DECUS - April, 1980

RSTS/E

HANDOUTS

RSTS/E V7.0
Terminal Service
Internals

Andy Riebs

DEC - Merrimack

U. S. DECUS

Spring, 1980

Chicago

Components of Terminal Service

TTDVR

- o The heart of Terminal Service
- o Distributed in source
- o Assembled during SYSGEN
- o Primary subject of this discussion

TTDINT

- o Defines most Terminal Service tables
- o Defines device interrupt vector dispatch
- o Distributed in source
- o Assembled during SYSGEN
- o Table of Contents shows physical/logical KB assignments

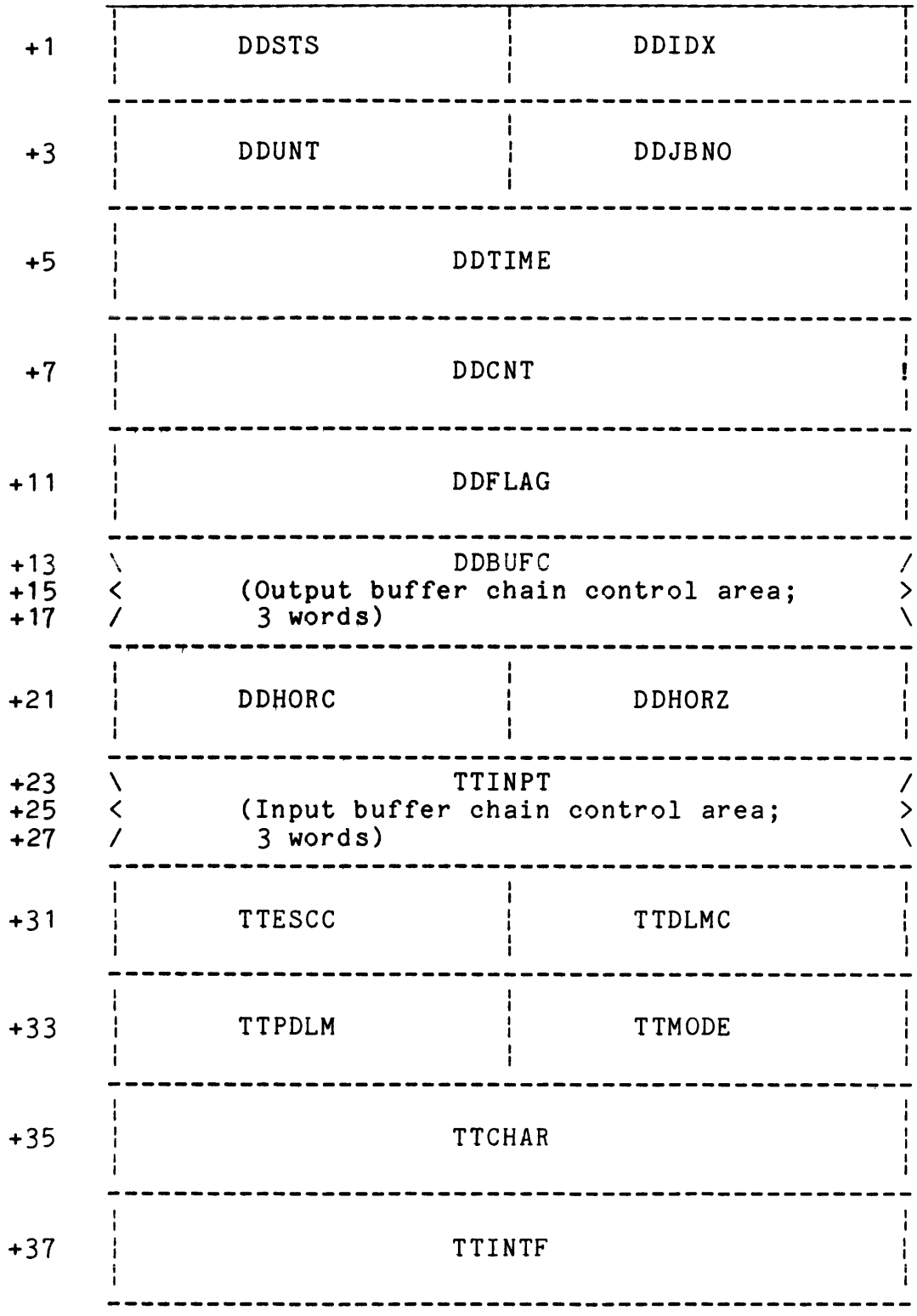
TRM

- o UUO (SYS call) processing for
    - o Set terminal characteristics
    - o Hangup/enable dataset
    - o Disable terminal
- o Distributed in binary form
- o Always included in the system

TTSYST

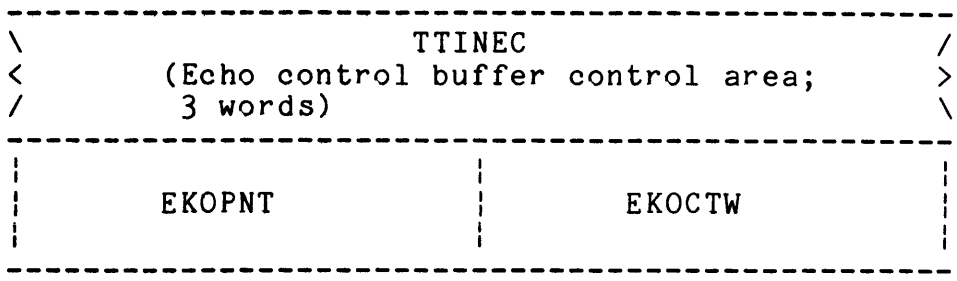
- o Special ^T "mini-systat" code
- o Distributed in binary form
- o Optionally linked in

Fixed Terminal DDB layout

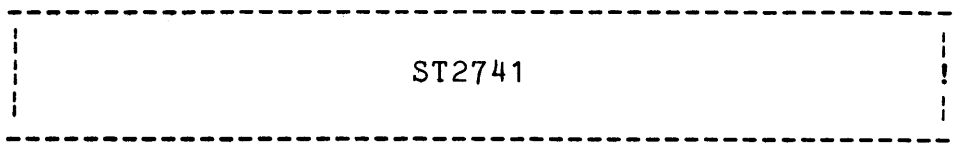


[End of fixed DDB]

[Echo Control]



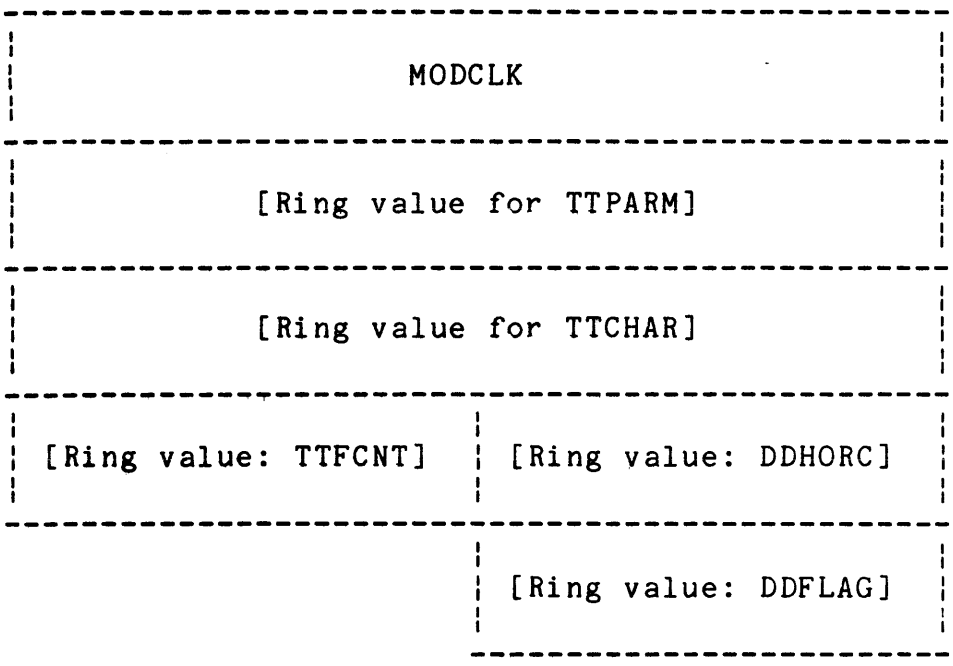
[2741 Support]



[Interfaces]

[DM11BB or DZ 2741 Support]

[Modem Support]



[Always present]

[Multi-TTY Support]

DDS.KB:: Defines size of a terminal DDB in bytes

STANDARD TERMINAL DDB LAYOUT

*** Fixed Region ***

DDIDX:	Driver index (IDX.KB = 2)
DDSTS:	Status and access control byte
DDJBNO:	Owner job number times 2 (0 if free)
DDUNT:	Device unit number
DDTIME:	Time assigned or opened
DDCNT:	Open count and assignment control
DDFLAG:	Device dependent flags
DDBUFC:	Buffer chain control area (output)
DDHORZ:	Horizontal position, kept as DDHORC - (positions from left margin)
DDHORC;	Characters per line + 1
TTINPT:	Buffer chain control area (input)
TTDLMC:	Delimiter counter
TTESCC:	Incoming ESCAPE SEQUENCE control
TTMODE:	Current OPEN MODE
TTPDLM:	Terminal's private delimiter
TTCHAR:	Terminal's characteristics
TTINTF:	Terminal interface type

*** Variable Region ***

TTINEC:	Buffer chain control area (type-ahead for Echo Control)
EKOCTW:	Echo Control field size and mode
EKOPNT:	Paint character and "Field Active" flag
ST2741:	2741 control and status
TTPARM:	Interface parameter word
TTAUXP:	Pointer to DM11BB CSR or DZ BREAK byte
MODCLK:	Modem timing and status
TTFCNT:	Fill characteristics
TTMSCN:	Base KB number for round-robin multi-TTY scan

DDSTS (V7.0 assignments)

DDPRVO:	Ownership requires privileges
DDSTAT:	Junk programmed output (CONTROL/O)

DDCNT (V7.0 assignments)

DDCONS:	Device is the console device
DDASN:	Device assigned through command

DDFLAG (V7.0 assignments)

TTLFRC:	Set after force/broadcast
TTNBIN:	Output next in Binary mode
TTRSX1:	RSX mode (see below)
TTDDT:	DDT sub-mode
TAPE:	TAPE mode
NOECHO:	Inhibit echoing
LCLCPY:	Local echo
TT2741:	2741 type terminal
TTRSX2:	RSX mode (see below)
TTHUNG:	Terminal "hang" pending
TTDFIL:	Delay fill to next character
TTMSG:	Processing incoming message
RUBOUT:	Processing RUBOUT's
TTSXOF:	Send XOFF as soon as possible
TTXOFF:	An XOFF was sent
TTSTOP:	Output temporarily stopped (CONTROL/S)

RSX Flags in DDFLAG

TTRSX1	TTRSX2	Meaning
0	0	Not in RSX mode
0	1	"Extra <LF> Mode" -- Don't print <LF> if user's next request specifies TO.PRE
1	0	"Normal" RSX mode
1	1	RSX mode -- Need a <LF> before another character is printed

TTMODE (Terminal's OPEN mode)
(V7.0 assignments)

TTBIN:	Binary input mode
TTTECO:	TECO mode (reserved; subject to change)
TTCRLF:	No auto CR/LF mode
TTECTL:	Echo Control mode
TTGARD:	Guarded terminal mode
TTPCOL:	Check incoming XON/XOFF mode
TTTECS:	TECO Scope mode (reserved; subject to change)

TTCHAR (Terminal's characteristics)
(V7.0 assignments)

TTXANY:	Do "XON" on any character
TTFUNC:	Disable special function characters
TTESC:	Terminal has real ESCAPE
TTSCOP:	Scope type terminal
TTESCI:	Allow incoming ESCAPE Sequences
TTLCOU:	Allow lower case output
TTPODD:	Desired parity is odd
TTPRTY:	Check and send with parity
TTUPAR:	Prefix control characters with uparrow
TTSYNC:	Stop output if XOFF received
TTXON:	Sending XOFF will stop input
TTFORM:	Hardware FORM FEED/VERTICAL TAB exists
TTTAB:	Hardware HORIZONTAL TAB exists
TTLCIN:	Allow lower case input

DH11: TTPARM defined as follows:

<1-0>	Character length (0->5, 1->6, 2->7, 3->8.) *
<2>	Stop bit (0->1, 1->2)
<3>	0
<4>	Parity Enable (0->no parity, 1->parity)
<5>	Odd parity (0->even parity, 1->odd parity)
<9-6>	Input speed code **
<13-10>	Output speed code **
<15-14>	0

DZ11: TTPARM defined as follows:

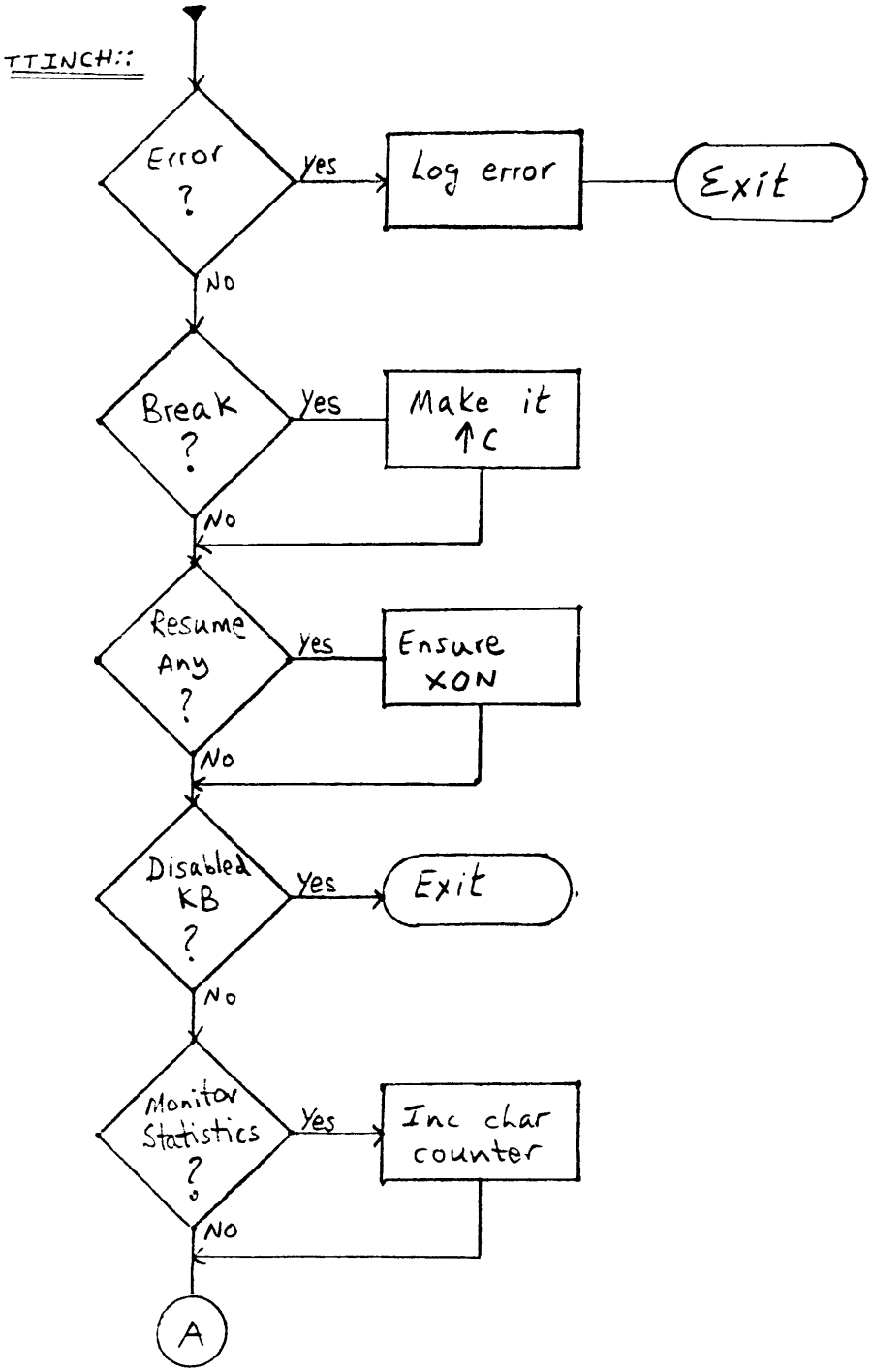
<2-0>	Subline number
<4-3>	Character length (0->5, 1->6, 2->7, 3->8) *
<5>	Stop bits (0->1, 1->2)
<6>	Parity Enable (0->no parity, 1->parity)
<7>	Odd parity (0->even parity, 1->odd parity)
<11-8>	Speed code **
<12>	Receiver Clock Enable
<15-13>	0

* This field refers to the length of the data portion of the character, exclusive of parity bits, if any.

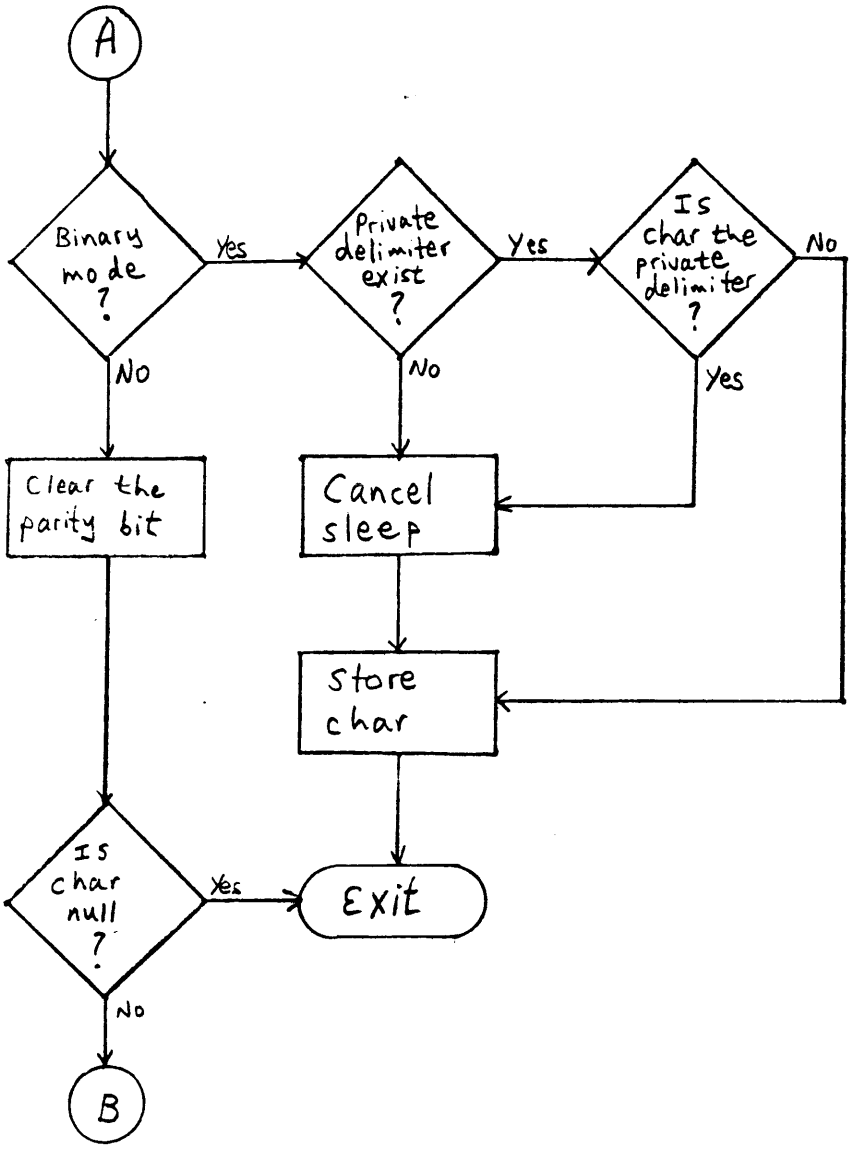
** Hardware dependent; refer to the description of the Line Parameter Register for the DH11 or DZ11 in the "DIGITAL Terminals and Communications Handbook."

Note that the TTYSET UUO (SYS call) passes data in the DH11 format; TRM will do any necessary conversion for other device types.

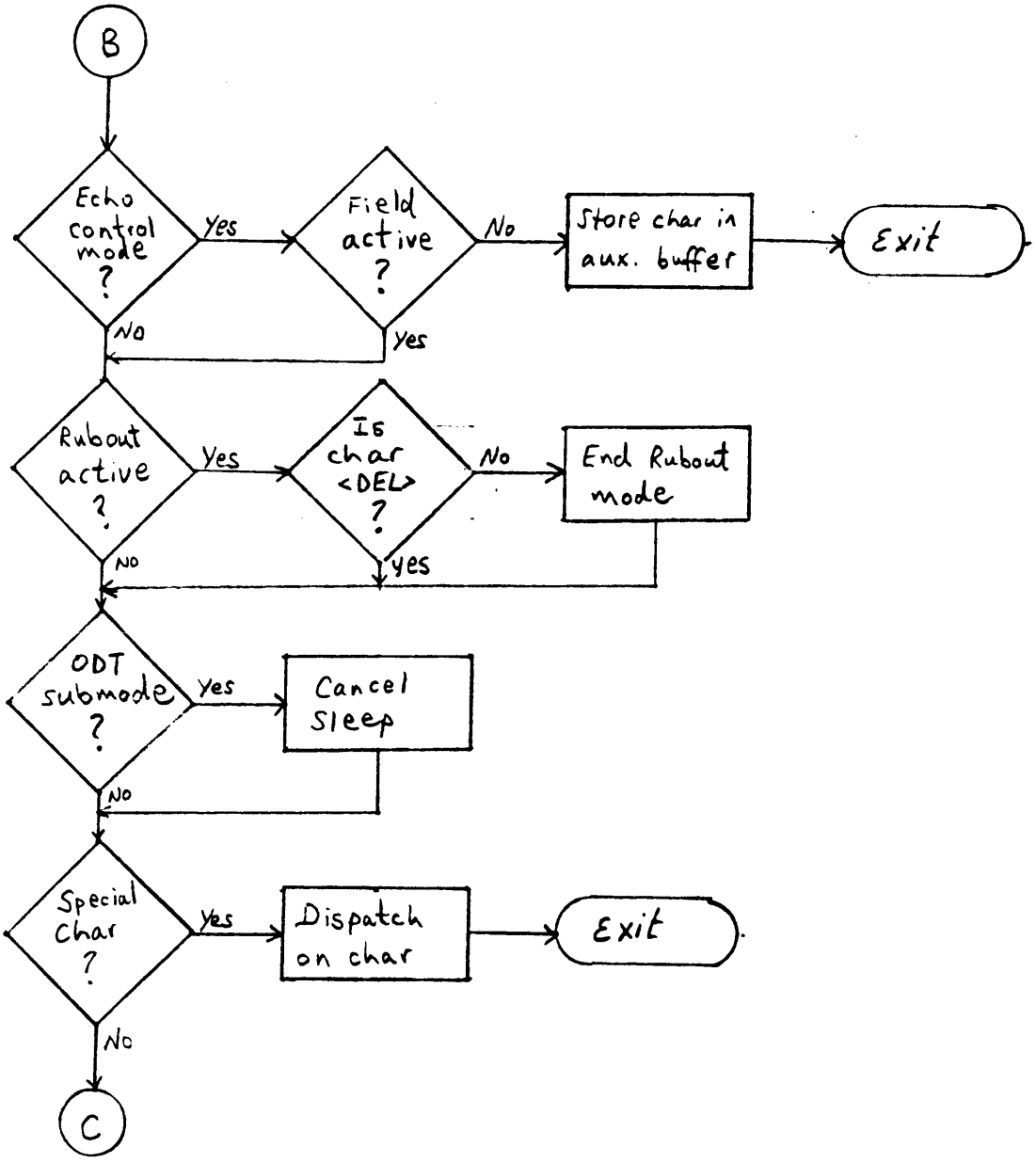
Character In (Terminal Interrupts disabled)



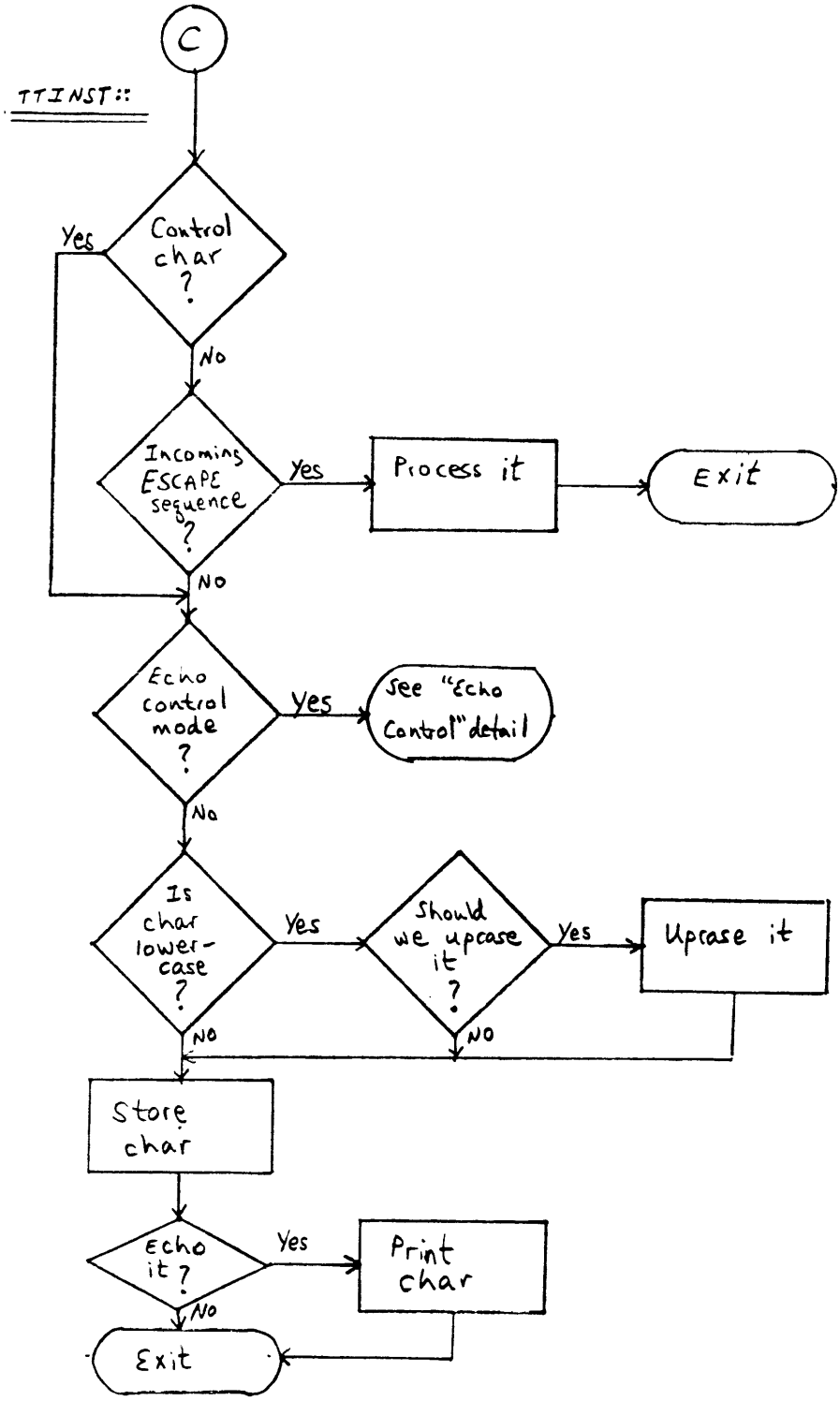
Character In



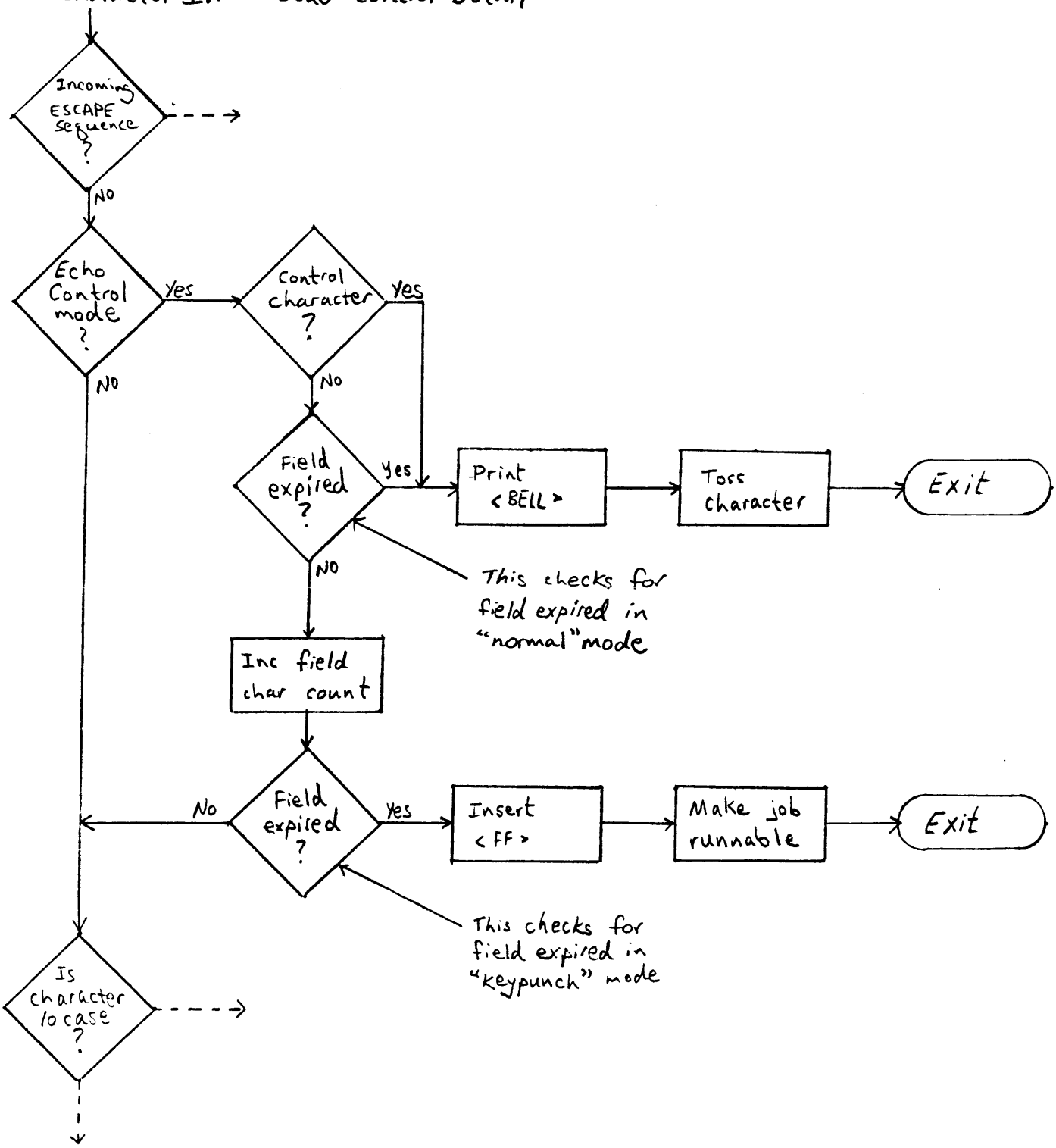
Character In



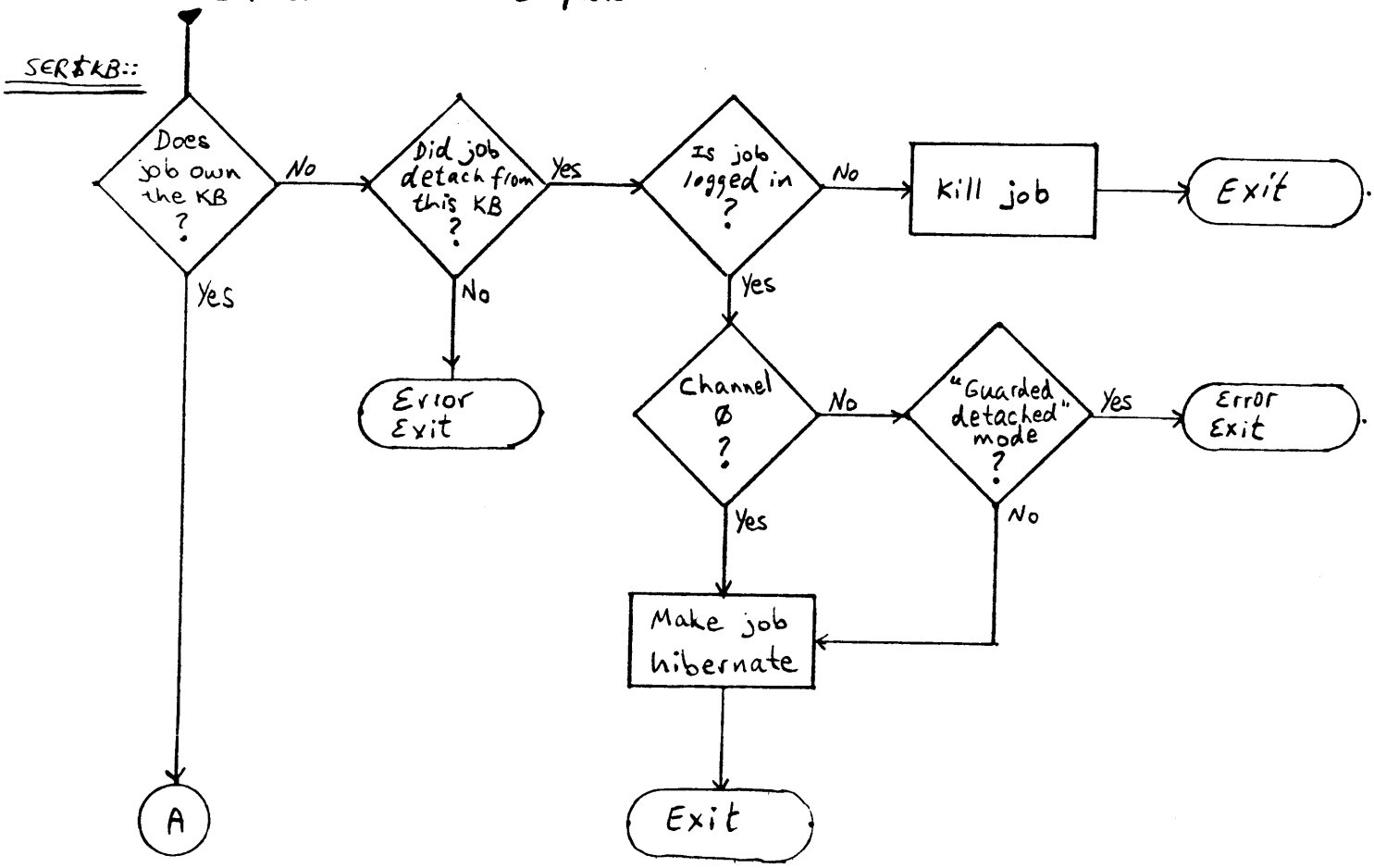
Character In



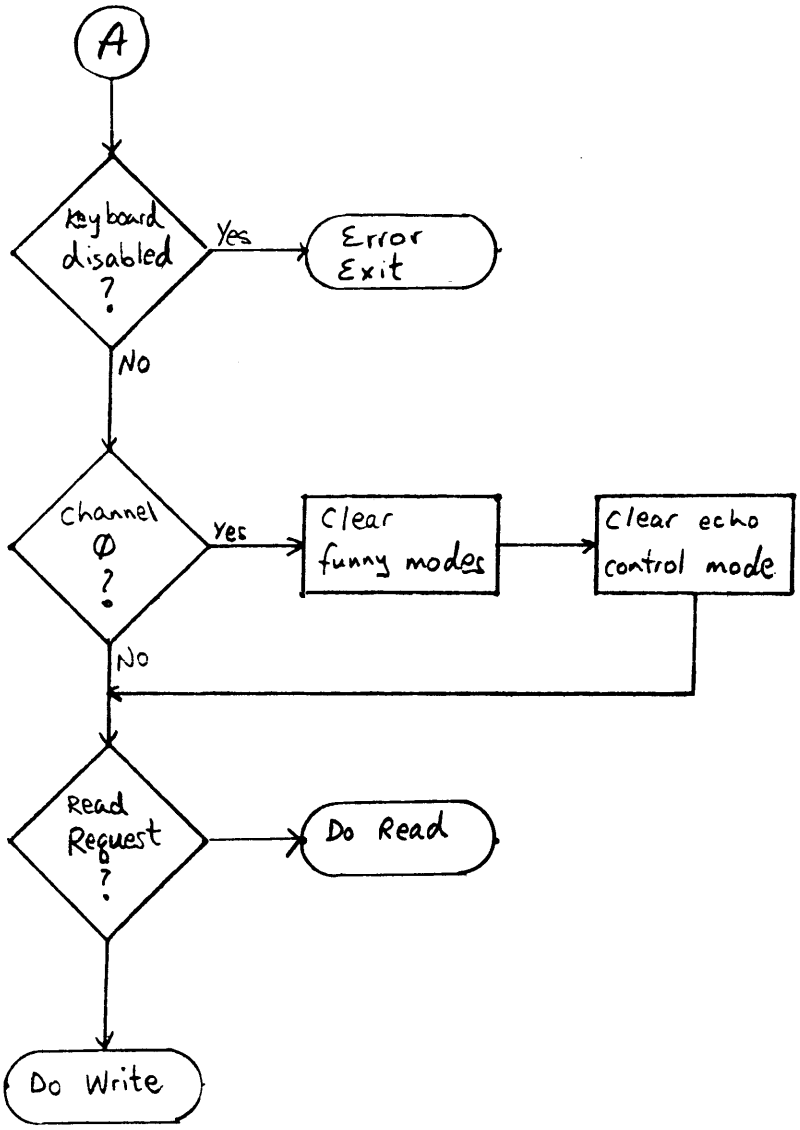
Character In - Echo Control Detail



User Level I/O Dispatch

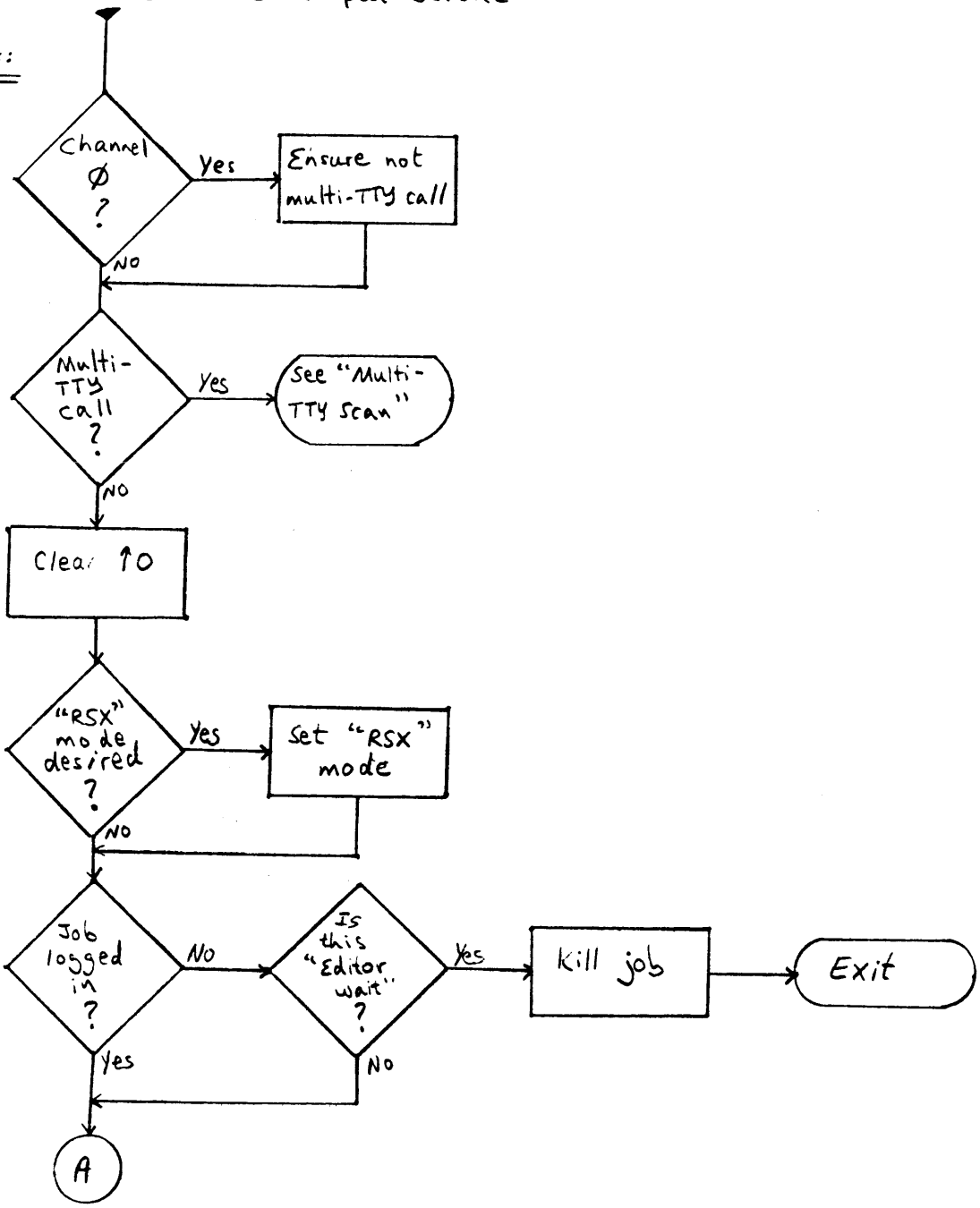


User Level I/O Dispatch

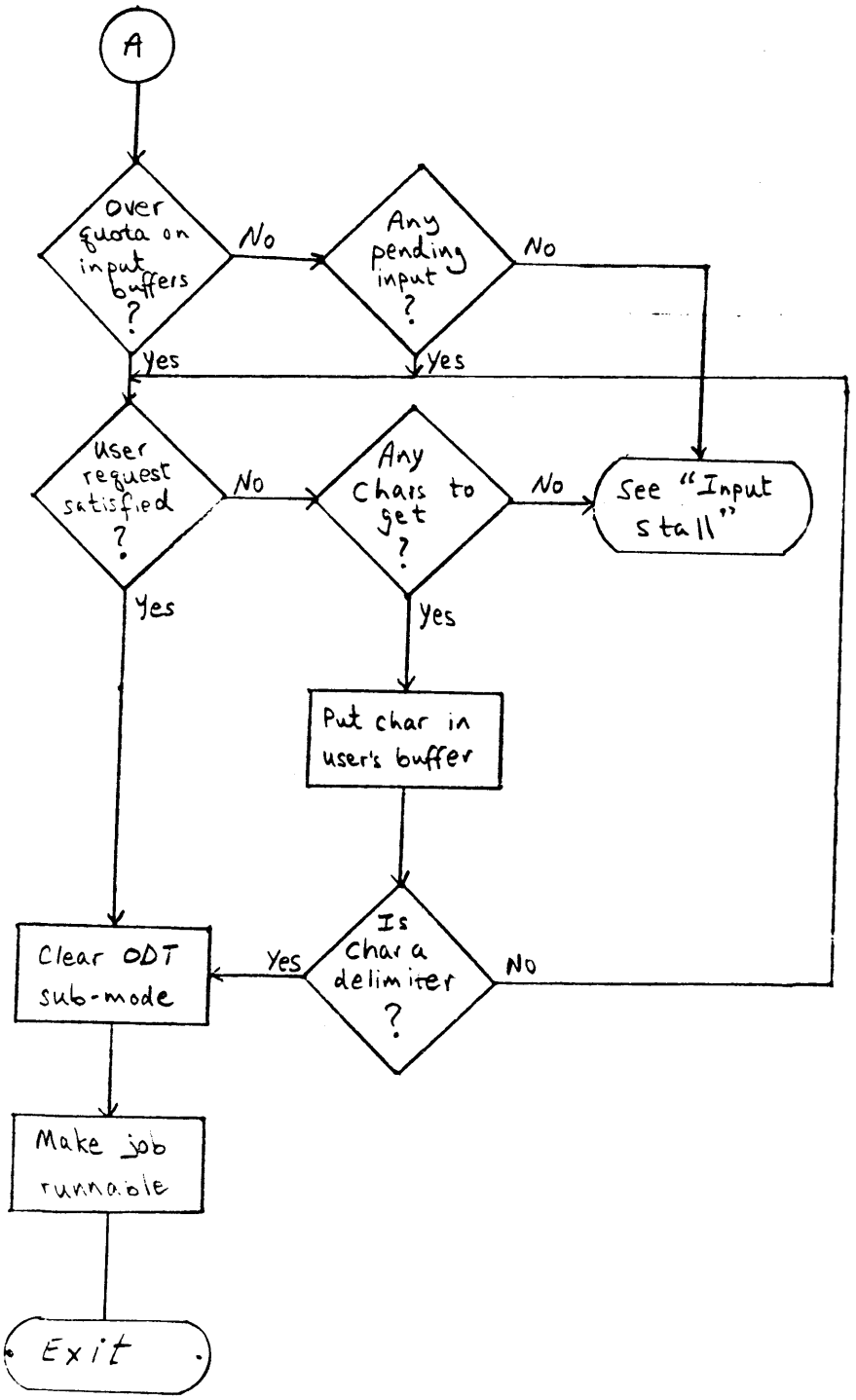


User Level Input Service

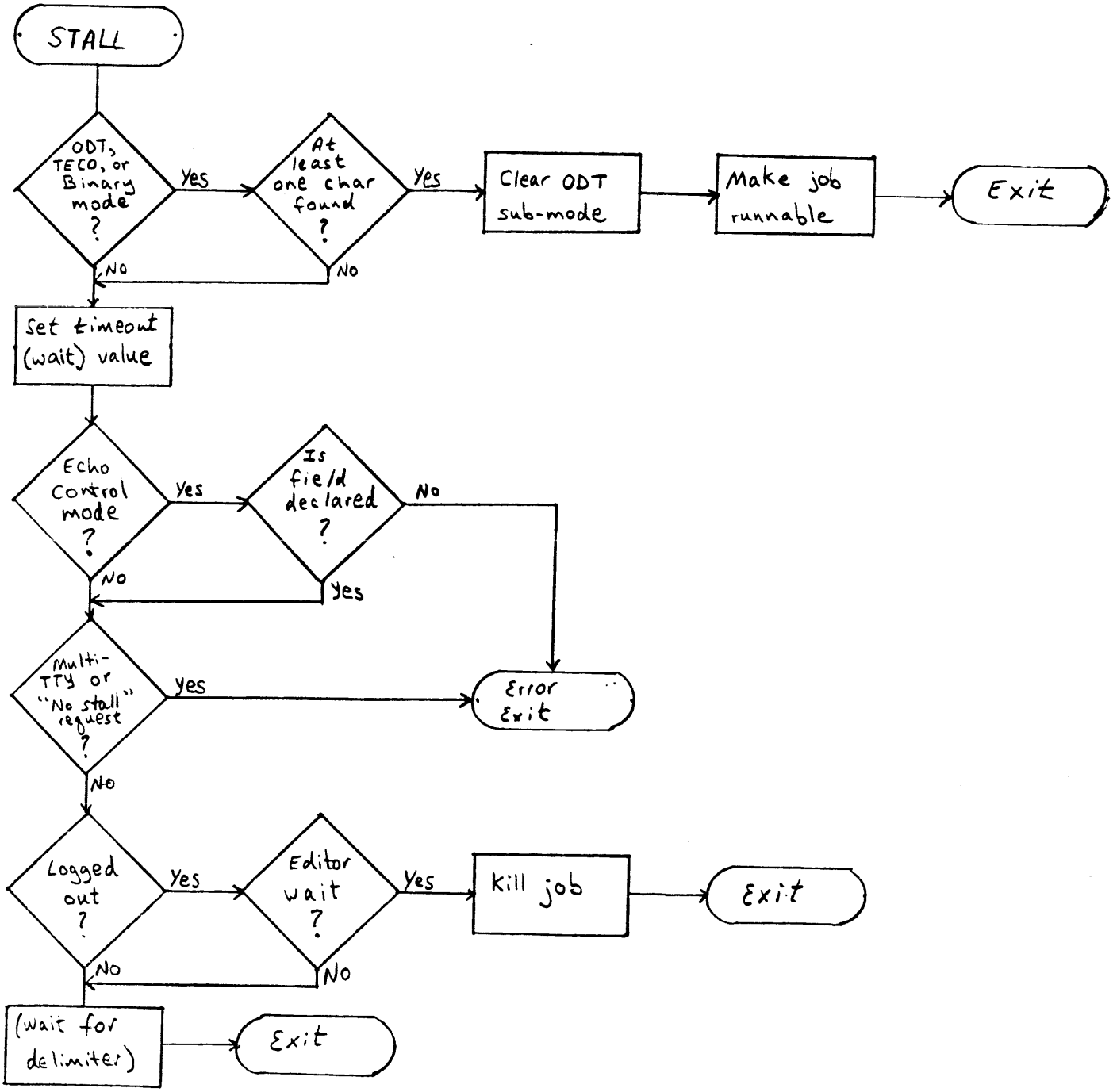
TTISER::



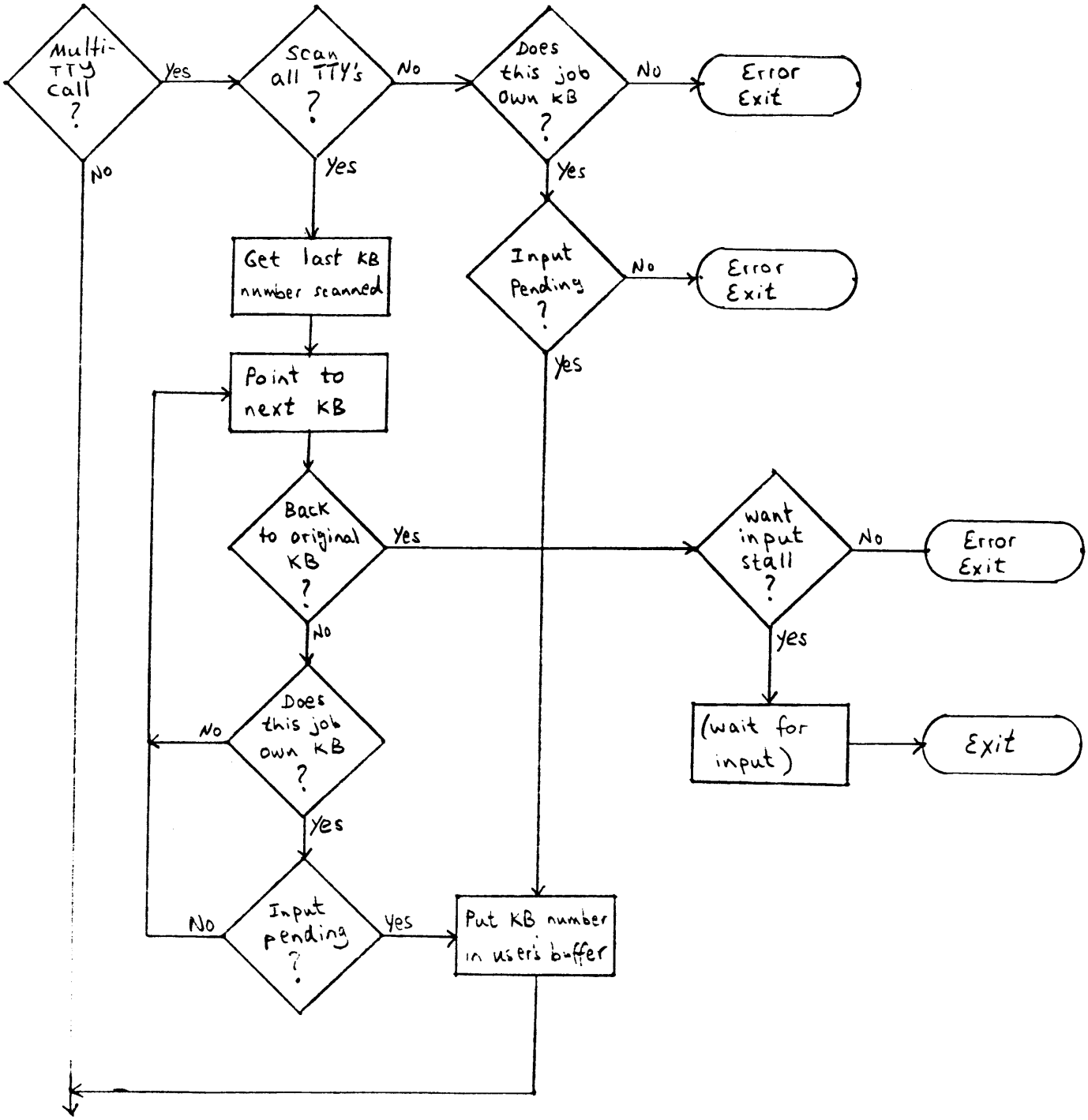
User Level Input Service

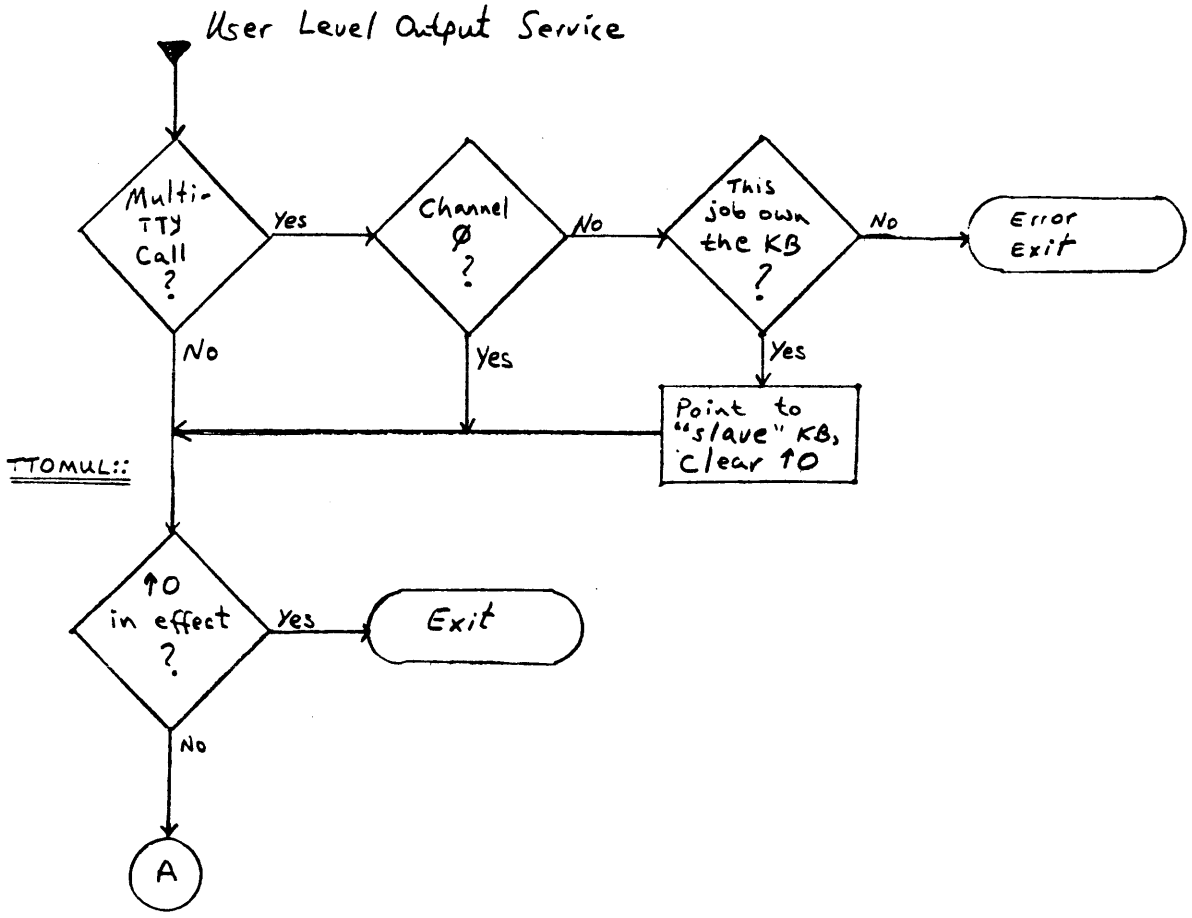


User Level Input Service - STALL

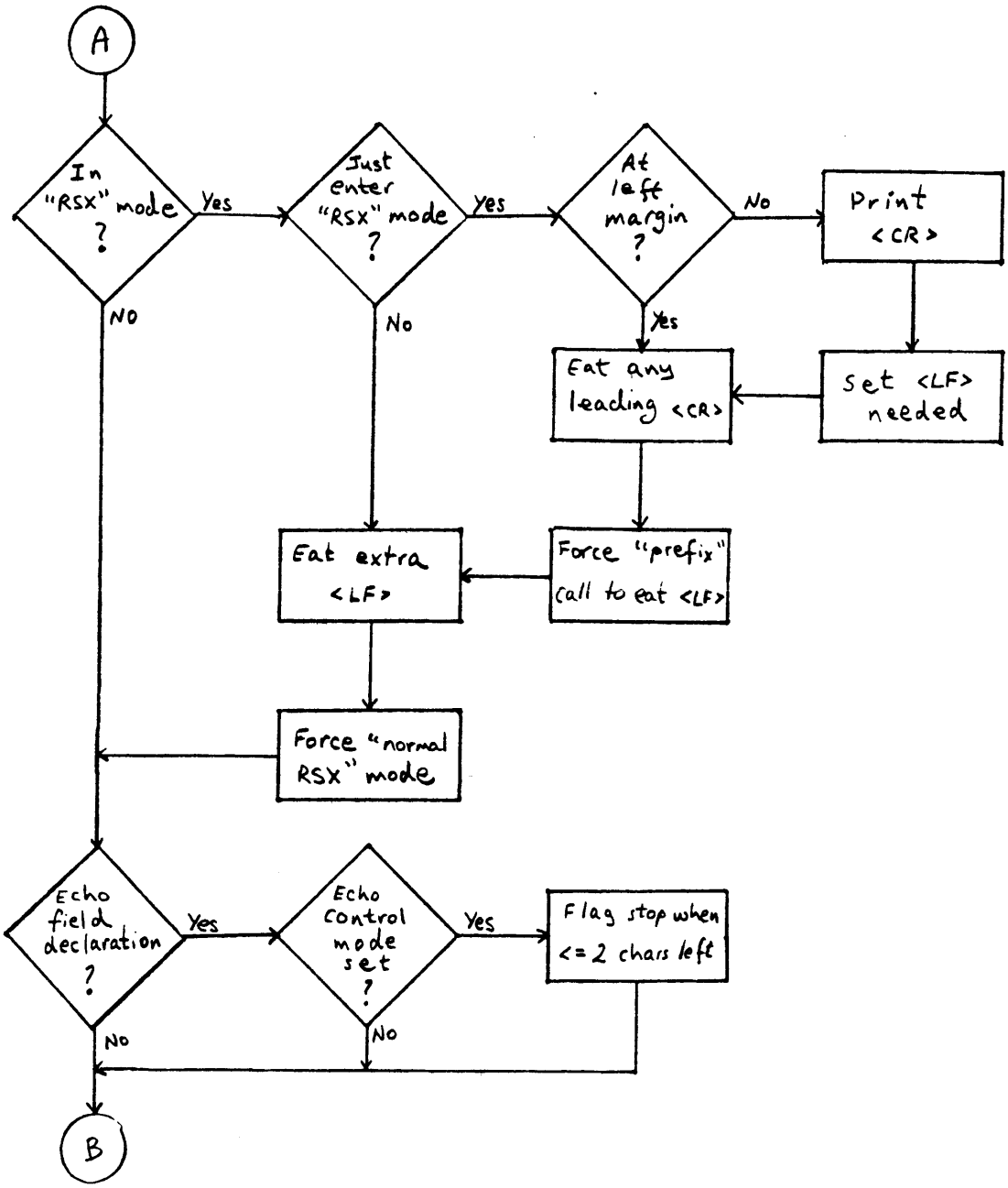


User Level Input Service - Multi-TTY scan

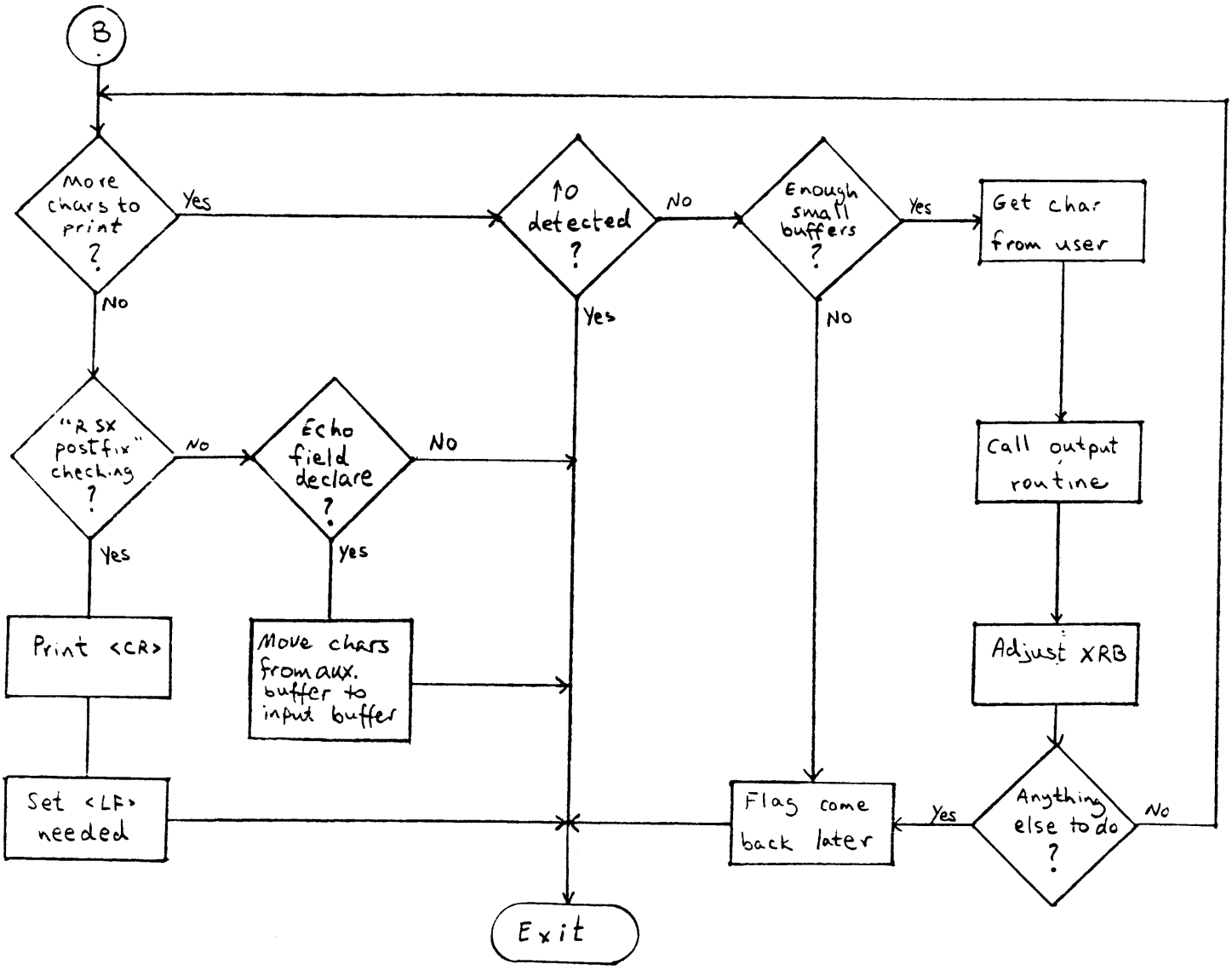



User Level Output Service
Multi-TTY Call ?
Yes
Channel Ø ?
No
This job own the KB ?
No
Error Exit
No
Yes
Yes
Point to "slave" KB, Clear ↑O
TTOMUL::
↑O in effect ?
Yes
Exit
No
A

User Level Output Service



User Level Output Service



Disk Internals and SAVE/RESTORE

Nancy Covitz
Chicago DECUS
April, 198Ø

1.Ø INTRODUCTION

Prior to the release of RSTS/E V7.Ø, RSTS/E users had no way of making a fast copy of a disk with bad blocks. Many of our customers were extremely unhappy with the existing BACKUP package. While it offers several important features, notably a high degree of flexibility, it may be quite slow. The BACKUP dialogue is also fairly complex. In many cases, users have misunderstood a BACKUP question and proceeded to transfer the wrong set of files.

A conscientious installation should back up its production disks at least once a week, preferably more often. Bypassing a regular backup procedure could lead to the loss of weeks, or even months of work in the event of a disk crash. Extremely long BACKUP times have made it nearly impossible for some users to back up large disks in an acceptable amount of time.

It was apparent that the advent of newer, increasingly larger disks would raise the level of discontent among current customers even higher. It was also felt that potential customers were being discouraged by the lack of a fast backup facility on RSTS/E.

Several items were considered to be major goals of the development effort. Specifically, SAVE/RESTORE had to be

- reliable
- fast
- easy to use, and
- capable of handling bad blocks

If these goals could be met, SAVE/RESTORE would provide RSTS/E customers with a high-speed, volume backup capability.

The next section describes the RSTS/E on-disk structure. This information is important for two reasons: First, the existing BACKUP package is written in BASIC-PLUS and, therefore, utilizes the system file processor for file transfers. As can be seen by examining the structures described in the ensuing sections, this can be an extremely long process if large numbers of files need to be selected and transferred. More importantly, any program which properly handles bad blocks on a non-file structured RSTS/E disk must be able to "understand" the on-disk directory structure. In particular, if a transfer would cause a bad block to be included in a file, the data in the file must be moved to a new location and all directory pointers changed accordingly.

Note that some of the information included in this document is not specifically needed by SAVRES (e.g., various status bits or accounting data). In addition, some information which follows the description of the on-disk structure is not directly related to the directory structure. These "extras" are included to give you a complete picture of the directory structure, as well as some indication of why certain elements of SAVRES "are the way they are".

2.Ø DISK STRUCTURE OVERVIEW

Access to disk media under RSTS/E is normally controlled by the file processor (FIP) and by user level disk I/O routines. In order to access disk data, the file processor must be called to associate the disk data (e.g., a disk file, a disk directory, or a whole disk) with a channel. FIP does access and existence checking and then "opens" that channel. Read/write access is checked and handled by the user level I/O routines. These routines rely on FIP having set up enough information to map logical user I/O requests (e.g., a "GET" statement in BASIC-PLUS) into physical medium read/writes. When processing is done, it is again FIP who "closes" the file and releases the channel.

The sections which follow describe the on-disk information structure that permits FIP to subdivide a disk into files.

2.1 Physical Disk Subdivisions

The following terminology is used to describe the basic subdivisions of a RSTS/E disk itself. These definitions are referenced by many other sections and are the global basis for the on-disk structure. The primary users of the terms described here are the disk drivers and FIP.

2.1.1 Block - A block is 256 words of disk data.

2.1.2 Logical Block (LB) - All disks are divided (either by the disk hardware itself or artifically by the software) into a collection of LB's. A LB is the smallest unit of disk data addressable by software under RSTS/E.

2.1.3 Logical Block Number (LBN) - Each LB is assigned a unique 23-bit LBN. LBN's start at Ø and increase by +1 all LB's have been assigned a LBN. LBN's are ordered such that the data of LBN n+1 is physically adjacent to and immediately after the data of LBN n.

2.1.4 Cluster - A cluster is a collection of LB's with sequential LBN's, i.e., a series of contiguous blocks. The number of LB's in a cluster is always a power of 2 and may range from 1 to 256 inclusive.

2.1.5 Device Cluster (DC) - All RSTS/E disks are divided into DC's, which are the primary storage unit of a RSTS/E disk. A DC is the smallest allocatable unit of disk data.

2.1.6 Device Cluster Number (DCN) - Every DC is assigned a unique 16-bit DCN. DCN's start at Ø and increase sequentially until all LB's of the disk are contained in a DC.

2.1.7 Device Cluster Size (DCS) - Every disk type has a permanently assigned DCS. The DCS is always a power of 2 and between 1 and 16 inclusive. The DCS assigned to any given disk type is chosen such that the maximum DCN remains a 16-bit number. The following table lists the DCS for all disks currently supported by RSTS/E.

Disk	DCS	Device Size in 256-wd blks	Device Size in bytes
RS64	1	256*(# of platters)	
RS11	1	1Ø24*(# of platters)	
RSØ3	1	1Ø24	524,288
RSØ4	1	2Ø48	1,Ø48,58Ø
RKØ5	1	48ØØ	2,457,6ØØ
RKØ5F	1	48ØØ per unit;2 units per drive	
RLØ1	1	1Ø22Ø	5,232,64Ø
RLØ2	1	2Ø44Ø	1Ø,465,3ØØ
RKØ6	1	271Ø4	13,877,2ØØ
RKØ7	1	53768	27,529,2ØØ
RPØ2	2	4ØØØØ	2Ø,48Ø,ØØØ
RPØ3	2	8ØØØØ	4Ø,96Ø,ØØØ
RMØ2/RMØ3	4	131648	67,4Ø3,8ØØ
RPØ4/RPØ5	4	167ØØØ	85,6Ø6,4ØØ
RPØ6	8	334ØØØ	171,213,ØØØ

2.1.8 FIP Block Number (FBN) - FBN's are used internally by the file processor. FBN's are a special set of block numbers arranged such that FBN 1 corresponds to the first block of DCN 1 (the file structure's root is the MFD which always starts at DCN 1). While there is a unique LBN for every FBN, there are always (DCS-1) LBN's not describable by FBN's. These blocks are "lost" on file structured disks. Since at most (DCS-1) blocks are not describable by FBN's, the FBN is a 23-bit number.

The following equation relates the above terms.

$$FBN = (DCS * (DCN-1)) + 1$$

This is the major conversion used by FIP to convert its on-disk storage of retrieval information (which is always 16-bit DCN's) into its internal 23-bit FBN's.

2.2 Logical Disk Subdivisions

The following terms describe logical subdivisions of a RSTS/E disk.

2.2.1 Pack Cluster Size (PCS) - When a disk is initialized to the RSTS/E file structure, an additional cluster factor called the Pack Cluster Size (PCS) is superimposed on the structures defined above. The PCS Size is also a power of two between 1 and 16 and must be greater than or equal to the DCS.

2.2.2 Pack Cluster (PC) - A Pack Cluster is a contiguous group of n blocks where n = PCS. The PC is the smallest unit of storage which can be allocated on the RSTS/E disk. Each PC is represented by one bit in the Storage Allocation Table, which is discussed later.

2.2.3 Pack Cluster Number (PCN) - Every pack cluster is given a unique 16-bit number called the Pack Cluster Number (PCN). PCN's start at Ø and increase sequentially by 1. PCN Ø is always aligned with DCN 1. The LB's contained in DCN Ø are not described by a PCN.

2.2.4 Files, Virtual Blocks, and Virtual Block Numbers - All data accessible through file structured disk operations are contained in a "file". A file is an ordered set of virtual blocks, where a virtual block is equal in size to a disk block. The virtual blocks of a file of size n are consecutively numbered from 1 to n. The number assigned to a virtual block is called its Virtual Block Number (VBN). All file structured access to file data is by VBN. For example, in BASIC-PLUS

one might access the sixth virtual block of a file via a command of the form "GET #1%, RECORD 6%". The file processor, in turn, maps the specified VBN to a unique LBN.

2.2.4.1 File Cluster Size (FCS) - The blocks of a file are also grouped into clusters. A file's cluster size is specified when the file is created and may contain 1, 2, 4, 8, 16, 32, 64, 128, or 256 blocks. In general, the effect of a large file cluster size is to decrease access time to file data, possibly at the expense of wasted disk space.

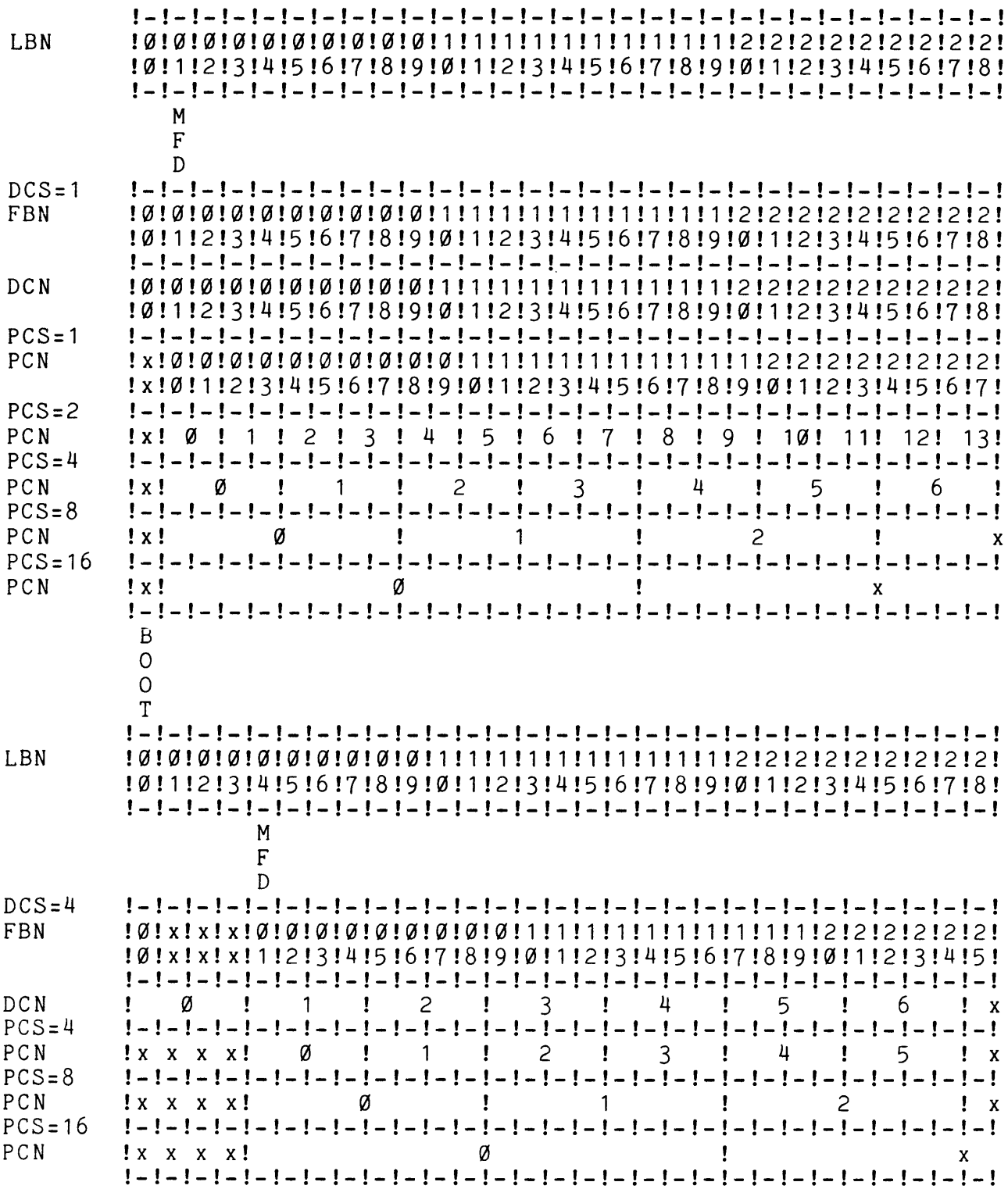
2.3 Clustersize Restrictions

The previous sections described device clusters, pack clusters, and file clusters. Two others types of clusters, MFD clusters and UFD clusters will be detailed in the sections which follow. Before introducing these items, the overall restrictions on cluster sizes should be noted (maximum cluster size for each type is shown in parentheses):

```
                                MFD  Cluster Size (16)
DCS (16) <= PCS (16) <= UFD  Cluster Size (16)
                                File Cluster Size (256)
```

2.4 Cluster Alignment

the alignment of lb's, dc's, and pc's can be confusing and is best described by a drawing. the following figure shows alignment and boundary conditions for a hypothetical disk whose MLBN (Maximum LBN) is 28 (i.e., total disk size is 29 blocks). Hypothetical DCS's of 1 and 4 are shown.


LBN
MFD
DCS=1
FBN
DCN
PCS=1
PCN
PCS=2
PCN
PCS=4
PCN
PCS=8
PCN
PCS=16
PCN
BOOT
LBN
MFD
DCS=4
FBN
DCN
PCS=4
PCN
PCS=8
PCN
PCS=16
PCN

2.5 Use of Disk Subdivisions

LBs are the fundamental units that all RSTS/E disk drivers deal with. All non-driver modules, however, deal exclusively with FBNs. These modules pass the disk driver dispatcher a starting FBN, along with the number of blocks to read or write. The dispatcher computes the correct LBN from the FBN, using a table lookup to find the disk's DCS. The resulting LBN (along with the number of blocks to read/write) is then passed on to the correct disk driver for the actual physical operation. It is the disk driver's responsibility to calculate the correct physical disk address needed by that type of disk controller. Note that the whole file processor works on a logical entity (the FBN) that can be manipulated without regard to the actual disk type.

All on-disk information that describes the location of ("points to") further disk data is in terms of DCN's. The upper limit of a DCN is fixed at the largest 16-bit number (65535) so that the on-disk storage of DCN's is confined to a single word per DCN.

The PC is the primary unit of storage allocation and the PCN is used as a pointer to a bit in the Storage Allocation Table.

SAVE/RESTORE uses most of these units in one way or another. For example, FBN's are used to retain certain bad block information, DCNs are used in directory scans and updates, and PCNs play the major role in SAVE/RESTORE's transfer algorithms.

It should also be noted here that much of the complexity of the on-disk structure stems from the development of large disks which necessitated some form of mapping for block numbers greater than 16-bits. Rather than creating a new file structure which would invalidate disks already existing in the field, extensions were made to the original RSTS/E on-disk structure.

3.Ø RSTS/E DIRECTORIES

The RSTS/E disk structure uses a two level directory hierarchy to organize user accounts and files. The primary directory structure is the Master File Directory (MFD) which catalogues accounts. There is one (and only one) MFD on each disk pack. The second level structure is the User File Directory (UFD) which catalogues user files. There is one UFD for each account which may contain user files. This section defines directory terminology, presents detailed descriptions of directory entries, and discusses the implications of the directory structure.

3.1 Directory Terminology

This section defines the terms used throughout the discussion of the RSTS/E directory structure.

3.1.1 Directory Clusters, Blocks, and Entries - RSTS/E directories are files consisting of Ø to 7 clusters. The maximum directory cluster size is 16, which implies that each Directory Cluster may contain 1, 2, 4, 8, or 16 blocks, subject to the restrictions mentioned earlier. Each directory block is further subdivided into 32 Entries of 8 words each. Entries come in many flavors depending on the type of directory (either MFD or UFD) and the function of the entry. The format of each type of entry is detailed in the sections which follow.

3.1.1.1 MFD Cluster Size (MCS) - As with any RSTS/E directory, the MFD Cluster Size may be 1, 2, 4, 8, or 16. The MFD Cluster Size is specified when the disk is initialized. The MFD Cluster Size determines an upper limit on the number of user accounts which can be catalogued on the disk.

3.1.1.2 UFD Cluster Size (UCS) - Like the MFD Cluster Size, the UFD Cluster Size may also be 1, 2, 4, 8, or 16. The UFD Cluster Size is specified when the account is created and determines the number and size of files which can be catalogued by the UFD.

3.2 Master File Directory (MFD)

Disks on a RSTS/E system fall into three categories: system, public, and private. The system disk, which contains such things as the initialization code and the installed monitor, is itself a part of the public structure.

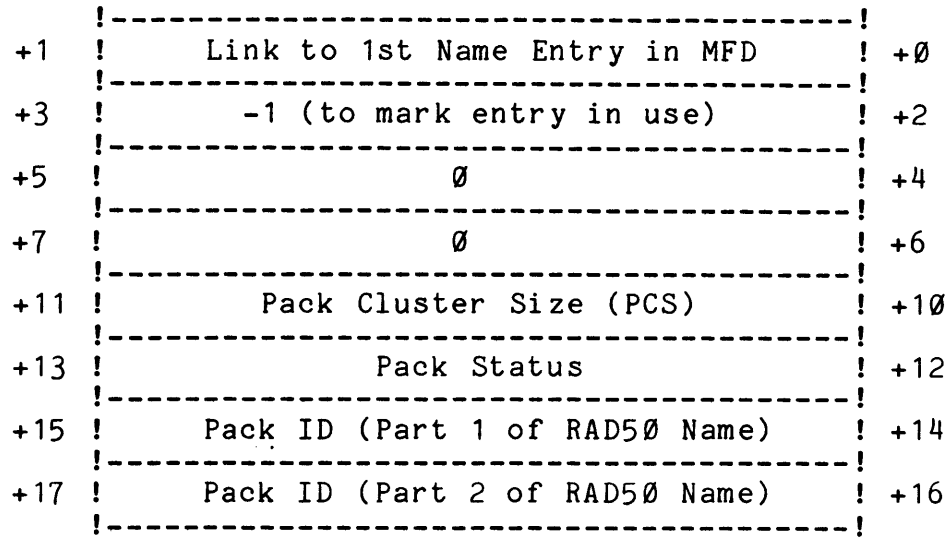
Each disk on the system, regardless of category, contains an MFD account in [1,1]. The MFD contains account and storage information for each account on the disk, as well as pointers to the UFD for the account. In addition to storing various accounting statistics the MFD also contains the information needed when mounting the disk (e.g., Pack ID).

In order for a user to gain access ("login") to the system, he or she must have an account catalogued on the system disk. If this is the case, the user may create files on any of the disks in the public structure. The system normally allocates files on the public disk which has the most free space. If a file is created on a public disk other than the system disk, the user's account is dynamically added to the MFD for that disk if it does not already reside there.

The MFD for a disk is created either by running an on-line disk initialization program (DSKINT) or by using the DSKINT option of the RSTS/E initialization code. The disk initialization process creates a minimal MFD consisting of two accounts ([0,1] and [1,1]) for non-system disks, or three accounts ([0,1], [1,1], and [1,2]) for system disks. Account [0,1] is the system files account which is required on all disks to contain the bad block file (BADB.SYS) and the Storage Allocation Table file (SATT.SYS). Account [1,1] is the MFD itself (the MFD serves as the UFD for account [1,1]). Account [1,2] is the standard system library account. The system manager or a privileged user may create additional accounts during normal timesharing.

Since the MFD is the root of the whole directory structure, the first cluster of the MFD is located in a fixed location on all disks. (Aside from the bootstrap, this is the only part of RSTS/E which must reside at a specific location.) The MFD always begins at FBN 1, which is also DCN 1. Other clusters of the MFD may reside anywhere on the disk. The MFD contains one label (or home) entry, a name and an accounting entry for each account, attributes entries, unused entries, and cluster maps to describe the MFD. The MFD may also catalogue files for account [1,1]. The sections which follow describe each of the MFD entries. SAVE/RESTORE needed to understand most of these in order to scan and/or update directory information.

3.2.1 MFD Label Entry - The MFD label entry stores basic information required to mount and access the pack. There is one MFD label entry per disk, and it is always the first physical entry in MFD Cluster Ø. The MFD label entry is created when the disk is initialized by DSKINT.



+Ø Link to 1st Name Entry in MFD WORD ULNK

This word is the root of the list of MFD name entries. This particular link will never be null since disk initialization creates a minimal file structure consisting of two accounts (i.e., there are at least two name entries in the list).

+2 (marker) WORD

This word must be non-zero to mark this entry in use. It has no other use.

+4 (Not Used) WORD

+6 (Not Used) WORD

+1Ø Pack Cluster Size WORD

The Pack Cluster Size (PCS) was defined earlier. Briefly, PCS is the size of a Pack Cluster (PC) and a PC is the smallest unit of storage which can be allocated on this pack (i.e., each PC is represented by one bit in the SAT). PCS is specified and stored in the MFD label entry at DSKINT time.

+12 Pack Status WORD

This word contains pack status bits interpreted as follows:

<15> Bit 15 is set to 1 on a "mount" and cleared to Ø on a dismount. This bit provides some level of pack protection. If a non-mounted disk has bit 15 set to 1, then it was not properly dismounted and needs to be "cleaned" (e.g., the system crashed, or the pack was physically dismounted before it was logically dismounted).

<14> This is the Public/Private bit. A pack is declared public or private at DSKINT time. Bit 14 = 1 indicates the pack is private, otherwise it is public. System disks are always created as private; when used as a system disk, they are treated as public, but when mounted on another system, they are treated as private.

<13> Currently not used.

<12> This bit is the "New Files First" (or Link-to-Top) bit.

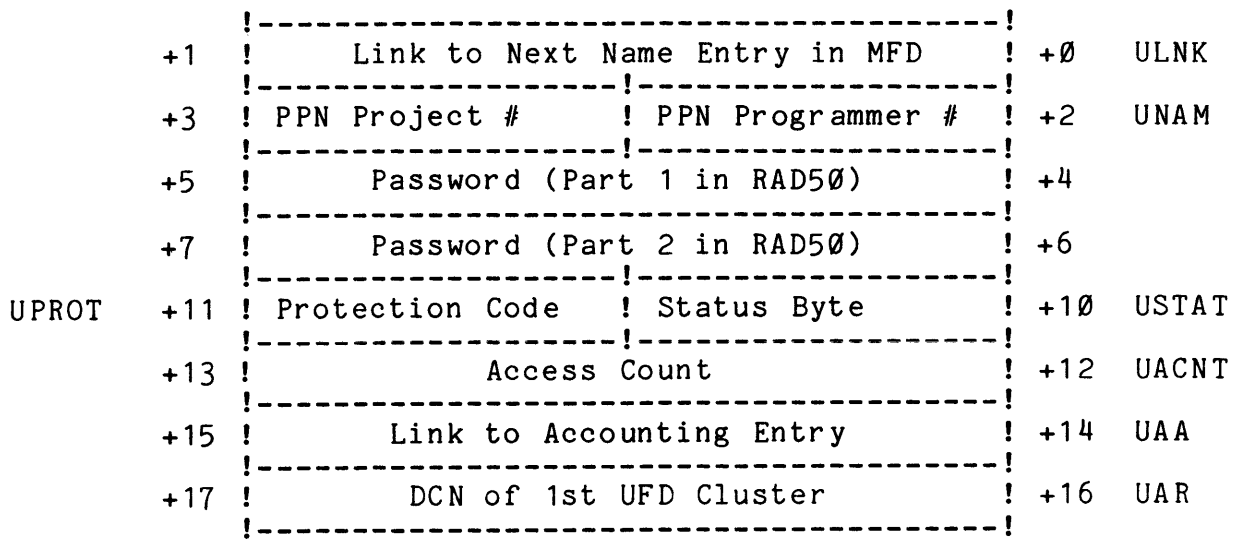
<11> This bit is the "Data of Last Write" bit. This bit governs the use and meaning of the UDLA field in a file's Accounting Entry. This is described when Accounting Entries are introduced.

<1Ø:Ø> Currently not used.

+14 Pack ID (Part 1 in RAD5Ø) WORD
+16 Pack ID (Part 2 in RAD5Ø) WORD

The Pack Id is a name specified at DSKINT time and used at mount time to verify that the correct pack has been physically mounted. The Pack Id can also be used as a logical device name so that the actual physical unit need not be known by applications code. The Pack ID may be up to 6 characters in length and is stored in two words in RAD5Ø format. The first three characters are packed into word +14, and the last three characters use word +16. SAVRES accesses these words when determining the pack id of input RSTS disks or outputting the pack id to RSTS disks.

3.2.2 MFD Name Entry - One MFD Name Entry exists for each account catalogued by the MFD. The name entry contains information needed to identify an account and to control login access to the system under this account.



+0	Link to Next Name Entry in MFD	WORD	ULNK

As stated earlier, MFD name entries are chained together as a singly linked list. The last name entry in the list will have a null link.

+2	PPN Programmer Number	BYTE	UNAM
+3	PPN Project Number	BYTE	

These two bytes store the account Project Programmer Number (PPN). The PPN is specified and stored when the account is created. The PPN cannot be [0,*] (except for [0,1]), [*,255], or [255,*].

+4	Password (Part 1 in RAD50)	WORD
+6	Password (Part 2 in RAD50)	WORD

These two words store the account password in RAD50. Passwords can be 1 to 6 characters in length. The first 3 characters are stored in word +4, and the last 3 characters are stored in word +6. The account password is specified and stored when the account is created but may be changed through a FIP directive.

+10	Status Byte	BYTE	USTAT
+11	Protection Code	BYTE	UPROT

These fields contain status and protection bits used to control read and write access to the UFD. The protection byte is currently ignored since using the MFD or a UFD as a file is a privileged operation and the normal protection mechanisms do not apply. A future potential use for these bits might be to limit cataloguing access to directories. Status bits have nearly identical interpretations in the MFD and UFD name entries. Bit level descriptions of these bytes are included later in this document.

+12	Access Count	WORD	UACNT

The current access count signals whether or not this account is in use. It is incremented by 1 every time the UFD is opened as a file and decremented upon each close. 400(octal) is added for each login under the account and 400(octal) subtracted each logout.

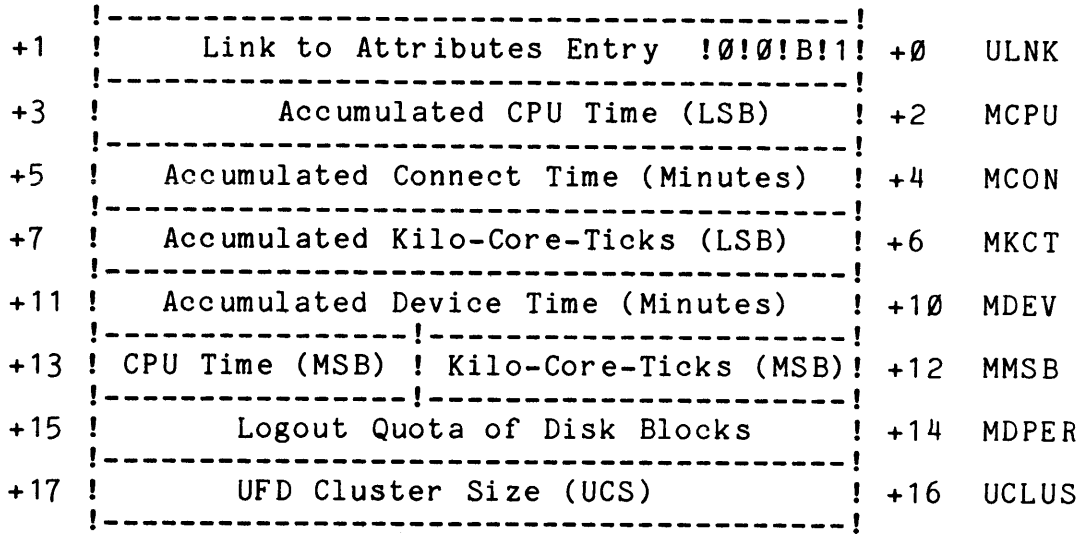
+14	Link to Accounting Entry	WORD	UAA

This link points to the accounting entry for this account. Since each account has an accounting entry, this link will never be null.

+16	DCN of 1st UFD Cluster	WORD	UAR

This word is a pointer to the first cluster of the UFD for this account. It is a Device Cluster Number (DCN). This word will be zero if the UFD does not exist. This is the case when an account is first created since the UFD is not created until a file is stored under the account.

3.2.3 MFD Accounting Entry - One MFD Accounting Entry exists for each account catalogued by the MFD. The accounting entry stores accumulated resource usage information for the account. CPU time, connect time, kilo-core-ticks, and device time counters in the accounting entry have counterparts in the Monitor's on-line tables. While the job exists, the in-memory counters are maintained by the monitor. On logout (or when the job is killed) the in-memory statistics are added to the accumulators in the MFD Accounting Entry.



+Ø Link to Attributes Entry WORD ULNK

For compatibility with the UFD accounting entry, this word is reserved for a link to an attributes entry, and 4 flag bits. The four flag bits are interpreted as follows:

<Ø> This bit is always set in the MFD accounting entry to mark this entry in use.

<1> If this bit is set, the UFD contains a bad block.

<2:3> Currently not used.

+2	Accumulated CPU Time (LSB)	WORD	MCPU
+12<15:10>	Accumulated CPU Time (MSB)	6 Bits	MMSB

Accummulated CPU time is stored in tenths of a second. Word +2 contains the least significant 16 bits of CPU time, and bits 15 through 10 (inclusive) of word +12 contains the most significant 6 bits. The 22 bit counter will store 4,194,303 ticks or 116.5 hours of CPU time. In earlier versions only a 16 bit counter was kept and it overflowed frequently. The high order bits were added to increase the time before overflow occured. The only word available in the accounting entry (word +12) was used to hold the high order bits of CPU time and KCT's.

+4	Accumulated Connect Time (Minutes)	WORD	MCON

Connect Time is the number of minutes that a job owns a console terminal. The 16 bit counter stores 65,535 minutes or 45.5 days of connect time. Connect time is updated in memory on logout, kill, and detach. Thus, when the accumulated connect time stored here is updated (on logout and kill) it will reflect the actual connect time.

+6	Accumulated Kilo-Core-Ticks (LSB)	WORD	MKCT
+12<9:0>	Accumulated Kilo-Core-Ticks (MSB)	10 Bits	MMSB

Kilo-Core-Ticks are a measure of CPU time used biased by the amount of memory required for the job. KCT's is CPU time in tenths of a second (ticks) multiplied by the (instantaneous) size of the job in K words. Word +4 contains the least significant 16 bits, and bits 9 through 0 (inclusive) of word +12 contain the most significant 10 bits. The 26 bit counter will store 67,108,863 KCT'S which is equivalent to 116.5 hours of CPU time at 16 K. The size of this counter (26 bits) was chosen to correspond to the maximum CPU time which can be accumulated multiplied by an average job size of 16 K. The KCT counter was also a 16 bit counter in early versions. KCT's overflowed much faster than CPU time, of course. The extra 10 bits were added when CPU time was expanded to 22 bits.

+1Ø Accumulated Device Time (Minutes) WORD MDEV

When a device is assigned (either by explicit assignment or a utilty assign), the time assigned is recorded in the DDB. When the device is deassigned (which may be forced by logout), the time assigned is subtracted from the current time of day, midnight correction applied if necessary, and the resulting device time is added to the in-memory accumulated device time. Device time, therefore, is the total number of minutes that devices (except the job's console) were used by the job. The 16 bit counter can record 65,535 minutes or 45.5 days of device time. Admittedly, usage of an expensive magtape drive should be weighted heavier than usage of something like a floppy. RSTS/E makes no attempt to weight device time by device type and does not store usage time for each device.

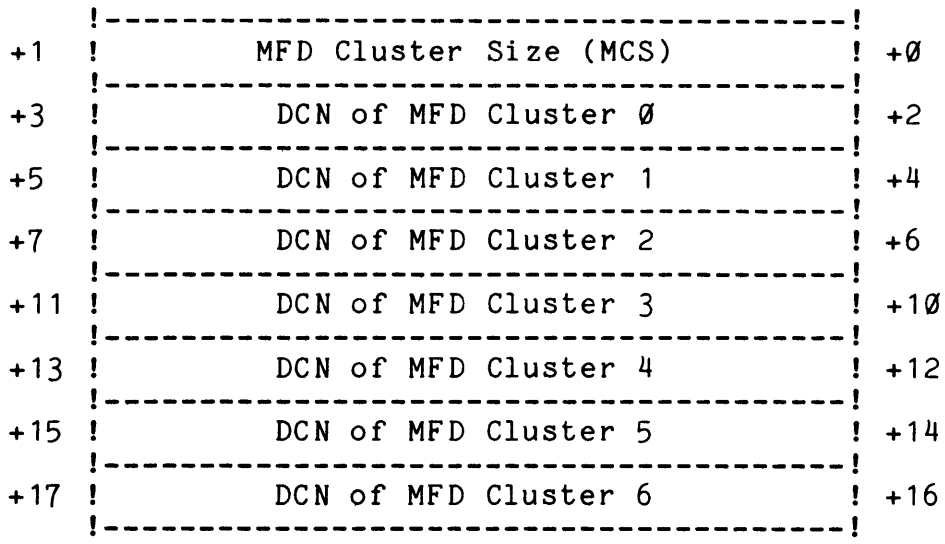
+14 Logout Quota of Disk Blocks WORD MDPER

This word stores the number of blocks which may be retained by this account at logout time. The logout quota is specified when the account is created but may be changed by a FIP Directive. The quota can have any value from Ø to 65535 blocks. A zero quota is interpreted as an infinite quota - the account is permitted to own any number of disk blocks. No quota enforcement is done by the RSTS/E monitor. The RSS utility LOGOUT enforces the quota by not permitting a job to log out until the account is below quota. The user must delete files before he can log out.

+16 UFD Cluster Size (UCS) WORD UCLUS

The UFD Cluster Size is specified when the account is created and cannot be changed without deleting the account. UCS limits the number and size of files which can be catalogued by the UFD.

3.2.4 MFD Cluster Map Entry - The MFD cluster map entry contains the retrieval pointers required to access the clusters of the MFD. There is one cluster map entry in each block of the MFD beginning at offset 760 (octal). The cluster maps in each directory block are all identical. Each time a directory is extended (which can occur a maximum of 7 times in the life of the directory), all of the cluster maps (up to 16 * 7 = 112 of them) are updated. Directory links break down into 3 bits which select one (out of 7) retrieval pointer from the cluster map, and bits which select block within cluster, and the addressed entry within the block. Given a link and the cluster map, only one disk access is required to move from one MFD block to any other MFD block.



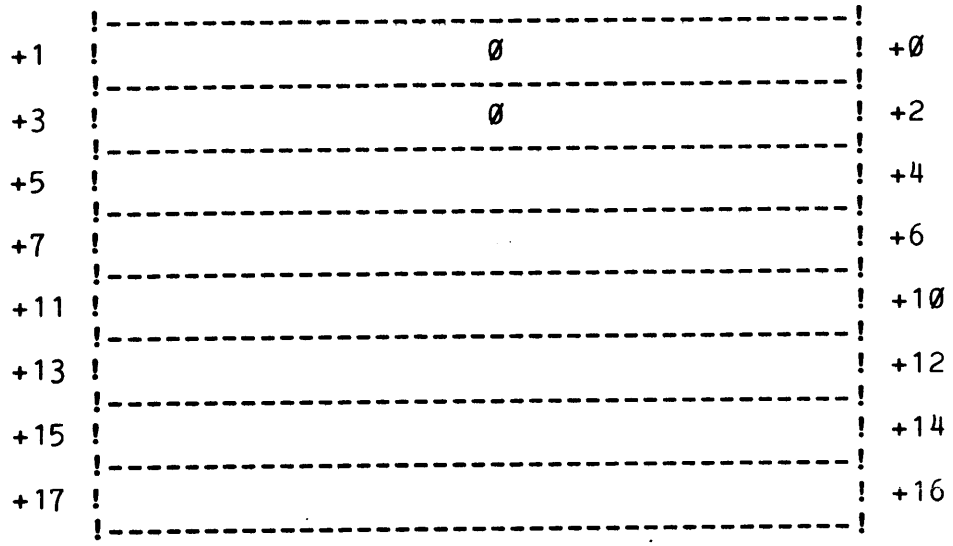
+0 MFD Cluster Size WORD

The MFD cluster size is stored in this word in every cluster map in every block of the MFD. It is used when a new link is created to an entry in the MFD. MCS is not needed to interpret a Link. MCS is specified at DSKINT time and may not be changed.

+2 to +16 DCN of nth MFD Cluster WORDS

These are the retrieval pointers to the clusters of the MFD. Non-existent MFD clusters are indicated by a zero DCN. Note that the cluster map is never sparse (i.e., if the DCN of cluster n <> 0, then the DCN's of every cluster < n are <> 0 also).

3.2.5 Unused Entries - MFD and UFD entries are allocated and deallocated as required. Obviously, there must be some way to determine which entries are in use and which are free. By convention, if the first two words of an entry are both zero, the entry is not currently in use and may be put into service as a name entry, accounting entry, or whatever. You should note in the descriptions of the various types of entries, that the first two words are never zero. The remaining 6 words in an unused entry are irrelevant.



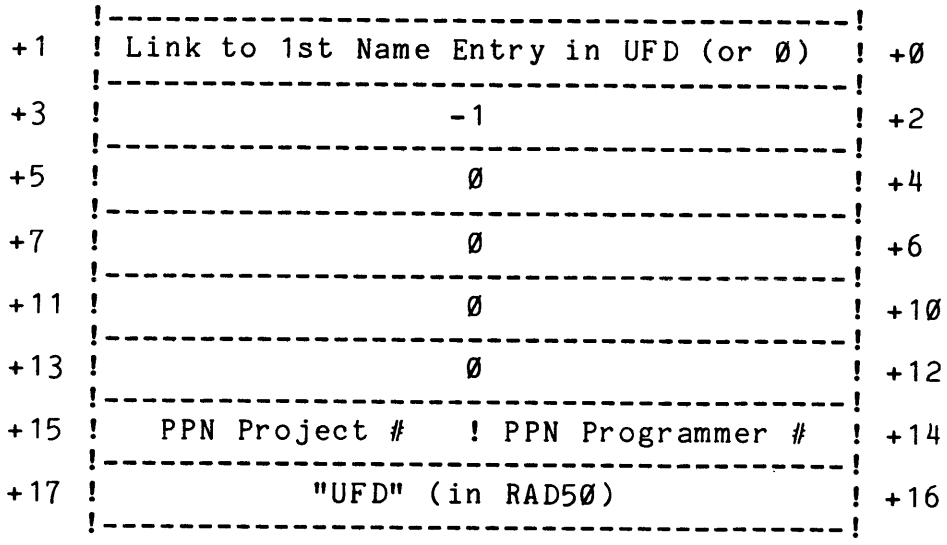
3.3 USER FILE DIRECTORY (UFD)

The User File Directory catalogues files. It stores sufficient file identification information, retrieval pointers, protection and status information to provide access to and protection of file data.

The general structure of the UFD is identical to the MFD so that common code can be used to process either type of directory. In the descriptions of UFD entries which follow, note that, whenever possible, similar entities such as links and retrieval pointers are located at the same offsets used in the corresponding MFD entries. Note also that the UFD contains name entries, accounting entries, and cluster maps like the MFD. The UFD, however, also contains retrieval entries which were not found in the MFD account describing entries. These entries are used when the actual data of a file is accessed. SAVE/RESTORE uses these file retrieval pointers, along with those in MFD and UFD cluster maps, to determine if "pieces" of a larger cluster have been moved.

The MFD name entry and the MFD accounting entry for an account are created when the account is created. The UFD is not created, however, until the first file is created under the account.

3.3.1 UFD Label Entry - The UFD label entry serves only as the root for the list of UFD name entries. The label entry is created when the UFD is created (i.e., when the first file is stored under the account). There is one UFD label entry per UFD and it is always located in the first physical entry in UFD cluster Ø.



+Ø Link to 1st Name Entry in UFD WORD ULNK

This word is the actual root of the list of UFD name entries. Like the MFD name entries, UFD name entries are chained together as a singly linked list. The link in the UFD name entry will be null if all files are removed from the account.

+2 (marker) WORD

Since the link to the 1st UFD name entry may be zero, this word must be non-zero to mark this entry is use.

+4 to +12 (Zeroes) WORDS

These words are currently not used but are reserved and must be Ø.

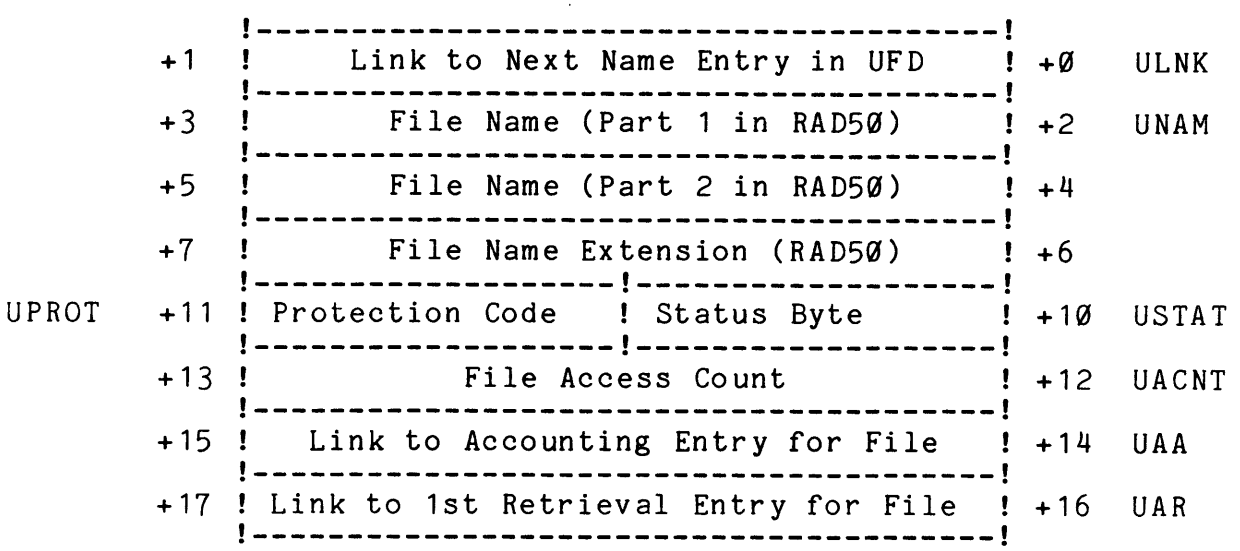
+14 PPN Programmer Number BYTE
+15 PPN Project Number BYTE

The PPN of the UFD is stored here.

+16 "UFD" WORD

The RAD5Ø of "UFD" is stored here.

3.3.2 UFD Name Entry - One UFD name entry exists for each file catalogued by the UFD. Each name entry contains the basic information needed to identify and control access to a file.



+Ø	Link to Next Name Entry in UFD	WORD	ULNK

UFD name entries are chained together as a singly linked list. The last name entry (last file in the directory) will have a null link. New files are normally added to the end of the list and hence, files are catalogued in historical order of creation (i.e., the name entry for the most recently created file will appear at the end of the list). Note that no system code depends on the historical ordering. We could change to alphabetical or reverse historical ordering in the future.

+2	File Name (Part 1 in RAD5Ø)	WORD	UNAM
+4	File Name (Part 2 in RAD5Ø)	WORD	
+6	File Name Extension (RAD5Ø)	WORD	

These three words store the file name and extension. File names may be 1 to 6 characters in length. The first three characters are stored in word +2, and the second three characters are stored in word +4. The file name extension may be Ø to 3 characters. If there is no extension, word +6 will be zero. Since the file name is required to have at least one character, the word at +2 must be non-zero. This is important since the link word may be zero and this entry must not appear to be an unused entry.

+1Ø	Status Byte	BYTE	USTAT
+11	Protection Code	BYTE	UPROT

The status and protection fields control read and write access to the file. The interpretation of bits in these fields is nearly identical for the MFD and UFD name entries. Bit definitions are described later.

+14	File Access Count	WORD	UACNT

This word is incremented when the file is opened and decremented when the file is closed. If this file is a Run-Time System, the access count is also incremented on the RTS "ADD" and decremented on "REMOVE". The access count controls file deletion. On a request to delete a file, if the access count is non-zero, the file is marked for deletion by setting USTAT <7>. The file will not be deleted until access count goes to zero, i.e. the file is deleted after the close which causes the access count to go to zero.

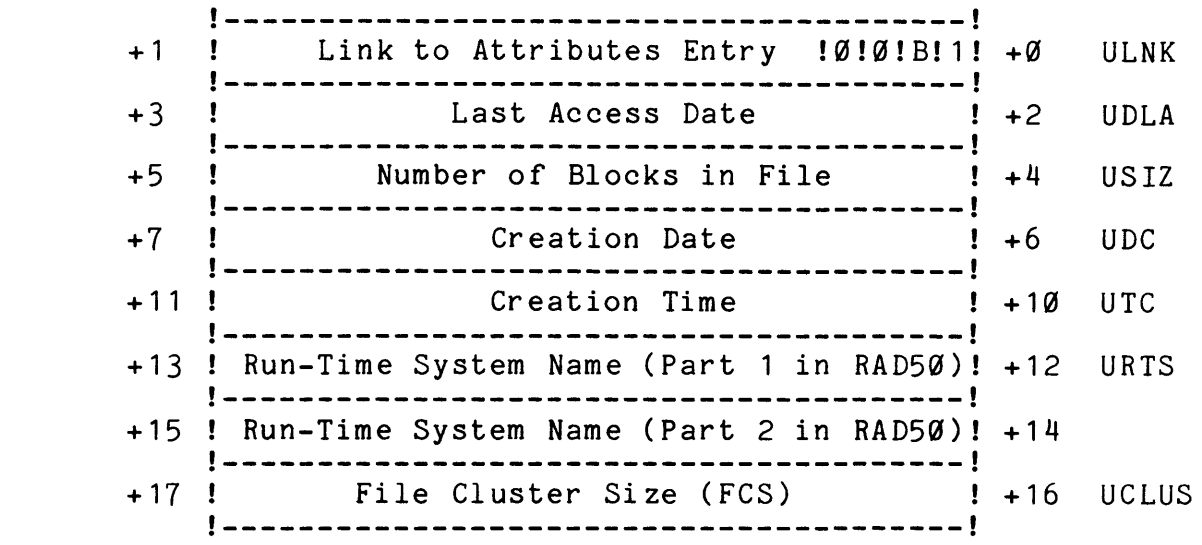
+14	Link to Accounting Entry	WORD	UAA

This link points to the accounting entry. The link will never be null since each file has an accounting entry.

+16	Link to 1st Retrieval Entry	WORD	UAR

This word is the root of the list of retrieval entries for this file. This link will be null only for a null only length file (i.e., no retrieval entries currently exist).

3.3.3 UFD Accounting Entry - One UFD Accounting Entry exists for each file catalogued by the UFD. The accounting entry stores access, length, and some static information about the file.



+Ø<15:4> Link to 1st Attributes Entry WORD ULNK

RMS file structures require attribute entries to store record and file attributes. If the file has attributes, this word contains the link to the first attributes entry. The 12 bit link field will be null if there are no attribute entries. This word also contains four flag bits as defined below.

<Ø> This bit is always set in the UFD accounting entry to mark this entry in use.

<1> If this bit is set, the file contains a bad block.

<2:3> Currently not used but must be zero.

+2 Last Access Date WORD UDLA

The last access date is set equal to the system date when the file is opened. The date is stored in RSTS/E internal format. Last access date is important to backup utilities which can do selective backup and possibly delete files which have not been used for some period of time. If the pack label indicates "Date of Last Write (DLW), this word contains the date the file was last modified.

+4	Number of Blocks in File (LSB)	WORD	USIZ

The number of blocks in the file is set to one when the first cluster is allocated and incremented by the file cluster size on each subsequent allocation of a cluster. During normal file creation and extension (i.e., the file is not pre-extended), USIZ will be smaller than the actual number of blocks in the file by some number of blocks less than or equal to FCS-1. USIZ is updated to reflect the actual number of blocks written when the file is closed. The number of blocks allocated to the file (but not necessarily written yet) is given by:

# of Allocated Blocks = FCS * INT [(USIZ+FCS-1)/FCS]

Note that USIZ is never smaller than the actual number of blocks allocated to the file by more than FCS-1 blocks.

Note also that USIZ is the complete file size on a small file system.

+6	Creation Date	WORD	UDC

This word is set equal to the system date when the file is created. The date is stored in internal RSTS/E format.

+10	Creation Time	WORD	UTC

This word is set equal to the system time of day when the file is created. Time is stored in RSTS internal format.

+12	Run-Time System Name (Part 1 in RAD50)	WORD	URTS
+14	Run-Time System Name (Part 2 in RAD50)	WORD	

or (large file system only)

+12	Zero	WORD	URTS
+14	Number of Blocks in File (MSB)	BYTE	

All executable files (UPROT <6> = 1) must have an associated Run-Time System. These two words store the 1 to 6 character RTS name in RAD50. On a small file system, both words are ignored for non-executable files. On a large file system (which may have files greater than 65535 blocks), a zero in the first word and a non-zero value in the second indicate that this is a "large file" (large files may only be non-executable). In this case, the actual size of the file is

USIZ + (MSB*65536.)

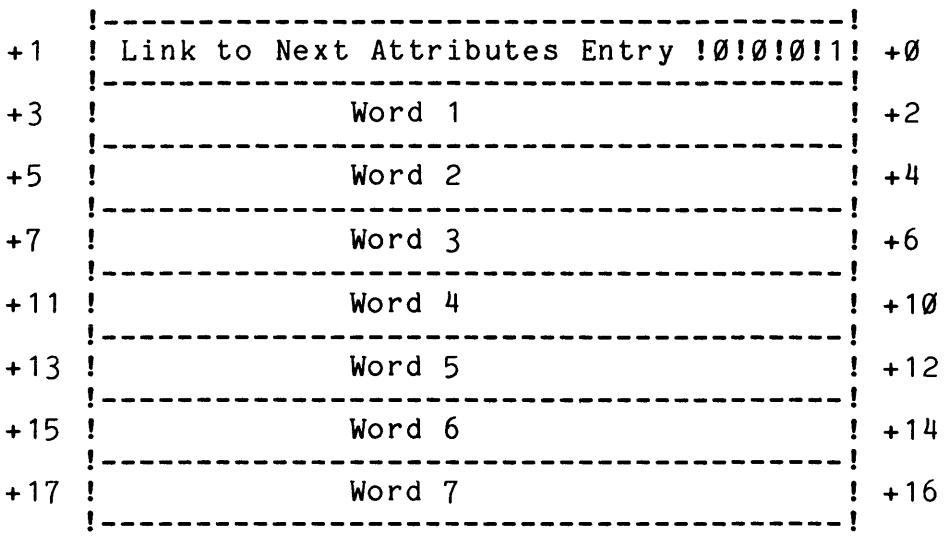
Note that an executable file must use these two words to store the associated Run-Time System name. Hence, such files may not be larger than 65535.

On any run command, the Monitor will grab the RTS name from the UFD accounting entry, lookup the RTS in its list of Run-Time System descriptor blocks, load the named RTS (if found and the RTS is not already in memory), and finally open the file on channel #15 so the RTS can load and execute it.

+16	File Cluster Size	WORD	UCLUS

File cluster size is specified when the file is created and cannot be changed without recreating the file. File cluster size can be any power of two up to 256.

3.3.4 UFD Attributes Entry - RMS file structures require attribute entries to store record and file attributes. Each attribute entry will hold seven words of attribute information.



+Ø<15:4> Link to Next Attributes Entry WORD ULNK

Attribute entries are chained together as a singly linked list. The 12 bit link field will be null in the last attribute entry in the list. This word also contains four flag bits as defined below.

<Ø> This bit is always set in the UFD attribute entry to mark this entry in use.

<1:3> Currently not used but must be zero.

+2 to +16 Attributes

These words hold the file attributes. They are read and written through FIP directives. The actual contents of the attribute words is not defined by the RSTS/E file structure. The contents of these words are inlcuded here for completeness.

Word 1 Bits are defined as follows:

<Ø:3> Record format
Ø = Undefined
1 = Fixed length records
2 = Variable length records
3 = VFC (Variable with fixed control)
4 = Stream ASCII

<4:7> File organization
Ø = Sequential
1 = Relative
2 = Indexed

<8:11> Print control
1 = FORTRAN
2 = Carriage Return
4 = VFC records contain print control
1Ø = Does not span blocks

<12:15> Unused

Word 2 Record size (actual size for fixed length records or maximum size for variable records.

Word 3 Highest virtual block number (MSB)

Word 4 Highest virtual block number (LSB) (corresponds to the file size accounting entry)

Word 5 EOF block number (MSB).

Word 6 EOF block number (LSB) (block that is the logical end of file)

Word 7 Offset to first usable byte in EOF block.

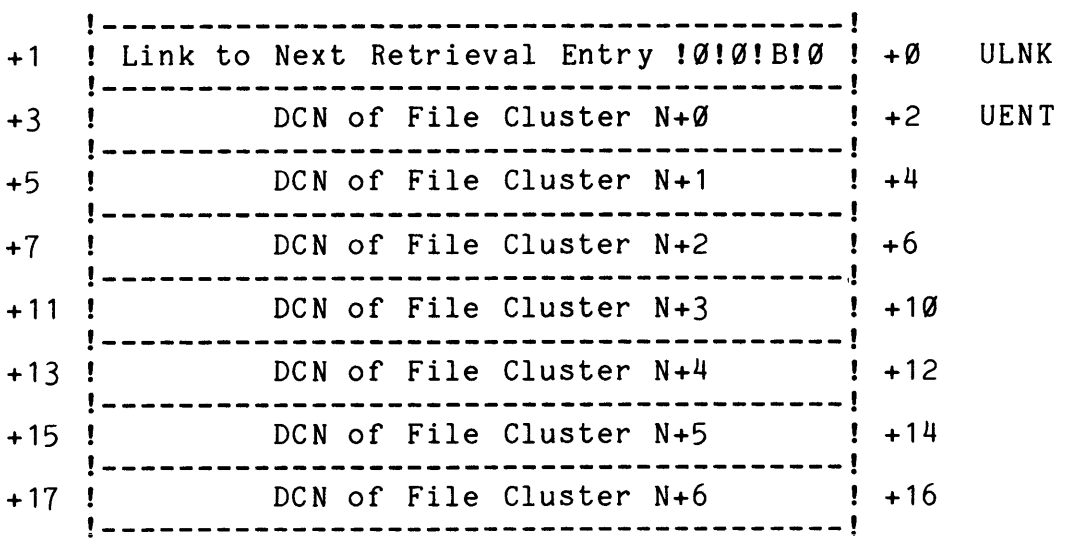
(These are in the second attributes entry)

Word 8 Number of bytes in fixed control area (high byte). Bucketsize in blocks (low byte).

Word 9 Maximum length of record actually read by RMS

Word 1Ø Default extension quantity.

3.3.5 UFD Retrieval Entry - Retrieval entries provide the necessary information to access the blocks of the file. They store retrieval pointers in the form of Device Cluster Numbers (DCN's) required to map a Virtual Block Number (VBN) to a Logical Block Number (LBN). LBN's are translated to physical disk addresses of file data by the disk subsystem.



+0 Link to Next Retrieval Entry WORD ULNK

Retrieval entries are also chained together as a singly linked list. The list can be as long as required to describe the file consistent with the capacity of the UFD. The last retrieval entry for the file has a null link.

This word also contains four flag bits as defined below.

<0> Zero.

<1> Indicates one or more at the described clusters contains a bad block.

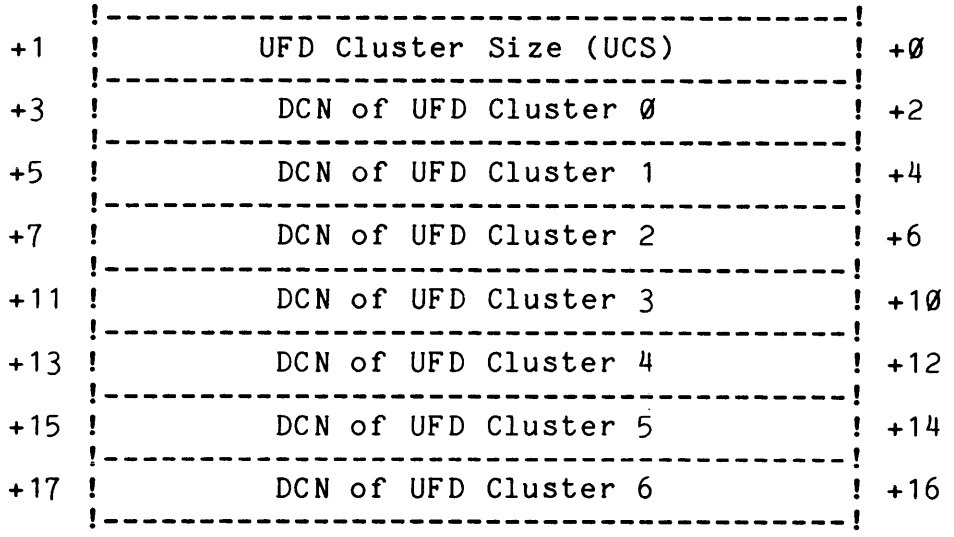
<2:3> Currently not used, but must be zero.

+2 to +16 DCN of File Cluster N+(0-6) WORDS UENT

Each retrieval pointer is a DCN of one file cluster of FCS blocks. Each retrieval entry can hold up to 7 DCN's which implies that each entry can map up to 7*FCS virtual blocks of the file.

When a file is opened, the seven DCN'S contained in the first retrieval entry are copied into the Window Descriptor Block (WCB) or Small Control Block (SCB - small file system) in memory. (Note that on RSTS/E V7.Ø, an SCB on a small file system replaces the old FCB.) These seven DCN's (or the area of the WCB or SCB which contains them) are termed a "window". With this window in memory, the first 7 * FCS blocks of the file can be accessed without any further reference to the directory. A reference to a VBN which cannot be mapped by the in-memory window will cause another window to be loaded into the WCB (possibly requiring one or more disk accesses to find the proper directory block). The process of loading a new window into the WCB or SCB is called a "window turn".

3.3.6 UFD Cluster Map Entry - The UFD cluster map performs the same function as the MFD cluster map. It contains the retrieval pointers required to access the clusters of the UFD. There is one UFD cluster map entry in each block of the UFD beginning at offset 760 (octal). As was the case with the MFD cluster maps, the maps in each UFD block are identical. Each time the UFD is extended, all of the cluster maps are updated. Given a link and any UFD cluster map, only one disk access is required to move from one UFD block to any other UFD block.



+0 UFD Cluster Size (UCS) WORD

The UFD cluster size is stored in this word of the cluster map in every block of the UFD. It is used when a new link is created to point to another entry in the UFD (UCS is not needed to interpret a link). UCS is specified and stored in the MFD accounting entry when the account is created. It is copied to the UFD cluster maps when the UFD is created (i.e., when the first file is stored under the account).

+2 to +16 DCN of Nth UFD Cluster WORDS

These are the retrieval pointers to the clusters of the UFD. Non-existent UFD clusters are noted by a zero DCN. As was the case with the MFD cluster map, the UFD cluster map is never sparse, i.e., if the DCN of cluster n is <> 0, then the DCN's of every cluster < n are <> 0. Note also that these seven words of the UFD cluster map are identical in format to a file's retrieval entry.

3.4 Directory Status and Protection Bytes

The status and protection bytes in the file and account name entries control read and write access to file data or directory information. Most of the bits apply to "files" but the status bits are meaningful in an account name entry if the UFD is opened as a file. The protection byte is ignored in a an account name entry since opening a UFD as a file is a privileged operation which bypasses the normal protection mechanisms. The bit definitions are consistent for both the file and account name entries. Wherever qualification specific to a file or an account name entry is necessary, notes are added preceded by "FILE:" or "ACCT:" as appropriate.

+1Ø Status Byte BYTE USTAT

This byte contains status bits interpreted as follows:

<Ø> US.OUT File data space is physically on another disk ("Out of SAT").

ACCT: Always Ø.

FILE: The system files SWAPn.SYS, OVR.SYS, ERR.SYS, and BUFF.SYS could historically reside on a swapping disk. Swapping disks were considered logical extensions of the system disk. Files residing on a swapping disk are included in the UFD for account [Ø,1] on the system disk.

<1> US.PLC File (or UFD) has been "placed".

<2> US.WRT File (or UFD) has given write access to someone.

<3> US.UPD ACCT: Always Ø
FILE: File currently in update mode.

<4> US.NOX File may not be extended.

ACCT: Always 1.

FILE: This bit protects the system files from extension under normal timesharing. A 1 means the file is contiguous.

<5> US.NOK File may not be killed (no delete and/or rename).

ACCT: Always 1.

FILE: This bit protects the system files from deletion and rename under normal timesharing. Only the Initialization Code can delete the critical system files.

<6> US.UFD This bit when set indicates this is an account name entry which contains information about an account, as opposed to a file name entry which contains information about a file. This allows files to be stored under account [1,1], which is actually the MFD.

ACCT: ALWAYS 1.

FILE: ALWAYS Ø.

<7> US.DEL File marked for deletion.

ACCT: Always Ø.

FILE: Files may not be deleted until the access count goes to zero (i.e., UACNT in the UFD name entry must be zero). FIP will honor a delete directive when UACNT > Ø by setting this bit. On the (last) close which decrements UACNT to Ø, the file will be deleted.

+11 Protection Code BYTE UPROT

This byte contains file or UFD protection code bits as defined below. These bits are meaningful in the MFD name entry if the UFD is opened as a file.

<8> UP.RPO Read protect against owner.

<9> UP.WPO Write protect against owner.

<1Ø> UP.RPG Read protect again group.

<11> UP.WPG Write protect against group.

<12> UP.RPW Read protect against world.

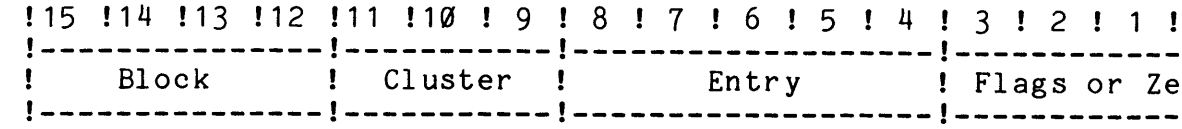
<13> UP.WPW Write protect against world.

<14> UP.RUN ACCT: Always Ø.
FILE: Executable file.

<13> UP.PRV ACCT: Always Ø.
FILE: Executable file is privileged or wipe out on delete.

3.5 Directory Links

Links were mentioned throughout the discussion of the directory structures. They are pointers to other directory entries and serve to tie the whole directory structure together. Given a link and a cluster map, it is possible to move from one directory block to any other directory block with only one disk access. The fields contained in a link are described below:



Block <15:12> These 4 bits (UL.BLO) select the block within the directory cluster. Since the maximum directory cluster size is 16, the legal range of values for this field is 0 to 15.

Cluster <11:9> These 3 bits (UL.CLO) select the 1 out of 7 possible directory clusters. They actually select one of seven DCN's from the directory cluster map. The legal range of values is 0 to 6.

Entry <8:4> These 5 bits (UL.ENO) select the entry within the directory block. The legal range of values is 1 to 30 (10) for block 0 of cluster 0 (i.e. the label entry is never the target of a link). The legal range for all other blocks and clusters is 0 to 30 (10). Note that there are 32 entries in a directory block numbered 0 to 31 (10). The cluster map entry is the 31st entry in each block and is never the target of a link.

Flags <3:0> These 4 bits are normally zero so that the entry number in bits <8:4> can be used directly as an index into the directory block (i.e., the range of possible values of bits <8:0> is 0 to 740 (8) in multiples of 20 (8). There is one exception to this rule in the ULNK word of the MFD and UFD accounting entries where the low four bits of the link are used for flags. The flag bits have the following meanings when set:

<0> UL.USE This bit marks the entry in use. Required in MFD and UFD accounting entries since the link fields may be zero (i.e., no link).

<1> UL.BAD File or UFD contains a bad block.
<2> UL.CHE Cache (Name Entry), Sequential Cache (Accounting Entry).
<3> UL.CLN Reserved for CLEAN.

3.6 DIRECTORY STRUCTURE EFFECT ON FILE ACCESS TIME

The linked structure of directories may have a detrimental effect on file access time. As accounts and files are created and destroyed, RSTS/E directories become fragmented - the unused entries become scattered throughout the existing clusters of the directory. Since files tend to be created and deleted more frequently than accounts, the effects of fragmentation are more apparent in UFDs.

As noted previously, given a link and a cluster map, it is possible to access any directory block in the UFD with one disk access. FIP owns and manages a single 256 word buffer (FIBUF) for transient processing of directory blocks. FIBUF is a system resource shared among all users.

As file directories become fragmented, it becomes necessary to create directory entries in different directory blocks. Attempts to open files in fragmented accounts will require several separate reads. Any attempt to transfer several files from the same account would make the increase in access time even more obvious.

4.Ø MINIMAL FILE STUCTURE

When a disk is initialized to the RSTS/E file structure by DSKINT, a minimal file structure is recorded on the disk. This structure includes a boot block, MFD entries for the MFD itself ([1,1]), the system account ([Ø,1]), and the system library account if the disk is to be used as a system disk. Disk initialization also creates a bad block file and a storage allocation table

4.1 BOOT Block

A bootstrap is located at LBN Ø on all RSTS/E disks. On a system disk this block contains the secondary bootstrap responsible for loading the Initialization Code. On non-system disks, the boot block contains a message printing routine which asks the user to "to please boot from the system disk".

4.2 BAD Block File

The bad block file is catalogued in account [Ø,1] as the file BADB.SYS. The DSKINT routines of the Initialization Code extract factory bad blocks (those determined by manufacturing to be unsafe) and can also perform pattern tests to detect additional bad blocks. Since the minimum unit of disk storage allocation is the pack cluster, any PC which contains a bad block is allocated to the bad block file.

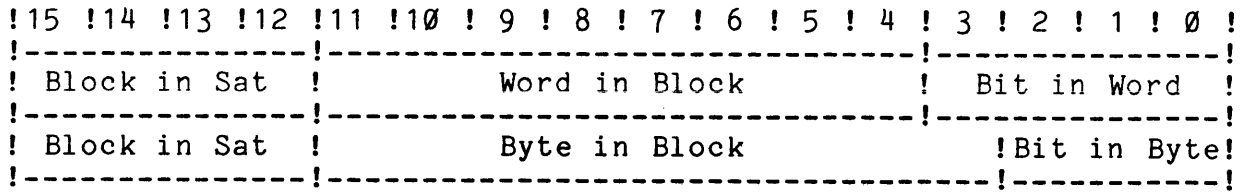
BADB.SYS is clustered at the PCS and is a standard non-contiguous file except that the file "data" is of no consequence. DCN's contained in retrieval entries serve as a list of clusters which contain bad blocks. An additional initialization code option permits BADB.SYS to be extended to include clusters which are found to be defective during normal timesharing operations.

4.3 STORAGE ALLOCATION TABLE (SAT)

The Storage Allocation Table is a bit map used to control allocation of space on a RSTS/E disk. It is catalogued in account [Ø,1] as the file SATT.SYS. The (contiguous) file is created by DSKINT and initially reflects only the space allocated to the minimal MFD, the UFD for account [Ø,1], any bad blocks detected by DSKINT, and the space allocated to the SAT itself. Each bit in the SAT represents one Pack Cluster on the disk. The appropriate bit is set to one when the PC is allocated and cleared to zero when the PC is deallocated.

Since the SAT controls allocation for the disk on which it resides, its size is dependent on the size of the disk and the pack cluster size. There are 256 words or 4Ø96 bits per SAT block. The device clustering scheme described previously ensures that there are never more than 65535 pack clusters, which further implies that there are never more than 16 blocks in any SAT.

Bits are numbered starting at Ø up to the maximum pack cluster number of the disk. The PCN serves as a bit level index into the SAT with fields broken down as shown below.



5.Ø SAVRES Notes

One of the major goals for SAVE/RESTORE was that it be "fast". In order to be "fast", therefore, SAVRES had to avoid one the overhead of the file processor. FIP transfers imply that a file must first be located in the directory structure (conceivably several separate reads) and then have the data pointed to by retrieval entries transferred.

Since the Storage Allocation Table provided an extremely easy way of finding all allocated blocks on a disk, it was chosen as the "way to go". By checking the SAT, SAVE/RESTORE could determine which pack clusters were allocated on the disk (ignoring those that were not allocated), and then use read or write routines to input or output the data immediately. Thus, this method became the basis of the design for the program.

it was decided that SAVE/RESTORE should perform three main functions:

- SAVE - transfer allocated disk clusters sequentially to an intermediate device (magnetic tape or disk)
- RESTORE - recreate a RSTS/E disk from a "SAVE Set", and
- IMAGE Copy - produce a functionally equivalent copy of a RSTS/E disk

Before a complete design of these routines could be attempted, however, several other problem areas had to be tackled.

SAVE/RESTORE was to be designed so that it produced bootable media, thus allowing users to restore data they had SAVEd. In order to do this, the output media would have to contain both a bootstrap program and a program which would be loaded and run by the bootstrap when the device was booted. Since the media could only be booted while the RSTS/E system was not running, the loaded program would have to be capable of running in an off-line environment.

SAVE/RESTORE also had to run on-line, under normal timesharing, so that users could create copies of disks on the system. The SAT-based transfer we chose necessitated two restrictions on the types of disks that could be used as input. First, the disk must be physically but not logically mounted (i.e., users could not access or change the disk in any manner while it was being copied). If this restriction were not imposed, files could be created or deleted after the transfer had started, thus making the SAT we were working with obsolete.

This restriction lead to the second. Namely, the system disk itself, which cannot be logically dismounted while a RSTS/E system is running, could not be copied on-line. Since back up of this disk was extremely critical, it amplified the need for SAVE/RESTORE to run off-line.

5.1 Initialization Code

In the RSTS/E environment, being able to run off-line usually means being able to run as a part of the RSTS/E initialization code.

The boot block of a RSTS/E system disk (or distribution medium) contains pointers to the initialization code ([0,1]INIT.SYS). When the device is booted, the initialization code is loaded into memory. The Initialization code ([0,1]INIT.SYS, or simply "INIT") is a collection of routines used to create the file structure, system files, and start-up conditions required for the normal operation of RSTS/E timesharing. INIT routines, called "options", allow the user to do such things as simulate hardware bootstraps (BOOT), copy system files to a RSTS/E disk (COPY), and initialize and perform pattern checks on disks (DSKINT).

The INIT code is essentially one large stand-alone program (written in RT-11 MACRO) with many functions. Since INIT is too large to be run in one piece, it is overlaid using the RT-11 overlay structure. The program is broken into pieces (each no larger than 8K words) which are assembled separately. The assembled pieces are then linked together with an overlay handler. One segment of the program is called the "root" and must reside in memory at all times when the program is running. The root contains various information needed by other segments of the program, which are called the overlay segments. Some of the items in the root are the overlay handler, frequently used subroutines, a mini file processor, and buffers. When an INIT segment which is not resident is referenced, the overlay handler reads the segment into memory, overlaying another segment which is not currently needed. Note that communication between separate segments must be through the root, which, as mentioned before, is always resident. While a few of the INIT options are in the root, most are in overlays.

It was conceivable, therefore, that SAVE/RESTORE could be run off-line by writing it in RT-11 MACRO and making it an INIT option. As such, it would allow the system disk to be copied off-line. In addition, if an appropriate boot and a copy of INIT were placed on SAVE/RESTORE output media, a user could indeed have an instant recovery medium.

in order to reduce duplicated code and maintenance problems, SAVE/RESTORE was structured in a manner quite similar to INIT itself. A root segment (SROOT) was written to contain data structures, device information, buffers, and other information which would be needed by different portions of the program but which would be lost when overlaying was performed. Off-line, this root was combined (linked) with the existing INIT root.

Since overlay regions in INIT are, by RSTS/E convention, restricted to 8K, each main portion of the program would be made a separate overlay segment. The main portions were defined to be the dialogue and initial mount routines (DIA), the SAVE operation (SAV), the corresponding restore operation (RES), and the disk image copy operation (IMA).

Off-line, a single new option (SAVRES) was added to INIT. When this

command is specified, INIT calls the dialogue overlay to ask all required questions and mount the specified devices.

One overlay cannot directly call a routine in a separate overlay; the root must be used for such a transfer of control. The dialogue routine, therefore, sets up a flagword indicating the operation requested and calls a special routine in INIT to dispatch to the proper transfer overlay. Upon completion of the transfer, SAV, RES or IMA returns control to the INIT main-line so that the procedure can be repeated if the user wishes to backup a diferent disk.

If this was how it was to be done off-line, a means of doing the same thing on-line was needed. This was possible by creating a special routine called the ONLine SAV and restore emulator (ONLSAV) which exists in the same overlay region as SROOT. ONLSAV routines perform the same control operations. When the on-line version of the program is run, ONLSAV sets up various items before calling the dialogue routines. As above, the dialogue is carried out and control passed back to ONLSAV so that the transfer itself can be performed.

This overall structure allows a large amount of common code to be used. Specifically, the DIA, SAV, RES, and IMA modules, all of which are quite complex, are exactly the same both on and off-line.

5.2 ONLSAV - The On-line Emulator

The structure described above serves another purpose besides the use of common code. Many things which can be done successfully off-line are either prohibited or must be done in a different manner on-line. ONLSAV provides a repository of special routines to handle these differences.

ONLSAV contains all I/O routines and data manipulation/conversion routines required by the on-line version of SAVRES. Frequently these are exact copies of routines in INIT. In addition, it handles three major INIT capabilities which are not normally available on-line.

5.2.1 Accessing the first disk device cluster - On RSTS/E, a user must "open a channel" in order to access a device. Off-line, the first device cluster of a disk (which contains the boot block) can be accessed at any point after the corresponding channel has been opened. On-line, however, it is necessary to bypass FIP for non-file-structured operations, particularly those involving the boot block. One may access this cluster only with the first I/O request to the disk.

SAVRES needs to access this cluster at several times, notably when updating the boot block at the END of a transfer. As previously mentioned, the boot block contains pointers to [Ø,1]INIT.SYS. SAVRES does not fill in these pointers until the end of a transfer since part of the file may have been relocated because of bad blocks. Because of

this requirement, the disk I/O routines in ONLSAV make a special check to see if the block being read or written contains the disk's first device cluster. If it does, the disk is reset (closed then re-opened) so that the I/O request can succeed.

5.2.2 Temporary file - SCRBUF - Off-line, it is possible to access a single disk block. On-line, the smallest amount of data which can be accessed is a device cluster, which may be up to 8 blocks. Frequently, for example, while scanning directories, SAVRES needs to read or write only one block, and, in fact, has no internal buffer space to access any more. An I/O buffer was available during such scans but the information stored in it was important and could not be lost. A method of saving and restoring this buffer before and after its use for single block extraction was, therefore, mandatory.

In addition, there are two special 16 block buffers (IOBUFF and SATBF2), outside the normal address range, which can be accessed off-line. Several routines (GETSB2/PUTSB2 and GETIOB/PUTIOB) exist for handling I/O to and from these buffers. On-line, these buffers cannot directly be used.

In order to solve both of these problems, routines were added to the ONLSAV to create and handle a 16 block temporary file (SCRBUF). This file, in combination with ONLSAV versions of GETSB2/PUTSB2 and GETIOB/PUTIOB, emulated the off-line use of IOBUFF and SATBF2. In addition, it could be used as temporary storage during a single block transfer.

5.2.3 Write-Checks - The scratch file described is also used in solving the third major problem: SAVE/RESTORE needed to perform a verify operation ("write-check" that what it had written matched what it had read). Off-line, a real "write-check" hardware function is available. On-line, the hardware "write-check" function is inaccessible. To overcome this deficiency, the "write-check" function is emulated by executing a word by word comparison of the data in two buffers, one of which is SCRBUF.

5.3 Major INIT Changes

5.3.1 Error Handling - As previously noted, the mainline code in SAVRES frequently calls external subroutines. It was critical that routines appearing in both ONLSAV and INIT return the exact same values, be they error codes, flags, or pointers. This posed a problem because many existing INIT routines that SAVRES needed to use did not allow errors. In general, they required that certain conditions be met and aborted if they were not met. For example, several routines

are used to see if a disk contains a valid RSTS/E file structure. If it is found that a specified disk does not, a fatal message is printed and the user must start again. When mounting output volumes, SAVE/RESTORE had to determine if a disk had no file structure, a RSTS/E file structure, or a SAVE volume file structure. In most cases, any of these three was perfectly acceptable. Disks not having a RSTS/E structure could simply be weeded out if the routines mentioned above could return an error indication rather than failing completely.

Many other routines, notably those used to scan directories, also had the same problem. In order to alleviate the situation, all the routines to be used by SAVRES were modified so that they would return error indicators if a SAVE/RESTORE operation was in progress and an unexpected or normally illegal condition occurred.

5.3.2 The use of magnetic tapes - Since SAVRES was to run only on RSTS/E systems, the input of a SAVE or IMAGE copy operation could be restricted to RSTS/E formatted disks. Likewise, output of an IMAGE or RESTORE would also be (reconstructed) RSTS/E formatted disks. The only major device requirement remaining was what to allow as output of a SAVE (by default, this covers input of a RESTORE). DECtape are no longer produced and were, therefore, removed from the list of possibilities. Floppies are sold but the number required to store an RPØ6 (334ØØØ blocks) would be enormous! Disk seemed an obvious device to allow. In addition, magnetic tape seemed desirable since they offer a cost effective media which can be easily stored.

Using tape had one major drawback: its use was not generally supported off-line. Prior to RSTS/E v7.Ø, the COPY option was the only component of INIT which allowed the use of tape (input only). Since RSTS/E is distributed on both tape and disk, COPY contained simple routines to copy a fixed set of files off of a tape. Since SAVRES had to both write and read tapes, this was insufficient. One alternative was to prohibit the use of tape off-line, a choice which would have enraged many users! Another alternative was to modify COPY's tape routines and include them in the appropriate SAVRES routines. This choice would have left three sets of tape routines: those in COPY, those in off-line SAVRES, and those used by all on-line programs. In order to minimize maintenance and offer complete magnetic tape support off-line, it was decided to alter the on-line tape drivers so that they could be used both on and off-line. This would remove similar/duplicate code, allow SAVRES to easily use tape output off-line, and also allow any future INIT modules to utilize tape input or output.

This use of common drivers had an additional benefit. In the past, when a tape (normally only RSTS/E distributution tapes) was booted, the secondary bootstrap was able to load the root segment of INIT along with its first overlay. Since tape is not a random access device, no further overlays could be loaded, and, therefore, no options which were not contained within the first overlay could be used. The first overlay contained those components required to

created a disk-based system (e.g., COPY and DSKINT). After using these options, a user had to boot the newly created RSTS/E disk if further action was desired.

This limitation posed a problem to SAVRES. Since it was necessary to be able to boot the output of a SAVE operation, overlays beyond the first INIT overlay (e.g., DIA) had to be accessible. This was made possible by modifications to the overlay code and the BOOT option, enabling them to use the now common tape drivers.

5.4 Device Bootstraps

The addition of common tape drivers and tape overlay capabilities made bootable SAVE sets feasible. It was also desirable that disk media be bootable. If this could be done, the output of a RESTORE or IMAGE copy operation would supply the user with an almost instant recovery medium. A user could simply boot his or her new RSTS/E disk, perhaps containing a copy of a destroyed system disk, and continue timesharing in a relatively painless manner.

In order to create bootable media, two things are necessary. One must be able to put the correct boot block on a device (boots for each device are all different) and, in the case of a disk containing INIT.SYS, one must be able to "hook" the device, i.e., put pointers in the boot block to allow INIT to be loaded and started by the bootstrap program.

The procedure for hooking a RSTS/E disk is fairly straightforward. At the end of a RESTORE or IMAGE, INIT.SYS on the output device is located in the directory in order to find the required pointers. (If INIT does not exist, as is the normal case for a private pack, the pack would not be hooked.) The pointers are then entered into the second half of the boot block.

The boot for the first volume of a SAVE set (which is the only volume that need be bootable), is handled in a similar manner. The volume does not need to be hooked but does include a boot which "points" to the program to be loaded.

The other half of the problem is obtaining the proper device bootstrap. Two main methods were considered. The COPY option of INIT contains a series of boots, corresponding to each device supported by RSTS/E. Copies of these boots could simply be put into SAVE/RESTORE code. This, however, would mean that INIT itself would have two copies of the boots. One version could be eliminated by putting special entry points into INIT so that off-line SAVRES could simply access the existing COPY boots.

This still left two copies of the boots, one off-line, one on-line. This was undesirable primarily because new devices are added each release and the boots themselves occasionally change. There would always be the risk that one set of boots would be updated while the other set was forgotten. (This did in fact happen during the first

field test) To allow the on-line version to use the same copy of boots, special pointers were included in INIT so that SAVE/RESTORE could locate the start of the boot table and scan for the appropriate boot code. note that this is just one of the reasons savres will not run on vØ6c.

5.5 Factory Bad Blocks

In general, all output devices were expected to be either RSTS/E file structured disks or volumes (either disk or tape) which had previously been used as the output of a SAVE volume. If the user's specified output was any of these, SAVE/RESTORE could find known bad blocks by examining [Ø,1]BADB.SYS on RSTS/E disks or the bad block file on SAVE sets.

If the user tried to use a disk pack which did not have a RSTS/E structure, there was no bad block file from which known bad blocks could be extracted. Many of the newer disk packs have a "Factory Bad Block" track, located at the end of the disk, which contains a list of blocks that manufacturing has determined are not safe to use. This data can be accessed off-line, and is, in fact, used by the DSKINT option of INIT. If such a pack were mounted off-line, therefore, SAVRES could get at the same information. On-line, however, the bad block track is not accessible; it is NOT possible to emulate this feature. For this reason, a user trying to mount such a pack in the on-line version is prohibited from doing so, and is told that the pack has to be initialized before it could be used.

6.Ø MORE NOTES

In a SAVE operation, a boot, various labelling information, a copy of INIT.SYS, and copies of the input SAT are stored on the first SAVE volume. All allocated clusters of the input disk are then dumped sequentially on the output medium. The last SAVE volume also contains an extra set of directory blocks, the use of which is detailed later.

In a RESTORE operation, the first SAT copy is read into an in-core buffer. (The second is kept in case a bad block appears in the first copy.) By scanning the SAT, data stored on the SAVE set can be restored to the same pack cluster it lived in on the original disk.

An IMAGE copy operation, is, basically, a combination SAVE and RESTORE, minus the intermediate volume(s).

6.1 Ease of Use

SAVE/RESTORE had to be "easy to use" for several reasons. As previously noted, a major reason was problems with the existing backup package. BACKUP has an exceedingly complex dialogue, including

several non-obvious questions that require careful examination of two manuals and frequently result in a Backup run that does not quite match what the user wanted. Since Backups are often the responsibility of operators, who cannot be expected to delve into reams of documentation, we wanted to avoid this problem.

In general, the goal was that a user could "RUN SAVRES" and be able to understand and answer all questions that were asked.

The dialogue for SAVE/RESTORE was, therefore, designed along the following guidelines. All questions come in both a short and long form, the default being the short form. If a user is not sure what the question means or what form of answer is required, a carriage return can be typed. If this is done a more detailed form of the question is printed.

Most questions also have a default answer, which can be chosen by typing a line feed.

SAVE/RESTORE also allows a user to accept default answers or back up to a previous question if necessary.

Detailed error messages are printed if the user types an unacceptable response. If this occurs, the question is repeated so that a new response may be entered.

Since even a short series of questions may seem tedious to the experienced user, alternate methods of instructing the program what to transfer are also available (switches and single line commands). SAVRES dialogue is described in detail in the RSTS/E System Manager's Guide.

6.2 "User blunder proof"

There was another SAVE/RESTORE goal which falls somewhere between "Easy to Use" and "Reliable". That is the goal of making the program as user error proof as possible. In one sense, this means being lenient enough to understand an incorrect response and allowing the user another chance to supply an acceptable answer.

On an even more important level, this means protecting the user from himself. One method of preventing intentional or accidental destruction is by restricting the use of the program. To this end, SAVRES is a privileged program, meaning it can only be run by users who have been given privileged accounts by the system manager. Several other protection mechanisms have been built into SAVRES. Some of these are described in the following paragraphs.

Since SAVRES can write output volumes, SAVRES can wipe out whatever information had been previously contained on the specified volume. In order to prevent the accidental destruction of data, SAVE/RESTORE identifies each output volume and normally asks if the user is sure that the volume can be obliterated. This feature can be disabled by

including a /SCRATCH switch on output volumes.

If the user is careless in the use of the /SCRATCH switch, it is conceivable that a newly written SAVE volume might be reused in the same SAVE operation. SAVRES prevents this by storing the date and time on which a run was started on each SAVE volume and prohibiting the use of any SAVE volume created on the same day at exactly the same time (matching the date might be easy but getting the time down to tenths-of-seconds is theoretically impossible).

In order for a RESTORE or IMAGE operation to be successful, there must be enough space on the output disk to contain all allocated clusters which existed on the original input disk. When the user performs the SAVE (or IMAGE) portion of a transfer, SAVRES checks to see what percentage of the disk is full. If this figure is 90% or more of the disk, the program prints a warning message for the user. When the RESTORE is done, SAVRES examines the output disk to see if restoration of the input disk is even possible. Specifically, it adds the number of allocated input clusters to the number of known bad blocks on the output disk to see if there is a chance the operation can succeed. Again, if this indicates the disk will be more than 90% full the user is warned. Note that if the original input warning was ignored the transfer may indeed fail.

All RSTS/E disks contain a status word which indicates if the disk was properly dismounted the last time it was used. If it was not, the pack is considered to be "dirty". Under certain conditions, packs which are marked as dirty can contain a corrupted disk structure. Since it is possible that a user may be purposely copying a corrupted disk before trying to "un-corrupt" it, SAVRES allows the disk to be used but warns the user before continuing.

Finally, SAVRES always offers the user one last chance to abort a run. This is done by asking an additional question, "Proceed (Yes or No)?" before any disks or tapes are erased. This question is asked regardless of the manner in which the user answered the dialogue questions, i.e., even if a full line command was specified.

6.3 "Handle Bad Blocks"

Bad blocks are the major reason why SAVRES had to "understand" the RSTS/E directory structure. Bad blocks may occur on either an input or output disk and may be either known (in an existing bad block file) or previously unknown.

Before proceeding with this discussion one point should be clarified. The sections which follow mention that names and accounts are reported to the user. This reporting actually happens in two steps. During an actual transfer SAVRES has no idea whether it is moving directory blocks or file blocks. For this reason, bad blocks are first reported to the user as pack cluster numbers. Upon completion of a transfer SAVRES scans directories to pinpoint the exact account and file in which a bad block occurs.

6.3.1 Bad Blocks on Input Disks - Bad blocks can occur on input RSTS/E disks or on input SAVE sets that reside on disk. Bad blocks can occur on SAVE sets in two ways. If the error is detected while reading a data block, i.e., data from a file, the block could result in the output file being corrupt. No corrective action, other than reporting the name and account of the file, can be made. It should be noted that such an error indicates that a block has "gone bad" since the time at which the SAVE set was created. If the error occurred while reading a directory block, SAVRES does try to correct the situation. When a SAVE set is written, the last volume of the set is ended by writing an extra copy of all directory blocks on the original input disk. If a bad directory block is later encountered on a RESTORE, the offending directory block is retrieved from the extra copy at the end of the SAVE set. This step is critical since a bad block at any point in a directory may render a SAVE set useless. Note that a bad block in a data block harms only that single file. In order to protect against all possible data corruption, a second copy of ALL clusters would be necessary.

If a new bad block is detected on an input RSTS/E disk, the user's copy of a file may contain bad data. In such a case, SAVRES reports the PCN of the cluster which was bad and proceeds with the transfer. At the end of the transfer SAVRES scans the input directory (starting with the MFD and progreesing through each UFD) and reports the name and account in which the bad block occurred.

6.3.2 Bad Blocks on Output Disks - If disk has been specified as the output of a SAVE operation, a bad block will never affect the integrity of data being stored. SAVRES writes SAVE sets in a "linear format". Specifically, the input SAT is scanned for bits which are set, indicating allocated clusters. If a pack cluster bit is set, the corresponding input cluster is transferred to the output disk. The cluster corresponding to the next set bit is dumped immediately after the data of the previous allocated cluster. If an output block is found to be bad, it is simply skipped, its block number stored in a file, and the transfer proceeds. Likewise, the file is scanned during the restore phase and entries skipped as necessary.

Handling a bad block on an output disk in a RESTORE or IMAGE is more complicated. Normally, SAVRES restores data to the same pack cluster on which it originally existed. Before transferring the data, the programs checks the output bad block file to see if there was an entry for the pack cluster needed. If so, the SAT is examined to see if there is a free cluster available for relocation. If there is, the sat is updated and entries are made into a pair of relocation tables, indicating the old and new pack clusters of the data. If a pack cluster was not in the bad block file, but is found to be bad while trying to write the data, the cluster is added to an in core bad block file and the cluster relocated as above.

Upon completion of the transfer, a directory pass must be made in order to track down accounts or files in which bad blocks occurred. Because data was moved, links and retrieval entries would now be

pointing to the wrong places. The directory, therefore, must be updated to reflect all relocated items. The directory is examined, starting with the first cluster of the MFD. The MFD cluster map is checked to see if any of the retrieval entries have been moved. If they were, all copies of the cluster maps are updated.

At this point, SAVE/RESTORE proceeds to scan individual accounts. The program first reads an MFD name entry into memory. This entry contains the DCN of the first UFD cluster of the account referenced in the name entry blockette. If the first cluster of the UFD was moved, this pointer is updated. The first cluster is then read in, so that the UFD and the files in the account can be checked

As in the case of the MFD, the UFD cluster map is first checked and updated as necessary. UFD label name entries are then traversed so that individual files may be checked. As each label entry is read, links to retrieval entries are followed, the retrieval entries are examined, and pointers to relocated file clusters are updated.

After an account has been thoroughly checked, the previous MFD name entry is read so that the link to the next account can be found. Each account is scanned in the same way, until all relocated clusters have been found.

This procedure neglects one thing. As was described in the section on the RSTS/E disk structure, pack clusters may also be parts of larger entity, e.g., directory clusters or file clusters. It is quite likely that the single pack cluster relocated because of a bad block was part of such a larger entity. Therefore, during the scan mentioned above, an additional check must be done. Entity cluster sizes are extracted from various parts of the directory. If a relocated item was part of an entity whose cluster size was the same as the pack cluster size, pointers can merely be updated. If, however, it was only a piece of a cluster, SAVRES must go back to the disk and "re-relocate" the entire entity cluster. This is done by examining the SAT (after deallocating the clusters currently allocated to the pieces of the entity) to see if there are free, contiguous pack clusters which add up to the size of the needed item (and which meet certain boundary restrictions). If there is room, the data is moved once again and the retrieval entries updated accordingly. If there is not, the operation is aborted.

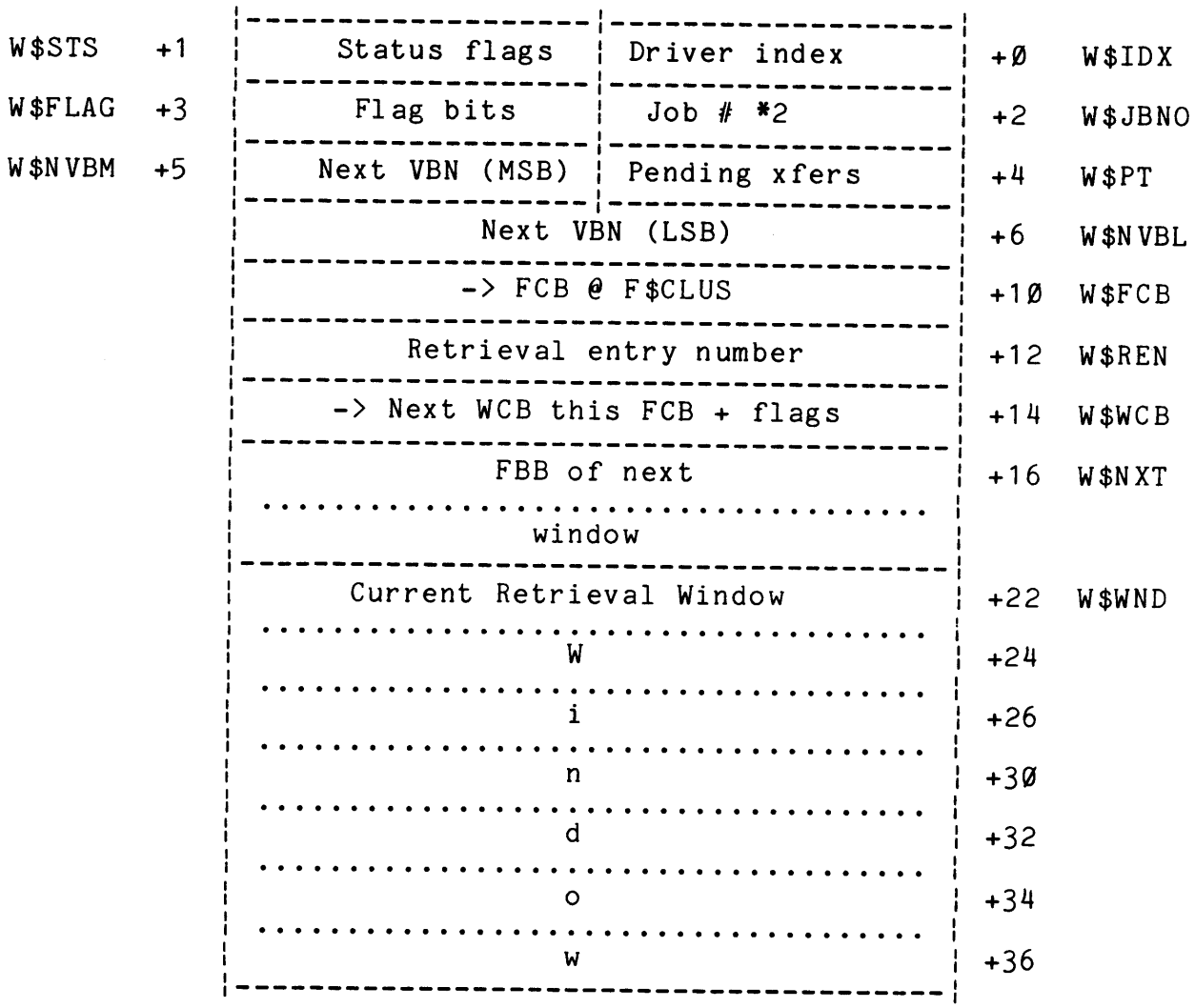
All of the above is handled transparently to the user with a few exceptions. It is possible that data relocated because of bad blocks were part of contiguous files or fell on the first cluster of a placed file. If this happens, the file's status byte is altered to indicated that the file is no longer placed and/or contiguous. Note that it would be possible to relocate an entire contiguous file. If the file were quite large, this could take a substantial amount of time. It might also be impossible to find enough contiguous space on the disk to move the entire file. In addition, there are frequently many files (such as compiled BASIC-PLUS programs) which were created contiguously to improve access time but will work even if they are made non-contiguous. It is for these reasons that contiguous files are simply "de-contigged". Critical files may be recreated to be contiguous at a later time.

APPENDIX A

WCB/FCB/SCB Layouts

A.1 Window Control Block (WCB)

On a large file system, one per open disk channel



+Ø	Driver Index	BYTE	W$IDX
+1	Status bits for file	BYTE	W$STS

Bit definitions for W$STS as a WORD (high byte):

<8> DDNFS - if set, non-file structured.

<9> DDRLO - if set, user may not read file.

<1Ø> DDWLO - if set, user may not write file.

<11> WC$UPD - if set, the file is open for update.

<12> WC$CTG - if set, the file is contiguous.

<13> WC$LCK - if set, the current block is locked.

<14> WC$UFD - if set, the file is a UFD.

+2	Job number * 2 of owner	BYTE	W$JBNO
+3	WCB flag bits	BYTE	W$FLAG

Bit definitions for W$WFLAG:

<Ø:4> WC$LLK - Length of current implicit lock.

<5> WC$EXT - if set, WCB is an extended WCB.

<6> WC$DLW - if set, update file size and date of last write.

<7> WC$NFC - if set, non-file structured in cluster mode.

+4	Pending xfers	BYTE	W$PT
+5	Next VBN (MSB)	BYTE	W$NVBM

W$PT is the pending transfer count and W$NVBM is the MSB of the next virtual block to read/write (FBN if NFS)

+6	Next VBN (LSB)	WORD	W$NVBL

LSB of the next virtual block to read/write (FBN if NFS)

+1Ø	-> FCB @ F$CLUS	WORD	W$FCB

Pointer to the FCB for the file at F$CLUS.

+12	Retrieval entry number	WORD	W$REN

Retrieval entry number of current window.

+14	-> Next WCB this FCB + flags	WORD	W$WCB

Pointer to the next WCB open on same FCB plus flag bits. Flag bits in W$WCB (as a word, rest is address):

<Ø> WC$RR - if set, the file is open 'Read Regardless'.

<1> WC$SPU - if set, the file is open in special update mode.

<2> WC$AEX - if set, always do a real extend.

<3> WC$CHE - if set, file is open for user data caching.

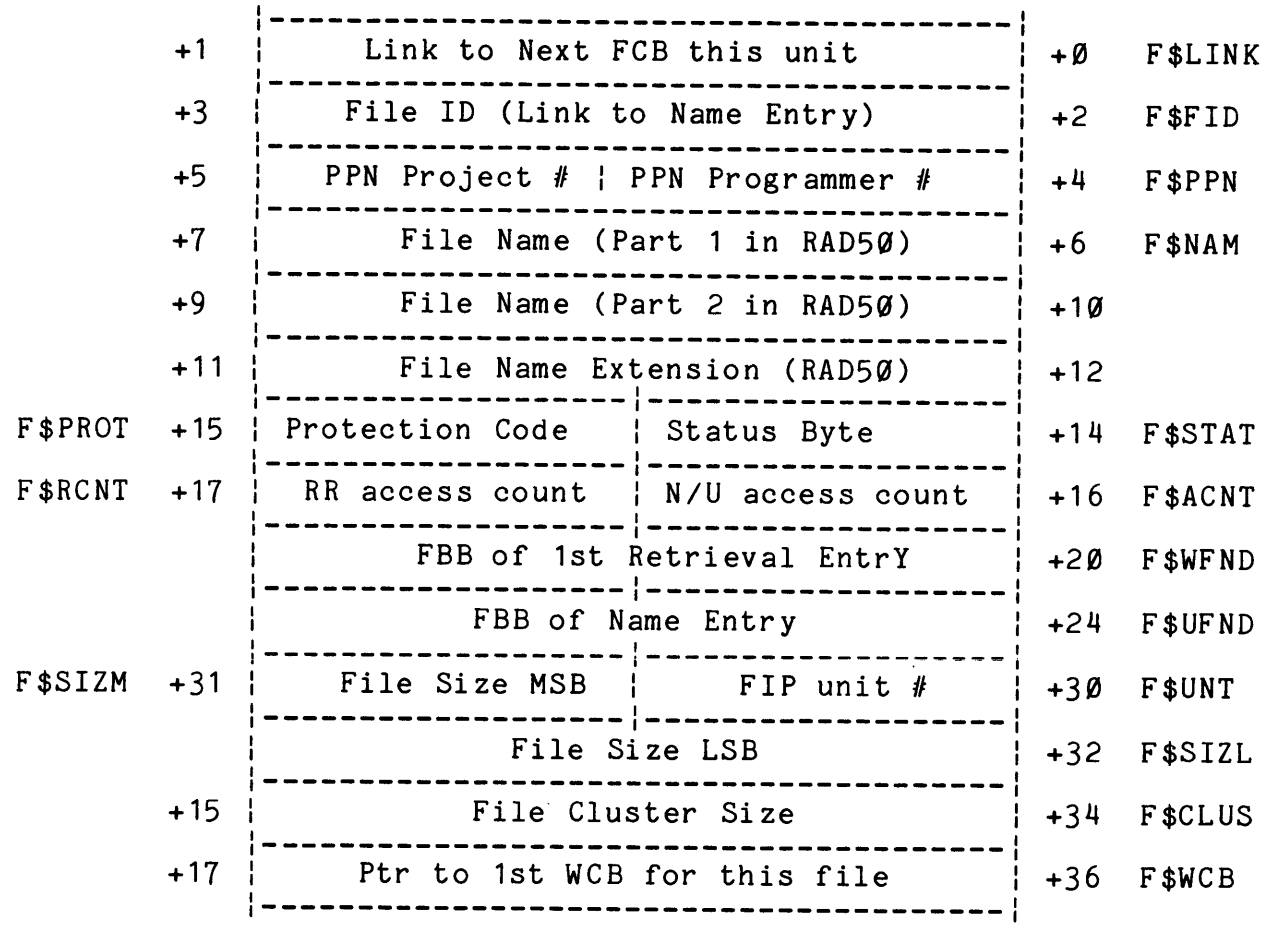
<4> WC$CSQ - if set, user data caching is sequential.

<5:15> Address of next WCB

+16	FBB of next window	W$NXT
+22	Current Retrieval Window	W$WND

A.2 File Control Block (FCB)

One per open disk file on large file system.



+Ø	Link to Next FCB this unit	WORD	F$LINK

Pointer to next FCB on this FIP unit.

+2	File ID (Link to Name Entry)	WORD	F$FID
+4	Project, Programmer number	WORD	F$PPN
+6	File Name (Part 1 in RAD5Ø)	WORD	F$NAM
+1Ø	File Name (Part 2 in RAD5Ø)	WORD	
+12	File Name Extension (RAD5Ø)	WORD	
+14	Status Byte	BYTE	F$STAT

+15	Protection Code	BYTE	F$PROT

Bit assignments are the same as USTAT and UPROT.

+16	N/U access count	BYTE	F$ACNT
+17	RR access count	BYTE	F$RCNT

Access count for Normal/Update opens and access count for Read Regardless opens.

+2Ø	FBB of 1st Retrieval Entry		F$WFND
+24	FBB of Name Entry		F$UFND
+3Ø	FIP unit number	BYTE	F$UNT
+31	File Size MSB	BYTE	F$SIZM

F$SIZM is number of FBN's if NFS.

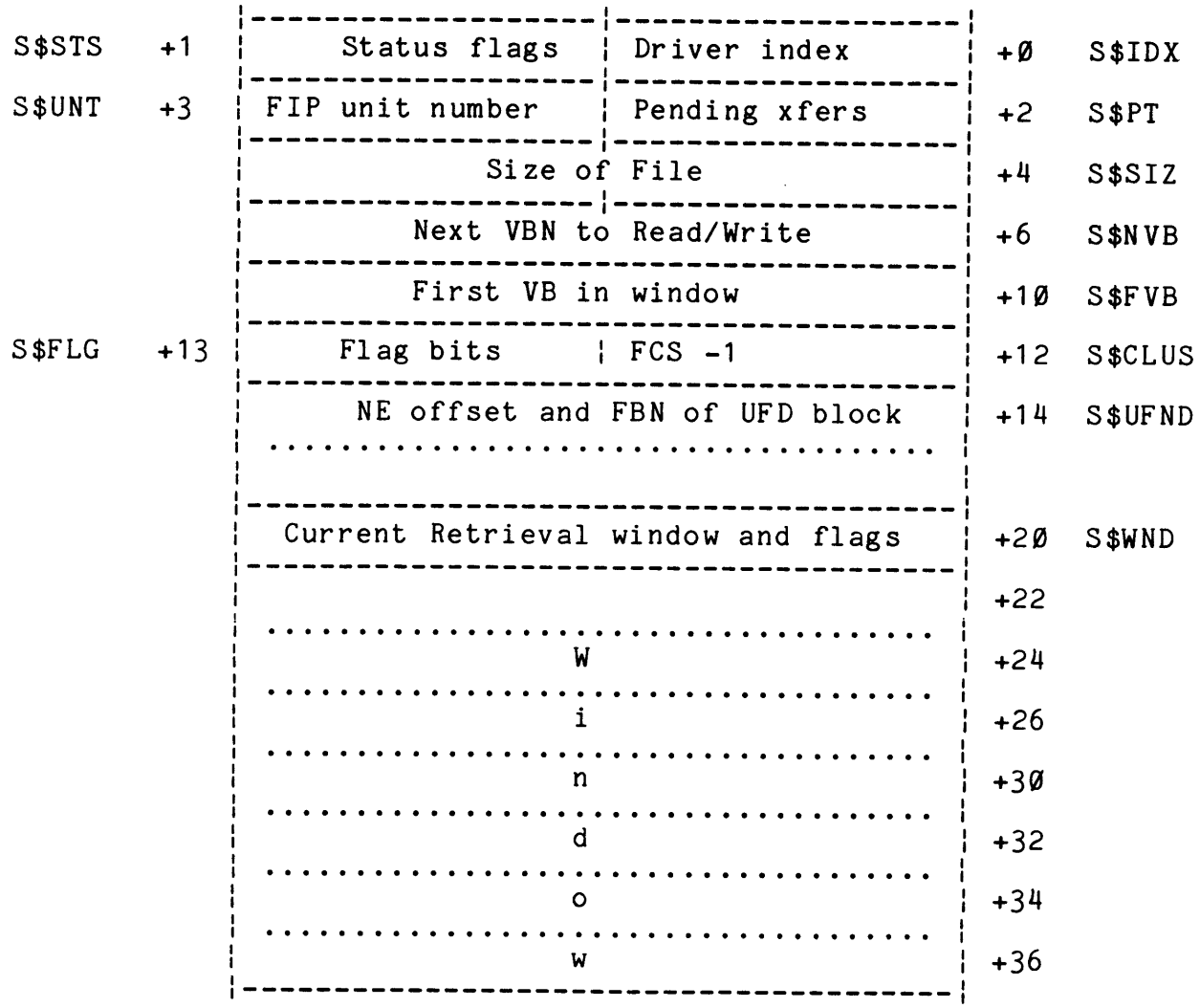
+32	File Size LSB	WORD	F$SIZL
+34	File Cluster Size	WORD	F$CLUS

W$FCB points here.

+36	Pointer to 1st WCB for this file	WORD	F$WCB

A.3 Small Control Block (SCB)

On a small file system, one per open disk channel.



+Ø	Driver Index	BYTE	S$IDX
+1	Status bits for file	BYTE	S$STS

Bit definitions for S$STS as a WORD (high byte):

<8> DDNFS - if set, non-file structured.

<9> DDRLO - if set, user may not read file.

<1Ø> DDWLO - if set, user may not write file.

<11> SC$UPD - if set, the file is open for update.

<12> SC$CTG - if set, the file is contiguous.

<13> SC$LCK - if set, the current block is locked.

<14> SC$UFD - if set, the file is a UFD.

<15> SC$USE - if set, this user has the write privileges.

+2	Pending Transfers	BYTE	S$PT
+3	FIP unit number	BYTE	S$UNT
+4	Size of file	WORD	S$SIZ
+6	Next VBN to Read/Write	WORD	S$NVB
+10	First VB in window	WORD	S$FVB
+12	File Cluster size minus 1	BYTE	S$CLUS
+13	Flag bits	BYTE	S$FLG

Bit definitions for S$FLG:

<0:4> SC$LLK - Length of current implicit lock.

<5> SC$RR - if set, the file is open 'Read regardless'.

<6> SC$EXT - SCB is an extended SCB.

<7> SC$DLW - if set, update file size and date of last write.

+14	NE offset and FBN of UFD block	WORD	S$UFND
+20	Current Retrieval Window + flags	S$WND	

Flags are:

<0> SC$SPU - if set, the file is open in special update mode

<1> SC$AEX - if set, always do a real extend

<2> SC$CHE - if set, file is open for user data caching

<3> SC$CSQ - if set, user data caching is sequential

Using RSTS/E Resident Libraries

Mark Goodrich

April 25, 1980

Spring DECUS U.S. Symposium

I. INTRODUCTION

A. Purpose

II. REVIEW OF CONCEPTS

A. Address Space
B. APR
C. Memory Mapping
D. PLAS Directives
E. TKB

III. WRITING SHAREABLE CODE

A. Position-Independent Code (PIC)
B. R/W Impure Areas
C. Memory Resident Overlays
D. Structure of Library
E. TKB
   1. Global Entry Points
   2. APR's

IV. CONVERTING EXISTING APPLICATIONS

A. Resident Library vs Run-Time Systems
B. Overlaid Programs
C. Higher Level Languages
D. MACRO Programming
E. Examples
   1. Supported libraries.
   2. Converting and editor
      a. Separating Code and Data
      b. 3-link Trick
F. Read/Write Resident Libraries

V. WRITING NEW APPLICATIONS

A. Resident Commons
B. Manual PLAS mapping

VI. SUMMARY

A. Potentials
B. Costs

REVIEW OF CONCEPTS

Address Space

The PDP-11 processor handles 16-bit operand addresses, so the address range is from Ø through 2^16-1 = 64K bytes, or 32K words.

All RSTS/E systems use the memory management feature available on PDP-11/34 through 11/7Ø processors.

APR

An APR consists of two 16-bit registers. These registers define a "page" of contiguous memory. The Page Address Register (PAR) defines an actual memory location where the page begins. The Page Descriptor Register (PDR) defines, among other things, the maximum length of the page and how it can be accessed (read/write, read-only).

The PAR is made up of two fields: a 3-bit APR number and a 13-bit byte offset. Thus, each APR can map a virtual address range of up to 2^13 = 8192 bytes, or 4Ø96 words, and there are 8 APR's (Ø-7) which map the entire 32K word address space.

Memory Mapping

A job is mapped by at least one and by at most 8 APR's. The job's low segment is mapped read-write. The job's high segment is usually mapped read-only. The job's middle segment (Resident Library) is mapped according to the usage applied by the job to the library.

PLAS Directives

ATRG$ - Attaches the job to a resident library. Specified is the type of access, and the name of the resident library.

DTRG$ - Detaches the job from a previously attached resident library. Any windows mapped to the library are unmapped and eliminated.

CRAW$ - Creates a window (a range of virtual addresses) to be used to map to a range of actual addresses in an attached resident library. Windows begin on APR boundaries and can specify more than a 4K word range.

ELAW$ - Eliminates an address window that was created by the job, unmapping the window if necessary.

MAP$ - Maps a previously created address window to an attached resident library. This relates the virtual address range defined by a CRAW$ directive to actual locations in memory within a resident library.

UMAP$ - Unmaps a specified address window from a resident library. The unmapping does not eliminate the window, nor does it release the APR's used by the window.

Resident libraries are built using the "/-HD" switch on the task file name and a symbol table file must be created to link subsequent tasks as shown below:

```
TKB>LIB/-HD/PI,LIB,LIB=LIB
TKB>/
TKB>STACK=Ø
TKB>PAR=LIB:Ø:2ØØØØ
TKB>//
```

The "/PI" is optional and signifies that the code is position-independent code. The resultant .TSK and .STB produced are used by TKB when linking other tasks to the resident library.

When you build a task that links to a resident library, you must indicate to the Task Builder the name of the resident library (1- to 6-characters), the type of access desired (read/write or read-only), and optionally the first APR that TKB is to allocate for mapping the library into the task's virtual address.

Four options are available for this action:

```
RESLIB = dev:[p,pn]filenm/RO:[apr]   (User Resident Library)

RESCOM = dev:[p,pn]filenm/RO:[apr]   (User Resident Common)

LIBR   = filenm:RO:[apr]             (LB: Resident Library)

COMMON = filenm:RO:[apr]             (LB: Resident Common)
```

WRITING SHAREABLE CODE

Position-Independent Code (PIC)

The Task Builder binds one or more modules together to create an executable task image. Once built, a task can generally be loaded and executed only at the virtual address specified by the task builder at link time. Such a task is considered position-dependent.

However, it is possible to write code that is not dependent on the virtual addresses to which it is bound. Such a body of code is termed position-independent and can be loaded and executed at any virtual address. This is especially useful when the code is to be shared in a single physical copy of common code. This allows the code to be placed anywhere within a task's virtual address space when linked to by the task builder.

When two libraries are built non-PIC at the same virtual address only one can be referenced by any one task. If both libraries, or at least one of the libraries were built PIC, then both libraries could be referenced by the same task.

The construction of position-independent code is closely linked to the proper usage of PDP-11 addressing modes.

1. All addressing modes involving only register references are PIC.

```
MOV     (RØ)+,2(R4)     ;RØ AND R4 ARE ABSOLUTE PTR.
```

2. Relative addressing modes are PIC when a relocatabe address is referenced from a relocatable instruction.

```
MOV     #1,FIRST        ;FIRST IS RELOCATEABLE.
MOV     #1,FIRQB        ;FIRQB IS ABSOLUTE.
```

3. Immediate mode references are PIC only when then value is absolute.

```
MOV     #FIRST,RØ       ;NON-PIC REFERENCE.
MOV     #FIRQB,RØ       ;PIC REFERENCE.
```

4. Absolute mode addressing is PIC only where an absolute virtual location is being referenced.

```
MOV     @#FIRQB,RØ      ;PIC REFERENCE.
MOV     @#FIRST,RØ      ;NON-PIC REFERENCE.
```

Use MOVPIC macro from COMMON.MAC on distribution kit to change non position-independent instruction to position-independent instruction.

```
MOV     #FIRST,RØ       ;NON-PIC REFERENCE.
MOVPIC  #FIRST,RØ       ;PIC REFERENCE.
 MOV    PC,RØ           ;GET CURRENT PC (GENERATED)
 ADD    #FIRST-.,RØ     ;ADD IN OFFSET  (GENERATED)
```

R/W Impure Areas

When writing code that is to be shared, don't include read/write data within the same program section (.PSECT). Separate data into R/W PSECTs and code into R-O PSECTs.

```
.PSECT  CODE,I,RO
.PSECT  DATA,D,RW
```

Memory Resident Overlays

It is important to be careful in choosing whether to have memoryresident overlays in a resident library. Careless use of these segments can result in inefficient allocation of virtual address space. This is because the task builder allocates virtual address space in blocks of 4K words. Consequently, the length of each overlay segment should approach that limit if you are to minimize waste. (A segment that is one word longer than 4K words, for example, will be allocated 8K words of virtual address space. All but one word of the second 4K words will be unusable.

The primary criterion for choosing to have memory-resident overlays is the need to save virtual address space when disk-resident overlays are either undesirable (because they would slow the system down), or impossible (because the segments are part of a resident library).

Structure of library

A resident library can be shared code, shared data, or both. It can be read-only or read-write. It can take up only as much physical address space as virtual address space (with no memory resident overlays) or consume much more physical address space than virtual space (with memory resident overlays).

TKB

When creating a resident library, only the global symbols that exist in a root segment of a memory resident overlay library are put in the symbol table. In order to access the entrypoints which may be in the memory resident overlays, the user must add the entrypoints into the symbol table by use of the GBLREF option.

The format of the GBLREF option is:

```
GBLREF=symbol-name:.....symbol-name
```

where symbol-name is a 1-6 character name of a global symbol in a memory resident overlay.

When assigning APR's, TKB will start from APR 7 and allocated downward towards the task image. Also, when a run-time system is to be linked with the task, it must appear first in the options before any other resident library options.

CONVERTING EXISTING APPLICATIONS

Resident Libraries vs Run-Time Systems

Previous to Resident Libraries, the only way to share code was to write a run-time system. This had two main drawbacks:

1. learning the secrets of what a run-time system does; and

2. no overlays allowed in run-time systems.

With Resident Libraries, code can be easily made shareable, and overlays are allowed (memory resident only, still no disk overlays within Resident Libraries).

In addition, libraries can link to other libraries just like tasks.

Overlaid Programs

To convert an un-overlaid program to a resident library is usually a simple matter of task building the object modules with the "/-HD" switch and create the resident library file with MAKSIL. If data is required by the resident library, then references must be satisfied at the time of building the library. This usually means changing some code in the library to eliminate global references to local data and substitute using pointers to the data instead.

An existing overlaid program requires more careful work to make it into a shareable resident library. This is complicated by the 4K word boundary condition mentioned previously. Code will usually have to be re-structured to fit code which previously existed as disk overlays, into self-contained 4K word modules.

No easy solution exists. Knowing calling sequences between the code withing the overlays helps tremendously, as does duplicating some routines within several different memory resident overlays to satisfy calling sequences within the 4K word boundary.

Higher Level Languages

Languages like BP2, COBOL, and FORTRAN on the PDP-11 do not lend themselves to sharing user-written subroutines. What can be shared is the Object Time System (OTS) written to support the language compiler.

There is a released BASICS resident library which allows the BP2 user to share some of the code currently linked into every BP2 task. This has the potential of reducing the task size by as much as 8K words in the low segment, and a corresponding saving on the disk image size.

Also, shared R/W COMMONs (ala FORTRAN and BP2) are possible with resident libraries. BP2 programs can link to a resident library and access the data using the COMMON or MAP language constructs.

There are no plans for BASIC-PLUS programs to be able to use resident libraries.

Currently, the full usefullness of resident libraries can only be exercised by programs written in MACRO.

MACRO programming

With access to the PLAS directives supported through either the RSX run-time system or the interal Monitor EMT's, a MACRO program can control more completely the mapping of windows within the task.

TKB will automatically set up calls to map the resident libraries linked to by a task. This does mean, however, that a user is restricted to the number of libraries that can be accessed by the task. TKB will not allow linking to more libraries than will simultaneously fit within the task virtual address space. This means that if a task desires to access two libraries which are linked to the same APR range, or which together span more than the remaining virtual address space, TKB will not permit it. However, using the PLAS directives, a task can manually attach to first one library, create an address window, map to it, and then attach to another library and use the same address window to map to the second library.

When this approach is taken, then any calls to global entrypoints in the mapped libraries must be accomplished without the help of TKB. It is simple enough though, to use common vector tables to be able to access code or data in the library.

For the most part, linking automatically by TKB will serve most users needs for resident library applications. But if the need arises for more complex mapping applications, the mechanisms are provided through the PLAS directive interface.

Examples

With V7.Ø and BP2 V1.6 we now have several examples of working resident libraries:

1. RMSRES - a 2-APR 24K word memory resident overlay library.
2. RMSSEQ - a 1-APR 4K word sequential resident library.
3. BASICS - a 2-APR 8K word sequential resident library.

These can be routinely linked to by TKB to produce sharing of code with a corresponding reduction of task image size and potentially physical memory requirements.

If a user has an existing application that is desired to convert to a resident library, there are two approaches that can be taken.

Take the case of an editor. What may be desired is the old runtime approach where a copy of the R-O code is in memory once, but the R/W data area is duplicated in each users task. The first approach that could be tried is to link all the code into a library and link the task which contains the code separately against the library. This does require that the code in the library must be able to link without direct reference to the symbols of the impure data area, since those data program sections were not built with the library. This could result in considerable conversion to change the code to access the data in a more general fashion then simply being able to know the fixed address of the data.

But there is a second approach which lends itself to an easy solution to this problem. This is the 3-link trick. This method allows existing code to remain unchanged and still access data symbols in the user's task.

The user first links a .TSK and .STB which contains only the symbols for the R/W data areas and a call to the entrypoint in the resident library. For example:

```
TKB EDT,,EDT=EDT/LB:DATA:ENTER
```

This will result in an "UNDEFINED REFERENCE" to the entrypoint in the library but WILL produce a symbol table which reflects all the global data areas in the task.

Next, the user links all the R-O code into the resident library, and includes the EDT.STB file produced by the first link. For example:

```
TKB>EDTLIB/-HD,EDT,EDT=EDT.STB,EDT/LB:CODE
TKB>/
TKB>ENTER OPTIONS:
TKB>STACK=Ø
TKB>PAR=EDTLIB:4ØØØØ:14ØØØØ
TKB>//
```

This will result in all code being assembled with external references to the R/W data satisfied by the EDT.STB from the first link.

Finally, the user re-links the R/W data task using the library just produced to satisfy the global entrypoint in the resident library. For example:

```
TKB>EDT,EDT=EDT/LB:DATA:ENTER
TKB>/
TKB>ENTER OPTIONS:
TKB>RESLIB=EDTLIB/RO
TKB>//
```

This has accomplished what putting the code into a run-time system used to do but with a lot less effort. Each user who now runs EDT.TSK will only duplicate the R/W data and not the code which will be resident once in physical memory.

Of course the R-O code in the library now is linked to absolute locations in each user's task, so anyone else linking to this resident library to possibly further share the code MUST have the R/W data areas set up exactly like those in the linked EDT.TSK.

Read/Write Resident Libraries

Another example of adapting existing programs to resident libraries would be the set of applications which use either SEND/RECEIVE or a disk workfile to pass data between programs. A R/W resident library could be linked to those programs and the information passed directly into a R/W region of physical memory which is being shared by those programs which access to the library. There are a few drawbacks with this approach: namely, even the smallest R/W resident library will take 4K words minimum out of each tasks virtual memory since this is the boundary size for each window. Also, there are no LOCKing or semaphore mechanism to keep each task from writing over shared data. An explicit locking mechanism has to be code as part of the R/W data to keep shared information from being overwritten by mutiple running tasks.

However, if the virtual address space lost can be utilized effectively in a R/W environment, this method could enhance inter-job communication.

Resident Commons

In BP2, COMMONs are used effectively to pass information between the main program and any sub-program linked with the main program. An extension of the R/W Resident Library can allow the BP2 COMMONs to map a R/W Resident Library, giving to BP2 programs as well as MACRO programs a means of communicating between jobs.

This is effected by naming the COMMON to the same name as the .PSECT name of MACRO program which allocates R/W data space for the Resident Library. The following example shows the interaction between a MACRO source which is made into the resident library and a BP2 program which can then access the data.

MACRO source = BUFFER.MAC

```
        .TITLE  BUFFER
;
.SBTTL  RESIDENT LIBRARY DATA BUFFER
;
        .PSECT  RINGBF,D,RW
;
LOCK::  .WORD   -1              ; >Ø = LOCK IN USE
INPTR:: .WORD   Ø               ; INDICATES WHERE TO STORE DATA
RING::                          ; RING BUFFER DATA
        .REPT   1Ø.
         .WORD  Ø               ; CHARACTER COUNT
         .BLKB  128.            ; LINE BUFFER
        .ENDM
RINGSZ=.-RING

        .END
```

BP2 source = READER.B2S

```
1       EXTEND
2Ø      COMMON(RINGBF)  lock%,inptr%,ring$(9%)=13Ø%
                        ! >Ø = lock in use
                        ! next free buffer address
                        ! ring buffer
                        !   ring$(i%) [1:2]   = character count
                        !   ring$(i%) [3:128] = line buffer
1ØØ     PRINT "lock="; lock%; " inptr="; inptr%
        \ PRINT "index  count   text"
        \ FOR i%=Ø% TO 9%
        \   count% = SWAP%(CVT$%(LEFT(ring$(i%),2%)))
        \   PRINT i%; " "; count%; "    "; MID(ring$(i%),3%,count%);
        \   PRINT IF CCPOS(Ø%)
        \ NEXT i%
        \ GOTO 32767
32767   END
```

The TKB command files to build the library and the task are:

```
TKB>BUFFER/-HD/PI,,BUFFER=BUFFER
TKB>/
TKB>ENTER OPTIONS:
TKB>STACK=0
TKB>PAR=BUFFER:0:4000
TKB>//

TKB>READER,READER=READER,LB:BP2COM/LB
TKB>/
TKB>ENTER OPTIONS:
TKB>RESLIB=BUFFER/RW:6
TKB>UNITS=12
TKB>ASG=SY:5:6:7:8:9:10:11:12
TKB>//
```

This example shows the BP2 program reading the contents of the buffer to see what has been deposited. Any other BP2 program with the same COMMON statement could equally as well deposit into the buffer.

The read-write resident library MUST be made PIC (fortunately it's all data anyway) because only PIC libraries will have the .PSECT symbol stored in the symbol table as well as the global symbols. The BP2 program accesses the data by position alone. When linked against the library, TKB will position the references to the COMMON data in the BP2 program to coincide with the actual address that reside in the library.

Manual PLAS mapping

As mentioned previously, MACRO applications have the ability to extend the virtual address space of the task to include a large number of resident libraries, but only when circumventing TKB's automatic allocation of APR's. Manually attaching and mapping to resident imposes the burden of knowing the structure of the data or code being mapped. With sufficient memory, whole disk data files could reside in memory and be accessed by a database package written to using the mapping directives instead of updating records on disk.

SUMMARY

Potentials

Resident libraries can be used effectively on some of the smaller systems, being careful to stay away from the memory resident overlay resident libraries, but the real potential is for larger systems with lots of memory. The more code or data that resides in memory can greatly speed up applications which are diskbound due to heavily overlaid tasks, or applications which can take advantage of the R/W COMMONs.

Using the PLAS directives and bypassing TKB's automatic loading has the greatest potential for increasing a program's virtual address space, but also carries with it the most burden of resolving access to global symbols by the task itself.

Costs

Memory resident overlay libraries use lots of memory. The RMSRES library alone uses 24K words. As more and more libraries are used by a single task, the swapping activity can increase to bring in a task because all the libraries must be resident when the job is running. When the job swaps out, then the libraries are candiates to swap out as well, unless marked as being permanently in memory.