF process, the user steps above the available memory limit. Owing to the limited capacity of the KSB, the legality check is made only on the first address opened. Hence in the Programmers Handbook, the user is advised to be careful.

and waits for the user reply (1). The characters entered by the operator are stored on top of the command in the Listeners buffer, and during this period, RUBOUT or CTRL/U are assumed to indicate erasure of previous input as in the original command. The input stops at the first CR or LF when valid octal input, if any, is used to reset the content of the address under the pointer. A LF entry continues the dialogue; CR causes an exit to the interrupted program by Special Driver recall.

An invalid address in the command or illegal characters in the octal content entered stop further action on the command with an appropriate message being printed again through the Special Driver.

Exit State:

As shown above, the eventual return to the program is always through the Special Driver; hence Register and stack state are covered by Section 6.2.5. On error exit, R1 may contain one of the following:

-3 = Illegal characters during input, so Faulty Syntax
-6 = Illegal address - not on word-bound in available core

Detailed Processing:

The sequence of operations in MODIFY processing is illustrated at Figure 6-11 and its outline is as follows:

1. Restore Registers passed by the Interpreter and clear its return parameters from the stack.

2. Convert any valid octal ASCII input into binary and store as a memory-access pointer. Reject the command through the Special Driver for "Faulty Syntax" for no argument, for one containing characters other than octal ASCII digits or for single-word overflow. Also exit, if the converted value is an "Illegal Address" as specified above.

----
1. The interrupt facility cannot of course be used since this requires the reloading of the Special Driver- an impossible operation while the MODIFY module is still required. The input is therefore controlled by flag testing and the character-handling processes are embedded.

3. Convert the pointer content to its corresponding ASCII digits and dispatch to the console printer followed by a /. Likewise, process the content of the address under the pointer, terminated by a :

4. Using the position of the command terminator in the Listeners command buffer, set pointers for the user reply. (There is still ample room since the buffer can accept a full line - see Section 6.3.1).

5. Wait for the input as noted under "General Description" and when available store the character, without parity (bit 7), in the byte at the start of thw new string (to utilize the same techniques as in the original driver (see Section 6.3.2))

6. If the entry is CTRL/U, echo ^U, and restart from step 4 (note no CR/LF in this case!)

7. Check whether a RUBOUT is part of a sequence (next input byte = 200 - see step 5 above). If not, echo as ◊ before printing the character deleted (if any remaining). Set the switch for the next input and repeat from step 5.

8. Store any other character in the input buffer (followed by 0 as a "no RUBOUT" flag). If the previous character was RUBOUT, echo 0; if the new one is not a terminator (=basically CR or LF) (1), merely echo it also and repeat from step 5.

9. For any terminator, print CR/LF. If any other input has occurred (including 0), convert valid octal ASCII digits into a binary value (2) and store this in the address under the pointer in place of the original content.

10. If LF was entered, increment the memory-access pointer and repeat from step 3. Otherwise exit through the Special Driver.

The method used to control the octal conversions deserves mention here. The sequence for any dialogue can be simply considered as:

1. Convert octal input to binary - initially the command argument

2. Convert binary address to octal and print

3. Convert binary content to octal and print

1, The check method used in fact allows any CTRL character with an ASCII value less than CR (015) to act as a terminator with all but LF stopping the MODIFY sequence. Hence CTRL/C entered at this time has no other effect: to issue a new command with a program running, a second CTRL/C is thus necessary.

----

2. A subroutine used in step 2 is also called here. This may have two consequences:

a. The error conditions leading to command rejection by Special Driver recall again have this effect.

b. Space is acceptable in lieu of CR at the end of the command argument to allow separation of comment preceded by ;. Since recognition of space therefore stops the conversion validly, it is possible to enter comment after replacement content, or instead of it. However, as implied at step 4, the user must take case as the buffer is now less than line-size and no checking occurs!

4. Return to 1 to convert new octal content and loop.

To save on the restricted capacity of the KSB, the two subroutines are in fact set up as a co-routine of the main sequence - using a similar technique to that described for the .READ/.WRITE processor in Section 3.2.2.2.

6.4.9 The ASSIGN Command (KBI.AS)

In Chapter 3, it was shown that the Monitor allows the user to leave until program run-time his final selection of the I/O devices and files to be associated with the datasets servicing the program. Section 3.1.2.2, in particular, described the table which can be set up within the Monitor area to contain the necessary assignment information used by the .INIT processor (see Section 3.2.1.1) and the File-management subroutine LUK (see Section 4.4.1) when preparing for I/O operations. The ASSIGN command is provided principally as the means whereby this table is established. In this respect, it can be entered at any time, though with different effect depending upon the system state. Moreover as noted in Section 3.1.2.2, it is of value only if the running program has not yet completed its data-set initialization. The command may also be used to clear outstanding assignments, but only if no program is loaded.

Command Format:

The general form of the command is as follows:

AS[SIGN] [Dataset Specifier, Logical Name]<CR>

where "Logical Name" is the three-character identifier for the relevant dataset as shown in the program Link-block (see Section 3.1.2.1) and "Dataset Specifier" may consist of Device name and unit #, Filename and extension and UIC (but no switches), exactly as defined for CSI (see Section 5.4) and obeying the same rules for element-omission. (1)

The presence of the argument string implies a new entry into the Device Assignment Table (DAT) and as indicated such string must include at least one acceptable element for data-set specifier and the logical name. No default assumptions are made; these are expected to come from the program itself.

----

1. Because of its delimiting function in the command language, "space" is of course not similarly ignored.

If no argument is given, the command signifies that any existing table is to be removed. This form is meaningful and is therefore acceptable only when no program is loaded. (see Section 3.1.2.2)

General Description:

As noted above, the ASSIGN command permits dataset specification in CSI format. However its processor cannot use CSI since there may be a running program already using the MSB. Therefore it must, as mentioned earlier, provide its own checking and decoding as well as making the appropriate DAT entry. This operation needs three overlays:

1. Syntax Analyser - scans the argument string for accuracy in accordance with CSI rules and prepares a table on top of the stack which lists the string start addresses for each of the potential elements. At the same time, valid characters for elements to be packed in radix-50 format are replaced in the string by their corresponding conversion codes (see Section 5.3).

2. Decoder - translates each string element into the DAT format using its own radix-50 and octal ASCII conversion routines. The results are stored in the table on the stack instead of the start addresses.

3. DAT-entry Routine - reduces the table on the stack to its DAT entry form as illustrated in Figure 3-3 and adds it to any existing DAT (removing any other entry of the same logical name) or initiates a new table if none presently.

If no argument is entered, the first overlay passes control immediately to the third and provided that no program exists, any current DAT is removed by the return of its area to free core.

Exit State:

In all cases, control is returned to the interrupted state by Special Driver recall, thus again Registers and stack are as described in Section 6.2.5. Errors in the input argument as usual cause rejection of the command with the appropriate message, for which R1 is set as follows:

- -2 = Program is loaded so null-argument command Invalid.
- -3 = Faulty Syntax (illegal characters or elements)

-7 = No buffer space for a new assignment while a program is underway.

Detailed Processing:

The overall sequence of operations performed by the ASSIGN command is illustrated at Figure 6-12. As indicated under "General Description", this is covered by three overlays which are individually discussed below. In each case, the first task is the restoration of Register contents saved by the call and the removal of the call return parameters from the stack.

The Syntax Analyser Overlay is loaded by the Interpreter call. It takes the following steps:

1. Clear a table on top of the stack for use as depicted in Figure 6-13. If the command has no argument, exit to call overlay #3.

2. Scan the argument for : which can only mean device specification in this case (no switches, hence no values). If none is found, go to step 3 to look for Filename or UIC. Otherwise the first character of the argument must be a letter, since device name, by definition, is alphabetic only. Provided that such is the case, store the start address of the argument in the stack-table slot for "Device" and translate up to three letters into their radix-50 code within the command buffer. Assume that the next character not being the : signifies the start of a unit number and store its address in the stack-table "Unit." if the character after the : is space or comma go to step 5 to check for logical name.

3. For a non-alphabetic next character - or first if no device - go to step 4. Store the address of a letter as the start of "File-name" in the stack-table and again translate it and up to five subsequent letters or digits into radix-50 code in the buffer. If the next character is then ., repeat the process for up to three letters or digits more, storing their start as "Extension". on recognition of space or comma at any point, go to step 5 to handle logical name.

4. Save the address of a [as the start of a UIC and scan for] when found go to step 5, provided that the next character is space or comma.

5. Store the address of the first character after a space or comma in the stack-table as the start of "Logical Name" and translate the up to 3 characters into their Radix-50 format as previously, provided that they are letters or digits. Call overlay #2 from the next block on the system-device when complete.

6. If one of the following conditions is encountered during the processing so far described, reject the command for "Faulty Syntax" through an immediate Special Driver recall (with the pointer table scrapped):

   a. An argument beginning with a character other than a letter or [(including space or comma since there must be at least one dataset element).

   b. An argument terminating before a space or comma followed by at least one letter or digit of logical name.

   c. Excess characters in Filename or Extension.

   d. [not followed by].

At the exit to overlay #2, the table on the stack now contains the addresses in the buffer storing the elements specified in the input. Omitted elements are identified by a slot still set at the original 0 from step 1 in the previous paragraph. The overlay is now responsible for coding the elements as follows:

1. Set pointers to the start of the table on the stack and to a corresponding table of conversion sub-routine despatch addresses (actually relative off-sets to allow for PIC). At the same time, save the end address of the valid input string to delimit logical name - see below, next step.

2. Collect the element start address. If 0, bump the table pointers and try again. Otherwise scan up the stack table for the next non-0 entry to fix an end point for the current conversion (allowing for an expected terminating character, i.e. : after device, ., after filename etc.)

3. Set a standard count for a three-character conversion and go to the appropriate sub-routine as follows, returning when either the end point is reached or the count runs out (1):

   a. Device name - adjust the end point depending upon the next included element (since no punctuation mark occurs between device name and unit whereas there are two between device and UIC (:[) or logical name (:,). Pack three characters (or actual number with trailing spaces to force left justification if need be) into a Radix-50 word and set into the stack table.

   b. Device unit - also allow double punctuation if no file-entry. Convert up to three valid octal digits into a single byte and store. If any other character occurs before the end-point is reached or if the byte overflows, clear the sub-routine address table pointer as an error flag.

   c. File name (word 1) - radix-50 pack three characters (again with forced trailing spaces) and store in the stack table. Save the start address of the next three characters in the second Filename slot in the table.

   d. File name (word 2) - similarly perform radix-50 pack and store.

   e. Extension - likewise.

   f. UIC - adjust the end point for double punctuation (],) and convert up to three valid octal digits into a single byte as Group Identifier (see Section 4.1.4). Provided that the next character is a comma, repeat for User Identifier. In addition to errors noted under "Unit," set the flag for a non-comma separator or if either conversion produces 0 or 377 (the latter being reserved for * under CSI).

4. If the subroutine address table pointer is still set upon return from a conversion, repeat from step

----

1. Each subroutine in fact calls one of two common conversion routines, one to pack characters in radix-50 format, the other to form a binary word from octal ASCII digits. These in their turn are co-routines with a single byte processor in much the same way as that described for the Conversion Utilities Package - see Section 5.3.

2. Otherwise scrub the stack-table and recall the Special Driver to reject the command for "Faulty Syntax".

5. On completion of all the routines in step 3, radix-50 pack the logical name (also left justified) and call overlay #3.

Thus on exit to overlay #3, the stack table now contains the actual entries for the DAT proper. However there may still be some original 0-slots for items unspecified. As shown in the DAT illustration at Figure 3-3, the final form may be contracted. This is done on the stack before the move into the DAT as follows:

1. Set up relevant pointers. If the stack-top (Logical Name slot) is 0 - impossible unless there is no argument at all - go to step 10 to remove the DAT.

2. Set a full count (4) in the "# of words to follow" byte of the "Unit" word. Decrement the count and move the Logical Name entry down the stack until the first non-0 table entry is seen, stopping at "Unit".

3. Collect the content of BAT in the SVT (the DAT pointer - see Section 2.1.1). If it is non-0, a table already exists, hence go to step 5 to find its end. Otherwise check the MUS in the SVT. If its high byte is 1 (program loaded and running) or -1 (program loaded and started but now suspended), go to step 8 as a buffer must be claimed from free core (see case 3 in Section 3.1.2.2).

4. With no program underway, collect EOM from the SVT and adjust for the two reserved DAT link words - see Sections 2.1.4 and 5.1. Leave one as a switch however if a program is loaded without additional Monitor routines (see case 2 in Section 3.1.2.2). Store the result in BAT and go to step 9 to set up the entry.

5. Search for the end of an existing DAT as follows (see also Figure 3-2):

   a. If the first word of an entry is -1, identifying the end of a DAT segment, collect the next word as the start of a new segment. Try again, unless his next word is 0 meaning the end is now reached.

   b. Check whether the dataset named in the command is the same as that for the entry. If so, re-

place the entry by a link to the next ("-1, address" sequence) - thus erasing the entry, in case the new specification is larger than the original.

c. Repeat from (a) for the next entry if this does not start with 0, which also signifies the DAT end.

6. If the DAT end is a -1,0 sequence (caused by a BEGIN unlinking assignments in free core buffers - see Section 6.4.5) remove the -1.

7. Check the MUS for a program underway as in step 3. If none, go to step 9 to transfer the new entry, providing a link to the current EOM if this does not coincide with the DAT end (owing to the fact that the previous entries were made before a program was loaded and now additional Monitor routines have been made resident).

8. When a program is underway, request a 16-word buffer unit from free core through the Monitor S.GTB routine (see Section 2.4.2). For a first assignment, store the buffer start address at BAT; for any other, link the buffer to the current DAT end. (If no buffer is available, however, recall the Special Driver to tell the operator.)

9. Transfer the Logical Name from the new specification into the DAT followed by the other items and two reserved Link-words of 0. Adjust the EOM, TOB and its stack-stop (see Section 2.4.1), if no program is underway. Exit through the Special Driver.

10. Accept a command without argument only when no program is loaded. In this case, save the DAT start as the new EOM, clear BAT and exit through step 7 to reserve the 2 links and adjust the end of Monitor pointers. Reject the command as invalid through the Special Driver otherwise.

6.4.10 The DUMP Command (KBI.DU)

The DUMP command was originally intended to provide a means for transferring core images to or from any appropriate peripheral device at any time, regardless of the present system state - as opposed to the load images obtained through SAVE with the system necessarily idle. However for the following reasons, its current implementation is limited:

1. Paper-tape requires some form of formatting to ensure data-accuracy. Thus a DUMP in this case effectively copies the SAVE facility supplemented by RUN or GET and is not therefore considered entirely worthwhile merely to allow occasional usage while a program is running.

2. Normal usage of bulk-storage devices requires the help of the File-management routines which can only be guaranteed if no program is underway to need the MSB - hence the restriction on SAVE (see Section 6.4.2). The only safe alternative is to have a pre-determined area reserved on each device which can then be accessed by simulation of .TRAN. This, though, raises two problems which are presently deemed to be too restrictive for the majority of users.

   a. The area reserved is lost to the user even though he never requests a DUMP.

   b. Any dump destroys the previous content of the reserved area and the user must therefore save this each time if it is still of value.

Hence for these devices, SAVE remains the only general provision. On the other hand, SAVE to a Line-printer is both meaningless and illegal since binary operations are not permitted. Dump is therefore provided in its case only.

Command Format:

The full format allowed by the DUMP command is as follows:

```
DU[MP] [Device][,[I OR O][,[LOW][,HIGH]]]
```

Where:

```
device = Standard name code and unit
         (: unnecessary)
I or O = Dump direction (assumed O)
LOW    = Dump area start address (assumed 0)
HIGH   = End address (top of core assumed)
```

(LOW and HIGH being octal and on word bounds within available memory).

It follows from the introduction, of course, that device can only be LP and direction O. However in case it is later included, the default device is that for the system.

General Description:

Despite the current restriction, the DUMP command processor is designed for generality of device. In order to allow usage at any time, it cannot assume that a necessary driver is in core to service the device. Even if it is, it need not be readily available for the DUMP operation. Hence the routine must have its own drivers for the purpose. In the interest of possible expansion, the DUMP module is therefore organized as a common argument decoder which performs any checking required and conversion of the various items in the input and then calls an appropriate overlay containing the requisite driver.

Currently two overlays are provided. The first is entirely taken up by a driver for the Line-printer and contains the processing needed to produce a print-out as shown at Figure 6-14. This gives both the content of each word in the DUMP area in octal ASCII and that of each byte as an ASCII character, when modified as necessary to produce a printable character in the 6-bit range. It also accepts a CTRL/C input at the keyboard as a signal that a DUMP underway should be terminated at the end of a current page. The second overlay provides the hooks for the later inclusion of other drivers but presently recalls the Special Driver immediately to reject the command for "Illegal Device."

Exit State:

The DUMP processor always returns to the interrupted program through the Special Driver, so the Register and stack state are as described under Section 6.2.5. Command rejection as usual follows errors, indicated in R1 as follows:

- -3 = Faulty Syntax
- -4 = Device cannot be used for DUMP, so is Illegal
- -6 = An Illegal address Specified is not on a word-boundard within available core or low is not less than high.

Detailed Processing:

Figure 6-15 illustrates the sequence of operations in the DUMP processor. As noted under "General Processing", overlaying occurs. The segments concerned are outlined in the following paragraphs.

The segment called by the Interpreter decodes the argument and then calls the appropriate driver overlay as under:

1. Restore Register contents passed by the Interpreter and clear its return parameters. Prepare a four-word table on the stack for use as element storage after conversion:

```
@SP:  END ADDRESS
      START ADDRESS
      DEVICE NAME
      DEVICE UNIT/TRANSFER DIRECTION (0=O;1=I)
```

2. Radix-50 pack leading letters from the argument string until stopped either by a defined delimiter (space, comma or CR) or by a non-alphabetic value. Store the result in the stack-table "Device" slot.

3. If the radix conversion ends with a non-alphabetic character, allow for a single octal ASCII digit as unit (DUMPing is never expected to be a facility on devices having more than eight units to a single controller). Store on the stack if found. Ignore : but reject the command for "Faulty Syntax" by Special Driver recall for any other character(s) before the first delimiter.

4. Accept one character before the second delimiter. If it is I, increment the 0 "Transfer Direction" byte in the stack-table. (No increment occurs for any other input, thus forcing O). Again reject as "Faulty Syntax" for more than one character.

5. Convert valid octal ASCII digits up to the next delimiter and store as LOW limit, provided that no overflow of a single binary word occurs. Then repeat for HIGH limit. (In this case "Faulty Syntax" means invalid input or overflow).

6. Check that both limits are on word-boundaries within available memory and that the LOW value is less than HIGH (replaced by top of core if none entered). This time reject the command for address illegality.

7. Replace the HIGH limit by the appropriate number of words to be DUMPed (positive value) and check whether a device has been specified. If none, replace with the name of the system-device as shown in the DDL (Section 2.1.3).

8. Search a table of potential devices for the DUMP and call overlay #2 if LP is required or #3 for other possibly valid specifications. Reject the command for "Illegal Device" otherwise.

If either of the overlays is called the Decoder exits with the stack-table cleared and its content set in Registers as follows:

R1 = Start of the DUMP area.
R2 = # of words to be dumped.
R3 = Positive code for a valid device, based on table-position (LP=0)
R5 = Unit and Transfer Direction.

Thus, the appropriate parameters for a DUMP are received by the overlay on the stack as saved by the EMT Handler (see Section 2.2.1).

The Line-printer dump has a basic format in which each page covers 1000(octal) locations divided into 8 blocks of 100(octal) locations. Each block is printed as four lines of 8 words in octal ASCII followed by 16 ASCII bytes. The overlay is set up to control this format as follows:

1. Restore Register contents passed by the decoder and remove its return parameters from the stack. If Output is not the direction specified, recall the Special Driver immediately to reject the command for "Illegal Device."

2. Save the current value of the stack pointer in the SVT word reserved for the purpose (SVT+52 - see Section 2.1.1) so that it also may appear on the DUMP if required. (1)

3. Save the DUMP area start specified and adjust an access pointer to commence printing from the top of a full page, i.e. from the previous location ending in ...000. Print FF to start a new page.

4. Print the address of the location starting a block (always ends ....00) followed by LF.

5. Start a detail line with the last two digits of the first address on the line (e.g. 00, 20, 40, 60) preceded by space and followed by :.

----
1. Since the program Registers are saved on the stack at this point, these can also be examined on an appropriate DUMP. The saved SP points at the address of the console typewriter buffer (saved R4 for the Special Driver Recall) which is then followed by the return parameters for the recall - see Figure 6-2 for further detail.

6. If the access pointer has not yet reached the saved actual start, print seven spaces. Otherwise output in octal ASCII the content of the location under the pointer (6 digits) preceded by one space. Repeat for eight locations in all unless the Word Count specified runs out. In this case again pad the rest of the line with spaces.

7. Print a terminal : and space. Reset the access pointer to the beginning of the line and set another to the last valid byte to be printed. As with words in step 6, print spaces for each byte in the line prior to the actual DUMP area. Otherwise convert each byte stripped of bit 7 into a 6-bit ASCII value on the following basis and print until the line-end.

   7 Bit value: 000-037 040-137 140-177
   6 Bit value: 100-137 040-137 040-077

8. If the end of the DUMP is not yet reached, print LF. Return to step 5 unless the access pointer ends in ....00 indicating the end of a block. In this case, print a second LF to give a gap between blocks and return to step 4 if the block does not also end a page (pointer=....00). For the latter situation, replace the second LF by FF and return to step 4 provided that no CTRL/C has been struck at the Keyboard.

9. Terminate the DUMP listing with FF and return to the interrupted program through the Special Driver.

It should be noted that any hardware failure at the printer which causes an error report will stop the DUMP with an "Illegal Device" message. If the printer is not a valid device on the system, by normal hardware trapping a fatal error message will be demanded and will result in hanging the system owing to the conflict in KSR usage, noted in Chapter 7. (Monitor reboot is necessary if this happens.)

As shown under "General Description", the second overlay at present merely cleans up the stack and reports the device as illegal. However, should extra devices be added to the DUMP facility, it is envisaged that the following provisions may be useful:

   a. The driver required can be identified by the code in R3 on entry (after Register restoration) and the validity of direction can be checked in R5.

b. By saving SP in the SVT on output, and restoring it on input a program might be automatically resumed from a point of dumping by the normal Keyboard Command exit process (provided of course that enough core is DUMPed.)

c. The driver itself must provide any necessary formatting of the DUMP.

6.5 Between-program Services

The previous sections of this Chapter have described the methods used to execute Keyboard Commands which may be entered by the operator at times when a user program either can or must be loaded. It has been shown that in order to respect that programs potential usage of the system, the command processors are obliged to restrict their operations to the confines of the KSB, even providing their own utility functions rather than use those already contained elsewhere in the Monitor. However it seems unreasonable that the same limitations should also be extended to the handling of commands which by their nature imply that there can be no program in memory when they are given and hence the whole of the system is solely at the Monitors disposal. For this situation therefore a special transient Monitor section becomes the "user program" - able to use all the normal facilities available but also privileged in its access.

This transient Monitor is the subject of this Section. In the first place, Section 6.5.1 discusses its responsibilities to the system and describes its general structure to satisfy these. Section 6.5.2 then examines the commands which are processed under its aegis.

6.5.1 The Transient Monitor (TMON)

As noted in Sections 2.1.4 and 5.5, the transient Monitor section is brought into core immediately after the initialization process following a Monitor boot or as a result of the unloading of a program through a .EXIT or console KILL. It is then responsible for satisfying the following functions:

1. Clean-up after the vacating program, if any

2. Preparation for the loading of the next program

3. Acceptance of any operator instructions from the console

4. Actual processing of commands restricted to the unloaded-memory situation and not therefore already covered by the normal language (currently: LOGIN, RUN/GET, FINISH)

Calling Sequence:

```
        .EXIT           ;EMT+60

                  OR

        ^C KILL
```

Section 5.5 showed that TMON is read into the top of available memory by the .EXIT processor, directly accessing the Monitor Library on the system-device, and is started automatically on successful loading. At this time, the stack is cleared to its load point and the only relevant register content is:

R5 = Address of the start of the Monitor DDB chain (DCO in the SVT - see Section 2.1.1)

General Description:

As indicated in the Programmers Handbook, neither .EXIT nor a console KILL require that a running program should be detached from any existing I/O commitment. It is therefore the Monitors task to ensure that, in general, any outstanding output still remaining in its internal buffers is dispatched to the appropriate device and, in particular, any files still open are closed. This cannot be done by XIT since then the MSB is not available for the use of the I/O processors; hence this is the first operation carried out by TMON, using its own Link-block associated successively with any DDBs still extant in the Monitor chain.

The unloaded program, of course, disappears under TMON. However it may leave temporarily resident Monitor modules behind or perhaps device assignments made strictly on its behalf (see cases 2 and 3 in Section 3.1.2.2). Also the evidence of its presence certainly still exists within the SVT. TMON therefore removes these relics especially:

a. The DAT is cleared of all but the pre-load entries, if any

b. The MRT is restored to its permanent state by a .TRAN of the copy stored on the system-device.

c. Relevant entries in the SVT and the DDL are restored to their pre-load state

e. EOM and TOB in the SVT (and the stack-stop - see Section 2.4.1) are set to reflect the new situation.

This last operation poses a problem. It will be shown shortly that TMON performs its I/O just like any other user program - implying the use of free core as buffers. By the normal process, these are allocated space immediately above EOM. However the operator may at this point enter some new device assignments which can change EOM (and overwrite the first buffers) - see Section 6.4.9. TMON, as a result, manipulates EOM. While its datasets are now set-up or later cleared, an area of free core is added to the true Monitor top as a potential DAT Extension. EOM then being set above this area, the buffers are allocated or released out of harms way. For normal running, EOM reverts to the true top, thus forcing any assignment entries into the correct place. (see Figure 6-15)

The I/O generally performed by TMON is of course concerned again with the acceptance of keyboard commands and possible printer responses (in particular $ since now the Monitor is always in a listening state). As a user program, TMON can take advantage of the facilities of the full console driver and use .READ or .WRITE - a particularly significant practice since it then allows normal device-independence and hence gives DOS initial Batch-processing capabilities (1).

----

1. TMON already contains odd references to hooks provided for the originally intended implementation of an OTHER command for a simple form of Batch - but currently held in abeyance pending the possible provision of a fuller system. Nevertheless TMON can still be used in a very basic way (on a non-file structured device at least) merely by a console ASSIGN of CMD. However two points should then be noted:

a. By the standard buffering scheme, after a RUN or GET from the new device, valid data could remain in TMONs buffer and be lost with it unless appropriately spaced.

b. CTRL/C must only be used when TMON is not in control (for the reasons given above).

c. Re-assignment of KB to CMD must precede the loading of the last batched program to force TMON to return correctly set-up - it is too late when already in and initialized.

Again though there is a problem. All command must now come through TMON so that its own particular entries can be trapped. This automatically occurs as long as the operator merely types in the command itself in response to the $ output. However for compatibility with the remaining commands, he should be able to enter CTRL/C, either initially or to cancel current input. As shown in Section 6.3.1, this would in the normal way be intercepted by the Listener and the whole command would be handled by the Interpreter, causing rejection of the TMON commands since they are unknown except at this time. To prevent this, therefore, TMON first initializes the appropriate datasets to bring the full driver into core. It then unlinks the Listener from the keyboard interrupt vector, substituting its own version providing direct access to the full drivers interrupt servicing routine and specifically handling CTRL/C in its own way (1).

Thus all commands initially come to TMON. By a table look-up, those which are especially appropriate are passed to an embedded processor for execution as detailed in the next section. The remainder are passed through the normal operations described in Section 6.4. For this purpose, TMON simulates an interface to the Interpreter as shown in Section 6.3.2, having saved its own Registers and raised the priority level to 4 as in the general case. This means that the command is accessed from the TMON buffer and that on completion, control is returned through the Special Driver with the printer iterrupt enabled. (again see Section 6.3.2). To allow the Listener to perform its normal pointer re-initialization, TMON simulates the Listeners pointer save operation (see Section 6.3.1) and lets the interrupt through (since the printer is still linked). When cleared, TMON requests and awaits its next command.

The handling of errors detected by the special processors is similarly accomplished by a simulated recall to the Special Driver as described in Section 6.2.4 with the interface set up as shown in Section 6.2.5. This permits utilization of the drivers error messages. Follow-up action is exactly the same as in the last paragraph - hence the original command is ignored as usual. TMON eventually leaves after acceptance of a valid RUN and GET (see next section) or is reloaded afresh after FINISH.

----

1. CTRL/C is in fact converted to VT, since this as a standard delimiter forces input termination but stays on the same printer line. TMON can then simulate the normal echo as:

^C<CR>
. <VT>

Exit State:

When TMON passes control to one of its embedded processors, the relevant register contents are as follows:

R0 = Command output Link-block address
R1 = End address of the command input
R2 = Start address of the processor called
R3 = Command input Link-block address
R4 = Address of OSW in the SVT
(see Section 2.1.1)
R5 = Start address of the command argument

In addition the stack-state is:

@SP: (Scratch pad)
Start address of the command input
Listener keyboard vector link
0 (1)
Current buffer base = dummy eom.

As noted under "General Description", the Register and stack state when the Interpreter is called complies with the requirements of that routine as detailed in Section 6.3.2.

Detailed Processing:

Figure 6-16 illustrates the general processing performed by TMON. Its outline is as follows:

1. Using the Monitor DDB chain, successively link each extant DDB to an interal Link-block and examine its contents (see Section 3.1.2.3). If the associated driver is file-structured or is magnetic tape as shown by its Facilities Indicator (see Section 3.3.1), and an apparently valid file remains open for output, call .CLOSF and .RLSE. In all other cases, call .RLSE only. (As shown in Section 3.2.1.3, this forces last buffer output on a non-file (but no device close action, i.e. punch trailer. However to force acceptance on a file-device without the .CLOSE, the DDB "Buffer Address" must be cleared first).

2. When all DDBs have been actioned, for safety, clear any interrupt flags still enabled - though restart the clock if present (vector not pointing to error

----
1. This could be MRT information for a non-resident .READ/.WRITE processor see "Comments".

trap - see Chapter 7)

3. Trace the DAT, if any, and unlink any entries made after the last program was loaded (reset link words on completion):

    a. Whole table if it does not start immediately after the Buffer Allocation Table (see Section 3.1.2.2) - clear SVT pointer in this case

    b. From the end of the first segment otherwise.

4. Adjust EOM & TOB in the SVT (see Section 2.1.1) and the stack-stop, leaving DAT space as noted under "General Description".

5. Clear the Buffer Allocation Table completely (see Section 2.4.2). Remove MUS, PLA, PSA, DSA and Program Name from the SVT and all core connections to non-resident drivers from the DDL (see Sections 2.1.1 and 2.1.3).

6. Prepare a TRAN-block to read the MRT copy from block #5 on the system-device into the MRT area in core, using the SVT addresses for the MRT and DDL to determine its start address and size. Call .INIT and .TRAN on a dataset provided to effect the transfer.

7. Initialize a dataset to the console printer and output CR/LF to ensure carriage-restoration. Save the current content of the console keyboard vector and link to as internal minimal Listener, as noted under "General Description."

8. Call .INIT and .OPEN on dataset CMD, normally assigned to the keyboard. Restore EOM in the SVT to true Monitor top in case an ASSIGN command is entered. (see step 4)

9. Print $ and request .READ for the command input. This can return with a VT terminator representing CTRL/C (see "General Description"). Reproduce normal echo if so and try again.

10. Otherwise remove any comment preceded by ; from the input, and move CP up accordingly. Repeat from step 9 when no input remains.

11. Compare the first two characters of the actual command input with a table of those for the embedded processors and dispatch to the appropriate routine if found. (see Section 6.5.2)

12. For commands normally handled by KBI perform the simulated entry noted under "General Description" as follows:

    a. Through the saved keyboard vector content, get the address of the Listeners Pointer Store (see Section 6.3.1) and indicate "Command Underway" state in the start byte of the Echo Buffer, for eventual Listener exit.

    b. Save significant Register contents on the stack and reset as required by the Special Driver/Interpreter interface (see Section 6.3.2). Simulate the necessary stack-state by pushing a level 4 priority Status followed by the TMON recall address (done by JSR in fact) to equate to the normal Listener recall (see Figure 6-2).

    c. Set the priority level now to 4 and call the Interpreter (Section 6.3.3)

13. On return through the Special Driver, save the relevant pointers in the Listeners Pointer Store, drop the priority to level 0 and wait until the console printers interrupt is disabled - indicating that all command clean-up is now complete. Return to step 9 for a new input.

Comments:

Doubtless the reader will note that two features are included in TMON, as shown by its source listing, but are not covered here. The first of these, the OTHER hooks, was mentioned briefly in a "General Description" footnote and, as indicated, is presently irrelevant. The second is an embedded .READ/.WRITE processor. This also serves no purpose currently because, as shown in Section 3.2.2.2, this routine must be resident at all times. However, this may not always be the situation. Since TMON is not loaded as a user program, although it operates like one, it does not have the same opportunity to have a resident RWN so it provides its own and if necessary modifies the MRT to use it. Moreover the Loaders, described in Section 5.1 originally also used .READ rather that .TRAN - hence TMON, in the RUN/GET processor, moves its copy into a buffer for their access when required and in fact passes on data in R5 (the MRT information on RWN system-device location - or 0) to signal this. This data though is now ignored by the latest Loader versions. The processes remain in case they should be needed for later developments.

6.5.2 The TMON Commands

This section discusses commands handled only by the transient Monitor in order to allow them to have full access to all the normal Monitor facilities because no user program can exist in core when they are entered. They are in fact illegal at any other time. The commands covered here are:

LOGIN
RUN/GET
FINISH

As shown in the previous section, they all receive control from the TMON general routine with Registers and stack appropriately set for their acccess to the command by which they are called. Like the other command processors described earlier, they are then responsible for decoding any argument entered and for checking its validity before performing the function required by the operator. On completion, they must ensure that the correct machine-state is passed to the next routine in control. If in fact this is again the TMON general processor, the contents of R0, R3, and R4 remain as on entry: the stack is cleared of the first two words, but the others stand.

The routine must also take care of any errors detected either in processing the argument or in execution of the command. For compatibility with the other processors, they use the Special Drivers facility for printing error messages as described in Section 6.3.2. They do this by simulating the recall discussed in Section 6.2.4 with a system-state conforming to the details given in Section 6.2.5. On return, they pass control back to the general processor to ignore the commands. Again Registers and stack are as shown above.

6.5.2.1 The LOGIN Command

In Chapter 4, it was shown that each DOS user can effectively reserve areas on bulk-media within the system for the storage of his own files of programs or data, with some measure of protection against other users. The LOGIN command is provided to enable him to identify himself to the system in order to access those files. Currently this is its only purpose: it is in fact uneccessary if no unique file-structured operations are to be performed (since any user can call programs stored in the system Library (user 1,1) or if the users identification is entered as part of the actual file-specification for each task. However its use is recommended as a standard practice for ease of operation now and in case in later DOS developments it becomes necessary to impose more restrictions upon unauthorized ac-

cess to the system generally.

Command Format:

The only argument currently required is the identification code for the user entering the system. Hence the command format is:

LO[GIN] Group,User <CR>

Where "Group" and "User" each consist of up to three octal ASCII digits in the range 1-376.

Not only is the LOGIN command illegal if a program is loaded; it is invalid if any other user is seen to be still occupying the system.

General Description:

Provided that the system is free to accept a new user, the LOGIN processor calls the Conversion Utilities package described in Section 5.3 to convert the two arguments into their equivalent binary bytes. The results are stored at UIC in the SVT (see Section 2.1.1). In order to warn the new user that Date and TOD may need updating, or for later reference, it then outputs the current content of the appropriate SVT stores at the console printer. This is accomplished by a normal .WRITE to the full driver to print the appropriate heading "DATE:" or "TIME:"; this is followed by a simulated call to the keyboard Interpreter with a supposed command input without arguments, thereby utilizing the corresponding processors conversion facilities to obtain the required format. In each case, the second call is made through the TMON general processors subroutine for the purpose, as discussed in the last section. On completion, the general processor is recalled for a new input.

Exit State:

As noted in the introduction to Section 6.5.2, the LOGIN command calls the Special Driver to handle its error messages. Registers and stack state for this comply with the requirements of Section 6.2.5. In particular, R1 may contain:

- -2 = Another user on the system, hence the command is Invalid.
- -3 = No argument or one containing illegal characters or codes outside the specified range 1-376, hence Faulty Syntax.

The simulated call to the Interpreter establishes the necessary state as detailed in Section 6.3.2. The remarks on general processor recall on completion contained in the introduction to Section 6.2.5 apply.

Detailed Processing:

The simple sequence followed by the LOGIN processor needs no illustration. It is outlined below:

1. If the current content of UIC in the SVT is non-0, indicating another user on the system, call the Special Driver to print an "Invalid Command" message; on return, clean-up the stack and return to the TMON general routine.

2. Otherwise call the Octal ASCII converter in the Monitor utilities package to convert "Group Code" and store in the SVT. Repeat for "User Code", ignoring the comma separation. If the argument end is reached before the two conversions have been completed or if either byte is 0 or 377 (or above), clear any value stored in the SVT and call the Special Driver to output "Faulty Syntax". Exit as in step 1.

3. Clean-up the stack and call .WRITE to the command output dataset to print "DATE:". On completion, move the first two letters of the message followed by CR into the command input buffer and call the DATE command processor (see Section 6.4.1) through the Interpreter.

4. Similarly print "TIME:" and call the TIME command processor (see Section 6.4.7). Return to the TMON general routine when done.

6.5.2.2 The FINISH Command

The FINISH command naturally complements LOGIN in that it removes the user identification from the system, thereby allowing a new user access. It also ensures that the memory side of the system is clean as possible for the new user by rebooting the Monitor afresh from the system-device. However it does not presently attempt to perform any restitution of peripheral devices, even though this is implied by the provision of the .KEEP I/O request for file-structured devices (see Section 4.6) and the relevant bit in the File Protection Code (see Section 4.1.4).

Command Format:

The FINISH command needs no argument. Its format is therefore simply:

FI[NISH] <CR>

Apart from the overall restriction that no program can be loaded, this command can be given at any time.

General Description:

The FINISH processor uses a similar technique to that described for LOGIN in the last section in order to print the time of final log-off for user reference. It then determines the system-device from the first entry in the DDL (see Section 2.1.3) through its SVT pointer; by table look-up, this is converted to the address of the appropriate hardware register as supplied to the ROM bootstrap (see "Starting Procedure" in the Programmers Handbook). The boot is actually accomplished through an embedded version of the ROM sequence.

As noted above, this completely refreshes the memory loading, in particular removing any currently logged-in users identification. However it also clears DATE and TOD. So that every user is not obliged to reset these, the FINISH processor does not call for the boot until it has saved the present entries out of harms way above the top of TMON (assumed XX7400 if correctly linked for XIT loading - see Sections 5,5 and 8.1.2). They are then restored by the Monitor initialization routine discussed in Section 2.1.4.

Exit State:

No register or stack state is relevant.

Detailed Processing:

The processing of a FINISH command is again quite straightforward as shown below - hence no illustration:

1. Call .WRITE to print "TIME:" upon the command output dataset. Move "TI <CR>" into the command input buffer and through the Interpreter call the TIME command processor (see Section 6.4.7) to output the TOD stored in the SVT.

2. Save the current SVT entries for DATE and TOD starting at location XX7400, using the content of

CSA (top of memory) in the SVT to compute this. (see Section 2.1.

3. Get the name of the system-device from the DDL and by table look-up determine the address of its hardware word count register.

4. Request RESET to clear all the hardware registers for the device; set its word-count to -64 and its Status to 5 (Read & Go for all disks) - this then reads 64 words from block 0 to memory 0.

5. Wait on the device done flag and repeat from step 4 if hardware errors are reported. If satisfactory, go to location 0 to execute the routine loaded - assumed the Monitor Loader (see Section 8.2.2.4)

6.5.3 The RUN & GET Commands

The RUN and GET commands perform the same basic function - that of initiating the loading of user programs. It was shown in Section 5.1 that the load proper is effected by two non-resident modules LDR and LD2 operating from within the restricted environment of the MSB. To simplify their task, all the preliminary work is carried out by the TMON routine which processes the two commands initially and which thus has more facility. This RUN/GET processor is therefore responsible for decoding the input specification and for ensuring that the program is available as specified. Further, it undertakes the reading of the first part of the program module and the extraction and storage of the general control information at its front. Hence the actual Loader need then be concerned merely with the processing of the data for the program itself.

Command Format:

The general format for both commands is the same, i.e.

```
RU[N]   Dataset Specifier <CR>
GE[T]   Dataset Specifier <CR>
```

Where "Dataset Specifier" complies with the requirements and rules for Command String Interpreter input - but without switches and with space, being a valid keyboard delimiter, not ignored (see Section 5.4), namely:

```
[Device:][Filename[.Extension][UIC]]
```

"Filename" must, of course, be present if the device is any form of bulk storage medium (including magnetic tape). Otherwise the following default conditions apply, as further explained in the next paragraph:

a. Device = the system-device

b. Extension = null or LDA

c. UIC = the logged-in user or the System [1,1].

General Description:

As noted above, one processor handles both RUN and GET. Its two entry points merely set a switch appropriately to show the Loader later whether or not to start the program automatically. CSI is first called to perform the input decoding operation. (The input buffer used by TMON is already set up to include the required workspace, CMDBUF - see Section 5.4). Provided that some argument has been entered, the RUN/GET processor first checks that the specified device exists within the system as shown by the DDL. It then uses the I/O call .LOOK to verify the existence of the file on true file-structured devices or the ability of a non-file device to provide input.

For the former case, the following search algorithm is used (1):

1. If no "UIC" is given, four attempts may be made to find the file as follows:

   a. "Filename.Extension" for the logged-in user

   b. "Filename.Extension" under the System [1,1]

   c. "Filename" replaced by "Filename.LDA" as for (a) and (b)

2. No substitution of "System" occurs if "UIC" is explicitly stipulated - hence only two trials occur:

----

1. This applies only for full file-structured devices: it does not currently include magnetic tape. In its case, the correct complete specification must be entered as no preliminary search is carried out - since this may involve an entire pass across the tape (see Section 4.7).

a. "Filename.Extension" for the specified user

b. "Filename.LDA" replacing "Filename" for the specified user.

If the file cannot be found or the device cannot support input, the command is rejected by an appropriate message through the Special Driver, as noted in the introduction to Section 6.5.2, and the TMON general routine is recalled.

Otherwise the RUN/GET processor closes and releases all the TMON datasets currently linked (EOM being set to ensure correct buffer-release - see see Section 6.5.1). With the system then restored to normal state, i.e. true EOM (since no more pre-load DAT entries can be made), keyboard Listener re-linked to the keyboard interrupt vector (also see Section 6.5.1) and the Monitor/User switch temporarily set to program run-state (to allow standard keyboard responses to load failure - now reported through EDP- see Chapter 7), RUN/GET re-initializes one dataset for the program module. It claims an appropriate buffer for the device by a direct call to the Monitor S.GTB subroutine, if the device is file-structured or via .OPEN otherwise

Using .TRAN, it then reads data from the program module into this buffer, until it has been able to extract the COMD block (see Section 5.1) and store relevant data from it in the SVT or on the stack. (in the case of codes identifying additional Monitor modules also to be loaded) Finally it establishes the interface required by the Loader and calls it by EMT 61. As noted in the previous paragraph, errors detected during this process are reported as fatal (F022 = module format incorrect - no COMD; F023 = program too large for the available core). The operator must then KILL the load to continue.

Exit State:

Registers and stack state required by the loader were detailed in the introduction to Section 5.1.

Similarly if the command is rejected through the Special Driver and TMON proper is recalled, conditions are as described in the general introduction to Section 6.5.2.

Detailed Description:

The RUN/GET processor is illustrated at Figure 6-16. Its basic outline follows:

1. Set a switch to indicate RUN/GET (1=the latter). Reset EOM to the bottom of buffers and relink the

Listener to the keyboard interrupt vector (see Section 6.5.1). Re-enable keyboard interrupts and reset MUS in the SVT to show program in and running [1,1] in case errors occur for EDP reporting (see Section 2.1.1 and Chapter 7).

2. Reshuffle the command string to leave only the argument, at the same time converting space to comma to trap syntax errors. Call CSX to analyse the argument syntax (see Section 5.4.1). Then set up a CSI-block on the stack and call CSM to prepare a Link-block and File-block based upon the argument. If errors are detected at either call or no argument is entered, go to step 7 to exit through the Special Driver for "Faulty Syntax".

3. Through its pointer in the SVT, search the DDL for the device specified (automatically becomes system-device through CSI if none is given). If non-existent, similarly exit for "Illegal Device". Otherwise call .INIT on the prepared dataset (1).

4. Call .LOOK to test the existence of the file or the validity of the device for input. (see Section 4.6.6). Go to step 8 if satisfactory.

5. Otherwise check whether a UIC specification was stored in the File-block by CSI (because specifically entered). If so, temporarily change the content of the UIC in the SVT to the same specification and call .LOOK again (in case the user is attempting to access someone elses file and the request is rejected because Read privileges are not allowed - see Section 4.1.4). If the second check is successful, determine the relationship of the logged-in user to the files owner from the UICs. Provided that the file protection returned by .LOOK indicates relevant run privileges, go to step 8. Otherwise check the file extension; if null, repeat both checks from step 4 with LDA as a default.

6. When no UIC is actually specified, recall .LOOK with the System UIC substituted. Go to step 8 if

----

1. .INIT, of course, also performs the device check (see Section 3.2.1.1). However this might then result in a system error (A003) - somewhat confusing to the user especially when he has no knowledge of the internal operations at this time. Hence RUN/GET forces its own diagnostic.

successful. Repeat both checks from step 4 with a null Extension replaced by LDA

7. Should all the checks fail, set up for "No File" print. Reverse the actions of step 1 to restore the system-state as required by TMON and call the Special Driver to action the message. Revert to the general routine on completion.

8. If the input is thus now acceptable, release all datasets currently set up (also .CLOSE required on command input - since originally opened). Reset EOM to its true position and also TOB and its stack-stop. (By the manipulation noted in Section 6.5.1, the first two are held above the dummy DAT and buffer area).(1)

9. Reinitialize a dataset for the load. Call .STAT (see Section 3.2.4.2) to determine the associated device characteristics. If the driver Facility Indicator shows a file-structured device, determine its standard buffer size (see Section 3.3.1) and call the Monitor S.GTB routine to provide the necessary buffer (see Section 2.4.2). Prepare a TRAN-block appropriately (see Section 3.2.1.2). Remember the file type as returned by .LOOK and whether the device is DECtape from the driver indicator. Set pointers to force a buffer-fill and go to step 11.

10. When the device is not file-structured, call .OPEN, mainly to utilize any driver facility for checking device-readiness (see Section 3.3) and also to get automatic buffer allocation and a first buffer fill (see Section 3.2.2.1). From the DDB data for the dataset, prepare a TRAN-block as above (see Section 3.1.2.3). When the .OPEN is complete, set pointers to the buffer, check for magnetic tape (also from the driver indicator). Since no buffer fill automatically occurs in its case (see Section 4.7) reset pointers to force this.

11. Process data from the buffer, a word at a time, as follows - see Section 5.1 for details of format for the COMD expected:

----

1. This step is followed on the listing by a possible RWN move - see "Comment" in Section 6.5.1.

a. Ignore leading nulls

b. Check the first non-0 word. If not 1, as required by formatted binary mode (see Section 3.2.2.2), call "read" error (F021) through EDP (see Chapter 7)

c. Store the Line Byte Count and ignore dummy Load Address

d. Check the next word. If it is not 3401 (COMD code 1 with 7 words following), call "Format Error" (F022)

e. Save the Program Load-point and check Size. If the ultimate end is above available memory, call "Too Large Error" (F023). Otherwise store the load-point at PLA in the SVT (see Section 2.1.1) and similarly the Program Start and ODT Start addresses at PSA and DSA. If RUN is requested, store PSA in the RUN/GET switch (see step 1).

f. If the relocation flag is set (currently not possible), call "Format Error" as above. Otherwise store "Program Name" in the SVT.

g. Prepare to store a list of the additional Monitor routines to be loaded on the stack. If the next word contains a byte of 2 with a non-0 high-byte, store the latter as a counter and move the following words onto the stack until the counter runs out.

h. If the next word is 0, the end at the COMD is reached. So save the total number of words moved on the stack for loader use and go to step 13.

i. At each of the above steps (b) through (h), accumulate a byte checksum and decrement the byte count for the line, saved at (c). If this goes 0, verify the checksum and call "Read Error" (F021) on failure. Otherwise repeat steps "a" through "c" and resume from "g".

12. Should the data in the buffer run out, call .TRAN and .WAIT for a refill (or fill if forced above), provided that no EOD was seen at the last transfer (this should never happen - if it does call "Format Error" (F022)). On completion, adjust the buffer end if not completely filled (EOD now set). Prepare the Device Block # in the TRAN-block for

next time, as follows and then resume from step 11:

a. Increment current block if the device is file-structured and the file itself is contiguous (see Section 4.1.1.2) - if the device is not file-structured, this also occurs but is irrelevant.

b. If the input is a linked file (see Section 4.1.1.3), extract the next link-word - set EOD for next time if it is 0.

c. If the link is negative and the device is DECtape, adjust for reverse tape-motion (see Section 4.2.3)

13. Ensure that the final COMD checksum is correct (F021 error if not). Then complete Register and stack state preparation required by the Loader. (Moving any EOD marker into the DDB for later reference) and call EMT 61.

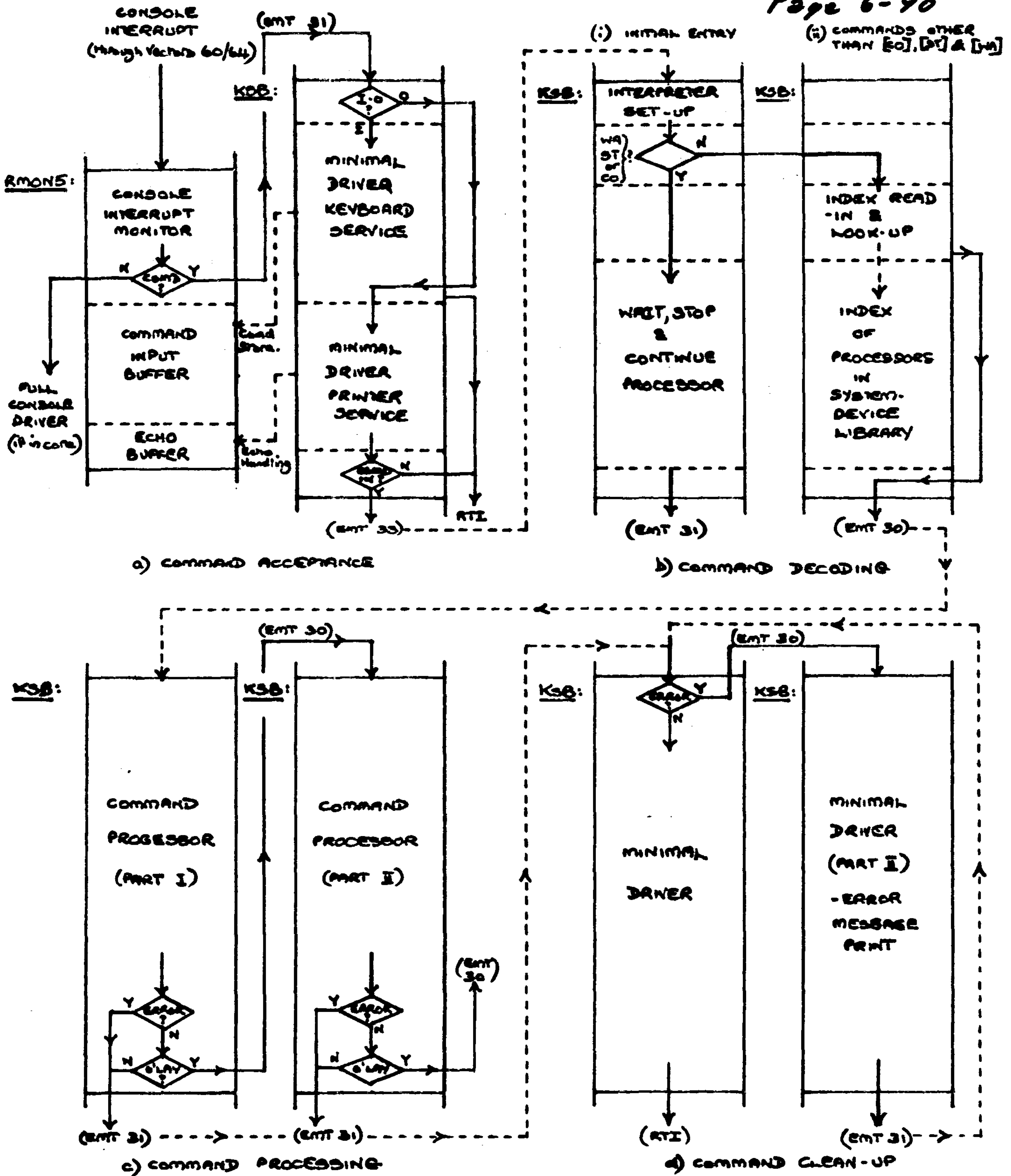
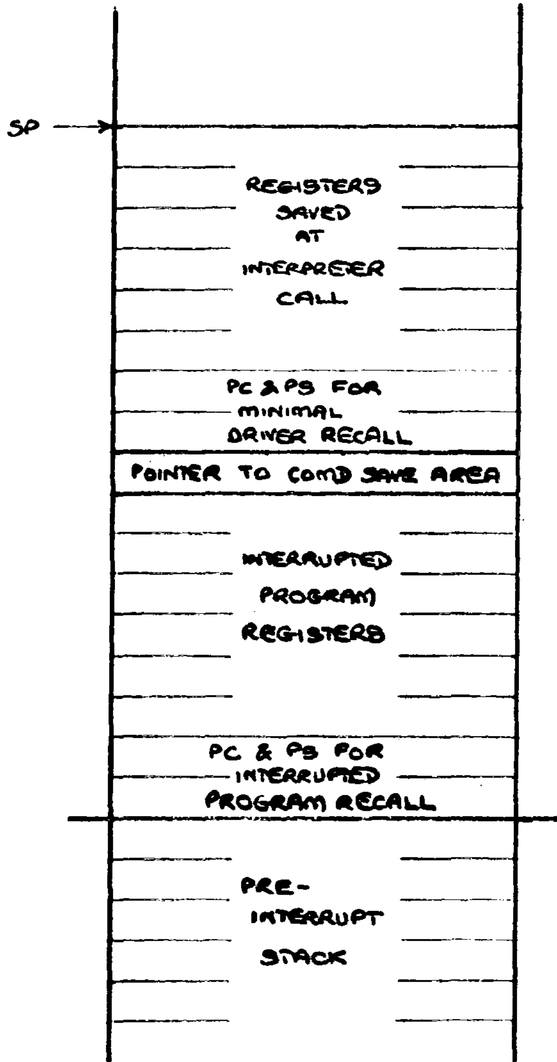


Figure 6.1: USE OF THE KEYBOARD SWAP BUFFER IN COMMAND PROCESSING



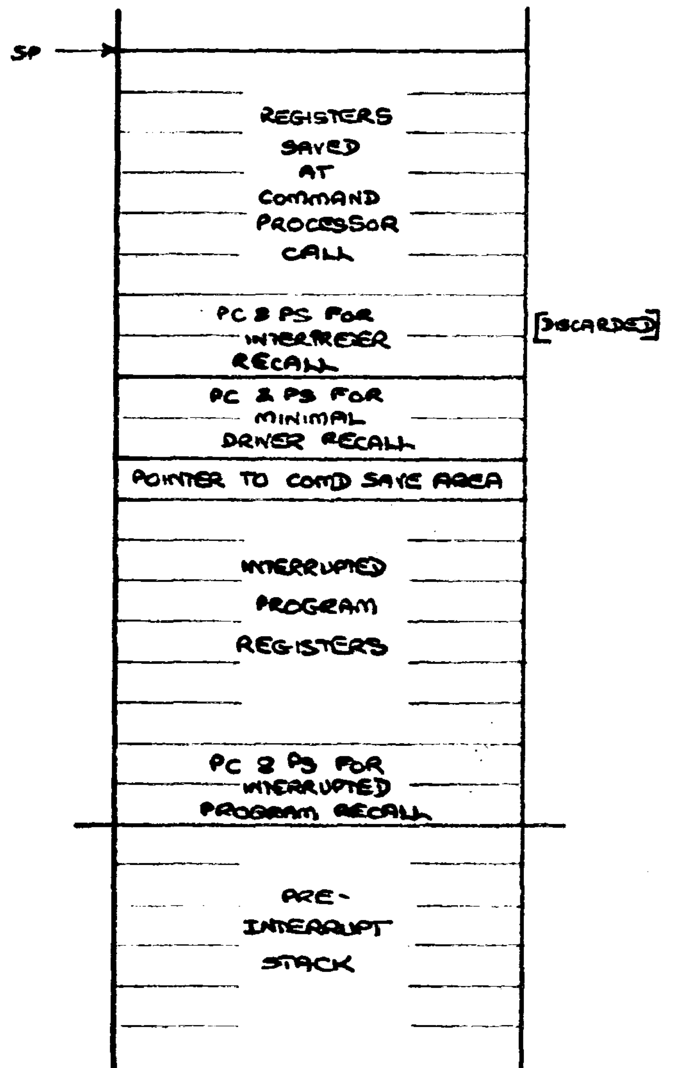


Figure 6-2: STACK STATES DURING COMMAND PROCESSING

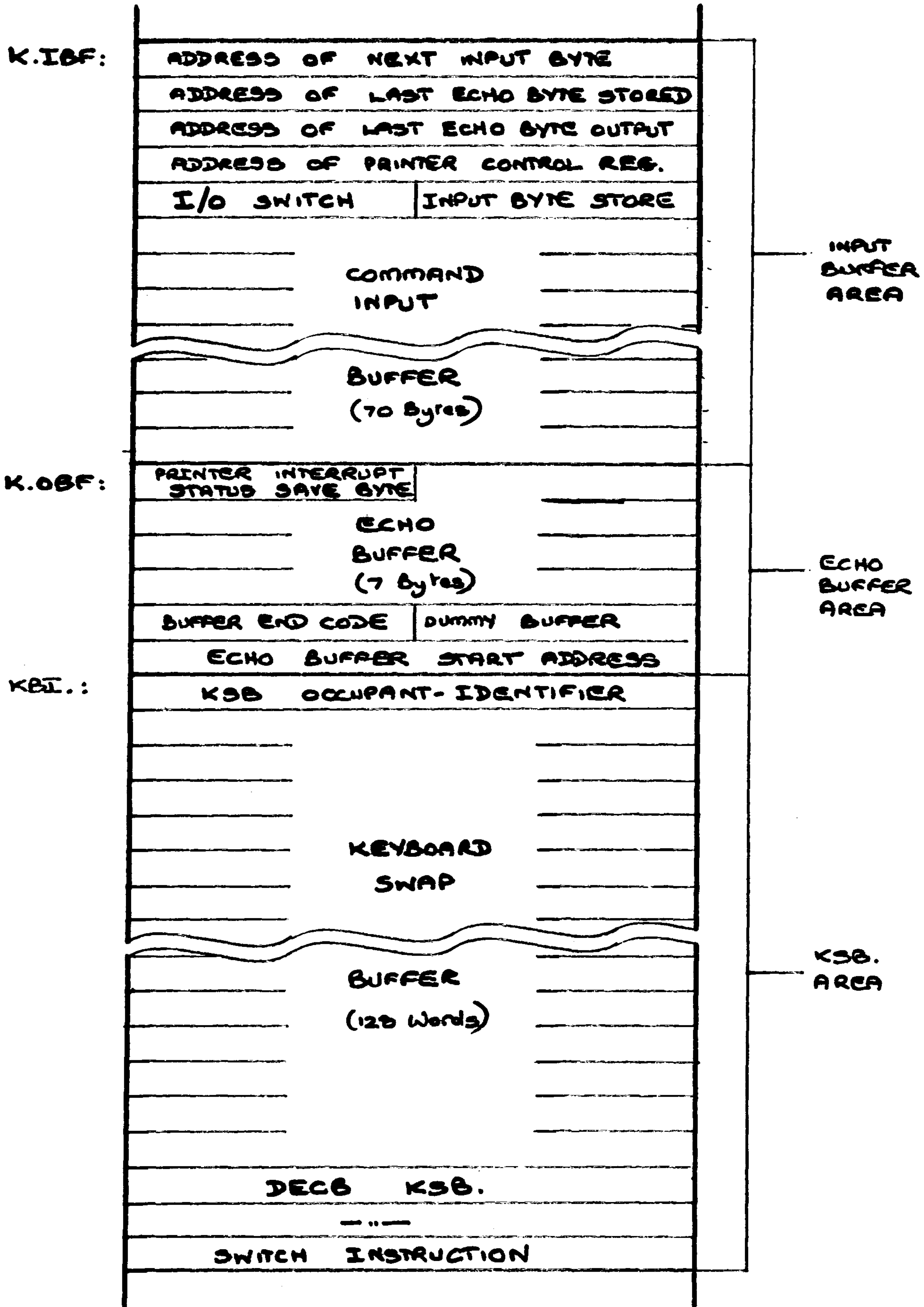


Figure 6-3: KEYBOARD COMMAND BUFFERS

KEYBOARD LISTENER OPERATIONS

[RMON5]

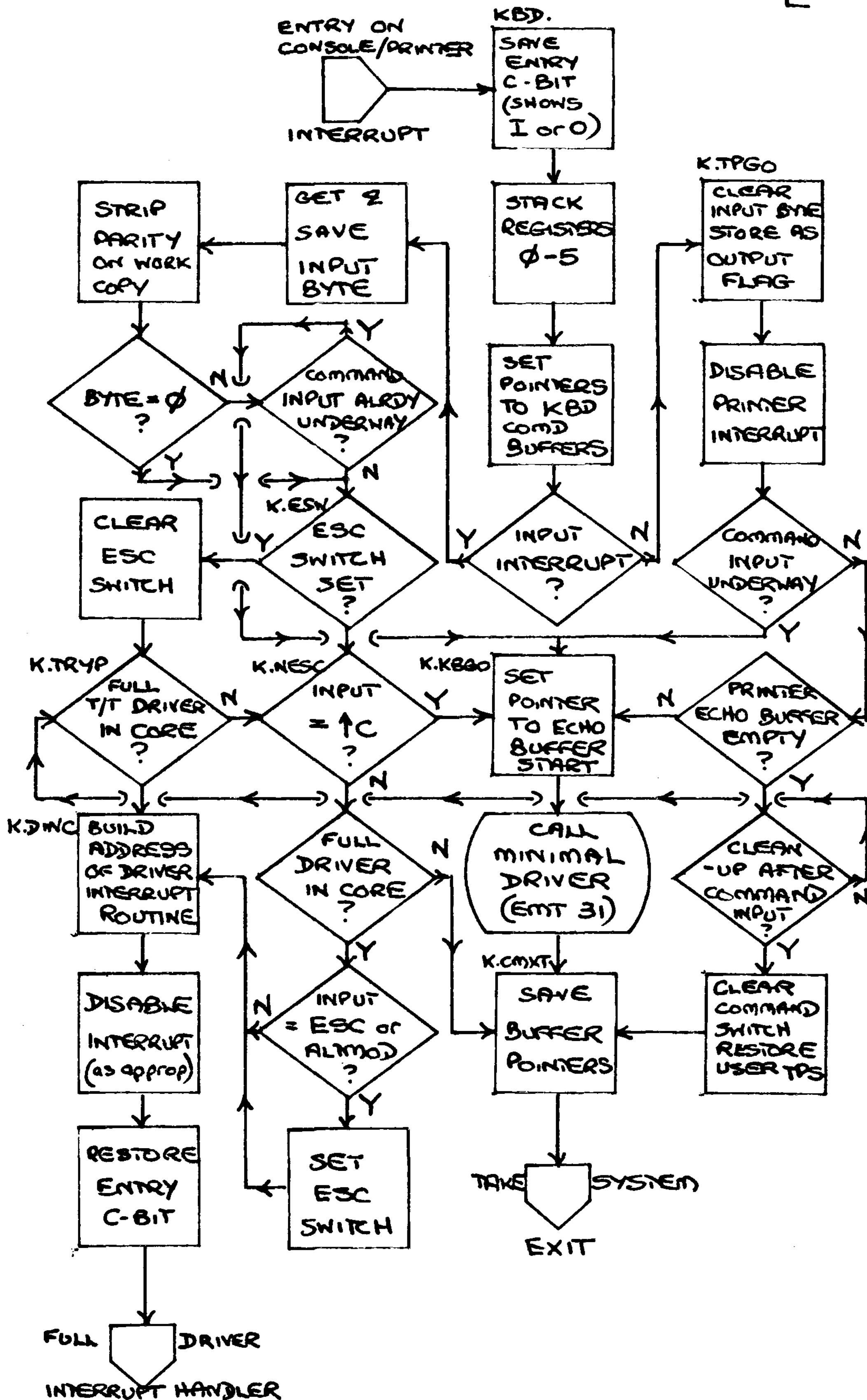


Figure 6-4

SPECIAL KEYBOARD DRIVER OPERATIONS

ENTRY VIA EMT 31 HANDLER

KBL

RESTORE STACKED REGISTERS

R5 = Ø ?

Y

N

IGNORE RETURN ADDRESS ON STACK

RECALL KBL (R1 NOW POSITIVE)

IGNORE RETURN ADDRESS ON STACK

KB.EPC

PRINT ERROR MESSAGE

KB.GO

SET POINTER AT KBD HARDWARE STATUS

N

INPUT BYTE STORE =Ø (I.E. OUT)

Y

IN CASE ERROR, SET UP OVERLAY CALL

CALL OVERLAY

KBL.EB

RESTORE REGISTERS FROM CALL

KB.EIG

USING R1, INDEX TABLE OF MESSAGES

INPUT = CTL/C OR CTL/U ?

N

KB.TRO

IF RUBOUT & NO R/O SW OR R/O SW & NOT R/O

N

COMPUTE DISK ADDRESS OF OVERLAY

ERROR IN FACT? (R1 REG) ?

Y

SET PTRS TO DUMMY ECHO BUFF (IN FRONT OF KSB)

KB.CC

Y

STORE '↑' AND 'C'/'U' IN ECHO BUFFER

Y

MOVE '\' INTO ECHO BUFFER

KB.PR

LAST BYTE PRINTED = ⤶ ?

Y

PREPARE INTERPRETER CALL. OUTPUT LINEFEED

STORE '.' IN ECHO BUFFER

KB.OXT

INPUT COMMAND COMPLETE ?

N

Y

N

SET/RESET POINTER TO START OF INPUT BUFFER

INPUT = ⤶ ?

N

INPUT = RUBOUT ?

Y

N

INPUT COMMAND UNDERWAY ?

Y

REPLACE '.' BY '$' IN ECHO BUFFER INIT. IN-POINTER

IF FIRST TIME IN, SAVE TPS FOR USER

KB.CRN

Y

STORE ';' IN INPUT BUFFER

N

MORE ROOM IN INPUT BUFF ?

N

KB.RO

MOVE LAST INPUT TO ECHO BUFF (IF ANY)

KB.MOR

N

MORE ECHO TO BE DONE

Y

Y

EXIT TO SYSTEM WAIT ?

N

KB.LD

REPLACE INPUT BY ⤶ · DISABLE KBD INTS.

SCAN INPUT FOR FIRST ';' TO REMOVE COMMENT

Y

STORE INPUT IN INPUT BUFFER

SET RUBOUT SWITCH

OUTPUT NEXT BYTE

CLEAR INPUT POINTER REENABLE PRNT. INTS

KB.CXT

KB.RG

KB.MXT

KB.OMT

Y

LINE JUST BEGUN OR FULLY DONE ?

Y

ANY INPUT LEFT ?

CLEAR RUBOUT SWITCH

IF ECHO BUFFER FULL, DISABLE KBD INTS.

HOLD PTR ON BYTE OUTPUT, REENABLE PRINTER INTS.

RESET ECHO BUFFER POINTERS RE-ENABLE KBD. INTS.

N

N

N

KB.UXT

KB.IXT

FOR NO LINE IF COMMENT GONE, CLEAR IN. PTR.

ADJUST POINTER TO FORCE EXIT FROM CMD STATE

STORE BYTE (OR REPLACEMENT) IN ECHO BUFFER

PRINTER BUSY?

Y

SET UP LISTENER RECALL

EXIT TO MAKE READ CALL

Figure 6-5

KEYBOARD
INTERPRETER
OPERATIONS

ENTRY VIA
EMT 33 HANDLER

KBI.
RESTORE STACKED REGISTERS IGNORE CALL RETURN

SET UP OVERLAY CALL IN CASE, CMD ≠ CO, ST, WA

EXTRACT CMD CODE FROM INPUT, CLEAR FROM BUFFER (RO SW. NEXT CMD)

2 BYTES ENTERED AS CODE ?
Y / N

KBI.AG
MOVE IN POINTER PAST FIRST COMMA/SPACE OR TO ↵

KBI.GO
SET UP MUS MASK (FOR CO, WA ST). SET POINTER TO SVT

CMD = CO ?
Y / N

CMD = WA ?
Y / N

CMD = ST ?
Y / N

KBI.CO
MOVE SVT POINTER TO MUS SET UP 'INV CMD' ERROR

CMD O.K. FOR SYSTEM STATE ?
Y / N

SWITCH ADDRESS IN WRA (SVT) WITH FINAL EXIT ON STACK

ADJUST MUS AS APPROPRIATE CLEAR ERROR FLAG (R1)

STOP SWITCH SET ?
Y / N

'RESET' TO STOP ALL I/O + RESET MUS

KBI.QY
SET UP SPEC. DVR RECALL SET DONE FLAG (R5=0)

KBI.WA
ADJUST MUS MASK

KBI.ST
SET FLAG TO SHOW 'STOP'

KBI.OG
SET POINTER TO INTERNAL DDB & TO OVERLAY AREA FOR INDEX

DETERMINE INDEX SEGMENT FOR RANGE OF CMD CODE

SEARCH SUCCESSFUL ?
Y / N

STORE AS COMPLETION RETURN (DDB) SYS. EXIT FOR DRIVER DEQUEUE

CONVERT INDEX SEGMENT TO DISK DATA. SAVE FOR LATER

SAVE REGISTERS ON STACK

SET OV'LAY AREA ADDRESS AS BUFFER IN DDB

STORE COMPUTED BLOCK IN DDB

GET & STORE IN DDB ADDRESS OF SYS. DRIVER

PREPARE DUMMY LINK-BLOCK ON STACK

CALL SYS DRIVER TO LOAD INDEX.SEG

KBI.WS
CALL .WAIT TILL DONE

RESTORE SAVED REGS. + DISK DATA

SEARCH INDEX FOR CODE

AT SAME TIME, BUILD UP ON DISK DATA FOR REQUIRED BLOCK

CODE FOUND ?
Y / N

KBI.BY
SET UP 'ILL CMD' ERROR FLAG

STORE OV'LAY CALL FOR KSB. DROP THRU

KBI.XT
EXIT AS SET UP IN ROUTINE

Figure 6-6

Figure 6-7

SAVE COMMAND PROCESSING (PHASE II)

(Executed from a buffer claimed from free core during Phase I)

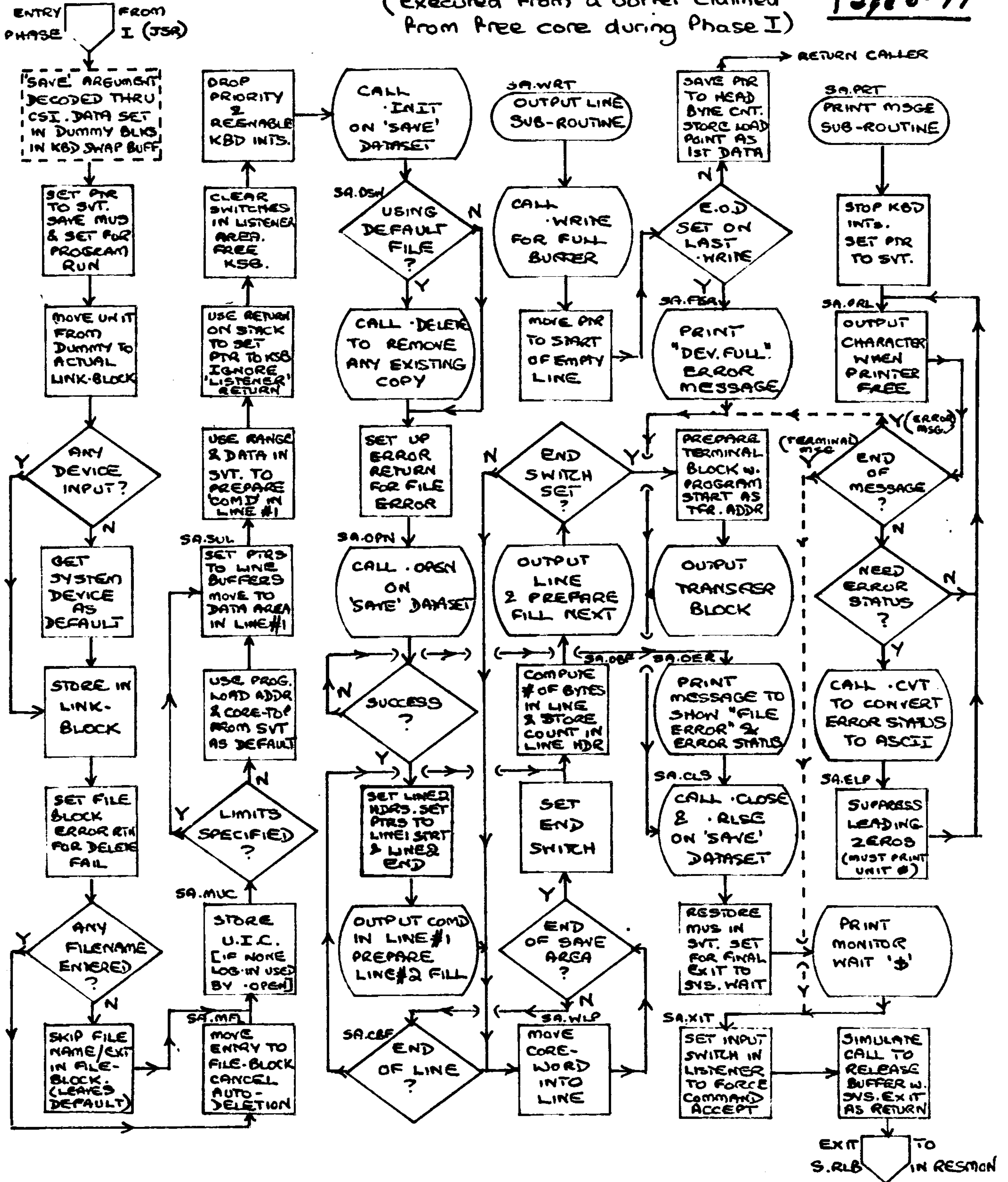


Figure 6-8

BEGIN/RESTART COMMANDS PROCESSING

ENTRY EMT 30 VIA HANDLER

KBI.BE

RESTORE REGISTERS SAVED BY CALL. IGNORE RETURN

PREPARE FOR 'INV CMD' ERROR

PROGRAM LOADED ? (N / Y)

ERROR NOW "ILL. SYN" CLEAR REG. STORE

BE.GTA

AT END OF COMMAND INPUT ? (Y / N)

NEXT BYTE = DIGIT AFTER STRIP ASCII? BASE (N / Y)

MULTIPLY PREVIOUS INPUT BY 8 & ADD NEW DIGIT

BE.QY

CLEAR DONE FLAG. SET UP RECALL TO SPECIAL DRIVER

RE.XT

MOVE SP TO NEW STACK RE-ENABLE KBD. INTS.

EXIT AS SET UP

BE.CKS

IF 'BEGIN' SET FLAG IF 2ND ARG (S). MAKE ERROR 'ILL. ADDR'

BE.CKA

IF NO ADDRESS INPUT, USE DEFAULTS FROM SVT.

BE.CVA

ADDRESS ON WORD-BOUND IN CORE ? (N / Y)

BE.GO

STOP ALL I/O WHEN PRINTER DONE - RESTART CLOCK

SET UP NEW STACK BELOW PROG LOAD POINT WITH EXIT TO ADDRESS

CLEAR KBD LISTENER SWITCHES & RESET ITS PTRS.

FREE THE MSB. & RE-INIT SVT. AS REQD

SET UP OV'LAY CALL IN CASE "BE"

RE.SMU

SET MUS TO 'PROG. RUN'. SET 'RTI' EXIT. RESTORE USER. REGS

ENTRY OV'LY TO #2

BE.CON

RESTORE REGISTERS SAVED BY CALL. IGNORE RETURN

BUILD CALL TO OV'LY 3 & SAVE. SET PTR TO INTERN'L LINK-BLOCK

ANY DDBs IN MONITOR CHAIN ? (Y / N)

BE.NDI

CALL OVLY 3

BE.CLR

RESTORE REGISTERS SAVED BY CALL. IGNORE RETURN

BE.ZAT

UNLINK D.A.T SEGMENTS IN BUFFERS (ABOVE EOM) IF ANY

BE.ZDL

CLEAR BUFFER ALLOCATION TABLE. RESET TOB & STACK STOP

RESTART REQUESTED ? (Y / N)

RE.ZDC

CLEAR ALL I/O BUSY FLAGS FROM DDBs & DRIVERS

BE.DEF

IF DEVICE IS DECTAPE, CLEAR BLOCK REF. TO FORCE NAMED. EXIT

WRITE OUT BLOCK IN CORE BY .TRAN.

UNLINK CHAIN FROM SVT. CLEAR LINK-BLOCK ERR. RENEW

BE.ZDC

GET DVR ADDRESS FROM DDB CLEAR ANY & LINKAGE

ANY DRIVER ATTACHED ? (Y / N)

BE.ZDN

MORE DDB's IN CHAIN ? (Y / N)

CLEAR BIT-MAP LINKS IN SYSTEM DRIVER

BE.ZDG

CLEAR DDL CORE ADDRESSES & INT. VECT FOR NON-RES DRIVERS

CLEAR REGISTERS SET MUS TO 'PROG RUN'. SET 'RTI' EXIT

EXIT TO PROGRAM START

UNLOCK FILE. SET PTR TO EXTENSION START

BE.CLY

ERROR ON LAST .TRAN ? (N / Y)

AT Ø LINK IN FILE CHAIN ? (Y / N)

READ LINK FROM NEXT BLOCK IN FILE BY .TRAN

CLEAR DRIVER BUSY FLAG

DEVICE FILE-ORIENTED WITH FILE OPEN & FIB ? (N / Y)

BE.CDN

REMOVE BUFFER FROM STACK

BE.CLF

CALL .CLOSE ON FILE

FORCE BLOCK WRITE (Ø LINK)

BEDOF

CLEAR FILENAME FROM DIR'Y ENTRY. SET PTR TO FILE START

FILE OPEN FOR EXTENSION ? (Y / N)

READ-IN DIRECTORY BLOCK BY .TRAN

DIRECTORY BLOCK ENTRY IN FIB ? (Y / N)

BE.DLT

PREPARE TRAN-BLOCK BASED ON DDB DATA

SAVE SP. ALLOCATE STD BUFFER ON STACK & LINK TO DDB.

LINKED FILE OPEN FOR OUT ? (N / Y)

USER WANT 'SAVE' ? (N / Y)

BITMAP IN CORE WITH FIB LINKED ? (Y / N)

Figure 6-9

TIME COMMAND PROCESSING

ENTRY 30 VIA EMT HANDLER

KBI.TI

RESTORE REGISTERS SAVED BY CALL IGNORE RETURN

SET PTR TO TIME IN THE SVT

NEXT INPUT BYTE = DIGIT ? (Y / N)

TI.OUT

GET TIME FROM SVT.

REMOVE LOW ORDER SIGN

SET PTR TO TABLE OF REDUCTION FACTORS (1 DIGIT)

TI.IGO

CLEAR STORE FOR ELEMENT RESULT

SET PTR TO TABLE OF MULT FACTORS

GET INPUT BYTE & REMOVE ASCII BASE

CLEAR STORE FOR NEW TIME ON STACK SET PTR TO IT

RESULT = DIGIT ? (Y / N)

MULTIPLY PREVIOUS RESULT BY 10 & ADD NEW DIGIT

TI.IMG

RESULT FOR ELEMENT ACCEPTABLE ? (Y / N)

RESULT = Ø ? (Y / N)

INPUT ELEMENT-1 = Ø ? (Y / N)

ADD MULTIPLY FACTOR TO TIME STORE (DP)

TI.IGN

DONE SECONDS INPUT ? (Y / N)

MORE INPUT TO DO (Y / N)

MOVE POINTER TO NEXT FACTOR

NON-DIGIT BYTE = ':' ? (Y / N)

TI.IDN

STORE NEW TIME IN SVT. CLEAR ERROR FLAG

TI.ERR

SET UP 'ILL SYN' ERROR FLAG

TI.OGO

SET CONTROL COUNT

TI.OEL

GET LOW ORDER OF REDUCTION FACTOR

TI.ODL

BUILD DIGIT BY SUCC^VE SUBTRACT LEAVE POSITIVE REMAINDER

SET ASCII DIGIT BASE IN CNTR.

TI.OPL

PRINT DIGIT PREPARE FOR COLON PRINT

DONE 2 DIGITS ? (Y / N)

PRINTED SECONDS ? (Y / N)

PRINT CR/LF CLEAR ERROR FLG

TI.IXT

CLEAR DONE FLAG SET UP SPECIAL DRIVER RECALL

EXIT TO DO RECALL

Figure 6-10.

MODIFY COMMAND PROCESSING

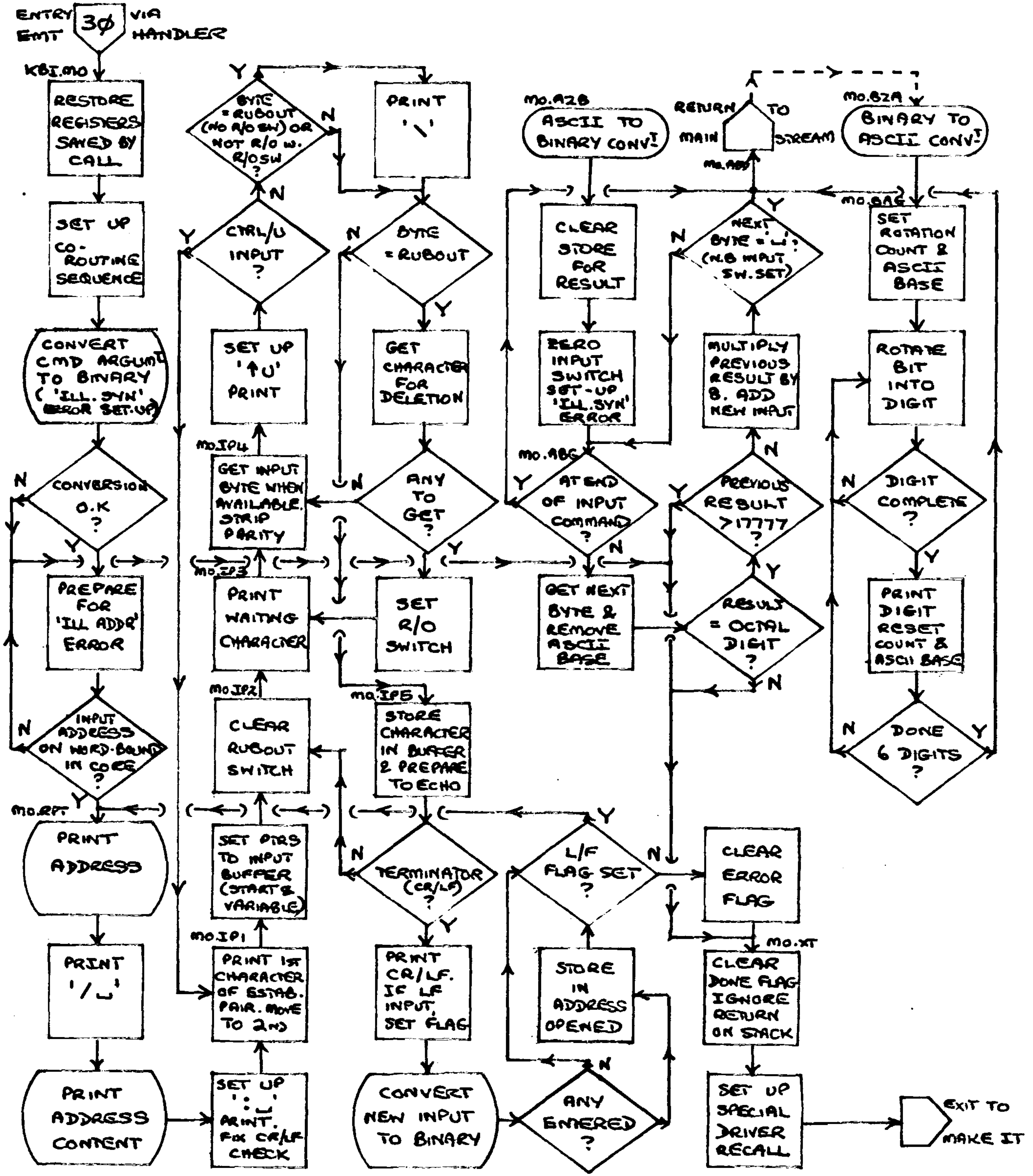


Figure 6-11

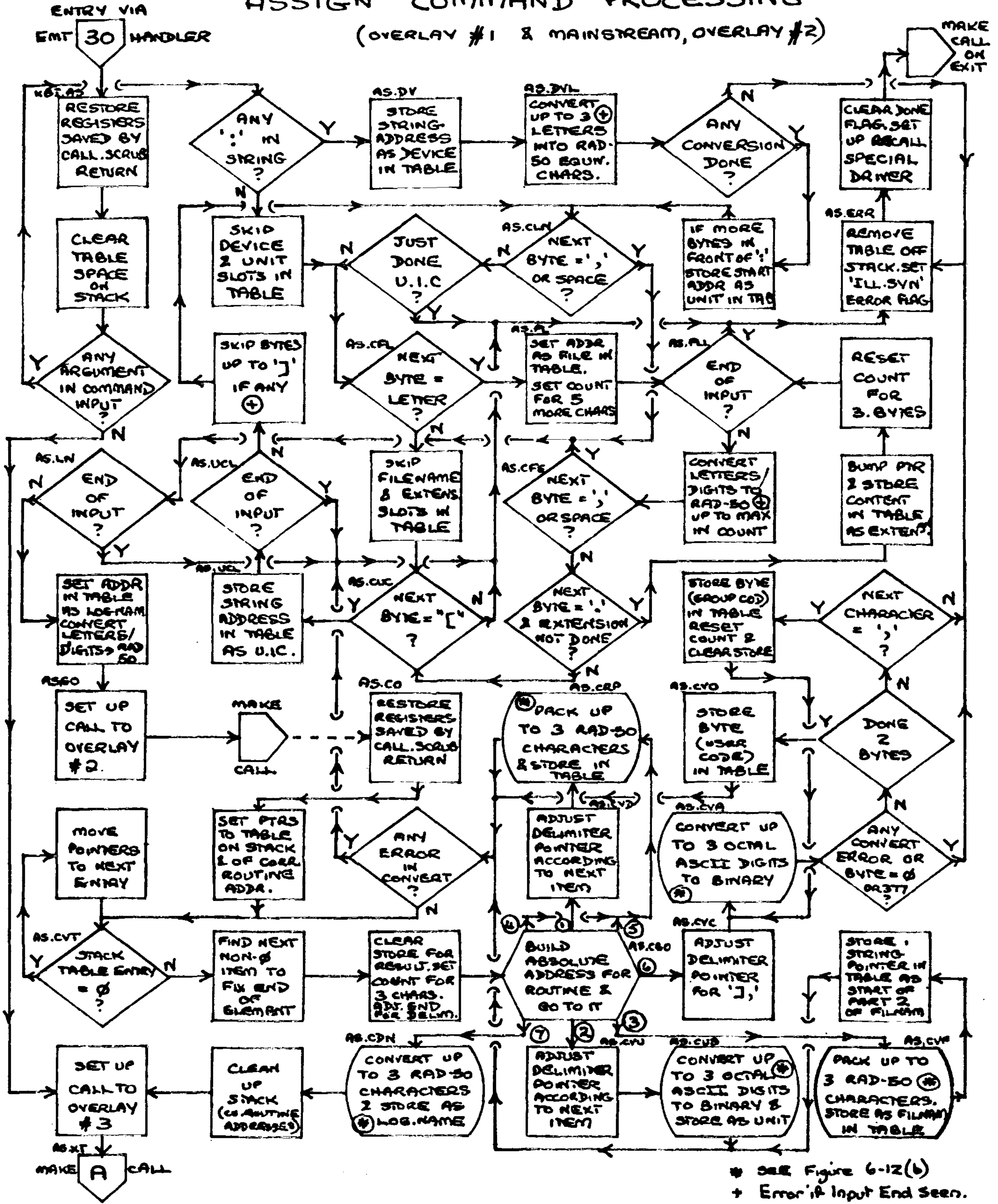


Figure 6-12(a)

ASSIGN COMMAND PROCESSING
(OVERLAY #3 & SUB-ROUTINES, OVERLAY #2)

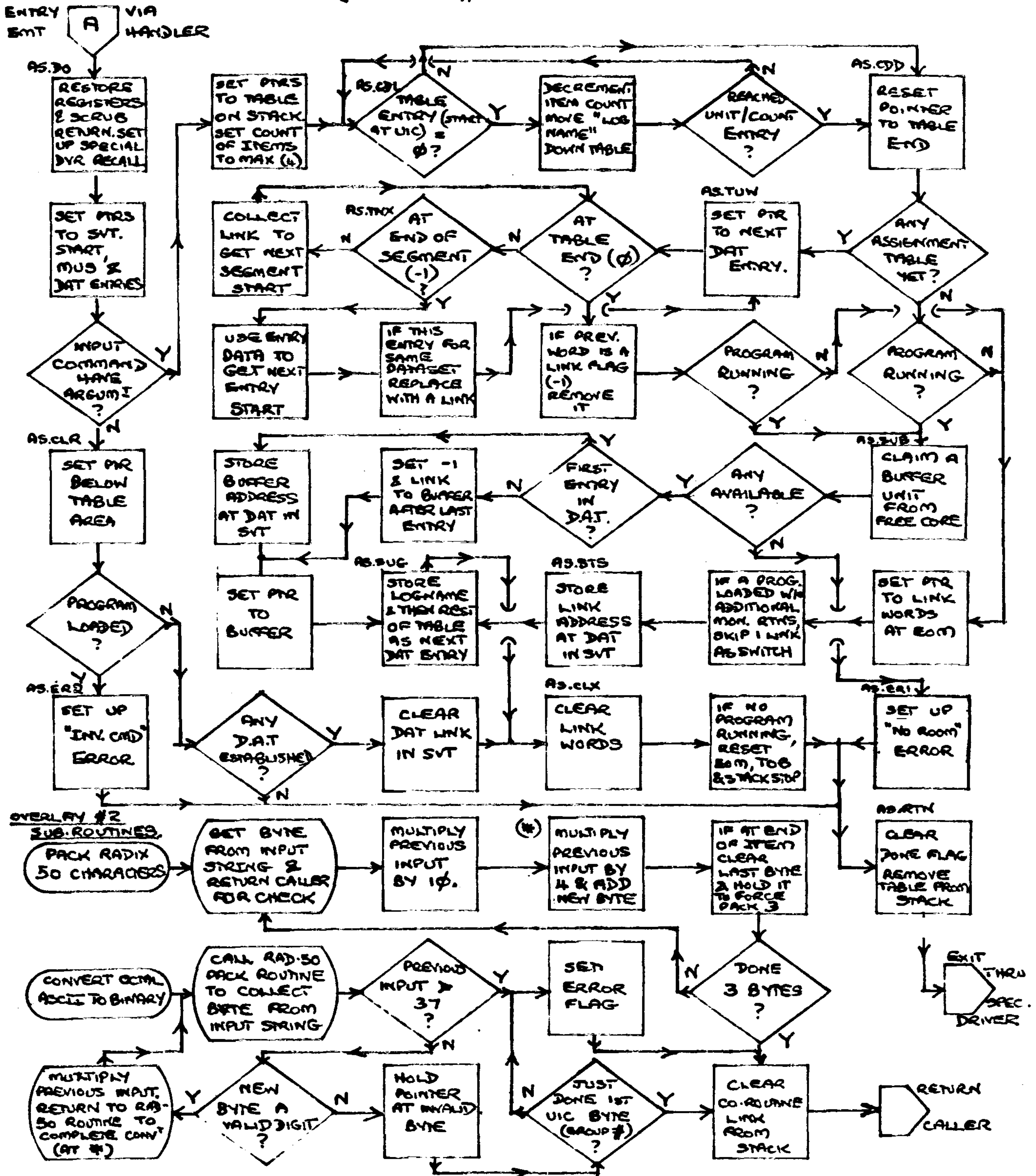


Figure 6-12(b)

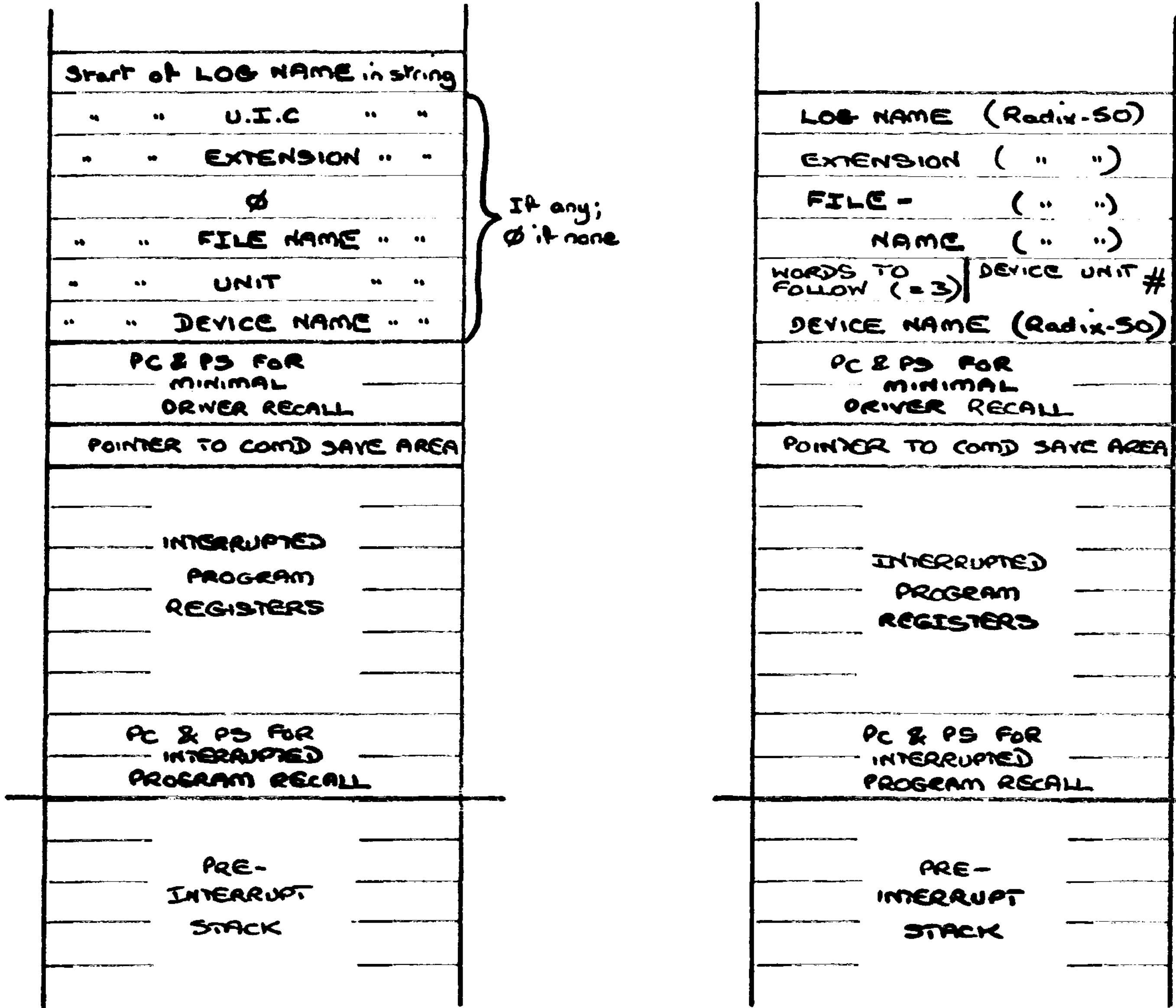


Figure 6-13: STACK STATE DURING 'ASSIGN' PROCESSING

```
001000:
 00:                                                        :
 20:                                                        :
 40: 014204 012703 001402 120427 000067 101033 005000 120427: DXCUBCW17@[B@JWI
 60: 000027 101014 005743 013100 001421 152715 000340 005710: W@LB#K@VQCMU @HK

001100:
 00: 001023 130504 001424 010210 111615 022341 006304 062704: SBD1TCHPMSISDLDX
 20: 170474 011404 010416 006004 103432 012607 120427 000006: <1DSNQDLZGGUWIF@
 40: 001764 010216 010346 000004 105404 011102 000401 005722: 4CNP&PD@DKBRAARK
 60: 012625 005741 010100 106204 100707 103401 022520 014110: UUIK@PDLGAAGPXHX

001200:
 00: 010240 010541 004567 001314 062606 000002 011646 030516:  PIQ7ILBFXB@&SN1
 20: 001750 012704 001460 012403 121227 000030 103405 121227: (CDU0CCUW"X@EGW"
 40: 000040 103002 012704 003702 151715 012343 001044 021624:  @BFDUBGMS#TSBT#
 60: 001012 005714 100002 105714 001036 005224 014116 041716: JBLKB@LK^BTJNXNC

001300:
 00: 020021 112615 010407 105714 001026 011644 010046 010300: Q MUGOLKVBS&P@P
 20: 010223 112715 000200 012301 012413 042713 140001 006223: SPMU@@ATKUKEA@SL
 40: 010423 014413 042713 037777 001001 006213 000313 005413: SOKYKE??ABKJK@KK
 60: 004767 001220 012600 010026 001401 005010 000605 005046: 7IPB@UVPACHJEA&J

001400:
 00: 005760 000012 100660 010016 004767 001276 005036 000402: BKJ@BANP7I>B^JBA
 20: 105367 000040 004567 001074 000002 000000 000000 004350: 7J @7I<BB@@@@(H
 40: 000000 000010 000345 003704 177600 000004 001376 000000: @@H@X@DG@?D@>B@@
 60: 001440 140317 177400 010102 012260 000002 014142 020106:  CO@@?BPBTB@"XF

001500:
 00: 001375 014002 022022 013703 000040 016303 000020 001432: =BBXRSCW @C@P@ZC
 20: 005004 156304 000006 060304 012401 100416 010403 022400: DJD@F@D AUNACQ@X
 40: 001403 106201 103374 000416 005726 130143 001116 005070: CCAL<FNAVK#0NB8J
 60: 000002 005010 000167 000432 022400 001403 106201 103374: B@HJ7@ZA@XCCAL<F

001600:
 00: 000401 005044 010004 012701 000010 060104 014426 001451: AASJDPAUH@D VY)C
 20: 011244 100473 060104 004767 000016 010046 013746 000052: SR;AD 7IN@&P&W*@
 40: 004736 005036 005005 000571 012614 032744 000002 001417: ^I^JEJ9ALUSB@OC
 60: 026464 177774 000004 103013 010046 010602 011046 010246: 4-<?D@KF&PBQ&R&P

001700:
 00: 013746 000050 004736 016600 000004 104001 012636 014002: &W(@^I@]D@AH^UBX
 20: 022022 016046 000006 005046 116216 000102 013746 000056: RS&@F@&JN@B@&W.@
 40: 004536 005742 010201 162701 000016 013703 000040 016304: ^I"KAPAXN@CW @D@
 60: 000050 016405 000006 020264 000002 001413               : (@E]F@4 B@KC
```

LINE START ADDRESS | OCTAL WORD CONTENT | BYTE ASCII EQUIVALENT

EXAMPLE OF LINE-PRINTER DUMP
(DU LP.,,1040,1778)

Figure 6-14.

DUMP COMMAND PROCESSING

ENTRY VIA EMT 30 HANDLER

KBI.DU

RESTORE REGISTERS SCRUB RETURN CLEAR TABLE SPACE

SET "ILL.ADDR" ERROR FLAG.

DU.GDV

MULTIPLY PREVIOUS DEVICE NAME INPUT BY 50₈

AT ELEMENT DELIMITER ? (N / Y)

NEXT BYTE A LETTER ? (Y / N)

ADD RAD -50 VALUE TO PREVIOUS INPUT

DU.GHL

RESULT A WORD-ADDRESS ? (N / Y)

NEXT BYTE A DELIMITER ? (N / Y)

NEXT BYTE A DIGIT & PREV. INPUT ≤ 17777 ? (N / Y)

MULTIPLY PREVIOUS INPUT BY 8 & ADD NEW DIGIT

DU.GDU

NEXT BYTE A DIGIT ? (Y / N)

STORE AS DEVICE UNIT IN TABLE

DONE BOTH LIMITS ? (N / Y)

DU.GLL

CLEAR BUFFER FOR LIMIT CONVERSION

NEXT BYTE A DELIMITER ? (Y / N)

DU.GTD

IF NEXT BYTE = 'I' SET "IN" SWITCH IN TABLE

DU.DVD

IS NEXT BYTE = ':' FOLLOWED BY DELIMITER ? (Y / N)

NEXT BYTE A DELIMITER ? (Y)

IF NO UPPER LIMIT, USE TOP OF CORE

COMPUTE DUMP AREA SIZE

DU.ER1

RESET ERROR TO "ILL.SYN"

DU.ER2

CLEAR TABLE FROM STACK

DU.ER3

CLEAR DONE FLAG SET UP SPECIAL DRIVER RECALL

EXIT TO MAKE CALL

UPPER LIMIT GIVEN IN CORE ? (Y / N)

RESULT NEGATIVE (LO > HI) ? (N / Y)

ADJUST ERROR TO "ILL DEV." USE SYS. DEV IF NONE

DU.GO

DEVICE ACCEPTABLE ? (Y / N)

MOVE TABLE TO REGISTERS SET UP OVERLAY CALL

DEVICE = LP ? (Y / N)

DU.LL8

CVT & PRINT START ADDRESS FOR PAGE PRINT ':' FORCE L/F

DU.LLA

FORCE FORM-FEED

SAVE SP IN SVT. SAVE ACTUAL START & RESET PTR TO PAGE ST.

DU.LBO

OUTPUT DUMP REQUEST ? (Y / N)

DU.LP

RESTORE REGISTERS & SCRUB RETURN

CALL OV'LY #2

DU.LLC

PRINT LAST 2 DIGITS FOR LINE START. PRINT ':' SET FOR '␣'

DU.LLI

AT LINE END ? (Y / N)

DU.LLJ

SET UP L/F PRINT

AT DUMP END ? (N / Y)

DONE 8 LINES ? (Y / N)

PRINT LINE FEED

DU.LLD

BEFORE START & AFTER END PAD BLANK. OTHERWISE O/O CONTENT

DU.LLH

PRINT SPACE OR CHARACTER AS READ

BEFORE START & AFTER END PAD BLANK OTHERWISE ADJUST BYTE TO L/P RANGE

DU.LOD

PRINT FORM-FEED CLEAR ERROR

CTL/C ENTERED AT KBD ? (Y / N)

PAGE COMPLETE ? (Y / N)

DU.LLF

DONE 8 WORDS ? (N / Y)

DU.LLG

RESET PTR TO LINE START. ADJUST LINE END PTR IF DUMP DONE.

DU.EXT

CLEAR DONE FLAG. SET UP SPECIAL DRIVER RECALL

DU.ER5

SET UP "ILL DEV" ERROR

DU.IXT

RESTORE REGISTERS & SCRUB RETURN

CALL OV'LY #3

EXIT TO MAKE CALL

Figure 6-15

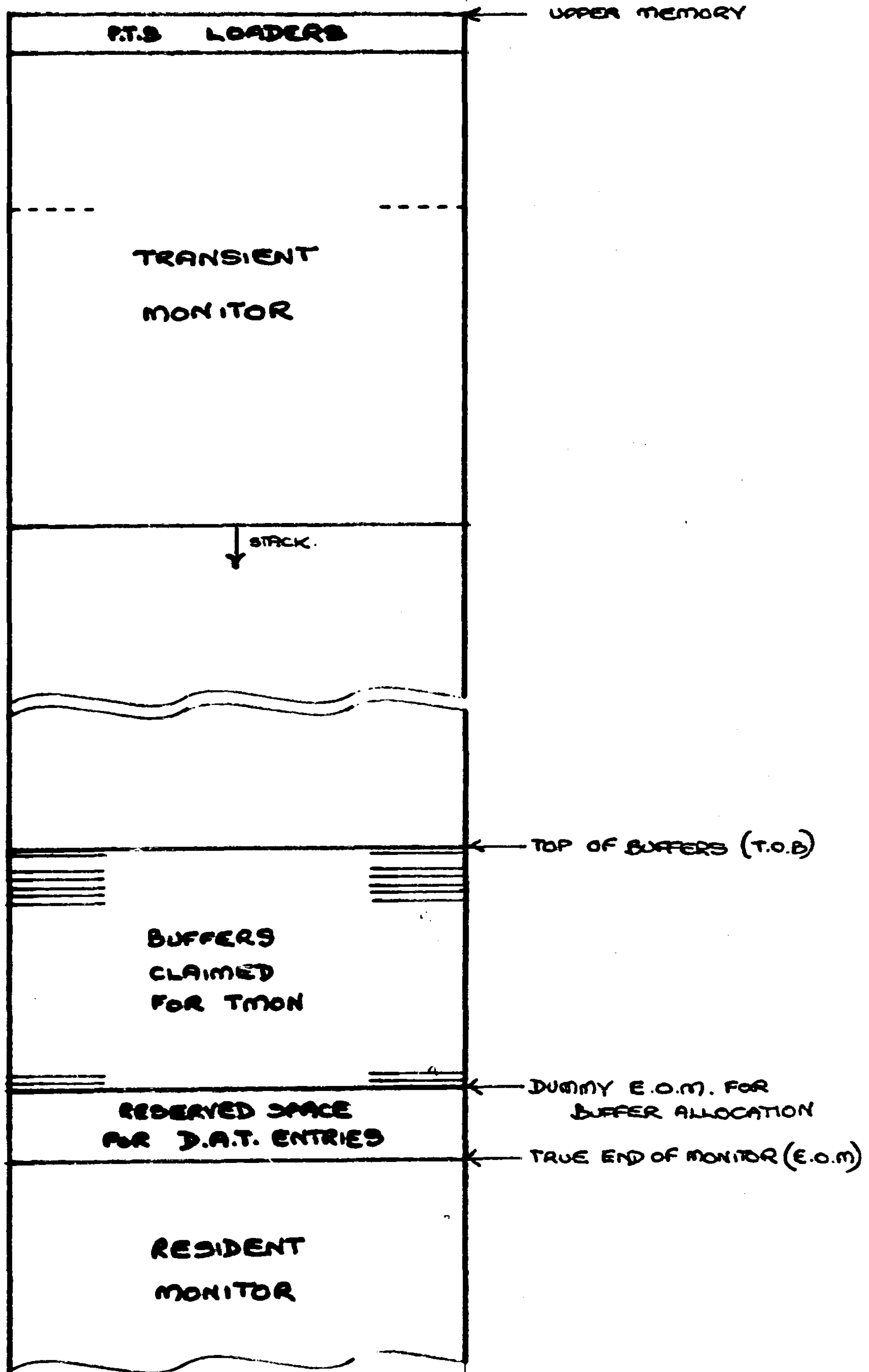


Figure 6-16: E.O.M. Adjustment by TMON

TRANSIENT MONITOR - GENERAL PROCESSING

CALLED & ENTERED FROM .XIT

TMON: ENABLE KBD INTS. SET PTR TO GENERAL LINK-BLOCK

TM.GO: ANY DDB IN MONITOR CHAIN?

ASSOC DEVICE FILE-ORIENTED OR MAGTAPE?

DATA BUFFER STILL ATTACHED?

FILE OPEN FOR OUTPUT?

MAGTAPE?

TM.IOC: STOP ALL INTERRUPTS RESTART CLOCK (IF ANY). GET EOM (SVT)

TM.NFS: CALL .RLSE

ENSURE BUFFER NOW DETACHED

TM.MTC: CALL .CLOSE TO CLEAN-UP

FIB ATTACHED & BIT-MAP IN CORE?

ANY ASSIGNMENT TABLE?

SPARE LINK-WORD (DAT IS POST-LOAD)?

CLEAR DAT LINK IN SVT

TM.ATC: CLEAR 2 LINK-WORDS FOR LATER USE

SAVE TRUE EOM

LEAVE ROOM FOR NEW ASSIGNS. SET DUMMY EOM, TOB & STACK STOP

CLEAR BUFFER ALLOCATION TABLE

TM.SAT: SEARCH FOR END OF FIRST DAT SEGMENT

TM.ATE: SEGMENT END AT READ 'EOM'?

SET EOM AT SEGMENT END

ENTER SYS. DEVICE NAME INTO GENERAL LINK-BLOCK

TM.ZDL: REMOVE CORE-ADDRESSES FOR NON-RES DRIVERS FROM DDL

REMOVE PROGRAM-DEPENDENT ENTRIES FROM SVT

TM.BIF: CALL .INIT ON KEYBOARD & .WAIT FOR PRINTER COMPLETION

SAVE KBD INT VECTOR & ATTACH SPECIAL INTERRUPT HANDLER

TM.RWR: CALL .INIT ON PRINTER & .WRITE TO RESTORE CARRIAGE

PREPARE TRAN-BLK FOR MRT START & SIZE

CALL .INIT ON GENERAL DATASET

REFRESH MRT FROM SYS. LIBRARY BY CALL .TRAN

CALL .OPEN ON KEYBOARD (OR ? BATCH DEVICE)

RESTORE TRUE EOM & HOLD DUMMY

SET LOAD FLAG, SAVE MRT DATA RESET TO USE RWR EMBEDDED

READ/WRITE PROCESSOR RESID.?

SET PTRS TO CONSOLE LINK-BLOCKS

TRANSFER FAIL?

FØ ERR.

TM.RDS: CALL .WRITE TO PRINT '$'

CALL .READ & .WAIT FOR COMMAND INPUT

CTL/C ENTERED (PASSED AS VT)?

TM.DEC: STOP KBD INTERRUPT. REMOVE COMMENT FROM INPUT

ANY COMMAND LEFT?

REENABLE KBD. INTS. SET UP KBI CALL

TM.ADC: SET PTRS TO READ BUFFER & INTERNAL CMD TABLE

CALL .WRITE TO ECHO '↑C'

TM.EKI: SET CMD O/P SWITCH IN SPECIAL DRIVER BUFFER AREA

END OF TABLE?

CMD = ENTRY IN INTERNAL TABLE?

WAIT TILL PRINTER FREE FROM COMMAND CLEAN-UP

DROP PRIORITY & RESTORE REGISTERS

SAVE RETURN PTRS IN SPECIAL DRIVER BUFFERS

TM.KIG: SIMULATE CALL TO KBI THRU KBL

SAVE VALID REGISTERS SET LEVEL 4 PRIORITY

TM.CMD: MOVE PTR TO START OF ARGUM' PREPARE STACK & REGISTERS

A: RETURN ON RUN/GE FAILURE

PROCESS SPECIAL COMMAND [LO, FI, RU, GE]

Figure 6-17

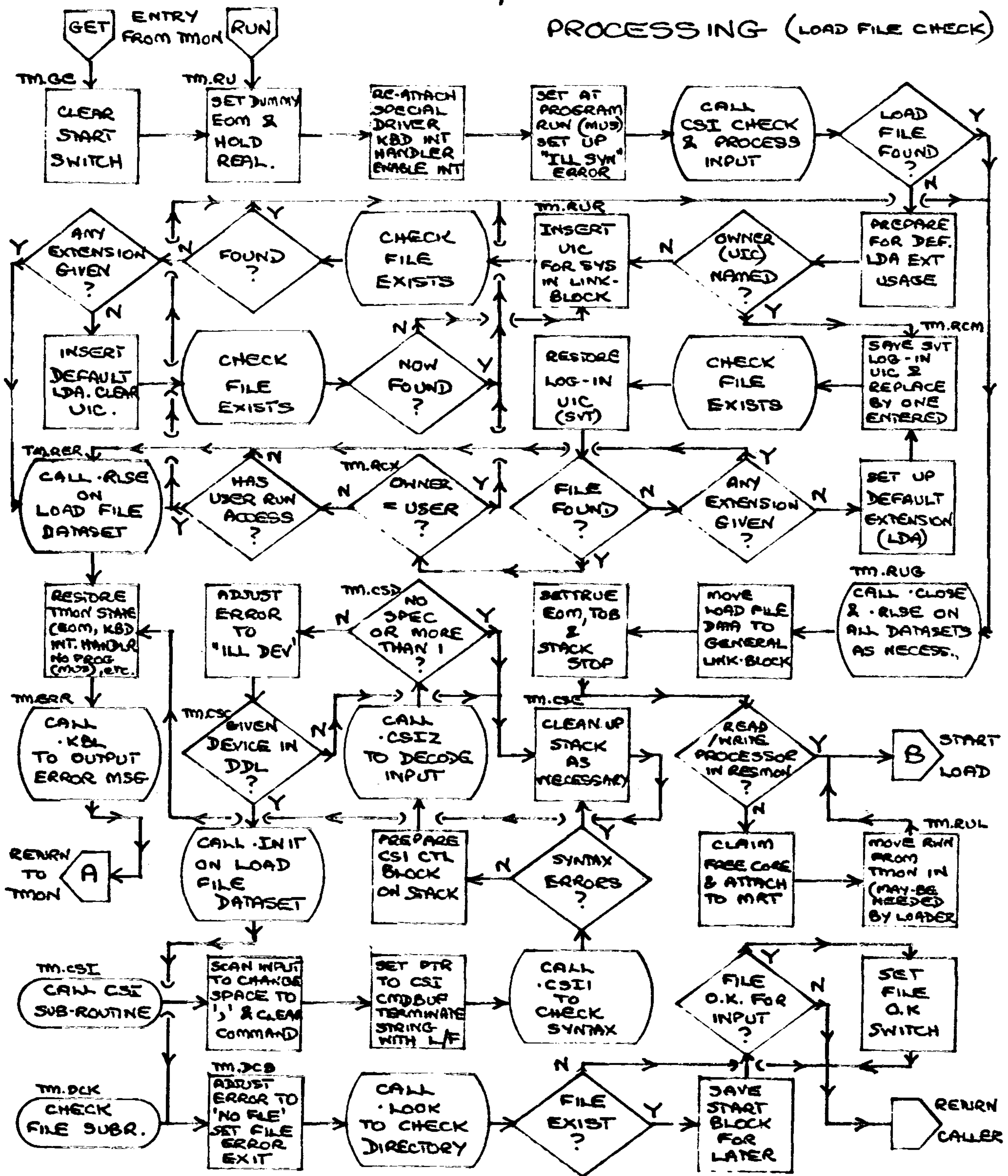


Figure 6-18(a)

RUN/GET COMMAND PROCESSING (PRE-LOAD)

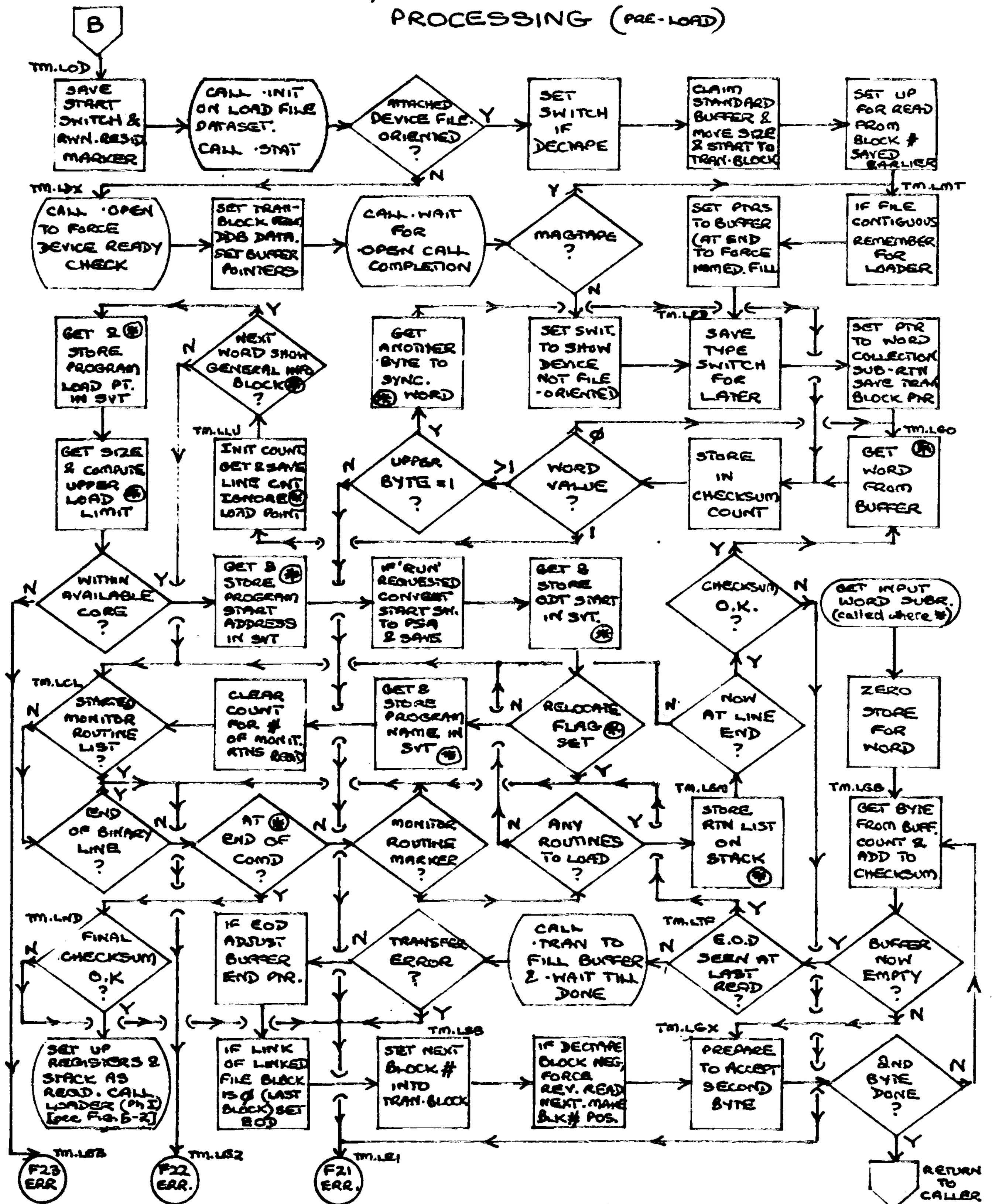


Figure 6-18(b)

CHAPTER 7

ERROR HANDLING

It has been noted at various places in this Manual that recognition of some error condition can result in a call for the printing of a diagnostic message at the console typewriter. The printing service is provided centrally by a special Monitor module and is used by the Monitor and the System Programs. The purpose of this chapter is to describe this service.

Section 7.1 explains the coding scheme used to identify the errors. Section 7.2 shows how the special module is brought into memory and the module itself is outlined in section 7.3.

7.1 Types of Error

The Programmer's Handbook shows that all errors handled by the Diagnostic Print routine appear as an identifying number prefixed by a code letter followed by an item of support information, e.g.

F007   23462

meaning that an I/O request was made from location 23462 for a service for which no buffer room could be found.

The purpose of the code letter is to inform the user of the seriousness of the error and to indicate the follow-up action taken by the Monitor. Internally it is represented by a number which forms the high-order byte of a word of which the error number is the low-order byte. The code number is even for errors which are more advisory than drastic or which the program can handle itself. For these, therefore, the program is recalled after the message has been printed and processing continues. The odd number codes are reserved for errors which at best need some external remedial action and at worst mean normal program continuation is out of the question.

The codes currently assigned are:

0. Informational (I) - this does not signify an error condition in the strictest sense. It allows usage of the print routine by a program wishing to notify the operator of its current state, e.g. A Stop number is printed in this way by a running FORTRAN program before it exits to the Monitor.

1. Action (A) - this indicates that the operator must perform some rectifying process before the program can proceed, such as correctly setting up a device needed by the program. If the operator requires resumption, he can effect this by entering a console CONTINUE command. The program is then continued immediately following the print request.

2. Warning (W) - this is mainly provided for the use of programs which need to inform the user that his results may be suspect but in the meantime processing continues, for instance, when LINK-11 enters the second pass of a link operation with some global references still undefined.

3. Fatal (F) - this shows that the error is too serious to allow resumption. It usually means that the Monitor has detected some hardware or software problem which cannot be corrected immediately. Following such an error, the program must be started afresh by a console BEGIN or RESTART or it must be removed from memory with a KILL.

4. System Program Error (S) - this is used by a System Program which cannot proceed with its current task but may be able to accept a new one after some form of re-initialization, e.g. because of an invalid command string input.

The complete range of the errors within these categories is listed in the Programmer's Handbook.

5. Real-Time Error (R) -- see RSX manual.

7.2 Calling Diagnostic Print

Whenever a Monitor routine or System Program needs to request the diagnostic printing service, it must supply the code and number for the error and also the support information to be included in the message. These items are passed on the stack. The call itself is made by an IOT instruction initially, e.g. the full sequence might be as follows:

```
        MOV     (PC)+,-(SP)         ;PASS DEVICE NAME
        .RAD50  'DT'
        MOV     (PC)+,-(SP)         ;SHOW 'DEVICE NOT READY'
     -> .BYTE   2,1                 ;I.E. ACTION #2
        IOT                         ;CALL ERROR PRINT
```

see page 7-6

The reason for this use of IOT is that it provides immediate access to a special checking routine which is loaded with the permanently resident Monitor, as part of the initialization module described in section 2.1.4. In particular, the ultimate error message - coded 0,0 - which signifies hardware failure of the system-device, must be intercepted since there is then no point in trying to bring the Diagnostic Print module from that same system-device. Thus if this error is detected, the monitoring routine halts the system.

This special routine is also entered directly as a result of illegal trapping through unused vectors in the memory locations below 400. It was also noted in section 2.1.4 that the Monitor initialization routine stores a value in the codition code positions in each vector status word and this is of course combined with the trap priority level set at 7. The routine moves this status to the stack as an error number with a high byte set at 3, thus producing a valid fatal error code. The last entry on the stack at this time is the PC content saved by the trap; this automatically serves as useful support information. For compatibility with the normal error call through IOT, which itself causes a PC and status save, two words are then 'pushed' onto the stack in the direct entry case. (see Figure 7-1)

The Diagnostic Print module is called by the special routine by means of EMT 32. Because of its code it is brought into the subsidiary Keyboard Swap Buffer. Thus as noted in section 2.3.1, it is possible for diagnostic print calls to be made from other routines temporarily resident in the main Swap Buffer, though not by the processors within the Keyboard Language (As shown in section 6.4, these therefore handle their own errors)(1)

----

1. A problem can arise, however, if an Error Diagnostic Print is called from an interrupt, (because, for instance, a driver has detected some hardware failure in its device), at the same time that a keyboard command is being processed. The resulting conflict for the use of the KSB cannot currently be resolved. Hence the user is advised to take care in his keyboard usage while other I/O transfers are underway, particularly on the more complex bulk-storage devices (see "Getting DOS on the Air" DEC-11-SYDD-D).

7.3 Diagnostic Print Routine (EDP)

The Diagnostic Print routine has two main functions:

a. To convert the error code and its support information into the appropriate ASCII string and dispatch it to the console typewriter.

b. To control follow-up action as dictated by the error code.

Call Sequence:

EDP is simply called by:

EMT 32

However it expects a stack state as discussed in the previous section.

Processing:

The sequence of operations is quite straightforward and therefore requires no illustration. The basic steps are:

1. Save the control status of the console typewriter, to enable its restoration after the message has been printed; wait for any character currently being output to be finished (1)

2. Output CR/LF followed by the error code letter derived by simple table look-up based on the code number.

3. Convert both the error number and support information into octal ASCII, using an embedded routine, (since the normal Swap Buffer may not be available for use by the Conversion Utilities routine described in section 5.3.). Print the resulting values with a dividing space and finally a second CR/LF.

4. Check the error code. If it is even (see section 7.1), restore the typewriter status and the program

---

1. If the error is called in the middle of a line of output, the message will in fact intervene.

Registers saved by the EMT call and remove both the return PC and status and the call arguments from the stack. Free the KSB and recall the program.

5. For odd codes, print '$' to warn the operator that command input is expected (see section 6.1) and set the Keyboard Listener input underway switch (see section 6.3) to force acceptance of that input without CTRL/C. Also transfer the saved typewriter status to the Keyboard Listener (unless the latter already has a saved value - indicating that the error message has interrupted the processing of a command (1))

6. Save the program return address in the SVT (WRA) and replace it on the stack with the address of the System Wait Loop in the SVT (see section 2.1.1) - this is actually a switch of the contents of WRA as in the console WAIT command (see section 6.3). Also clear the priority saved on the stack to provide for low-level waiting.

7. Set the Monitor/User switch in the SVT to show the program to be in a wait state (i.e. -1,1). For fatal error (code 3) show it also to be stopped to prevent acceptance of a keyboard CONTINUE (i.e. -1,-1) - see Section 6.3.3.

8. Return to step 4, to clean-up the stack, free the KSB and exit to the System Wait Loop with Registers reset to their content prior to the error call in readiness for program resumption should a console CONTINUE be acceptable ('A' errors)

Comments:

The Diagnostic Print routine in its present form is not re-entrant in any sense. Because of this, it must always be brought from the system-device so that it can be protected by SAM as shown in section 2.3. Moreover any device interrupt must be deemed a potential source of another error message. If this occurred while a previous message were being processed, the system would hang while SAM waited for the KSB to be freed - an impossible event when the print routine could not regain control to complete its first task. Hence presently, the print routine also locks the system by maintaining the priority level of 7 set by the original IOT call. This obviously may mean approximately a second during

----

1. Any command currently being input at this time will in fact be lost, since the setting of the input underway switch as noted will effectively delete the string so far entered.

which no interrupt can be accepted. The user should therefore be particularly careful to avoid foreseeable error conditions, if he cannot afford to have device access prevented for this period.

It should also be noted that the current method is liable to change. The coding system was originally designed to provide a form of access to a file of error messages stored on the system-device, with no restriction upon the amount of additional support information needed in each case. It is only set at one item for the time being for the sake of simplicity in the earlier stages of DOS development. Users therefore who wish to produce programs which also call the Diagnostic Print routine may do so but they must then be prepared to amend these programs should the original intention be implemented later.

Error Code Hierarchy

0 - Informational
1 - Action required
2 - Warning
3 - Fatal
4 - System

The routine prints the error code and for codes 1 or 3, enters the wait state. An operator request will cause an exit for fatal errors (3), or program continuation for code 1. Codes 2 and 4 cause message print and proceed only.

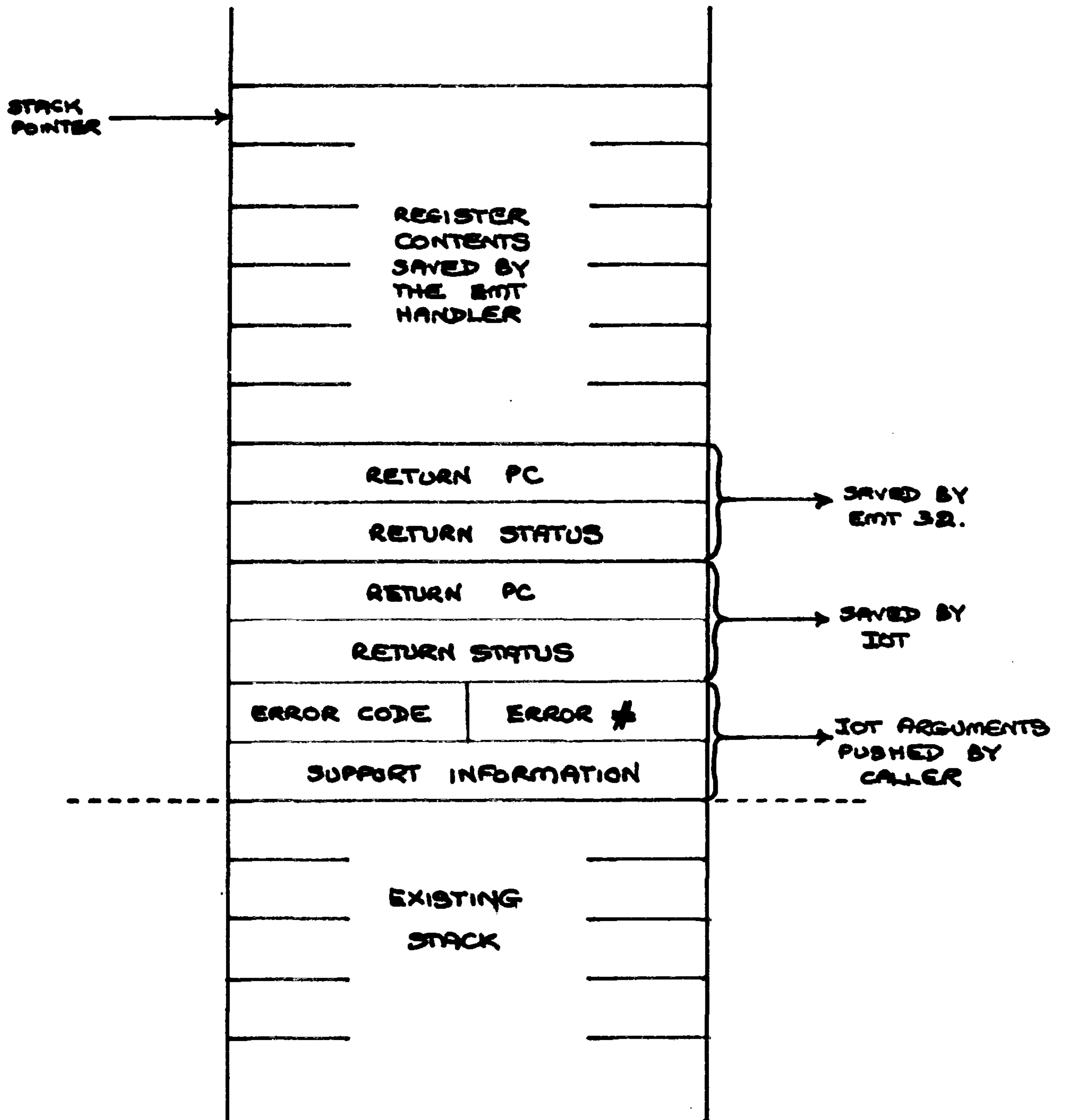


Figure 7-1: STACK STATE ON ENTRY TO THE ERROR DIAGNOSTIC PRINT ROUTINE.

CHAPTER 8

MONITOR GENERATION and MODIFICATION

The earlier chapters have shown that the DOS Monitor consists of an in-core control section supported by individual processing modules stored within a Library on the system device. It is the object of this chapter to describe how the system is built originally into this form and then to discuss the procedures by which it may be extended or modified.

Monitor generation begins with the assembly of the module sources and their subsequent linking. Section 8.1 identifies the requirements for these two stages. Section 8.2 then explains the actual building of the system-device Library. Section 8.3 is concerned with the problems of preparing and checking out new or replacement modules prior to their final inclusion in the system.

8.1 Monitor Module Preparation

In Chapter 1, it was noted that a major feature of the Monitor is the fact that each routine is a completely stand-alone module at the source-level. The type of disk being used as a system-device and the extent of the permanently resident Monitor Section, however, require that the modules are correctly assembled and linked before they are included in the system. These operations are therefore illustrated in the following paragraphs.

8.1.1 Module Assembly

The individual Monitor modules currently available through the DEC Program Library are listed at Figure 8-1. (Refer to "Assembling the DOS source Programs", DEC-11-SXDA-D.) They are all prepared for processing by the DOS Assembler, PAL-11R, as described in the relevant Programming Manual (DEC-11-ASDC-D). In general, each is simply assembled alone: thus assuming a DECtape source, binary output on the system-device and listings on a line-printer, the necessary command string to the Assembler might be:

```
#RMON2.OBJ,LP:,/PA:2<DT3:RMON2.PAL<CR>
```

There are nevertheless two types of exception:

1. System-device Driver - this must execute special sequences which do not apply if the same device is

used as an ordinary peripheral in a differently-based system. As noted in Section 3.3.4, the driver source is conditionalized for the two versions: the correct one is identified by the presence or absence of a definition for the parameter SYSDV at assembly. If this is omitted, a normal driver is obtained. The system-device driver requires the following procedure:

```
#DF.OBJ,LP:,/PA:2<KB:/PA:1,DT3:DF.PAL<CR>

SYSDV=0<CR>         ;ENTER DEFINITION VIA 'KB'
^C
.END <CR>           ;SIGNAL END OF INPUT
<CR>                ;(SEE SECTION 6.4.3)
```

2. Keyboard Command Language - as mentioned in Sections 6.2.3 and 6.2.4, indexing and overlaying operations necessitate special assembly to reflect the type of disk in use as a system-device because of varying block-size. This again means the entry of the appropriate parameters; they are provided on separate source modules: KBIPAR.PAL for RF11 and RC11 disks; KBIPAR.4DK for RK11. For the modules marked '*' in Figure 8-1, the command string format might therefore be:

```
#KBL,LP:,/PA:2<DT3:KBIPAR.PAL/PA:1,KBL<CR>
```

8.1.2 Module Linking

The basic Monitor structure for an installation is in fact determined when the assembled modules are converted into load format by the DOS Linker, LINK-11 (see the relevant Programming Manual, DEC-11-ZLDC-D). At this time also, the Keyboard Command Language is completely set-up. The sequence for linking indicated in the following operations is recommended particularly if the output is to paper tape (i.e. Device 'PP') - see next section:

8.1.2.1 Resident Monitor

The modules which together form the permanently resident Monitor section for an installation are linked into one load image in order to complete direct references and initially prepare the Monitor tables (see Section 2.1). Because this image contains the correct settings for the fixed vector locations 40-57, it must be origined at location 0. Hence the command string to LINK-11 for the essentially resident mo-

dules, discussed in Chapter 2, might be as follows:

```
#RMON,LP:<RMON1F,RMON2,RMON3,RMON5,DF,RWN,CLOCK/B:0<CR>
#RMON6/E<CR>
```

The following points should be noted:

1. RMON1, containing the Monitor tables must come first (see Section 2.1.1). Moreover, this currently is system-device dependent, in order to provide the proper first entry in the DDL (see Section 2.1.3). (Hence 3 versions are available: RMON1F for RF11, RMON1C for RC11, RMON1K for RK11)

2. DF (or DC or DK) as the system-device driver must be assembled as noted in the previous section.

3. RWN (the .READ/.WRITE processor) is presently a necessary inclusion for reasons given in Section 3.2.2.2.

4. CLOCK can be omitted if the configuration does not include a line-clock.

5. RMON6, as stated in Section 2.1.4, must come last since it contains the once-only initialization sequence.

As implied by these notes, the order of the intermediate modules is immaterial. Moreover, the list can be extended to include other modules which a particular user wishes to be always in core. That user need merely insert the modules concerned, for the appropriate global links have already been established in the MRT (or DDL for device drivers) - see Sections 2.1.2 and 2.1.3. However, he is referred to the comments on residency in the appropriate descriptive sections in case there are restrictions on such usage (Keyboard Command Modules and Diagnostic Print, especially).

8.1.2.2 The Transient Monitor

It was shown that the program exit module expects the transient Monitor section to be correctly linked with a top at 17400 and relocates it to the upper end of available memory on this basis (see Section 5.5). Section 6.5 also indicated that an embedded version of the .READ/.WRITE processor is a requirement. The resulting command string input in this case is as follows:

```
#TMON,LP:<TMON,RWN/T:17400/E<CR>
```

TMON must come first, to be recognized by MODS.

8.1.2.3 Keyboard Language

The assembly process for the Keyboard Language modules noted in Section 8.1.1 serves to prepare each overlaying module in the right form and to structure the Interpreter Index appropriately. However as pointed out in Section 6.2.3, this Index is dependent upon the fact that the modules concerned are linked together in a fixed sequence. Hence the command string is as follows (an origin at 0 is advised; module KBL being uniquely called by its own EMT code is excluded - see Section 6.2.1):

```
#KBI.LP:<KBI,KBI.DA,KBI.SA,KBI.KB,KBI.OD/B:0<CR>
#KBI.BE,KBI.KI,KBI,TI,KBI,MO,KBI.AS,KBI.DU/E<CR>
```

8.1.2.4 Other Modules

The remaining modules which will normally reside only within the system-device Library must be individually linked since thereafter their only identification is their Title (transmitted by LINK-11 as "Program Name" - see Section 5.1 and Figure 1-2). Again an origin at 0 is recommended as:

```
#INR,LP:<INR/B:0/E<CR>
#RLS,LP:<RLS/B:0/E<CR>
etc.
```

8.2 System Building

Once all the modules have been linked as described in the last section, the actual process of setting up the system can begin. The final objective is a properly established Library on the system-device in the format illustrated at Figure 1-2. In general, the linked modules can be stored within this Library in any order, with the following exceptions:

a. The Monitor ROM Bootstrap assumes that the first module is the permanently resident Monitor image. (see Section 8.2.2)

b. The Program Exit module, XIT, expects the transient Monitor Section to come second (see Section 5.5).

The Library build is effected by a special System Loader, SYSLOD, which is described in Section 8.2.2. This program can accept paper-tape input from device 'PR' and provided that the suggested sequence for linking noted in Section 8.1.2 is followed, the resulting output is immediately ready for SYSLOD. However DECtape is also acceptable. Section 8.2.1 discusses methods of preparing this.

8.2.1 Setting up a System Dectape

It is shown in Section 8.2.2 that the System Loader requires only a successive string of the modules to be incorporated into the system-device Library. On DECtape, this string is a file, MONLIB.SYS, consisting of a concatenation of the individual modules. Basically this can be prepared using the DOS File Utilities Package, PIP-11 (see Programming Manual DEC-11-PIDB-D).

If the linked modules are the only ones with the extension .LDA currently in the user's directory, the concatenation can be effected in one command string. However it should be noted that the DECtape must be identified with the System as user. Hence LOG-IN as [1,1] is necessary. The sequence of commands to PIP-11 might then be:

```
#DT:/ZE<CR>    ;ZERO THE DECTAPE FOR [1,1]
#DT:MONLIB.SYS/FB<*.LDA[User's own code]<CR>
```

The use of formatted binary mode is recommended in order to take advantage of its validity-checking facility (see Section 3.2.2.2). Moreover it avoids excessive gaps between modules in the resulting file, since unformatted binary merely copies everything seen, including EOF padding.

An alternative method is to prepare a series of subsidiary concatenations first, perhaps also on the system-device. This method is useful when other files with .LDA extensions exist within the user's directory (and cannot be deleted).

Also, while it may take longer to enter the individual module names intially, it does allow retries of smaller sections, rather than repeating the whole process should transfer failures occur. Thus a sequence of PIP-11 operations as below is a possibility:

```
#RMONA[1,1]/FB<RMON.LDA,TMON.LDA,KBT.DA<CR>
#RMONB[1,1]/FB<INR.LDA,RLS.LDA,TRA.LDA<CR>
#RMONC[1,1]/FB<etc.
```

This is then followed by console commands to 'KILL', 'FINISH', 'LOGIN [1,1]' and again 'RUN PIP':

```
#DT:/ZE <CR>
#DT:MONLIB.SYS/FB<RMONA,RMONB,RMONC...<CR>
#RMONA,RMONB,RMONC...../DE<CR>
```

The resulting DECtape can now be used by SYSLOD, provided that the latter can be loaded from some alternative device. As a rule, of course, it must be assumed that there is no Monitor in core to help. A paper-tape version can utilize the Paper Tape System Absolute Loader (see DEC-11-GGPC-D). However, since a general purpose bulk-storage device ROM loader is a necessary element in a DOS configuration, for booting the Monitor from the system-device, it seems reasonable that this should also be used to load the SYSLOD program from the DECtape, preferably the one containing the Monitor module file. A Monitor DECtape Set-up program (MODS) is therefore provided.

The usage of MODS is described in the document "Getting DOS on the Air" (DEC-11-SYDD-D). Basically it structures a new DECtape to contain the following items, using either an old DECtape version or paper-tape or both as its input:

a. Core-image of the System Loader as a contiguous file SYSLOD.SYS in blocks 1-37.

b. Special loader for this image in block 0 (where it can be accessed by the ROM Bootstrap).

c. MONLIB.SYS as a linked-file, with forced storage at the front end of the tape for simpler handling.

MODS also has an update facility for modifying an existing MONLIB.SYS file (see Section 8.3).

8.2.2 Loading the System-Device (SYSLOD)

As noted earlier, the System Loader, SYSLOD, is responsible for the building of the system-device Library either from DECtape or paper tape. When required, it also performs general intialization of the system-device as outlined in Section 4.1. On completion, the user may request immediate Monitor start-up. This section describes these operations in detail and also shows how SYSLOD itself is built initially.

8.2.2.1 Preparation of SYSLOD

SYSLOD actually performs its I/O by normal Monitor processes; however by implication the normal Monitor does not effectively exist until SYSLOD can be run. The problem is solved by loading an in-core Monitor version containing all the modules called by SYSLOD. This forms the first part of the load module for SYSLOD on DECtape or paper-tape and is called to initialize itself immediately after entry into memory as described in Section 2.1.4. At this time, though, the system-device cannot be accessed for MRT & DDL set-up. Hence module RMON6 is modified at source level to remove this feature which is contained in a subroutine BG.MDI - see the appropriate listing. It is replaced by an automatic call to SYSLOD, if this has already come from DECtape, or for its loading if it follows the special resident Monitor on the same papertape - see below. The appropriate code sequence is:

```
BG.MDJ: MOV CSA,R0      ;SET ABS.LDR START
        BIC #276,R0     ;(ASSUMED AT TOP OF MEMORY)
        JMP @R0         ;GO TO ABS LDR
BG.MDI: JMP @(PC)+      ;ENTRY POINT - A SWITCH...
        .WORD BG.MDJ    ;OVERWRITTEN BY 'SYSLOD' START
        .END BG.BGN     ;...IF ON DT THRU MODS
```

The whole image - resident Monitor and SYSLOD - is naturally entered at one time if a DECtape is prepared by 'MODS'.

The Monitor module is linked as described in Section 8.1.2.1. Assuming that the modified RMON6 is named RMON6X, the relevant LINK-11 command strings for paper-tape output (recommended for MODS use also) are

```
#PP:,LP:<RMON1F,RMON2,RMON3,RMON5,DF,RWN/B:0<CR>
#INR,RLS,TRA,OPN,CLS,DT,PR,FOP,LUK,CKX,FCL<CR>
#RMON6X/E<CR>
```

As shown, the appropriate system-device driver and RMON1 version are necessary.

SYSLOD itself is also device-dependent; its source is therefore conditionalized and requires the entry of a parameter at assembly, i.e. DC=0 or DK=0 (DF is assumed by default). Thus the PAL-11R command string for SYSLOD and subsequent parameter entry might be:

```
#SYSLOD,LP:,/PA:2<KB:,DT3:SYSLOD<CR>

DC=0<CR>            ;DEFINE SYSTEM DEVICE 'DC'
^C
.END<CR>            ;SIGNAL END OF INPUT
<CR>                ;(SEE SECTION 6.4.3)
```

The linking of SYSLOD is straightforward; however so that error and completion halts may be identified by their addresses, an origin at 30000 is recommended. Advisedly, the linked module should be output to the same paper tape as the resident Monitor module built above (otherwise the automatic Absolute Loader start cannot be effective; the operator must halt the computer and restart at XX7500). Thus the LINK-11 command string is:

```
#PP:,LP:<SYSLOD/B:30000/E<CR>
```

8.2.2.2 SYSLOD Usage

The dual-module thus produced on paper-tape can be used directly or be processed into a DECtape core-image by MODS. In either case the operating procedure is fully detailed in "Getting DOS on the Air" (DEC-11-SYDD-D). On completion, the Monitor is booted into memory as described.

8.2.2.3 SYSLOD Processing

As shown in Section 8.2.2.1, the loading of SYSLOD is preceded by that of the special in-core Monitor which calls SYSLOD as soon as it has initialized itself. The functions performed by SYSLOD are then as follows:

1. Buffer Pointer Initialization - SYSLOD performs its input of the Monitor module file prepared as shown in the previous sections by normal .READ whereas its output while building the system Library uses .TRAN. The latter in fact requires two forms, the modules themselves and the Library Index (see Figure 1-2). Moreover all I/O is double buffered; hence six buffers in all are required and pointers are set accordingly; in particular, the first block

perhaps written is block 0 - set to contain the Monitor Loader (see Section 8.2.2.4).

2. Input Determination - the first SYSLOD halt allows the operator to set two console switches:

   a. Bit 0 = 0 to indicate all input from paper-tape; 1 for a first input from DECtape (MONLIB.SYS under user [1,1] on unit 0).

   b. Bit 15 = 0 to show initial system-device initialization is to be omitted.

   The input dataset Link-block (see Section 3.1.2.1) is adjusted to reflect switch 0. Calls to .INIT and .OPENI for file "MONLIB.SYS" follow and an output dataset, associated with the system-device, is similarly initialized. If switch 15 is 0, the next operation is Library-build (see 6 below).

3. Disk Clear - if the user is starting completely from scratch, i.e. Switch 15 = 1, SYSLOD must first determine the capacity of available system-device surface. On RK11, this is pre-determined (see Section 4.2.2). However as shown in Section 4.2.1, RF11 and RC11 are treated as a continuous surface, regardless of the number of physical platters involved. SYSLOD therefore uses the hardware facility which calls an error for Non-existent Disk by attempting to address the beginning of each potential platter. The resulting information is stored for later use (see next paragraph) and is also set as a control while the whole disk surface is cleared. This is accomplished by an initial transfer at block 0 as mentioned in (1) followed by that of an empty buffer to every third block (3,6,11,etc) up the surface. This is then repeated over two more passes to clear the intermediate blocks.

4. Bit-map Initialization - as indicated in Section 4.1.3, the available system-device surface is controlled by an appropriate number of bit-map segments and these are stored at the top end. SYSLOD writes out the necessary preamble to each segment and sets bits to 1 as follows:

   a. On the first map, the bits for all blocks up to the Hardware Protect Line, illustrated at Figure 4-9.

b. On the last map, all bits corresponding to non-existent blocks

5. Directory Structure - SYSLOD prepares the basic MFD and System UFD described in Section 4.1.2 as follows:

   a. MFD#1 - in block #1, containing a link to MFD#2, Interleave Factor and pointers to the bit-map blocks.

   b. MFD#2 - in the block preceding the bit-maps (see Figure 4-9), containing UFD Index entries for the System (user [1,1]) with a directory starting at block #2 and for a general user [200,200] with no current UFD pointer.

   c. System UFD#1 - in block #2, with a link to block #3 as UFD#2 and with only one file established - a contiguous one, 'MONLIB', starting at block #4 and ending at the Hardware Protect Line -

6. Library Build - SYSLOD expects to start the System Library Index in block #4 and the first module in block #11 (intermediate blocks being reserved for the permanent MRT copy - see Section 2.1.2). The COMD for the first module is read and its information is transferred as required into the Library Index buffer (see Section 5.1 and Figure 1-2). At this time 'EMT Code' is not known and is filled in later by Monitor initialization (see Section 2.1.4). The rest of the module is converted into its core image, due allowance being given to the Program Load Address at the start of each text block to ensure the requisite gaps. After conversion the image is written out to successive system-device blocks until the terminal Transfer Block at the end of the linked image is seen. At this point, the last block for the module is output and the Index buffer is checked to verify further entry space. If any, the next module is processed in the same way, starting from the next system-device block. When the Index block is full, the next block is linked to it (in the first word as in a normal linked file) and it is written out. The next module is then indexed in the new block and starts from the following block, thus producing the format illustrated at Figure 1-2.

7. End of Input - if at any time the original input terminates, SYSLOD halts the computer after closing and releasing the input dataset. As shown in "Get-

ting DOS on the Air", the user then can indicate further paper-tape input - even after DECtape, in order to enter modifications or extensions - by setting switch 0 to 0 and pressing the CONTINUE switch. SYSLOD reinitializes the input dataset for 'PR' and processing is resumed as shown under (6). When SYSLOD however detects a 1 in switch 0, it assumes no further input. The last Library Index block is then written out and the output dataset is released. A call to the internal copy of the Monitor Loader follows - see next section.

It will be noted that SYSLOD does not attempt to check the sequence of the modules in the input file. Hence the user must himself ensure that the first two are RMON and TMON - as specified in the introduction to Section 8.2. Moreover there is no restriction on there being only one copy of a particular module. Therefore as stated in Section 2.1.4, replacement modules can merely be added at the end of the input and these will be used rather than the originals during Monitor initialization. The user of course must take care not to go beyond the available area defined by the Hardware Protect Line. At the end of input halt, R0 shows the next block to be used - and thus this is the value displayed as "Data" in a valid halt. Errors in SYSLOD operation are as described in "Getting DOS on the Air".

8.2.2.4 Monitor Booting

In the previous section, it was shown that SYSLOD stores a Monitor Loader in Block 0 of the system-device. The ROM Bootstrap reads this block (see outline in Section 6.5.2.2) and starts automatically at location 0. The initial sequence executed by the Loader is one which moves the section performing the Monitor load into the memory area starting at 37000, in order to leave all lower core for the module to be entered.

The load sequence is then entered. This first requests a direct hardware read from block #4 (Library Index #1 - see last section). Provided this transfer is satisfactory, the starting block for the first module is extracted from the Index (see Figure 1-2), and this combined with the entry for size is set into the hardware Registers for a transfer of the module - assumed RMON as noted earlier - into memory starting at location 0. If again, this is completed without error, the Index-stored starting address is set into the PC and Monitor initialization commences.

As indicated, the loader must drive the device because no Monitor theoretically exists. Hence the routine, like SYS-

LOD is device-dependent: it must perform its own conversions from "System-block" to actual device block (see Sections 4.2.1 and 4.2.2); and it does not use interrupts but depends on flag-testing both to determine completion and to recognize error-detection. In the latter case, the computer is halted and the user must try the Monitor Root afresh.

8.3 Monitor Modification

By no means has the DOS Monitor so far described in this manual reached its final stages of development; moreover it is expected that some users will wish to adapt its facilities to their own specific needs. Monitor modification, as indicated earlier, is relatively simple owing to its modular structure. The aim of this section therefore is to assist those responsible for producing the new or replacement modules required, by summarizing their more significant characteristics and by illustrating some of the practices found useful during the preparation and check-out of the existing Monitor.

Basically the modules fall into one of four categories, which are individually discussed in sections 8.3.1 to 8.3.4:

a. Resident Monitor

b. Program services

c. Device drivers

d. Keyboard commands

However it should again be noted that because all the modules, with the exception of those in category 1, may be system-device or core-resident, they must observe the following common rules:

1. Position-independence is essential.

2. The facilities of the resident Monitor may be accessed but only through the fixed vector locations 40-56 - see Section 2.1.

3. Registers or stack may be freely used internally to a module or between modules as long as they are restored to the original program state at any time or for whatever reason control is returned to that program.

4. Impure code, including fixed storage areas, is permissible only if some form of protection against re-entrancy is actually provided, such as the forced .WAIT on incomplete I/O (see Section 2.2.1) or driver-queuing (see Section 3.1.2.4) or is implied by the imposition of some restriction, e.g. non-resident use only.

5. It may be assumed that the user program takes care of its own re-entrancy - in particular preventing corruption of prescribed data-blocks, passed to the Monitor for access or storage, until some requested service has been satisfied.

The new modules should all be prepared for check-out as described in Section 8.1. As far as possible, initial testing in-core is advised - the following sections show how. For the final stages which require access from the system-device, the new modules should be added at the end of the Library through SYSLOD (RMON & TMON excepted), if need be without destruction of the other content of the system-device - see Section 8.2. They can remain so stored for normal usage (unless MODS is used to update an appropriate system DECtape) until a complete monitor rebuild becomes necessary. (1)

8.3.1 Resident Monitor Modules

Modules which are known to be always core-residnet are not, of course, so obliged to be position-independent, since linking assures correct address evaluation, provided that any necessary global references are indicated. Furthermore they are rather easier to check out. Their residency, however, does make re-entrancy a matter of greater concern and if external access to them is required, the necessary linkage must be provided - perhaps through the System Vector Table - see Section 2.1.3.

There is also the problem of loading the resident Monitor, after it has been relinked to include the new module. For initial check-out, this can be temporarily accomplished if the linked image is on paper-tape through the Paper-tape System Absolute Loader (see DEC-11-GGPC-D). In this case, reloading by the same means must follow any console FINISH command - see Section 6.5.2.2. For normal operation,

----

1. Before general release, updating the version number printed as Monitor identification after booting is suggested as a necessary practice to reflect any revisions introduced (module RMON6).

however, the resident Monitor image in the Library on the system-device must be replaced by the new version through SYSLOD. As noted in Section 8.2, it must be the first module. Thus users who make DECtape the SYSLOD input must prepare a new file through MODS in update mode as shown in "Getting DOS on the Air" (1).
Paper-tape users should physically examine the first MONLIB tape to find the terminal Transfer Block for the original RMON module (see Section 5.1) and tear the tape immediately after it. SYSLOD is then executed with the new resident Monitor as its first input, the second half of the original tape entered next and thereafter as normally. While the new version is being tested, ODT-11R (see DEC-11-OODA-D) can be used in the usual way.

8.3.2 Program Services

As shown in Section 1.3.1, all program services are called by EMT; relevant data may be passed on the stack and this must be removed before the program is recalled. The stack may also be used for data return. For new or revised modules providing these services, the following points are particularly relevant:

1. A new module must be allocated an EMT call code, in accordance with the conventions given in Sections 2.2 and 2.3 (spare slots being already provided), i.e.:

   Code 0-27 = I/O Request
   Codes 30-37 = Emergency service requiring KSR usage
   Codes 40-77 = General service using MSR.

2. This new module must also comply with the naming convention at Section 1.3.2.1.

3. Call state is as described for transfer of control from the EMT Handler - see Section 2.2.1.

----

1. This facility is not available however for users who have DECtape but no paper-tape reader other than that on the ASR-33 Teletype. The driver for this device does not permit binary data. Thus although MODS is especially written to get around this, LINK-11 or PIP-11 cannot be used to produce its necessary input. However, this restriction applies only to modifications to RMON and TMON; other revisions can be effected as footnoted in the next section.

4. As potential users of Swap Buffers, the rules detailed in Section 2.3.5 must be observed.

5. For I/O services:

   a. Driver calls must be made through the Queue Manager S.CDB and obey its conventions - see Section 3.1.2.4.

   b. If control is not returned to the user program during the transfer, .WAIT should be called, perhaps with link-block simulation, as used in the .INIT processor (see Section 3.2.1.1) or the File-management modules (see Sections 4.4 through 4.6).

6. Modules which cannot satisfy their purpose within a single Swap Buffer frame may possibly benefit from one of the overlaying techniques discussed in either the File-management operations (see Section 4.3) or the Keyboard Language (see Section 6.2). However the limitations in both cases should be noted.

The new module should be assembled as shown in Section 8.1.1. For the initial stages of check-out, it should be linked and loaded with its test program. Before testing begins, however, the relevant MRT slot (see Section 2.1.2) must be modified to show the module's in-core start address (Actual Start+2 - see Section 2.3.1). Assuming ODT-11R is also loaded with the test program, a simple way of accessing the slot required is by means of the SVT (see Section 2.1.1) as follows:

```
ODT-11R V002A              ;ODT CALLED AFTER 'GET'

40/000400<CR>              ;SVT START - ADD 46
446/000466<CR>             ;MRT START - ADD CODE*2
502/000135 XXXX <CR>       ;ENTER NEW .INIT, SAY
```

Thereafter testing under ODT-11R can be carried out in the normal way as for any in-core program (although the user should avoid setting breakpoints actually on EMT instructions). There is no need to restore the MRT as this is automatically refreshed, when the test is KILLED.

Once the module is apparently satisfactory when in-core, it can be added at the end of the Library on the system-device by a fresh SYSLOD, as noted earlier, for check-out under swapping conditions (1).
Another step may be necessary, though, if the module is completely new, i.e. It uses a previously unallocated EMT call code. The MRT must be modified, in RMON1, to show the module's global name and the Monitor initialization routine's reference sheet in RMON6 must be similarly amended. (see Section 2.1). This means that a new resident Monitor must be linked and also stored in the Library as discussed in the previous section.

----
1. Although the user who has only ASR-33 paper tape facilities is generally at a disadvantage at present, as footnoted in the previous section, he can add his new module to the MONLIB.SYS file on DECtape under PIP as follows and then SYSLOD in the normal way (provided that the module is only a replacement for an existing one:

```
$LOGIN 1,1<CR>
$RUN PIP<CR>

PIP-11 V004A

#MONLIB.SYS/FB<DT:MONLIB.SYS,DF:MODULE.LDA[UIC]<CR>
#DT:MONLIB.SYS/PR:0<CR>
#DT:MONLIB.SYS/DE<CR>
#DT:/FB<MONLIB.SYS<CR>
```

Checking the module now that its presence in memory is ephemeral is of course somewhat more difficult. The technique found most satisfactory is to breakpoint the test program just prior to the EMT call and then the SAM routine at the instruction which actually passes control to the module when loaded in the Swap Buffer (currently RMON2+300) e.g. the ODT-11R sequence might be:

```
30000;1R          ;MARK PROGRAM START
40/000400<CR>     ;SVT START = ADD 50
454/000646<CR>    ;DDL START = ADD 6
652/000766<CR>    ;DDL END = EMT START
766;2R            ;SET RELOCATION FACTOR
1,XXX;B           ;BREAKPOINT THE TEST PROG.
2,300;B           ;...& SAM
1,0;G             ;START THE TEST
B0;1,XXX          ;TEST BREAK
;P                ;CONTINUE TO SAM BREAK
B1;2,000300
;1S               ;SINGLE STEP TO...
;P                ;...MODULE START
B8;2,000462
2,460;3R          ;MARK MSB. START
502/001001<CR>    ;PERHAPS CHECK MRT
3,-2/001001<CR>   ;...AGAINST MODULE IN MSB
;S                ;...(SEE SECTION 2.3.1)
3,YYY;B           ;BREAKPOINT THE MODULE
....              ;... & CONTINUE
```

Again the module can be tested under ODT-11R control. One word of warning however: all breakpoints within the module must be removed before it returns control to the test program if there is any possibility that a different routine may be brought into the Swap Buffer to replace it.

8.3.3 Device Drivers

The format and operations expected of a device-driver were discussed in Section 3.3, and this referred to the more detailed description given as Appendix G of the Programmer's Handbook. Once prepared, the new driver can be assembled and linked for normal usage as shown in Section 8.1.

As with program request modules covered in the last section, the driver should first be linked and loaded with its test routine for in-core checkout. In this case the DDL must be modified to reflect its presence (see Section 2.1.3). For this, the use of an existing entry for a device not actually in the configuration is recommended, e.g. the ODT-11R action might be:

```
40/000400<CR>                      ;SVT START - ADD 50
450/000646<CR>                     ;DDL START - ADD 20
666/063320  ;X  PR <CR>            ;ADD 10
676/06320  0;X  PP <CR>            ;ADD 10
706/046600;X  LP  DV <LF>          ;REPLACE
710000000   XXXXXX<CR>             ;ENTER CORE ADDRESS
```

The user must appropriately set the interrupt vectors at this time, since .INIT only does this for drivers brought from the Library (see Section 3.2.1.1).

For final checkout when brought from the system-device, the new driver may also be added to the end of the Library by entering it from papertape through SYSLOD (see Section 8.2). At the same time, the DDL in RMON1 must be updated if necessary, in much the same way as the MRT, as indicated in the last Section. No modification of RMON6 however is needed. ODT-11R control for the check-out is a little simpler; the driver's core location can be readily determined immediately after the dataset using it has been .INITED by examining the appropriate DDL slot. Driver break points can of course remain until the dataset is .RLSEd.

8.3.4 Keyboard Commands

The implementation of a new Keyboard Commands depends upon their intended usage. As shown in Chapter 6, if a command can only be given when no program is loaded, it is processed by a routine embedded in the transient Monitor. On the other hand, if it can be given at any time, its processing routine must be incorporated into the Keyboard Interpreter module for swapping in as required. In neither case is the procedure simple, as the following paragraphs will show; Monitor extension in this area by the general user is not therefore advised.

8.3.4.1 New TMON commands

The inclusion of new commands which can only be effected when TMON has control of the machine are fairly simple to prepare since they can assume all normal Monitor facilites and can use the provisions of the TMON general processor described in Section 6.5. Moreover it is only TMON itself that need be modified. However check-out is a problem, since the use of ODT-11R implies a loaded program and by definition, this does not exist. Various techniques were therefore tried during the development of the current version of TMON - perhaps the simplest is to link ODT-11R to a TMON module which contains HALT as its first instruction (1) and store the resulting linked image in the system-device Library as described in Sections 8.1 and 8.2 (2). The HALT then allows the user to restart the machine under ODT-11R control through the console switches (starting at ODT-11R load-point + 172). Check-out operations may then proceed as normally except that a loss of debugging control can only be rectified manually, i.e. The console ODT command is not available. A new SYSLOD without ODT-11R linked must of course follow successful testing.

8.3.4.2 Other Commands

As noted in the introduction to Section 8.3.4, new commands which can be entered by the operator at any time must be processed within the Keyboard Interpreter (see Section 6.2.3). Not only therefore, must they in general observe the points listed at the start of Section 8.3.2, since they also effectively result in an EMT call; they should also comply with the conventions discussed in Section 6.2.5. Moreover, the following considerations are also relevant:

a. Module KBI must be modified to include any completely new command in its index as indicated in the comments on the appropriate listing.

----

1. It is not expected that TMON expansion is a likely requirement of a user who has an 8K configuration. The process described is almost certainly impossible in this amount of available memory.

----

2. In practice, the use of MODS to update a system DECtape is advised. Paper tape users (device 'PR' that is) can use the same technique as that described for RMON in Section 8.3.1 - in this case, the second module of the first MONLIB tape is physically replaced.

b. The module itself must handle its own operations - it cannot depend upon other Monitor facilities (see Section 6.2.4).

c. It follows from (b) that overlaying of the available 128-word KSB is especially probable. Reference to Section 6.2.3 and the listing for a similar existing command (ASSIGN perhaps) is strongly recommended, (noting particularly its structure for ensuring correct system device-storage when assembled with the correct parameter tape - see Section 8.1)

Checkout of the new command is also complex. As noted in Section 6.2.5, the Keyboard Interpreter and its associated modules can never be core-resident; hence the composite linked-image must be built from the start (see Section 8.1) and ODT-11R control can only be maintained basically through the SAM entry into the module as illustrated in Section 8.2.

Even this requires some effort, because of the overlaying normally effected for the language. The following sequence depicts this (assuming ODT-11R has been loaded as a running program and a breakpoint has been set in SAM as in the example in Section 8.2):

```
                      ;USER ENTERS CTL/C
B1;2,000300           ;FIRST KBL CALL
;P
^B1,2,000300          ;'^' ECHO
;P
CB1;2,000300          ;'C' ECHO
;P
B1;2,000300           ;<CR>ECHO
;P

B1;2,200300           ;<LF> ECHO
;P
.B1;2,000300          ;'.' ECHO
;P                    ;USERS ENTERS 'D'
B1;2,000300
;P
DB1;2,000300          ;'D' ECHO
......                ;FOLLOWED BY COMMAND
;P                    ;...(2 BREAKS PER INPUT)
B1;2,000300           ;TERMINAL <CR>ECHO
;P

B1;2,000300           ;<LF>ECHO
;1S                   ;SINGLE STEP TO...
B8;2,750              ;...GET KSB.START
2,746;3R              ;MARK IT
446/466<CR>           ;MRT START - ADD 2*31
550/000535<CR>        ;CHECK KBL ENTRY
554/001021<CR>        ;CHECK KBI ENTRY
3,-2/000535<CR>       ;CONFIRM KBL IN KSB
;S                    ;...& CONTINUE
;P
/001021<CR>           ;CONFIRM KBI IN KSB
;P
/001201<CR>           ;CONFIRM CMD PROCESSOR
3,XXX;B               ;SET FIRST B/P.
;P                    ;... & CONTINUE
```

Now the appropriate processor is in the KSB, debugging can continue as before. Particular care is obviously needed, not to lose control; the comments on breakpoint removal in Section 8.2 are again relevant. Trapping the SAM entry into a further overlay can be left to occur naturally if necessary, e.g.

```
;B                  ;REMOVE ALL B/P
2,300;B             ;RESET AT SAM ENTRY
;P
B0;2,000300         ;OVERLAY NOW IN ...
3,-2/001205<CR>     ;... THOUGH CHECK
3,YYY;B             ;SET NEW FIRST B/P
;P                  ;... & CONTINUE
```

MODULES PRESENT IN DOS-11 MONITOR (V4A)
(EMT Call Codes in parentheses)

1. Basic Resident Monitor:-

RMONIF (RF-11) / RMONIC (RC-11)/RMONIK(RK-11)
RMON2
RMON3
RMON5
CLOCK
RWN. (2/4)
DF (RF-11) / DC (RC-11)/DK (RK-11)
RMON6

2. Principal I/O Routines:-

INR (6)	STT (13)	REN (2Ø)
RLS (7)	DIR (14)	DEL (21)
TRA (1Ø)	ALO (15)	APP (22)
BLO (11)	OPN (16)	PRO (24)
SPC (12)	CLS (17)	

3. Subsidiary File-management Routines:-

FOP (43)	GMA (5Ø)	DCN (54)
FCR (44)	CBA (51)	AP2 (55)
FCL (45)	CKX (52)	MTO (63)
LUK (46)	DLN (53)	GNM (34)
LBA (47)		

4. Other Monitor Services:-

GUT (41)	CSM (57)	LDR (61)
CVT (42)	XIT (6Ø)	LD2 (62)
CSX (56)		

5. Keyboard Services:-

KBL (31)	KBI.EC	KBI.MO
EDP (32)	KBI.OD	KBI.AS
KBI (33)	KBI.BE	KBI.DU
KBI.DA	KBI.KI	TMON
KBI.SA	KBI.TI	

6. Device Drivers:-

2 from DFX/DCX/DKX

KB	LP	CR
PR	DT	MT
PP		

Figure 8-1

CHAPTER 9

SYSTEM FILE FORMATS

9.1 Object Module Format

An object module is the fundamental unit of input to LINK-11. Furthermore, each input file specified in the command string may contain multiple object modules. Each object module contains several functional parts. Two required parts are the global symbol directory (GSD) and text blocks (TXT). At least one relocation directory (RLD) is also required. Normally there are many RLD's present. Each RLD is associated solely with the text block which immediately precedes it. The optional fourth part (not yet implemented) is the internal symbol directory (ISD). The end of the object module is specified by a block containing only the "end-of-module" command. Refer to Figure 9-1.

9.1.1 Object Module's Contents

9.1.1.1 Global Symbol Directory

The GSD holds all information necessary to assign absolute addresses to all global symbols, to cause the proper library searches and to create the load module's COMD.

The GSD contains three types of global symbols (in addition to other types of information specified below):

1. Program section names
2. Definitions for symbols which have been declared to be entry symbols (entry points)
3. References to symbols which have been declared to be external (external references)

The GSD must be constructed such that once a program section's name has been declared, all the entry symbol definitions within that section must be declared. The declaration of a reference to an external symbol may appear anywhere in the GSD. Upon completing a section's declarations, the next program section's name may be declared.

The GSD will generally consist of several formatted binary lines. Each line will start with a command indicating that it is a part of the GSD. The first line of the GSD will have the modules's name, packed in .RAD50 form, immediately following the GSD command. The following lines will have the GSD command and a continuation of the GSD data. The GSD is terminated by a line containing only the GSD termination command.

Every object module must begin with a GSD line which contains at least the object module's name. The remainder of the GSD, however, may be anywhere in the module. The linker will ignore all non-GSD lines while it is gathering the GSD. Refer to figure 9-2.

The GSD is terminated by the end GSD object command.

The entries in the GSD adhere to the following conventions:

1. Every object module contains exactly one GSD.

2. Every symbol (name) must be presented as a double word. This double word represents a packed .RAD50 expansion of the symbol's six ascii characters. The first word holds the packed representation of the symbol's first three characters.

3. Every entry must possess a word containing flag information. The bit assignments (individual flags) must conform to the following specifications:

Bit #	Reset/Set
3	undefined/defined
6	internal/global
5	absolute/relocatable
0,1,2,4,7	not used
8-15	coded octally as follows: 0=object module name 1=program section name (.ASECT or .CSECT assembler directives) 2=internal symbol table name (not yet implemented) 3=transfer address specification 4=symbol declaraction 5=local section name 6=version identification

   Note that the high order byte of the flags is used as a command to identify the type of entry. The low order byte is used to specify particular qualities of the entry.

4. Every program section and local section entry must declare the total dimension (in bytes) of the entire section.

5. Every relocatable entry symbol definition must state its definition, relative to the base of the section.

6. All entry symbols of a section must follow the declaration of the section's name. However, external symbol references may occur anywhere in the GSD.

7. The size of a section is defined as follows:

   a. The absolute section has a size of 0.

   b. A control section's size is one plus the largest relocatable PC value assigned during assembly.

8. The transfer address is specified in the GSD so that the linker can include it in the COMD of the load module. If no transfer address is specified in the GSD, the linker will assume an address of 000001. The first transfer address specified will be kept. The expression on the .END statement is resolved down to a displacement from a program section (or local section) name.

   Example:

```
        .CSECT          NAME
   A:   0
   B:   0
   C:   MOV   MUMBLE,SP
        .
        .
        .
        .
        .END C          ;THE RELOCATABLE TRANSFER
                        ;ADDRESS
                        ;IS DEFINED TO BE THE VALUE OF
                        ;THE BASE OF CONTROL SECTION
                        ;"NAME" PLUS 4.
```

   The linker places the finalized absolute transfer address in the COMD and in the TRA block of the load module.

   Refer to Figure 9-3 for an illustration of the GSD without the Blocking Structure.

9.1.1.2 Text Block

Upon input to LINK-11, the text blocks normally contain bytes and/or words which have yet to be determined. That is, positions remain within the text block whose contents must be determined by the processes of relocation and linking. The contents of these positions are totally specified by the relocation directory which follows the text block. If a text block does not need modification, then an RLD need not follow. Thus, multiple text blocks may occur contiguously in the module. Upon input to the absolute loader, the load module's binary blocks contain solely absolute binary load data and absolute load addresses.

Each text block must begin with the object command which declares this block to be TXT. This command is followed by a word which states the memory location where the first data byte of this text block is to be placed. This word is followed by the body of the text block. Refer to Figure 9-4.

9.1.1.3 Relocation Directory

Relocation directories contain the necessary object commands and their supportive arguments required for modifying (relocating and/or linking) the preceeding text block. Refer to the definition of the object language, section 9.1.2. Refer to Figure 9-5 for a RLD block.

Each RLD may contain load address modification commands and/or text modification commands. Load address commands may not be followed by text modification commands in the same block (line).

Each object module must have at least one RLD which preceeds the first TXT block. These RLD's must contain load address modification commands which start the linker at the proper place.

9.1.1.4 Internal Symbol Directory

(Not yet implemented)

9.1.2 Object Language

The object language is the set of all commands which indicate the type of a block and which appear in an RLD to control the linking and relocating process.

The general structure of all object commands is:

1. A command is a byte quantity
2. The sign bit indicates a byte operand (if set)
3. Potentially 128 commands

The object language contains several general types of commands:

Type I, Block Type Declarative commands — Code (Octal)

		Code (Octal)
a.	Declare the GSD	1
b.	Declare a TXT block	3
c.	Declare a RLD block	4
d.	Declare the ISD	5
e.	Declare the modules's end	6
f.	Declare GSD termination	2
g.	Declare a library	7
h.	Declare the library directory end	10
i.	Declare a core library	11
j.	Declare a core library end	12

Type II. Text Modification Commands

This group of commands appear in the RLD and perform the relocating and/or linking function. They operate upon specific bytes or words in the TXT block and therefore require a method for referencing quantities in the associated text block. Thus, the second byte of each of the following commands is a byte which specifies the displacement from the base of the text block to the referenced text data byte (or word). The base of the text block is the location of the first byte in the block; that is, it is a displacement from the TXT command.

1. Internal Relocation (Octal Code 1)

Add the current program section's base to the specified constant and place the result where indicated.

```
CMD (Byte)
Relative Reference (Byte)
Constant (Word)
```

This command relocates a direct pointer to an internal relocatable symbol.

Example:

```
A:  MOV   #A,X0 ; THE RELATIVE DEFINITION
                ; OF "A" APPEARS AS THE
                ; CONSTANT IN THE COMMAND
```

2. Global Relocation (Octal Code 2)

Place the value of the specified global symbol where indicated.

```
CMD (Byte)
Relative Reference (Byte)
Global Symbol (2 Words, Packed .RAD50)
```

This command generates a direct pointer to an external symbol.

Example:

```
                ; ASSUME "A" IS
                ; A GLOBAL
    MOV   #A,X0 ; THE NAME "A"
                ; APPEARS IN
                ; THE COMMAND
```

3. Internal Displaced Relocation (Octal Code 3)

Calculate the displacement from the position of the current location counter (plus 2) to the specified absolute address. Place the result where indicated.

CMD (Byte)
Relative Reference (Byte)
Constant (Word)

Example:

```
        CLR   177550 ; A 177550
                     ; APPEARS AS
                     ; THE CONSTANT
                     ; IN THIS COMMAND
```

Note that this type of relocation occurs only when there is a reference to an absolute symbol from a relocatable section.

4. Global Displaced Relocation (Octal Code 4)

Calculate the displacement from the position of the current load location counter (plus 2) to the specified global. Place the result where indicated.

CMD (Byte)
Relative Reference (Byte)
Global Symbol (2 words, packed .RAD50)

Example:

```
                    ; ASSUME THAT
                    ; "A" IS A GLOBAL
        MOV   A,%0  ; THE NAME "A"
                    ; APPEARS IN
                    ; THE COMMAND
```

5. Global Additive Relocation (Octal Code 5)

Add the value of the specified global symbol to the specified constant and place the result where indicated.

```
CMD  (Byte)
Relative Reference  (Byte)
Global Symbol  (2 Words, Packed .RAD50)
Constant  (Word)
```

Example:

```
                   ; ASSUME THAT
                   ; A IS A GLOBAL
MOV   #A+6,X0      ; THE 6
                   ; APPEARS AS THE
                   ; CONSTANT
```

6. Global Additive Displaced Relocation (Octal Code 6)

Calculate the displacement from the position of the current location counter (plus 2) to the address specifeid to be the sum of the specified constant and the specified global symbol and then place the result where indicated.

```
CMD  (Byte)
Relative Reference  (Byte)
Global Symbol  (2 Words, Packed .RAD50)
Constant  (Word)
```

Example:

```
               ; ASSUME THAT
               ; A IS A GLOBAL
CLR   A+6      ; THE 6
               ; APPEARS AS
               ; THE CONSTANT
```

Example: (FORTRAN)

```
COMMON/LABEL/L,N,M/
M=0
```

These statements would cause the following instruction to be generated:

```
CLR   LABEL+4 ; LABEL IS A
              ; GLOBAL
```

7. Set Program Limits (Octal Code 11)

This command (generated by the .LIMIT assembler directive) causes two words in the TXT block to be modified. The first word is set to the lowest relocated address of the program. The second word is set to the highest relocated address +1; that is, the first free location following the relocated code. The command consists of just one word.

CMD (Byte)
Relative Reference (Byte)

Note that both words to be modified must appear in the same TXT block.

8. .CSECT Text Modification Commands

Four .CSECT text modification commands have been added to the assembler's object language repertoire to accommodate .CSECT relocation:

		OCTAL COMMAND
1.	.CSECT relocation	12
2.	.CSECT displaced relocation	14
3.	.CSECT additive relocation	15
4.	.CSECT additive displaced relocation	16

The LINKER currently maps these commands into the corresponding .GLOBL text modification commands. However, if and when the LINKER is modified to distinguish between .GLOBL and .CSECT names it will be possible (without modification to the assembler) for the user to have .CSECT's and .GLOBL's with the same name.

Type III, Location Counter Modification Commands

These commands appear in the RLD and control the setting of the LINKER's current location counter; that is, they control where the absolute load data, within the next text block, is to be placed into memory.

The object module must have at least one RLD, containing the proper location counter setting and/or modification commands, preceding the first TXT block.

These commands, which must be last in an RLD, set the load address for the next TXT block read.

1. Location counter definition (Octal Code 7)

   Set the current location counter equal to the sum of the specified program section's base address and the specified constant.

   ```
   CMD  (Byte) followed by a null byte
   Program Section's Name
   (2 Words, Packed .RAD50)
   Constant  (Word)
   ```

   This command is generated whenever a .ASECT, .LOCAL, or .CSECT directive is used to initiate or continue a program section.

2. Location Counter Modification (Octal Code 10)

   Define the Current Location Counter to be the sum of the current program section's base address and the following constant.

   ```
   CMD  (Byte) followed by a null byte
   Constant (word)
   ```

   Example:

   ```
   .=100       ; 100 appears as the constant
   .=.+1000    ; The value of '.+1000'
               ; appears as the constant.
   ```

9.2 Load Module Format

The normal output of the linker is a load module which may be loaded and run. The Phase I implementation will produce only a load module. Phase II will include the option of producing a load-time relocatable module.

9.2.1 Load Module Contents

A load module consists of formatted binary lines holding absolute load addresses and object data as specified for the paper tape system absolute loader. The first few words of data will be the communications directory (COMD) and will have an absolute load address equal to the lowest relocated address of the program. The absolute loader will load the COMD at the specified address but then the program will overlay the COMD. The disk monitor will expect the COMD and will load it where the monitor wants it. The end of the load module will be indicated by a TRA block; that is, a line containing only a load address. The byte count in the formatted binary line will be 6 on this line. On all other lines the byte count will be larger than 6. The load module lines, the linker output, have the following format:

1st Byte:	Low order byte mode (001)
2nd Byte:	High order byte status (000)
3rd and 4th Bytes:	Actual byte count (from 1st thru last byte, not including checksum)
5th and 6th Bytes:	Starting absolute load address of block (line)
4th thru last byte:	Code or data to be loaded

There is also an extra byte of checksum.

Refer to figure 5-1.

9.2.1.1 COMD Contents

The COMD holds the following information to be passed to the monitor:

1. An indication of whether the module is load-time relocatable.

2. The program's transfer address.

3. The lowest address loaded by the program (not including absolute code). Under RSX, this address is the lowest address to be used for STACK space.

4. The size (in bytes) of the program. Under RSX, this size includes the STACK space (just below the program) and the real-time header (just above the program).

5. ODT's transfer address (0 if ODT not included). Ignored under RSX.

6. A list of monitor routines required to be loaded along with the module (none under RSX)

7. The load module name.

8. The version identification

The COMD appears as binary data to be loaded at the lowest relocated address of the program. The monitor loader will examine the COMD, process it and call the proper routine to load the program.

The COMD consists of possibly several formatted binary lines. Each line has a load address (ignored by the monitor loader) followed by sections of information. Each section begins with an identifying word indicating the type of section and the length of the section as follows:

Low Byte/High Byte	Meaning
0 / ---	End of COMD
1 / N	N words of general information follows.
2 / N	N monitor routine requests follow (0 under RSX)

Thus the COMD has one or more sections of general information followed by one or more sections of monitor routine requests followed by a section terminating the COMD. The order of the general information is:

WORD NUMBER	CONTENTS
1	Lowest address loaded by program.
2	Program size (in bytes) this is the sum of the sizes of the relocatable sections.
3	Program transfer address
4	ODT's transfer address (0 if ODT not loaded)
5	0=absoluete 1=load time relocatable
6-7	load module name (2 words, .RAD50)
10-11	load module version identification (2 words, .RAD50). This data is acquired from the version identification of the first object module linked.

The monitor routine requests are in the form of words containing the routine's number as specified in the monitor library.

Refer to figure 5-1.

9.3 Object Module Library Format

An object module library that exists as a formatted binary linked file is a sequence of object modules preceded by a directory of the contents.

A library is a useful file for three reasons:

1. It saves having separate directory entries in a UFD for single object modules.

2. It expedites the linking process.

3. It allows for standardization and controlled updating of commonly used routine, e.g. Fortran cosine routine.

The first data byte of the first line of an object module library contains the code 7 to indicate the beginning of a library file.

This line is followed by x lines of directory information terminated by the end directory line (code 10).

Now appear the object modules in the same order as their entries in the directory. Refer to figures 9-6 and 9-7.

9.3.1 Directory Contents

The directory contains one entry per object module in the same relative position as its associated object module. An entry is essentially the object modules GSD (refer to figures 9-2 and 9-3) with two additional requirements to allow for optimum searching of the directory:

1. The GSD as produced by the assembler is re-ordered so that all external .GLOBL's appear after all the .GLOBL entry points; that is, all the external .GLOBL's which appear with the Absolute Program Section Name are moved to just before the end GSD declaration in the GSD.

2. Lines may contain up to 93 decimal data words. If an object module has no named .CSECT's or .LOCAL's, this buffer will allow for 20 decimal entry points (80 words) and 13 words for

GSD declaration	(1)
Object module name	(4)
Absolute Program Section Name	(4)
Unnamed Relocatable Program Section Name	(4)

   in the first line.

9.3.2 Object Module Contents

An object module in an object module library is identical to its stand alone version (refer to section 9.1) except that the GSD is reduced to its first and last lines. Refer to figure 9-2. The first line contains the name of the object module for verification with its directory entry and the last line indicates the end of the GSD.

9.4 Load Module Library Format (not yet implemented)

A load module library, that exists as a formatted binary linked file, is a sequence of load modules preceded by a directory of the contents.

The first data byte of the first line of a load module library contains the code 13 (octal) to indicate the beginning of a load module library file.

This line is followed by x lines of directory information terminated by the end directory line (code 14).

Now appear the load modules in the same order as their entries in the directory. Refer to figures 9-8 and 9-9.

9.4.1 Directory Contents

The directory contains one entry per load module in the same relative position as its associated load module. An entry is the first line of the load module (refer to figure 5-1) which is the first line of its Communications Directory (COMD).

9.4.2 Load Module Contents

A load module in a library is identical to its stand alone version (refer to figure 5-1).

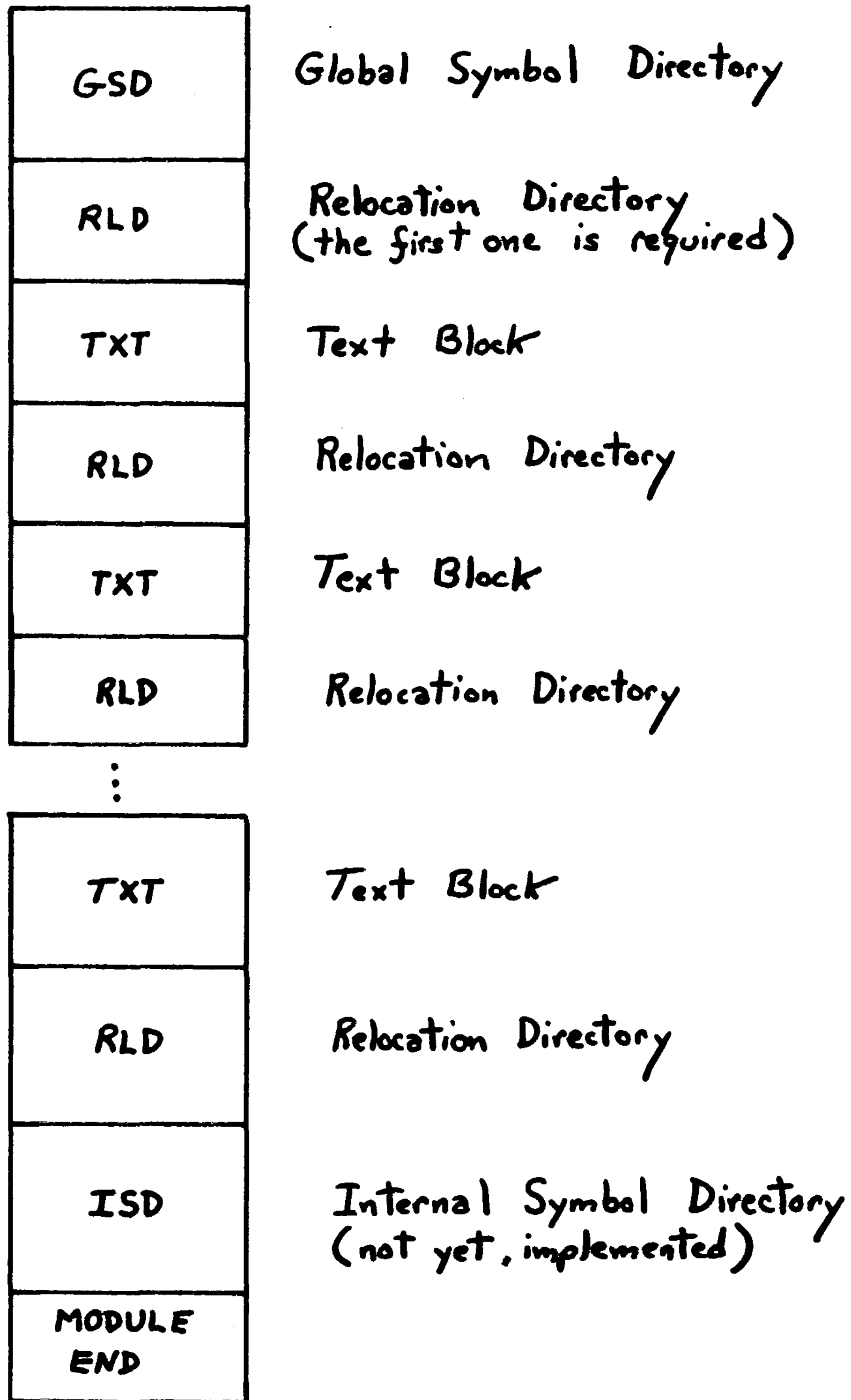


Figure 9-1 Object Module Format

Line 1 of GSD

Ø	GSD
Object Module Name	
(2 words, .RAD5Ø)	
FLAGS	
Ø	
DATA ⋮ DATA	

Object Command Byte (GSD = 1) Followed by a null byte

4 words --- the 1st symbol in the last .TITLE assembler directive encountered

Line 2 of GSD

Ø	GSD
DATA	

More declarations of section names, global names, etc.

Lines 3 to N-1 of GSD follow the format of Line 2

Last Line of GSD

Ø	GSDEND

Object Command Byte (GSDEND = 2) Followed by a null byte

Maximum length of line accepted by Linker: 93_{10} words

Figure 9-2 Blocking of the GSD

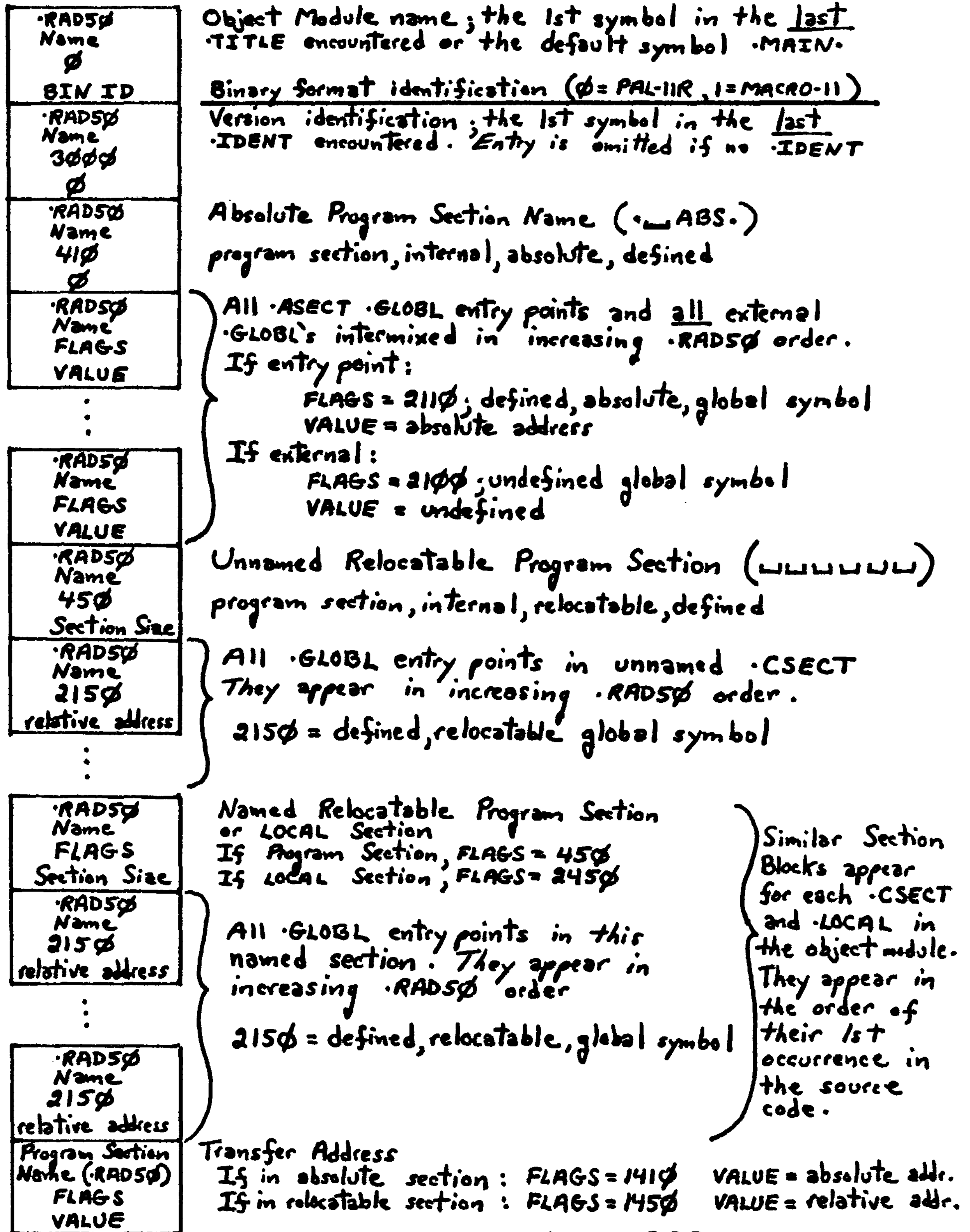
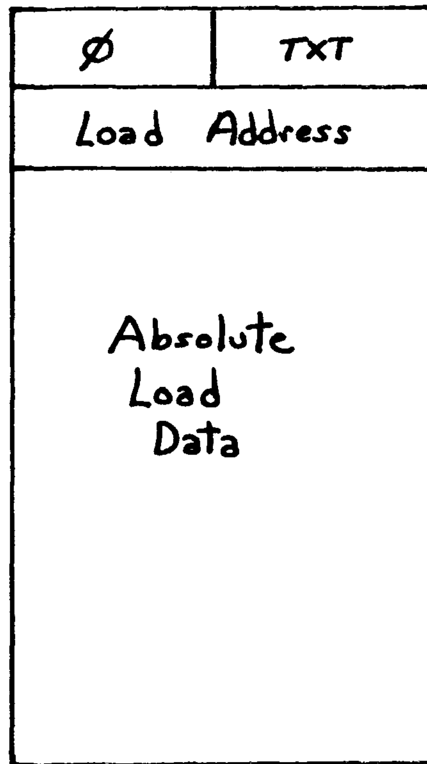


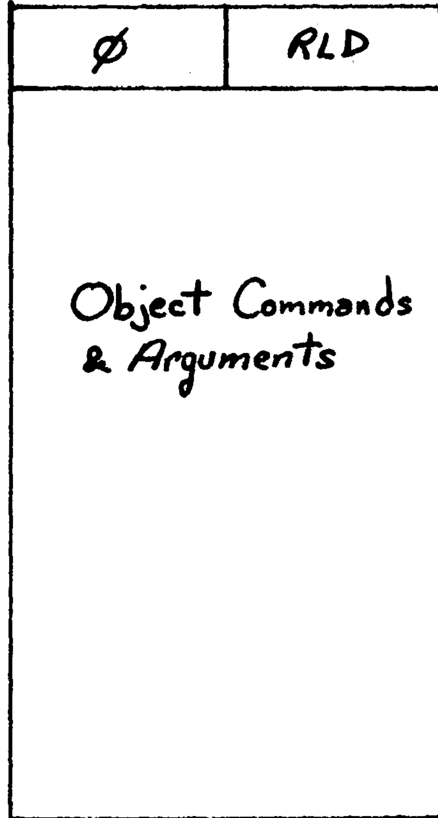
Figure 9-3 Contents of GSD



Maximum length of line accepted by Linker: 93_{10} words

Figure 9-4 Format of a Text Block

RLD
Line



Object Command Byte (RLD = 4)
followed by a null byte

Body of the
Object Commands

Maximum length of line accepted by Linker: 93_{10} words

Figure 9-5 Format of a Relocation Directory

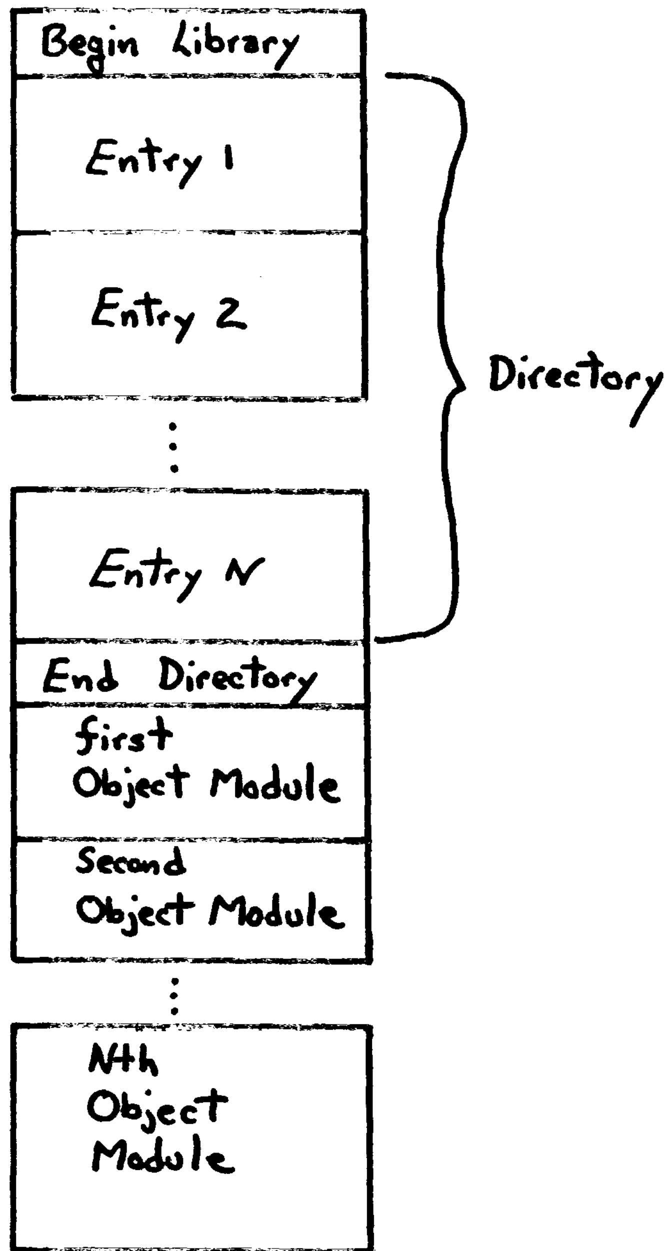
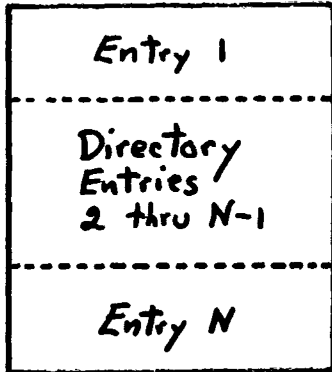


Figure 9-6 Object Module Library Format

Line 1 | Ø | 7 | Begin Library

Lines 2 thru m



an entry may consist of >1 line; each line may contain up to 93 decimal data words.

Line m+1 | Ø | 1Ø | End Directory

Lines m+2 thru n

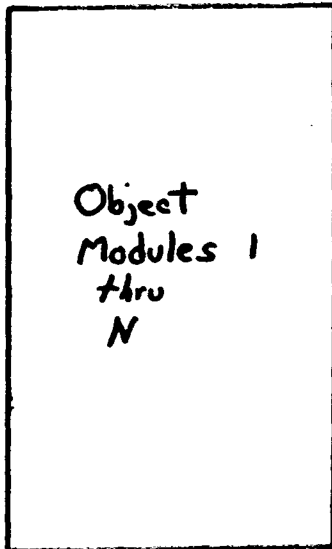


Figure 9-7 Blocking of Object Module Library

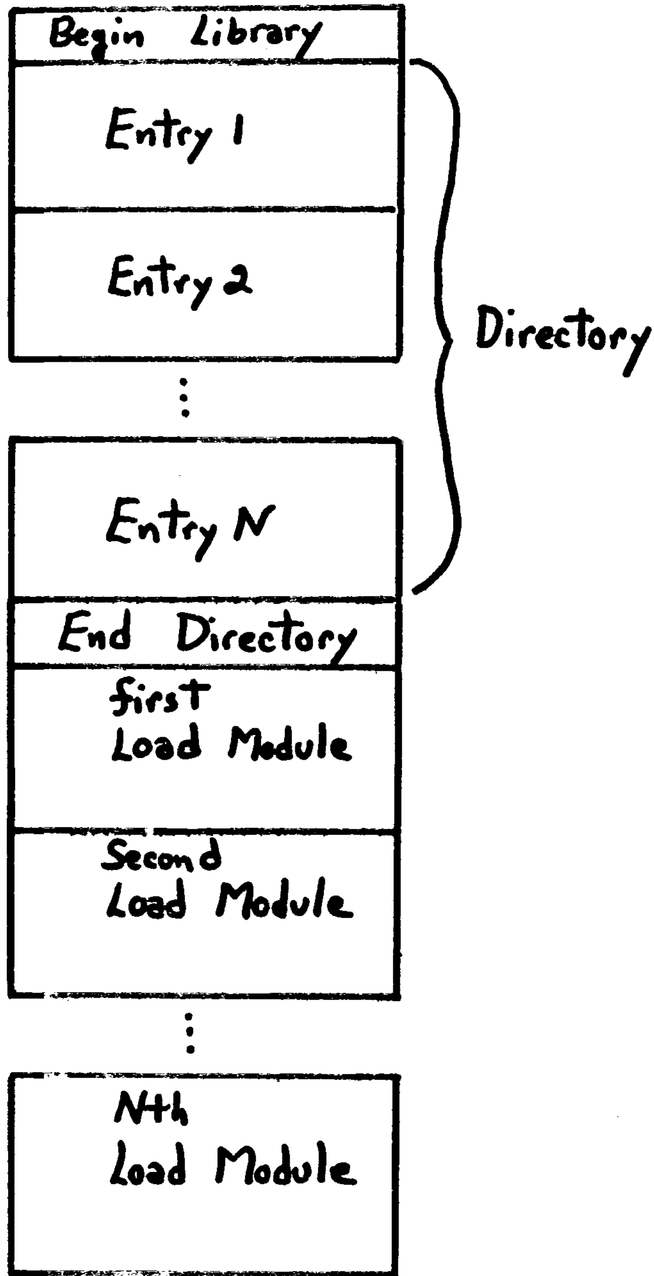


Figure 9-8 Load Module Library Format

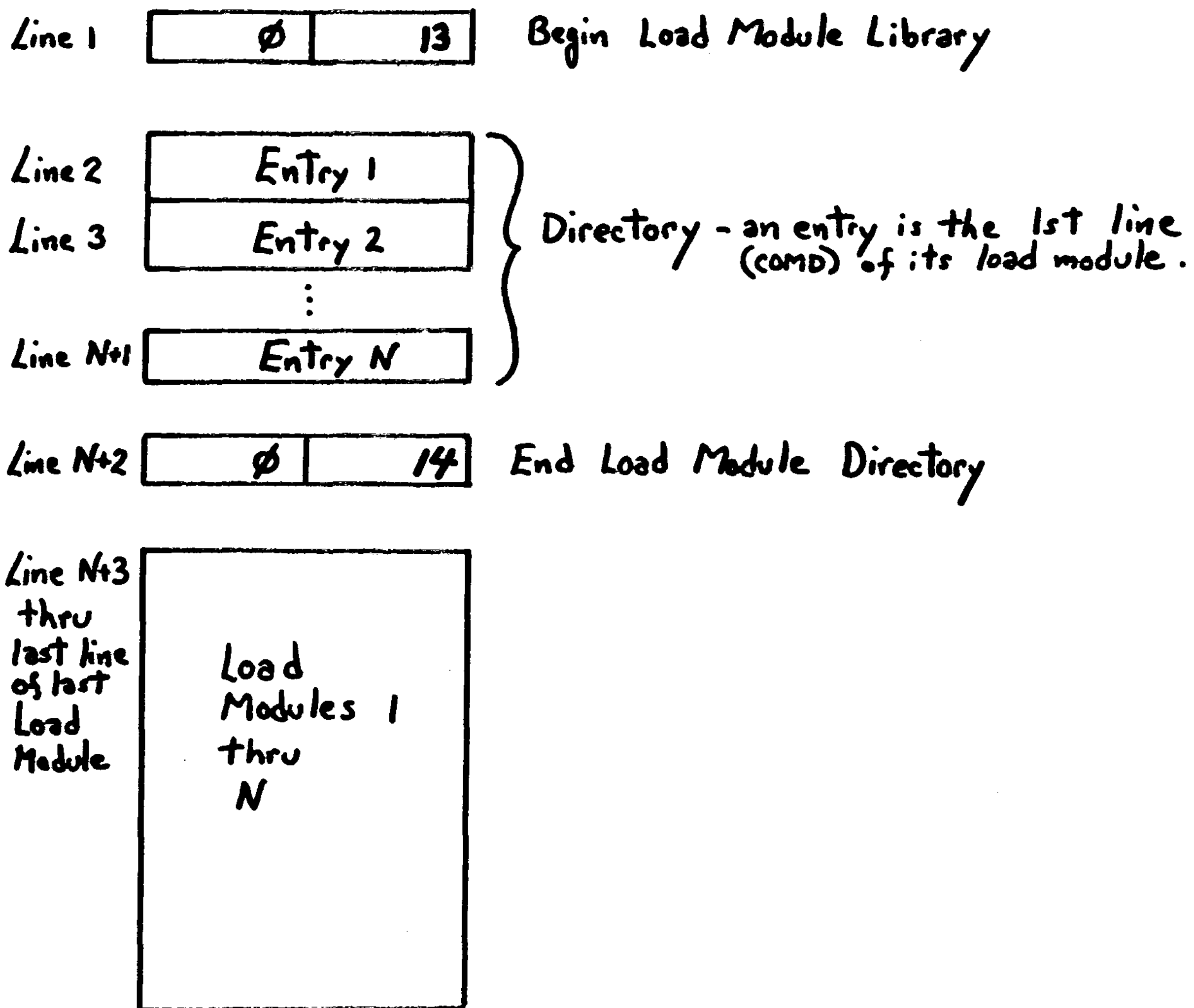


Figure 9-9 Blocking of Load Module Library

HOW TO OBTAIN SOFTWARE INFORMATION

Announcements for new and revised software, as well as programming notes, software problems, and documentation corrections are published by Software Information Service in the following newsletters.

Digital Software News for the PDP-8 & PDP-12
Digital Software News for the PDP-11
Digital Software News for the PDP-9/15 Family

These newsletters contain information applicable to software available from Digital's Program Library, Articles in Digital Software News update the cumulative Software Performance Summary which is contained in each basic kit of system software for new computers. To assure that the monthly Digital Software News is sent to the appropriate software contact at your installation, please check with the Software Specialist or Sales Engineer at your nearest Digital office.

Questions or problems concerning Digital's Software should be reported to the Software Specialist. In cases where no Software Specialist is available, please send a Software Performance Report form with details of the problem to:

Software Information Service
Digital Equipment Corporation
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754

These forms which are provided in the software kit should be fully filled out and accompanied by teletype output as well as listings or tapes of the user program to facilitate a complete investigation. An answer will be sent to the individual and appropriate topics of general interest will be printed in the newsletter.

Orders for new and revised software and manuals, additional Software Performance Report forms, and software price lists should be directed to the nearest Digital Field office or representative. U.S.A. customers may order directly from the Program Library in Maynard. When ordering, include the code number and a brief description of the software requested.

Digital Equipment Computer Users Society (DECUS) maintains a user library and publishes a catalog of programs as well as the DECUSCOPE magazine for its members and non-members who request it. For further information please write to:

DECUS
Digital Equipment Corporation
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754

PDP-11 Disk Operating System Monitor
System Programmer's Manual
DEC-11-OSPMA-A-D
May 1972

READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback -- your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability and readability.

Did you find errors in this manual? If so, specify by page.

How can this manual be improved?

Other comments?

Please state your position. ______________________ Date: __________

Name: ______________________ Organization: ______________________

Street: ______________________ Department: ______________________

City: ____________ State: ____________ Zip or Country ____________

digital equipment corporation