RSTS/E SOFTWARE SUPPORT NOTES

Contributions by:

M. Bramhall, J. Miller, J. Wooldridge, M.Smith,
J. Jurgens, M. Duda, P. Goyette, G. Alles,
D. Pavlock, G. Wolfendale, M. Minnow, T. Sarbin,
J. Barclay, J. Berman, and S. Johnson

Please direct any comments or requests for further copies to....

Gary Alles
PK3-1/S44
X4954

CTS 500 AND RSTS-11 VERSION 4
OBSOLETE PRODUCTS

John Sudduth
x3551 MA

CTS-500 and RSTS-11 Version 4 are obsolete products and have been removed from the DEC Standard Price List. No orders will be accepted for this software by the Corporation. The recently announced DATASYSTEM 535 provides a 4-user CTS-500/E (RSTS/E) system on a 11/40 with 96K bytes, LA36 console, and two RK05 disks at a packaged price that will meet the needs of the market for a low cost, entry-level system.

CONFIDENTIAL

CONTENTS

TABLES

BLOCKS

SPECIAL MONITOR INFO

WORDS

DIRECTORY STRUCTURE

CIL MAP

DIAGRAMS

JOB STRUCTURE

CORE MAP OF MONITOR

MONITOR ROUTINE NARRATION

PROGRAM OPTIMIZATION

DEVICE DRIVER INFO

ERROR LOGGING AND CRASH ANALYSIS

2780

VT50/VT05 Info

DH11 INFO

HARDWARE

ODT

BASIC - PLUS FILE PROCESSING

TABLES

Job Statistics Buffer Conditionally Assembled

At Location 234_8 (156.) is a three-word table of pointers

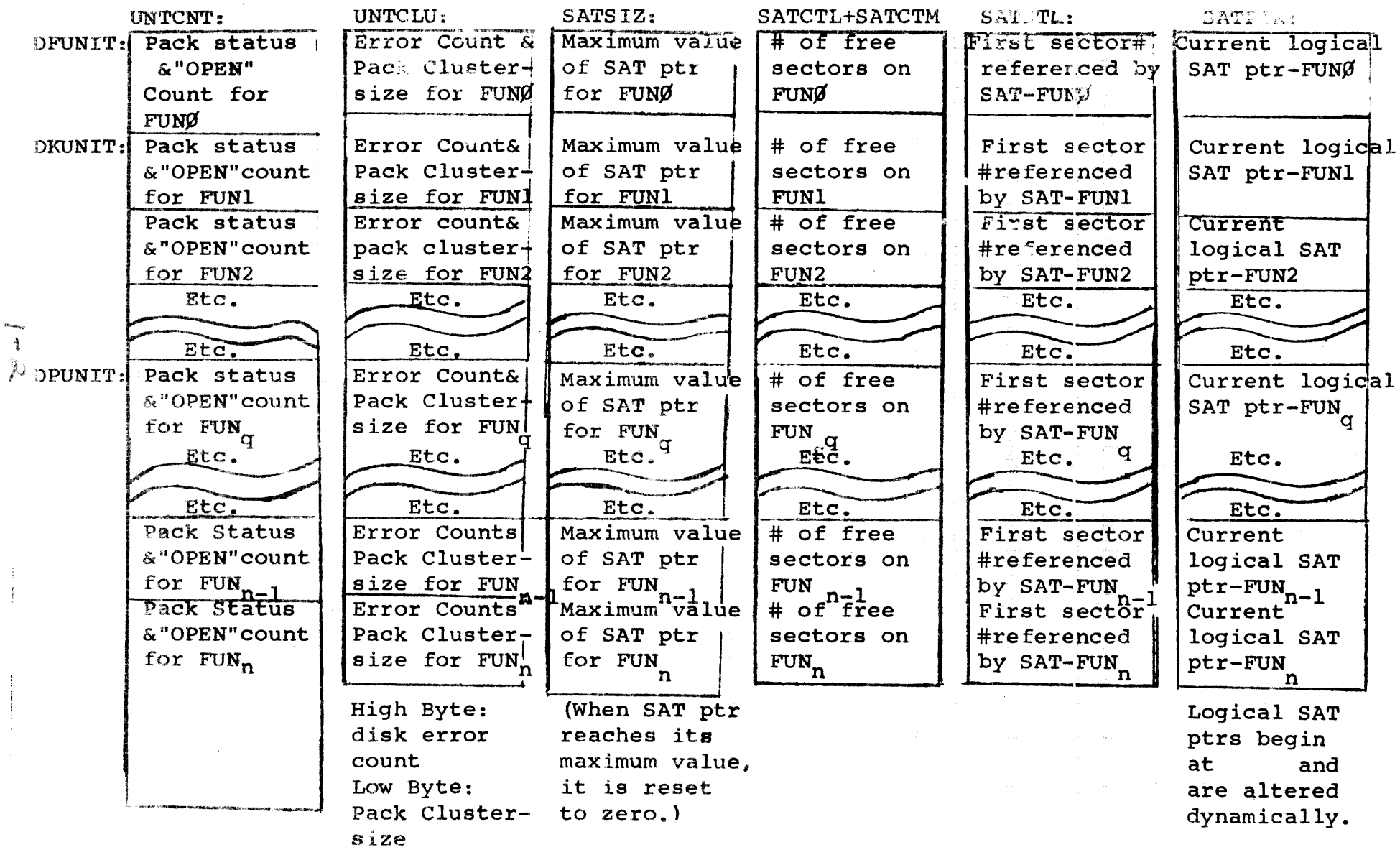
DSTATB	Disk statistics buffer
JSTATB	Job statistics buffer
QSTATB	Queue statistics buffer

At each clock tick, **the system checks the switch register** for the proper action:

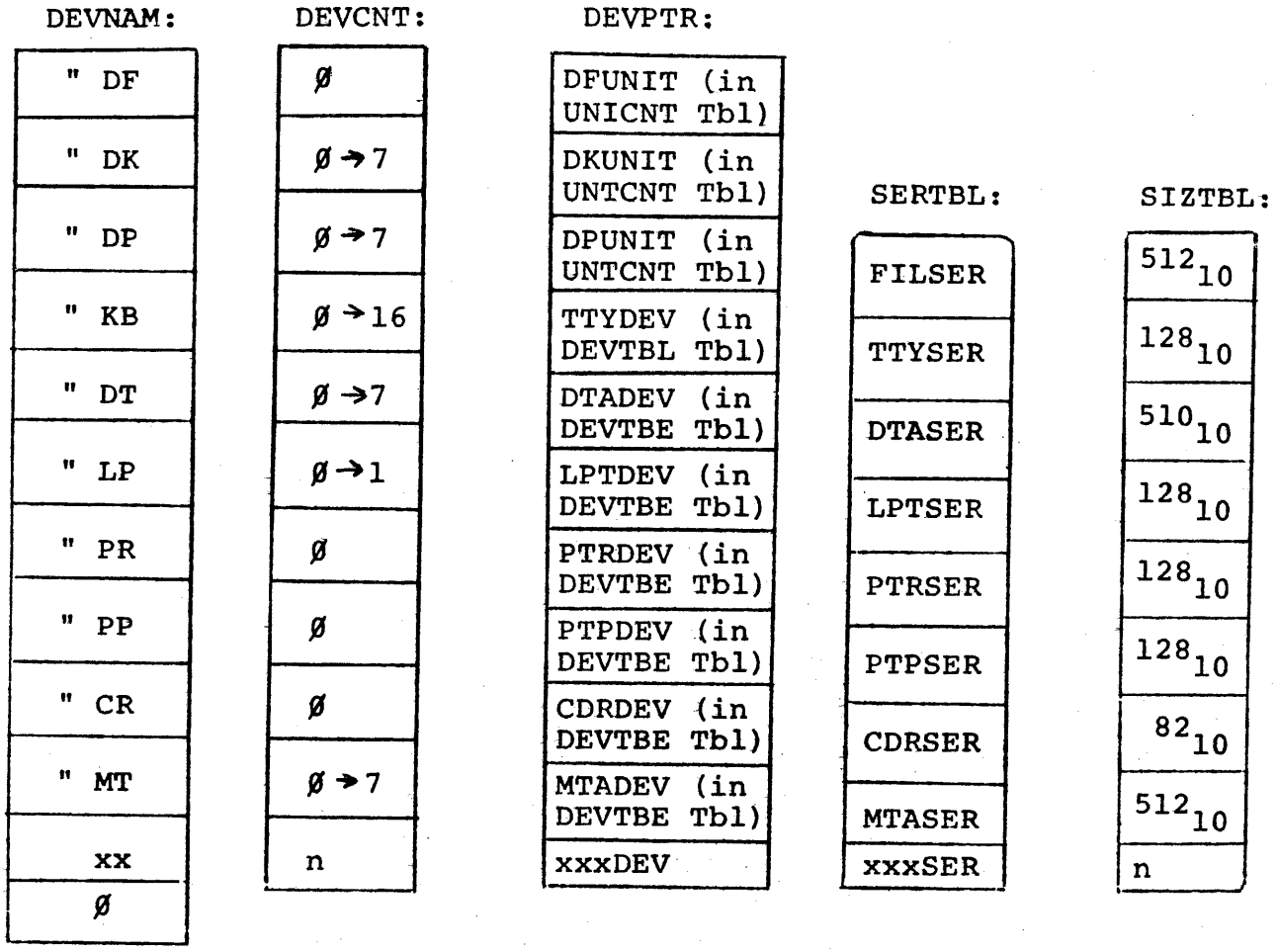
Contents	
177777	Collect statistics
1xxxxx	Freeze statistics
0xxxxx	Clear statistics

The job statistics area contains the following entries:

Entry size	Usage
1 word	Number of ticks per second
2	Uptime
1	Smallest number of free small buffers
2	Level 3 accounted
2	Lost time
2	Level 3 unaccounted
2	Null job time
2	Fip needed
2	In system mode
2	Fip waiting
2	for code
2	for disk information
2	for disk storage allocation
2	for something else
12	One two-word entry for each processor prior
2	Number of lockouts
1	Number of KW11P clock overruns



The proper entry in these tables is indexed by 2*FUN.

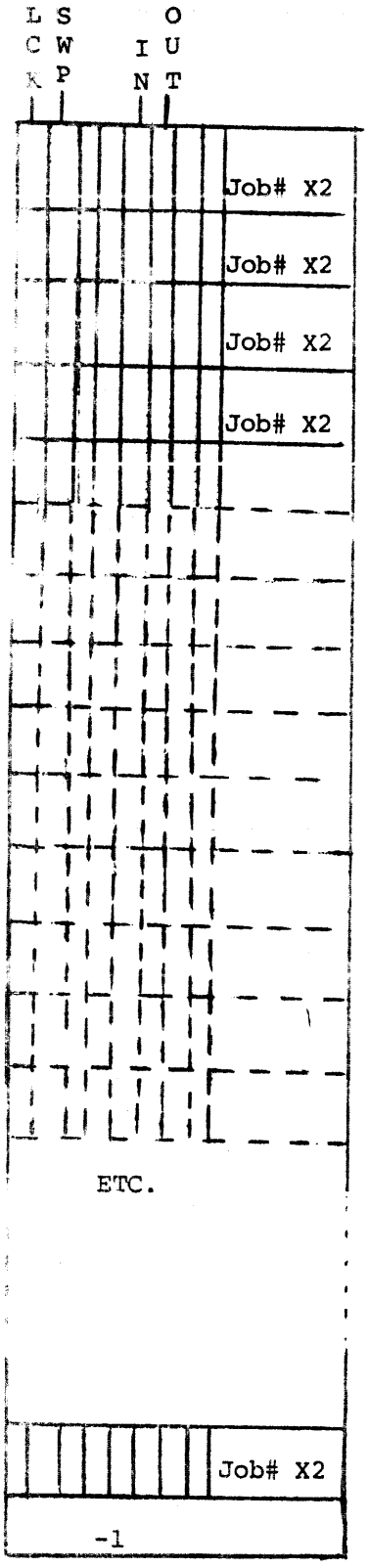


2 byte ASCII names of all devices

Maximum unit # for each device (if entry is -1, then that device does not exist.)

Pointers to slots in UNTCNT or DEVTBL appropriate to each device

addresses of service routines used by USERIO for read/write requests



CORTBL:

for first 1K words of core

for 2nd 1K words of core

for 3rd 1K words of core

for 4th 1K words of core

for nth 1K words of core

end of CORTBL indicator

LCK = This entry to be locked in core
SWP = This entry to be swapped as soon as possible
IN = Swap direction = IN
OUT = Swap direction = OUT

LEVEL 3 QUEUE TABLE

MODULE	entries are defined with the macro "L3QENT"	L3QTBL:
		+0
FIP	FIPRET-file processor return	+2
MON	SWPRET-swap completions	+4
DSK	FILRET-I/O completed	+6
DTA	DTSYM-reads or writes a block of DECtape for FIP	+10
DTA	BUFQ-big buffer pool starter	+12
DSK	BUFSMQ-small buffer pool starter	+14
DTA	DTQCON-dispatches to appropriate driver completion address, unqueues next driver request	+16
DTA	DTWCON-return after block allocate	+20
CDR	CDRL3Q-card reader service	+22
MTA	MTACON-process a request	+24
MTA	MTADNE-process finished request	+26
MON	BRINGQ-bring a job into core	+30
MON	SCHED-scheduler	+32
MON	FORCEQ-force a job to dump himself	+34
MON	TIMERS-once a second timer service	+36

EMTS (added to 104 000)

Identifier	Value		EMTTBL:
CALFIP	0	→ 0	FIPEMT FIP entry point for user
.READ	2	→ 2	USERIO: I/O interface for direct
.WRITE	4	→ 4	USERIO access READ/WRITE
.CORE	6	→ 6	CORE. Expand/shrink the job
.SLEEP	10	→ 10	SLEEP. Sleep job for n seconds
.PEEK	12	→ 12	PEEK. Peek at an address for user
.SPEC	14	→ 14	SPEC. Special functions
.TTAPE	16	→ 16	TTAPE. Enable tape mode
.TTECH	20	→ 20	TTECH. Enable echo/disable tape mode
.TTNCH	22	→ 22	TTNCH. Disable echo
.TTDDT	24	→ 24	TTDDT. Enable DDT submode
.TTRST	26	→ 26	TTRST. Restore programmed output
.TIME	30	→ 30	TIME. Give job timing info.
.POSTN	32	→ 32	POSTN. Get current position on line
.DATE	34	→ 34	DATE. Get date infor. for job
.PRIOR	36	→ 36	PRIOR. Set/clear special run priority
.STAT	40	→ 40	STAT. Get jobs current statistics
.RUN	42	→ 42	RUN. Run job (via channel #15)
.NAME	44	→ 44	NAME. Put names of job in NAMTBL
.EXIT	46	→ 46	EXIT. Return to default RTS
.RTS	50	→ 50	RTS. Change RTS to named RTS
.ERLOG	52	→ 52	ERLOG. Log the error called
.LOGA	54	→ 54	LOGS. Check for logical errors

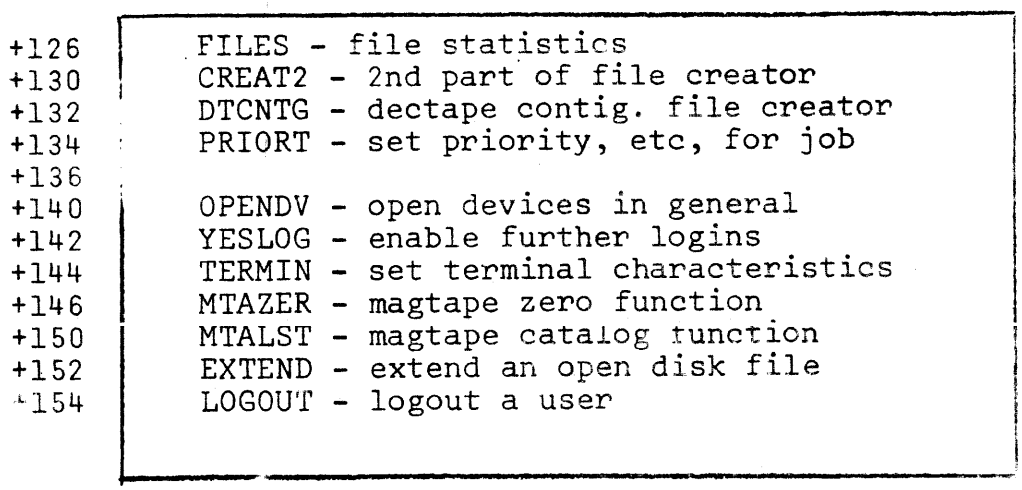
EMTs and EMTTBL dispatch table

FIPTBL - ADDR TO ROUTINES

Possible values for FQFUN (high order bute of word 2 in FIRQB)

+0	CLOSE - close an open channel	OPN
+2	OPEN - open a channel	OPN
+4	CREATE - create/extend/open a channel	ORN
+6	DELNAM - delete a file by name	DLN
+10	RENAME - rename a file	OPN
+12	DIRECT - directory information	DIR
+14	UUOF - process UUO	UUO
+16	ERRFIL - get error message text	OPN
+20	RESET - reset (close) all channels except Ø	OPN
+22	LOOKUP - file lookup	OPN
+24	ASSIGN - assign a device	ZER
+26	DEASØØ - deassign a device	ZER
+30	DEALL - deassign all devices	OPN
+32	CRTMP - create a .TMP file	OPN
+34	CRE8BØ - create an image file	OPN
*		
+36	WINDOW - window turner for disk files	WIN
+40	EXTCRE - extend a closed disk file	EXT
+42	OPENCR - open a newly created/extended file	OPN
+44	LOGIN - login a user	LIN
+46	PASSWD - create user account	ZER
+50	DLUF - delete user account	DLN
+52	RADF - read or read/reset accounting data	DIR
+54	CHUF - change password/kill job	DLN
+56	CLEAN - clean up a disk pack	DLN
+60	ZUSF - zero things	DLN
+62	ATTF - attach to a detached job	LIN
+64	DETF - detach from a terminal	DIR
+66	MOUNT - mount/dismount/lock/unlock disks	OVRUTL
+70	MTAOPN - magtape opener	MTU
+72	MTACLS - magtape closer	MTU
+74	MTAUTL - magtape utility	MTU
+76	DTALOC - dectape allocator	DTU
+100	DTSERR - dectape error processor	DTU
+102	DTLIST - dectape directory information	DTU
+104	DTKILL - dectape file deleter	DTU
+106	RUNJOB - run some program for a user	OPN
+110	DTNAME - dectape renamer	DTU
+112	DTRF - dectape directory reader	DTU
+114	DETCLOS - dectape closer	DTU
+116	DTOPEN - dectape opener	DTU
+120	DTZERO - dectape zeroer	DTU
+122	BACKUP - change file date/time for backup	UUO
+124	HANGUP - hangup a dataset	UUO

* Functions above this line are the only functions the run-time system may issue.



+126	FILES - file statistics	UUO
+130	CREAT2 - 2nd part of file creator	UUO
+132	DTCNTG - dectape contig. file creator	DTN
+134	PRIORT - set priority, etc, for job	ZER
+136		UUO
+140	OPENDV - open devices in general	OPN
+142	YESLOG - enable further logins	LIN
+144	TERMIN - set terminal characteristics	TRM
+146	MTAZER - magtape zero function	MTU
+150	MTALST - magtape catalog function	MTU
+152	EXTEND - extend an open disk file	EXT
+154	LOGOUT - logout a user	OPN
		ZER

Module names containing various functions are included on the right.

BLOCKS

JOB DATA BLOCK

JDIOB - pointer to IOB		Ø
JDFLG - job status flags		2
JD Post - post to RTS	JDIOST -IO status	4
JDWORK - pointer to job work block		6
JDRTS - pointer to RTS block		1Ø
JDRESB - L3QUE bits to set on residency		12
JDUFDR - RIB entry of UFD for user		14
JDKCTM - MSB of KCT used	JDFLG2 - monitor jobflg	16
JDSIZØ - Size now	JDSIZ1 - Size next	2Ø
JDCPU - CPU in lengths of sec		22
JDCON - Connect time in min		24
JDKCT - KCT used		26
JDDEV - Device time in min		3Ø
JDPPN - Jobs PPN		32
JDCORM - jobs core max	JDPRI - jobs priority	34
JDBRST - jobs burst	JDSWAP - jobs swap param	36

BITS In JDSWAP

Ø-3	Bit in Bit Map Word
5-4	Selects 1 of 4 map words
7-6	Selects Map Word

RUN-TIME SYSTEM DESCRIPTION BLOCK

R. LINK	Link to More Descriptions	Ø
R. NAME	2 wd Rad 50 name of pure code	2
		4
R.CNT	Count of active users	6
R.USIZ	Max size in K for user	1Ø
R.MSIZ	Min size in K for user	12
R.DISK	CILUS Disk Block #	14
R.LOAD	Low order real memory address	16
R.BACK	Pointer to NB of RTS	20
R.BYTE	Byte Count of Image	21
R.XMEM	Hiorder real memory address	22
R.ISIZ	Initial size ink for user	24
R.RDSK	Double word RSTS Disk	26
	Address	30
R.CPTR	PTR to last core table entry +2	32
R.1KCT	Size of RTS in K	34
R.REDO	Full 4K description pattern	36

FIRQB

FQQQ	Queue in link set by FIP	Ø
FQJOB	Issuing Job #X2	2
FQFUN	Function requested	3
FQFIL or FQERNO	Channel Slot in IOB of issuing job Error message code and text start	4
FQPPN or FQCOM3	Project/programmer no. Internal data	6
FQNAM1	2 word file name in RADIX50	1Ø
FQEXT	Extension in RADIX50	14
FQSIZ or FQCOM2	No sectors in file or sectors to extend Internal return addr (DECtape)	16
FQNAM2 or FQSWIT or FQBUFL	3 word new file name EXT in RADIX50 Open for output switch Default buffer length	20
FQMODE	Mode Indicator	22
FQFLAG	File's flag	24
FQCOM1	Internal Return (DECtape)	26
FQPROT	New protection code (-1 if passed)	27
FQDEV	2 byte ASCII device name	30
FQDEVN	1 byte unit number	32
FQCLUS or FQCOM	File cluster size for file creation DDB link addr (DECtape)	34
FQFUN1	Chaining function FQNENT no of directory entries	36
FQFUN2	Internal extra function	37

File request queue block (FIRQB) (pronounced "Furk-be")

All requests for file processing are made by setting the necessary parameters in the FIRQB, and calling the Monitor with "CALFIP".

Only relevant parameters need be supplied. When FIP exits, the returned values come back in the FIRQB along with an error code.

FIRQB entries include the following:

Multiple entries are for different FIP functions

File Request Queue Block FIRQB **- Used by FIP**

Symbol	Offset$_8$		Usage
FQQQ	0		Queue word, set by file processor
FQJOB	2	byte	2*job number of request issuer
FQFUN	3	byte	internal function number
FQFIL	4		2*channel number
FQPPN	6		Project/programmer number
FQNAM1	10		two-word file name in Radix-50
FQEXT	14		Extension in Radix-50
FQSIZ	16		Segments in file or segments to extend
FQSWIT	20		Open for output switch
FQMODE	22		Mode indicator
FQFLAG	24		File's flag
FQCOM1	26		Internal return dispatch
FQDEV	30		2-byte Ascii device name
FQDEVN	32	byte	1-byte unit number (high-byte = -1 if no-1)
FQCLUS	34		File cluster size
FQFUN1	36	byte	Chaining function
FQFUN2	37	byte	Internal extra function

Alternate entries (for special situations)

Symbol	Offset$_8$		Usage
FQERNO	4		Error message code and text beginning
FQCOM3	6		Internal data
FQCOM2	16		Internal return address
FQNAM2	20		3-word new file name.ext in Radix-50
FQBUFL	20		Default buffer length
FQCOM	34		DDB link address
FQNENT	36		Number of directory entries
FQPROT	37	byte	New protection code (low-byte = -1 if not)

XRB TRANSFER CONTROL BLOCK

XRLEN	length of IO buffer in bytes	0
XRBC	byte count of transfer	2
XRLOC	pointer to buffer start for Xfer	4
XRCI	channel number for Xfer	6
XRBLK	starting device block no for Xfer	10
XRTIME	wait time for TTY input	12
UNUSED		14
XRBSIZ	reserve 7 words for XRB	16
		20
		22
		24
		26
		30
		32
		34
		36

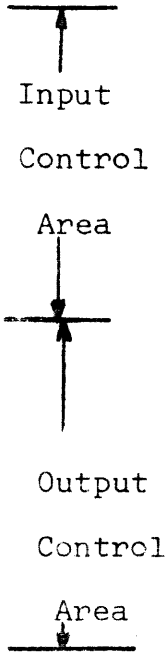
Used by User to initiate an I/O request and for Monitor/user data requests.

Values in 'XRLEN' and 'XRLOC' and 'XRCI' must be even!

Value in 'XRLEN' must be > = the default value

Value in 'XRBC' must be < = value in 'XRLEN'

LINE PRINTER DDB



Status and access control	device handler index	+0
device unit number	owner job index	+2
Clock time at which device was assigned		+4
Vertical position	Horizontal position	+6
		+10
		+12
		+14
		+18
		+20
Output fill buffer pointer		+22
Output fill count		+26
Output empty buffer pointer		+28
Output empty count		+30
Output buffer count		+32
# lines per page	# characters per line	+34
		+36

Offsets into the Input/Output control groups (relative to DDINP and DDOUT respectively).

Symbol	$Offset_8$	Usage
FP	0	Fill pointer
FC	2	Fill count (-37 to -1)
EP	4	Empty pointer
EC	6	Empty count (-37 to -1)
BC	10	Buffer count (as a byte)

Bit assignments for the DDB status words are as follows:

DDSTS

Symbol	Value	Usage
DDSTAT	100000	Internal status, cleared by "close"
DDWLO	2000	Write-lock for device if set
DDRLO	1000	Read-lock for device if set
DDNFS	400	Device is not file-structured

DDCNT

Symbol	Value	Usage
DDASN	100000	Device assigned through command
DDUTL	40000	Device assigned for utility sequence
DDCONS	20000	Device is the console device

vice Data Block DDB

Each time a device is initialized, the init count in DDCPT is incremented; when the device is closed, the count is decremented. If the count goes to zero, and the device was not assigned, then it is returned to the system; otherwise it is retained by the job.

The non-device-dependent entries in the DDB are:

Symbol	Offset$_8$	Usage
DDIDX	0 byte	Handler index (index into DEVNAM etc.)
DDSTS	1 byte	Status and access control byte
DDJBNO	2 byte	Owner job index (zero if free) Job# X 2
DDUNT	3 byte	Device/Disk unit number
DDTIME	4	Time assigned or initialized
DDHORZ	6 byte	Horizontal position
DDVERT	7 byte	Vertical position
DDINP	10	Input control area
DDOUT	22	Output control area
DDHORC	34 byte	Characters per line + 1
DDVERC	35 byte	Lines per page
DDCNT	36	Init count for device, high byte has assignment status.

Unique offsets within DDB

Symbol	Offset		Usage
TTSTS1	DDSTS	1	Status 1
TTLINE	DDUNT	3	Keyboard number times 2
TTSTS2	DDVERT	7	Status 2
TTSTS3	DDVERC	35	Status 3

TTY Status 1 values (Word values)

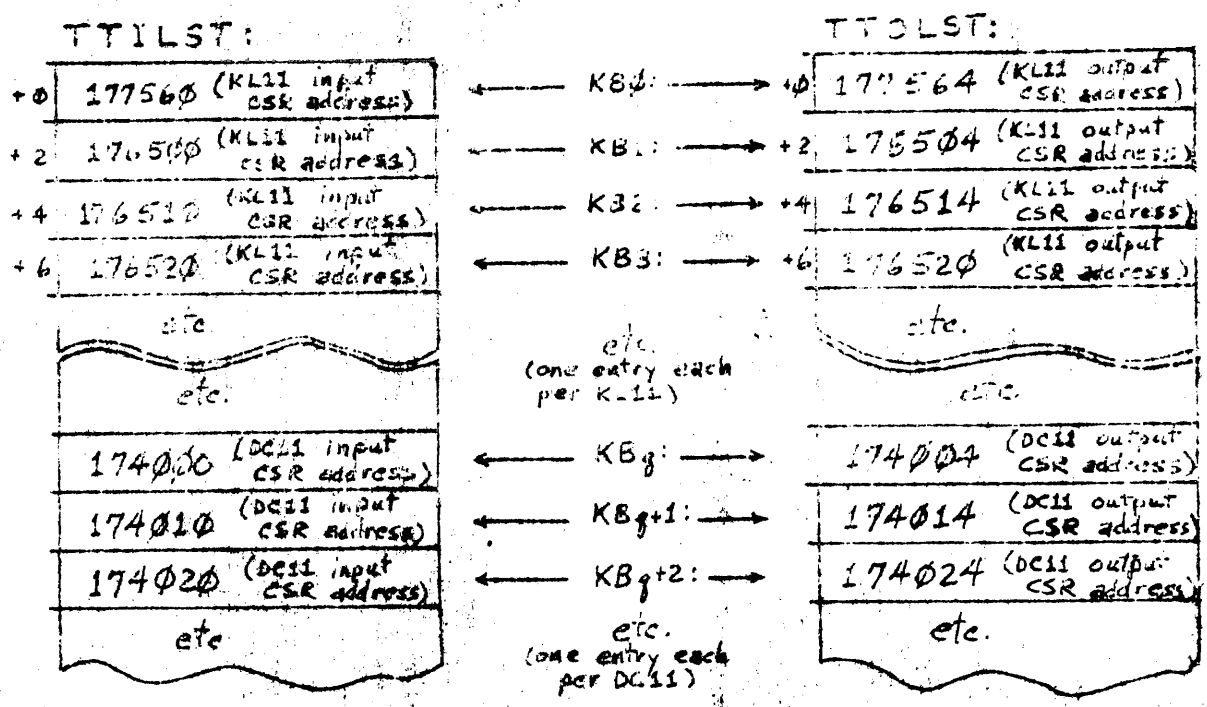
Symbol	Value	Usage
NOOUTP	100000	Junk programmed output
HAFDUP	40000	Local echo tty
NOECHO	20000	Don't echo on tty
TAPE	10000	Tape mode
TT2741	4000	Terminal is an IBM 2741
TTDDT	400	Into DDT-submode

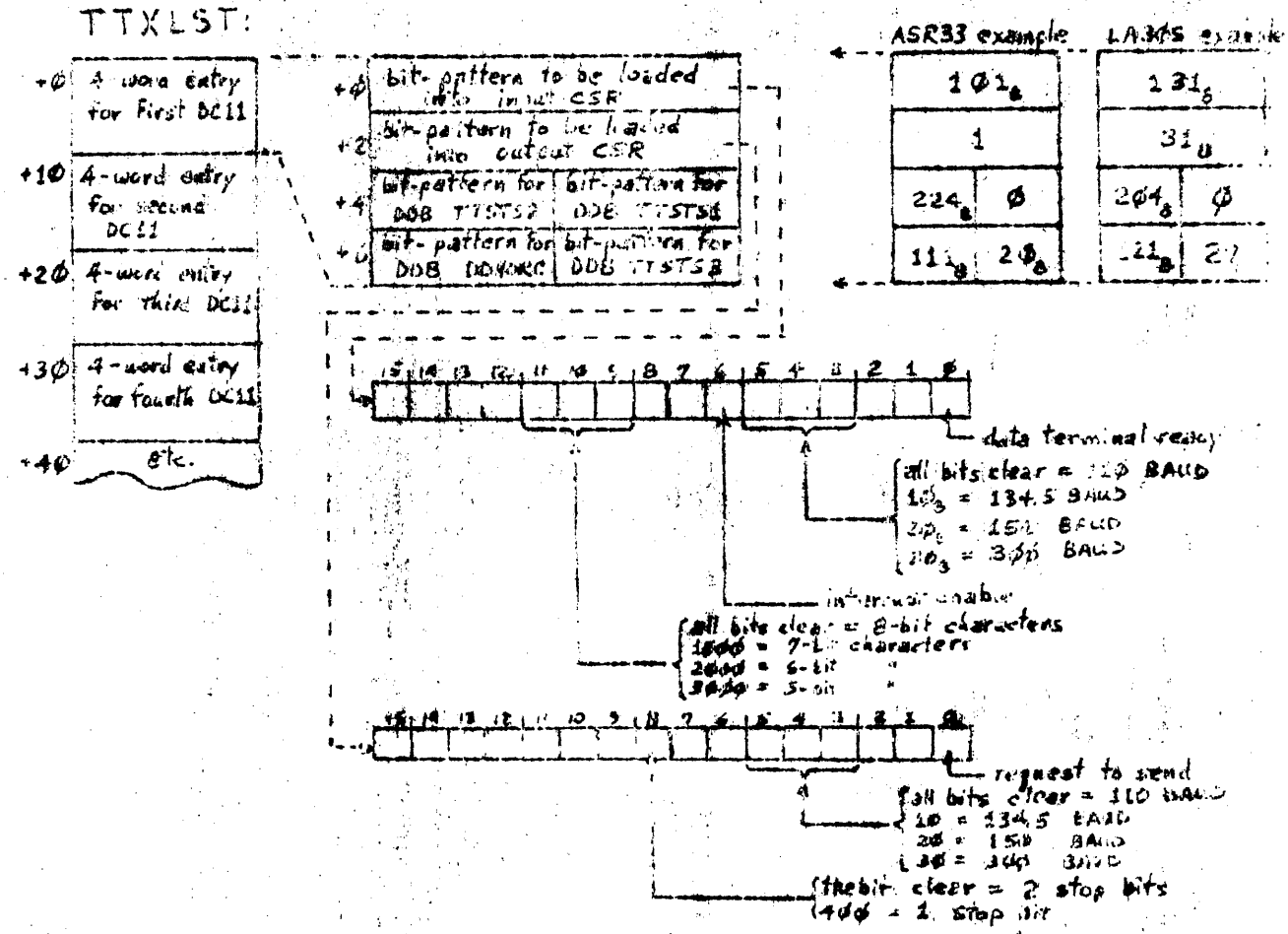
TTY Status 2 values (Byte values)

Symbol	Value	Usage
TT33	200	Model 33 tty (no hardware tab)
TT35	100	Model 35 tty (hardware VT and FF)
TTESC	40	Terminal has Ascii-68 escape (ESC = 33_8)
TTXON	20	Terminal has XON/XOFF feature
TTRUB	10	Special rubouts
TTBIN	4	Binary input mode
TTTECO	2	Teco special mode
TTSTAL	1	Terminal may send XON/XOFF to RSTS

TTY Status 3 values (Byte values)

Symbol	Value	Usage
TTLA30	200	Terminal needs computed fill
TTPRTY	100	Terminal wants parity added
TTPODD	40	Terminal wants odd parity
TTUC	20	Translate to upper-case on input
TTUPAR	10	Don't print control characters with uparrows
TTFILL	7	Fill amount (0 to 7)


TTILST:
+0 177560 (KL11 input CSR address)
+2 177500 (KL11 input CSR address)
+4 176510 (KL11 input CSR address)
+6 176520 (KL11 input CSR address)
etc.
etc.
174000 (DC11 input CSR address)
174010 (DC11 input CSR address)
174020 (DC11 input CSR address)
etc
KB0:
KB1:
KB2:
KB3:
etc. (one entry each per KL11)
KBq:
KBq+1:
KBq+2:
etc. (one entry each per DC11)
TTOLST:
+0 177564 (KL11 output CSR address)
+2 175504 (KL11 output CSR address)
+4 176514 (KL11 output CSR address)
+6 176520 (KL11 output CSR address)
etc.
etc.
174004 (DC11 output CSR address)
174014 (DC11 output CSR address)
174024 (DC11 output CSR address)
etc.


TTXLST:
+0 4 word entry for First DC11
+10 4-word entry for second DC11
+20 4-word entry for third DC11
+30 4-word entry for fourth DC11
+40 etc.
+0 bit-pattern to be loaded into input CSR
+2 bit-pattern to be loaded into output CSR
+4 bit-pattern for DDB TTSTS2 / bit-pattern for DDB TTSTS2
+6 bit-pattern for DDB [illegible] / bit-pattern for DDB TTSTS2
ASR33 example
LA36S example
data terminal ready
all bits clear = 110 BAUD
interrupt enable
all bits clear = 8-bit characters
request to send
all bits clear = 110 BAUD
the bit clear = 2 stop bits
400 = 1 stop bit

TTY DDB

status DDSTS device handler index DDIX	+0
device unit num DDUNT DDJBNO owner job index	+2
Clock time at which device was assigned or inited. DDTIME	+4
DDVERT DDHORZ	+6
Input fill buffer pointer DDINP	+10
Input fill count	+12
Input empty buffer pointer	+14
Input empty count	+16
Input buffer count	+20
Output fill buffer pointer DDOUT	+22
Output fill count	+24
Output empty buffer pointer	+26
Output empty count	+30
Output buffer count	+32
Status DDVERC horizontal max DDHORC (# characters/line)+1	+34
Assignment Status init count for device DDCNT	+36

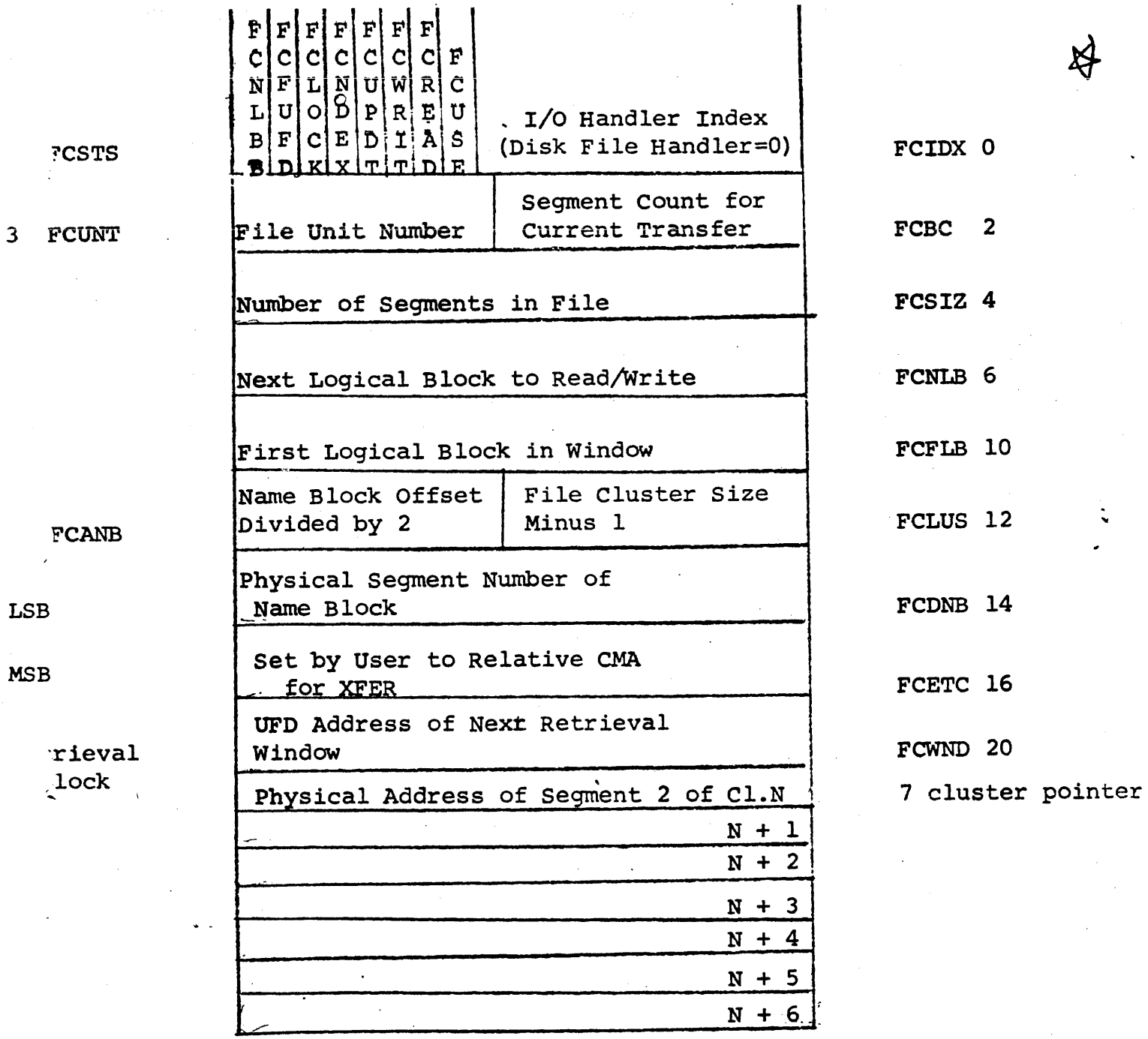
DEVICE DATA BLOCK (DDB)

These 16 word blocks are permanently assigned to each device and teletype in the system. The device table (DEVTBL) contains pointers to each DDB.

If a device is assigned to a job, then the entry in "DDJBNO" contains the number of the owner job. If 0, the device is available.

Each time a device is init'ed, the init count in "DDCNT" is incremented. When the device is closed, the count is decremented. If the count goes to 0, and the device was not "assigned" then it is returned to the system; otherwise it is retained by the job.

The non-device-dependent entries in the DDB are labeled.



FCB and RETRIEVAL BLOCK DEFINITIONS

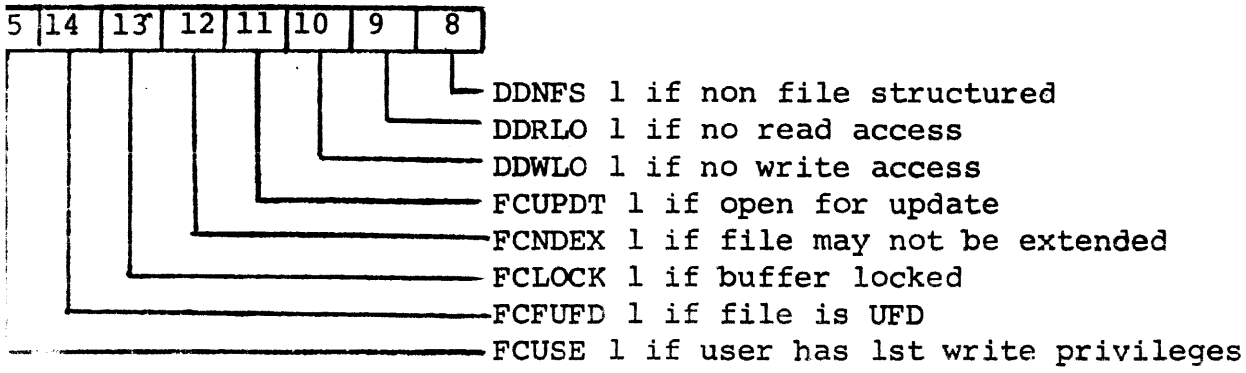
FCSTS DEFINITIONS

FCNLBB =	100000	If 1, the 'FCNLB' is backed up
FCFUFD =	40000	If 1, the open file is a UFD
FCLOCK =	20000	If 1, the current buffer is locked
FCNOEX =	10000	If 1, the file may not be extended
FCUPDT =	4000	If 1, the file is open for update
FCWRIT =	2000	If 1, user may not write file
FCREAD =	1000	If 1, user may not read file
FCUSE =	400	If 1, this user got 1st write priviledge

File Control Block

DDSTS	Status	DDIDX	Handler Index	Ø
FCUNT	File Unit #	FCBC	# Blks in Buffer	2
FCSIZ	# Segments in File			4
FCNLB	Next Logical Blk to Read/Write			6
FCFLB	First Logical Blk in Window			10
FCANB	Name Blk Offset/2	FCLUS	File Cluster Size	12
FCDNBL	Phys Seg # of name blk			14
FCDNBM	Extended Seg #			16
FCWND	UFD Address of next RB			20
				22
				24
				26
Window				30
				32
				34
				36

DDSTS



Disk Request Queue Entry Block

DSQ

DSQ Queue Link		ø
DSQERR Retrycnt & Error Flag	DSQJOB Job**2	2
DSQL3Q L3QUE Bit to Set at completion		4
DSQXDA Ext. Disk Address		6
DSQDMA Disk Segment #		10
DSQXMA Ext Core address of transfer		12
DSQCMA Core address of transfer		14
DSQUNT Unit #	DSQCNT Seg CNT of Xfer	16
DSQFAR Que Fairness or Priority	DSQFUN Op. Function Code	20
DSQMSG FCB Ptr		22
DSQTOT Total Transfer Counter		24
DSQPDA Phys. Disk Address		26
DSQOPT Disk Optimization Word		30
DSQ SAV Saved Function	DSQDUN Unit # * 2	32
DSQPUN Unit #	DSQTFN Temp Disk Function	34
DSQTMP Temp Storage or Drivers		36

This block of 16 words is passed to and from the disk drivers to initiate disk data transfers and to signal that a transfer has been completed. If the transfer was not done the block is passed back with a zero count in DSQERR.

DSQFUN
read - 105
write - 103

IMPORTANT INFORMATION IN MONITOR

Virtual 000000 to 000776 is for vectors

DATE	Current date	1000
TIME	Current time	1002
TIMSEC	SECS to next min	1004
TIMCLK	Interrupts to next sec	1005
JOB	Current job *2	1006
NEXT	Next job to run *2	1007
JOBDA	PTR to current job	1010
JOBF	PTR to current flags (JDFLG)	1012
IOSTS	PTR to current IO status (JDIOST)	1014
JOBWRK	PTR to job work block	1016
JOBTIM	Run time in tics this run	1020
JOBQNT	Used residency quantum in ticks	1022

(lative)	IDENT	CONTENTS
0	.HOLD	Start of Holding Chain
2		End of Holding Chain
4	ODTARS	ODT Saved Addr Register
6	ODTDRS	ODT Saved Description Reg
10	WAIT2T	2 Tick Delay L3Q Bits
12	WAITNT	Next Tick L3Q Bits
14	L3QUE	Level 3 Queue Bit Pattern
16	FREES	Pointer to 1st Free Small Buffer
20		Count of Free Small Buffers
22	FREEB	Pointer to 1st Free Big Buffer
24		Count of Free Big Buffers
26	SMLFLG	Non-Zero if out of Small Buffers
30	QUANT	Running Job's Quantum
32	BIGFLG	Non-zero if out of Big Buffers
34	MAPHI	Extended Address Bits 4 + 5
36	MAPLOW	Low Order Address
40	SWAPF/NXTRES	
42	CORPTR	Pointer for Next Spot to Sub-schedule
44	JOBPTR	Round-Robin Job Numbers (3 bytes)
47	CORFOR	Round-Robin Job Number to Force Out
50	FPPFLG/FPPFEC	FPPInterrant Flag/FPP Exception Code
52		FPP FEA goes here on error
54	RTSLNK	Link to RTS Description Blocks
56	RTSLOD	Non-zero if Ptr to RTS Desc. Block to Load
60	RTSLOW	Cortbl Ptr to start of load (or Ø)
62	RTSHGH	Cortbl Ptr to End + 2 of load (or Ø)
64	RTSJOB	Job #X2 to signal on land done
66		

Monitor Core Map //.1

(Part of MONCTL area)

SWAP CONTROL PARAMETERS

(Relative) to Start of MONCTL Area	IDENT	CONTENTS
66	SWPMAX	
70	SWDONQ	
72	SWPDSQ	Swapping Parameter (DSQ) Block
		Job Slot in DSQ
		Start Swap Completion Routine
		Fill - up to 16 wds
132	SWBASE	Swap Ø Sys [Ø,1] File Base
136		SWAP 1 Sys [Ø,1] File Base
142		SWAP 2 Sys [Ø, 1] File Base
146		SWAP 3, Sys [Ø, 1] File Base
152	SWPCNT	# of Active Bytes in "SWPMAP"
154	SWPMAP	SWAPØ Bit Map
164		SWAP 1 Bit Map
174		SWAP 2 Bit Map
204		SWAP 3 Bit Map

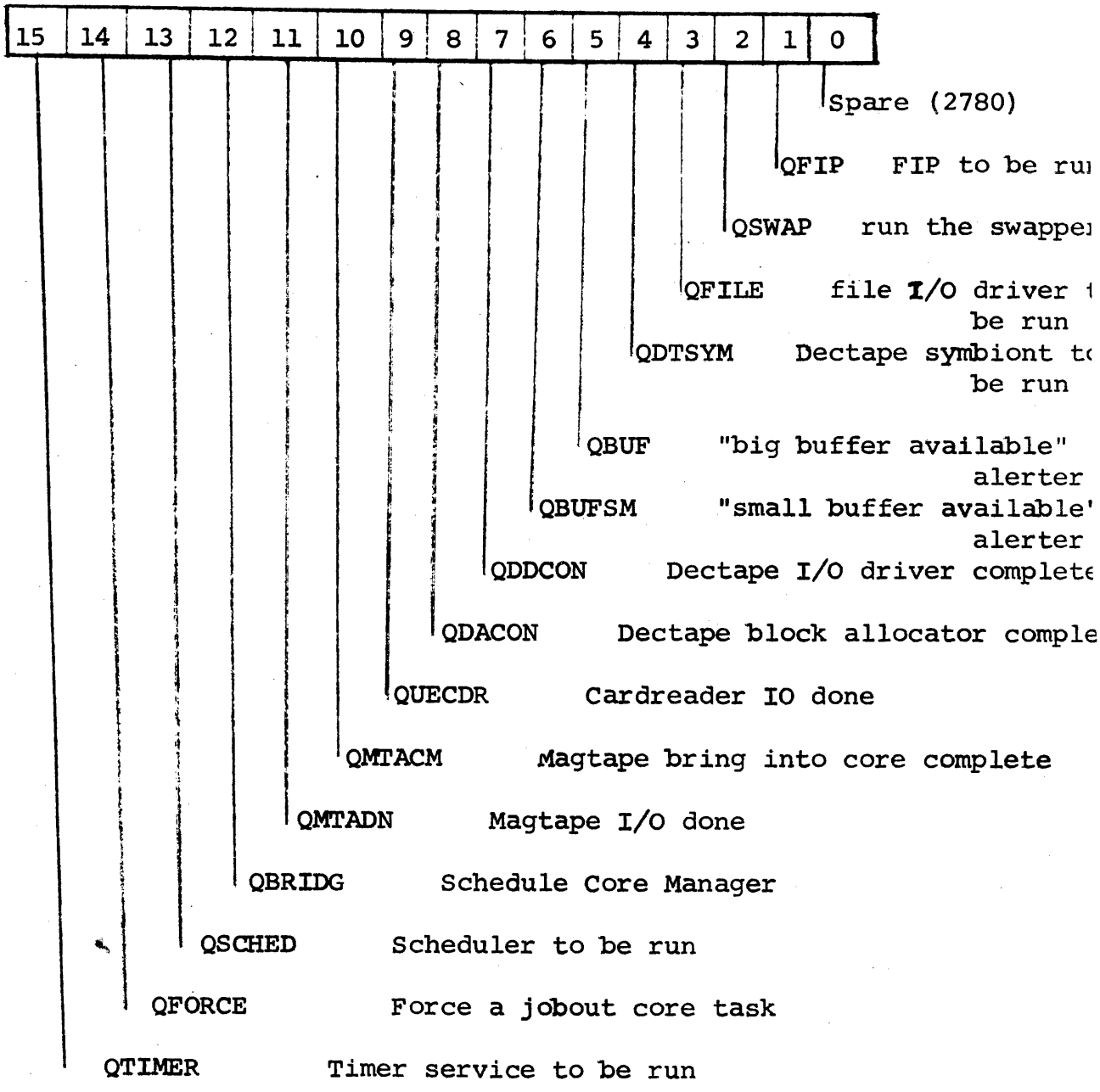
Monitor Core Map //.2
(Part of MONCTL area)

	DSKPUN	
	CSKQUE	
	TBL SRT	
	DEVNAM:	two byte name table for device types (RF, RK...XX, YY, ZZ)
	DEVNKB:	
	DKSIZL:	disk sizes in sectors (least significant)
	DKSIZM:	disk sizes (most significant)
	DEVCNT:	Device unit number maximums
	DEVPTR:	Device unit list (disks, Y's...ZZZ). Pointers to DDB pointer table.
	TTXLST:	terminal characteristics list (modems only DC11, DH11, etc.) Address of CSR's status bits.
	TTILST:	table of I/O CSR's for terminals (INPUT)
	TTOLST:	table of I/O CSR's for terminals (OUTPUT)
	XXXCRS	
	XXCRAS	
	XCRSEX	
	XCRSEG	
	XCRSIZ	
	EVERYS	

12.1 Elements of TBL Module

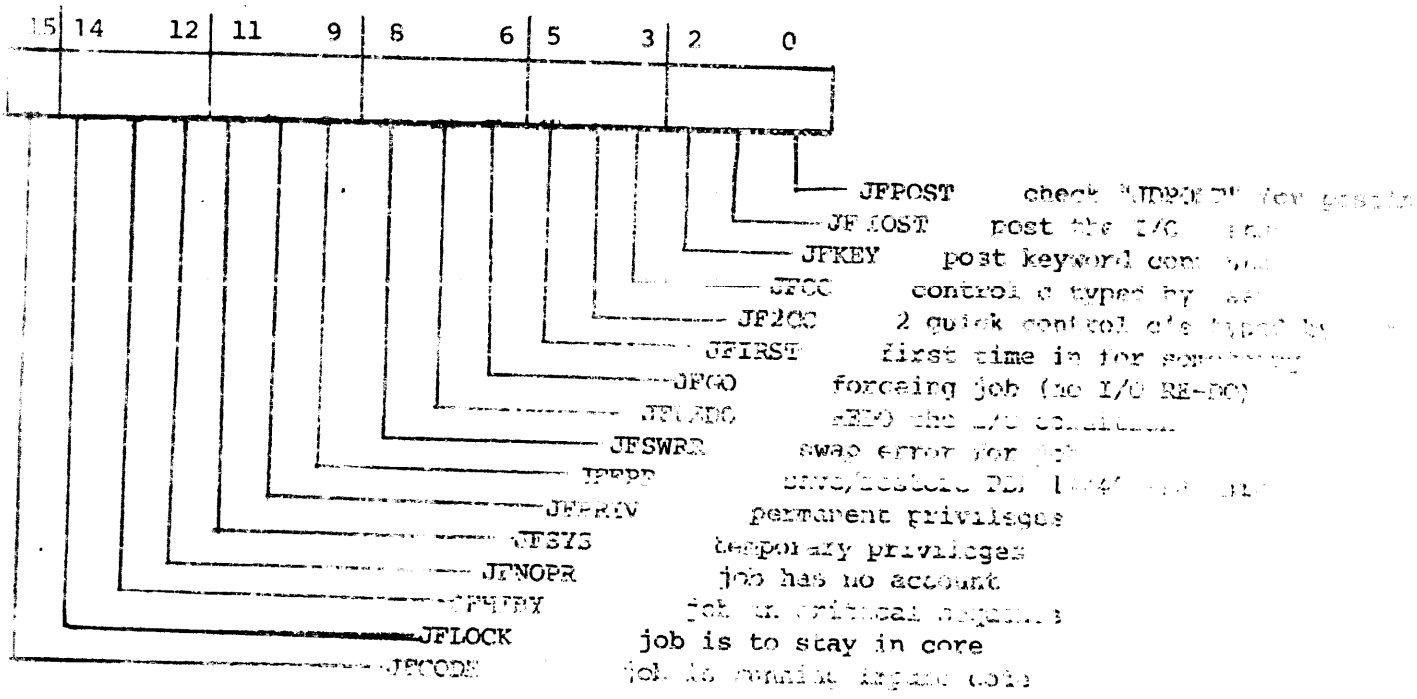
WORDS

LEVEL 3 QUEUE

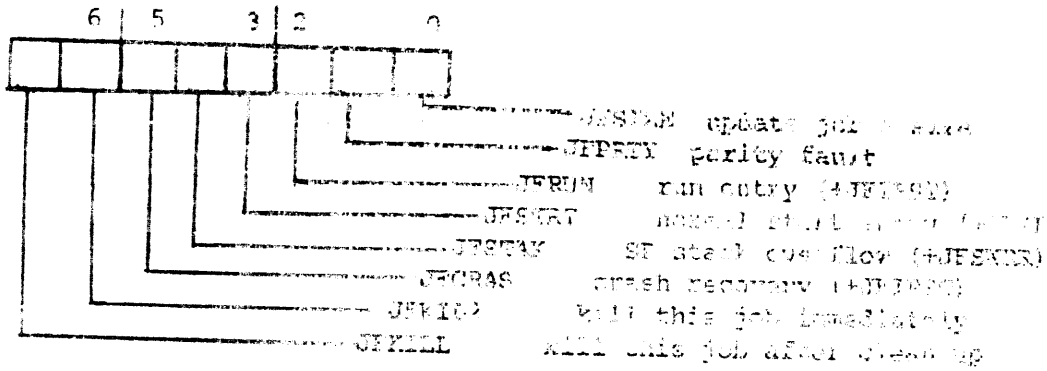


JOB FLAG ASSIGNMENTS

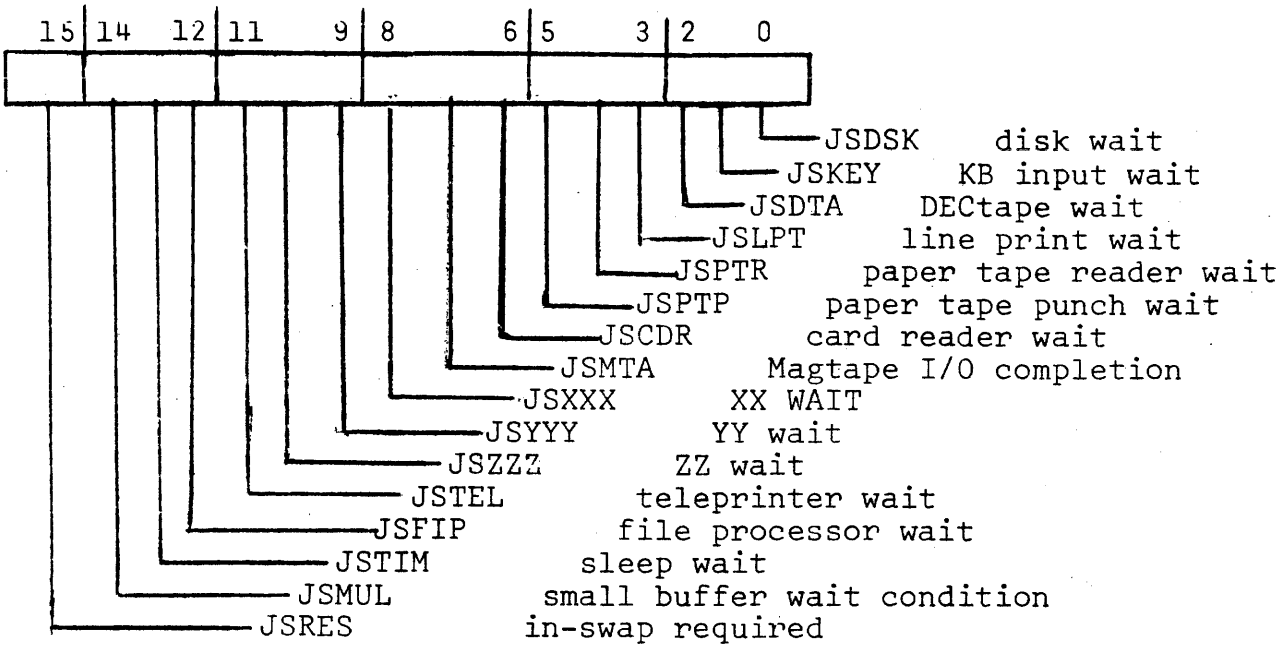
JDFLG



JDFLG2



JBSTAT AND JBWAIT



JSALL=64172

DIRECTORY STRUCTURE

Disk File Optimization

1. Optimize the Pack Cluster Size for each disk.

   The pack cluster size is optimum when the entire SAT (Storage Allocation Table, stored as (0,1) SATT.SYS) can fit into the 256-word buffer in core beginning at SATBUF. To effect this, each RK disk should have a minimum pack cluster size of 2, and each RP03-disk should have a minimum pack cluster size of 2 and preferably of 4, RF disk units with less than 5 platters can have a pack cluster size of 1, but RF units with more than 4 platters should have a pack cluster size of 2. RP04-disks have a minimum of 4 and should use 8?

2. Use private packs for production files rather than the public structure.

   Whenever a file is opened on the public structure, the directories on every public disk have to be searched, either to verify the file's existence or to ensure its non-existence. This overhead of searching more than one disk can be avoided by placing the file on a private pack.

3. Dedicate complete structures to large production files.

   If possible, dedicate an entire private pack to a single large production file. This will ensure

efficient directory organization and will minimize disk "seek" time for data processing/accessing. When it is necessary to put more than one file on the same pack, dedicate a whole account to each large production file. This will minimize directory-search overhead. When it is necessary to put more than one file under the same account on the same disk, then create the large files before the small ones. This ensures better organization of the directory structures.

4. When possible, keep distinct files accessed by the same program on distinct disks.

If a program accesses more than one file and if all of these files are on the same disk pack, then disk head movement will be required every time the program's current reference to a file is different from its preceding file reference; thus a large percentage of execution time will be spent in moving the disk head back and forth. On the other hand, if each file referenced by the program exits on a distinct disk, then no head movement is required when changing from one file reference to another in the program; the heads will simply move once when the location of the data itself requires it.

5. Pre-extend files to their maximum length.

Creating a file at execution time by sequential printing of data, by sequential writing of a virtual core array, or by sequential putting of records entails a high overhead of repeated calls to non-resident parts of the File Processor in order to extend the file each time another sector is required. This needless overhead can be avoided if the file is already in existence at its maximum length or if it is created at its maximum anticipated length before any significant data is written upon it. Thus at the beginning of any program which is to create a long file, the highest record number should be put or the highest element of the virtual array should be written (even with meaningless data) before any other writing is done. This applies also to scratch files. The FILSIZ option can also be used to pre-extend files.

6. Use Record I/O disk accesses wherever possible.

Formatted ASCII is the slowest RSTS I/O. It examines each character one by one for delimiters, line terminators, etc. and executes conversions between internal binary forms and external ASCII representations for all numerical data. Virtual core I/O is fast for a given individual array, but it is not record-oriented. When a logical record includes more than

one item, virtual core I/O will scatter the elements of the same logical record into different areas of the disk, grouping them by their array type and name rather than by any relevance to the same record. This may multiply disk accesses in a program. Record I/O as its name implies has been designed for processing records, in either sequential or random-access mode. It packs data efficiently and groups data by record organization. This minimizes the disk accesses needed for processing any record.

7. Optimize files' cluster sizes.

Since at any given moment the current retrieval information for a file is stored in core in the "window" of the file's File Control Block, retrieval overhead can be reduced to a minimum if the file's cluster size is such that the file does not need more than 7 total clusters.

8. Re-start from cleaned structures whenever possible.

When an account is first created and the first file is created under the new account, the linkages in the directory structures will be in the optimum order for fast and efficient directory handling. After a number of deletions, extensions, and new creations, however, this may no longer be true: directory

linkages may then point forward and backward in whatever fashion was necessary to update the directory. Consequently, to ensure that directory structures exist in a form optimized for fast access, it is best that disk pack periodically be re-structured. This can be done by using the DSINT program to prepare a fresh pack and then transferring to that pack all files which are to be accessed on it. Likewise an account should periodically be re-structured by first "zeroing" the account and then transferring to that account all files which are to be accessed under it. (Save files elsewhere before zeroing the account).

9. Optimize record size if possible.

If a file has a cluster size larger than 1 and if the size of the logical records to be processed exceeds 512 bytes, then an explicit RECORDSIZE of 1024 or greater will speed execution. This is true because RECORDSIZE defines the size of the user's buffer area and when the cluster size of the file allows it, the system will fill/empty the buffer in one disk access rather than in multiple accesses. If, however, the logical records to be processed are less than 512 bytes in length, then little is to be gained by departing from the default record size of 512 bytes.

10. Keep production accounts distinct from development accounts.

Because extensive file deletions, modifications, and creations can leave directory structures in an order ill suited for fast and efficient disk processing and because development work should always be done under accounts distinct from those under which production files are kept.

The DSKINT option will initialize a disk to the minimum RSTS file structure. A single or multi-platter RF disk, a single RKØ5 cartridge, or single RPØ3 disk pack may be initialized as a public, private, or system disk. In addition to writing the minimal file structure, DSKINT checks the whole disk for bad blocks by performing write and write-check operations over the total disk area. In checking for bad blocks, DSKINT will use from 1-8, test patterns as specified by the user during the DSKINT dialog,

The minimal RSTS file structure includes the master file directory (MFD) and the UFD for the system file account [Ø,1]. In the system file account, two files are created during the DSKINT process. The file BADB.SYS contains all bad blocks detected during the bad block checking phase of DSKINT. Bad blocks are permanently allocated to this file and may not be used during normal Time-Sharing operations. The second file is SATT.SYS which contains the storage allocation table (SAT) for the disk. The SAT contains one bit for each cluster on the disk. The size of SATT.SYS is therefore dependent on the size of the disk and the pack cluster size specified in the DSKINT dialog, the size of BADB.SYS depends on the number of bad blocks detected. Public and private disks contain only this minimal structure. The only difference between public and private disks is a "private" bit is set in the MED for private disks. In initializing a system disk, DSKINT will also write, in addition to

the minimal file structure, the MED entries required for the system library account [1,2], the library UFD and all library files are created under normal time-sharing operations.

When a new system is created by the system generation "Core Image Library" or "CIL" is written somewhere on the disk and a bootstrap is written into physical sector zero. If the CIL is written on a blank disk (i.e. freshly formatted disk). the disk should be booted as explained in System Manager's Guide and then the DSKINT option must be used to write the minimal file structure. DSKINT will not destroy the CIL which was written during SYSGEN. In this case, DSKINT will also create the file RSTS.CIL under account [Ø.1] to map the CIL. The refresh option should be used to create the other system files under [Ø,1].

Bootstraping is the mechanism by which the CIL area is defined. Hence, initializing a disk as a system disk will only preserve the CIL if it was booted. It is however possible to initialize a disk as a system disk and write the CIL later with SYSGEN. The procedure in this case is to mount any RSTS system disk on unit Ø (RK or RP) or to load a RSTS system onto an RF disk, boot this system disk, mount the disk to be initialized on any other disk drive, request the DSKINT option, and answer the DSKINT dialogue questions for the disk to be initialized. This procedure merely writes the minimal file structure plus the library account. It does not preserve anything on the disk except CIL AREA if booted device.

If a new CIL is written on an old RSTS system disk (containing RSTS files), the DSKINT option should not be used since it will destroy everything except the CIL. In this later case, the refresh option should be used to map the new CIL into the file RSTS.CIL and to verify that the file structure has not been corrupted.

Disk terminology

sector	the number of words transferred: 256. words for RK and RF, 1024. words for RP
physical disk address	the hardware-oriented address
number	an index to a sector relative to a known physical address.
logical number	an index to a sector of a file relative to the beginning of the file.
cluster	a contiguous group of file sectors.
cluster size	the number of sectors in a cluster (must be a power of 2)

Disk file structure

RSTS disk files exist in clusters of 256. (1024.) word sectors. The sectors within a cluster are always physically contiguous. The clusters themselves may be physically contiguous to each other; however, they may be scattered randomly across the surface of the disk.

Directory files

MFD (Master file directory)

The MFD on any file unit is a single file which [illegible] all user accounts (UFDs) on that file unit.

Directory files (cont'd)

(b) maintains accounting information for those accounts, and

(c) contains pointers to the beginnings of their respective UFDs.

(d) The MFD also contains all needed information for accessing any part of itself.

UFD (User file directory)

There is one UFD for each user account on a file unit which

(a) catalogs all program and data files under that account on that file unit,

(b) maintains accounting and access information for those files, and

(c) contains all needed retrieval information for those files.

(d) The UFD also contains all needed information for accessing any part of itself.

Program and data files exist on the disk as pure data clusters. They contain no retrieval information (i.e., no link-words) and no structural information.

Allocation mapping (the storage allocation table SAT)

A bit-map table (in file SATT.SYS) is used to record which disk clusters are allocated and which are free.

Each bit in the SAT corresponds to one pack-cluster. When a cluster is allocated, the corresponding bit is

Allocation mapping (cont'd)

is cleared. Note that the SAT records pack-clusters, and that one file-cluster may represent several (up to 256.) pack-clusters.

The SAT is manipulated in core. It is read from SATT.SYS into a 256. word buffer called SATBUF.

Directory structures and retrieval linkage

Each 256. word sector of a directory is logically subdivided into 32. block(ette)s of 9. words each. The following blockette-types exist:

label	only in sector 0 of cluster 0 of the UFD or MFD.
FDCM	file directory cluster map.
NB	name block.
AB	accounting block.
RB	retrieval block (only in UFDs).
HO	hole (blockette which is not in use)

The first sector of an MFD on any file unit is always located at sector 1.
The sector number of the first sector of each UFD is contained in a pointer within the MFD name blockeete for that account.

The first name blockette contains the beginnings of

threaded lists of NBs (NBs and RBs in the UFD0 chained together by link words.

Cataloging limits

The maximum number of user accounts that can be catalogued in an MFD is 1735.

The maximum number of user files that can be catalogued in a UFD is 1157.

Directory Structures and Retrieval Linkage

1) Directory Structures.

1 The MFD is always located at sector #1 on RK05's and at sector #2 on RP03's

2 Each 256-word segment of a file directory is logically subdivided into 32 blocks of 8 words each.

3 The first block of the MFD is a label block

4 The last block of every segment of a directory is the FDCM (File Directory Cluster Map). Every copy of it throughout a given directory is identically the same.

5 Any of the other blocks in a directory segment can be used as:

NB --- Name Block (in MFD or in UFDs).

AB --- Accounting Block (in MFD or in UFDs).

RB --- Retrieval Block (only in UFDs).

Any block never used or returned to free status is

classified as a

HO --- Hole (in MFD or in UFDs).

2) Retrieval Linkage.

The first segment of an MFD on an RP03 is always located at sector #2 and at sector #1 on on RKØ5

The segment # of the first segment of each UFD is contained in a pointer within the MFD Name Block for that account.

The first word of a MFD or UFD contains a pointer to the first Name Block in that MFD or UFD.

That first Name Block contains the beginnings of threaded lists (of NBs in the MFD, of NBs and RBs in the UFDs) chained together by link words.

The linking directs the access forward to any desired part of the directory. The linkage, however, is forward only. To regress, it is necessary to start once again from the beginning.

3) Addressing modes.

Two modes of addressing are used in the directories.

Some "address" entries are pointers containing the segment# (relative to the beginning of the disk) where some desired information begins. E.g. UAR entry in MFD's NB, and entries in FDCMs and RBs.

Other "address" entries are link-words, structured as indicated in the illustration.

4) Cataloguing limits.

The structure of the directories imposes certain maximum limits on the number of accounts and files which

can be catalogued.

Maximum number of user accounts that can be catalogued in MFD is 1735.

Calculated thus: [(7 clusters/MFD) * (16 sectors/cluster) * (31 blockettes/sector)-(1 label blockette)]/ MFD (2 blockettes/account).

Overhead for cataloguing 1735 accounts is

11% for RF system disk with one platter,

2.5% for RC/RK system disk, and

Ø.3% for RC/RP system dis.

Maximum number of user files that can be catalogued in a UFD is 1157.

Calculated thus: [(7 clusters/UFD) * (16 sectors/cluster) * (31 blockettes/Sector - 1 label blockette)]/ UFD (3 blockettes/file).

Clustersize

Advantages of large cluster sizes:

Reduces size of directories.

Reduces required number of disk-accesses to directories.

Reduces fragmentation of files on disk surface and fewer "seeks" will consequently be required to access a data record. But seek time may increase for individual seeks.

Advantages of small cluster sizes:

Avoids wasting disk space.

Lessens risk of being unable to create or extend a file --

i.e. after many creations, extensions, and deletions, a disk may still have sufficient free space to accomodate small clusters but lack sufficient contiguous space to accomodate large clusters.

Clustersize	minimum	maximum	When defined
ack (for non-system disk, public and private)	1 2 for RP03 only	16.	At initialization time via DSKINT option (or REFRESH for RSTS V4)
Directory (both for MFD and UFD)	Pack	16.	At creation of the directory via either DSKINT, REACT, or the SYS function.
File	Pack	256.	At creation of the file via either OPEN or OPEN FOR OUTPUT

File unit designation

(a) General unit designation:

DF:, DK:, DP:, or the null unit designation refer to all "public" file units.

When a file is to be created under this designation, the system will select the public file unit containing the most free space.

When a file is to be retrieved under this designation, the system will scan all public file unit directories until it finds the designated file.

(b) Explicit unit designation:

DFØ:, DKØ:, DK1:, DP3:, etc. refer to specific and distinct file units, which may be either "public" or "private".

Access to files

(a) Protection codes and privileges:

The protection code byte designates

1 read-protect against owner
2 write-protect against owner
4 read-protect against group
8 write-protect against group
16 read-protect against others
32 write-protect against others

64 compiled Basic program (only RSTS V5)

128 privileged program status for compiled program

If a user is logged in under a [1,x] account, or is running a privileged program, the program is allowed to bypass all protection-code limitations -- i.e., a privileged program may read a file irrespective of the file's status.

If neither the user nor the program is running "privileged", the restrictions imposed by a file's protection code apply. In such cases, no one can read, write, or delete any file protected against him, and the system will not allow renaming or modifying .BAC files (RSTS V4) or files whose extension is .BAC and/or have a protection code indicating that they are compiled (RSTS V5). In such cases, only the system editor is able to create or transfer .BAC files.

Normal and Update accessing.

Normal accessing of disk files.

When a user attempts to open a file in normal mode, the system will issue a "protection violation" error (which may be processed using the ON ERROR GOTO statement) if the file has already been opened by

someone else in update mode.

Then system then determines the user's read and write privileges as based on his account number and the file's protection code. The result may be read privileges, write privileges, both or neither.

The system then determines whether the job is running in privileged status, either because it is a privileged program (bit 128 set) or because it is under a system-manager type account [1,x]. If such be the case, priveleges are changed to read-write.

A check is then made to determine whether someone else has already obtained write-access to the file. If so, any write-privileges are taken away from the current petitioner.

If, as a result of the preceding, the job has neither read nor write privileges, the system issues "protection violation"; otherwise, it proceeds to open the file.

Update accessing of disk files.

If a user attempts to open a file in update mode which

is already open in normal mode, the system issues "protection violation".

The system then determines the user's privileges as outlined above.

If the job has, as a result of the preceding, both read and write privileges, it is allowed to proceed; else the system issues "protection violation".

The above outlines the process under RSTS V4. RSTS V5 may have some subtle differences (especially regarding privileged users). The privileged user or program under RSTS V5 is allowed to open disk and DECtape units in non-file-structured mode. GET and PUT thus refer to physical blocks. For example

```
OPEN "DKØ:" AS FILE 1%
```

opens DKØ in non-file-structured mode.

Pointer values in the Bootstrap Block

$Offset_8$	Meaning
160	The absolute starting logical block of the first CIL core image
162	The load address of the first CIL core image
164	The size in bytes of the first CIL core image

Pointer values in the Bootstrap Block (cont'd)

166 The transfer address of the first CIL core image

170 The CIL logical block size in words per block

172 The first address (Pseudo load address) of the bootstrap (I.e., where bootstrap relocates to).

174 The external page address of the system device

176 The absolute starting logical block of the CIL

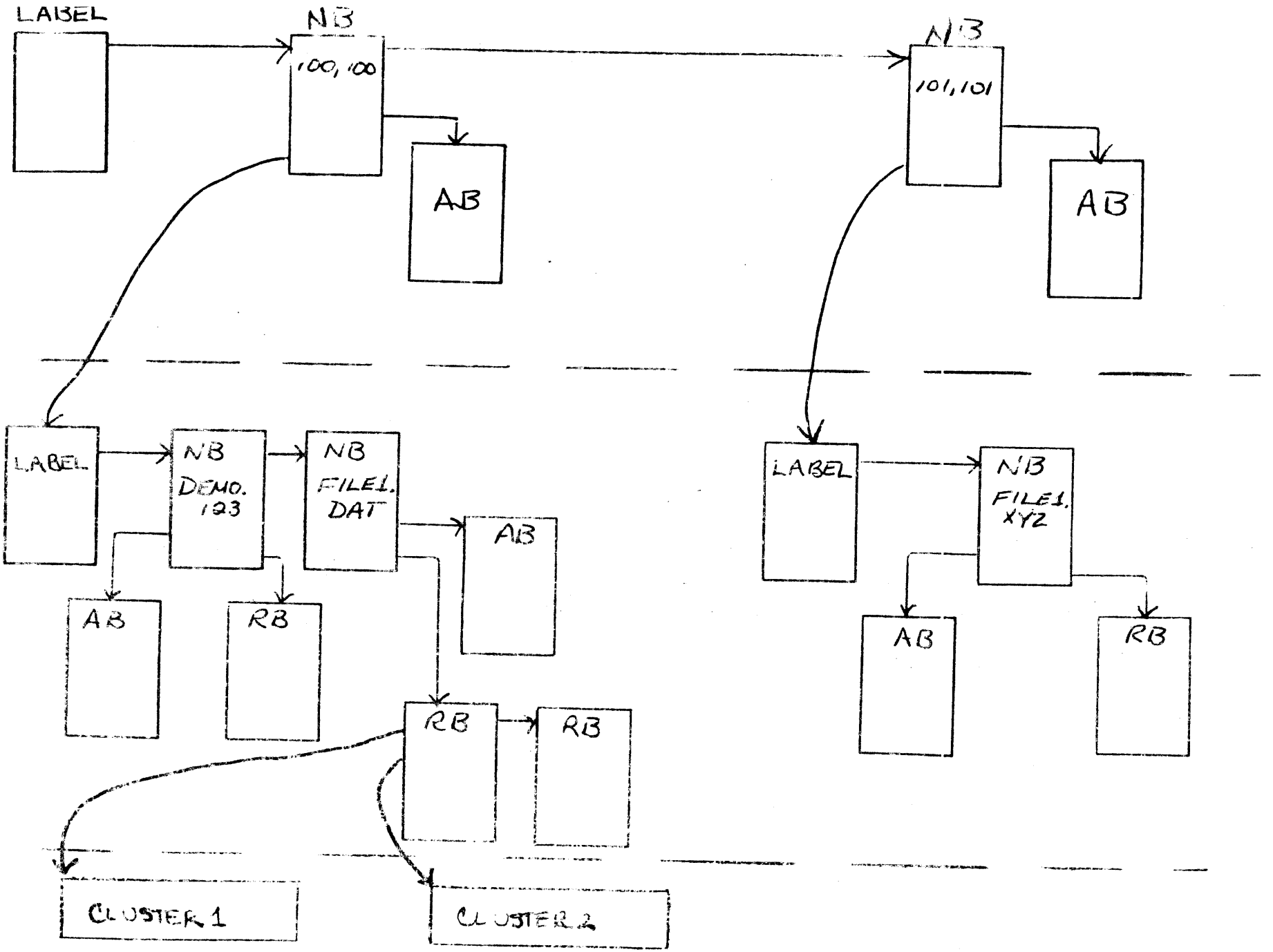
Bootstrap operation:

After relocating itself to high-core (using the address set in location 172), the bootstrap sets up a transfer from the disk using the following parameters:

disk address	word 160
word count	word 164
memory address	word 162

If the transfer was successful, the program is started using the address stored in word 166.

The bootstrap normally relocates itself to XXX250 where 'XXX' refers to the highest 4K bank of core available (up to 28K, of course).


LABEL
NB
100, 100
AB
NB
101, 101
AB
LABEL
NB
DEMO.
123
NB
FILE1.
DAT
AB
AB
RB
RB
RB
CLUSTER 1
CLUSTER 2
LABEL
NB
FILE1.
XYZ
AB
RB

CC-5

$$\text{Physical Sector \#} = \left[\left[\begin{matrix}\text{DEVICE}\\ \text{CLUSTER}\\ \text{SIZE}\end{matrix} * \left(\begin{matrix}\text{RETRIEVAL}\\ \text{INFORMATION}\\ \text{BLOCK ENTRY}\end{matrix} - 1\right)\right] + 1\right]$$

Actual Seg #'s Locations

Pure Data

@ Sector 2 RP03

1st segment of MFD is @ Sector 1 RK05

MFD

UFD

FILE

MFD/PACK LABEL
NB_1
AB_1
NB_2
AB_2
NB_3
AB_3
etc.
etc.
etc.
FILE DIRECTORY CLUSTER MAP

256_{10} WORDS (32 8-WORD BLOCKETTES)

dynamically allocated

UFD LABEL (Dummy Block)
NB_α
AB_α
$RB_{\alpha 1}$
$RB_{\alpha 2}$
$RB_{\alpha 3}$
NB_β
AB_β
$RB_{\beta 1}$
$RB_{\beta 2}$
$RB_{\beta 3}$
$RB_{\beta 4}$
NB_γ
AB_γ
$RB_{\gamma 1}$
$RB_{\gamma 2}$
FILE DIRECTORY CLUSTER MAP

256_{10} WORDS (32 8-WORD BLOCKETTES)

8 WORDS

LINK TO NEXT RB
POINTER TO CLUSTER 0
POINTER TO CLUSTER 1
POINTER TO CLUSTER 2
POINTER TO CLUSTER 3
POINTER TO CLUSTER 4
POINTER TO CLUSTER 5
POINTER TO CLUSTER 6

8 WORDS

LINK TO NEXT RB
POINTER TO CLUSTER 7
POINTER TO CLUSTER 8
POINTER TO CLUSTER 9
POINTER TO CLUSTER 10
POINTER TO CLUSTER 11
POINTER TO CLUSTER 12
POINTER TO CLUSTER 13

CLUSTER # 0
SEG # 1
SEG # 2
SEG # 3
SEG # 4
ETC.
SEG # n-2
SEG # n-1
SEG # n
SEG # n+1

CLUSTER # 1
SECTOR # n+2
SECTOR # n+3
SECTOR # n+4
ETC
ETC
SECTOR #2n-1
SECTOR #2n

CLUSTER # 2
ETC.

ETC.

ETC

CLUSTER # 13
SECTOR # 13n+1
SECTOR # 13n+2
ETC
SECTOR # 14n-1
SECTOR # 14n

ETC

ETC

CLUSTER # Z/C - 1
SECTOR #N-n+1
SECTOR #N-n+2
ETC
SECTOR #N-1
SECTOR #N

LOGICAL
#0
#1
#2
ETC
ETC
#n-2
#n-1
#n
#n+1
#n+2
ETC
ETC
#2n-2
#2n-1
#2n
#N-n
#N-n+
ETC
#N-3
#N-2
#N-1

NAME

BLOCK

Accounting Block

Retrieval Block

→ denotes a link (offset relative to beginning of sector)

→ denotes a pointer (absolute sector address)

N = # of sectors in file

n = # of sectors in cluster

Each sector = 256_{10} WORDS.

The number of retrieval blocks per file is Not limited ...As many as needed are used and are contained in UFD.

Large

There are seven cluster pointers per Retrieval Block

Each cluster pointer points to a beginning segment number for the File Cluster of sectors. The Segment Number is Not a physical disk address, but is converted into one.

All of the sectors in a cluster are physically contiguous.

In 5-21 PAK always cleaned on Start

In 5B-24 PACK cleaned if Bit 15 Pack Status word of MFD is set during Start

Refresh Pack is _____

Default Pack is _____

MASTER FILE DIRECTORY (MFD) BLOCKETTE FORMAT

MFD LABEL
(ONE ONLY PER MFD)

LINK TO FIRST NB IN MFD
-1
Ø
Ø
PACK CLUSTER SIZE
PACK STATUS
PACK ID. (CHARS 1 - 3) RAD5Ø
PACK ID. (CHARS 4 - 6) RAD5Ø

MFD ACCOUNTING BLOCK (AB)
(ONE PER ACCOUNT CATALOGUED IN MFD)

+1 (TO SIGNAL THAT BLOCK IS IN USE) (-2 IF BAD DISK BLOCK)	
ACCUMULULATED CPU TIME FOR THIS ACCOUNT	
ACCUMULATED CONNECT TIME FOR THIS ACCOUNT	
ACCUMULATED KILO-CORE-TICKS FOR THIS ACCOUNT	
ACCUMULATED DEVICE TIME FOR THIS ACCOUNT	
ACCUMULATED # OF LOG-OUTS	ACCUMULATED # OF LOG-INS
DISK QUOTA - OF DISK BLOCKS PERMITTED AT LOG-OUT	
UFD CLUSTER FACTOR	

MFD FILE DIRECTORY CLUSTER MAP (FDCM)
(ONE BLOCK OF THE MFD)

FILE DIRECTORY CLUSTER SIZE
BLOCK # OF FIRST BLOCK IN CLUSTER Ø IN MFD
BLOCK # OF FIRST BLOCK IN CLUSTER 1 IN MFD
BLOCK # OF FIRST BLOCK IN CLUSTER 2 IN MFD
BLOCK # OF FIRST BLOCK IN CLUSTER 3 IN MFD
BLOCK # OF FIRST BLOCK IN CLUSTER 4 IN MFD
BLOCK # OF FIRST BLOCK IN CLUSTER 5 IN MFD
BLOCK # OF FIRST BLOCK IN CLUSTER 6 IN MFD

MFD NAME BLOCK (NB)
(ONE PER ACCOUNT CATALOGUED IN MFD)

LINK TO NEXT NB IN MFD (OR Ø IF END)	
PROJECT #	PROGRAMMER #
PASSWORD (CHARS 1 - 3) RAD5Ø	
PASSWORD (CHARS 4 - 6) RAD5Ø	
PROTECTION CODE	STATUS CODE
CURRENT ACCESS COUNT	
LINK TO AB FOR THIS ACCOUNT	
BLOCK OF FIRST UFD BLOCK FOR THIS ACCOUNT (OR Ø)	

HOLE (HO) - UNUSED BLOCK

Ø
Ø

CURRENT ACCESS COUNT - SIGNALS WHETHER ACCOUNT IS IN USE. IT IS INCREMENTED BY 1 AT LOG-IN AND DECREMENTED BY 1 AT LOG-OUT.

PROTECTION CODE IS MEANINGFUL IF UFD IS OPENED AS A FILE.

PACK STATUS:
IF BIT 14 IS SET TO 1, THE PACK IS PRIVATE. ELSE IT IS PUBLIC. SYSTEM DISKS ARE ALWAYS MARKED AS PRIVATE; WHEN USED AS THE SYSTEM DISK, THEY ARE TREATED AS PUBLIC, BUT WHEN MOUNTED ON ANOTHER SYSTEM THEY WILL BE PRIVATE.
BIT 15 IS SET TO 1 ON A "MOUNT" AND TO Ø ON A"DISMOUNT". (IF A NON-MOUNTED DISK HAS BIT 15 SET TO A 1, THEN IT WAS NOT PROPERLY "DISMOUNTED" (EG. SYSTEM CRASH) AND NEEDS TO BE "CLEANED".
BITS 13 - Ø ARE CLEARED TO Ø

MFD ID BLOCK

MFD Address (Link) of First Name Block	Ø	ULNK Ø
-1	2	Dummy Entry (not to
0	4	be confused with
0	6	account [255,255])
Pack Cluster Size (1,2,4,8, or 16)	1Ø	
Clean Priv 0	12	Public/Priv.Pack Ind. Clean/Not Clean Ind.
Pack ID (2 wds in RAD 50)	14	
Pack ID (2 wds in RAD 50)	16	

UFD ID BLOCK

UFD Address of First NB	Ø	
-1	2	Dummy Name
0	4	(Impossible)
0	6	

File Directory Cluster Map

File Directory Cluster Size	0	
Seg # of first segment in cluster Ø of MFD	2	
1	4	One for
2	6	each sector
3	10	of directory
4	12	whether MFD
5	14	or UFD
6	16	

	Link to next name, 0 is end of chain	ULNK	0
	Project No. | Programmer No.	UNAM	2
	Password Rad 50		4
	Password Rad 50		6
11 UPROT	Protection | Status	USTAT	10
	CURRENT ACCESS COUNT	UACNT	12
	ACCOUNT BLOCK LINK	UAA	14
	UFD Segment Start, 0 is no UFD	UAR	16

MFD NAME BLOCK

	+1 if block in use, -2 if bad block	ULNK	0
	Accum CPU Time for Account *	MCPU	2
	CPU TIME 4 bits | Connect time for account	MCON	4
	Accum Kilo Core Ticks *	MKCT	6
	Accum Device Time *	MDVT	10
	CPU TIME 6 bits | 10 Bits KCT	MLDGI	12
	Total Segments Permitted	MDPER	14
	UFD Cluster Size	UCLUS	16

MFD ACCOUNT BLOCK *Reset by Read Data Reset Accounting

Protection Byte Bit Assignments (UPROT) UFD/MFD

Bit 15	Privilege Bit, used to set JFSYS
14	1 = Run only
13	Write Protect from World
12	Read Protect from World
11	Write Protect from Group
10	Read Protect from Group
9	Write Protect from Self
8	Read Protect from Self

Status Byte Bit Assignments (USTAT) UFD/MFG

Bit 7	Marked for Delete if 1
6	File Entry if 0, MFD/UFD Entry if 1
5	Not to be Killed if 1
4	Not to be Extended if 1
3	File Already Open for Update
2	File Already Open for Write
1	Must be Zero
0	File is 'Out of Sat' if 1

USER FILE DIRECTORY (UFD) BLOCKETTE FORMAT

UFD LABEL
(ONE ONLY PER UFD)

LINK TO FIRST NB IN UFD. (OR Ø)
-1
Ø
Ø
Ø
Ø
Ø
Ø

UFD NAME BLOCK (NB)
(ONE PER FILE CATALOGUED IN UFD)

LINK TO NEXT NB IN UFD (OR Ø IF END)	
FILENAME (CHARS 1 - 3) RAD5Ø	
FILENAME (CHARS 4 - 6) RAD5Ø	
EXTENSION RAD5Ø	
PROTECTION CODE	STATUS
FILE ACCESS COUNT	
LINK TO AB FOR THIS FILE	
LINK TO FIRST RB FOR THIS FILE (OR Ø IF FILE IS OF Ø LENGTH)	

UFD ACCOUNTING BLOCK (AB)
(ONE PER FILE CATALOGUED IN UFD)

+1 (TO SIGNAL THAT BLOCK IS IN USE) (-2 IF BAD DISK BLOCK)
LAST ACCESS DATE
NUMBER OF BLOCKS IN FILE
CREATION DATE
CREATION TIME
(UNUSED)
(UNUSED)
CLUSTER FACTOR FOR THIS FILE

UFD RETRIEVAL BLOCK (RB)
(AS MANY PER FILE AS LENGTH REQUIRES)

LINK TO NEXT RB FOR THIS FILE (OR Ø IF LAST RB)
BLOCK# OF FIRST BLOCK IN CLUSTER Ø (7,14,21 etc) OF FILE
BLOCK# OF FIRST BLOCK IN CLUSTER 1 (8,15,22 etc) OF FILE
BLOCK# OF FIRST BLOCK IN CLUSTER 2 (9,16,23 etc) OF FILE
BLOCK# OF FIRST BLOCK IN CLUSTER 3 (1Ø,17,24 etc) OF FILE
BLOCK# OF FIRST BLOCK IN CLUSTER 4 (11,18,25 etc) OF FILE
BLOCK# OF FIRST BLOCK IN CLUSTER 5 (12,19,26 etc) OF FILE
BLOCK# OF FIRST BLOCK IN CLUSTER 6 (13,2Ø,27 etc) OF FILE

UFD FILE DIRECTORY CLUSTER MAP (FDCM)
(ONE PER BLOCK OF THE UFD)

FILE DIRECTORY CLUSTER SIZE
BLOCK# OF FIRST BLOCK IN CLUSTER Ø OF UFD
BLOCK# OF FIRST BLOCK IN CLUSTER 1 OF UFD
BLOCK# OF FIRST BLOCK IN CLUSTER 2 OF UFD
BLOCK# OF FIRST BLOCK IN CLUSTER 3 OF UFD
BLOCK# OF FIRST BLOCK IN CLUSTER 4 OF UFD
BLOCK# OF FIRST BLOCK IN CLUSTER 5 OF UFD
BLOCK# OF FIRST BLOCK IN CLUSTER 6 OF UFD

UFD HOLE (HO UNUSED BLOCK)

Ø
Ø

UFD LABEL
(ONE ONLY PER UFD, AT
CLUSTER Ø, SECTOR Ø, BLOCK Ø)

+Ø	LINK TO FIRST NB IN UFD (OR Ø)
+2	-1
+4	Ø
+6	Ø
+1Ø	Ø
+12	Ø
+14	Ø
+16	Ø

UFD NB - NAME BLOCK
(ONE PER FILE CATALOGUED IN UFD)

+Ø	LINK TO NEXT NB IN UFD (OR Ø IF END OF CHAIN)		ULNK=Ø
+2	FILE NAME (PART 1) RAD5Ø		UNAM=2
+4	FILE NAME (PART 2) RAD5Ø		
+6	EXTENSION RAD5Ø		
+1Ø	PROTECTION CODE	STATUS	USTAT=1Ø UPROT=11
+12	FILE ACCESS COUNT		UACNT=12
+14	LINK TO AB FOR THIS FILE		UAA=14
+16	LINK TO FIRST RB FOR THIS FILE (OR Ø IF FILE IS OF Ø LENGTH)		UAR=16

UFD AB - ACCOUNTING BLOCK
(ONE PER FILE CATALOGUED IN UFD)

+Ø	+1 (TO SIGNAL THAT BLOCK IS IN USE) (-2 IF BAD DISK BLOCK)	ULNK=Ø
+2	LAST ACCESS DATE	UDLA=2
+4	NUMBER OF SECTORS IN FILE	USIZ=4
+6	CREATION DATE	UDC=6
+1Ø	CREATION TIME	UTC=1Ø
+12	RTS NAME RAD5Ø	
+14	RTS NAME RAD5Ø	
+16	CLUSTER FACTOR FOR THIS FILE	UCLUS=16

UFD RB - RETRIEVAL BLOCK
(AS MANY PER FILE AS LENGTH
OF FILE REQUIRES)

+Ø	LINK TO NEXT RB FOR THIS FILE (OR Ø IF LAST RB)	ULNK=Ø
+2	SEG. # OF FIRST SEG. IN CLUSTER Ø (7, 14, 21, ETC.) OF FILE	UENT=2
+4	SEG. # OF FIRST SEG. IN CLUSTER 1 (8, 15, 22, ETC.) OF FILE	
+6	SEG. # OF FIRST SEG. IN CLUSTER 2 (9, 16, 23, ETC.) OF FILE	
+1Ø	SEG. # OF FIRST SEG. IN CLUSTER 3 (1Ø, 17, 24, ETC.) OF FILE	
+12	SEG. # OF FIRST SEG. IN CLUSTER 4 (11, 18, 25, ETC.) OF FILE	
+14	SEG. # OF FIRST SEG. IN CLUSTER 5 (12, 19, 26, ETC.) OF FILE	
+16	SEG. # OF FIRST SEG. IN CLUSTER 6 (13, 2Ø, 27, ETC.) OF FILE	

UFD FDCM - FILE DIRECTORY CLUSTER MAP
(ONE FOR EACH SECTOR OF UFD)

+Ø	FILE DIRECTORY CLUSTER SIZE
+2	SEG. # OF FIRST SEG. IN CLUSTER Ø OF UFD
+4	SEG. # OF FIRST SEG. IN CLUSTER 1 OF UFD
+6	SEG. # OF FIRST SEG. IN CLUSTER 2 OF UFD
+1Ø	SEG. # OF FIRST SEG. IN CLUSTER 3 OF UFD
+12	SEG. # OF FIRST SEG. IN CLUSTER 4 OF UFD
+14	SEG. # OF FIRST SEG. IN CLUSTER 5 OF UFD
+16	SEG. # OF FIRST SEG. IN CLUSTER 6 OF UFD

HO - HOLE (UNUSED BLOCK)

+Ø	Ø
+2	Ø
+4	
+6	
+1Ø	
+12	
+14	
+16	

Link to Next Name Block,0 is End of Chain	ULNK	0
Filename	UNAM	2
Filename		4
Extension		6
Priv | WR | ALL | RD | ALL | WR | GRP | RD | GRP | WR | OWN | RD | OWN | Status Byte	USTAT	10
Access Count	UACNT	12
Account Block Link	UAA	14
Retrieval Block Link,0 is No File Space	UAR	16

UPROT

UFD NAME BLOCK

+1 if Block in Use,-2 if Bad Block	ULNK	0
Date of Last Access	UDLA	2
File Segment Size	USIZ	4
Date of Creation	UDC	6
Time of Creation	UTC	10
RTS Name Rad 50		12
RTS Name Rad 50		14
File Cluster Size	UCLUS	16

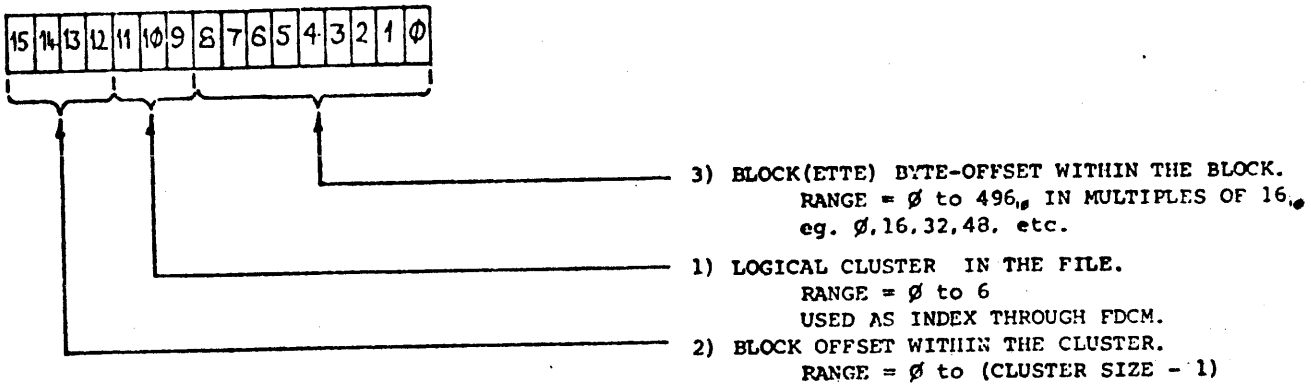
UFD ACCOUNT BLOCK

Link to Next Rib, 0 is End of Chain	ULNK	0
Physical Segment # of Cluster 0 Start		
Physical Segment # of Cluster 1 Start		
If 0 then end of list		

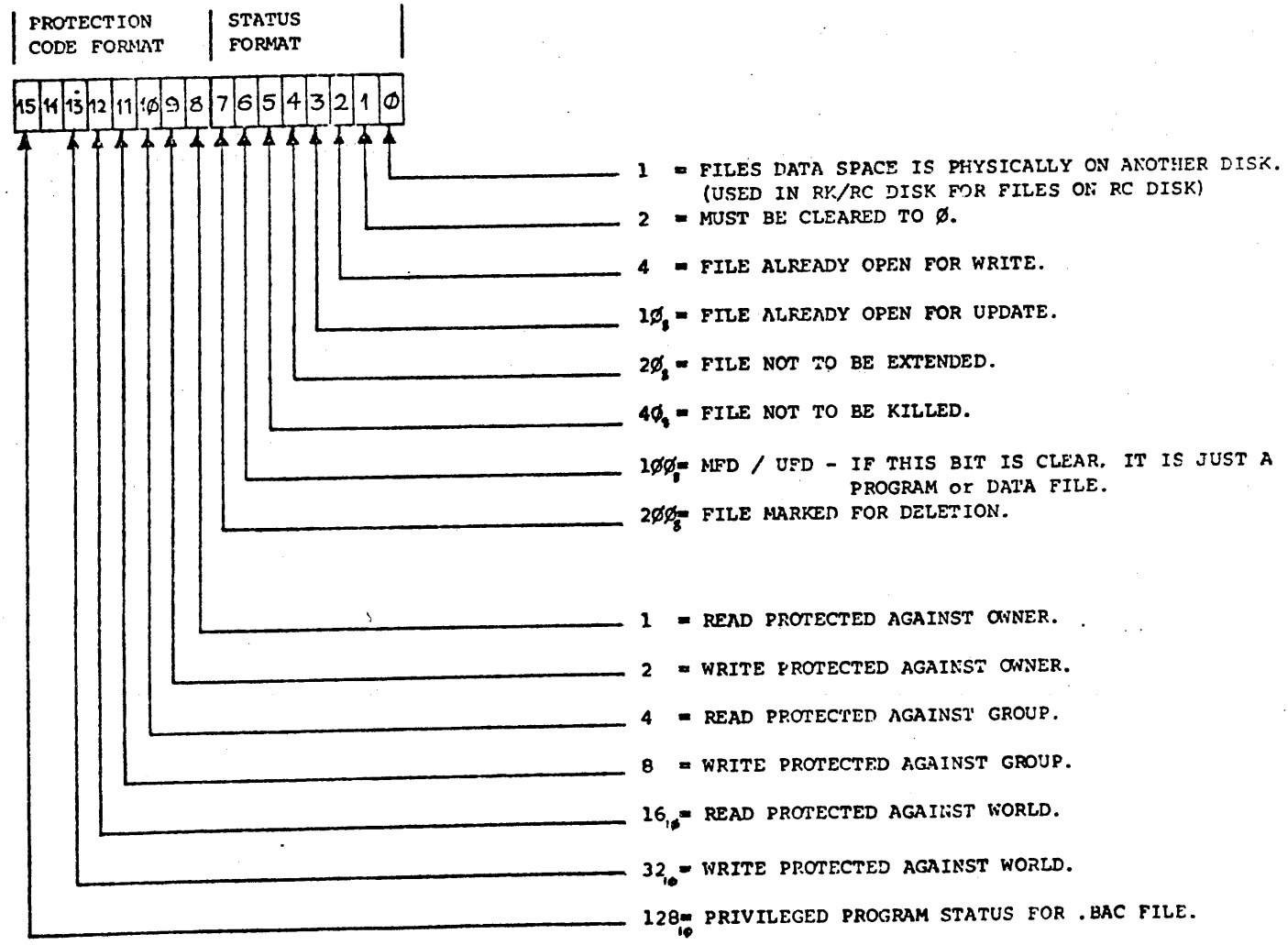
RIB

Retrieval Information Block

FORMAT OF LINK WORDS IN DISK DIRECTORIES



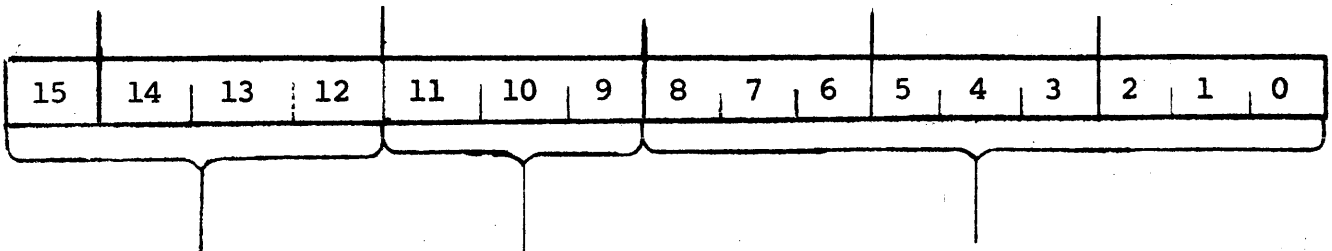
PROTECTION CODE AND STATUS FORMATS IN NAME BLOCKS OF DIRECTORIES



Protection Code							Status Word								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	Ø

STATUS BIT	WORD	
Ø	1_8	Files data space is physically on another disk
1	Ø	Must be zero
2	4_8	File open for write
3	$1Ø_8$	File open for read'
4	$2Ø_8$	File may not be extended
5	$4Ø_8$	File may not be killed
6	$1ØØ_8$	MFD/UFD = 1; program or data = Ø
7	$2ØØ_8$	File marked for deletion
PROTECTION CODE		
8	1	Read protect against owner
9	2	Write protect against owner
10	4	Read protect against group
11	8	Write protect against group
12	16_{10}	Read protect against world
13	32_{10}	Write protect against world
14	64_{10}	BAC file
15	128_{10}	BAC Program is privileged

FORMAT OF A LINK IN DIRECTORY

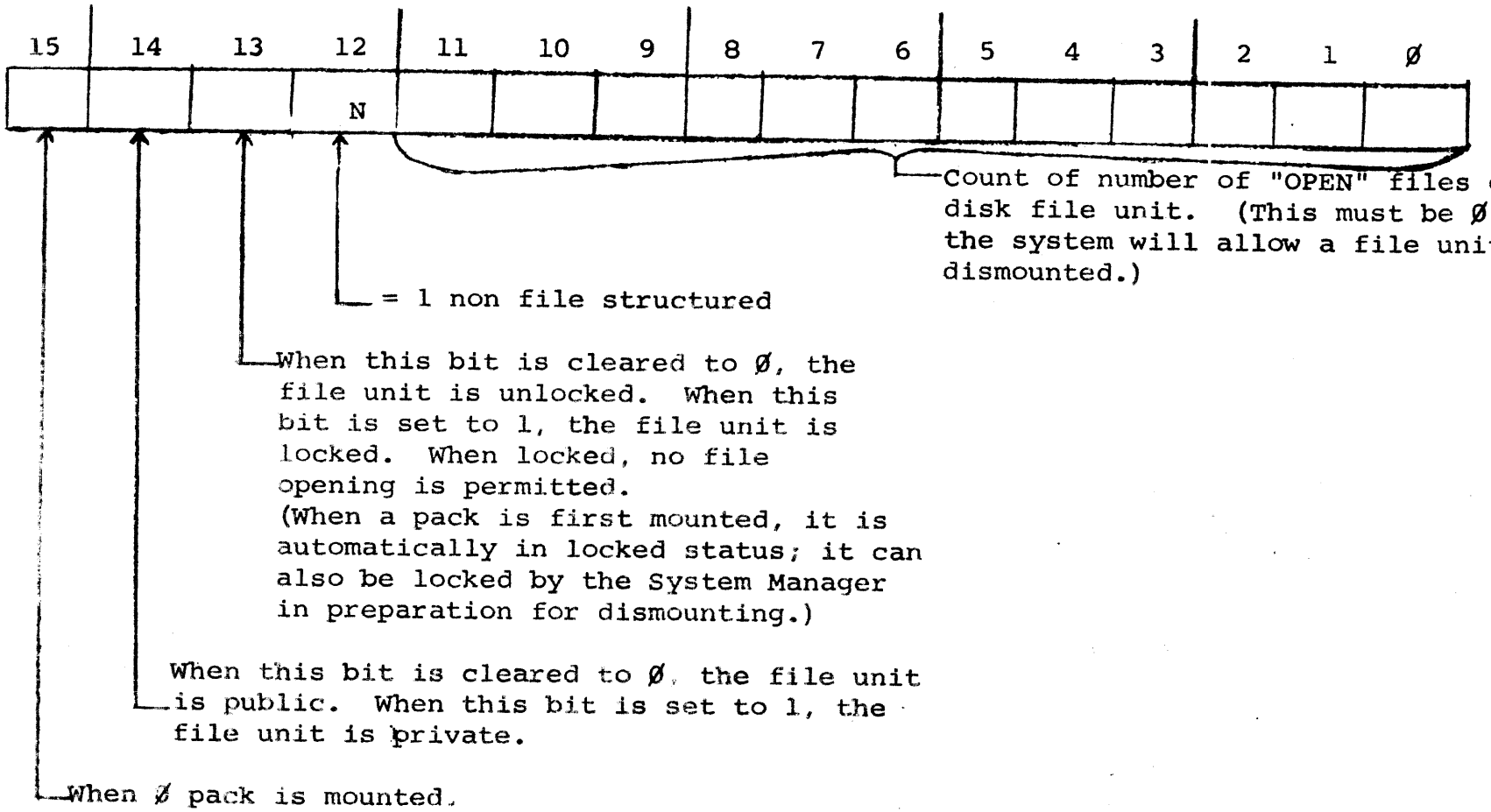


3) Blockette byte-offset within sector
Range Ø to 496_{10} in multiples of 16_{10}

1) Logical cluster number in the file
Range Ø to 6

2) Sector offset within the clust
Range Ø to clustersize - 1

PACK STATUS AND OPEN COUNT FOR UNTCNT



Notes on DECtape processing

1) DECtape file handling.

A DECtape request to the File Processor causes the following actions:

a) Parameters are set in the DECtape DDB.

b) The DDB is linked into the DECtape Symbiont queue (DTSQ)

c) If the symbiont had been inactive, the QDTSYM bit is set in L3QUE.

q d) An exit is made from the File Processor.

Sooner or later the L3QUE mechanism (extended priority arbitration) will start the DECtape Symbiont.

The Symbiont will obtain needed buffers (directory information on BUFF.SYS).

The Symbiont will place the request in the DECtape driver queue (DTDRQ).

The DECtape driver will initiate the transfer.

When I/O is complete, the driver causes DTSRET to be run a level 3.

If there are errors, these are passed to the FIP function DTSERR.

If there are no errors, the FIRQB is re-inserted into the FIP queue, and FIPSYS is entered.

If other requests are in the symbiont queue, DTSRET sets the QDTSYM bit in L3QUE, causing the Symbiont to be rerun as soon as possible.

2) DECtape directory handling.

The DECtape UFD (blocks 1Ø2,1Ø3) and Permanent Bit Map (block 1Ø4) are copied onto disk when first required (into BUFF.SYS).

The disk sector number (in BUFF.SYS) assigned to a given DECtape drive for holding its directory blocks is stored in that drive's DDB+26 (DTDSKØ word).

Byte 1 (DTDSTS) of a DECtape DDB indicates the status of that directory on the disk.

Bit 15 (DTDSK) is set when the directory on the disk represents that of the currently mounted DECtape.

Bits 14,13,12 are set respectively when the UFD no.1, UFD no.2, or PBM have been altered on the disk and therefore need to be written back onto the DECtape in their updated form.

The directory blocks are written back onto the DECtape at the time of a delete, a renaming, or a CLOSE. If there is an open output file, the directory re-writing is delayed until the CLOSE.

Bit 15 is cleared whenever the Init count (DDB+36) of files opened minus files closed becomes zero.

3) Cautions for users.

Users can avoid unnecessary repeated re-reads of the DECtape directories by keeping at least one DECtape file open in the program until the program has completed all its DECtape processing.

When a user mounts a DECtape on a drive, he should not

assume that the last person who used that drive under his account closed the last file. It is safer to force the system to read in the DECtape directory afresh by the command CATALOG DTn:. (writes Directory on BUFF.SYS)

DOS/RSTS DECtape Comaptibility

Problems:

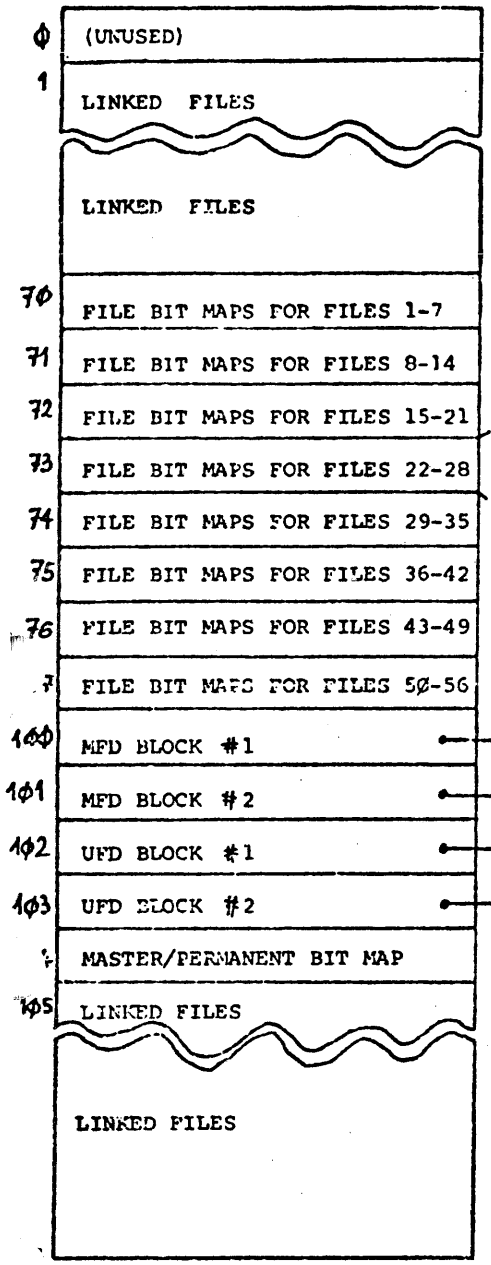
An improperly closed DOS generated DECtape file (usage and/or lock bits non-zero) will print with an invalid protection when cataloged under RSTS.

Restrictions:

1) RSTS will not allow processing (OPEN, CREATE, or DELETE) of contiguous DECtape files.
2) In processing (INPUT) DOS generated linked DECtape files, RSTS will ignore the "partial block" indications.
3) A RSTS generated (always linked) DECtape file cannot be appended to or extended under DOS.

This means that formatted ASCII linked DECtape files (properly closed) are still 1ØØ% compatible in the following sense:

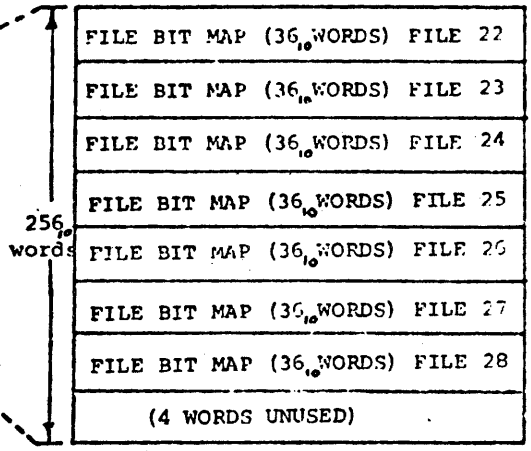
1) If I write it under DOS, when I read it under RSTS I get the same data.
2) If I write it under RSTS, when I read it under DOS I get the same data.



A DECtape has 576_{10} ($=1100_8$) blocks of 256_{10} words each ($=1000_8$ byt

The first word in every block of a linked file is a pointer to the next logical block of that file.
(the pointer contains the physical block # of the next logical block; it is positive for forward tape motion and negative for backward tape motion.)

The remaining 255_{10} words are data.
DECtape directory structure can catalog and map a max of 56_{10} file



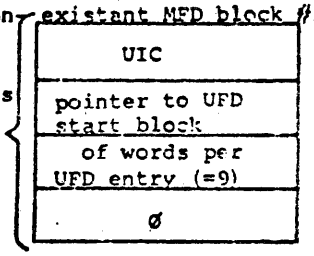
Each File Bit Map has 36_{10} word ($=576_{10}$ bits)

Each bit maps 1 block of the DECtape. (A set bit means an allocated block; a clear bit mean a free block.)

Each File Bit Ma maps the entire DECtape.

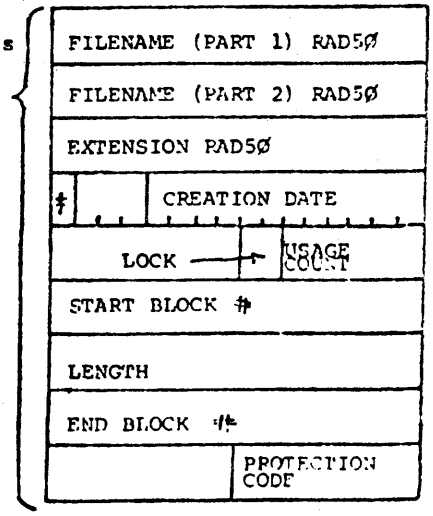
256_{10} words:
- 1st word: 101_8 (link to MFD block #2)
- 2nd word: 4 (interleave factor)
- 3rd word: 104_8 (pointer to 1st Master Bit Map)
- 4th word: 104_8 (pointer to non-existant 2nd Master Bit Map)
- 252 words unused

256_{10} word:
- 1st word: 0 (link to non-existant MFD block #3)
- up to 63_{10} 4-word entries each with format shown



- remainder unused.

256_{10} words:
- 1st word: 103_8 (link to next UFD block) or 0 (end of chain)
- 28_{10} file entries (max) 9 words each with this format:



‡ = File Type

- 3 words unused

256_{10} words:
- 4 word header
- 36_{10} word bit map of entire DECtape (a logical ORing of all File Bit Maps)

MFD BLOCK #2
RSTS does not read or check DECtape UIC's.
When zeroing a DECtape RSTS enters a UIC of [1,1].

UFD BLOCKS
RSTS ignores - LOCK Bits, USAGE COUNT, and END BLOCK entries in the UFD.
RSTS checks the "TYPE" bit (bit 15) and will not allow a "CONTIGUOUS" file to be OPENed.
When creating a DECtape file, RSTS writes a protection code of 233_8 (for DOS-11 compatibility). But RSTS does not read or check DECtape protection codes.

Update accessing of disk files.
If a user attempts to open a file in update mode and that file has already been opened by someone else in normal mode, the system issues a "protection violation" message. Otherwise the system proceeds to the next step.
The system then determines the user's read and write privileges from the user's account number and the file's protection code.
If the job is running in privileged status (as described above), then its privileges are extended to read-&-write.
If the job has, as a result of the preceding, both read and write privileges, it is allowed to proceed; else a "protection violation" message is issued.

f) Miscellaneous Notes.

1) Basic file structures are imposed on disks through DSKINT.

2) UFDs are created only when needed.

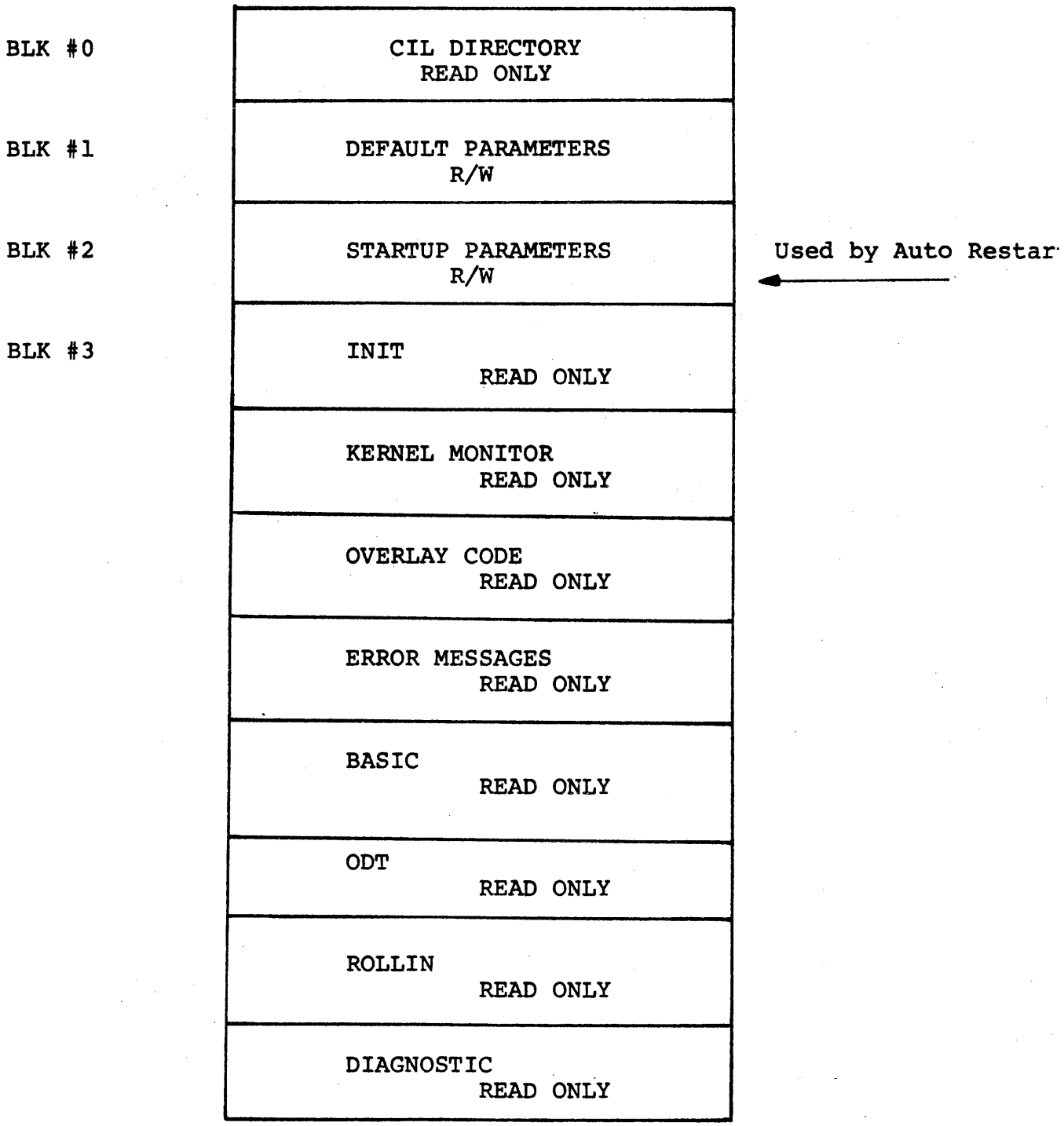
There is a distinct interrupt handler for each type of disk
A maximum of 5 retries are attempted when a disk error is reported.
If the error persists after the fourth retry, the operation is aborted with an error flag for the caller.
All five errors are logged. All disk I/O routines are core-resident (but not all File Processor routines). FIP routines are optionally resident.
RP and RK disk driver code optimizes "seeks".

Distance which arm travels in "seek" is minimized by picking the next operation to be performed on the basis of the current position of the arm.
Simultaneously a fairness-count is kept for each waiting operation lest it wait in the queue indefinitely.
Thus the selection of the next operation is made on a twofold basis: (a) closeness of the operation to the arm's current position and (b) time in the queue.

BADB.SYS created at DSKINTIME, catalogs all known bad disk sectors.
The system will read this file and mark those sectors in the SAT as allocated.

CIL MAP

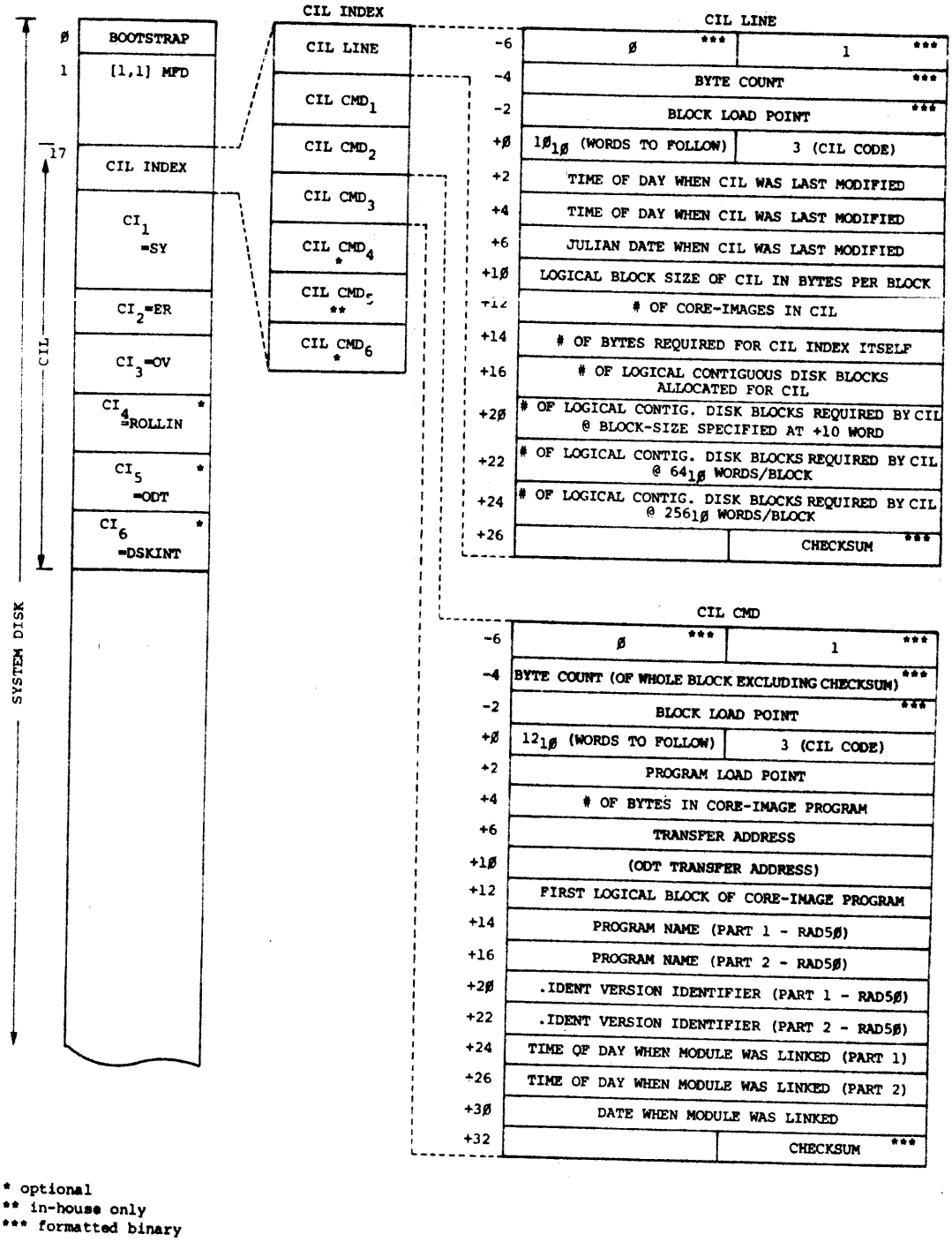
RSTS.CIL DESCRIPTION



"Startup Parameters" are in effect for this run only.

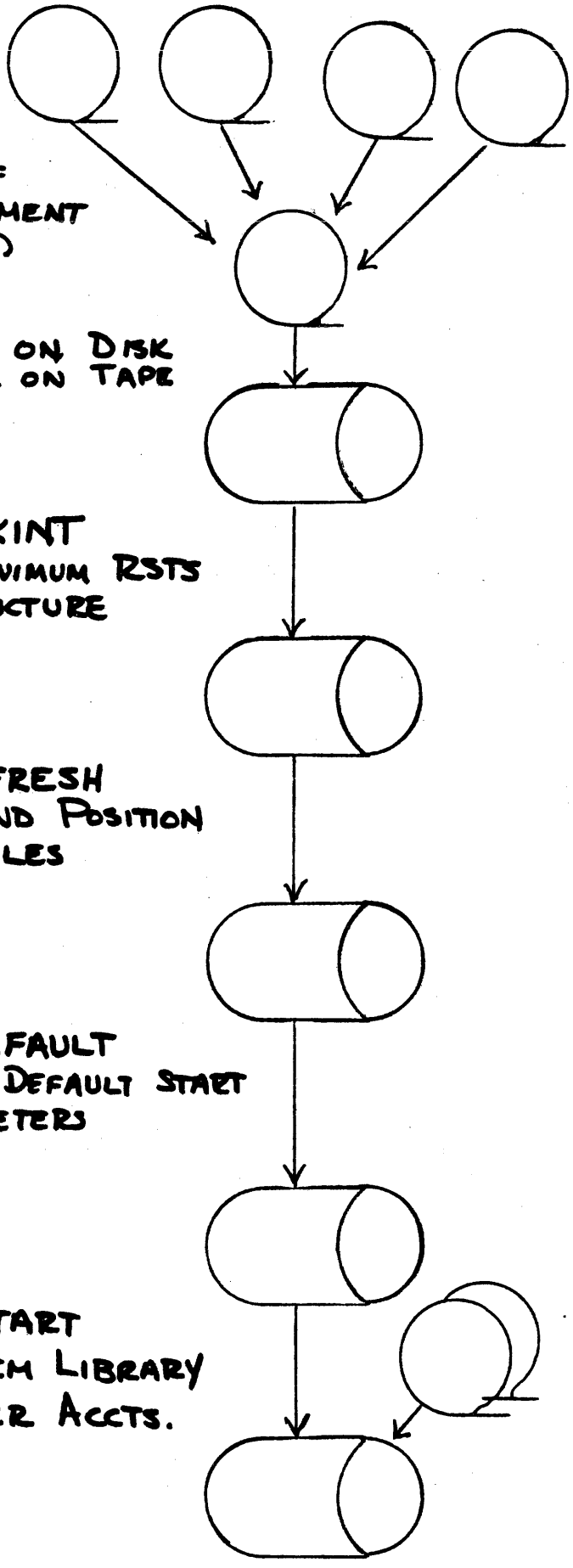
"Default Parameters' are established by the Default establishment in reply to the Default option.

Startup Parameters may or may not be the same as Default.



* optional
** in-house only
*** formatted binary

DIAGRAMS



DISTRIBUTION TAPES

1. CREATION OF SYSTEM ELEMENT (DOS/BATCH)

LCL

2. SYSLOD
   BUILD CIL ON DISK FROM LCL ON TAPE

CIL ON SYSTEM DISK

3. OPTION: DSKINT
   CREATE MINIMUM RSTS FILE STRUCTURE

CIL
0,1 SYSTEM FILES
1,1 MFD
1,2 LIBRARY ACCT

4. OPTION: REFRESH
   CREATE AND POSITION SYSTEM FILES

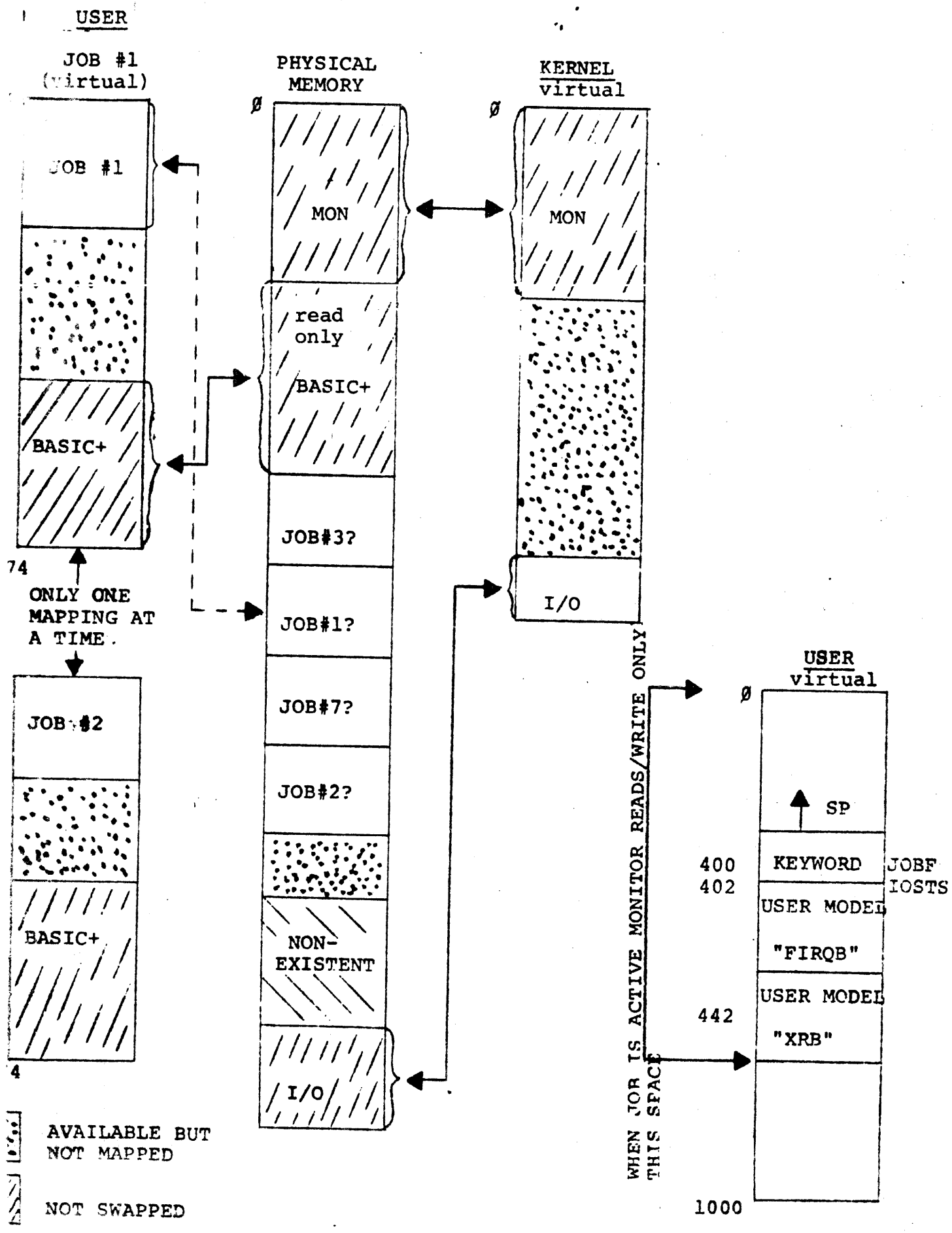
ALL 0,1 FILES PLUS THE ABOVE

5. OPTION: DEFAULT
   ESTABLISH DEFAULT START UP PARAMETERS

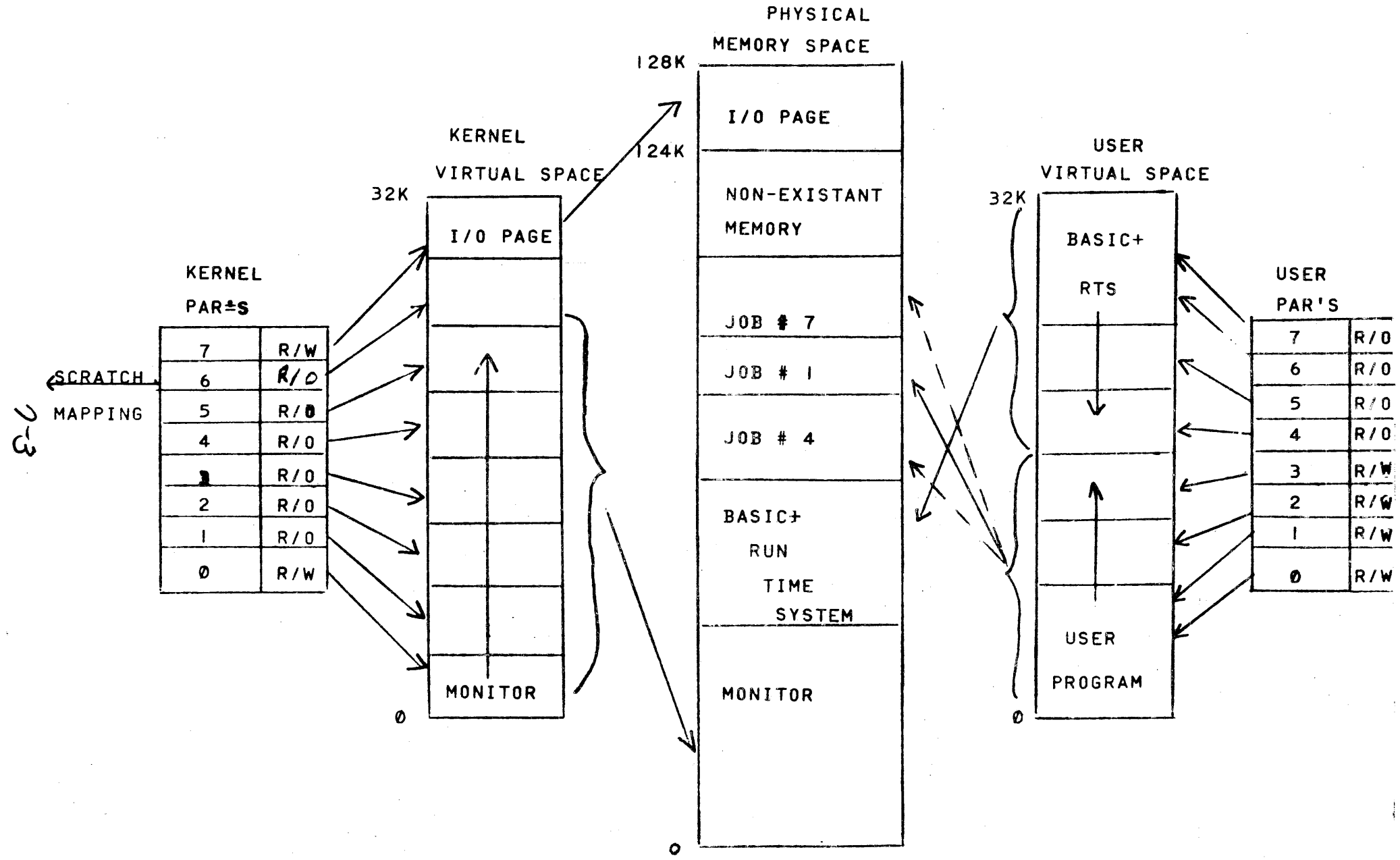
REFLECTED IN CIL

6. OPTION: START
7. BUILD SYSTEM LIBRARY
8. CREATE USER ACCTS.

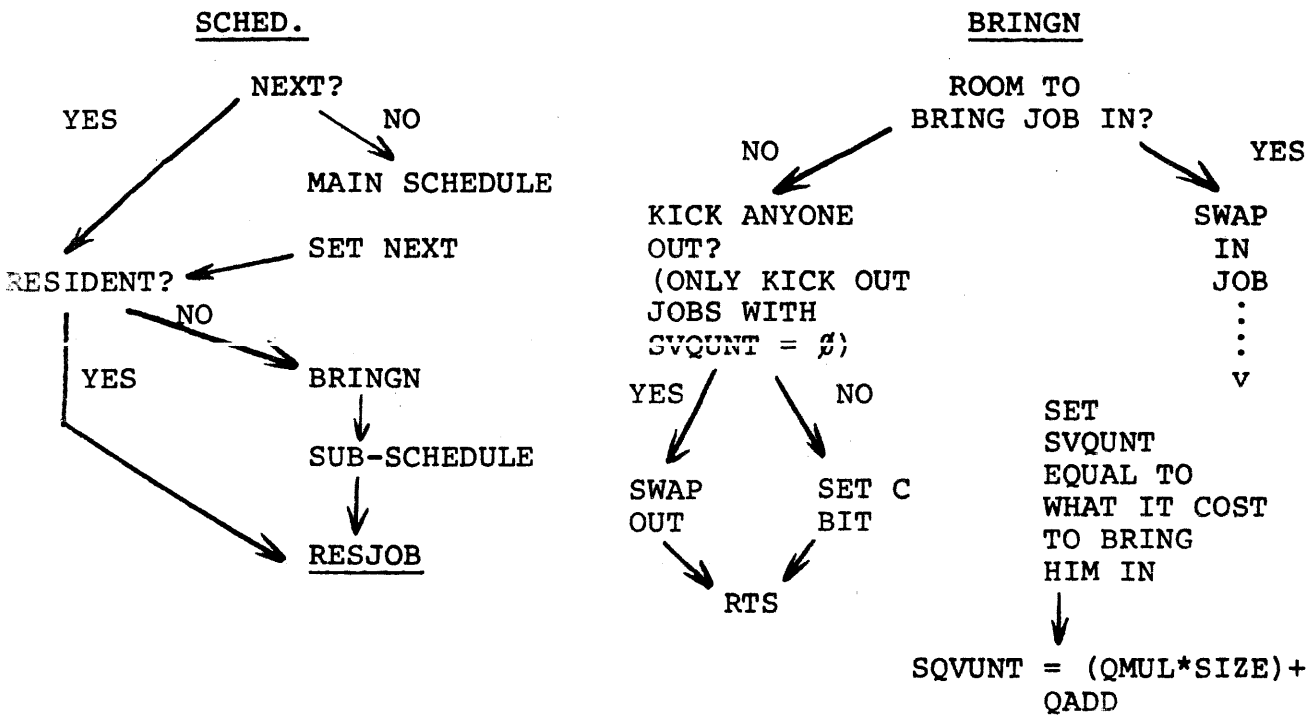
LIBRARY TAPES


USER
JOB #1 (virtual)
JOB #1
BASIC+
74
ONLY ONE MAPPING AT A TIME.
JOB #2
BASIC+
4
PHYSICAL MEMORY
Ø
MON
read only
BASIC+
JOB#3?
JOB#1?
JOB#7?
JOB#2?
NON-EXISTENT
I/O
KERNEL virtual
Ø
MON
I/O
WHEN JOB IS ACTIVE MONITOR READS/WRITE ONLY THIS SPACE
USER virtual
Ø
SP
400
KEYWORD
JOBF
402
IOSTS
USER MODEL "FIRQB"
442
USER MODEL "XRB"
1000
AVAILABLE BUT NOT MAPPED
NOT SWAPPED

MEMORY ORGANIZATION



OPERATION OF THE SCHEDULER



RESJOB

START UP A RESIDENT JOB

IF: 1) TIMED OUT [BURST EXHAUSTED]
2) DISK STALL
3) FIP STALL

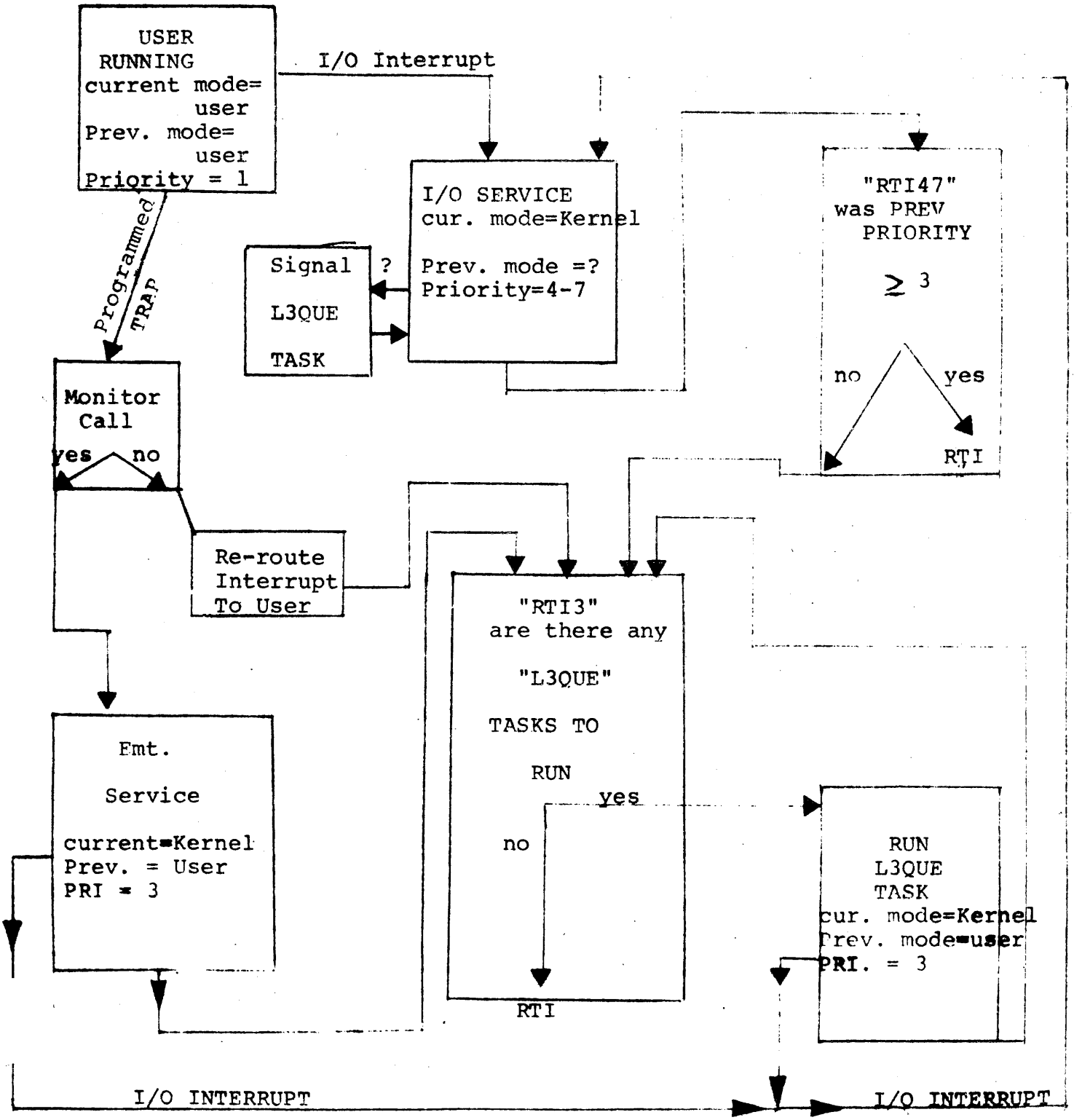
THEN: SVQUNT ← (SVQUNT)-(# TICKS RUN THIS RUN)

IF SVQUNT<Ø THEN SVQUNT = Ø

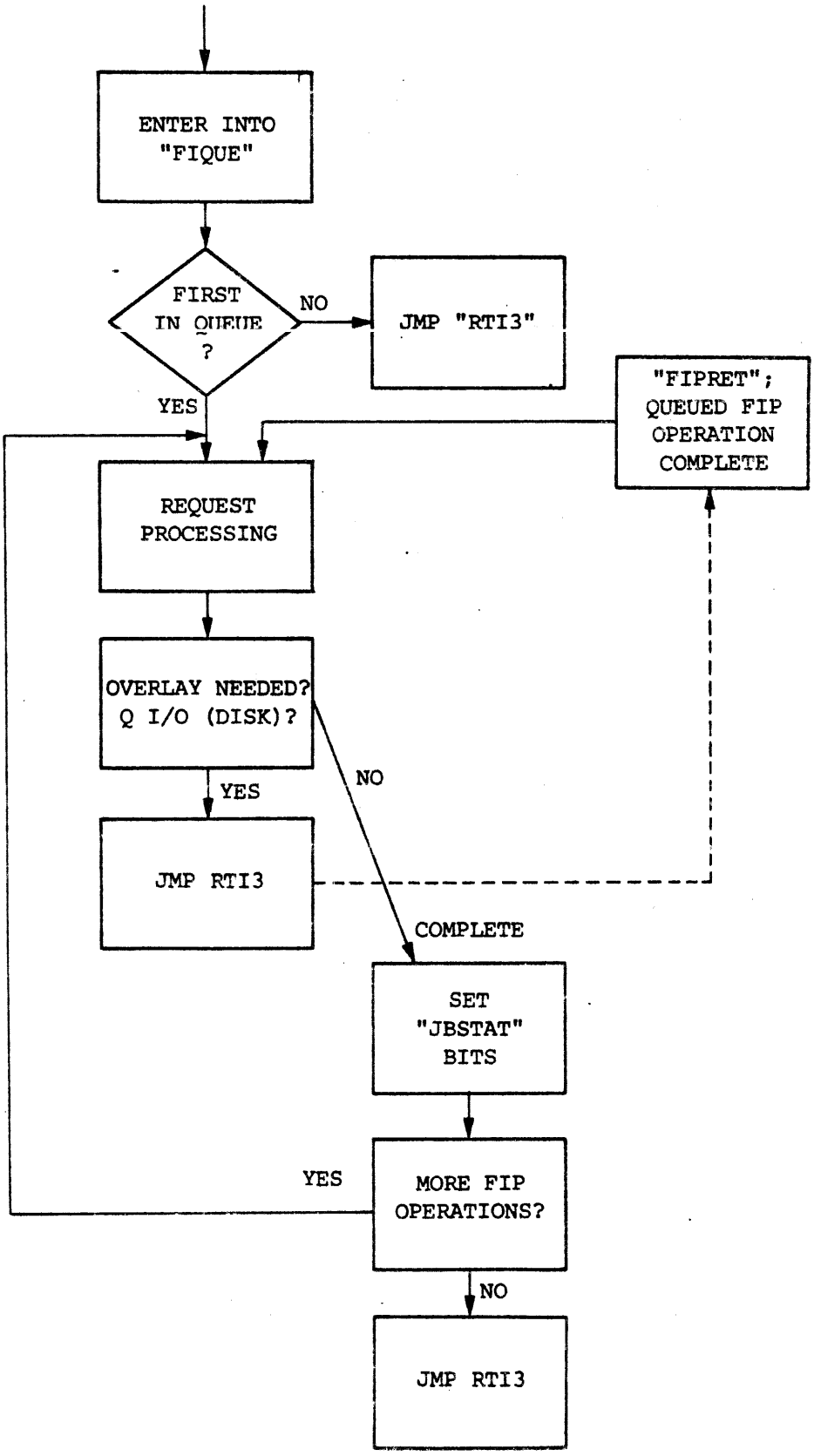
IF STALL WAS NOT DISK OR FIP THEN SVQUNT = Ø

"SVQUNT" IS THE GUARANTEED RESIDENCY
QUANTAM (RUN BEFORE SWAPPED) IF COMPUTE BOUND.
SVQUNT = (QMUL * SIZE) + QADD
[2] ← DEFAULT SETTINGS → [4]

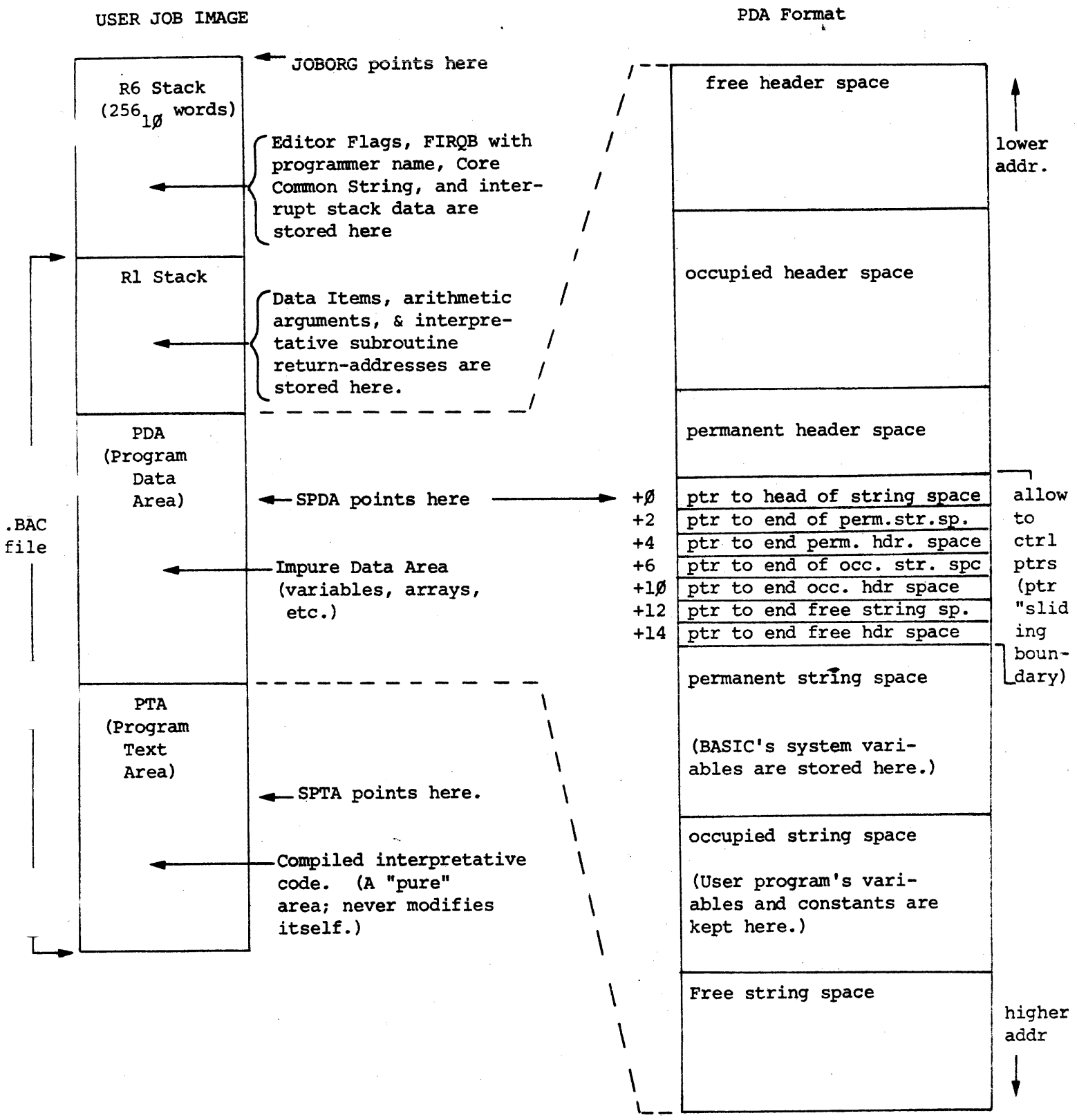
INTERRUPT HANDLING



"FIP" (FILE PROCESSOR) OPERATION

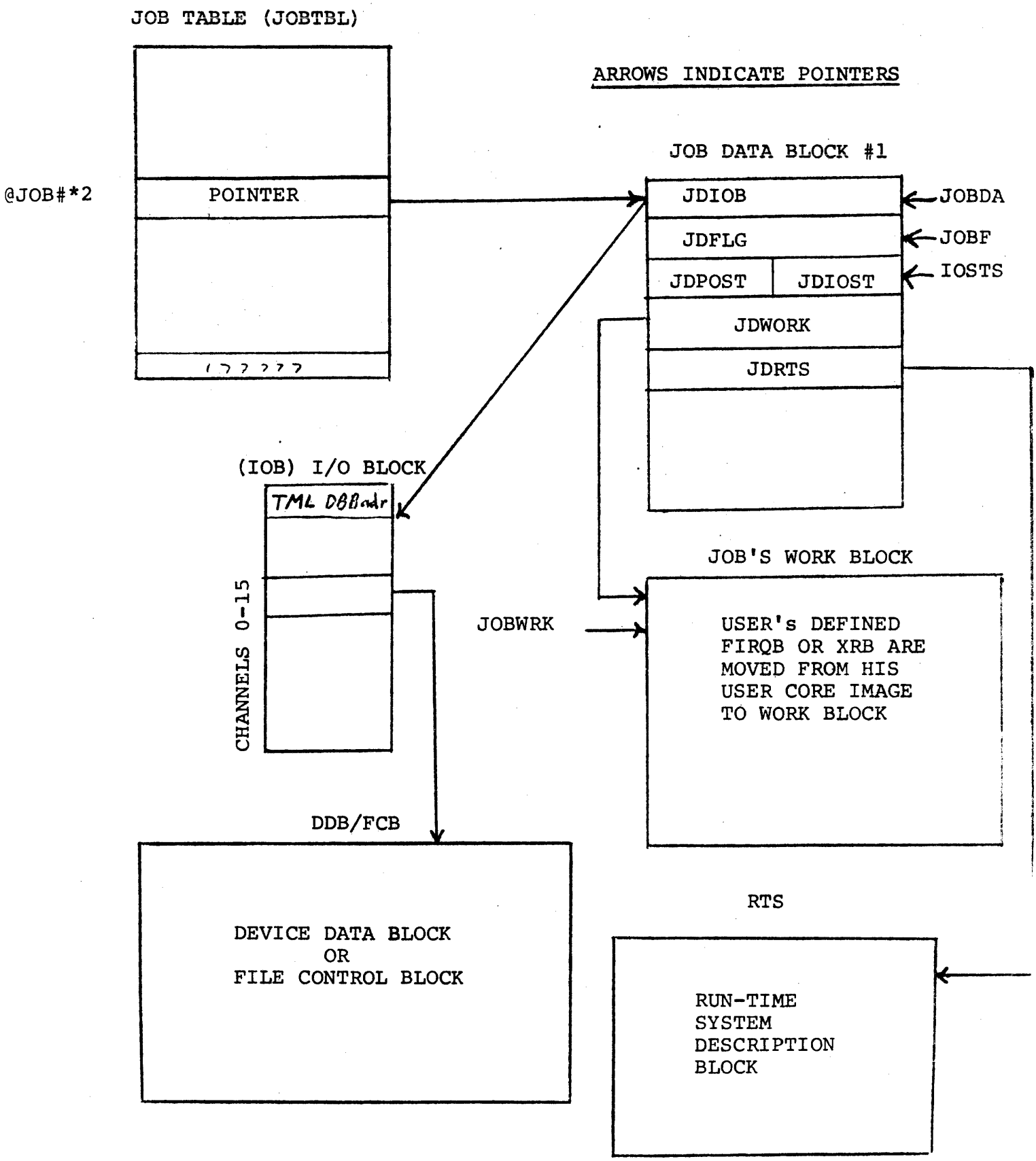


JOB STRUCTURE


USER JOB IMAGE
PDA Format
JOBORG points here
R6 Stack
(256 10 words)
Editor Flags, FIRQB with programmer name, Core Common String, and interrupt stack data are stored here
R1 Stack
Data Items, arithmetic arguments, & interpretative subroutine return-addresses are stored here.
PDA
(Program
Data
Area)
SPDA points here
Impure Data Area
(variables, arrays,
etc.)
PTA
(Program
Text
Area)
SPTA points here.
Compiled interpretative code. (A "pure" area; never modifies itself.)
.BAC
file
free header space
lower
addr.
occupied header space
permanent header space
+Ø ptr to head of string space
+2 ptr to end of perm.str.sp.
+4 ptr to end perm. hdr. space
+6 ptr to end of occ. str. spc
+1Ø ptr to end occ. hdr space
+12 ptr to end free string sp.
+14 ptr to end free hdr space
allow to ctrl ptrs (ptr "sliding boundary)
permanent string space
(BASIC's system variables are stored here.)
occupied string space
(User program's variables and constants are kept here.)
Free string space
higher
addr

RC	LA	TR	ED	RX	MA	TI	RC	PU	MX	PA	VE	BA
				MSG	ATH PACKAGE	IME	ECORD IO	RINT USING	ATRICES	ATCH SPACE	EPOINTER	SIC+ ODT
FT, RC, EC, SC, SU	Lexical Analyzer	Translator	Editor									

Optional


JOB TABLE (JOBTBL)
ARROWS INDICATE POINTERS
@JOB#*2
POINTER
JOB DATA BLOCK #1
JDIOB
JOBDA
JDFLG
JOBF
JDPOST
JDIOST
IOSTS
JDWORK
JDRTS
(IOB) I/O BLOCK
TML DBBadr
CHANNELS 0-15
JOB'S WORK BLOCK
JOBWRK
USER's DEFINED
FIRQB OR XRB ARE
MOVED FROM HIS
USER CORE IMAGE
TO WORK BLOCK
DDB/FCB
DEVICE DATA BLOCK
OR
FILE CONTROL BLOCK
RTS
RUN-TIME
SYSTEM
DESCRIPTION
BLOCK

CORE MAP

ADDR	IDENT	CONTENTS
0		BR JMPTOØ
2		PR7
4		FTLØØ4 (Time Out)
6		PR7
10		FTLØ1Ø (Illegal & reserved)
12		PR7
14		ODTØØ Address (BPT)
16		PR7
20		BUFØØØ (IOT)
22		PR7
24		POWERF (PWR Fail)
26		PR7
30		EMTØØ (EMT)
32		PR7
34		TRAPØØ (TRAP)
36		PR3
40	RELOAD:	JMP @ (PC)+
42		RELOAD
44	IDATE:	Initial Date
46	ITIME:	Initial Start Time
50		BR RELOAD
52		BR XCRASH
54	AUTORH:	HALT
56		BR RELOAD
60		TTI INT Addr
62		PR4
64		TTO INT Addr
66		PR4
70		PTRINT
72		PR4
74		PTPINT
76		PR4
100		CLOK (KW11-L)
102		PR6
104		CLOK
106		PR6 (KW11-P)

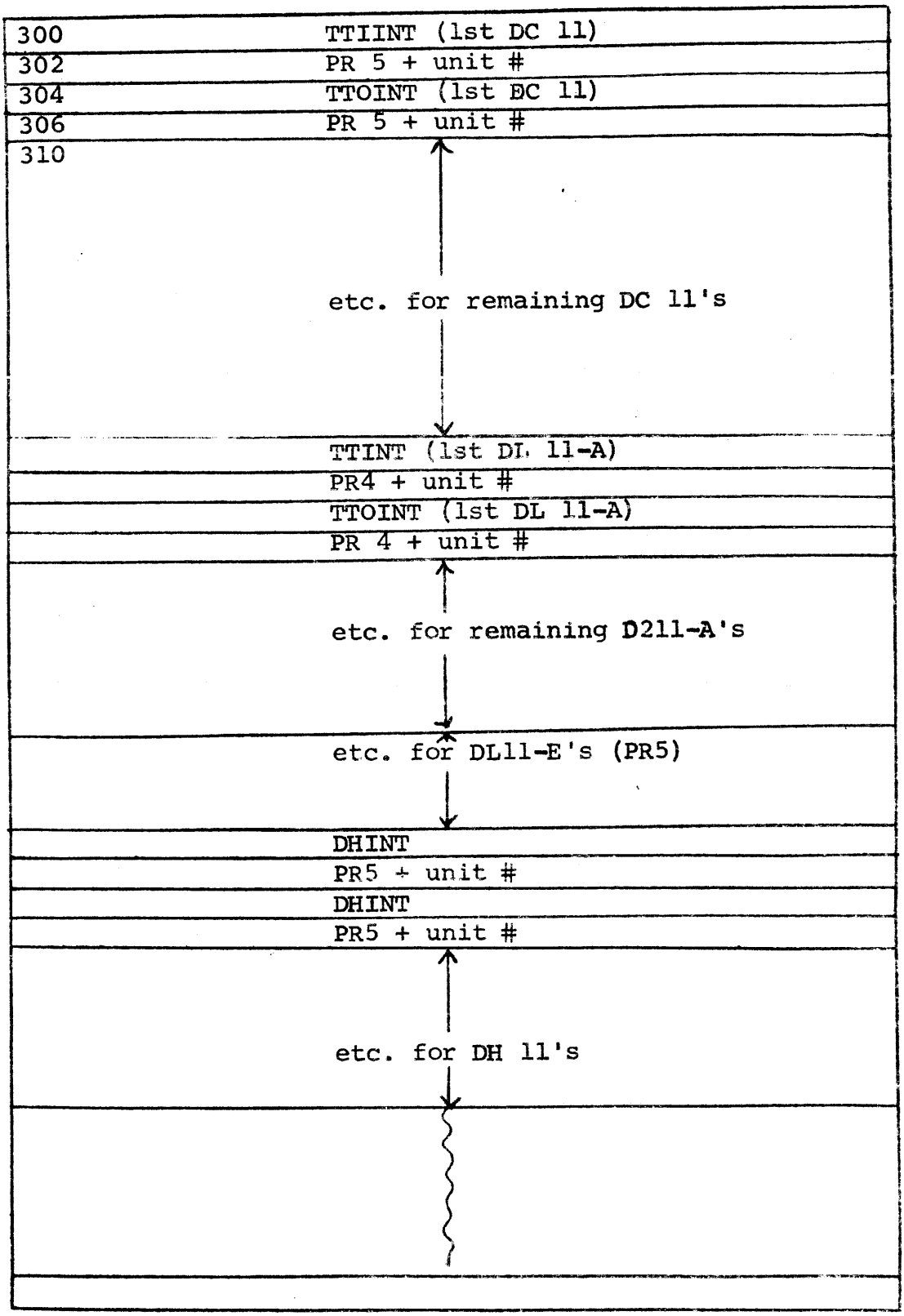
Monitor Core Map 1

ADDR	IDENT	CONTENTS
110	JMPTOØ	JMP FTLØØØ
112		
114		FTL114 Parity)
116		PR7
120		
122		
124		
126		
130		
132		
134		
136		
140		
142		
144		
146		
150		
152		
154		
156		
160		
162		
164		
166		
170		
172		
174		
176		
200		(LP11)
202		
204		(RF11)
206		
210		(RC11)
212		
214		(TC11)
216		

Monitor Core Map 2

ADDR	IDENT	CONTENTS
220		(RK11)
222		
224		(TM11)
226		
230		(CD11)
232		
234		WRKCTL (Pointer to Stats Pointer)
236		
240		Program Int
242		
244		MONFIN (Floating Point Error)
246		PR7
250		FTL 25Ø (Segmentation)
252		PR7
254		(RP11)
256		
260		
262		
264	XCRASH	JMP XXCRAS
266		
270		
272		
274		
276		
300		(start of floating vectors)
302		
304		
306		
310		
312		
314		
316		
320		
322		
324		
326		

Monitor Core Map 3



Monitor Core Map 4

(See Reliability and Test Manual)

ADDR	IDENT	CONTENTS
1000	DATE:	Internal Form
1002	TIME:	Internal Form
1004		Timsec/Timtic
1006	JOB:	Next/Job Now Running
1010	JOBDA:	Job ADR
1012	JOBF:	JOBFLG ADR
1014	IOSTS:	JDIOST ADR
1016	JOBWRK:	Work Block ADR
1020	JOBTIM:	TIME Left
1022	JOBQNT:	
1024	PARTBL	16 Possible Parity
1026		CSR Addresses
1030		
1032		
1034		
1036		
1040		
1042		
1044		
1046		
1050		
1052		
1054		
1056		
1060		
1062		
1064		
1066	PARRNG	16 Possible Ranges
1070		of CSR Addresses
1072		
1074		
1076		
1100		
1102		
1104		
1106		

Monitor Core Map 5

ADDR	IDENT	CONTENTS
1110		
1112		
1114		
1116		
1120		
1122		
1124		
1126		
1130	PARZER	Non-Error Parity GSR Sample
1132	PARMIN	Error Parity CSR Sample
1134	PARDSP	Exit Dispatch Address
1136	PARBAD	
1140		
1142		
1144		
1146		
1150		
1152	FTL 114	Parity Error Code
1154		Bad Log is checked to see if soft error
		also user/kernel mode
		check - parity error in kernel mode is
		fatal
1432	PARØØØ	

Monitor Core Map 6

ADDR	IDENT	CONTENTS
		System Stack Area
		68. words
2210	SYSTAK	
		8 wds User Area
		FIP's Stack Area
		68 words
2440	FISTAK	
	TTYDDB	TTY DDB's
	DTADDB	Dectape DDB's
	LPTDDB	Line Pointer DDB's
	PTRDDB	Paper Tape Reader DDB's
	PTPDDB	Paper Tape Punch DDB
	CDRDDB	Card Reader DDB's
	MTADDB	Magtape DDB's
	XXXDDB	
	YYYDDB	
	ZZZDDB	
	RTSDFT	Default Runtime System Description
	FREES1	Start of Small Buffer Pool
	FREEB1	Start of Big Buffer Pool
	FIPBUF	Non-Resident Buffer

Monitor Core Map 7

ADDR	IDENT	CONTENTS
	FIBUF	Directory Buffer
	FIBMAP	Map Start
	FIBENT	Map Entries Start
	SATBUF	SAT Buffer
	CORTBL	Core Management
	JOBTBL	Job Data Adr's
		1 word/Possible Job
	JBSTAT	Job Status Table
	JBWAIT	Job I/O Wait Condition
	TTYCLK	TTY Timer Table
	JOBCLK	Job Timer Table
	DEVCLK	Hung Device Timing
	SVQUNT	JOB's Saved Quantum
	DSDONQ	Pointer to Start of Disk Done Queue
	DSKQUF	RF11/RC11 Disk Queue Start → OSQ
	DSKQDK	RK11 Disk Queue Start
	DSKQDP	RP11 Disk Queue Start → OSQ
	SATPTR	One per Disk Unit. Points to
		where in the devices SAT to start
		looking for free blocks.

PCLOFF PCLSRT

DRVSOK DRVSOF DRVBOF

Monitor Core Map 8

ADDR	IDENT	CONTENTS
	SATSIZ	SAT BYTE Size List
	UNTCLU	Cluster Size List
	SATSTL	
	SATSTM	SAT Starting Segment Lists
	SATCTL	Counts of Free Sectors in SAT Lists
	SATCTM	Counts of Free Sectors in SAT Lists
	LOGNAH	Device Done Table
	LOGEND	
	CLURAT	Table of cluster Ratios Pack/Device
	UNTCNT	Pack status and Open Count
		PUT, Mounted, NFS, locked
	DEVPTR	Points to slots in UNTCNT or
		DEVTBL appropriate to each device
	DEVTBL	
	TTYDEV	
	DEVTBE	
	DTADEV	
	LPTDEV	
	FTRDEV	
	PTPDEV	
	CORDEV	
	MTADEV	
	XXXDEV	
	YYYDEV	
	ZZZDEV	
	DEVEND	

Monitor Core Map 10

ADDR	IDENT	CONTENTS
	NAMTBL	Program Name Table
	TTYCTL	Terminal Control
		Receivers table
	MESTBL	Message Sending Table
	ERRCTL	Error Control
	DTACTL	Dectape
	MTACTL	Magtape
*	MONCTL	Monitor
	FIP CTL	File Processor
	CDR CTL	Card Reader
	RKREST	RK Dirty Area
	RPREST	RP Dirty Area
	XXXCTL	
	YYYCTL	
	ZZZCTL	

Monitor Core Map 11

*See 11.1 and 11.2.

ADDR	IDENT	CONTENTS
	Z.ISR	2780 ISA
	Z.TAP	2780 TAP
	Z.SCIP	2780 SCIP
	Z.USER	2780 User
	CLOK	
	QSTATB	
	WRKCTL	
		Disk Stat Ptr
		Job Stat Ptr
		Queue Stat Ptr
	DSTATB	
	JSTATB	
	POWERS	Power Fail Data Space
	PATCH	Patch Space
End of Read/Write Area		
Bgn of Read Only Area		
	TBL	
	ERRTBL	[illegible]
	FIPTBL	
	SERTBL	I/O Svc TBL
	SIZTBL	I/O LineLength Tbl
	L3QTBL	
	EMTTBL	EMT Dispatch Tbl

Monitor Core Map 12

12.2

Addr	Ident	Contents
	OPNTBL	
	CLSTBL	
	MSKTBL	
	SPCTBL	
	NULJOB	
	MTA	
	UPDATE	
	ERNLOG	
	MONFSV	
	MONFRE	
	MONFIN	
	MONFER	
	MCFAIL	JUMP XXXCMS
	DISKGO	
	YYY	
	TTY	
	XXX	
	CDR	
	DCDCDR	
	LPT	
	DTA	
	MTACHK	

Monitor Core Map 13

	IDENT	CONTENTS
	MON	
	FIP	
	DUF	
	DSK	
	WIN	
	EXT	
	RFDSK	
	CPTDSK	
	CCLDSK	
	CPN	
	OP1	
	OP2	
	op3	
	OP4	
	OP5	
	UUO	
	SND	
	MES	

Monitor Core Map 14

MONITOR ROUTINE NARRATION

DIGITAL EQUIPMENT CORPORATION

Notes on Small & Big Buffers

The system's Small Buffers are each 16_{10} words in length.

The system's Small Buffers are used for storing job and I/O parameters and as I/O data-transfer buffers for character-oriented devices.

Every active job uses two Small Buffers: one as Job Data Block and the other as Job I/O Block.

When active, each KB, the PR, and the PP should best have 5-6 Small Buffers each, and the LP should have 10 Small Buffers. These serve as data-transfer buffers.

(Note -- The CR is record-oriented and has its own permanent Input buffer at CDRBUF. DT is file-oriented and uses a Big system Buffer for its I/O. MagTape and disk use no intermediate buffers but transfer directly into/from the user's job area.)

Every open disk or DT file requires one Small Buffer as FCB (File Control Block).

Every File-Processor request uses a Small Buffer for the FIRQB (File Request Queue Block).

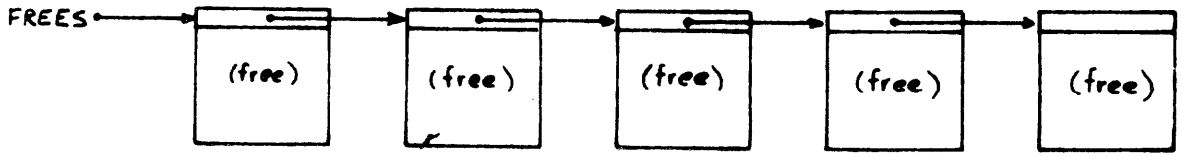
Every disk-transfer request (except those made by the File Processor and the Swap manager) uses a Small Buffer for the Disk Parameter Block.

When estimating the number of Small Buffers which a system should be configured to have, the totals derived from the above facts should be moderated by the consideration that not all jobs will be active at the same time nor will all devices and files be open at the same time.

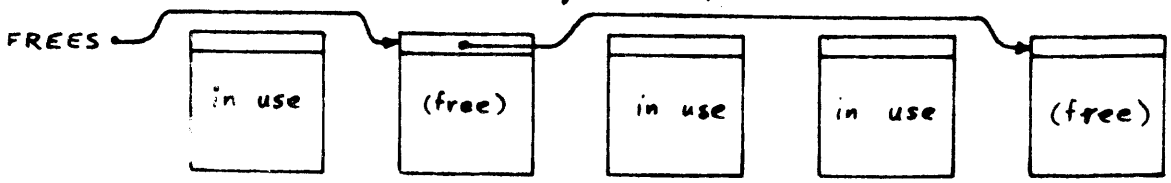
The system's Small Buffer pool begins at FRESML.

When free (not in use), the Small Buffers are linked together in a queue. The origin of the queue (i.e. pointer to the first free Small Buffer in the chain) is at FREES

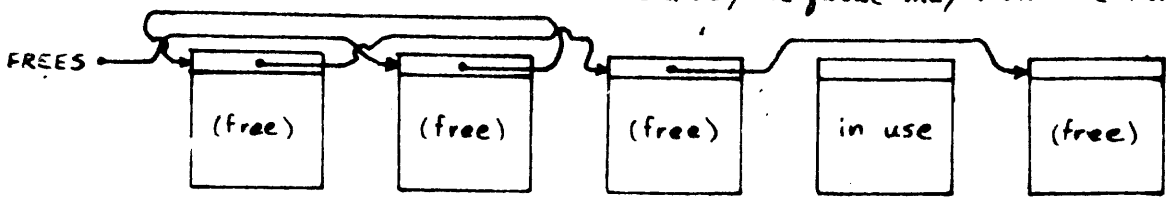
At initialization (before any buffer is in use) queue looks like this:



After the system has been running, the queue may look like this:



After a Small Buffer have been returned to free status, the queue may look like this:



Reading and writing is done by Monitor Code. The part of the user space with I/O Buffer is mapped into Monitor by using scratch R6 of PAR5 for disk & MAGTAPE

No device can own more than 25% of the system's small buffers. If less than 20% of the system's small buffers are left, they can't user more than a particular device's quota.

I/O Drivers us Monitor subroutines to get data. (character I/O)

Disk and MAGtape is direct from/to user area.

DECtape I/O to Monitor and from Monitor to User is via Big Buffers.

I/O stall occurs when small buffer limit is reached by device
→I/O REDO also when disk xfer can't be transferred immediately

User Buffer I/O area in user image is as large as the user can make it. Max 14K.

R6 of KISAR is used as scratch register and is used to map user space I/O Buffer Area into Monitor Space for DT & LP

Location & BYTE count in XRB are used by Monitor to set up all I/O Transfers.

XRB's are also used for EMT's

Firqb's are used for opening closing & deleting files.

3 Levels of main scheduling

To reflect 1 Interactive Jobs
2 ↑ C
3 Sysfunction to increase quantun for this swap.

So the Job Table is scanned 3 times before running NULL JOB

Also have 4 possible swap files which it set up correctly will keep most active jobs on fastest swapping device. The slots in the swap files are fixed to swap MAX size each.

Highest priority jobs get lowest swap slot numbers which are allocated to fastest swapping device.

There are 4 swap files & swap MAP in core for each File because the RS04 is a UNIT addressable device.

SWS NOTES

RSTS/E

	SCHEDULER		INFORMATION
1006	JOB	Low Core Area	Current running job (locked)
1007	NEXT	Low Core Area	Job to run A.S.A.P.
	SUBJOB		Job who was sub-scheduled and running because he is in core and runnable and the job that really should be running is being brought in to core.
	NXTRES		Job to be made resident next.
	JOBPTR	Monitor Control Area	Is Round-Robin pointer to job number(s) in job table
	CORPTR	Monitor Control Area	Is Round-Robin core table sub-schedule pointer (next spot to sub-schedule)
	CORFOR	Monitor Control Area	Is Round-Robin core table force pointer used to determine who should be forced out of core to make room for a job that needs residence. (Job number for Force Out)

SCHEDULING CRITERIA

1) Must be runnable.

2) Highest priority.

3) Highest quantum (runnable time period)

1 Overview of Monitor Functions

The RSTS/E monitor performs the following range of functions:-

- It manages timeshared jobs by means of a round-robin scheduler which selects runnable jobs in order of priority.
- It manages the memory required by swapping jobs in and out of main storage. 'Hole' selection for a job is on a best fit basis.
- It performs I/O and other service functions for timesharing jobs. The interface is by means of EMTs, XRBs and FIRQBs (Transfer Control Blocks, File Request Queue Blocks)
- It performs memory mapping functions for communications and transfers between modes
- It dispatches I/O requests from the user, using buffers to enable flexible control of tasks not actually involved in data transfers
- Users can handle their own traps. RSTS/E fully supports that facility by resuming the job at an address specified in the job image at known virtual addresses.
- It will run impure or pure code jobs
- It acts as a communications and scheduling system between the following:-
  - USER/Device handlers
  - Device handlers/RSTS MONITOR
  - RSTS MONITOR/USER
  - User/file management software
- It handles I/O errors at device levels and supports a log for such errors.

The modules described in this section form the basis of these functions and define the essential mechanisms for RSTS/E.

2.5 User/Monitor Interface

This section covers the parameter and data passing interface between a user job and the monitor. There are two main devices:

Transfer Control Block (XRB)

XRB is used by the user to initiate an I/O Request and for Monitor/User data requests.

File Request Queue Block (FIRQB)

Each block is 40_8 bytes long (small buffer). The FIRQB holds functions in slot FQFUN. A subset of these functions is available to the RSTS/E user, and the rest are used by the monitor itself. When a FIRQB is set up, EMT CALFIP (EMT Ø) is issued which is dispatched to FIB. FQFUN then defines the function to be performed either by resident or overlaid code. Only relevant parameters are needed.

2.6 L3QUE (Intra-Monitor Scheduling Flags)

This word is checked before a return is made from the monitor to the user. Also device handlers and high priority functions (levels 4-7) return to the monitor (RTI47) where L3QUE bits are checked before returning to the user. If a bit is set in L3QUE a corresponding entry in the table of handlers (L3QTBL) is used for dispatch to the appropriate level 3 (monitor) routine.

"L3Q" LEVEL THREE QUEUE TASKS

QTIMER	once per second - check for
	Hung TTY's
	Sleeping JOBS
	Exhausted "KB" waits
	Hung Devices
QFIP	"FIP" I/O Completions
	i.e. continue in "FIP"
QSWAP	SWAP Completion
	i.e. update CORTBL "SWPRET"
QFILE	Disk I/O Service Completion
QDTSYM, QDDCON, QDACON	DECTAPE
QUECDR	Card Done "CR:"
QBUFSM	Small Buffer ALERTER sets "JSNUL"
	;small buffers are now available
QMTADN, QMTACN	MAGTAPE
QSCHED	Scheduler
QBRING	Bring JOB INTO Memory
QFORCE	Force a JOB out of Memory
QBUF	Big Buffer Available

2.7 EMTs, Their Identifiers and Dispatch Table

EMTs are used in order to enter the monitor from a user job to effect the same functions normally handled by the monitor.

See the list of EMTs and the EMT dispatch table

2.8 TRAPs and Their Identifiers

Synchronous traps (in this case TRAP(1Ø44ØØ) plus low-byte code) are dispatched back to the handler in the RTS defined by P.TRAP. The handler in the monitor is TRAPØØ which simply returns to RTS at the address in P.TRAP.

Many of the traps are used by BASIC-PLUS for error handling. The location of TRAPØØ is in Loc 34. P.TRAP is maintained in the Pure Code Area Critical Pointers table.

2.9 Buffer Requests (IOTs)

An IOT (ØØØØØ4) is used to get and return, big (256. word) and small (16. word) buffers from and to the appropriate buffer pools. The parameters are passed immediately following the IOT.

2.1Ø The Monitor Control Area

Apart from the low core variables, the monitor requires many data cells and pointers for its functioning. These are contained in the Monitor Control Area. It contains swapping parameters, Job Mapping, Buffer, and Job Status Information. The control area is assembled in CSECT MONCTL.

EMT SERVICE

1 Simple

A) Get Data or perform other operation

B) JMP RT13

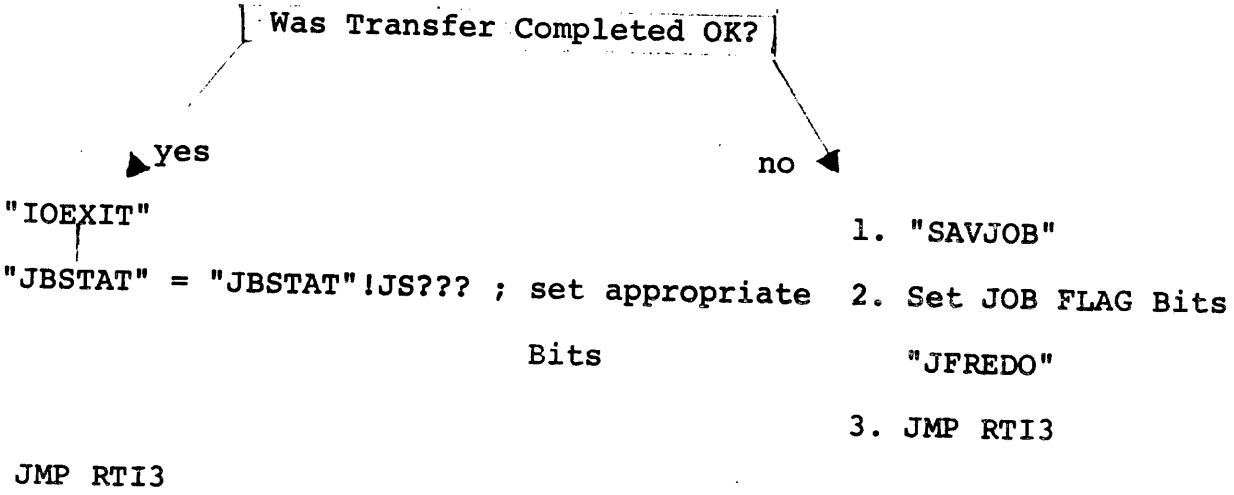
2 I/O Request

A) Character I/O (KB, LP, CR, PR, PP)

set "JBSTAT" = ø

set "JBWAIT" = JS???

DO TRANSFER



B) NPR TRANSFER (DT, MT, DISK)

JBWAIT = JS???

JBSTAT = ø

1. Set up
2. "SAVJOB"
3. Position Device and/or wait in queue
3. "FNDJOB" ; find JOB/bring into core
4. Do transfer
5. Set JOB status bits

"JBSTAT" = "JBSTAT"!JB???

6. JMP RTI3

ob Management and Scheduling

The RSTS/E monitor software concerned with job management and scheduling falls into the following routines and functions:-

- Main Scheduling routine (SCHED)
- CORTBL search for a job in core (CORE)
- Find and lock a job into core or initiate a swap (FNDJOB)
- Bring a job into core (BRINGN)
- Start (more) swaps if possible (SWAPP, SWAPIT)
- Process swap completions (SWPRET)
- Start a resident job and synchronise flags (RESJOB)
- Force a job to dump itself (FORCEQ)
- Kill a job (KILL)
- Dump the current job (SAVJOB)
- The Null Job (NULJOB)

These routines are described in the following sections.

3.1 Main Scheduling Routine (SCHED)

In RSTS/E scheduling is based on

- a round-robin scan of jobs (via JBSTAT and JBWAIT tables)

and

- a priority mechanism for the selection of the next job run

If the next selected job is both runnable and resident, it becomes the next job to run. If it is runnable but not resident, an attempt is made to get it into core. This may mean swapping out resident jobs in order to make room for the job in question. Thus, the scheduling of jobs may entail the initiation of a number of swap-outs before the required job can be swapped in and run. While jobs are being swapped out and the scheduled job is swapped in, a search is made for a resident and runnable job which is then run as a sub-scheduled job. If no such job can be found, NULJOB is run until a swap is completed, which on completion sets QSCHED IN L3QUE to recall the scheduler.

3.2 CORTBL Search for a Job (CORE)

This routine is called via a JSR PC, CORE with register RØ holding the number (*2) of the job which is the object of the CORTBL search. The CORTBL is searched word by word until the index in RØ is found in a low-byte. If the whole of CORTBL is searched without finding the job, the N-condition bit is set true and CORE returns. If the job is found, CORE returns with the N-bit = Ø and R2 is holding the address of the highbyte of the word corresponding to the start of the job in core.

3.3 Find and Lock a Job into Core or Initiate a Swap (FNDJOB)

This routine first of all calls routine CORE to find the job passed on in RØ as a parameter (job number *2). If the return is negative (the job is not in core), FNDJOB calls BRINGN to initiate the swapping in of the job. If the job is in core, the high byte (in R2) of the job's corresponding CORTBL word is tested, and if non-zero, routine BRINGN is called to initiate the swap-in, if the swap-in bit is not already set, then FNDJOB just exists to RTI3, where the L3QUE flag word is set with the flags passed on in the FNDJOB call in R3.

If the CORTBL high-byte on return from CORE is zero and the job is resident then COROFF is called to compute the real address (mapped into two words, MAPHI and MARLOW which are stored in the Monitor Control Area). On return from COROFF, R2 still points to the CORTBL entry (high byte) for the start of the job, and the lock bit(LCK) is set in the high byte for all CORTBL words holding the corresponding job slot. FNDJOB then returns (RTS PC).

3.4 Bring a Job into Core BRINGN

BRINGN is called via JSR PC, BRINGN with RØ by holding the job number (*2) for the job to be brought (swapped) into core and R3 holds the L3QUE bits to be set when the job is made resident.

BRINGN initiates the swapping out of jobs to make room for the swapped-in job (in RØ) or, if this is not necessary, finds a hole in core (smallest one that will accomodate job) and initiates a swap into the hole.

An entry point in BRINGN is BRINGQ which is the entry point on dispatch from L3QTBL when the QBRING bit is set in L3QUE.

3.5 Start More Swaps Going If Possible (SWAPP, SWAPIT)

This routine logically falls into two phases;

1. A search through CORTBL for jobs whose SWP bit is true,

2. The setting up of a DSQ (disk request queue block) and initiation of the swap for the job (IN or OUT depending on the IN or OUT bits). This is followed by a return to phase 1 to seek for more jobs to swap.

This routine is a job housekeeping utility called on swap completions (SWPRET) and when jobs are brought into core by BRINGN.

The CORTBL search has three entry points:-

1. SWAPP - which pushes#RT13 on the stack as the return address.
2. SWAP - immediately following SWAPP, calls REGSAV to save the caller's registers.
3. SWAPLP - after SWAP. As the name implies it is a loop return from SWAPIT (phase 2), and also is an entry point from BRINGN which has already saved the registers.

Since SWAP references many data items, these are explained in the following subsection.

3.6 Process Swap Completions (SWPRET)

Initially SWPRET checks SWDONQ, the queue of DSQ's of completed swaps. If the queue is empty then SWPRET calls SWAPP which seeks out and processes swaps. If a DSQ for a swap return is on the SWDONQ queue, then it is accessed to get the job index for the job from DSQJOB. If the job has been swapped out, then this slot is zero, otherwise it holds the index of the swapped in job.

If the slot is non-zero, a check is made on DSQERR in the DSQ. If an error is indicated, then the JFSWRR bit is set if the job's JDFLG slot is set true. From the DSQJOB, the job index allows access to JOBTBL to get the address of the job's JDB. The JDSIZØ (current size of job in K) is set equal to JDSIZ1, the swapped-in size.

DSQMSC slot of the returned DSQ holds the high byte of the job's CORTBL entry. If the job has been swapped-in, this is the current CORTBL entry, whereas if swapped-out, it holds the previous entry. From this byte, the type of swap (IN or OUT) can be ascertained from the appropriate bits. If the swap was 'OUT', SWAPF is decremented. SWAPF is a count of pending swaps OUT. Then, whether the swap was IN or OUT, the high bytes for the job's CORTBL entries are cleared. If the swap was out, then the low bytes (job index) are also cleared.

CLRSWP is called to deallocate swapslots after swaps in.

If the DSQ was SWAPAR (the fixed swap DSQ) then SWPRET branches back to the beginning to process further swap returns. If the DSQ

was a small buffer from the pool, then IOT (BUFFER RETSML) is effected, returning the DSQ to the small buffer pool. Then a loop is made back to the beginning.

3.7 (RESJOB) Start Resident Job and Synchronise Flags

RESJOB is called with R3 holding the job index (job number *2). This is used to enter JOBTBL. The low core job management database (JOBDA, JOBF, IOSTS, JOBWRK, and JOBTIM) is accessed. The time used by the job (JOBTIM) is cleared.

Routine USRMAP is called to map the job on the user Page Address Registers and set the stack pointer to SYSTAK. If the JFKILL bit is set in the job's JDFLG slot of the job's JDB, then a jump is made to KILL to kill the job.

If neither JFSWRR nor JFIRST bits in JDFLG are set, then the job is to continue where it left off. Therefore RESJOB branches to the 'return-to-job' code. Otherwise, RESJOB, enters an area of code in which the appropriate re-entry point to the RTS is found.

If JFSWRR (swap error) is true, RESJOB branches to 11$, where the re-entry point is set to P.BAD, the RTS bad-job entry point. DIOST (I/O status slot in JDB) is set with the B.SWAP (bad-swap) bit. If the job has caused a stack error, (from JFSTAK-bit in JDFLG), then B.STAK is also set true in JDIOST. After this the routine enters the general job return code (1Ø$).

In 2$ if the JFCRAS, JFRUN or JFSTRT bits in JDFLG are true then the return addresses are set to P.CRAS, P.RUN or P.START appropriately. The general job-return code is then entered (1Ø$). Even if none of these bits is set in JDFLG.

3.7 (RESJOB) Start Resident Job and Synchronise Flags

At 1Ø$, the JDFLG2 bits are cleared, and then routine MAPRTS is called to map the run-time system onto the Page Address Registers.

The user's PS is set up. This is pushed on the stack along with dummy registers (RØ-R5). The return address (PC) is then pushed on the stack. The job index is given to the user in the FIRQB at FQJOB.

If the return address is P.BAD (a bad job) then a branch is made to code which just returns to the RTS at the entry point. Otherwise the user's error flag work (KEY) is cleared, and the old register values restored along with resetting the floating point processor (FPP) if necessary. The stack pointer is then set up.

The QSCHED bit (schedule) is cleared from L3QUE to indicate that the scheduling request has been processed. Then RESJOB returns by jumping to RTI3, the general level 3 return.

3.9 Kill a job (KILL)

KILL is jumped to from RESJOB if the JFKILL bit is set in the JFLG2 byte of the resident job's JDB.

- Initially, KILL clears the JBSTAT word for the job so that it is non-runnable, and the JSFIP (waiting for FIP completion) bit is set true in the job's JBWAIT word. Setting the JBSTAT word to zero makes the job non-runnable since the scheduler runs a job when the logical 'and' of its JBSTAT and JBWAIT words is non-zero. The JSFIP bit is set true since killing involves a queued file service (FIP) request to clear out any pending IO requests... I/O rundown.

- If the file service request has already been made (JFKIL2 bit set in JDFLG) then KILL branches to the clearing up house-keeping at 1Ø$. (KILL immediately).

- If the JFKIL2 flag is not set, then it is set and JFHIBY and JFPRIV bits are set in the job's JDFLG.

  JFHIBY means that the job is in a critical state and JFPRIV means that it has permanent priveleges.

- The job's work block is accessed and used as a FIRQB in which is placed:-
  - the job's number (*2)
  - the clean-up function (UUOFQ)
  - the channel slot is set to +5 as a code for logging out the job.

- Routine UNLOCK is called to unlock the job from core. Routine SAVJNL is called to set up NULJOB to run and dump current job.

- KILL then jumps to FIPSYS in FIP which then processes the request defined by the FIRQB just created.

- At 1Ø$ (the immediate kill entry which is branched to within kill if JFKIL2 is set) the following routine actions are carried out:-

  - Priority is switched to 7 to prevent any interrupts
  - Mode is switched to Kernel mode
  - If the job is not already detached it is then detached (DDJBNO slot in terminal's DDB cleared)
  - CORE is called to find the job's entry in CORTBL
  - The job's entries in CORTBL are cleared
  - The job's JOBTBL entry is cleared
  - SAVJNL is called to set up null job and clear the current entries for the job
  - The IOT BUFFER is effected with parameter RETSML to return the JDB and WORK blocks for the job
  - The job count (JOBCNT) is decremented
  - and a jump is made to RTI3 for return and reschedule.

3.10 Dump the Current Job (SAVJOB)

This routine dumps the current job and optionally saves the job's quantum.

- At SAVJQX (an entry point prior to SAVJOB) a check is made on whether the current job is non-null
- Otherwise, At SAVJOB, if the current job is non-null, the current quantum is destroyed for the job (SVQUNT)
- If the current job is NULJOB, then SAVJOB branches to SAVJNL
- UNLOCK is now called to unlock the current job from core
- The JFCODE bit in the job's JDFLG work of its JDB (pointed to by JOBF) is tested
- If JFCODE is true then the job is running impure code (not BASIC-PLUS) then the JFFPP (save/restore floating point unit) bit in the job's JDFLG is cleared, otherwise the JFLOCK (lock job in core) bit is also cleared
- The user's key word (KEY) - flags to be set on re-residency or running - are moved into the job's JDFLG slot in its JDB
- The user's stack pointer is checked and if it has not overflowed, the current user's registers (which have already been pushed on the kernel stack) are popped onto the user's stack
- If the user's stack has overflowed into the stack save area, the JFSWRR is set in the job's JDFLG to signify a swap error (something wrong with the job)
- The JFSTAK (stack overflow) bit is set in the job's JDFLG2 byte of the JDB
- Then the user's registers (SP,RØ ---RS,PC,PS) are pushed on the user's stack

3.10 Dump the Current Job (SAVJOB) (Cont'd)

- Routine MONFSV is called to save FPP status if required
- Routine SAVTIM is called to save timing and core utilization information
- SAVJNL is then entered to set up the null job. The stack pointer is set to the system stack, the monitor data cells JOB, QUANT are cleared and bit QSCHED is set in L3QUE to schedule another job and then a jump is made to the caller's return address
- Routine SAVTIM is in the SAVJOB module
- SAVTIM computes the accumulative KCTs (Kilo-core ticks) from JOBTIM (CPU time) and JDSIZØ from the JDB (current size of job)
- If the job is detached, SAVTIM returns
- If not detached (by accessing terminal DDB from JOBDA slot in monitor data area and then looking at DDJBNO slot of DDB) then the time at which the job was attached to the terminal is found from DDTIME in the DDB and corrected for midnight if necessary. This is then added to the total in JDCON slot of the JDB. SAVTIM then returns.

3.11 The Null Job (NULJOB)

This job runs when no other user job is running. It displays a right to left movement of lights on the 11/45 data paths display.

4.0 Core and Memory Management Routines

4.1 Monitor core manager expander/shrinker (CORE)

This routine is dispatched to from EMTTBL as a result of a user EMT .CORE.

The caller's XRB holds the parameters of the call. The new core size requested (in K words) is passed at XRB+Ø, with the restriction that it is non-zero and less than or equal to .ORMAX.

If room is available for expansion or if the size is less than its current size, CORTBL is mapped accordingly. If the expansion cannot be carried out, then the job is swapped out and brought in again with the new size. On exit, if IOSTS is zero then the new size was effected, otherwise the new allocation was not carried out since the request was in error.

- CORE. first of all gets the new size from the user's XRB+Ø slot.
- IOSTS is set non-zero in anticipation of error.
- The job's JDB and RTS blocks are accessed (via JOBDA, and the JDRTS slot of the JDB respectively.
- Via the R.USIZ ENTRY in the RTS description Block, CORE. checks that the request is not bigger than the maximum for the job. If it is then CORE. again exits to RTI3.4 indicating an error in IOSTS.
- Otherwise IOSTS is cleared.
- Via JOB (job slot) the job;s slot is accessed and routine CORE is called to locate the job in CORTBL.

- The JDSIZ1 (new size) slot is set in the JDB to the requested value.
- If an expansion is required then CORE. branches to 3$.
- If the request is less than the present size, then CORE. enters 1$. If same 2$.
- At 1$, CORE. clears the extra CORTBL entries, and then enters 2$ where USRMAP is called to remap the user. and a test is made to see if changing RTS. If so, go to Res job entry SAUJQX If not, exit via RTI3.4.
- At 3$ (processing core expansions) CORE. checks to find whether there is free space to accommodate the new size. If there is then CORE. re-enters 2$ and again calls USRMAP to effect the new mapping and jumps to RTI3.4.
- If there is not enough room for expansion, CORE. gets the job's current size (JDSIZØ) and saves the current CORTBL pointer. Then it sets the SWP and LCK bits for all CORTBL entires of the job and branches to SWAPP to initiate the swap.

1.2 Map a job into user space USRMAP

The function of this routine is to set up the user Page Address Registers(PAR) to map a job.

- The job slot is obtained from JOB.
- Routine CORE is called to find the start of the job in CORTBL.
- On return it starts a loop which sets the LCK bits true in the CORTBL entries for the job.
- From this, the number of entries for the job are computed, so the job size is known.
- Since CORTBL maps 1 word for 1K words of store for all physical store, then the following computation sequence gets the entry for PARØ for the user:-
  - Let the offset in CORTBL for the job start (on a 1K boundary) = x. Then x = number of K (words) *2 (i.e. the byte address in K).
  - There are 16. blocks of 64. bytes in 1K bytes.
  - Hence, since PAR address in units of 64. byte, then x*16. gives the appropriate PAR entry.
- USRMAP, therefore, from the CORTBL entry computes the entry for PARØ. If the job exceeds 4K (words) then PAR1 is set equal to PARØ + 2ØØ (2ØØØØ = 4K) and so on.
- Also, the associated descriptor registers are updated (length = 8K bytes and access = R/W).
- Finally, the job's flag word (JDFLG) is tested, and if negative the job is running impure code and therefore there is no run-time system to map into the user's PARs and therefore USRMAP returns, If code is pure USRMAP enters MAPRTS.

4.3 Map a run-time system (MAPRTS)

If a job is associated with a run-time system (resident library) then this is mapped from PAR7 downwards.

- From the job's JDB, the RTS description block is accessed.
- From the R.CPTR entry in the RTS block, the CORTBL offset for the RTS is found.
- From the RlKCT slot of the RTS block, the size of the RTS in K words is given (no. of CORTBL words).
- In the R.REDO slot of the RTS block is the 4K description pattern for the descriptor registers of the map.
- MAPRTS then simply uses R.CPTR*16. as the PAR7 entry and loads R.REDO into the descriptor register 7.
- It then backs up to PAR6 and places PAR7-2ØØ there (4K less) and so on until the last entry, when the actual descriptor is used.
- MAPRTS then returns.

4.4 Scratch memory mapping for core-core transfers (SCRMAP)

At the start of this routine MAPHI and MAPLOW are already set up as the most and least significant parts respectively of the job start. (The real address).

SCRMAP is entered with R5 holding a user's virtual address.

The job of SCRMAP is to point the kernels PAR6 at the same real address as the user's virtual address in R5. But in V5C +V6 RSTS/E PAR6 also is used to point to monitor code at the proper time.

Therefore SCRMAP does the following:-

- It moves MAPHI into R4 (in preparation for R4/R5 double length register).
- It adds MAPHI to the virtual address in R5 and any carry to R4.
- Hence R4/R5 now holds the real address of the virtual address in R5.
- The combined register is shifted left 6 bits (divided by 64. to give a byte address on a 64. byte boundary).
- R5 now holds a valid PAR address which is then loaded into the kernel's PAR6.
- The descriptor register 6 of the kernel is set to 4K and R/W.
- Now, before R4/R5 was shifted left 6 bits (don't forget the remainder), R5 had been pushed on the stack.
- R5 is now restored from the stack and all but the lowest 6 bits are cleared.

4.4 Scratch memory mapping for core-core transfers (SCRMAP) (CONT'D)

- Therefore by adding 6*2ØØØØ to R5 we have in R5 the correct virtual address in kernel space for the user virtual address passed to SCRMAP.
- SCRUMP has been appended to SCRMAP to handle .Remaping into kernel window.

4.5 Compute real address (COROFF)

This routine computes a double length real address of a job start from its CORTBL entry (K address in bytes).

- The offset within CORTBL is found from R2 (which arrives holding the job's CORTBL entry), leaving the result in R2.
- Hence R2 is moved to R3 in preparation for treating R2/R3 as a double length register, and is then multiplied by 1Ø24 . ($2^{1Ø}$) to give the byte address.
- The least significant part (R3) is moved to MAPLOW.
- R2 is shifted left 4 bits to give some free bits (don't ask me why) and loaded into MAPHI as the most significant part of the real address (*16. of course).
- COROFF then returns.

4.6 Clock Routine CLOKØ

- This 'routine' is the line clock or KWII-P handler depending upon the system clock.
- It is activated at line frequency (Go CPS standard).
- The time currency in RSTS/E is the system tick (1/1Ø second).
- The WAITNT (wait for a system tick) bit is set in L3QUE to get the system to wait until 1 system tick is up.
- If a second has just passed, then the number of ticks to next second (in TIMCLK) is reset to equal the line frequency, and the QTIMER bit is set in L3QUE to schedule the timer service (TIMERS).
- If a minute has passed, the number of seconds to the next minute is reset.
- If a day has passed (144Ø. minutes) a new day is counted and no minutes to next day reset.
- JOBTIM (ticks used by current job is incremented) and the amount of his quantum remaining (in QUANT) is decremented.
- If the quantum has expired then QSCHED is set in L3QUE to effect a reschedule and QUANT is cleared.
- CLOKØ then jumps to RTI47 the common 'handler' return.

5.0 I/O Subroutines

These routines are I/O housekeeping and checking routines for the VRB I/O interface (i.e. non file structured I/O).

5.1 Routine to set status bits in JBSTAT on I/O completion (IOFINI)

It is called with R1 pointing to the DDB of the device concerned, and the address after the call (JSR R5, IOFINI) holding the required JBSTAT bits to set.

- From this the corresponding job number is obtained from the DDJBNO slot in the DDB.
- The number is then used to address the JBSTAT table and the bits in the word after the IOFINI call are set in the appropriate JBSTAT slot.
- IOFINI then returns.

5.2 Transfer a character from the user buffer (TTYS2Ø and TTYS3Ø)

At TTYS2Ø the current buffer address in user space (passed on in R5 at XRLOC in the XRB) is obtained. The current character (XRLOC points to it) is loaded into R2.

. The PS priority level is raised to 5 with kernel-mode new and user mode-old bits set. This prevents relevant interrupts.

. TTYS2Ø then returrs with R2 holding the character.

. On calling TTYS3Ø, if the C-bit is set then an error has been detected by the caller.

. R5 is pointing to the user's XRB at XRLOC the current byte in the user's buffer.

. Priority level is set to 3.

. If the C-bit is not set then the buffer address is incremented and the byte count decremented for the user in the XRB at XRLOC and XRBC respectively.

. TTYS3Ø returns.

. If the C-bit is true, then TTYS3Ø dumps the saved PC from the stack, and starts an I/O retry at IOREDO, described in the next section.

5.3 Re-effect I/O routine (IOREDO)

- IOREDO sets the JFREDO flag in the jobs JDFLG word.
- Calls SAVJOB to dump the job.
- Jumps to RTI3, the common level 3 exit.

5.4 Exit from I/O completion (IOEXIT)

- The job slot is obtained from JOB and the bit the job is waiting for (in its JBWAIT word) is set in the corresponding JBSTAT word.
- IOEXIT then branches to RTI3 to exit.

5.5 Response to EMT .SLEEP (SLEEP)

- SLEEP. is dispatched to form the EMTTBL
- The job slot is obtained from JOB.
- The entry in the user's XRB + Ø slot is obtained (being the sleep time in seconds) and entered into the job's JOBCLK entry.
- If the sleep is zero, SLEEP. immediately exits to RTI3.
- If the sleep is non zero, the JBSTAT word for the job is cleared making it non runnable.
- The job's JBWAIT slot is set to JSKEY and JBWAIT bits.
- SAVJOB is called to dump the job and
- SLEEP. then jumps to RTI3.

6 Character transfer from buffer pool to user (CHRUSR) and (CHRUØ1)

On calling both (JSR RØ, CHRUSR or CHRUØ1) R1 holds the devices DDB address add R5 points to the user's XRB at the current buffer address at XRLOC.

At CHRUSR:-

The PSW is set to priority 5 to inhibit relevant interrupts.

Routine FETCH is called to get a character from the buffer pool (placed in the current position (FP) offset to DDINP in the DDB).

On return from FETCH the priority level is set to 3 again and the C-bit is set if no more characters are available.

If this is the case, CHRUSR exits, otherwise it enters CHRUØ1.

At CHRUØ1, the character is moved into the user's buffer, the byte count incremented and the buffer address. If the byte count has now exceeded the buffer size (in XRB) then CHRUØ1 returns to the address given after the call if no user buffer room left.

If there is more room, then CHRUØ1 returns to the address given by the caller after the call.

5.7 Check availability of buffers (FREBUF)

At the call R1 points to the appropriate DDB. After the call is the Byte Count Fudge Factor.

It traces through small buffers from FREES (small buffer pool head), until enough have been found to accommodate request. FREBUF also checks to see if device has reached its quota yet as well as whether or not enough buffers are free to exceed quota to 24% of system total. If not enough can be found, the C-bit is set and FREBUF exits.

5.9 User I/O EMT interface (USERIX, USERIO)

. USERIX is entered from RTI3 for re-doing of IO

. USERIX tests the JFGO bit of the job's JDFLG word. If true, it exits to RTI3 at RTIØ3, to effect the force.

. If the job hasn't been forced out, it drops to level 3, clears the I/O re-do bit in JDFLG. Accesses the XRB and obtains the function (read/write) from XRCI+1 in the XRB.

. It then enters USERIO at USERI1, missing out on moving of XRB into the job's work block by MOVXRB.

. USERIO is dispatched to from EMTTBL on EMTs .READ and .WRITE and then, dispatches itself to the appropriate handler.

. On dispatch from USERIO the following contain:-

    RØ = 2 for read and 4 for write (access having been verified)

    R1 = address of DDB or FCB

    R2 = handler index (to access handler tables)

    R3 = pointer to XRB (at XRLEN)

    R5 = byte count pointer (XRB at XRBC)

. At USERIO, MOVXRB is called to move the XRB into the job's work block.

. The channel slot is got from XRCI in the XRB. The job's IOB (IO block) is entered at the channel slot to get the corresponding DDB or FCB address.

. If the channel is closed, the error status slot JDIOST in the job's JDB is set with bit NOTOPN and USERIO branches to RTI3.

. If the channel is open, then the legality of access is checked against the DDSTS slot of the devices DDB. I.E. you cant read from an output only device and vice-versa.

. If access is invalid then bit PRVIOL (privelege violation) is set in the job's JDIOST slot in its JDB.

- If access is valid, then the handler index is obtained from the DDB or FCB.
- The handler index is used to set the corresponding bit in the job's JBWAIT word, with the corresponding bit in JBSTAT being cleared.
- The handler index is then used to enter SERTBL (the table of handler addresses) and USERIO dispatches to the appropriate handler from the appropriate SERTBL entry.

6.0 Notes on the File Processor

General Functions of FIP.

1) opens & closes
   a) files (initializing & maintaining file structures)
   b) non-file-structured devices.
2) has facilities for executing non-resident system code (--hence it handles certain pieces of the compiler and RTS code).

FIP runs asynchronously from the rest of RSTS

- -has its own stack
- -has its own date base
- -runs at level 3
- -is started by EMT calls
- -is restarted from level 3 queue on disk I/O completion
- -serves one job at a time. Each request must be completed or reach an error condition before the next is begun. As FIP completes the service for each request, it scans its queue (orginating at FIQUE) and starts the next request if there is one.

When a user job requests a FIP service,

1) the request is placed in the FIP queue (originating at FIQUE),
2) the job goes into an I/O wait state for FIP. Normally such a job can be swapped out, but a few FIP requests cause a job to be locked in core.

The FIP queue is a chain of FIRQBs (File Request Queue Blocks) originating at FIQUE.

Each FIRQB specifies the parameters for a FIP request.

When FIQUE = Ø, FIP is no longer busy.

FIP uses 2 buffers of 256 words each.

1) FIBUF is a directory buffer.
2) FIPBUF is a buffer for non-resident code.

Typical steps in executing a FIP request:

1) initialize FIP's in-core data base.
2) if non-resident code is needed, read code module into FIPBUF.
3) dispatch to function handler.
4) set completion and/or error flags.
5) check FIQUE; if there is a next request, branch to it.
6) exit to RT13.

Approximately 5/6 of FIP is non-resident, swapped in as needed.

Notes on data errors:

1) There is no software error detection (such as checksumming).

   The only errors reported in data are those detected by the device controllers.
2) Data transmitted to a hung device is lost.

STRING HANDLING

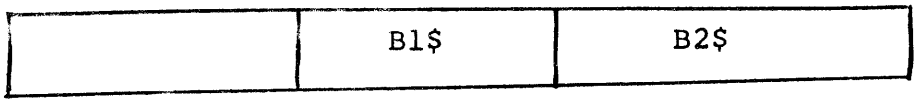
In processing strings, it is essential to arrange the program so as to avoid "garbage collection". Note that garbage collection swaps the job out and in again.

One technique is to pre-assign all strings to their maximum length using the SPACE$ function and then use LSET and RSET to assign strings. A variation of this technique is used by EDIT.
A data buffer is defined by opening a dummy keyboard file.
Then data is moved within the buffer by one of FIELD, LSET, and RSET statements. For example, to insert text into the middle of the buffer, the program acts as follows:

```
1000 FIELD #B%, D% AS BØ$, Z% - D% AS BØ$ :
     FIELD #B%, D% AS B1$, D8% AS B1$, Z% - D% AS B2$
```

The data buffer now has the following structure:

D% Z% - D%



D% D8% Z% - D%

We may now insert the string D8$ (length D8%) by the statements:

```
1100 RSET B2$ = BØ$ :                move what follows
     LSET B1$ = D8$ :                move in insertion
          Z% = Z% + D8% :            Z% is total length
          D% = D% + D8%              D% is length up to
                                     end of insertion.
```

To define the buffer, the program first executes:

```
100 B% = 2% : Z% = 50% * 74% :
      OPEN "KB:" AS FILE B%, RECORDSIZE Z%
```

As an example of three string-handling algorithms, consider the problem of truncating trailing blanks from a data record (for example, a card image). The two functions take as input any string, returning the same string without trailing blanks and CR-LF. The program segment performs the same function within an input buffer.

The slowest algorithm succesively reassigns the argument until it ends with a non-blank:

```
1000 DEF FNT$(X$) :
     X$ = LEFT(X$, LEN(X$)-1%)
          WHILE RIGHT(X$,LEN(X$)) <= " "
                AND LEN(X$) > 0%
1010      FNT$ = X$
1020 FNEND
```

Results: 113 sec. clock, 18.6 sec. cpu time.

The following is much more efficient. It scans backwards until a non-blank character is found. Only one assignment is made.

```
2000 DEF FNT1$(X$) :
          GOTO 2010 IF MID(X$,X%,1%) > " "
               FOR X% = LEN(X$) TO 0% STEP -1%
2010      FNT1$ = LEFT(X$,X%)
2020 FNEND
```

Results: 9 sec. clock, 6.0 sec. cpu time.

The most efficient algorithm uses the data buffer directly, avoiding the assignment caused by function calling and the

final assignment at line 2Ø1Ø above. (L% is the record length.)

```
3000 FOR K% = L% TO 1% STEP -1% :
          FIELD #2%, K%-1% AS L$, 1% AS L$ :
          IF L$>" " THEN
               FIELD #2%, K% AS L$ : GOTO 3Ø2Ø
3Ø1Ø      NEXT K% : LSET L$ = ""
3Ø2Ø...
```

Results: 7 sec. clock, 7.0 sec. cpu time.

Note that the more efficient algorithms are much more cpu-bound, showing that they are doing much less swapping.

Use of INSTR to scan a text.

Assume that a text is stored in the string S$ in the following format:

"word1 word2 word3 "

i.e. each word is followed by exactly one blank. A blank even follows the last word in the string.

The subroutine at line 1000 is executed once for each word. The word will be in W$.

note the following

W$ the word to process

L1% points to the first byte of the word in S$

L2% points to the blank following the current word

```
1ØØ L2% = Ø%                          !initialize for first
2ØØ L1% = L2% + 1%                    !L1% → first byte
3ØØ L2% = INSTR(L1%,S$," ")           !get trailing blank
```

```
4ØØ IF L2%
      THEN W$ = MID(S$,L1%,L2%-1%)  :
           GOSUB 1ØØØ : GOTO 2ØØ          !found, process it
5ØØ ! nothing left in S$
```

TIMING

The built-in functions TIME, TIME$, and DATE$ may be used to time programs. Note that the following subprogram will be complex if the program may run past midnite. To use these routines, execute

GOSUB 20010	to start timing
GOSUB 20000	to print the clock and restart timing.

```
20000 T0 = TIME(0) - T0 : T1 = (TIME(1) - T1)/10. :
      PRINT T0; "clock time", T1; "run time"; :
      IF T0 = 0.0 THEN PRINT
                  ELSE PRINT , T1/T0; "ratio"
20010 T0 = TIME(0) : T1 = TIME(1) : RETURN
```

RSTS -- STATEMENT HEADERS

EACH LINE-NUMBERED STATEMENT CONTAINS A 12-BYTE HEADER, USING MULTIPLE STATEMENTS CAN SAVE MUCH SPACE. FOR EXAMPLE,

```
100 X = Ø
110 Y = 1
120 Z = 2
```

SHOULD BE WRITTEN

```
100 X = Ø : Y = 1 : Z = 2
```

THIS WILL SAVE 24 BYTES.

NOTE, HOWEVER, THAT CERTAIN STATEMENTS ALWAYS HAVE A HEADER -- EVEN IF NO LINE NUMBER WAS WRITTEN. THEY ARE:

DEF	SINGLE- OR MULTIPLE-LINE
FNEND	
DATA	
FOR	
NEXT	
DIM	REQUIRES TWO HEADERS, ONE BEFORE AND ONE AFTER

THUS

```
100  S = Ø : FOR I = 1 TO N : S = S + X(I) : NEXT I
```

IS EQUIVALENT TO

```
100  S = Ø
110  FOR I = 1 TO N : S = S + X(I)
120  NEXT I
```

RSTS -- CODE GENERATION

RSTS COMPILES THE BASIC PROGRAM INTO SINGLE-BYTE CODES. THE VARIOUS COMPONENTS OF A STATEMENT HAVE THE FOLLOWING LENGTHS:

```
12   STATEMENT HEADER (FOR EACH LINE-NUMBER)
 1   STATEMENT END
 1   OPERATOR (+, -, MATRIX OPERATORS, ETC.)
 3   CONSTANT OR VARIABLE
 1   CONVERSION (EXCEPT THAT A = B% REQUIRES NO CONVERSION)
 4   FUNCTION
 6   USER-DEFINED FUNCTION
 3   INDEX
```

ALSO, ONE EXTRA BYTE MAY BE GENERATED TO FORCE THE LENGTH TO AN EVEN NUMBER OF BYTES. FOR EXAMPLE:

```
100 I% = A + B * SIN(C(X))
 12  3   3 1 3 1   4 3 3
                       1 (FOR CONVERSION TO INTEGER)
                       3 (FOR INDEXING)
```

TOTAL = 37 BYTES.

NOTE THAT A TEMPORARY VARIABLE MAY BE USED TO SAVE VECTOR ADDRESSING:

```
100  FOR I% = 1% TO N% :
          S = S + X(I%) : S2 = S2 + X(I%) * X(I%)      ! 42 BYTES
110  NEXT I
```

BUT

```
100  FOR I% = 1% TO N% :
          T = X(I%) :
          S = S + T : S2 = S2 + T * T                  ! 12+24 BYTES
110 NEXT  I
```

VTS -- VARIABLE STORAGE OPTIMIZATION

EACH DISTINCT NAME REQUIRES 4 BYTES PLUS THE NUMBER OF BYTES REQUIRED TO STORE THE DATA. IN ADDITION, EACH DISTINCT "FIRST NAME" REQUIRES 2 BYTES.

F, F%, F$, FNF, FNF%, FNF$, F(...), F%(...), F$(...)

EACH HAVE THE SAME FIRST NAME, WHILE

F AND F1 OR F AND G

DO NOT. THUS, IF YOU USE F, YOU SHOULD USE F% ALSO. TRY TO USE AS FEW DIFFERENT VARIABLES AS POSSIBLE -- REUSE VARIABLES WHENEVER POSSIBLE.

DATA STORAGE REQUIREMENTS ARE AS FOLLOWS

2	INTEGER
4 OR 8	FLOATING POINT
6 + N	STRING, WHERE N IS THE STRING LENGTH IN BYTES
4	IF THIS IS THE NAME OF A FUNCTION
26	BYTES FOR AN ARRAY HEADER, PLUS SPACE FOR EACH VALUE.

FOR CONSTANTS, ONLY THE VALUE SPACE IS NEEDED. NOTE THAT CONSTANTS ARE STORED WITHIN THE PROGRAM. THUS

```
100  A = 2.7 : B = 2.7
```

IS WASTEFUL, USE

```
100  A = 2.7 : B = A
```

INSTEAD.

OF COURSE,

```
100  A, B = 2.7
```

IS THE MOST EFFICIENT.

AVOID DYNAMIC ALLOCATION OF STRINGS. AT THE BEGINNING OF A PROGRAM, ALLOCATE ANY STRINGS AT THEIR MAXIMUM LENGTH:

```
100 X$ = SPACE$(MAX%)
```

WHERE MAX IS THE MAXIMUM LENGTH OF THE STRING. THEN, USE THE LSET AND RSET STATEMENTS TO MOVE DATA INTO THE STRING. DON'T RE-USE LET.

USE THE CHANGE STATEMENT TO ACCESS ELEMENTS AS AN INTEGER ARRAY. CONSIDER:

```
100 A$ = "ABCD" : B$ =SPACE$(5%)          : ALLOCATE SPACE
200 C$ = A$ : LSET B$ = A$                : DON'T ALLOCATE SPACE
300 D$ = A$ + "E" : E$ = LEFT(A$,2%)      : ALLOCATE SPACE
```

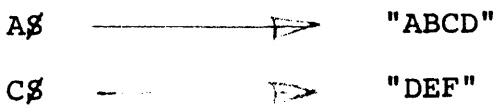
NOTE THE TWO TYPES OF STATEMENT IN LINE 200

`C$ = A$`	THE VARIABLE C$ POINTS TO THE SAME ADDRESS AS THE VARIABLE A$.
`LSET C$ = A$`	THE STRING THAT A$ POINTS TO IS COPIED INTO THE DATA AREA THAT C$ POINTS TO.

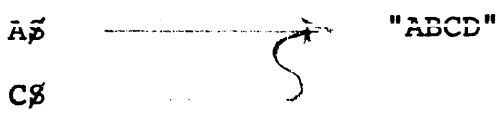
NOTE ALSO

`C$ = A$ + ""`	CONCATINATES A$ WITH THE NULL-STRING WHICH ALLOCATES MEMORY SPACE AND THEN COPIES THE STRING THAT A$ POINTS TO.

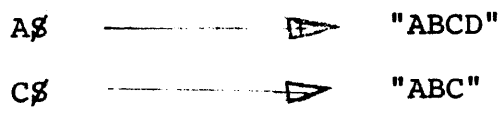
STRING MOVEMENT EXAMPLE. ASSUME THE SITUATION WHERE



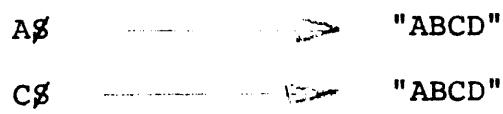
AFTER 100 C$ = A$



AFTER 100 LSET C$ = A$



AFTER 100 C$ = A$ + ""



NOTE THAT IF A$ POINTS TO AN I/O BUFFER (BY THE FIELD STATEMENT), THE STATEMENT

```
100  C$ = A$
```

WILL CAUSE C$ TO POINT TO THE SAME PART OF THE I/O BUFFER. IF A NEW DATUM IS READ (USING GET), C$ WILL POINT TO DIFFERENT DATA.

WHEN YOU MUST MANIPULATE STRINGS, TRY TO USE ASCII AND CHANGE STATEMENTS. AVOID USING TEMPORARY VARIABLES IF NOT NECESSARY. INSTEAD OF

```
100  M$ = LEFT(Q$, 3%)
110  N$ = RIGHT(P$, 2%)
120  T$ = M$ + N$
```

USE

```
100  T$ = LEFT(Q$, 3%) + RIGHT(P$, 2%)
```

OPEN FILES AT THE BEGINNING OF THE PROGRAM, NOT IN THE MIDDLE.

RSTS -- CODING OPTIMIZATION

USE INTEGER VARIABLES WHENEVER POSSIBLE, ALWAYS SPECIFY EITHER '%' OR '.' WHEN WRITING CONSTANTS.

USE MULTIPLE STATEMENTS PER LINE. (THE MAXIMUM STATEMENT LENGTH FOR MULTIPLE-LINE STATEMENTS IS 256 BYTES).

USE ! TO INDICATE COMMENTS INSTEAD OF REM.

IN IF STATEMENTS, DO NOT COMPARE VALUES WITH ZERO:

```
        IF A%<> Ø GO TO 1ØØ
```

SHOULD BE WRITTEN

```
        IF A% GO TO 1ØØ
```

USE STATEMENT MODIFIERS WHENEVER POSSIBLE (SEE BELOW).

FREQUENTLY USED CONSTANTS SHOULD BE DECLARED AS VARIABLES.

AVOID MAT COMMANDS.

AVOID ARRAYS -- USE INDIVIDUAL VARIABLE NAMES WHERE POSSIBLE.
IF YOU USE ARRAYS, ALWAYS DIMENSION THEM AND SPECIFY INTEGER SUBSCRIPTS. ARRAY ELEMENTS SHOULD BE INTEGERS WHENEVER POSSIBLE.

RE-USE PREVIOUSLY CALCULATED ITEMS. AVOID INTERMEDIATE TERMS:

```
A = B + C

D = A + E
```

SHOULD BE WRITTEN

```
D = B + C + E
```

UNLESS A IS NEEDED INDEPENDENTLY.

DON'T USE USER-DEFINED FUNCTIONS, ESPECIALLY WITH STRINGS. USE GOSUB.

ALWAYS EXIT FROM SUBROUTINES VIA RETURN, NEVER VIA GOTO.

RSTS -- FILE AND MOVING-HEAD DISK OPTIMIZATION

OPTIMIZE PACK AND FILE CLUSTERSIZE.

KEEP LARGE, FREQUENTLY USED FILES ON SEPERATE DISKS.

PRE-EXTEND FILES TO THEIR MAXIMUM SIZE.

PRE-ALLOCATE SCRATCH FILES. THEN, DO NOT KILL THEM, BUT CLOSE AND RE-USE THEM.

CLEAN DISK STRUCTURES: COPY ALL ACTIVE FILES TO MAGTAPE, THEN RE-STRUCTURE THE PACK USING DSKINT.

KEEP PRODUCTION AND DEVELOPMENT ACCOUNTS SEPERATE.

IF TWO FILES ARE TO BE OPEN AT THE SAME TIME, KEEP THEM ON SEPERATE DISKS IF POSSIBLE.

FILE CLUSTERSIZE

A FILE-CONTROL-BLOCK (FCB) IS USED TO LOCATE THE ACTUAL SEGMENTS OF A FILE. SEVEN SEGMENTS FIT IN ONE FCB. IF THE FILE REQUIRES MORE THAN SEVEN SEGMENTS, A SECOND FCB IS READ FROM DISK. TRY TO DEFINE FILE CLUSTERSIZE SO ONLY ONE FCB IS NEEDED.

THE NUMBER OF FCB BLOCKS IS GIVEN BY THE FORMULA:

N = 1 + INT((FILE LENGTH IN SECTORS / 7) x CLUSTERSIZE)

FOR EXAMPLE, FOR A FILE HAVING A LENGTH OF 200 SECTORS, N WILL HAVE THE FOLLOWING VALUES

N	CLUSTERSIZE
29	1
15	2
8	4
4	8
2	16
1	32

CHOOSE A CLUSTERSIZE OF 32, IF POSSIBLE.

RSTS -- RECORD TRANSFER OPTIMIZATION

USE RECORD I/O WHEREVER POSSIBLE.

AVOID READ AND PRINT AS THEY ARE VERY SLOW (CHARACTER BY CHARACTER).

WHILE VIRTUAL-ARRAY I/O IS FAST, IT MAY REQUIRE MULTIPLE ACCESSES.
FOR EXAMPLE:

```
100 DIM#4%, A(500%), B(500%)
200 C(I%) = A(I%) + B(I%) FOR I% = 1% TO N%
```

THIS WILL REQUIRE ONE READ FOR A(I%) AND ANOTHER FOR B(%). THE SECOND WILL BE TO A DIFFERENT AREA OF THE DISK. IF A FILE STRUCTURE WERE DEFINED WHERE A(I) AND B(I) WERE PARTS OF RECORD I IN A FILE ACCESSED BY GET, ONLY ONE ACCESS WOULD BE NEEDED. THUS, YOU TRADE A MORE COMPLEX PROGRAM (GET FOLLOWED BY FIELD FOLLOWED BY THE COMPUTATION) FOR A SIMPLER I/O STRUCTURE AND A MUCH FASTER-RUNNING PROGRAM -- ESPECIALLY IF THE FILES ARE LARGE AND ARE STORED ON A MOVING-HEAD DISK.

DEVICE DRIVER

DIGITAL EQUIPMENT CORPORATION

A Device Driver Needs:	OPEN Routine	Lev 3	OPN XXX
	Service Routine	Lev 3	SER XXX
	Close Routine	Lev 3	CLS XXX
	Interrupt Handler	INT Level	XXX INT

Three exits and nine sub-routines exist in the Monitor for use by Device Drivers.

1	REGSAV	ISR RO->R5 SAVE
2	FETCH	UPDATES DDB & GETS CHAR
3	STORE	STORE CHAR & UPDATE DDB
4	CIRBUT	CLEARS SMALL BUFFER CHAINS
5	FREBUT	FREE BUFFER AVAILABILITY CHECKER
6	CHRUSR	XFR CHAR FROM MON SMALL BUFFER POOL USRBFR
7	CHRUØ1	MOV CHR TO USER BUFFER
8	TTY520	MOV CHR FROM USER BUFFER TO DRIVER USING XRB
	TTY530	UPDATE XRB
9	IOFINI	IB STATUS TO SET FROM ISR
10	IOEXIT	I/O COMPLETE EXIT & ALSO RETURN ERROR CODE
11	IOREDO	CAN't DO NOW - DO LATER
12	RESRT4	ISR EXIT

ADDING A DEVICE DRIVER TO RSTS V5B

****

Introduction - There exist hooks in the RSTS Monitor for adding 3 device drivers to the system. Including a device driver requires a re-system generation. Two slots are presently used for optional devices. XXX is used for pseudo keyboards and ZZZ is used for the 278Ø Emulator. Therefore, the YYY slot is recommended.

General Philosophy - There are two types of device drivers in RSTS/E. Devices capable of NPR transfers move data directly from the device to the user data buffer. This requires the user to be locked in core during the transfer.

Slower character oriented devices perform data transfer to monitor buffer space, while the job is not resident in core. Later the data is tranferred from the monitor buffer area to the user buffer. Monitor buffers can be extracted from the small buffer pool or can be special buffers defined in the read/write portion of the monitor. Small buffers are 16 words long, the first word contains a link word leaving room to store 3Ø characters of data. Monitor subroutines are available for storing and extracting characters from these buffers. Pointers to these buffers and current position within these buffers is stored in a device data blocks for a particular device. If special buffers are defined within the monitor, the device driver must manager that buffer.

Definitions -

FIRQB - File Request Queue Block

The "FIRQB" is the means for queueing request for the file processor. Opening and closing of devices and files are processed this way. The FIRQB is actually a parameter block which contains information pertinent to the request.

XRB = Transfer Control Block

Reads and writes are initiated by the Run-Time System. The Monitor, not the file processor handles the request. Information such as byte count and transfer address is passed in the XRB. A copy of the XRB exists in both the user image and the monitor.

DDB = Device Data Block

There exists one DDB per device unit. The format for this device is basically the same for various devices. However, each device uses various words and bits differently.

for legal device names.

```
DEVNAM: "DP      ;IF RP IS SYSTEM DISK
        "DF
        "DK
        "KB
        "DT
        "LP
        "PR
        "PP
        "CR
        "MT
        XXXXXX
        YYYYYY
        ZZZZZZ
        -1       ;END OF TABLE
```

IOB = IO Block. The login procedure includes setting up the job data structure, which consists of 3 small buffers from the free small buffer pool.
One is used as a job data block, the first word of which points to another small buffer which is used as the IOB for thisjob. The IOB consists of one entry per I/O channel.
The third small buffer which is pointed to by the job data block is a work block for FIRQB's and XRB's.

JBSTAT/JBWAIT = the monitor contains a Job Status (JBSTAT) and Job Wait (JBWAIT) Table. Each table contains a one word entry per job. The JBWAIT table indicates job's waiting for a completion. The JBSTAT table indicates function completions.
The following describes the format of a one word entry in both of these tables.

```
JSDSK = 1        DISK WAIT
JSKEY = 2        KB: WAIT
JSDTA = 4        DT: WAIT
JSLPT = 1Ø       LP: WAIT
JSPTR = 2Ø       PR: WAIT
JSPTP = 4Ø       PP: WAIT
JSCDR = 1ØØ      CR: WAIT
JSMTA = 2ØØ      MT: WAIT
JSXXX = 4ØØ      XX: WAIT
JSYYY = 1ØØØ     YY: WAIT
JSZZZ = 2ØØØ     ZZ: WAIT
JSTEL = 4ØØØ     KB: TELEPRINTER WAIT
JSFIP = 1ØØØØ    FILE PROCESSOR WAIT
JSTIM = 2ØØØØ    .SLEEP WAIT CONDITION
JSNUL = 4ØØØØ    SMALL BUFFER WAIT CONDITION
```

Overview -

A device driver in RSTS requires the following main sections.

1. Open routine

2. Close routine

3. Level 3 service routine - read/write processor

4. Interrupt Service Routine

For each device driver in the RSTS monitor, there exists a handler index. These are assigned in the following manner:

Ø	disk
2	keyboards
4	DECtape
6	line printer
1Ø	paper tape reader
12	paper tape punch
14	card reader
16	magtape
20	XXX device
22	YYY device
24	ZZZ device

The RSTS monitor contains common code for processing OPEN, CLOSE, READs and WRITEs (GET, PUT, INPUT, PRINT) for devices. The open and close code is contained in a module named "OPN". Reads and writes to slower character oriented devices perform data transfers to monitor buffer space, while the job is not resident in core. Later the data is transferred from monitor buffer space to the user buffer area. Monitor buffers are extracted from the small/large buffer pool or defined as part of the device driver. Reads and writes to devices are processed in the module "MON". Completion of these functions depends on three dispatch tables which are accessed using the handler index.

1. Open table (OPNTBL) - this table contains one slot for every device, which contains the address of the open routine for that device.

2. Close table (CLSTBL) - similar to open table only contains pointers to close routines for every device.

3. Service table (SERTBL) - contains addresses for device service routines which handle reads and writes.

Open "PR:" as file 1%

The above BASIC PLUS command opens the paper tape reader on channel 1. This statement causes a file processor request to be issued. The code executed as a result of this request can be either resident or non-resident. One of the system generation questions asks whether you want resident file handling. This resident file handling includes the open code.

The open code functions as described below.

1. Check channel specified. If some device or file is already open on this channel, abort with an error message.

2. Check device name specified with device name table. If this device does not exist, abort with a no device error, the unit # is also verified.

3. Check DDB for device owner. If this job already owns the device, exit. If another job owns the device, abort with an error message. Otherwise update the DDB with the following information.

    1. Current time
    2. Clear status bit
    3. Set access count to Ø.
    4. Set job #.

4. Using the handler index which exists in the DDB, calculate the address for the appropriate device open routine. Now, execute the device open routine as a subroutine.

5. Increment access count in DDB.

6. Set up DDB address in IO block entry for specified channel.

CLOSE

1. Remove entry from IO block entry for this channel.

2. Using handler index from DDB, calculate address for appropriate close routine. Execute CLOSE routine as a subroutine.

3. Decrement access count.

4. If device is not in use update DDB and save device time job data block for the job who just used the device.

READ/WRITE Processing

USERIO - associated with this routine is the transfer control block (XRB). The XRB contains the following information:

1. Length of I/O buffer in bytes.

2. Byte count for transfer

3. Pointer to buffer start for transfer

4. Channel # for transfer

5. Starting device block # for transfer

6. Wait time for tty input

USERIO performs the following:

1. Check that specified channel is open. If not open abort with an error message.

2 Check DDB for legal access (read locked and write locked status). Insure we are not reading from a line printer. If access is illegal, abort with the error message "PROTECTION VIOLATION".

3. Set appropriate JBWAIT condition for this job.

4. Map the user buffer in kernel space with the monitor. This is done to allow core to core data transfers between the users buffer and monitor buffers.

5. Using the handler index calculate the address for the appropriate level 3 service routine. Then jump to the appropriate device service routine.

USERIX

This is essentially a different entry point to "USERIO". It is used for redoing I/O. If for some reason I/O was stalled by the or a special data buffer can be defined inthe read/write portion of the monitor.
used for redoing I/O. If for some reason I/O was stalled by the driver exiting to IOREDO, the I/O request would later be processed through this entry.

RELATED MONITOR ROUTINES AND ENTRIES
****

An understanding of the following monitor routines and entry points is necessary to implement a driver. Including any of the following requires an appropriate global definition (.GLOBL).

1. REGSAV
2. FETCH
3. STORE
4. CLRBUF
5. FREBUF
6. CHRUSR
7. CHRUØ1
8. TTYS2Ø AND TTYS3Ø
9. RESRT4
1Ø. IOREDO

12. IOFINI

1. REGSAV- saves registers Ø through 5 on the stack. Used by the interrupt service routine.

2. FETCH - get character from the small buffer chain when ready to output a character to the device.

3. STORE - takes a specified character and stores it in the small buffer chain, updating appropriate pointers and counts. Also attempts to allocate another small buffer if no free space is available in existing buffers.

4. CLRBUF - clears small buffer chain which is being used for either input or output, updates appropriate counters, pointers, and return buffers to the free buffer pool.

5. FREBUF - free buffer availability checker. If there are not at least 1Ø small buffers left in the system, return a negative indication. If this device has not used its quota return position indication.
Else if not at least 2Ø% of total free in system return negative.
Else if this device has 25% or more of total system small buffers return negative.
Else return positive.

6. CHRUSR - transfer a character from the monitor small buffer pool to the users buffer.

7. CHRUØ1 - move specified character to the user buffer.

8. TTYS2Ø - move a character from the user buffer to driver using information in the XRB.

   TTYS3Ø - update pointers and count in XRB.

9. RESRT4 - common return from interrupt processing at levels 4 through 7. If the interrupted level is processor level 3 or higher issue an RTI. Otherwise, look for other monitor routines to start up.

1Ø. IOREDO - signal I/O redo. This exit is used to indicate I/O cannot be serviced at this time, but should be redone later.

11. IOEXIT - exit saying I/O complete. Used by level 3 service. Also used to return an error code. Most common error types are:

EOF	end of file
DATERR	general parity error
HNGDEV	device is hung
NOROOM	device is "full"

12. IOFINI - subroutine called from Interrupt Service Routine.

"IOREDO".

1. Save and Restore Registers

```
CALL:   JSR     R5,REGSAV
        exits with SP→RØ,R1,R2,R3,R4,R5
        and the carry bit unchanged
CALL:   JSR     R5,REGRES
        pops RØ-R5 from stack leaving carry bit
        unchanged
```

2. Character Fetching Routine

```
CALL:   R1→DDB
        JSR     R5,FETCH
        +       OFFSET
        ...     RETURN [C=1 FOR NO CHARACTERS LEFT]
where:  OFFSET=DDINP+FP for input buffer
        OFFSET=DDOUT+FP for output buffer
        character is returned in R2
        R4 is clobbered
```

3. Character Storing Routine

```
CALL:   R1→DDB
        R2=character to store
        JSR     R5,STORE
        +       OFFSET
        ...     RETURN [C=1 FOR NO ROOM LEFT]
where:  OFFSET=DDINP+FP for input buffer
        OFFSET=DDOUT+FP for output buffer
        R3,R4 are clobbered by the process!
```

4. Buffer Chain Clearer Routine

```
CALL:   R1→DDB
        JSR     R5,CLRBUF
        +       OFFSET
        ...     RETURN
where:  OFFSET=DDINP+EP for inputbuffer
        OFFSET=DDOUT+EP for output buffer
        R3 is clobbered by the process!
        CPU priority raises to 5 during process
```

5. Free Buffer Availability Checker

```
CALL:   R1→DDB
        JSR     R5,FREBUF
        +       XXX.OB
        ...     RETURN [C=1 IF NOT ENOUGH]
where:  XXX.OB is BC count fudge factor
```

6. Character from Buffer Pool to User Routine

```
CALL:   R1→DDB
        R5→XRB @ XRLOC
        JSR     RØ,CHRUSR
        ...     RETURN IF NO USER BUFFER ROOM LEFT
        ...     RETURN IF ROOM FOR MORE
        ...     RETURN IF NO BUFFER POOL CHARS LEFT
```

7.

```
CALL:   SAME REGISTER AS 6.
        JSR     RØ,CHRUØ1
        ...     RETURN IS NO USER BUFFER ROOM LEFT
```

```
                    ...     return if room for more
                    R3 is clobbered by both routines
```

8. Character from User; IOF; Return

```
        CALL:  R5→XRB  @ XRLOC
               JSR     PC,TTYS2Ø
               ...     RETURN
        Character in R2
```

Check C bit and Wait or Adjust Counts and Return

```
        CALL:  R5→ XRB @ XRLOC
               CARRY=1 means buffering failure
               JSR     PC,TTYS3Ø
               ...     RETURN [only if no failure]
        R5 NOW →XRB @ XRBC
```

9. Return from Interrupt

```
        CALL:  JMP     RESRT4
```

1Ø. Re-Do the I/O Routine

```
        CALL:  JMP     IOREDO
```

11. Exit from I/O Completion

```
        CALL:  JMP     IOEXIT
```

Exit from I/O Completion with an Error

```
        CALL:  MOVB    #HNGDEV,@IOSTS
               JMP     IOEXIT
        HNGDEV can be replaced with DATERR, EOF, NOROOM, ETC.
```

Routine to Set Status Bits in JBSTAT

```
        CALL:  R1→DDB
               JSR     R5,IOFINI
               +       JBSTAT BITS TO SET
               ...     RETURN
        JOB # is in R4 on return
```

REGISTER DEFINITIONS FOR OPEN, CLOSE, AND SERVICE

The following registers are set up before entering the following sections of the device driver.

OPEN

```
R1→DDB
R4→FIRQB
```

The following instructions should be included in the OPEN subroutine to transfer the default buffer size and flag value to the Run-Time System.

```
MOV #???,FQBUFL[R4)    ;SET DEFAULT BUFFER LENGTH
                          THIS IS ALSO THE MINIMUM
                          BUFFER SIZE (RECORD SIZE)
                          BUFFER SIZE (RECORDS SIZE)
MOV #YYYFLAG,FQFLAG(R4)  ;SET FLAG VALUE
```

CLOSE

R1→DDB
R4→FIRQB
R5→DDB

Read/Write Service

RØ - 2 for read, 4 for write (access verified)
R1 - FCB/DDB pointer
R2 - handler index
R3 - XRB pointer(@ XRLEN)
R4 - kernel window address to user's buffer
R5 - byte count pointer (XRB @ XRBC)

MACROS

The following macros must be defined in the device driver

1. VECTOR ORIGIN,ADDRES,PRI

where: ORIGIN is the address of the interrupt vector for this device

ADDRESS is the address of the interrupt service routine for this device

PRI is the priority at which the interrupt service routine should run. This is actually the value which goes into the second word of the interrupt vector. The following symbols were defined in the module kernel which is assembled with device driver and may be used here.

PR7 - defined as 340 to indicate priority 7.
PR6 - defined as 3ØØ to indicate priority 6.
PR5 - defined as 24Ø to indicate priority 6.
PR4 - defined as 2ØØ to indicate priority 4.

2. $$$DEV DEV,STS,IBC,OBC,SIZ

where: DEV indicates the device. For example, YYY would be used if adding a YY device driver.

STS indicates status. The following symbols defined in kernel may be used here.

DDWLO - device is write-locked

FLGPOS - file's position needs checking

FLGFRC - file is force type, not blocked

Run-Time System on output even if the data buffer in the job is not full).

FLGKB - file is keyboard type

FLGRND - file is random access type

DDRLO - device is read-locked

DDNFS - device is non file structured

IBC is the number of small buffers allowed for input.

OBC is the number of small buffers allowed for output.

SIZ indicates line size.

NOTE: Although IBC and OBC specify the number of small buffers a device may allocated, buffer management will allow a device to allocate more buffers if there are a sufficient number of free small buffers in the system.

$$$FLG DEV,Q1

where: DEV indicates the device. For example. YYY would be used for a YY device.

Q1 indicates the appropriate flag values for a device. Any of the following symbols may be OR'ed together.

DDNFS - file is non-file structured

DDRLO - file is read locked

DDWLO - file is write locked

FLGPOS - file's position needs checking. Used by terminal and line printer drivers. Keeps vertical and horizontal positions. Checks control characters.

FLGFRC - file is force type, not blocked. Only significant for output devices. Used for character oriented devices, data is transferred from the Runtime System without waiting for the buffer to fill up. Should always be set for input only devices so that error messages will be returned immediately.

FLGKB - file is keyboard type
should indicate device is human oriented device not used by the system. Meant to be checked

by the user by checking the status

L FLGRND - file is random access type
The Run-Time System will not accept the "RECORD" option. It will return an error message instead.

4. ORG YYYCTL - Only necessary if a portion of the driver must be read/write. Otherwise, all code in the driver must be read only code. This line of code must be included if you want to add your flag words and buffers.

INCORPORATING A NON-STANDARD DEVICE DRIVER IN RSTS/E

The following modifications to the system generation procedure are required.

After answering all SYSGEN questions the following message appears on the terminal.

```
SYSGEN:IF YOU HAVE ANY SPECIAL REQUIREMENTS WHICH REQUIRE
SYSGEN:EDITING EITHER THE CONFIGURATION FILE (CONFIG.MAC)
SYSGEN:OR THE BATCH GENERATION FILE (SYSGEN.BAT), ABORT
SYSGEN:NOW BY TYPING "CONTROL/C" AND THEN "TE". RESUME AT
SYSGEN:THIS POINT BY TYPING "BATCH SYSGN2". OTHERWISE,
SYSGEN:TYPE "CO" TO CONTINUE WITH SYSTEM GENERATION:
```

At this point the files "CONFIG.MAC" and "SYSGEN.BAT" should be modified. Therefore type "CONTROL/C" and then "TE".

The configuration file (CONFIG.MAC) must be modified to include the following parameters.

```
YYYY11=N        ;N=Number of units for this device
YYYYYY="AB      ;device name, must not conflict with existing
                devices (DT, MT, PR, etc.)
```

The batch stream (SYSGEN.BAT) must be modified to include assembly and linking of the new driver.

"SYSGEN.BAT" includes the following commands for assembling TBL and TTY. A command for assembling YYY must be included.

```
        $RUN MACRO
        #TBL,TBL/CR<CONFIG,COMMON,KERNEL,TBL
        $RUN MACRO
        #TTY,TTY/CR<CONFIG,COMMON,KERNEL,PKB,TTY
***     $RUN MACRO
***     #YYY,YYY/CR<COMMON,KERNEL,YYY
```

A command must also be added to the link the driver to RSTS.

```
        $RUN LINK
        #UPDATE
        #NOFAIL
        #PATCH
```

```
***     #YYY
        #RSTS/IN:OPN:SND/E
```

After including the above modification, resume by typing "BATCH SYSGN2".

ERRLOGGING AND CRASH ANALYSIS

RH11/RPØ4 ERROR LOGGING IN RSTS/E

RSTS/E error logging uses 13 word packets. The first 4 words are standard for all packets. The first 4 words contain:

ERROR CODE
DATE
TIME (in seconds)
JOB
PROCESSOR STATUS

RPØ4's use 6 of the remaining 9 words to save the following.

RHCS1	RH11 Control Status 1
RHCS2	RH11 Control Status 2
RHDS	RH11 Drive Status
RHER	RH11 Error Status
RBDC	RPØ4 Desired Cylinder
RBDA	RPØ4 Desired Track/Sector
RBOFF	RPØ4 Offset

1 of the remaining 3 words is used as follows:

11/7Ø:	RHBAE	RH7Ø Address Extension
	RHCS3	RH7Ø Control Status 3

Non-11/7Ø: Flag indicating no RHBAE or RHCS3

The final 2 words are used as follows:

RBER2 & RBER3 are Ø

Flag indicating RBER2 = Ø
Flag indicating RBER3 = Ø

RHWC	RH11 Word Count
RHBA	RH11 Bus Address

ERROR LOGGING ON RSTS/E

March 21, 1975

RBER2 ≠ Ø
RBER3 = Ø
Flag indicating RBER3 = Ø
RBER2 RPØ4 Error Register #2
RHBA RH11 Bus Address

RBER2 = Ø
RBER3 ≠ Ø

Flag indicating RBER2 = Ø
RHWC RH11 Word Count
RBER3 RPØ4 Error Register #3

RBER2 & RBER3 ≠ Ø

RBER2 RPØ4 Error Register #2
RBER3 RPØ4 Error Register #3

****

THERE EXISTS A CORE TABLE CALLED "ERRTBL" WHICH HOLDS TEN 12-WORD DECIMAL ENTRIES. THE LENGTH OF THIS TABLE IS DEFINED BY THE PARAMETER "ERRSIZ" WHICH APPEARS IN THE CONFIGURATION FILE "CONFIG.MAC" AND DEFINES THE NUMBER OF TABLE ENTRIES.

ERRLOG CREATES A TABLE ENTRY ON THE STACK, COMPARES THIS ENTRY FOR A POSSIBLE DUPLICATION IN THE TABLE "ERRTBL". IF A DUPLICATION IS FOUND, A REPEAT COUNT IS INCREMENTED IN THE ALREADY EXISTING TABLE ENTRY AND THE ROUTINE IS EXITED.

A MORE DETAILED DESCRIPTION FOLLOWS:

ERRLOG FIRST DECIDES WHETHER AN ERROR IS BEING LOGGED OR THE SPECIAL SYS FUNCTION TO FIP (-15) FOR ERRLOGGING IS BEING EXECUTED. IF A PARAMETER OF 0 IS SPECIFIED IN THE SYS FUNCTION, ERRLOG INTERPRETS THIS AS AN ANNOUNCEMENT THAT THIS JOB IS THE ERROR LOGGER AND SHOULD BE MADE RUNNABLE EVERY TIME AN ERROR IS LOGGED. ERRLOG RECORDS THE JOB NUMBER AND JOB DATA ADDRESS OF THIS PARTICULAR JOB.

THE PARAMETER IS NON-ZERO, TABLE ENTRIES FROM "ERRTBL" ARE PASSED BACK TO THE USER. ERRCPY USES THIS SYS FUNCTION TO EXTRACT ENTRIES FROM CORE AND COPY THEM TO THE DISK FILE "$ERRLOG.FIL".

IF AN ERROR IS BEING LOGGED, A 12 DECIMAL WORD ENTRY IS CONSTRUCTED ON THE STACK. THE FORMAT FOR THIS ENTRY FOLLOWS.

WORD 0	LOW BYTE CONTAINS error code The HIGH BYTE CONTAINS A REPEAT COUNT
WORD 1	BITS 1 THROUGH 15 CONTAIN THE DATE BIT 0 IS THE HIGH BIT OF THE TIME IN HRS/MINS
WORD 2	BITS 6 THROUGH 15 ARE THE LOW BITS OF THE TIME IN HRS/MINS BITS 0 THROUGH 5 REPRESENT SECONDS TILL MIDNIGHT
WORD 3	BITS 6 THROUGH 15 REPRESENT THE PSW IN PACKED FORMAT BITS 0 THROUGH 5 REPRESENT THE JOB NUMBER
WORD 4	*VIRTUAL PC
WORD 5	*HIGH 12 BITS OF ABSOLUTE PC, RIGHT JUSTIFIED
WORD 6 THROUGH WORD 11	VARIOUS REGISTERS, DEPENDENT ON ERROR TYPE

*INCLUDED FOR THE FOLLOWING ERRORS ONLY
- TRAP THROUGH 4
- RESERVED INSTRUCTION TRAP
- MEMORY MANAGEMENT ERROR
- PARITY ERROR

POWERFAIL AND TRAP THOUGH 0 SAVE NOTHING BEYOND WORD 3.

IF ANY OF THE FOLLOWING CONDITIONS ARE TRUE, AN ILLEGAL PSW IS ASSUMED AND THE PSW ENTRY IS SET TO ALL 1'S.

T BIT SET
UNUSED BIT SET
CURRENT OR PREVIOUS MODE SUPERVISOR "01"

THE PSW IS PACKED INTO BITS 6 THROUGH 15 OF WORD 3 IN THE FOLLOWING MANNER:

BIT 15	1=CURRENT MODE USER 0=CURRENT MODE KERNEL
BIT 14	1=PREVIOUS MODE USER 0=PREVIOUS MODE KERNEL
BIT 11-13	PRIORITY
BIT 10	GENERAL REGISTER SET, 1 IF 11/45
BIT 9	N
BIT 8	Z
BIT 7	V
BIT 6	C

CALCULATING ABSOLUTE (PHYSICAL) ADDRESS FROM VIRTUAL ADDRESS:

N= BITS 15-13 OF VIRTUAL ADDRESS
XXX= BITS 12-06 OF VIRTUAL ADDRESS

C(PAR N) + XXX = HIGH ORDER 12 BITS OF PHYSICAL ADDRESS

LOW ORDER 6 BITS ARE THE SAME IN BOTH PHYSICAL AND VIRTUAL SPACE.

MISSED ERRORS

IF "ERRTBL" IS FULL AFTER A NEW ENTRY IS CREATED ON THE STACK, AN EXISTING ENTRY WILL BE OVERLAYED. BEFORE THIS HAPPENS, THE REPEAT COUNT FROM THE TABLE ENTRY ABOUT TO BE LOST IS ADDED TO A COUNTER WHICH IS ALSO INCREMENTED 1 FOR THAT ENTRY. THE CONTENT OF THIS COUNT IS LATER PASSED TO THE USER AS IF IT WERE A TABLE ENTRY. THIS ENTRY CONTAINS ONLY TWO WORDS.

WORD 0	0
WORD 1	NUMBER OF MISSED ERRORS

IF THIS COUNT REACHES -1, IT IS RESET TO THE HIGHEST POSITIVE VALUE.

THE "ERRTBL" COULD OVERFLOW IN THIS MANNER, IF "ERRCPY.BAS" IS NOT RUNNING, OR ERRORS ARE BEING LOGGED AT AN EXTREMELY RAPID RATE.

THE FOLLOWING 16 ERROR TYPES ARE LOGGED AND DISPLAYED WITH THE ERROR LOGGING FACILITIES IN RSTS/E.

DECTAPE
RF11
RC11
RK11
RP11
MAGTAPE
KEYBOARD
TRAP THROUGH 4
POWER FAIL
TRAP THROUGH 0
RESERVED INSTRUCTION
JUMP TO 0
RUN-TIME SYSTEM ERROR (CHECK-SUM ERROR)
MEMORY MANAGEMENT
DH11
PARITY

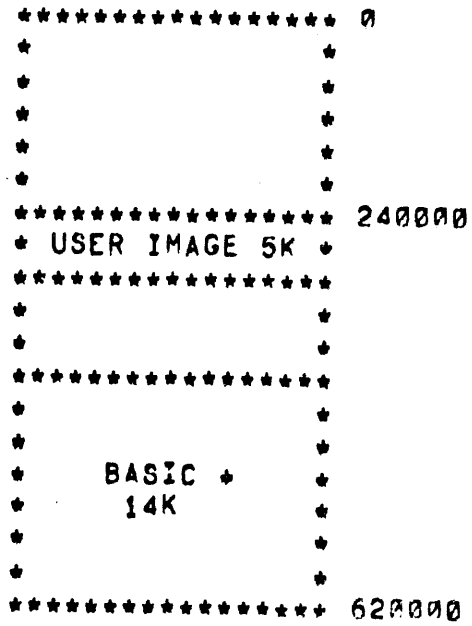
THE INFORMATION DISPLAYED BY ERRDIS REGARDING MOST ERRORS IS SELF-EXPLANATORY. THAT IS HARDWARE REGISTERS ARE DISPLAYED. SELECT ERRORS CAN BE CAUSED BY REFERENCING A UNIT # WHICH IS NOT PHYSICALLY SELECTED.

THE DISPLAY OF TWO ERROR MESSAGES WHICH ARE NOT OBVIOUS ARE DISCUSSED BELOW. (DH11 AND CHECKSUM)

1. CHECKSUM ERRORS

THE FOLLOWING REGISTERS ARE DISPLAYED. SAMPLE VALUES ARE INCLUDED TO HELP EXPLAIN THEIR MEANING.

*UISAR0	2400	USER IMAGE BASE = 2400*100 = 240000
UISAR7	6000	TOP OF BASIC = 6000*100+20000 = 620000
SIZE	5	USER IMAGE SIZE
RTS SIZE	16	RUN-TIME SYSTEM SIZE IN OCTAL
R3		
R5		



1. TRUE CHECKSUM.
   R3<>R5 = R3 = CHECKSUM FROM DISK
            R5 = COMPUTED CHECKSUM

   THIS ERROR WOULD OCCUR AFTER A RUN OR CHAIN COMMAND.
   THE "PROGRAM LOST-SORRY" MESSAGE WOULD APPEAR ON A
   USER'S TERMINAL.

2. R3=R5=0
   ON 11/40 WITH NON-FIS MATH PACKAGE, THE FIS INTERRUPTED.
   ON 11/45 WITH NON-FPP MATH PACKAGE, THE FPP INTERRUPTED.

3. R3="FLOAING EXCEPTION CODE" ; R5="FLOATING EXCEPTION ADDRESS"
   ON 11/45 WITH FPP MATH PACKAGE, THE "FLOATING EXCEPTION CODE"
         WAS ILLEGAL.

   THE MESSAGE "FLOATING POINT ERROR PROGRAM LOST-SORRY" SHOULD
   APPEAR AT SOME USER'S TERMINAL AT THE TIME OF THIS ERROR

   FOR CASES 2. AND 3..

   NOTE: SEE 11/45 PROCESSOR HANDBOOK FOR DETAILED FPP INFORMATION.

   * USER INSTRUCTION SPACE ADDRESS REGISTER (PAR)
     UISAR0 THROUGH UISAR3 ALWAYS MAP USER IMAGE.
     UISAR4 THROUGH UISAR7 ALWAYS MAP RUN-TIME SYSTEM, STARTING
     WITH 7 FOR THE TOP SEGMENT.

2. DH11 ERRORS

   THE INFORMATION DISPLAYED INCLUDES THE FOLLOWING:

   JOB
   PSW
  *CH,EB
   CSR ADDRESS
   CSR CONTENTS
   SILO STATUS
   LINE PARAMETER REGISTER
   CURRENT ADDRESS REGISTER
   BYTE COUNT REGISTER
   BUFFER ACTIVE REGISTER

   THE FOLLOWING EVALUATION PROCEDURE SHOULD BE FOLLOWED:

   EXAMINE CSR CONTENTS.

      IF BIT 14 IS SET THEN THIS WAS A SILO OVERFLOW, THE
      INFORMATION IN CH,EB IS MEANINGLESS.

      IF BIT 10 IS SET THEN NON-EXISTENT MEMORY, THE INFORMATION
      IN CH,EB IS MEANINGLESS.

OTHERWISE, CH.ER SHOULD BE EXAMINED.
IF BIT 14 IS SET THEN THERE WAS A DATA OVERRUN.
IF BIT 12 IS SET, INDICATING PARITY ERROR THE LINE PARAMETER REGISTER SHOULD BE EXAMINED FOR BIT 4 WHICH IS THE PARITY ENABLE INDICATOR. THIS CASE SHOULD NEVER OCCUR FOR RSTS/E DOES NOT ENABLE PARITY ON THE DH11. HOWEVER, THIS CONDITION HAS OCCURRED. THE CAUSE WAS SOLDER SPLASH.

*CH.ER IS AN ABBREVIATION FOR CHARACTER - ERROR BIT REGISTER

****

IN ORDER TO ANALYZE SYSTEM CRASHES PROPERLY, THE CRASH DUMP FACILITY MUST BE ENABLED DURING THE START OPTION, AND THE CONSOLE SWITCHES MUST LEFT IN AN UPWARD POSITION.

WHEN THE RSTS/E SYSTEM CRASHES, APPROXIMATELY 12K OF CORE WILL BE WRITTEN INTO THE DISK FILE CRASH.SYS UNDER ACCOUNT [0,1].

THREE PROGRAMS MUST BE RUN TO EXTRACT THE INFORMATION NECESSAY TO PERFORM CRASH ANALYSIS. THESE PROGRAMS WHICH ARE DISTRIBUTED WITH THE SYSTEM LIBRARY ARE THE FOLLOWING:

ANALYS
ERRCRS
ERRDIS

THE EXECUTION OF THESE THREE PROGRAMS SHOULD BE INITIATED FROM THE CRASH.CTL FILE. THE FOLLOWING CRASH.CTL FILE EXAMPLE INCLUDES THE NECESSARY COMMANDS.

```
FORCE KB0: RUN $ANALYS
FORCE KB0: [0,1]CRASH.SYS
FORCE KB0: KB:
FORCE KB0: RUN $ERRCRS
FORCE KB0: TEMP.TMP
FORCE KB0:
FORCE KB0: RUN $ERRDIS
FORCE KB0: TEMP.TMP
FORCE KB0: KB:
FORCE KB0: ALL
FORCE KB0: /KILL
FORCE KB0: RUN $INIT
SEND I'M NOW ATTEMPTING TO RECOVER FROM A CRASH -
SEND THANK YOU FOR YOUR PATIENCE...
END
```

THE LAST FORCE COMMAND IN THIS EXAMPLE FORCES THE SYSTEM PROGRAM "INIT" TO RUN AGAIN, THIS TIME EXECUTING ALL THE COMMANDS IN THE START.CTL FILE. THIS ELIMINATES THE NEED FOR REPEATING THOSE COMMANDS IN THIS FILE. THE INIT PROGRAM CONTAINS TWO ENTRIES, ONE WHICH REQUIRES IT TO USE THE START.CTL FILE FOR COMMANDS AND THE OTHER WHICH CAUSES IT TO ASSUME COMMANDS FROM CRASH.CTL.

THE INFORMATION OUTPUT BY THE COMMANDS IN THIS CRASH.CTL AND THE LOAD MAPS FOR THE SYSTEM ARE THE NECESSARY TOOLS FOR ANALYZING CRASHES.

****

NOTE: JOB #'S IN THE CRASH DUMP ANALYSIS MUST BE CONVERTED FROM OCTAL TO DECIMAL AND DIVIDED BY 2 TO CORRESPOND WITH JOB #'S IN THE CRASH DUMP STATUS, WHICH IS DISPLAYED AT THE BEGINNING OF THE CRASH ANALYSIS. NUMBERS UNDER "JOB; NEXT" AND "FIJOB" ARE THE JOB # MULTIPLIED BY 2 IN OCTAL.

****

1. ERROR CODE - THE FIRST ITEM TO EXAMINE ON THE CRASH ANALYS OUTPUT IS THE ERROR CODE. THIS CODE SHOULD CORRESPOND TO ONE OF THE FOLLOWING.

ERROR CODE	DEFINITION
-1 (177777)	UNKNOWN VECTOR
-2 (177776)	JUMP TO 0
41	TRAP 4
42	TRAP 10
43	TRAP 250 MEMORY MANAGMENT VIOLATION
44	KERNEL SP STACK OVERFLOW
46	TRAP 114 PARITY MEMORY ERROR
0	FORCED DUMP

IF A DUMP WAS FORCED (COMPUTER HALTED AND RESTARTED WRITING OUT THE CRASH FILE) ALL INFORMATION ON THE CRASH ANALYSIS FROM "SAVED R0 TO R5" DOWN TO AND INCLUDING "USER ADDR. REGS" IS MEANINGLESS. FORCED DUMPS OF THIS TYPE DO NOT CAUSE THE NOFAIL CODE TO BE EXECUTED WHICH SAVES THIS INFORMATION. IF YOU MUST FORCE A CRASH DUMP, IT WOULD BE MORE ADVISEABLE TO HALT THE COMPUTER, EXAMINE THE CONTENT OF LOCATION 4, SET THIS ADDRESS IN THE SWITCH ADDRESSES, LOAD ADDRESS, AND HIT START.

2. PC AND PSW - THE FIRST TWO ITEMS ON THE KERNEL STACK ARE THE PC AND PSW AT THE TIME OF THE CRASH.

PC - THE LOAD MAPS SHOULD BE REFERENCED FOR MODULES AND GLOBALS WHICH MOST NEARLY CORRESPOND TO THIS PC VALUE.

PWS - THE PSW SHOULD FOLLOW THE FOLLOWING FORMAT:

```
X   XXX   X00   0XX   X0X   XXX
*******   *
   *      * 1 IF 11/45, 0 IF 11/40.
   *
   *
```

0000 CURRENT AND PREVIOUS MODE KERNEL.
KERNEL CRASH FROM AN INTERRUPT ROUTINE.

0011 CURRENT MODE KERNEL, PREVIOUS MODE USER.
KERNEL CRASH FROM INTERRUPT ROUTINE OR MONITOR.

1111 CURRENT AND PREVIOUS MODE USER. THIS WAS A USER CRASH. THE JOB # UNDER "JOBS NEXT" WAS CURRENT.

!!! RSTS/E DOES NOT USE SUPERVISOR MODE. THE PSW SHOULD !!! NEVER INDICATE SUPERVISOR MODE.

3. KERNEL SP - SHOULD ALWAYS BE LESS THAN "FISTAK". THE RSTS/E MONITOR UTILIZES TWO STACKS; ONE FOR FIP(FILE PROCESSOR), AND THE OTHER STACK FOR OTHER MONITOR OPERATION. "FISTAK" MARKS THE BEGINNING OF THE FIP STACK. "SYSTAK"+20 MARKS THE BEGINNING OF SYSTEM STACK, AND THE BOTTOM OF LIMIT OF THE FIP STACK. THE FOLLOWING HELPS DETERMINE WHAT TYPE OF OPERATION WAS BEING PERFORMED.

SP<"SYSTAK"+20(OCTAL) A KERNEL (MONITOR) OPERATION WAS IN PROGRESS. THE JOB # UNDER "JOB; NEXT" WAS THE JOB BEING PROCESSED AT THE TIME THE SYSTEM CRASHED.

SP>"SYSTAK"+20(OCTAL) A FIP(FILE PROCESSOR) OPERATION WAS IN PROGRESS. THE JOB # UNDER "FIJOB" DETERMINES WHICH JOB WAS IN A FILE PROCESSING STATE.

IF THIS CASE IS TRUE THE INFORMATION UNDER "FIRQB"(FILE REQUEST QUE BLOCK) AND/OR "XRB"(TRANSFER CONTROL BLOCK) SHOULD BE EXAMINED. (REFER TO V5 S.W.S. NOTES FOR FIRQB AND XRB FORMATS)

NOTE: "FISTAK" & "SYSTAK" ARE GLOBALS IN THE <LOWCOR> SECTION OF THE RSTS.LDA LOAD MAP.

4. KERNEL ADDR. REGISTERS - KERNEL ADDRESS REGISTERS 0 THROUGH 5 ARE USED TO MAP THE MONITOR. REGISTER 6 IS USED BY THE MONITOR TO MAP A PORTION OF THE USER IMAGE FOR TRANSFER OF THE DECTAPE BUFFER FROM MONITOR TO USER SPACE. REGISTER 7 IS USED TO MAP THE I/O PAGE.

5. USER ADDR. REGISTERS - USER INSTRUCTION SPACE ADDRESS REGISTERS 0 THROUGH 3 ARE USED TO MAP THE USER IMAGE. REGISTERS 4 THROUGH 7 MAP THE RUN-TIME SYSTEM.

6. KERNEL DESC. REGS
USER DESC. REGS. - REFER TO PAGE 2-9 OF THE KT11 MEMORY MANAGEMENT UNIT MAINTENANCE MANUAL (DEC-11-HKTB-D).

FORCED SYSTEM CRASHES

TWO FORCED CRASHES EXIST IN THE RSTS/E MONITOR.

1. IN THE MODULE FIP, AT "FIPERR"+34

THE DISK DRIVER RETURNING AN ERROR DURING ONE OF THE FOLLOWING:

1. WINDOW TURN OR DISK FILE EXTEND.
2. READING NON-RESIDENT CODE NECESSARY FOR A MAGTAPE OPEN, CLOSE, ZERO, OR CATALOG.
3. READING NON-RESIDENT CODE NECESSARY FOR A DECTAPE OPEN, CLOSE, ZERO, OR CATALOG.

THE DISK DRIVER ALWAYS RETURNS THE ERROR BECAUSE THE DISK DRIVER IS PERFORMING THE DISK OPERATIONS WHICH HANDLE MAGTAPE AND DECTAPE OVERLAYS WHICH RESIDE ON DISK.

FIVE ATTEMPTS ARE MADE BEFORE FORCING THE CRASH.
THESE ERRORS SHOULD BE LOGGED IN THE ERROR LOG FILE.

2. IN MODULE MTA, AT <MTACHK>+16

IF THE CURRENT MEMORY ADDRESS (MTCMA) OR THE WORD COUNT (MTBRC) IS CLOBBERED, THE SYSTEM FORCES A CRASH.

****BOTH OF THE FORCED CRASHES DESCRIBED ABOVE WILL SHOW UP AS TRAPS THRU 4, .

THE FOLLOWING INSTRUCTION IS ALWAYS USED TO FORCE A CRASH.

```
(PC-4)  5737
(PC-2)     1
```

2780

DESCRIPTION: 2780 Information

RSTS/2780 is a software package which enables a suitably equipped RSTS/E system to act as a very powerful remote job entry (RJE) terminal. Using RSTS/2780, RSTS/E users may queue data and/or job control files for transmission to one of IBM's remote job entry packages (HASP, ASP, DOS/POWER or RJE), or to another PDP-11 based system. RSTS/2780 accomplishes this by appearing to a point-to-point synchronous data-link as an IBM 2780 Model 1 Data Transmission Terminal. The 2780 is supported by all of the above-mentioned IBM programs, as well as being able to communicate with another 2780. Thus, RSTS/2780 emulating the 2780 can communicate with the other PDP-11 based 2780 emulators (Core-2780, DOS-2780, RSX-11D/2780 or another RSTS/2780).

RSTS/2780 operates at data rates of 4800 bits per second over switched or private facilities using Bell System series 208 modems or their equivalents, or at 2400 bits per second on private lines and 2000 bits per second on switched lines using Bell System series 201 modems or equivalents. The Bell System 801 Autocall unit is not supported.

When communicating with an IBM system RSTS/2780 will operate using either a 2701 Data Adapter, a 2703 Transmission Control Unit, a 3704 or 3705 Transmission Controller, or a System/370 Model 135 Integrated Communications Adapter. The RSTS/2780 system itself uses either the DU11 or the DP11 synchronous interfaces.

RSTS/2780 consists of two software components; a driver for the DP11 or DU11 synchronous data-link interface which is linked into the RSTS/E Monitor, and a control program which manages the flow of data to and from the link. The control program, written in the BASIC-Plus language, provides both interactive and spooled modes of user interface. In interactive mode the system operator establishes the data-link and specifies directly the files to be transmitted to the remote system, as well as destination of received data. In queued mode, the control program transmits files as they are queued by RSTS/E users using the standard RSTS/E CUSP. Received data is stored in operator-named RSTS/E files; a seperate RSTS/E file for each received data file. The system automatically updates the name extension for each received file.

RSTS/2780 communicates with an IBM system in a standard 2780 format, transmitting logical records of 80 or fewer characters and receiving logical records of up to 132 characters. Blocks may be up to 400 characters in length. When RSTS/2780 is communicating with another DIGITAL 2780 emulator, a general mode of operation permits transmission as well as reception of 132 character logical records. In either case, both transmitted and received files may be considered to be ASCII, in which case an EBCDIC to ASCII (or vice versa) translation is performed, or binary, in which case the data is stored or transmitted untranslated. All data-link control characters are supplied and stripped automatically by the RSTS/2780 software.

RSTS/E 2780

RSTS/2780 enables RSTS/E (V05B) users to queue data and or job control files for transmission to one of IBM's Remote Job Entry packages (HASP, ASP, DOS/POWER, or RJE) or to another PDP-11 based 2780 system.

RSTS/2780 can be used in either an interactive or a spooled mode. In the interactive mode, the system operator establishes the data link and specified directly the files to be transmitted as well as the destination of the received data. In the queued mode, the control program transmits files as they are queued by RSTS/E users using the standard RSTS/E CUSP. Received data is stored in operator-named RSTS/E files.

PDP-11 - A valid RSTS/E or RSX-11D configuration with at least 48K words of memory, plus . . .

- o a synchronous line interface (DP11 or DU11)
- o a communications arithmetic element (KG11-A)
- o a system clock (KW11-L or KW11-P)

IBM - Either a 2701 Data Adapter, a 2703 Transmission Control Unit, a 3704 or 3705 Transmission Controller, or a System/370 Model 135 Integrated Communications Adapter.

Modems - Bell 208 or equivalent (4800 bps over switched or private facilities) or
Bell 201 or equivalent (2400 bps over private lines or 2000 bps over leased lines).

Questions and Answers

Answers to some of the more frequently asked questions on the two new 2780 packages are presented below:

Question: Can one Remote Computer System like RSTS-2780 communicate with another; say, DOS-2780?

Answer: Yes.

Question: Can multiple Remote Computer System be configured in networks?

Answer: No. A network has many implications which go far beyond a 2780 transmitting data. Just because one RCS can communicate with another does not mean that you can set up a network. Be careful not to associate these products with networks.

Question: I have a customer who would like RSTS with HASP; will RSTS-2780 suffice?

Answer: Be careful. People frequently confuse terminology. First be aware that HASP is a software system which is used in IBM 360 and IBM 370 systems. HASP software can support 2780 terminals and HASP Workstations. 2780 terminals and HASP Workstations are not the same. We have a HASP Workstation package (CORE-HASP) but it has no relationship to the 2780 packages. RSTS-2780 is a 2780 package.

Questions and Answers (Cont'd)

Question: DEC literature specifies Bell modems for use with the 2780 products. What if a user wants to use other types of modems?

Answer: We test our products with Bell modems. Since there are so many other modems in the marketplace, it is impossible for us to run tests with all of them. Our literature normally states "Bell Model (xxx) or equivalent." If the customer can find the appropriate equivalent fine. However, we can't help him if he runs into problems.

Question: Can a customer order a 2780 configuration without a line printer?

Answer: It is possible to receive data directly onto disk with the DOS, RSK-11D, and RSTS/E 2780 packages so it's possible that a customer may want to order a system without a line printer.

We do not recommend eliminating the line printer. It will just make support more difficult; running diagnostics and tests will be complicated. If the customer insists on no line printer, then he should be made well aware of the fact that he will incur some inconvenience.

Question: I've had difficulty getting programming documentation for the 2780 products?

Questions and Answers (Cont'd)

Answer: The 2780 Remote Computer Systems do not require programming manuals. Each package only requires an operator's manual which shows how to run the system. The documents that the customer should normally get are: the sales brochure and the software product description. An operator's manual is shipped with the system.

Question: Is the operation of RSTS/E-2780 identical to RSX-11D/2780?

Answer: No. RSTS/E-2780 can be used in two modes. In one mode, the RSTS operator controls the transmission. In the other mode, RSTS/E users can "queue" files on disk for transmission. RSX-11D/2780 only provides operator mode.

Question: Are there any subtleties one should be aware of?

Answer: The new 2780 packages are not complex. However, there are subtleties that can vary from application to application. We recommend that you consult someone who knows the 2780 packages well whenever you have a prospect.

Ordering Information

RSTS/2780	QPD10-AC, AD, AE, AF*	$4,000
	includes single use license, binaries manuals, and installation.	

DH11 HARDWARE/SOFTWARE

RSTS/E DATA COMMUNICATIONS

RSTS/E data communications systems have been designed for flexibility and ease of use. Line speeds of 10, 15, and 30 characters per second can be used remotely over dedicated or dial-up lines and speeds up to 960 characters per second can be used local to the computer system. RSTS/E supports only full duplex.

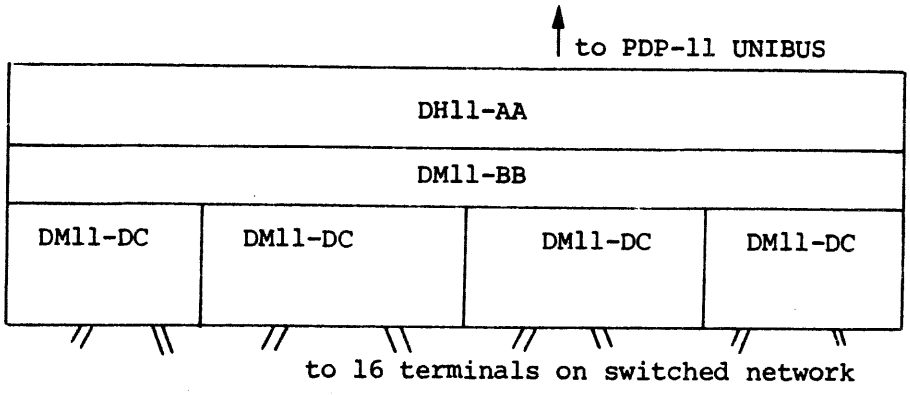
Several choices are available to configure DEC supplied hardware for connecting telephone equipment into the system. Among those choices are the DL11 and DH11 communications hardware. Economic trade-offs can be made between these two types of devices. DL11's have the advantage of small incremental growth costs, are slightly less expensive except in installation increments of 16 lines, but do not provide the multiple line speed capability on each line as the DH11's do.

Specific hardware attributes of the DH11 and DL11 equipment supported by RSTS are given below.

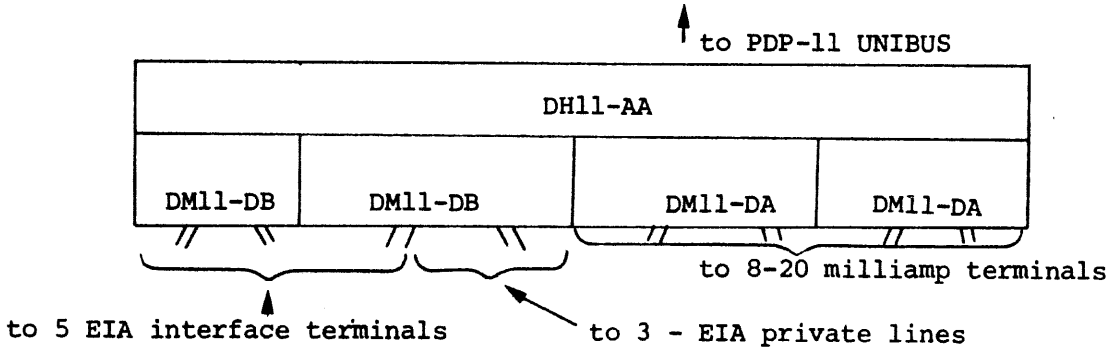
Unit	Description
DL11A	Current loop (20 ma) Serial Line Interface having one baud rate of up to 9600.
DL11B	EIA Serial Line Interface for connection to modem having one baud rate up to 2400 baud locally or up to 300 baud in private line remote communications
DL11E	EIA Serial Line Interface with modem control and supporting one rate up to 300 baud on the switched telephone network.
DH11-AA	Multiplexer which connects a PDP-11 computer with up to 16 asynchronous serial communications lines with individually programmable parameters. Remote lines have baud rates of 110, 150, and 300 baud. Local lines operate at up to 9600 baud.
DM11-BB	Modem Control Multiplexer which provides control to interface with up to 16 data sets.
DM11-DC	Provides lines for conditioning four EIA compatible lines with modem control. (EIA Modem)
DM11-DB	Line Adapter which implements four EIA lines for private line or local terminal connection. (EIA Local or Non-Dial-up Modem Control or private Line Applications.)
DM11-DA	Line Adapter which implements four 20 milliamp. terminals. (20 Mill local)

Typical DH11 configurations might be as follows:

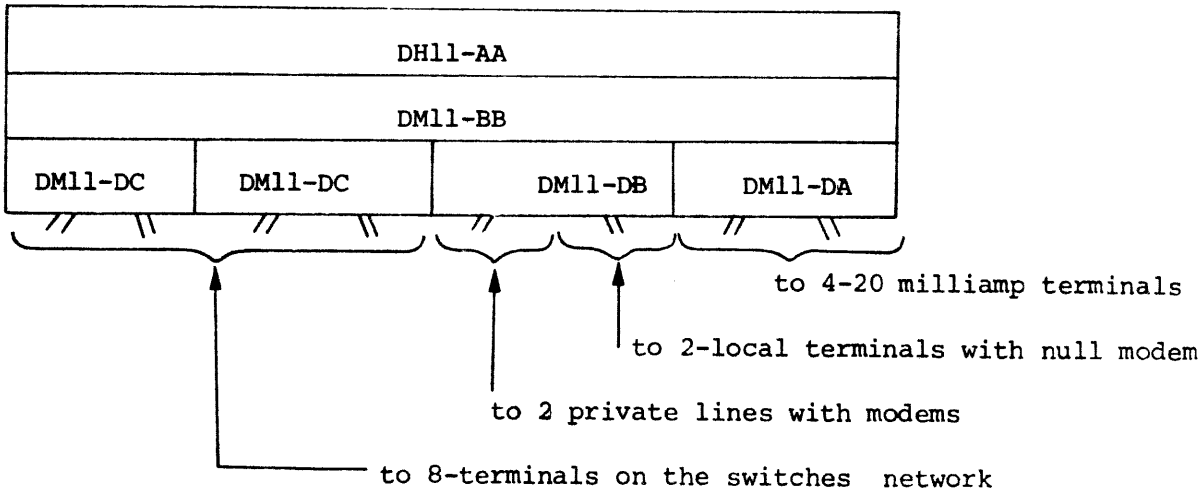
16 terminals connected to the switched network



16 terminals connected to local terminals



16 terminals connected to switched network as well as some local connections



Remember that the load imposed on the system by a terminal outputting at 9600 Baud is about the same as the load imposed by 32 terminals -- all outputting at 300 Baud. Considering that most input takes place no faster than 50 Baud (typing speed), five terminals running interactively at 9600 Baud in normal Interactive use probably won't significantly degrade system performance, although 9600 Baud output does place an instantaneous, heavy load on the system.

Few useful applications of terminals outputting at 9600 Baud exist. It is more reasonable to output to a terminal at 1200 Baud or less.

DH11 PROGRAMMABLE ASYNCHRONOUS MULTIPLEXER

FROM: Dimitri Dimancesco
x3553 MA

DECcomm recently announced two new versions of the DH11 programmable 16-line multiplexer. The new versions designated DH11-AE and DH11-AD are more compact and are priced lower than the existing DH11.

You can now order:

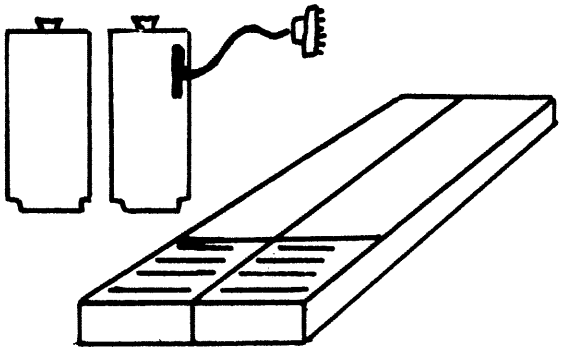
DH11-AA - This is the original version of the DH11. It requires DM11 type line adapters depending on the type of connection (DM11-DA for 20 mAmp, DM11-DB for private phone lines, DM11-DC/DM11-BB for switched telephone network connections).

The DH11-AA and appropriate DM11 modules should still be ordered when configuration requires a combination of 20 mAmp, private, or dial-up lines.

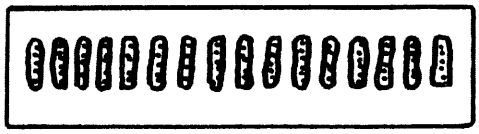
DH11-AD - New. Interfaces PDP-11 to 16 EIA/CCITT switched network lines. Includes modem control. Uses DJ11 type distribution panel, therefore cables must also be ordered for some configurations.

DH11-AE - New. Same as DH11-AD except no modem control.

DM11 Type Modules (include cables)



DH11-AA Distribution Panel
requires DM11 modules



DJ11 type interface panel
used with DH11-AD and -AE
(no cables provided)

DH11

		Prereq.	Price	Disc Type	Field Maint.	Install.	Mounting	Bus Load	Amp +5V
DH11-AA(C)	- Programmable Asynch. 16-line multiplexer and mounting panel. Includes space for up to 4 DM11 Line Adapters (16 lines).	11/CPU	$4,400	2	32	175	2 SU/ Sm Pan	2	4.7
DM11-DA	- Line Adapter for 4 20 mAmp terminals. Includes cables.	DH11-AA or -AC	170	2	5	40	*		
DM11-DB	- Line Adapter for 4 EIA/CCITT lines. Includes modem cables.	DH11-AA or -AC	485	2	5	40	*		
DM11-DC	- Line Adapter for 4 EIA/CCITT lines. When used with DM11-BB, provides modem control. Includes modem cable.	DH11-AA or -AC and DM11-BB	860	2	11	40	*		
DM11-BB	- 16-line modem control for program operation of control leads for 103, 202 or equivalent data sets.	DH11-AA or -AC and DM11-DC	1,295	2	19	80	*	1	2.8

*Mounts in DH11-AA or AC distribution panel

RSTS V5 DH11/DM11B HANDLER

The DH11/DM11B service for RSTS V5 provides essentially the same services for the RSTS user which are provided by the DC11 service. These services include support of the variable speed capabilities and auto-answer modem facility.

Relevent documents explaining the hardware are the DH11 Engineering Specification and "DM11-BB Modem Control Manual" (DEC-11-HDMBA-B-D).

The DH11 has 8 contiguous I/O page-addressable registers which are:

CSR	Interrupt control, extended memory, line select
NRC	Received character, line number and char. status
LPR	Line parameters: speeds, etc.
CAR	Current address
BC	Byte count
BAR	Buffer active: 1 bit per line
BCR	Break control: 1 bit per line
SSR	Silo status: silo fill level, silo alaram level

The error logging module is used to handle certain exceptions which are caught by the DH11 service. These include:

Non-existent memory referenced by NPR transfer
Silo overflow
Data overrun
Break
Parity error

Non-existent memory errors should never occur. If they do, then there is either a RSTS software problem or the hardware is malfunctioning.

Silo overflow errors occur when more than 64 received characters have accumulated in the silo, and indicate that RSTS is receiving more characters than the software can process.

Data overruns are generated when a received character is lost due to silo overflow.

A break condition is generated by reception of a break signal on a line, break errors are no cause for alarm, as the break signal can be generated by the keyboard operator on many terminals.

Parity errors would be generated by reception of characters with improper parity. However, since the parity sensing circuitry is disabled when RSTS is running, these errors should not occur.

The DH11 is operated with the following conditions set up:

XMIT+NEM INT ENB	ON
SILO OVERFLOW INT ENB	ON
RECEIVER INT ENB	ON
EXTENDED MEMORY BITS	Ø

AUTO-ECHO	OFF
HALF-DUPLEX	OFF
TRANSMIT SPEED	=RECEIVER SPEED — Not Always
PARITY ENABLE	OFF
TWO STOP BITS	ON — 2741 is different
CHAR LENGTH	8 BITS
BREAK CONTROL	NOT USED
SILO ALARM LEVEL	Ø

Since the small buffer pool and indeed, the whole V5 monitor has its virtual address equal to its physical address, the extended memory bits are always zero. The silo alarm level is set at zero so that a receiver interrupt will be generated on reception of a single character.

The DH11 service depends on the standard RSTS terminal tables having entries as follows:

TTILST	DH11 CSR	(Input side for KL11's)
TTOLST	DM11BB CSR	(Output side for KL11's)
TTSLST	Line parameter	(Word)
	Subline number	(Word)
	Same as KL11	(Byte)
	Same as KL11	(Byte)
	Same as KL11	(Byte)
	Same as KL11	(Byte)

The DH11 has two interrupt vectors, one for XMIT and another for receive. These vectors are located in the floating vector space.
Recieve Interrupts are aimed at DHINT
XMIT Interrupts are aimed at DHTI

The DH11 receiver interrupt processor will empty the silo completely before exiting. The received characters are processed by the standard TTY service code to move the received character to a small buffer.

The DH11 transmitter interrupt processor will check each DH11 line (for which a DDB has been allocated) for pending output and zero byte count before exiting. When a subline is found with a zero byte count and with pending characters in its small buffer chain all the characters present in the leading small buffer are sent out via the DH11. Thus the NPR capability of the DH11 is utilized to send up to 30 characters per interrupt.

The DM11BB has two I/O page addressable registers:

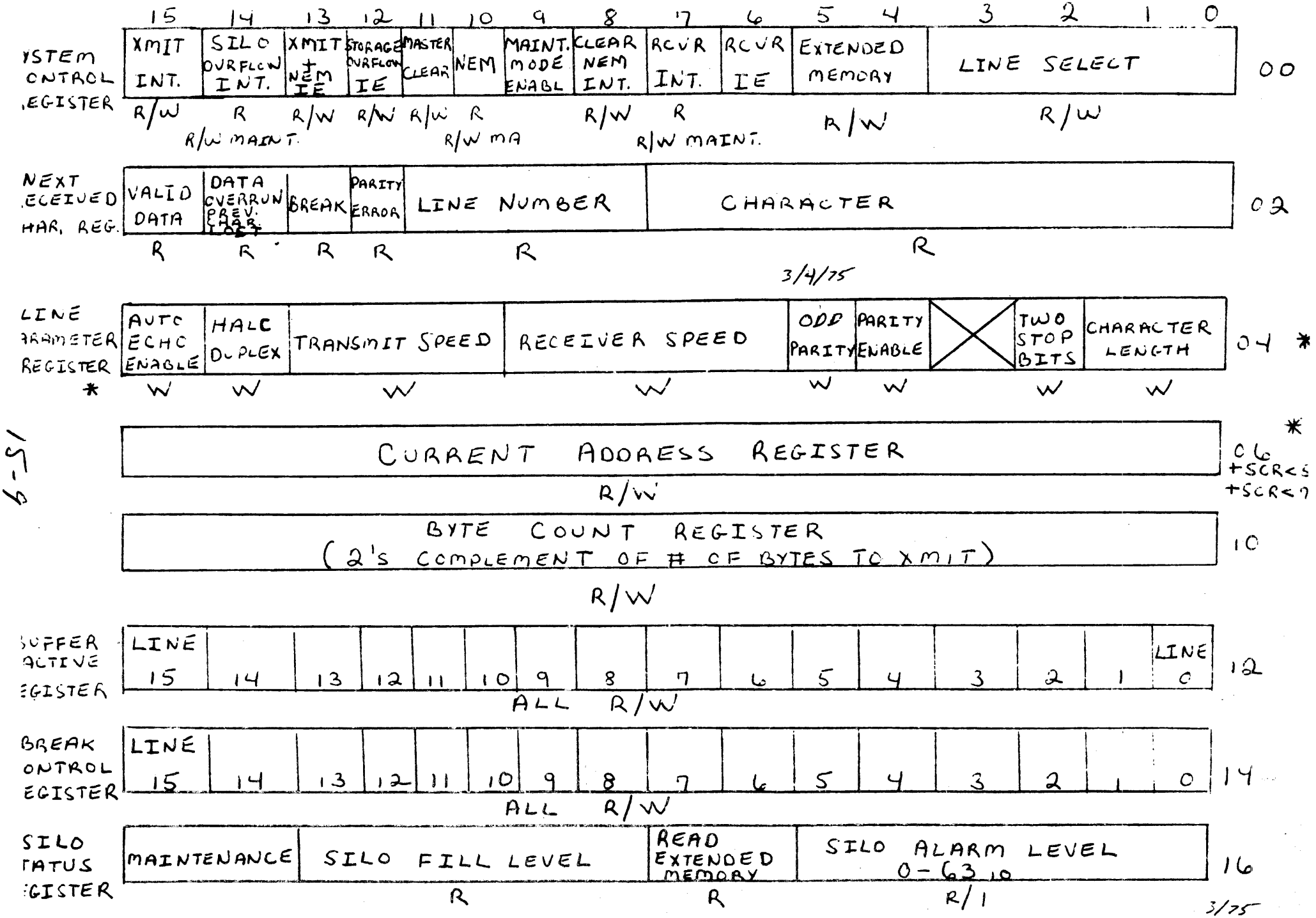
CSR Interrupt and scan control
Line status

The DM11BB is operated with the following conditions set up:

INT ENB ON
SCAN ENB ON

The treatment of DM11B interrupts is essentially analogous to the treatment of the corresponding conditions for the DC11. The modem clock service which runs once per second is used to detect lines on which carrier has been lost and lines for which carrier has not appeared after ring and anser. There is at present no "hung line" code for DH11 lines, since this condition has not yet been encountered.

DH11


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
SYSTEM CONTROL REGISTER
XMIT INT.
SILO OVRFLOW INT.
XMIT + NEM IE
STORAGE OVRFLOW IE
MASTER CLEAR
NEM
MAINT. MODE ENABL
CLEAR NEM INT.
RCVR INT.
RCVR IE
EXTENDED MEMORY
LINE SELECT
00
R/W R R/W R/W R/W R R/W R R/W
R/W MAINT. R/W MA R/W MAINT. R/W
NEXT RECEIVED CHAR. REG.
VALID DATA
DATA OVERRUN PREV. CHAR. LOST
BREAK
PARITY ERROR
LINE NUMBER
CHARACTER
02
R R R R R R
3/4/75
LINE PARAMETER REGISTER *
AUTO ECHO ENABLE
HALF DUPLEX
TRANSMIT SPEED
RECEIVER SPEED
ODD PARITY
PARITY ENABLE
TWO STOP BITS
CHARACTER LENGTH
04 *
W W W W W W W W
CURRENT ADDRESS REGISTER
06 *
+SCR<5
+SCR<7
R/W
BYTE COUNT REGISTER
(2's COMPLEMENT OF # OF BYTES TO XMIT)
10
R/W
BUFFER ACTIVE REGISTER
LINE 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 LINE 0
12
ALL R/W
BREAK CONTROL REGISTER
LINE 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
14
ALL R/W
SILO STATUS REGISTER
MAINTENANCE
SILO FILL LEVEL
READ EXTENDED MEMORY
SILO ALARM LEVEL 0-63 10
16
R R R/1
3/75

VT05/VT50

VT05 DIRECT CURSOR ADDRESSING

Direct cursor addressing of the VT05 display keyboard is done by using PRINT statements of the form

```
PRINT #N%, CHR$(X%);
```

where X% is one of the following codes. The ; following the argument is needed to prevent automatic carriage-return. The codes are as follows:

8%	backspace cursor
11%	cursor down (line feed)
14%	direct cursor control (see below)
24%	cursor right one position
25%	cursor left (don't use on VT05)
26%	cursor up
28%	home down (don't use on VT05)
29%	home up (move cursor to upper left-hand corner)
30%	clear to end of line
31%	clear to end of screen

Direct addressing is established as follows:

if Y% is the line number (Y-axis)

and X% is the column number (X-axis)

and the upper left-hand corner is (1,1)

```
PRINT #N%, CHR$(14%); CHR$(31%+Y%); CHR$(31%+X%);
```

positions the cursor to (X%,Y%)

Note the following functions:

```
2000 DEF FNQ2 (X%,Y%)    :
        PRINT #11%, CHR$(14%); CHR$(31%+Y%); CHR$(31%+X%);
2010 FNEND ' position cursor to (X%,Y%)
```

```
21ØØ DEF FNQ3(X%,Y%)   :
        PRINT #11%, CHR$(14%); CHR$(31%+Y%); CHR$(31%+X%);
                    CHR$(31%);
211Ø FNEND ! position cursor to (X%,Y%) and clear to e.o.s.
22ØØ DEF FNQ4(X%,Y%)   :
        PRINT #11%, CHR$(14%); CHR$(31%+Y%); CHR$(31%+X%);
                    CHR$(30%);
221Ø FNEND ! position cursor to (X%,Y%) and clear to e.o.l.
```

The following table will be useful when writing programs that interface with the VT50. It details the processing of each 7-bit ASCII character received at the terminal.

When the VT50 is in...

...Normal Mode the next ASCII character received is treated as data.

...Escape Mode the next ASCII character received is treated as a command.

Octal Code	Char	Action Taken	Resulting Mode	Action Taken	Resulting Mode
000	NUL	None	Normal	None	Escape
001	SOH	None	Normal	None	Escape
002	STX	None	Normal	None	Escape
003	ETX	None	Normal	None	Escape
004	EOT	None	Normal	None	Escape
005	ENQ	None	Normal	None	Escape
006	ACK	None	Normal	None	Escape
007	BELL	Rings Bell.	Normal	Rings Bell.	Escape
010	BS	Backspaces Cursor.	Normal	Backspaces Cursor.	Escape
011	HT	Horizontal Tab Moves cursor to the next tab stop. Tab stops are set every eight spaces to the 72nd character position. After the 72nd position, TAB moves the cursor to the right one position.	Normal	Horizontal Tab Moves cursor to the next tab stop. Tab stops are set every eight spaces to the 72nd character position. After the 72nd position, TAB moves the cursor to the right one position.	Escape
012	LF	Moves Cursor down one line and scrolls if required.	Normal	Moves Cursor down one line and scrolls if required.	Escape
013	VT	None	Normal	None	Escape
014	FF	None	Normal	None	Escape
015	CR	Moves Cursor to left margin of current line.	Normal	Moves Cursor to left margin of current line.	Escape
016	SO	None	Normal	None	Escape
017	SI	None	Normal	None	Escape
020	DLE	None	Normal	None	Escape
021	DC1	None	Normal	None	Escape
022	DC2	None	Normal	None	Escape
023	DC3	None	Normal	None	Escape
024	DC4	None	Normal	None	Escape
025	NAK	None	Normal	None	Escape
026	SYN	None	Normal	None	Escape
027	ETB	None	Normal	None	Escape
030	CAN	None	Normal	None	Escape
031	EM	None	Normal	None	Escape
032	SUB	None	Normal	None	Escape

When the VT50 is in...

...Normal Mode the next ASCII character received is treated as data.

...Escape Mode the next ASCII character received is treated as a command.

Octal Code	Char	Action Taken	Resulting Mode	Action Taken	Resulting Mode
033	ESC	Sets terminal in Escape Mode.	Escape	Sets terminal in Normal Mode.	Normal
034	FS	None	Normal	None	Escape
035	GS	None	Normal	None	Escape
036	RS	None	Normal	None	Escape
037	US	None	Normal	None	Escape
040	Space	Displayed	Normal	None	Normal
041	!	Displayed	Normal	None	Normal
042	"	Displayed	Normal	None	Normal
043	#	Displayed	Normal	None	Normal
044	$	Displayed	Normal	None	Normal
045	%	Displayed	Normal	None	Normal
046	&	Displayed	Normal	None	Normal
047	'	Displayed	Normal	None	Normal
050	(	Displayed	Normal	None	Normal
051	)	Displayed	Normal	None	Normal
052	*	Displayed	Normal	None	Normal
053	+	Displayed	Normal	None	Normal
054	,	Displayed	Normal	None	Normal
055	−	Displayed	Normal	None	Normal
056	.	Displayed	Normal	None	Normal
057	/	Displayed	Normal	None	Normal
060	0	Displayed	Normal	None	Normal
061	1	Displayed	Normal	None	Normal
062	2	Displayed	Normal	None	Normal
063	3	Displayed	Normal	None	Normal
064	4	Displayed	Normal	None	Normal
065	5	Displayed	Normal	None	Normal
066	6	Displayed	Normal	None	Normal
067	7	Displayed	Normal	None	Normal
070	8	Displayed	Normal	None	Normal
071	9	Displayed	Normal	None	Normal
072	:	Displayed	Normal	None	Normal
073	;	Displayed	Normal	None	Normal
074	<	Displayed	Normal	None	Normal
075	=	Displayed	Normal	None	Normal
076	>	Displayed	Normal	None	Normal
077	?	Displayed	Normal	None	Normal
100	@	Displayed	Normal	None	Normal
101	A	Displayed	Normal	Moves Cursor up one line.	Normal
102	B	Displayed	Normal	None	Normal

When the VT50 is in...

...Normal Mode the next ASCII character received is treated as data.

..Escape Mode the next ASCII character received is treated as a command.

Octal Code	Char	Action Taken	Resulting Mode	Action Taken	Resulting Mode
110	H	Displayed	Normal	Moves Cursor to home position.	Normal
111	I	Displayed	Normal	None	Normal
112	J	Displayed	Normal	Erases line from Cursor to bottom of screen.	Normal
113	K	Displayed	Normal	Erases screen from Cursor to right margin.	Normal
114	L	Displayed	Normal	None	Normal
115	M	Displayed	Normal	None	Normal
116	N	Displayed	Normal	None	Normal
117	O	Displayed	Normal	None	Normal
120	P	Displayed	Normal	None	Normal
121	Q	Displayed	Normal	None	Normal
122	R	Displayed	Normal	None	Normal
123	S	Displayed	Normal	None	Normal
124	T	Displayed	Normal	None	Normal
125	U	Displayed	Normal	None	Normal
126	V	Displayed	Normal	None	Normal
127	W	Displayed	Normal	None	Normal
130	X	Displayed	Normal	None	Normal
131	Y	Displayed	Normal	None	Normal
132	Z	Displayed	Normal	None	Normal
133	[	Displayed	Normal	Enables Hold Screen Mode. (See page 19 .)	Normal
134	\	Displayed	Normal	Disables Hold Screen Mode. (See page 19 .)	Normal
135	]	Displayed	Normal	All lines from top of the screen up to and including cursor line are printed.	Normal
136	^	Displayed	Normal	Enables Auto Print with top line printed first.	Normal

When the VT50 is in...

...Normal Mode the next ASCII character received is treated as data.

...Escape Mode the next ASCII character received is treated as a command.

Octal Code	Char	Action Taken	Resulting Mode	Action Taken	Resulting Mode
137	—	Displayed	Normal	Disables Auto Print after current lines are copied	Normal
140	\	Displayed as @	Normal	None	Normal
141	a	Displayed as A	Normal	None	Normal
142	b	Displayed as B	Normal	None	Normal
143	c	Displayed as C	Normal	None	Normal
144	d	Displayed as D	Normal	None	Normal
145	e	Displayed as E	Normal	None	Normal
146	f	Displayed as F	Normal	None	Normal
147	g	Displayed as G	Normal	None	Normal
150	h	Displayed as H	Normal	None	Normal
151	i	Displayed as I	Normal	None	Normal
152	j	Displayed as J	Normal	None	Normal
153	k	Displayed as K	Normal	None	Normal
154	l	Displayed as L	Normal	None	Normal
155	m	Displayed as M	Normal	None	Normal
156	n	Displayed as N	Normal	None	Normal
157	o	Displayed as O	Normal	None	Normal
160	p	Displayed as P	Normal	None	Normal
161	q	Displayed as Q	Normal	None	Normal
162	r	Displayed as R	Normal	None	Normal
163	s	Displayed as S	Normal	None	Normal
164	t	Displayed as T	Normal	None	Normal
165	u	Displayed as U	Normal	None	Normal
166	v	Displayed as V	Normal	None	Normal
167	w	Displayed as W	Normal	None	Normal
170	x	Displayed as X	Normal	None	Normal
171	y	Displayed as Y	Normal	None	Normal
172	z	Displayed as Z	Normal	None	Normal
173	{	Displayed as □	Normal	None	Normal
174	|	Displayed as /	Normal	None	Normal
175	}	Displayed as □	Normal	None	Normal
176	~	Displayed as >	Normal	None	Normal
177	Delete	None	Normal	None	Escape

Typing the key labeled	Transmits the following 7-bit ASCII code	Typing the key labeled	Transmits the following 7-bit ASCII code
A	101	T	124
B	102	U	125
C	103	V	126
D	104	W	127
E	105	X	130
F	106	Y	131
G	107	Z	132
H	110	BACK SPACE	010
I	111	TAB	011
J	112	LF	012
K	113	RETURN	015
L	114	ESC	033
M	115	SPACE BAR	040
N	116	\	134
O	117	DELETE	177
P	120	BREAK	Does not transmit an ASCII code. Acts as a forceable interrupt (see page 19).
Q	121		
R	122		
S	123		

The keys listed above are uneffected by the SHIFT key. For example, typing both the "A" key and holding down the SHIFT key and typing "A" transmit ASCII 101.

The following keys are effected by the "SHIFT" key.

Typing the key labeled	Unshifted transmits the 7-bit ASCII code	With the SHIFT key held down transmits the 7-bit ASCII code
1	061	041 (!)
2	062	100 (@)
3	063	043 (#)
4	064	044 ($)
5	065	045 (%)
6	066	136 (^)
7	067	046 (&)
8	070	052 (*)
9	071	050 (( )
0	060	051 ( ))
-	055	137 (_)
=	075	053 (+)
[	133	135 (])
;	073	072 (:)
'	047	042 (")
,	054	074 (<)
.	056	076 (>)
/	057	077 (?)

Holding down the CTRL key affects the ASCII signal transmitted by the next key typed. It forces bits 7 and 6 to 0. For example, holding down the CTRL key and typing the letter G (107) forces 107 to be converted to 007 (BEL). The one exception is holding down the CTRL key and typing the BREAK key. This reinitializes the terminal (see page 19).

The following keys do not transmit any signal outside of the terminal. They are used to direct internal terminal activities.

Typing the key labeled...	Unshifted directs the terminal to...	With the SHIFT key held down, directs the terminal to...
CONT.	Display (scroll) one new line when the terminal is in Hold-Screen Mode (see page 19).	Display (scroll) 12 new lines when the terminal is in Hold-Screen Mode (see page 19).
PRINT	Produce a hard copy image of the entire screen.	Compliments Auto-Print Mode (see page 20).

Some Pointers on RSTS/E

Hardware Problems

or

What To Do If The Diagnostics Run

& RSTS Doesn't

Confidential - NOT For Customer Use

Gary Alles

Sept. 9, 1974

Confidential

Notes on RSTS-E System Hardware Configuration

1. A KW11-L is less expensive and more reliable than a KW11-P.

2. An RP03/RP11-C is bigger, faster, and more reliable than an RK05. If your customer plans to expand, start him out with an RP.

3. An RP03, only system must be sold with TM11/TU10 for proper file backup. An RK/RP system should have tape.

4. Mos Memory helps to increase system thruput significantly, but not advised until purchasing memory from 88K, because core is cheaper. Mos is now much more reliable than it was in the early history of the 11/45.

5. FPP when installed on 11/45's helps thruput of Math Package routines. You can see the differance.

6. The DH11 is a much more desireable TTY interface compared to using several DL11's. It costs far fewer bus loads, and requires relatively little care for it's driver.

7. It is possible to get an 8K user space in a 32K system by leaving out most of the basic Plus features but the spooling programs and a few other cups such as UT5DPY require 12K user space to run. Unless you are very certain of your customers application, avoid 32K systems. It is far wiser to start your customer out with 48K core (3) MF11UP's. Parity is now required.

8. It is cheaper to buy 32K core than an RS03. Even if your system has exceptionally high Disk activity already, it is probably wiser to purchase core as opposed to swapping disk. 32K MF11UP=$12.6K

9. 1 RK11 D + 5 RK05's ≈ 31,400 = 6M wds. slow, unreliable
   1 RP11 C, + 1 RP03 ≈ 31,880 = 20M wds, fast, reliable

10. 1 RS03 + controller 256K wds = $14K
    1 RS04 + controller 512K wds = $18K
    2 RS03 + controller 512K wds = $23K

11. You should have enough core to keep ½ of your jobs in core. (Assume largest size jobs.

If you encounter similar errors during SYSGEN on the RK on the same system that has errors during SYSGEN on the RF, you may very well have a UNIBUS or memory problem.

Before getting too deep in this problem, first be certain that the distribution kit is valid, and that it is being read correctly. Tape skew and/or RKØ5 head alignment may be cause of problem. If RSTS/E has been configured at another site, and it too fails suspect the hardware. There are no known bugs in the sysgen procedure other than mis-interpretation of DL11-C's and DL11-E's under automatic sysgen. If you have two different types of DOS supported disks, and sysgen fails on one but not the other, you may have found a faulty disk unit.

Make sure all known patches are correctly installed.

If you suspect memory, be sure to adjust strobe according to latest technique, not by setting strobe in the middle of the window. A minimum 50 NS window is required, or your memory should be replaced. Strobe should be adjusted 25 NS back from latest failing point in strobe window. Do not swap boards between memory sets unless you re-strobe.

Strobe normally should not be bothered with on very new systems, but older systems (Apr '74 or older) often are not found to have sufficient window sizes, and are frequently out of adjustment.

If you suspect UNIBUS, be sure BUS-loading on both sides of BUS repeater is about equal. (18 loads or less) minimize UNIBUS cable length wherever possible even if you have to reconfigure system.

Foam the UNIBUS.

Inspect bus cables very carefully for holes, sharp crimps, bubbles, and cuts.

Check UNIBUS routing. Avoid proximity to Power Supplies & Power Cords.

Be certain that BUS signal levels on both sides of BUS repeater are same or nearly so. Watch for noise.

Extensive engineering changes are being made to the BUS structure itself. BG lines are now supposed to have pull-down resistors at each interface. BUS Driver & BUS Receiver chips have been re-engineered, and are to be replaced with new chips when chips are available & conditions warrant. M920's can cause problems, two new versions are being worked on.

Preferred BUS Configuration is for systems w/o Mass. BUS Devices.

1. Processor
2. Core
3. Non Buffered NPR Devices
4. Buffered NPR Devices
5. ROMS & Communication Options

1% precision terminators are available for the Unibus

Remove unused DL11's, they cause all kinds of problems for no easily explained reason.

Traps and/or Jumps to zero have been known to be caused by floating DL11's, faulty TM11's and time-out problems in RK05's.

Vectors and addresses on DL11's are commonly incorrectly jumpered. Check priority jumper card in DL11's and make sure it has insulating tape and is properly installed.

If you have static, use H7002 or H7003 line filters.

Recheck speed setting for XTAL in DL11. Check for loose connections.

Be absolutely certain all power supply connections are tight, power supplies are not susceptible to normal vibration, (noise or spikes are not induced) and that power supplies are correctly adjusted. This can not be assumed.

Check power plug polarity and proper grounding.

Line voltages from factory to customer site can and will vary. Check for noise and transients.

Margin H720 power supplies for DC Lo sensitivity. (-15 uts). Use of a DUM is preferred when adjusting power supplies.

1. Adjust +5 until right on

2. Scope DC-LO

3. Adjust -15 until DC-LO is in middle of its possible values

Pull on all Mate-N-Lock connectors and make sure wires are in tight.

Make sure moving head Drive Packs are formatted on your machine.

Make sure air filters are clean.

Check power-fail circuitry on 11/45's.

Monitor DC-LO while system is running.

Sector Transducer and Head Alignment should be correctly adjusted.

Never slow down processor to avoid problems. This will cost much in the long run, if not cause more problems.

Make sure all 11/40 processor options are correctly jumpered.

Margin Processor.

Investigate all grounding and static problems.

50% humidity is required.

Check for temperature variations and don't mount cold packs.

Clean tape & disk drive heads.

Check tape head alignment and visimag tape. Use quality tape.

Check for any potential mechanical connection problems.

Time-dependent power problems may occur.

Check ECO levels of all options.

If your system is a year behind in ECO levels, you may well get RSTS/E to run, but the system probably won't be as reliable as a new one would be.

ODT

$ ODT. BAS
BP ODT
KERNEL ODT
INIT ODT

To Find RK05 Pack Label

Run ODT
File? OK1:
* 1014/xxxxxx % PAC<LF>
1016/xxxxxx % KID

Kernel ODT

Mem Alloc Changes? Yes
Table OPTION? ODT
Where?

OPTION: ST
↑P
BE xxxxxx
-

Init ODT

OPTION: ODT
-

xxxxxx / word
\ byte
< last explicit open
>
LF next loc
CR close
↑
@ absolute open
P proceed
B set breakpoint
G Go To

Basic Plus Patch Space? Yes, No, ODT

DATE: APRIL 20,1973

SUBJECT: RSTS ODT, WITH DISK LOOK AND NEW SYNTAX

FROM: MIKE SMITH (FORMERLY IN DEC SOFTWARE DEVELOPMENT)
(CURRENTLY FIXING CARS FOR GRINS AT:)
(TINGLE'S LOTUS CENTER, 31R SPRING ST)
(WATERTOWN, MASSACHUSETTS, 02172)

PROJECT NUMBER: P-67-07668
RSTS DEVELOPMENT
RSTS/E DEBUGGING AID

FILENAME: ODTDOC.* DOCUMENTATION
ODT.* SOURCE PROGRAM

EDIT 5: ODTDOC.5
BY: JIM MILLER (MAYNARD, 12-2, 3173)

SYSTEM 40 ID: 140,1315 ACCOUNT "BRAMHALL"

THE MATERIAL INCLUDED IN THIS DOCUMENT, LIMITED TO BUT NOT INCLUDING, CONSTRUCTION SPEEDS AND OPERATING PURPOSES IS FOR INSTRUCTION TIMES ONLY. ALL SUCH CLAIM IS MATERIAL WITHOUT NOTICE, AND IS BOUND TO CHANGE THE SUBJECT.

NOTICE ---------- NOTICE ---------- NOTICE

THE PURPOSE OF THIS ODT IS TO BE USEABLE FOR DEBUGGING RSTS/E. SOME FEATURES OF THE ODT IN REGULAR USE ON THE PDP-11 HAVE BEEN LEFT OUT WITH VERY MUCH PURPOSE. IF ANYONE WANTS TO USE THIS ODT I WILL BE HAPPY TO TELL YOU WHAT YOU NEED TO KNOW TO USE IT.

IF YOU HAVE SUGGESTIONS AND/OR IMPROVEMENTS AND I LIKE THEM I WILL PUT THEM IN. IF I DON'T LIKE THEM I WILL TELL YOU HOW TO PUT THEM IN YOUR OWN COPY OF ODT.

ALL BUGS WILL BE FIXED WITH GOOD HUMOR AND REASONABLE SPEED.

SCOPE OF THIS DOCUMENT--

THE PURPOSE OF THIS DOCUMENT IS TO DESCRIBE THE USAGE OF THE ODT THAT WAS BUILT FOR USE BY RSTS. THIS ODT HAS A DIFFERENT SYNTAX THAN THE NORMAL ODT AND HAS CERTAIN "ENHANCEMENTS" (OF COURSE!).

THIS IS NOT TO SAY THAT IS ENTIRELY DIFFERENT, BUT ENOUGH SO THAT YOU SHOULD READ AT LEAST THIS FIRST PAGE.

THE MAJOR DIFFERENCE IN THE SYNTAX IS THE USE OF THE ";" CHARACTER. IN SHORT IT IS NOT NEEDED EXCEPT TO DELIMIT MULTIPLE ARGUMENTS GIVEN TO COMMAND. YOU SHOULD THINK ABOUT THIS ONE CAUSE IT CAN BE A PAIN IF YOU TYPE EXTRA ";"'S.

ANOTHER DIFFERENCE IS IN THE WAY REGISTERS ARE OPENED. THE OPEN COMMAND IS USED TO OPEN A REGISTER AND TO RETYPE THE CONTENTS OF A REGISTER. SO IF YOU OPEN A REGISTER AS A WORD, AND THEN TYPE THE CHARACTER THAT IS USED TO TYPE A BYTE THE REGISTER IS STILL OPEN AS A WORD. THE REGULAR ODT RE-OPENS THE REGISTER AS A BYTE.

YET ANOTHER DIFFERENCE IS THE USE OF THE "S" (SINGLE INSTRUCTION) COMMAND. THE "S" COMMAND DOES NOT SET A MODE BUT OPERATES LIKE THE " COMMAND IN THAT IT CAUSES AN INSTRUCTION TO BE EXECUTED. (MAYBE YOU SHOULD READ THE WHOLE DOCUMENT).

THE PURPOSE OF THIS VERSION OF ODT IS TO ALLOW THE USER TO LOOK AT THE DISK. THIS IS DESCRIBED LATER ON. WHAT REALLY HAPPENED IS THAT THE WHOLE THING GOT RE-WRITTEN CAUSE I COULDN'T RESIST IT.

NOW COMES THE OPINION PART. THE SYNTAX IS MUCH CLEANER AND MORE POWERFUL. THE CORE REQUIRED BY THE CODE AND DIRTY DATA IS ALSO MUCH LESS THAN THE STANDARD ODT, BUT WHEN YOU ADD THE DISK I/O BUFFER OF 1000 BYTES IT USES SOMEWHERE AROUND 6200 BYTES TOTAL.

THE CURRENT SOURCE OF ODT MAY BE ASSEMBLED, LINKED, AND LOADED JUST ABOUT ANYWHERE. THERE IS A SYMBOL CALLED "ORGODT" THAT IS USED TO SET THE LOWEST ADDRESS OF ODT IN ABSOLUTE ASSEMBLIES. IF THE ASSEMBLY IS RELOCATABLE THEN THIS SYMBOL IS SET TO "ORGODT=.."

THE SIZE OF ODT IS AS FOLLOWS:

DISK I/O	1000	(BYTES)
STACK	40	
CODE	5200*	
TOTAL	6240	

* THIS IS APPROXIMATE, FOR EXACT FIGURES TRY ASSEMBLING IT.

SOME THINGS TO NOTE ABOUT STARTING ODT.

THE SOURCE HAS ALL VARIABLES SET TO THEIR INITIAL VALUES AND SETS A VECTOR IN LOCATION 14 TO GO TO ODT. THIS IS TO ALLOW INITIAL ENTRY VIA THE "BKP" INSTRUCTION (000003) AND GIVES CLEAR BREAKPOINTS AND EMPTY RELOCATION REGISTERS.

THERE ARE TWO ENTRY POINTS DEFINED THAT MAY BE OF INTEREST.

"0,ODT" IS AN ENTRY THAT IS USED FOR A COMPLETE CLEAN START. ALL BREAKPOINTS AND RELOCATION REGISTERS ARE CLEARED

"0,ENTR" IS AN ENTRY THAT MAKES ODT THINK IT HAS HIT AN UNDEFINED BREAKPOINT. ALL ACTIVE BREAKS ARE REMOVED FROM THE USER'S CORE. IF YOU PROCEED FROM THIS ENTRY WITHOUT ALTERING YOUR "PC" YOU ARE RIGHT BACK WHERE YOU STARTED FROM.

THE USER'S CONSTANT REGISTER IS INITIALIZED BY THE SOURCE TO CONTAIN THE ADDRESS OF "0,ODT". THE ADDRESS OF "0,ENTR" IS EQUAL TO "0,ODT+2".

THE CUE FOR A COMMAND IS "<CR><LF>." AS OPPOSED TO "*" THIS IS SUPPOSED TO MAKE YOU THINK NEW SYNTAX. BESIDES THIS ODT WAS DEBUGGED WITH PDP-10 MIMIC AND "*" IS WHAT MIMIC TYPES FOR A COMMAND.

THE "RUB-OUT" CHARACTER IS USED TO RESET THE COMMAND INPUT ROUTINE VIA AN ERROR ENTRY. THE TYPE-AHEAD FEATURE IS DRIVEN VIA THE CHARACTER OUTPUT ROUTINE (IDEA COURTESY NATHAN TEICHHOLTZ) AND CHECKS EACH CHARACTER AGAINST A <RO>. THIS ALLOWS STOPPING OF THOSE LONG LISTS OF CELLS IN A WORD SEARCH.

THE TYPE-AHEAD SCHEME IS NOT INTERRUPT DRIVEN, BUT THE WAIT LOOP OF CHARACTER OUTPUT KEEPS LOOKING AT THE INPUT FLAG. WHEN THE INPUT FLAG COMES UP THE CHARACTER IS STASHED AWAY IN A RING BUFFER. THE CHARACTER INPUT ROUTINE DOES KEEP LOOKING AT THE INPUT FLAG. THE SCHEME DOES NOT WORK UNLESS ODT IS DOING OUTPUT AND THE CASE IS ALMOST ALWAYS THAT ODT IS DOING OUTPUT SINCE IT IS ECHOING INPUT BUT THERE ARE CASES SO BE NOTIFIED.

DESCRIPTION OF NOTATION--

ALL NUMBERS IN THIS DOCUMENT ARE OCTAL NUMBERS UNLESS THEY EXPLICITLY ARE SUFFIXED BY A "." IN WHICH CASE THEY ARE DECIMAL.

& /C CONSTRUCTS--

OCTAL NUMBERS-- NOTATION <OCT>

THIS IS AN OCTAL INTEGER IN THE RANGE 177777 TO 000000. IT'S USE IS CONTEXT DEPENDENT AND THE ABOVE RANGE MAY ALSO BE FURTHER LIMITED BY CONTEXT.

THERE ARE SPECIAL CHARACTERS DEFINED TO REPRESENT USEFUL OCTAL NUMBERS WITHIN ODT'S GUTS. THESE CHARACTERS ARE TO BE USED INTERCHANGEABLY WITH OCTAL NUMBERS. THEY ARE:

C	VALUE STORED IN USER'S CONSTANT REGISTER
Q	LAST OCTAL QUANTITY TYPED
.	ADDRESS OF LAST OR CURRENTLY OPEN REGISTER

VALUES-- NOTATION <VAL>

<VAL> DESIGNATES AN EXPRESSION, WITHOUT PARENTS, WITH THE FOLLOWING OPERATORS DEFINED:

-	SUBTRACTION, UNARY MINUS
+,<SP>	ADDITION
*	MULTIPLY BY 50 AND ADD
,	RELOCATION COMPUTATION

SEPARATORS--

VALUES ARE SEPARATED BY THE CHARACTER ";" AND BY COMMANDS.

NOTES:

A SPACE IS EQUIVALENT TO A PLUS FOR ADDITION. THIS ONLY HAS MEANING WHEN THE PREVIOUS CHARACTER WAS NOT A SEPARATOR WHICH SAYS YOU CAN TYPE ALL THE SPACES YOU WANT WHILE YOU ATTEMPT TO FIGURE OUT WHAT THE NEXT THING TO DO IS.

ADJACENT OPERATORS DO NOT EXIST. ALL OPERATORS ARE SEPARATED BY AT LEAST ONE NULL WHICH ODT LIKES TO CALL A "0" (A TRUE MATHEMATICIAN MIGHT WINCE).

REGISTER MANIPULATION COMMANDS--

OPENING REGISTERS--

FORM-- <VAL><OPEN>

THIS FORM OPENS THE REGISTER INDICATED BY <VAL> AND TYPES THE CONTENTS IN THE MODE SPECIFIED BY THE <OPEN> COMMAND. THIS MODE REMAINS EFFECTIVE UNTIL ANOTHER COMMAND OF THIS FORM IS TYPED.

FORM-- <OPEN>

THE LAST OPENED REGISTER IS PRINTED IN THE MODE SPECIFIED. THE MODE IS EFFECTIVE FOR THIS ONE COMMAND.

<OPEN>		
	/	OPEN AS OCTAL WORD
	\	OPEN AS OCTAL BYTE
	"	OPEN AS ANSII WORD
	'	OPEN AS ANSII BYTE
	%	OPEN AS RADIX 50 WORD

CLOSE REGISTER COMMANDS--

FORM-- <VAL><CLOSE>

THIS FORM CAUSES <VAL> TO BE STORED IN THE CURRENTLY OPEN REGISTER AND THE ACTION SPECIFIED BY <CLOSE> TO BE TAKEN. IF <VAL> IS NULL THEN NO DATA IS STORED AND ONLY THE ACTION IS TAKEN.

OSE>--		
	<CR>	CLOSE AND NO SPECIFIC ACTION
	<LF>	CLOSE AND OPEN THE NEXT LOCATION "DOWN"
	^	CLOSE AND OPEN THE NEXT REGISTER "UP"
	↑	CLOSE AND OPEN REGISTER SPECIFIED BY THE CONTENTS OF THE OPEN REGISTER TAKEN AS A PC RELATIVE ADDRESS.
	@	AS ↑ BUT OPEN IS ABSOLUTE ADDRESS
	>	AS ↑ BUT OPEN IS BRANCH DISPLACEMENT
	<	CLOSE AND RE-OPEN REGISTER LAST OPENED BY AND EXPLICIT OPEN COMMAND.

INTERNAL REGISTER REFERENCING--

FORMS--

$N	USER'S HARD REGISTERS, R0-R7. 0<=N<=7
$L	INTERNAL CONTROL REGISTERS
$NL	INTERNAL CONTROL TABLES WHERE:

L — IS THE CODE LETTER FOR THE SPECIFIC REGISTER OR TABLE.

AND

N — IS THE POSITIVE OFFSET TO THE TABLE. THIS IS A NUMBER 0<=N<=37 AND IT DOUBLED AND ADDED TO THE ADDRESS SPECIFIED BY "L" TO GET THE CORE ADDRESS.

DEFINITIONS OF L--

CONTROL REGISTERS--

S	USER'S STATUS REGISTER
P	ODT'S RUNNING PRIORITY
A	ARGUMENT REGISTER
M	MASK REGISTER
L	LOW MEMORY SCANNING REGISTER
H	HIGH MEMORY SCANNING REGISTER
C	USER'S CONSTANT REGISTER
Q	LAST QUANTITY TYPED REGISTER
Z	DISK I/O BUFFER START ADDRESS (256. WORDS)
F	ADDRESS OUTPUT FORMAT SWITCH

(THE ORDER ABOVE IS THE SAME AS THE ORDER IN MEMORY)

CONTROL TABLES--

B	0-7	BREAKPOINT ADDRESSES
G	0-7	BREAKPOINT PROCEED COUNTS
I	0-7	BREAKPOINT INSTRUCTION STORAGE
R	0-37	RELOCATION BASE ADDRESS OR SEGMENT NUMBER
T	0-3	CONSOLE/TTY DEVICE ADDRESSES
U	20-27	RELOCATION DISK UNITS (RK TYPE DISK)
U	30-37	RELOCATION DISK UNITS (RP TYPE DISK)

NOTES:

ODT TREATS THESE CONSTRUCTS AS A METHOD OF GENERATING ADDRESSES AS FOLLOWS:

THE OFFSET IS MULTIPLIED BY 2 AND ADDED TO THE ADDRESS SPECIFIED BY THE CODE LETTER. THE HARD REGISTER CASE IS THE NULL LETTER.

THERE IS ONLY A MAXIMUM CHECK FOR THE OFFSET VALUE ATTACHED TO A TABLE REFERENCE.

CONTROL REGISTERS CANNOT BE USED WITH OFFSETS.

REGISTER USAGE--

A BRIEF DESCRIPTION OF THE USAGE OF SELECTED REGISTERS APPEARS BELOW:

$C THE USER CAN PLACE A CONSTANT VALUE IN THIS REGISTER AND ACCESS IT BY TYPING A "C" AS PART OF A <VAL> CONSTRUCT. IT IS INTENDED AS A WAY TO CUT THE TYPING NEEDED TO GET THINGS DONE. WHEN ODT IS LOADED THE REGISTER "$C" HAS THE ADDRESS OF ODT'S STARTING POINT STORED THERE (TYPE "C$").

$Z THE ADDRESS OF THE DISK I/O BUFFER USED IN THE DISK LOOK IS KEPT HERE. THE USER MAY CHANGE THIS AT ANY TIME. ODT ASSUMES THAT THE BUFFER IS 256. WORDS (512. BYTES OR 1000 OCTAL) IN LENGTH AND THAT THE FIRST WORD'S ADDRESS IS IN "$Z". IF THE CONTENTS OF "$Z" ARE ZERO AN ATTEMPT TO USE THE DISK LOOK CAUSES ODT TO QUESTION YOU.

$F WHENEVER ODT TYPES AN ADDRESS (BREAKPOINTS, SEARCHES, OPENING LOCATIONS, ETC.) IT CONSULTS WITH THE "FORMAT" REGISTER TO SEE IF THE USER WANTS THE ADDRESS TYPED IN RELOCATABLE OR ABSOLUTE. IF "$F" SAYS "ONE" THAT MEANS ARSOLUTE, AND SAYING "ZERO" MEANS RELOCATABLE

$P ODT CAN RUN AT ANY CPU PRIORITY (AT LEAST FOR A WHILE). SETTING THIS REGISTER TO A "377" MEANS THAT ODT IS TO RUN AT THE PRIORITY OF THE USER WHEN THEY HIT THEIR BREAKPOINT. IF THE REGISTER DOES NOT CONTAIN A "377" THE REGISTER'S CONTENTS ARE SHIFTED LEFT 4 PLACES AND JAMMED INTO "PS". THE ODT ASSUMES THAT YOU TYPED THE PRIORITY IN THE LO-3 BITS OF "$P".

SYNTAX OF REGISTER REFERENCES--

BE WARNED OF THE FOLLOWING PROBLEM WITH ODT'S SYNTAX. ODT USES LETTERS AS PART OF A VALUE AND USES LETTERS AS A DELIMITER. FOR EXAMPLE:

$7G MEANS THE SEVENTH PROCEED COUNT, NOT GO AT REGISTER SEVEN.

THIS SHOULD CAUSE NO PROBLEMS WITH ODT AS IT STANDS BUT THE "I" AND "U" TABLE DESIGNATORS ARE NOT YET COMMANDS. IF THEY ARE EVER MADE INTO COMMANDS, BE CAREFUL.

GO TO THE PROGRAM, "G" ---------- "G" ----

THIS COMMAND SETS THE CPU'S REGISTERS 0-7, AND PS FROM THE USER'S IMAGE MAINTAINED BY ODT. THIS ACTION HAS THE NET RESULT OF STARTING THE USER'S PROGRAM.

SYNTAX--

<VAL>G	SET THE USER'S PC TO <VAL> AND START THE TARGET PROGRAM.
G	USE THE CURRENT SETTINGS OF THE USER'S REGISTERS TO START THE PROGRAM.

NOTES--

A "G" COMMAND SETS S.I. MODE TO "OFF", IE. GO GOES!!

ERRORS- ODD ADDRESS.

PROCEED FROM BREAK, "P" ---------- "P" ----

USE THIS COMMAND TO PROCEED FROM THE LAST BREAKPOINT HIT BY ODT. THIS IS USED FOR THE NORMAL BREAKS #0-7 AND THE SINGLE BREAK #8.

SYNTAX--

<VAL>P	PROCEED THRU THE CURRENTLY ACTIVE BREAKPOINT <VAL> TIMES BEFORE STOPPING AGAIN.
P	<VAL> DEFAULTS TO 1 IF NOT SPECIFIED.

NOTES--

A "P" COMMAND SETS S.I. MOOD TO "OFF".

THE INSTRUCTION AT THE BREAK ADDRESS IS EXECUTED WHEN THE PROCEED TAKES PLACE.

A "G" TO A BREAKPOINT CAUSES THE BREAK TO HAPPEN.

WHEN A BREAKPOINT IS HIT AND MAY BE PROCEEDED THRU, ODT TYPES THE FOLLOWING:

NB:AAAAAA	WHERE: N IS THE BREAK NUMBER, 0-7 WHICH ARE USER SET, AND 8 IS THE CODE FOR A SINGLE INSTRUCTION BREAK.

IF ODT FIELDS A "TRAP" INTERRUPT AND HAS NO BREAKPOINT DEFINED FOR THAT LOCATION IT TYPES "BE" FOR THE ENTRY. THE USER PC IS SET TO THE LOCATION FOLLOWING THE "TRAP"

ERRORS- NO ACTIVE BREAKPOINT.

SET A BREAKPOINT, "B" ---------- "B" ----

ALL BREAKPOINTS MAY BE CLEARED, SINGLE ONES CLEARED, AND INDIVIDUAL ONES SET VIA THE "B" COMMAND.

SYNTAX--

B	CLEAR ALL BREAKPOINTS
<OCT>B	CLEAR ONLY BREAKPOINT #<OCT>
<VAL>;<OCT>B	SET BREAKPOINT #<OCT> AT LOCATION <VAL>.
<VAL>;B	LET ODT PICK THE BREAKPOINT TO SET AT LOCATION <VAL>.

NOTES--

THERE ARE THREE CONTROL TABLES ASSOCIATED WITH A BREAK:

$NB	THE ADDRESS OF THE BREAKPOINT.
$NG	THE PROCEED COUNT OF THE BREAKPOINT.
$NI	THE CONTENTS OF THE ADDRESS AT THE BREAKPOINT WHILE THE USER IS RUNNING. 0<=N<=7 WHERE N IS THE BREAK #

IF YOU SHOULD PERCHANCE GET INTO ODT WITH BREAKPOINTS IN YOUR OWN PROGRAM, AND IT IS EASY TO DO, YOU FIRST MIGHT TRY LOOKING AT THE "I" TABLE TO FIND THE CONTENTS OF THE LOCATIONS IN THE "R" TABLE THAT GOT SCREWED UP. CLEARING BREAKPOINTS IN THIS STATE IS DISASTER. TRY LOOKING FOR "3"'S IN MEMORY FOR A START.

ERRORS- BAD BREAKPOINT NUMBER, ODD ADDRESS, RAN OUTTA BREAKS.

SINGLE INSTRUCTION MODE, "S" ---------- "S" ----

THIS COMMAND SETS THE MOOD WHERE ODT TRAPS ON EACH INSTRUCTION EXECUTED AS IF IT WERE A BREAKPOINT. THE "S" COMMAND CAUSES AN INSTRUCTION TO BE EXECUTED IN THIS MODE.

SYNTAX--

S	EXECUTE A SINGLE(1) INSTRUCTION AND THEN BREAKPOINT WITH #8.
<OCT>S	EXECUTE <OCT> INSTRUCTIONS AND THEN BREAK WITH #8.

NOTES--

THE USUAL PROBLEMS OF EXECUTION OF CERTAIN INSTRUCTIONS WITH THE "T-BIT" ON ARE SOLIDLY ENTRENCHED IN THIS ODT. THE STACK OF ODT IS ABOUT 40 WORDS IN LENGTH AND IF YOU CONSIDER THE 1000 BYTE DISK BUFFER THAT IS IN FRONT OF THE STACK THAT IS A LOT OF STACK SPACE.

ERRORS- NONE CHECKED FOR BY ODT.

COMPUTE OFFSETS, "O" ---------- "O" ----

THIS COMMANDS PRINTS THE PC RELATIVE AND BRANCH DISPLACEMENT ATWIXT TWO LOCATIONS. IT DOES IT CORRECTLY WHICH I NEVER AM QUITE ABLE TO DO.

SYNTAX--

<VAL>O	COMPUTE OFFSETS FROM "." TO THE ADDRESS <VAL>.
<VAL1>;<VAL>O	COMPUTE OFFSET FROM <VAL1> TO <VAL>

IF <VAL> IS NULL ODT THINKS ZERO.

NOTES--

THE OFFSETS ARE PRINTED ON THE LINE WITH THE COMMAND AS 16 BIT NUMBERS. THIS IS SO THAT YOU CAN FIGURE HOW FAR OUT OF BRANCH DISPLACEMENT YOU ARE.

THE PC RELATIVE OFFSET IS PREFIXED BY A "." AND THE BRANCH DISPLACEMENT IS PREFIXED BY A "±". TO SEE - AS + TYPE "-O=".

ERRORS- NONE POSSIBLE.

KOMPUTE RELOCATION VALUE, "K" ---------- "K" ----

THE RELOCATABLE EXPRESSION FOR GIVEN ADDRESSES IS TYPED BY THIS COMMAND. THE CORE (NOT DISK!) RELOCATION REGISTER AND OFFSET ARE DISPLAYED INDEPENDENTLY OF THE MODE IN "SF".

SYNTAX--

<VAL>;<OCT>K	COMPUTES AND TYPES THE RELOCATION EXPRESSION USING REGISTER <OCT> AND THE ADDRESS <VAL>.

IF <VAL> IS NULL THEN THE CURRENT LOCATION VIA "." IS USED. IF <OCT> IS EMPTY THEN ODT SCANS THE RELOCATION TABLES LOOKING FOR THE BEST FIT.

NOTES--

THE INFORMATION PRINTED BY "K" FOLLOWS ON THE SAME LINE AS FOLLOWS: =R,LLLLLL
WHERE: R IS THE RELOCATION REGISTER USED TO GET THE OFFSET L-L.

ERRORS- NO RELOCATION REGISTERS DEFINED.

SET RELOCATION REGISTER, "R" ---------- "R" ----

THERE ARE 40 RELOCATION REGISTERS AND 20 UNIT REGISTERS ASSOCIATED WITH THIS COMMAND. YOU SHOULD CONSULT THE "DISK LOOK" DESCRIPTION WITHIN THIS DOCUMENT FOR THE NITTY-GRITTY. IN SHORT-- "R" REGS. 0-7 ARE CORE, 10-17 ARE RF DISK, 20-27 ARE RK DISK, AND 30-37 ARE RP DISK.
"U" REGS 0-17 DON'T EXIST, 20-27 ARE RK UNIT REGISTERS, AND 30-37 ARE RP UNIT REGISTERS.
THE VARIATIONS ON THE "R" COMMAND SET AND RESET THOSE 60 REGS.

SYNTAX--

R	RESET ALL "R" REGS. TO -1 AND ALL "U" TO 0.
<OCT>R	RESET ONLY THE REGISTERS SPECIFIED BY <OCT> TO -1 AND 0.
<OCT1>,<VAL>,<OCT>R	SET RELOC REGISTER <OCT> TO CONTAIN <VAL> AND SET UNIT REGISTER <OCT> TO CONTAIN <OCT1>.

THE ABSENCE OF AN ARGUMENT TO THE MULTIPLE FORM OF THE "R" COMMAND MEANS THAT ODT SETS THAT VALUE TO 0. UNIT REGISTERS 0-17 ARE DEFINED TO CONTAIN ZERO'S.

NOTES-- THESE REGISTERS MAY BE ADDRESSED DIRECTLY AS INTERNAL REGISTERS BUT THE IS NO RANGE CHECKING DONE BY THE INTERNAL REGISTER ADDRESSING COMPUTATION ROUTINES SO IT IS RECOMMENDED THAT YOU USE THE "R" COMMAND TO DO THE SETTING AND THE "$R/$U" FORMS TO DO LOOKING.

ERRORS- YOU HAVE SPECIFIED A REGISTER THAT DOSEN'T EXIST

NOTES (AGAIN)--
ODT'S RP DISK DRIVER COULDN'T CARE LESS ABOUT WHETHER YOU HAVE A BUNCH OF RP02'S (USING RP11 CONTROLLER) OR RP03'S (USING THE RP11-C CONTROLLER) CAUSE THEY LOOK ALIKE ON THE DRIVER LEVEL. HOWEVER, THE RP03 IS A BIGGGGG DISK (80,000 BLOCKS) AND IT IS HARD TO REPRESENT 80,000 THINGS WITH A 16 BIT RELOCATION REGISTER. THE PROBLEM DOESN'T COME UP ON AN RP02 WITH A MERE 40,000 BLOCKS. THE PROBLEM IS RESOLVED BY A SPECIAL HACK WITH THE RP UNIT REGISTERS (30-37U). UNITS 0-7 (CONTENTS OF THE APPROPRIATE UNIT REGISTER) ARE USED TO INDICATE THE LOWER HALF OF AN RP03. UNITS 10-17 REFER TO THE UPPER HALF. WHAT HAPPENS IS THAT 40,000. IS ADDED TO THE RELOCATION SEGMENT NUMBER (CONTENTS OF 30-37R) IF THE CORRESPONDING UNIT REG (30-37U) IS >= 10. THE RP DRIVER DOES IT'S WORK IN DOUBLE PRECISION(32 BITS). FOR EXAMPLE IF 31U=15 AND 31R=10 AND YOU COMMAND ODT TO OPEN A WORD WITH ^31,26/ YOU WILL GET WORD 26 OF SEGMENT NUMBER 40,000.+10=40,008 ON RP UNIT 5 (IF YOU DON'T BELIEVE ME, TRY IT).

WORD SEARCH WITH MASK, "W" ---------- "W" ----

ALL LOCATIONS BETWEEN THE LIMITS DEFINED BY "SL" AND "SH" THAT ARE EQUAL TO THE VALUE IN "SA" UNDER THE MASK IN "SM" ARE PRINTED IN THE MODE SET BY THE LAST OPENED LOCATION. THE DATA IN CORE IS "ANDED" WITH THE MASK AND THE VALUE IN "SA" IS "ANDED" AND THE COMPARE IS MADE WITH THE MASKED VALUES.

SYNTAX--

<VAL1>;<VAL>W	SET "SM" TO <VAL1> AND SET "SA" TO <VAL> AND PERFORM THE SEARCH.
W	USE THE CURRENT SETTINGS OF "SA" AND "SM" TO DO THE SEARCH.

IF EITHER <VAL1> OR <VAL> IS NULL THEN THE CORRESPONDING REGISTER IS NOT SET.

NOTES--

THE SEARCH MAY BE INTERRUPTED BY THE <RO> COMMAND DESCRIBED LATER.

ERRORS- NONE.

NOT WORD SEARCH WITH MASK, "N" ---------- "N" ----

THIS COMMAND OPERATES SIMILARY TO THE "W" COMMAND EXCEPT THAT ALL LOCATIONS THAT ARE DIFFERENT UNDER THE MASK ARE PRINTED. THE SYNTAX IS THE SAME AS FOR "W", SAVE FOR THE COMMAND HISELF.

SYNTAX--

N	SEE THE "W" COMMAND FOR REGISTER SETTINGS.

NOTES--

THE SEARCH MAY BE INTERRUPTED BY THE <RO> COMMAND.

ERRORS- NONE.

EFFECTIVE ADDRESS SEARCH WITH MASK, "E" ---------- "E" ----

THE "E" COMMAND PRINTS ALL LOCATIONS BETWEEN "$L" AND "$H" THAT RELATE TO THE "$A" VALUE AS FOLLOWS:

EQUAL, SAME ABSOLUTE ADDRESS.

PC RELATIVE, THE LOCATION IN CORE IS ASSUMED TO BE A RELATIVE (MODE=6, REGISTER=7) ADDRESS.

"BR" DISPLACEMENT, THE LOCATION IN CORE IS ASSUMED TO BE A CONDITIONAL BRANCH INSTRUCTION.

SYNTAX--

E SEE THE "W" AND "N" COMMANDS FOR VARIATIONS

NOTES--

THE SEARCH MAY BE INTERRUPTED BY THE <RO> COMMAND.

THE EFFECTIVE ADDRESS IS ALSO A MASKED SEARCH. THIS IS TO ALLOW RANGE SEARCHES.

PLEASE NOTE THAT THE "E" SEARCH WILL ALMOST ALWAYS GIVE "FALSE DROPS"*.

ERRORS- NONE.

* A LEFT OVER FROM THE OLD DAYS IN I.R.

LIST MEMORY ON DEVICE, "L" ---------- "L" ----

THE CONTENTS OF THE CELLS ATWIXT "$L" AND "$H" AND LISTED UP ON THE SPECIFIED DEVICE.

SYNTAX--

<OCT>;<VAL>;<VAL1>L <OCT> IS THE OUTPUT DEVICE WHERE 0 OR NULL IS THE CONSOLE AND 1 IS THE LP11. <VAL> AND <VAL1>, IF PRESENT SET "$L" AND "$H" RESPECTIVELY.

NOTES--

THE "$L" LIMIT IS ANDED WITH A "177770" AFTER BEING PICKED UP AND BEFORE USE.

THE "L" COMMAND MAY BE INTERRUPTED AT ANY TIME VIA THE <RO> COMMAND.

THE LISTING IS PRINTED IN THE MODE SET BY THE LAST OPENED REGISTER.

ERRORS- NONE.

MISC COMMANDS--

FILL MEMORY WITH WORDS, "F" ---------- "F" ----

FILLS MEMORY WITH A GIVEN WORD. STARTS AT THE LOCATION SET IN THE REGISTER DEFINED BY "$L" AND STOPS AFTER THE LOCATION SET IN THE REGISTER DEFINED BY "$H" HAS BEEN FILLED.

SYNTAX--

F	USE THE CONTENTS OF "$A" TO DO THE FILL FROM "$L" TO "$H" INCLUSIVE.
<VAL>F	SET THE REGISTER "$A" TO <VAL> AND THEN DO AS ABOVE.

NOTES--

ERRORS- ALL YOUR OWN.

ALTER CONSOLE TTY ASSIGNMENT, "T" ---------- "T" ----

IF YOU HAVE A PDP-11 WITH LOTS OF DIFFERENT DEVICES, LIKE A TV SET, USE THIS COMMAND TO MAKE ODT USE THE DEVICE FOR IT'S CONSOLE I/O. THE OLD CONSOLE STATUS IS RESTORED AND THE NEW STATUS IS PICKED UP WHEN CONSOLES ARE SWITCHED.

SYNTAX--

<VAL>T	USE THE I/O PAGE ADDRESSES BEGINNING AT <VAL> FOR CONSOLE I/O.
T	RESET THE I/O DEVICE TO THE STANDARD PDP-11 CONSOLE TTY ASSIGNMENT (177560)

NOTES--

THE GIVEN ADDRESS IS ASSUMED TO BE THE READ STATUS REGISTER ADDRESS AND THE REMAINING THREE DEVICE REGISTERS ARE ASSUMED TO BE THE NEXT THREE WORDS IN THE I/O PAGE. THIS, I AM ASSURED, IS NOT AN ASSUMPTION BUT A DEFINITION. WE ALL KNOW WHAT THAT MEANS.

ERRORS- ALL YOUR OWN.

PRINT EXPRESSION ON LEFT ON RIGHT, "=" ---------- "=" ----

THE EXPRESSION OR VALUE ON THE LEFT OF THE "=" IS PRINTED AS AN OCTAL WORD.

SYNTAX--

<VAL>=	PRINT <VAL> AS AN OCTAL WORD.

CANCEL AND RETURN, "<RO>" ---------- "<RO>" -

THIS CHARACTER IS ABLE TO STOP THE CURRENT ACTION OF ODT, NOT TO BE CONFUSED WITH THE USER, AND RETURN TO THE COMMAND INPUT ROUTINE VIA THE ERROR ROUTINE. THE CHARACTER IS TESTED FOR IN THE TYPE-AHEAD ROUTINE AND WHEN DETECTED GOES TO THE ERROR ENTRY AND THIS RESETS TYPE-AHEAD AND CAUSES ODT TO WAIT FOR SOMETHING TO DO.

SYNTAX--

<RO> STOP ODT AND WAIT FOR COMMAND.

NOTES--

USE THIS COMMAND TO STOP SEARCHES SINCE THE TYPE-AHEAD USES THE OUTPUT ROUTINE TO LOOK FOR INPUT.

THE "DISK LOOK" EXTENSION TO ODT USES AN EXPANDED VERSION OF THE RELOCATION FACILITY. IN GENERAL THE USER IS ABLE OPEN ONE WORD ON A GIVEN DISK, EXAMINE THAT WORD, MODIFY AND CLOSE AT WORD. THE SAME HOLDS FOR BYTES.

THE DISK MAY BE OPERATED UPON IN THIS MANNER ONLY, AND NO ABILITY TO SEARCH DIRECTLY OR OTHER SUCH OPERATIONS IS IMPLIED. THE ASTUTE HACKER MAY FIGURE SOME THINGS OUT THOUGH.

THE USER IS GIVEN THE ABILITY TO CONTROL THE BUFFER USED IN THE DISK LOOK FACILITY. THE INITIAL BUFFER DEFAULTS TO THE 256. WORDS IMMEDIATELY PRECEEDING ODT'S WORKING STORAGE.

THE DISK IS ADDRESSED VIA AN EXTENSION OF THE ODT RELOCATION SCHEME. THIS WILL BE SPECIFIED IN DETAIL A COUPLE OF PAGES LATER. THE SCHEME ALLOWS THE USER TO SET UP TO EIGHT BASES ON AN RF AND RK (EIGHT FOR ALL EIGHT POSSIBLE UNITS) AND THEN SPECIFY UP TO A 16. BIT OFFSET TO THAT BASE. THE BASE IS A 16. BIT SEGMENT NUMBER IN THE STANDARD RSTS FORMAT.

DISK LOOK EXTENSION--

THERE ARE 20 RELOCATION REGISTERS DIVIDED INTO GROUPS AS FOLLOWS:

CL - REGISTERS, NUMBERED 0-7 ($0R-$7R)
EACH REGISTER CONTAINS A 16. BIT CORE RELOCATION CONSTANT.

DF DISK REGISTERS, NUMBERED 10-17 ($10R-$17R)
EACH REGISTER CONTAINS ONE 16. BIT SEGMENT NUMBER
(A SEGMENT BEING A 256. WORD BLOCK OF DISK ALA RSTS)

THERE ARE 20 ADDITIONAL REGISTER PAIRS DEFINED AS FOLLOWS:

RK UNIT AND DISK REGISTERS, NUMBERED 20-27
$20R-$27R RK ADDRESS (SEGMENT) REGISTERS
$20U-$27U RK UNIT REGISTERS

RP UNIT AND DISK REGISTERS,, NUMBER 30-37
$30R-$37R RP ADDRESS (SEGMENT) REGISTERS
$30U-$37U RP UNIT REGISTERS

THE UNIT REGISTERS CONTAIN THE UNIT NUMBER (0-7 ON RK OR RP02) (0-17 ON RP03) WITH ITS ASSOCIATED ADDRESS (SEGMENT) REGISTER. THE ADDRESS REGISTER CONTAINS A 16. BIT SEGMENT NUMBER.

THE UNIT REGISTERS ARE DEFINED TO CONTAIN A ZERO FOR ALL RELOCATION REGISTERS NUMBERED 0-17.

DISK I/O BUFFER--

DUE TO VARIOUS AND SUNDRY CHARACTERISTICS OF DISKS AND RSTS ALL I/O DONE BY THE DISK LOOK FACILITY IS DONE IN 400 WORD UNITS. THIS SCHEME REQUIRES A 400 WORD BUFFER, AND ODT USES THE VALUE STORED IN THE REGISTER ADDRESSED BY $Z TO DO IT'S I/O. THE INITIAL VALUE IN $Z IS SET TO POINT TO A BUFFER THAT IS DEFINED TO EXIST IN THE 400 WORDS IMMEDIATELY BEFORE THE SPACE USED BY ODT FOR IT'S STACK.

DISK I/O DESCRIPTION--

IT IS DEEMED IMPORTANT THAT THE USER BE ABLE TO FIND OUT WHAT IS DONE TO THE DISK WHEN HE USES THE DISK, IT IS ESPECAILLY SO IF THE USER IS DEBUGGING A DISK SERVICE. WHEN EVER ODT ACCESSES THE DISK THE FOLLOWING DRILL IS EXECUTED:

SAVE THE CALLING "PS".
WAIT UNTIL THE DISK IS NOT BUSY.
SET PRIORITY TO "7"
IF DISK IS BUSY AGAIN RESTORE CALLING "PS" AND START OVER.
ISSUE A CLEAR CONTROLLER COMMAND TO THE DISK.
AWAIT UNTIL READY AGAIN
PERFORM THE READ OR WRITE
WAIT UNTIL READY
RESTORE THE CALLING "PS" VALUE
CHECK THE DISK FOR ERRORS.
EXIT.

THE DISK I/O IS DONE IN THIS MANNER SO THAT CONSISTENCY IS INSURED. IT APPEARS TO BE IMPOSSIBLE TO COMPLETELY SAVE AND RESTORE THE DISK SATUS SO I HAVE TAKEN THE OTHER EXTEREME.

GENERAL FORM--

<OCT1>,<OCT><OPEN>

-OR-

<VAL><OPEN>

<OCT1> IS THE RELOCATION/UNIT PAIR TO BE USED (RANGE 0-37).

<OCT> IS THE OFFSET TO BE ADDED TO THE RELOCATION UNIT PAIR SPECIFIED (RANGE 0-177777).

IF THE CONSTRUCT "<OCT1>," IS ABSENT, THEN CORE IS ADDRESSED.

FOR EXAMPLE:
0;0;0R
0,1462/ XXX
HAS THE SAME OUTPUT (NOT EFFECT) AS TYPING:
1462/ XXX

IF <OCT1> IS ABSENT THEN THE VALUE DEFAULTS TO ZERO.

FOR EXAMPLE:
0,1324/ XXX
HAS THE SAME EFFECT (AND OUTPUT) AS TYPING:
,1324/ XXX

FOR EXAMPLE--

TO LOOK AT THE CORE IMAGE OF RSTS LOCATED BEGINNING IN SEGMENT 61(8) AND TO EXAMINE LOCATION 17362 OF THE CORE IMAGE I WOULD TYPE THE FOLLOWING: (FOR CIL ON RK UNIT 0)

```
61;20R          I.E. SET UNIT 0 BY DEFAULT, RELOCATE TO SEGMENT 61
20,17362/ XXX   OPEN WORD 17362(8) FROM TOP OF SEGMENT 61
                WHICH IS ACTUALLY WORD 362(8) OF SEGMENT 80(8)
```

THE OPENED LOCATION MAY BE OPERATED UPON AS IF IT WAS CORE.
(WHICH IN FACT IT IS!)

ODT COMMAND SUMMARY, LETTERS AND NUMBERS--

LETTER	COMMAND	REGISTER
/		ARGUMENT
B	BREAKPOINT SET/RESET	BKPT ADDRESSES (0-7)
C	CONSTANT	CONSTANT
D		
E	EFFECTIVE ADDRESS	
F	FILL	FORMAT CONTROL
G	GO	BKPT PROCEED COUNTS (0-7)
H		HIGH SCAN LIMIT
I		BKPT INSTRUCTION (0-7)
J		
K	RELOCATION VALUE COMPUTE	
L	LIST MEMORY	LOW SCAN LIMIT
M		MASK
N	NOT WORD SEARCH	
O	OFFSET VALUE COMPUTE	
P	PROCEED	ODT PRIORITY
Q	QUANTITY	QUANTITY
R	RELOCATION SET/RESET	RELOCATION (0-27)
S	SINGLE INSTRUCTION	STATUS WORD
T	ALTER CONSOLE TTY	TTY ADDRESSES (0-3)
U		UNIT REGISTER (20-27)
V		
W	WORD SEARCH	
X		
Y		
Z		DISK LOOK BUFFER
0-7	OIT'S	USER REGISTERS (0-7)

ODT COMMAND SUMMARY, CHARACTERS--

CHARACTER	ACTION
	OPEN OCTAL BYTE
	OPEN OCTAL WORD
	OPEN ANSII BYTE
"	OPEN ANSII WORD
%	OPEN RADIX 50 WORD, 3 CHARACTERS
<CR>	CLOSE LOCATION
<LF>	CLOSE, OPEN NEXT IN + SEQUENCE
^	CLOSE, OPEN NEXT IN - SEQUENCE
+	CLOSE, OPEN PC RELATIVE ADDRESS
@	CLOSE, OPEN INDIRECT OR ABSOLUTE
>	CLOSE, OPEN OFFSET AS BRANCH
<	RETURN TO LAST EXPLICTILY OPENED LOCATION
$,<ALT>	REGISTER REFERENCE PREFIX
,	ARGUMENT SEPARATOR
'	RELOCATION OPERATOR
+,<SP>	ADDITION OPERATOR
-	SUBTRACTION OPERATOR
*	MULTIPLY BY 50 AND ADD
.	CURRENT PLACE ADDRESS
;	TYPE LEFT SIDE ON RIGHT
<RO>	CANCEL COMMAND AND RETURN
	ERROR WARNING

UNUSED CHARACTERS (NON-ALPHA)-- [ ] ( ) ! & ? # : 8 9

UNUSED LETTERS IN COMMANDS -- A D H I J M U V X Y Z

UNUSED LETTERS IN REGISTERS -- D E J K N O V W X Y

LETTERS JUST PLAIN UNUSED -- D J V X Y

LETTERS TO BE CAREFUL WITH -- I U

BASIC - PLUS FILE PROCESSING

Theodore R. Sarbin
Digital Equipment Corporation

INTRODUCTION

These notes were prepared to introduce the BASIC programmer to the use of files in a RSTS system on the PDP-11. Most of the language features of BASIC-PLUS are used and it is assumed that the reader is familiar with them. For those who know BASIC but not the extended language features a terse summary is provided in Appendix A. A more complete exposition is included in the BASIC-PLUS language manual. If there is any difficulty in understanding the example programs then the language manual should be available for consultation. When reading these notes the temptation to just read the text and not to attempt to understand the example programs should be avoided. The examples and diagrams are integrated with the text to such an extent that there is little benefit to be gained from just reading the text itself.

1. GENERAL

There are fundamentally three methods of performing input/output operations in BASIC-PLUS. They are serial input/output, virtual array, and "Record IO". The first type is achieved through the use of the INPUT and PRINT commands; virtual arrays are defined through the use of the DIM statement; and "Record IO" (which is something of a mis-nomer) uses the GET, PUT, FIELD, LSET and RSET commands as well as the functions; CVTxx, CHR$, and ASCII. All types of input/output operations make use of the OPEN and CLOSE commands.

2. DEVICE NAMES, FILE NAMES, AND LOGICAL UNITS

I/O operations in BASIC-PLUS are designed to be as device independent as reasonable. Thus in the I/O statements there are no specific references to devices. Instead, each statement contains a reference to a logical unit or "channel". (This is strictly a conceptual channel which should not be confused with a hardware device called a channel). Each "channel" is simply a number from one to twelve which potentially identifies the specific device which is to perform an I/O operation. Thus in one program logical unit or channel 3 could be the line printer and channel 11 the card reader, while in another program channel 3 could be magtape unit #2, channel 1 the paper tape punch and channel 12 a file on the disk. The association of the specific device with a channel number is done with the OPEN statement. The OPEN statement specifies a device, if necessary a file name, and the channel number to associate with that device. The form of the command is:

```
100 OPEN file name AS FILE n
```

The file name is a statement which completely describes where the data is to be found or put. It specifies the physical device as well as the name that is to be assigned to the information. The complete file name consists of the following parts:

DV:	NAMEXX.EXT	[111,222]	<60>
Device Name	File and Extension Names	Project, Programmer Number	Protection Code

The Device Name specifies the name of the device on which the file is located. It also may specify the unit number. Thus a file might be located on the number three cartridge disk and would be referenced as "DK3:". A list of device names can be found in the language manual. If no device name is specified then the public disk is assumed to be the one referenced. Most of the examples cited in these notes use the public disk for simplicity but in real applications it is uncommon to use other than private disks for production data files.

The File Name and Extension specify the name of the data. The name itself is any sequence of six or ~~less~~ fewer characters from the set of letters and numbers. A filename need not be specified when the device is of the type which does not recognize file names. (The only devices which do recognize filenames are disks, DecTape and, in some cases, Magnetic Tape). The only requirement about the filename is that it be meaningful to those who use it. The extension is a three character field which is usually used to describe the contents of the file. An extension of ".BAS" means that the file contains a BASIC-PLUS source program and an extension of ".BAC" means that the file contains a BASIC-PLUS compiled program. The programmer can assign any file extension he wants as long as it is meaningful to him. He need not assign any extension at all. The file "ABCDEF" is unique and distinct from "ABCDEF.BAS" or any other file with "ABCDEF" as the file name and an extension.

The project-programmer number field specifies in whose account the file is to be found. If it is omitted (and it usually is) then the system assumes that the account that the job is logged in under is the account that the file is, or is to be, stored in. Writing into a file stored under someone else's account is possible only if he has explicitly permitted that. Creating a file in someone else's account is generally not possible. (It can be done only by a privileged user. The System Manager's Guide contains a discussion of privileged users.)

The last field is the protection code. This determines who can or cannot access the file. It is only meaningful when the file is being created. It is described in the Language manual and need not concern us further here.

The file name is specified either as a string literal (the file name in quotes) or as a string variable which contains the filename. As an example the following could be used:

```
100 INPUT "WHAT IS THE INPUT FILE'S NAME", A$
110 OPEN A$ AS FILE 4
```

In this case the user of the program supplies the file name at the time the program is run. The n in the OPEN statement is simply a number from one to twelve or a variable which contains such a value. To terminate the association of a device and a channel we use the close statement and specify the channel number to close. For example:

```
100 CLOSE n
```

alternatively we can close several channels in one command:

```
100 CLOSE m, n, o
```

The open and close statements have several other options which will be described later but these statements create and terminate an association between

device and/or file names and numbered logical units or channels. Thus by changing the definition of the device or file for a specific channel we can change the devices the program uses without revising the program.

3. SERIAL INPUT AND OUTPUT

This is the simplest form of input/output. Whatever device is in use is simply treated as if it was a (possibly very fast) teleprinter. Where we learned that to print on the teleprinter we used the PRINT statement we now make use of the same statement but specify on which channel to print. Of course it is necessary to first associate a device or file with the channel by means of an OPEN statement. Thus to print a table of squares and square roots on the teleprinter we would have used the program:

```
100 FOR I = 1 TO 100
110 PRINT I, SQR(I), I*I
120 NEXT I
```

To print the same table out on the line printer we would use the program:

```
100 OPEN "LP:" AS FILE 2          !THIS ASSOCIATES THE LINEPRINTER
                                  !WITH CHANNEL NUMBER 2
110 FOR I=1 TO 100                !START THE LOOP
120 PRINT #2, I, SQR(I), I*I      !THIS DOES THE PRINTING
130 NEXT I                        !STOP THE LOOP
140 CLOSE 2                       !TERMINATE THE ASSOCIATION
150 END
```

To write a file of names and addresses onto the disk we might use the following program:

```
100 OPEN "NAMES.DAT" FOR OUTPUT AS FILE 1        !CHANNEL INITIALIZATION
110 INPUT "HOW MANY NAMES"; NX                   !FIND OUT HOW MANY NAMES
120 PRINT #1, NX                                 !AND PRINT IT INTO THE FILE
130 FOR IX = 1X TO NX                            !START THE LOOP
140 PRINT "INPUT THE NAME"                               !GET THE INFORMATION
150 INPUT LINE N$                                        !ON EACH NAME AND
160 PRINT "INPUT THE NUMBER AND STREET"                  !AND ADDRESS
170 INPUT LINE A$
180 PRINT "INPUT THE CITY, STATE, AND ZIP CODE"
190 INPUT LINE C$                                        !NOW IT CAN BE OUTPUT
200 PRINT #1, N$; A$; C$;                                !PRINT IT INTO THE FILE
210 NEXT IX                                      !END THE LOOP
220 CLOSE 1
230 END
```

You will notice that a new element has been added to the OPEN statement. When we specify that a file is for OUTPUT that means that this is a new file and any previous files of that name should be deleted first. Now suppose that we want to use this file to print on the lineprinter all those names and addresses for which the ZIP code begins with a 9 (i.e. those in California). The following program would do that:

```
100 OPEN "NAMES.DAT" FOR INPUT AS FILE 3    !SET UP CHANNEL 3 AS THE FILE
110 OPEN "LP:" AS FILE 5                    !AND CHANNEL 5 AS THE LINE PRTR
120 INPUT #3, NX                            !GET THE NUMBER OF FILE ENTRIES
130 FOR IX = 1X TO NX                !START THE LOOP
140 INPUT LINE #3, A$(JX) FOR JX=1X TO 3X        !GET ONE ENTRY
150 JX = LEN(A$(3X)) :
        JX=JX-1X UNTIL MID(A$(3X), JX, 1X)=" "
                OR JX = 0X                       !LOOK FOR THE ZIP CODE
160 PRINT #5, A$(KX); FOR KX=1X TO 3X
        IF MID(A$(3X), JX+1X, 1X)="9" AND JX<>0X
                                                 !PRINT ON LINE PRTR
                                                 !IF ZIP CODE IS 9XXXX
170 NEXT IX                                  !STOP THE LOOP
180 CLOSE 3,5 : END                          !ALL DONE
```

The previous program is intended as an example. Of course if it were a real application we would want to do more checking for valid data and probably at the time the data was entered. Also you will notice a new specification in the OPEN statement. When we specify that a file is to be opened for INPUT that means that we expect the file to already exist and that if it doesn't then that is an error and the program should not proceed. If we specify neither FOR INPUT nor FOR OUTPUT then the system will use the file of the specified name if one exists and if not it will create one and use it. (A somewhat anomalous situation exists here in that a file may be opened FOR INPUT (ie: it must exist) and then we can perform output operations on it or we can open a file FOR OUTPUT thus creating it and later in the same program we can read it.) At this point we have seen how to associate the name of a device or a file with a logical unit or channel number and how to access devices and files in a serial manner. Next we will take up the two methods of accessing data randomly.

4. VIRTUAL ARRAYS

This is a very simple technique for accessing disk files. We simple equate files on the disk with dimensioned arrays and treat them in the program as if they were simple arrays. Thus we can associate the name of a disk file with an I/O channel through the use of an OPEN statement and then associate the name of an array with its size and with the I/O channel through the use of a DIM statement:

Thus:

```
100 OPEN "FILE.DAT" FOR INPUT AS FILE 2
110 DIM #2, A%(1023), B(1024)
```

The above statements simply state first that an association is to be made between the file "FILE.DAT" on the public disk and channel two and that the file is to already exist. Then we say that the file is a virtual array whose names

are to be A%(n) and B(n). Thus to read the file we simply reference the elements of the array. To find the sum (as S%) of the first 500 elements in the file we might use the following statement:

```
120 S% = 0% : S% = S% + A%(J%) FOR J% = 0% TO 499%
```

When the loop is executed we effectively read from the file 500 times. To write into the file we simply place our virtual array reference on the left side of a replacement operator (equals sign). Thus to write the first 1024 elements of the file with the first 1024 even numbers and then the next 1025 elements with the first 1025 real numbers starting at zero and increasing by one-half we could use the following program:

```
130 A%(I%/2%) = I% FOR I% = 0% TO 2046% STEP 2%
140 B(I%) = I%*0.5 FOR I% = 0% to 1024%
```

One thing to remember is that in the program it looks just like we are refering to an in-core array, however, we really are reading and writing on a disk which is very much slower than reading and writing in core memory. Thus we must expect that our program will run much more slowly when we use virtual arrays. Another special aspect of virtual arrays of strings is the handling of the length of the string. When we store a string in core we keep track of its length internally by means of a character count. However, when we place the string arrays on the disk we need to be able to uniquely identify each one without checking on the length of the others. To accomplish this efficiently, all of the strings in any one string array must be the same length and that length must be an integral power of two (1,2,4,8,16,32,64, 128, 256 or 512). To define this length it is placed following an equals sign just after the right parenthesis in the DIM statement. Thus:

```
100 DIM #8, A$(100)=8, B$(20,20)=64
```

would state that the file open on channel eight consists of two arrays of strings: A$ consists of 101 strings each 8 characters long and B$ consists

of 441 strings each 64 characters long. If you do not specify the length of a virtual string array then the system will assume that it is 16 characters. Virtual string arrays are handled just like in core string arrays would be handled except that they are much slower. Examine the following example which accepts a list of names from the teleprinter and places them on the disk in a virtual string array. Then it sorts the string array using a fairly inefficient sorting algorithm and it prints the sorted list on the line printer. Thus:

```
100 OPEN "NAMES" FOR OUTPUT AS FILE 2    !CREATE THE FILE
110 DIM #2, N$(200)=64                   !ARRAY AS FILE
120 INPUT "HOW MANY NAMES"; N%           !HOW MANY NAMES IN THE FILE
130 INPUT LINE N$(I%) FOR I% = 0% TO N%-1%
                                         !READ THE NAMES INTO THE FILE
140 M% = N%-2%          !NOW FOR THE BUBBLE SORT
150 F% = 0%             !F% IS THE INTERCHANGE FLAG
160 FOR I% = 0% TO M% :
        IF N$(I%) > N$(I%+1%) THEN T$=N$(I%) :
                                   N$(I%) = N$(I%+1%) :
                                   N$(I%+1%) = T$ :
                                   F% = 1%
170 NEXT I% : M% = M% -1% : GOTO 150 UNLESS F%=0% !THIS IS THE DISK SORT
180 OPEN "LP:" AS FILE 3                         !ALL SORTED SO NOW PRINT
190 PRINT #3, N$(I%); FOR I% = 0% TO N%-1%       !THE SORTED LIST
200 CLOSE 2,3 : END      !ALL DONE
```

This is a simple example of sorting a file on the disk when the file is a virtual array. Virtual arrays are very easy to use and when their lack of flexibility is not a problem they can be used in place of record I/O when random access is desired. Virtual arrays, however, have high overhead and programs which use them will run slowly. The third method of processing files which we will take up next is the most flexible and the most powerful, but it also requires more work on the part of the programmer.

5. RECORD ORIENTED INPUT AND OUTPUT

In this section we will study the most powerful and most flexible provisions for input and output in BASIC-PLUS. Here we will examine the use of the GET, PUT, FIELD, LSET and RSET commands. Also we will make use of the six conversions funtions (CVTF$, CVT$F, CVT%$, CVT$%, CHR$, ASCII) and the special system variables RECOUNT and COUNT. We will meet some new variations on the OPEN statement. We will add the qualifiers RECORDSIZE, CLUSTERSIZE, and MODE.

6. STRING PROCESSING - LET STATEMENTS

Before we can start to study record I/O it is necessary to have a more thorough understanding of how the system handles strings internally. When we talk about operations on strings we should keep in mind that when we use a string identifier we are referring to two parts of a string. One is the string header which contains the string length and a "pointer" which points to the other part of the string, the data or characters which make up the string. Thus when we execute the statement:

```
100 LET A$ = "ABCDEF"
```

we produce the following sort of structure in core:

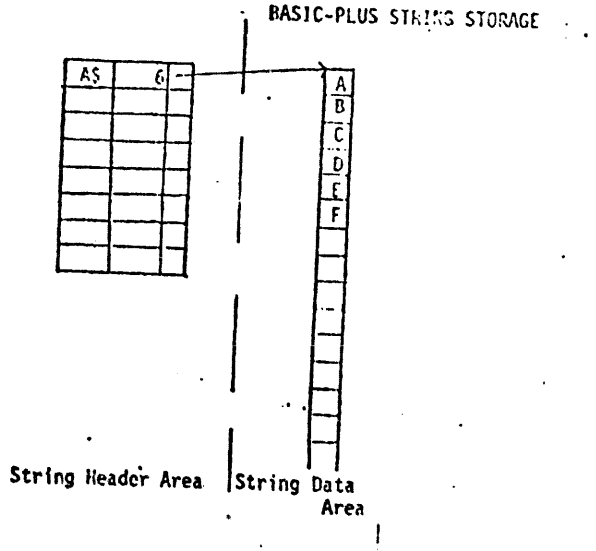


Figure 1

and when we next execute the statement:

```
110 LET B$ = "GHI"
```

we produce the following in core:

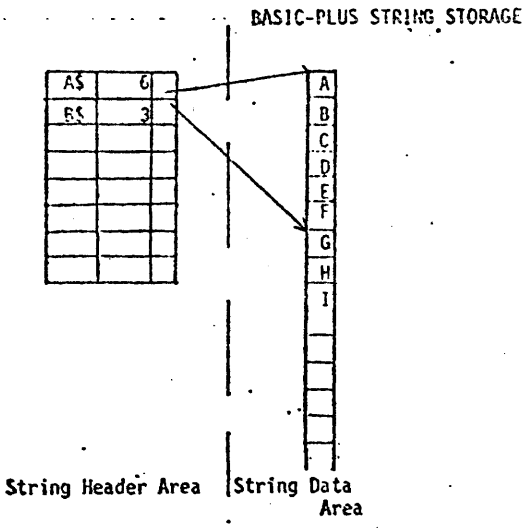


Figure 2

When an operation occurs which alters a string we necessarily create a new one and the old one becomes garbage. Thus the following program statement:

```
120 LET A$ = MID(A$, 2, 4)
```

causes the string header for the string A$ to point to a new string which has a length of 4 and the string which A$ used to point to is now garbage. As you see there is no string header pointing to it and thus it cannot be referenced.

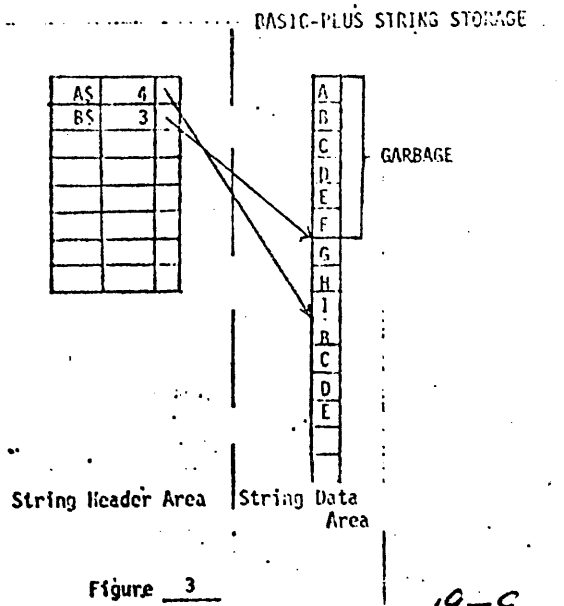


Figure 3

As a program proceeds it generates garbage in the string data area. Gradually this area fills up so that there is no more room for strings to be created. When this happens, the Garbage Collector (RSTS-11) or the Core Recycler (RSTS/E) is called and the strings are all collected into the beginning of the string data area.

When we execute a statement such as:

```
100 LET C = D
```

the operation which is performed is to take the contents of the variable D and move it to the variable named C. This explanation applies when we are talking about numbers, either integer or floating point. When we deal with strings, however, the operation of the = operator is different since we are now concerned with headers as well as string data. When the following three statements are executed, the string header area and string data area will look like figure 4:

```
100 LET A$ = "ABCD"
110 LET B$ = "UVW"
120 LET C$ = "ASDF"
```

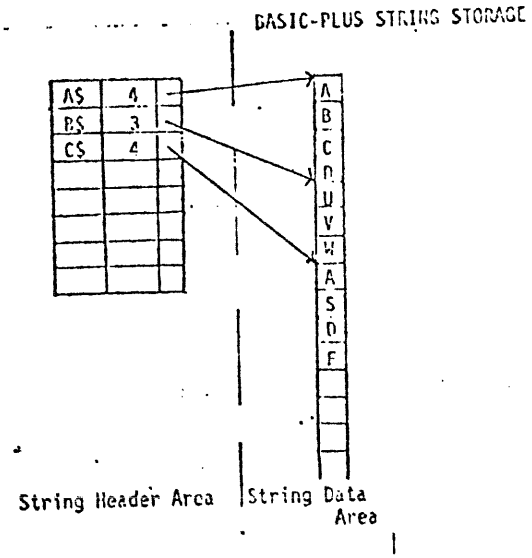


Figure 4

Thus each string header contains the length and a pointer to the start of the corresponding string. Now consider the effect of executing the following statement:

```
130 LET B$ = C$
```

The effect of this is to make the string header for B$ point to the SAME string in core as the string header for C$. Of course, the old string which B$ pointed to is now garbage and will be collected next time the garbage collector is called.

Figure 5 shows what the string area looks like after statement 130 is executed.

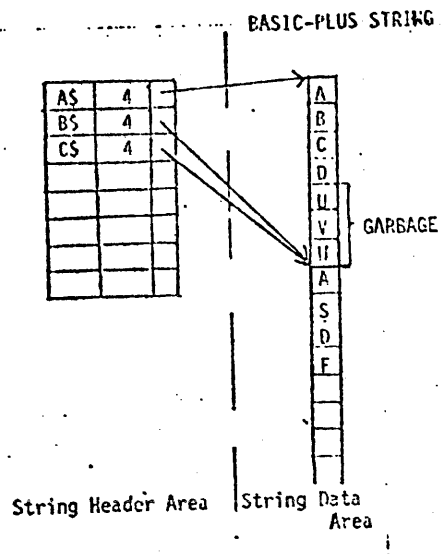


Figure 5

Any attempt to modify C$ will cause a new string to be formed with the new value of C$, however, the old value of C$ would not be garbage in this case because it is still pointed to by the B$ string header. Thus the statement below would cause a new string C$ to be created and the old C$ would still be B$ as shown in the following code and in figure 6.

```
140 LET C$ = MID(A$, 2, 2)
```

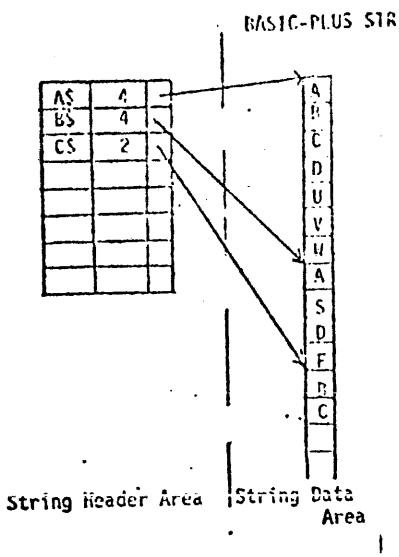


Figure 6

Thus we can see that a change to a string effected by a LET statement will not cause a change to another string which was equated with that string. Also any statement which is true of a LET statement is also true of an implied LET.

7. STRING PROCESSING - LSET AND RSET

There is another statement which is similar to the LET statement with strings except that it always operates on the data in the string data area and never on the string header. That is the LSET statement. The general form of the LSET statement is:

```
100 LSET string name = string formula
```

The effect of this statement is to evaluate the string expression or formula on the right side of the equals sign and then substitute that string for the string in the string data area which is pointed to by the string header for the string variable on the left side of the equals sign. Now consider the following:

```
100 LET A$ = "ZXCVB"
110 LSET A$ = "ABCDEFGH"
```

After statement 100 the string area looks like figure 7.

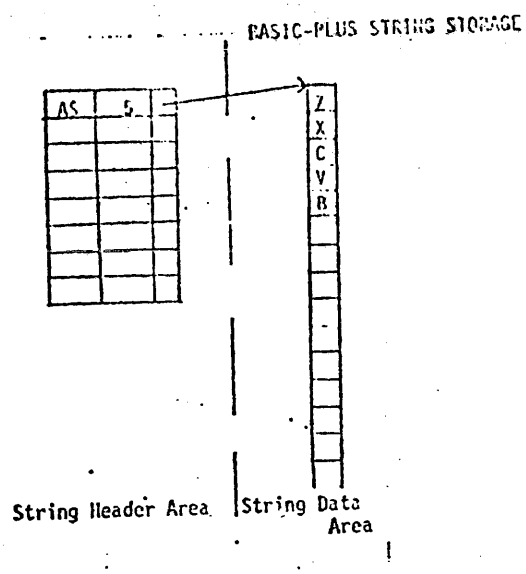


Figure 7

and after executing statement 110 it looks like figure 8.

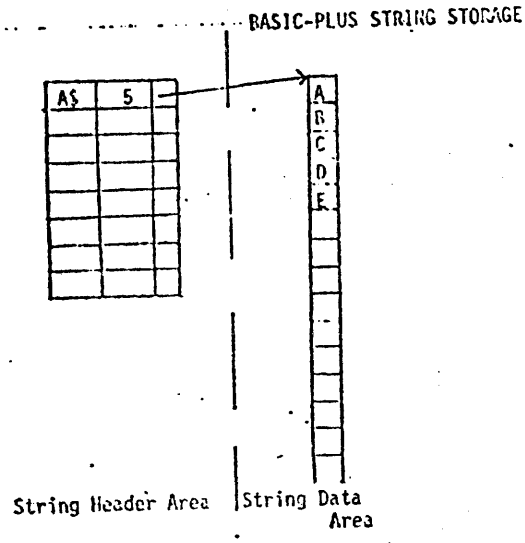


Figure 8

You will note that the excess characters were truncated (i.e. thrown away). The LSET operation can never change the length of the string nor can it move the pointer in the string header. If there are insufficient characters in the string on the right side of the equals sign then the remaining characters

are filled with blanks. There is another statement similar to LSET (although rarely used) called RSET. It differs only in that if the string on the right side of the equals sign is shorter than the one on the left side then the characters are put in the right end of the string and the left end filled with blanks. Also truncation proceeds from the left.

We said that the operation of the equivalencing of strings by having the string headers point to the same string would not cause a problem when the string is modified with a LET statement. This is valid for the LET statement but much the reverse for the LSET (and RSET) operation. The following example demonstrates what happens when two strings are equivalenced and then one is modified with an LSET statement. First we execute the following:

```
100 LET A$ = "123"
110 LET B$ = "ABCDE"
120 LET C$ = B$
```

and the string area is as shown in figure 9.

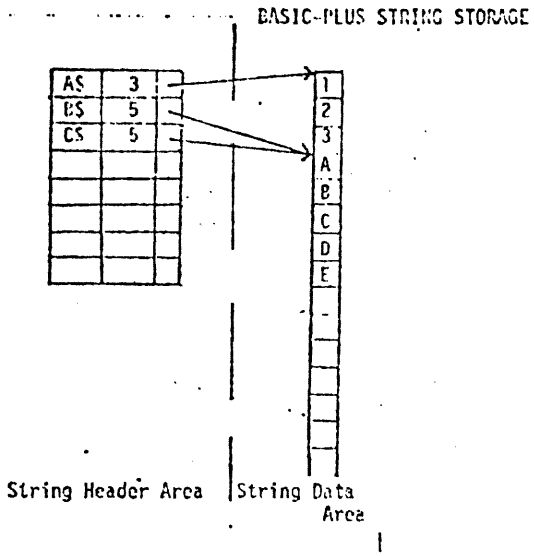


Figure 9

Now we execute the following statement:

```
130 LSET C$ = A$
```

This requests that the contents of A$ be copied into C$ and the excess over the length of C$ be truncated or that the part of C$ which is not filled by A$ be filled with spaces. The string area now is as shown in figure 10.

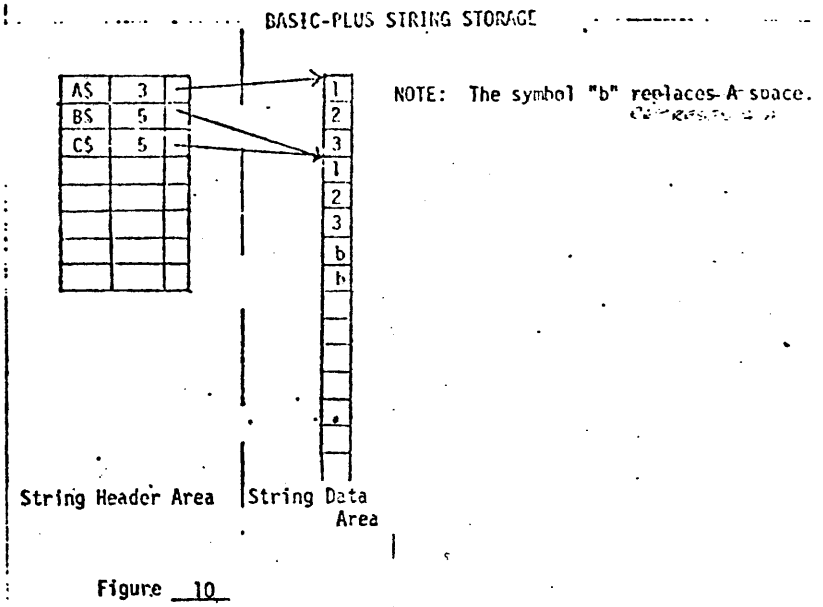


Figure 10

As you can see the string pointed to by C$ now has the same contents as that pointed to by A$. However no changes have been made to any headers and now the contents of B$ have also been changed. It's important to remember that LSET (and RSET) operate only on the string data area, never on string headers.

Now, suppose that you wanted to have two strings effectively equal but wanted them to be separate and distinct strings rather than simply two headers pointing to the same data. This is accomplished by using the LET statement and concatenating the string with the null string as shown in the following example:

```
100 LET A$ = "QWERTY"
110 LET B$ = A$ + ""
```

The effect of the operation at line 110 is to force the creation of a string in the string data area. The string area is shown in figure 11 as it would be after executing the above two statements.

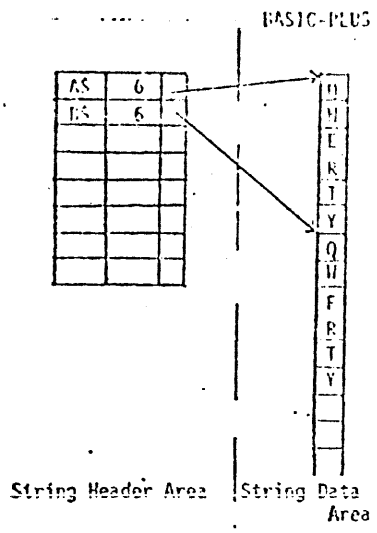


Figure 11

As you can see, the data from the string A$ has been copied into another string which is pointed to by the B$ string header. The significance of this will become more apparent when we discuss moving data out of an I/O buffer below.

NOTE: The string data structure as shown in the figures is for pedagogic purposes and not intended to completely represent the actual structure used by BASIC-PLUS which is somewhat more complex.

8. RECORD I/O - GETS AND PUTS

Now that our side trip into string data structures is complete we can return to the topic of random access input/output. We are about to consider the input/output method called "record I/O". Although this is applicable to all input/output devices, we shall limit our consideration to the disk. The disk itself is divided into units called sectors. The sector is the physical unit which is written or read. In the case of the PDP-11 the various disks have sectors of 256 words or 512 characters. This is the smallest amount of information which can be read or written and all attempts by the monitor to read or write the disk must specify an integral number of sectors. A sector is also called a block or a physical record. Even though we may need only one or a few characters from a sector, the nature of the equipment makes it mandatory that at least 512 characters be read each time from the disk. In order to accomplish this, a region of memory at least 512 characters long is set aside for the disk to read into or to write from. This region is called a channel buffer. There is one buffer for each channel which is currently open.

As mentioned above, the function of the OPEN statement is to create an association between a file or device name and a logical unit or channel. Also the OPEN command determines whether or not the requested file is present or if specified creates it. Another function of the OPEN statement is the creation of a buffer. When a channel is OPENed the program expands by 512 bytes (1/4K of core) and that area is set aside for use as a buffer for the channel. (Remember that we are talking only about disks. The size of the buffer for other devices varies according to the charac-

teristics of the device.) There are procedures whereby the buffer can be made larger and they will be described later. Once a channel is opened, then all data transfers take place between the device and the buffer and then between the buffer and the appropriate places in the user data area. Now consider the following example:

```
100 OPEN "FILEA" FOR INPUT AS 9%
110 INPUT #9%, A$, B$
120 INPUT #9%, C$
```

After statement 100 was executed, the association of name to channel would be established as would the buffer for channel 9. When statement 110 was executed the monitor would discover that the buffer was empty so it would read the first sector of the file into the buffer. Then it would find the first two strings in the buffer and copy them into the string area and set up the appropriate headers. At statement 120 there would still be information in the buffer so the monitor would set up a string header for C$ and would copy the next string in the buffer into the string data area. If during this time, the data in the buffer was exhausted then the monitor would refill the buffer by reading in the next sector. Thus with serial I/O the buffer management is entirely transparent to the programmer. Similarly, in the case of virtual arrays the data is read from a disk into the buffer and then accessed by the program but the user needn't concern himself with buffer management functions as the monitor does this for him.

In the case of record I/O the programmer is given control over the buffer and is responsible for its use. The system loads the buffer from the disk and writes the buffer onto the disk in response to specific instructions in the program. The transferring of data to and from the buffer and the use of data within the program is all undertaken by the user program. This provides for the maximum flexibility and gives the BASIC-PLUS pro-

grammer all the power in controlling devices and input/output functions that the system programmer normally has available in other systems.

The reading of data from the disk into the buffer is accomplished by the GET statement. We specify the channel number and the record (i.e. sector) in the file that we want read as follows:

```
100 GET #3, RECORD 7
```

This statement would read into the buffer the contents of the seventh sector or physical record in the file. As mentioned earlier a file on a disk consists of one or more sectors. Although their actual location is a function of the monitor's file processor, each block has a logical or relative block number. Thus the first block of the file is record 1, the next is record 2, and so on. Using record I/O we can access any block or sector of the file simply by specifying the relative record number in the GET statement. Of course before executing a GET statement it is necessary to associate a file name with the channel number through the use of an OPEN statement. This is the feature that gives rise to the description of record I/O files as random-access. We can access any (physical) record of the file without accessing those which preceed it in order to find that record. The method of searching through a file in order to find a specified record is called serial-access. If you wanted a program which read into the buffer a user specified record then the following program segment would accomplish that:

```
100 OPEN "FILE.DAT" FOR INPUT AS FILE 1%
110 INPUT "WHICH RECORD DO YOU WANT", J%
120 GET #1%, RECORD J%
  -
  -
```

In actual use the RECORD qualifier is optional. If we specify no record number the system GET's the next record which is the one next after the last access on that channel. Thus to read a file serially only a series of

GET's is necessary not specifing a record number.

To write a sector from the data in the buffer we use the PUT statement. The general format of the PUT is just like the GET except that the direction of transfer this time is from the buffer to the disk. The following statement writes the current contents of the buffer into relative record 23 of the file which is open on channel twelve:

```
100 PUT #12%, RECORD 23%
```

In the same way as the GET the RECORD qualifier is optional and its omission produces a serial write of the file. Thus to create and write a 35 block file (without attention to putting data into the buffer) the following program might be used:

```
100 OPEN "FILEB.DAT" FOR OUTPUT AS FILE 11%
110 FOR I% = 1 to 35%       !we want a thirty five block file
120 GOSUB 200               !the subroutine at 200 puts the data
                            !into the buffer.
130 PUT #11%                !write the buffer onto the disk
140 NEXT I%                 !do it 35 times
150 CLOSE #11% : STOP       !all done, close file

200 REM A subroutine to get data and move it into the buffer
 -
 -
 999 END
```

At this point the reader may be inclined to observe that although he can read and write between a channel's buffer and the disk he doesn't see any way to access the data in the buffer. That is the function of the next statement which will be described, the FIELD statement.

9. RECORD I/O - THE FIELD STATEMENT

It is the function of the FIELD statement to make accessible the data in a channel's buffer. This is accomplished through the use of the string header mechanism described earlier. The FIELD statement establishes string headers in the string header area which point not to the string data area but to the

channel buffer. Thus the FIELD statement allows us to define the record or segments of it (called fields) as string variables and then we can manipulate them using the string operators available in the BASIC-PLUS language. To set up this association of the string header to the channel buffer the FIELD statement has to have several pieces of information. It needs the channel number of the channel (in order to know which buffer is being referred to), it needs the length of the string for the string header, and it needs the name of the string. Thus a simple case of the FIELD statement might establish that the first fifteen characters in each sector of the file open on channel 4% is to be called X$ and would be written as:

```
100 FIELD #4%, 15% AS X$
```

The FIELD statement can be used to set up more than one header in one statement. Suppose that in addition to X$ as described above, we want the variable names Y$ to correspond to the next seven characters in the physical record after the fifteen characters in X$. The following statement would accomplish that:

```
100 FIELD #4%, 15% AS X$, 7% AS Y$
```

In general the FIELD statement can be continued as long as reasonable subject to the general limitation of BASIC-PLUS that no numbered line in a source program can be more than 255 characters long. Once the channel's buffer has been described in terms of string headers then the data within the buffer can be manipulated with the LET, LSET and RSET statements described earlier. Now we shall look at the mechanism by which this is accomplished.

If the following program segment were executed then the string header and storage areas would be as shown in figure 12.

```
100 OPEN "FILE" FOR INPUT AS FILE 4%
110 FIELD #4%, 10% AS X$, 3% AS Y$, 4% AS Z$
```

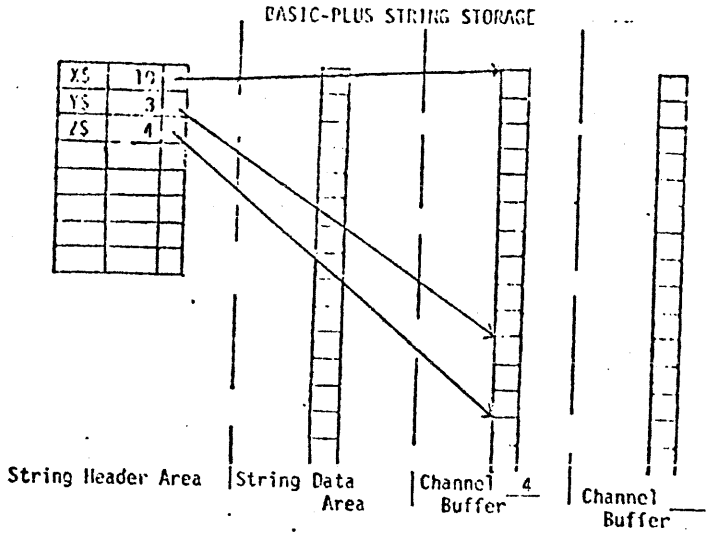


Figure 12

We now have three string pointers pointing into the channel four buffer. If we wanted to print on the terminal the contents of each record (block) the following statements would accomplish it:

```
120 FOR I% = 1% TO N%  :
     GET #4%  :
     PRINT X$, Y$, Z$  :
    NEXT I%
```

Of course a FIELD statement must not be executed before its channel is OPENed as there would not be a buffer for the string headers to point to. Now let us return to our example of the list of names and addresses. First we will accept a list of names and addresses from the terminal and then we will print them on the line printer. In order to do this with record I/O we must first set up a "Record Definition Table" which defines the fields and their lengths. Table 1 is an example of such a record definition table for this example.

RECORD DEFINITION TABLE

FIELD NUMBER	VARIABLE NAME	LENGTH	DESCRIPTION
1	N$	23	Name, last name first
2	S$	26	Street number and street
3	C$	19	City and State
4	Z$	5	Zip Code

Table 1. Record Definition Table

Now we can write the program to accept the data.

```
100 OPEN "NAMES.DAT" FOR OUTPUT AS FILE 1%          !OPEN THE FILE
110 INPUT "HOW MANY ITEMS"; N%                      !NUMBER OF NAMES &C.
120 FIELD #1%, 23% AS N$, 26% AS S$, 19% AS C$, 5% AS Z$
130 LSET N$ = NUM$(N%) :
        PUT #1%, RECORD 1%                  !STORE THE NUMBER OF ITEMS
140 FOR I% = 1% TO N%    :
        PRINT "INPUT THE NAME"   :INPUT LINE N1$ :
        PRINT "INPUT THE STREET AND NUMBER"      :INPUT LINE S1$ :
        PRINT "INPUT THE CITY AND STATE"         : INPUT LINE C1$
        INPUT "INPUT THE ZIP CODE"; Z1$
150 LSET N$ = N1$         :
    LSET S$ = S1$         :
    LSET C$ = C1$         :
    LSET Z$ = Z1$         ! MOVE THE DATA FROM THE STRING AREA INTO
                          ! THE CHANNEL BUFFER.
160 PUT #1%, RECORD I% + 1%      :
        NEXT I%           !WRITE THE BUFFER TO THE FILE
170 CLOSE 1%     :END     !ALL DONE
```

Now we have established our file of names and addresses. You will note that in the first record we stored the number of entries and used the other records to store the actual data. This is not actually necessary as BASIC-PLUS provides a method for determining the physical end of the file but it is frequently a useful technique, particularly when random access files are being used. Several system utilities such as the SORT package expect files to be set up this way. Next we want to write a program which will print on the line printer all the names and addresses for which the zip code begins with a "9". The following program will accomplish that.

```
100 OPEN "NAMES.DAT" FOR INPUT AS FILE 2%          :
        OPEN "LP:" AS FILE 3%     :
        FIELD #2%, 23% AS N$, 26% AS S$, 19% AS C$, 5% AS Z$
                  !OPEN FILES AND SET UP THE BUFFER
110 GET #2%, RECORD 1%    :
        LET N% = VAL(N$)
                  !HOW MANY ITEMS ARE IN THE BUFFER
120 FOR I% = 1% TO N%     :
        GET #2%, RECORD I%+1%
                  !FILL THE BUFFER FROM THE FILE
130 PRINT #3%, N$;S$;C$;Z$ IF LEFT(Z$, 1%) = "9"          :
        NEXT I%
                  !PRINT THE DATA
140 CLOSE 2%, 3%          :END     !ALL DONE
```

For a practical problem some editing of the input data would have been required. Now suppose we wanted to include in the line printer listing the record number of each name. We might suppose that the record number might also be a customer number or some other identification. Then we might change line 130 to be as follows:

```
130 PRINT #3%, I%+1%, N$; S$; C$; Z$ IF LEFT(Z$,1%)= "9"   :
    NEXT I%
```

Then we might want to write a program which printed out the name and address of a customer on the terminal if we were given as input the customer number. This type of inquiry-response program would make use of the random access nature of record I/O files. Look at the following program which does provide that type of information.

```
        FIELD #1%, 23% AS N$, 26% AS S$, 19% AS C$, 5% AS Z$
                !OPEN FILE AND DEFINE THE BUFFER
110 GET #1%, RECORD 1%  :
        LET N% = VAL(N$)            !NUMBER OF RECORDS
120 INPUT "WHICH CUSTOMER"; C%  :
        IF C% > N% THEN PRINT "NO SUCH CUSTOMER NUMBER", C%
        GOTO 120
                !GET RECORD NUMBER AND RANGE CHECK
130 GET #1%, RECORD C%+1%       :
        PRINT N$, S$; C$; Z$    :
        INPUT "ANY MORE"; A$    :
        GOTO 120 UNLESS A$ = "NO"       :
                CLOSE 1%        : END
```

In this program the file is opened and the field statement describes the channel buffer in terms of strings. Then the record (i.e. customer) number is obtained from the user and that record is obtained from the file. The information in the buffer is then printed out on the terminal directly from the buffer. The final few lines provide for stopping the program when the user is finished. Note also that there is a check for a customer number which is too large to be in the file.

10. LOGICAL AND PHYSICAL RECORDS

At this point we must learn the meaning of certain terms. These are logical record, physical record, blocked and unblocked records and spanned records. A record is usually thought of as an entity which is retrieved from a file

as a unit. A logical record contains all the information about something. A customer record might contain the customer name and address, his current balance, the maximum amount of credit which he is allowed, the salesman assigned to his account, and any other information which pertains to that customer. An inventory item record might contain the description of the item, the number of such items on hand, their price, whether a resupply of the items has been ordered, their cost, and such other information as would pertain to that item. The individual items which make up a logical record are usually called fields (thus the FIELD statement) and where a field is made of several more fundamental items of information they are called subfields. A physical record, on the other hand is usually defined by the nature of the recording mechanism. In the case of a disk on the BASIC-PLUS system a physical record constitutes one or more blocks each consisting of 256 words or 512 characters. Since we describe a buffer and thus a physical block in terms of characters (with the FIELD statement) we generally think in terms of a physical record of 512 characters. A physical record is the smallest entity which is retrieved from a disk file. In our example above we simply equated a physical record (containing 512 characters) with a logical record (73 characters) and wasted the remaining characters. The greatest efficiency in terms of fast response usually results when we can equate physical records with logical records but in most applications it is not practical to waste large amounts of storage to accomplish this. As a result we generally put several logical records into each physical record. This process is called blocking records. There are two methods of blocking records. In the most common as many logical records as will fit are blocked into one physical record and the remaining characters are wasted. This represents a compromise between the inefficiency of wasting a lot of space

by not blocking records and the amount of time needed to handle logical records which are spread across more than one physical record. When we do allow a logical record to cross (or span) the boundry between two physical records we refer to it as a spanned record. This is the most efficient approach in terms of the conservation of disk storage space but considerably more processing is required to handle such records so that normally in the course of our file design we try to arrange that the logical records will fill or almost fill a physical record without spanning physical records. Figure 13 shows examples of "unblocked", "blocked", and "blocked and spanned" records. Later we will cover some aspects of file design and try to show how a file system might be designed to minimize the wastage of storage without imposing the inefficiency of processing spanned records.

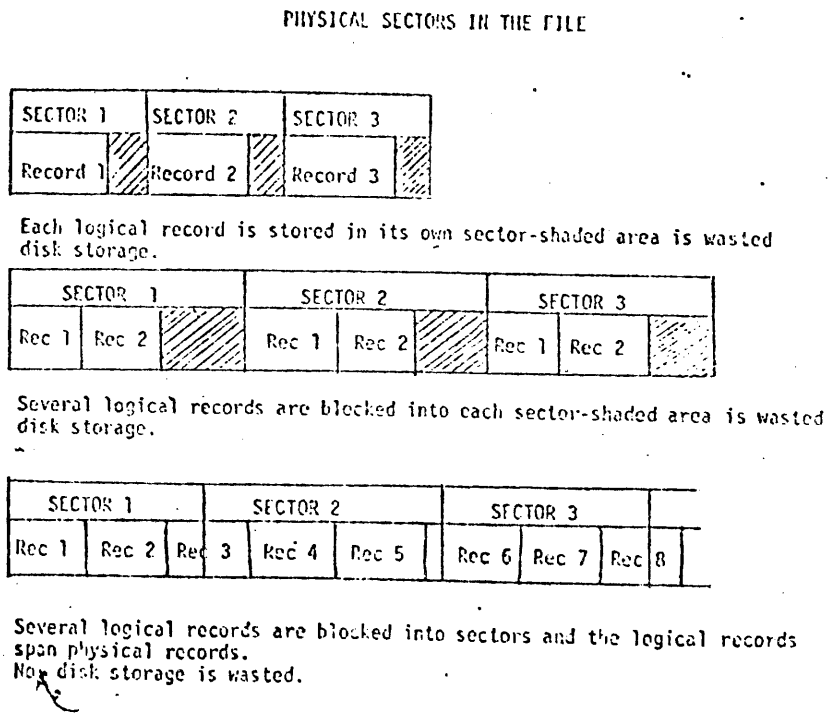


figure 13

11. HANDLING NUMBERS - CONVERSION FUNCTIONS

At this point you may have observed that no mechanism has been described which allows a number to be placed in a buffer using record I/O other than converting it into a decimal string with the NUM$ function. In terms of elemental operations there is, in fact, no provision for placing numeric items in the channel buffer. The actual hardware always works in terms of blocks of characters so the software is designed to take advantage of that characteristic of the hardware. None the less one generally wants to store numbers as well as strings of data in a file and functions have been provided for this. These functions take numbers in their internal representation (i.e. binary) and translate them into strings so that they can be placed into the buffer. Each such function has a complementary one that takes a string in the buffer and places it back into the numeric data area of the program. In BASIC-PLUS two type of numbers are available,

integers and floating point. (In some systems decimal data type replaces the floating point.) The integers occupy two bytes in core and have a range of ±32767. They also occupy two character positions in the buffer. Floating point numbers are four bytes long and occupy four character positions in the buffer. Many systems replace the floating point option with a double precision format which occupies eight bytes in memory and requires eight character positions in the buffer. Thus if we have a two byte integer K% in memory and we want to put it into a file we define a 2 character string in the channel buffer using the FIELD statement and then we move the binary value of K% into the buffer by converting K% to a string and LSETing the string into the channel buffer. If we wanted K% to be placed in the first two characters of the channel #3 buffer we would execute the following field statement:

```
100 FIELD #3%, 2% AS K$
```

and then we would place the value into the buffer by converting it to the string K$. The function which converts from integer to string is called CVT%$ and the code to put K% into the buffer would be:

```
110 LSET K$ = CVT%$(K%)
```

To retrieve an integer value from a channel buffer we first would use a FIELD statement to describe the two character positions which held that value as a string and would then use the function CVT$% (convert from string to integer) to make the conversion as follows:

```
150 LET K% = CVT$%(K$)
```

This converts the string to an integer and moves the data into the area used to store the values of numeric variables. Remember that we use the LSET statement to move the characters in a string into a buffer but a LET statement to move data into a core location (except where we deal with string headers).

We can handle floating point numbers in the same way except that we need to use four or eight character positions in the buffer and we use the function CVTF$ to convert from floating point to string and the function CVT$F to convert from a string in the buffer to a floating point number. Thus if we wanted to store two integers U% and C1% and two single precision floating point numbers X and Y in the channel 8 buffer we could use the following statement:

```
        -
        -
200 FIELD #8%, 2% AS U$, 2% AS C1$, 4% AS X$, 4% AS Y$
210 LSET U$ = CVT%$(U%)   :
    LSET C1$  = CVT%$(C1%)  :
    LSET X$ = CVTF$(X)   :
    LSET Y$ = CVTF$(Y)
        -
        -
```

Similarly we could retrieve the same data by converting it from string to the appropriate numeric form as follows:

```
        -
        -
200 FIELD #8%, 2% AS U$, 2% AS C1$, 4% AS X$, 4% AS Y$
210 LET U% = CVT$%(U$)      :
    LET C1% = CVT$%(C1$)   :
    LET X = CVT$F(X$)      :
    LET Y = CVTF$(Y$)
        -
        -
```

One more data type is provided for use in files which does not exactly correspond with any data type found in the BASIC-PLUS language. That is the byte data type. This is a numeric type used where it is desired to store a number whose value is in the range of zero to 255 in one character. This is very useful for type codes and similar data items where the range of values is limited. This type of data is transferred to and from the buffer in the same way as integers except that the strings are only one character long and the function CHR$ is used to convert from a value to a one character string and

the function ASCII is used to convert from a one character string to a value. Thus to store the small integer X% in the buffer we might write:

```
100 FIELD #8%, 1% AS X$,.....
110 LSET X$ = CHR$(X%)
```

Retrieving the value would be done similarly to retrieving integers except we would substitute the ASCII function for CVT$%.

12. MORE ON STRINGS IN BUFFERS

Since the FIELD statement is an executable statement the description of channel buffer can be changed at any time. In some cases it is convenient to have records of several different types intermingled in a file. In this case one of the characters in each record is a record type code. We GET the record into the buffer and then examine the type code using a previous field statement. Based on the value of the type code we can then execute one of several FIELD statements and process the record accordingly. Suppose we had a file TRANS.ACT which contained transactions pertaining to a supply system. If the transaction code (in character position one) was a one then the transaction was an issue and the quantity issued was in character positions two and three. If the item had been received and was to be put into the inventory then the transaction code would be a two and the number received would be in character positions three and four. In any case the stock number would be in character positions ten and eleven. Suppose the other positions were used for information needed only in other programs. If the type code was three then that would signal that this was the last record in the file and no other fields would be used. The following Record Definition Table defines the fields of the record.

Field Number	Type	Variable Name	Length	Description
Type 1 Record				
1	B	T$	1	Record type code (=1)
2	I	Q$	2	Quantity issued
3	D	D$	7	Unused
4	I	S$	2	Stock number
Type 2 Record				
1	B	T$	1	Record type code (=2)
2	D	D$	1	Unused
3	I	Q$	2	Quantity received
4	D	D$	6	Unused
5	I	S$	2	Stock number
Type 3 Record				
1	B	T$	1	Record type code (=3)

Table 2. Record Definition Table

The field types are B for byte, I for integer, F for floating point, S for string and D for dummy. Since the FIELD statement always counts characters positions from the left of the channel buffer we must space over to the fields we want by using a dummy field to describe the unused character positions. Now we want to write a program which will go through the transaction file and compute the net changes which occurred in the inventory. To make the program simple we will assume that the stock numbers are all in the range of one to twenty. The following program should make the requested calculation.

```
100 OPEN "TRAN.ACT" FOR INPUT AS FILE 1%          !OPEN THE FILE
110 FIELD #1%, 1% AS T$                           !DEFINE THE TYPE CODE
120 DIM I(20)
      I(J%) = 0 FOR J%=1% TO 20%                  !I WILL CONTAIN
                                                  !THE NET INVENTORY CHANGES. FIRST
130 GET #1%                                       !WE ZERO IT.
140 ON ASCII(T$) GOTO 200, 300, 400               !GET A RECORD SERIALLY FROM THE FILE
                                                  !GOTO THE APPROPRIATE ROUTINE
200     !HERE WE PROCESS EACH ISSUE TYPE RECORD   !TO PROCESS EACH RECORD
210 FIELD #1%, 1% AS D$, 2% AS Q$, 7% AS D$, 2% AS S$
                                                  !EXECUTE A TYPE 1 FIELD STATEMENT
220 I(CVT$%(S$)) = I(CVT$%(S$)) - CVT$%(Q$)       !UPDATE THE APPROPRIATE VALUE OF S()
      GOTO 130
300     !HERE WE PROCESS EACH RECEIPT TYPE RECORD
310 FIELD #1%, 2% AS D$, 2% AS Q$, 6% AS D$, 2% AS S$
320 I(CVT$%(S$)) = I(CVT$%(S$)) + CVT$%(Q$)
      GOTO 130                                    !UPDATE THE APPROPRIATE VALUE OF S()
400     !HERE WE PROCESS THE END OF THE FILE
410 CLOSE 1%
      PRINT J%, I(J%) FOR J% = 1% TO 20%          !PRINT OUT THE REPORT OF STOCK CHANGES
      END
```

In this program we first OPEN the file on channel one and then we execute a FIELD statement to set up the string T$ as a one byte field. After line 110 has been executed the string area is described in figure 14.

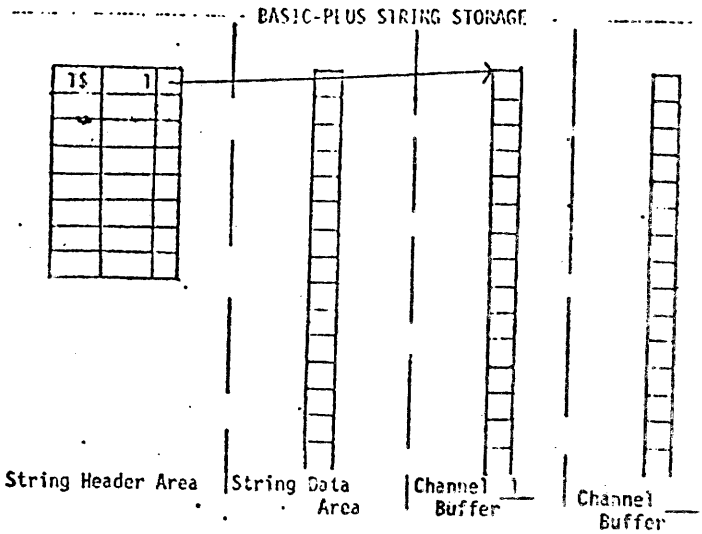


Figure 14

Then we zero out the I array which we will use to accumulate the changes to the inventory. At line 130 we GET one record and at line 140 transfer to the appropriate routine to process that type transaction. If the first record contained a type one record we would transfer to line 200 and after the field statement was executed the string area would be as shown in figure 15.

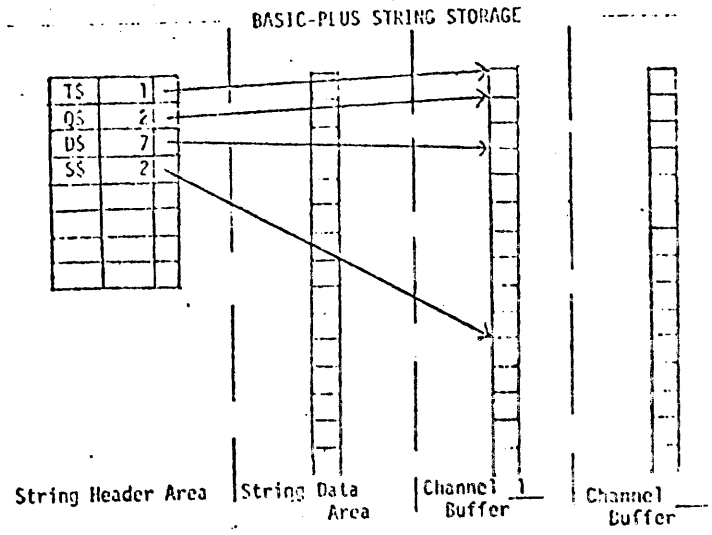


Figure 15

Note that the dummy variable D$ appears twice in the FIELD statement. This is alright since we don't really care about D$, we just want to space over the unused character positions in the buffer. When a variable is referenced more than once in a FIELD statement only the last use counts. Now at line 220 we update the appropriate entry in the I array using the CVT$% function to convert the values in the file to integer values. Since this is an issue we subtract the quantity from the value in I. Each time a type one record appears this process will be repeated. If the GET statement at line 130 gets a type two record we go to line 300 to process it. We execute the FIELD statement at line 310 and then the string area appears as shown in figure 16.

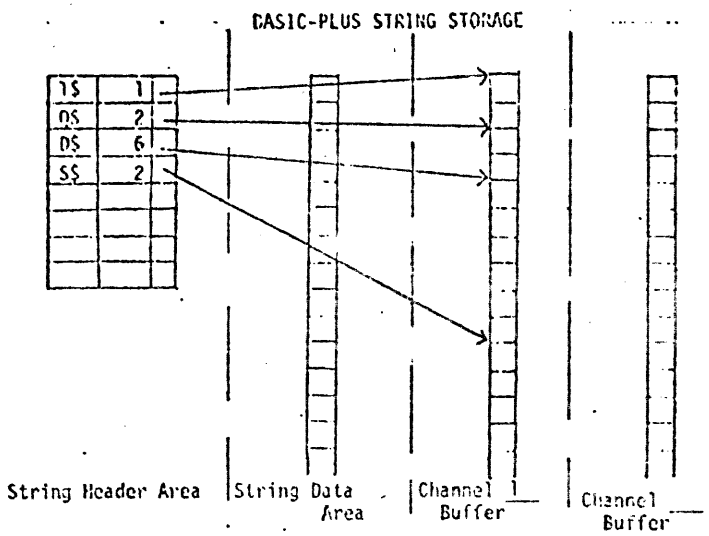


Figure 16

The FIELD statement has caused the positions of the strings (except for T$ which was not referenced in the FIELD statement) to change. At line 320 the summary data in I is updated, this time adding since this is a receipt. When a type three record is found we transfer to 400 where we close the channel and print out the summary data from I. Thus we can see that the

string definitions made with a FIELD statement are dynamic. They are established when a FIELD statement is executed and remain until another FIELD statement changes them or when a LET statement affects them. Let's look at another example. Consider the following FIELD statement:

```
100 FIELD #11%, 2% AS A$, 6% AS B$, 4% AS C$, 1% AS D$
```

At this point the string area is set up as shown in figure 17. Now we want

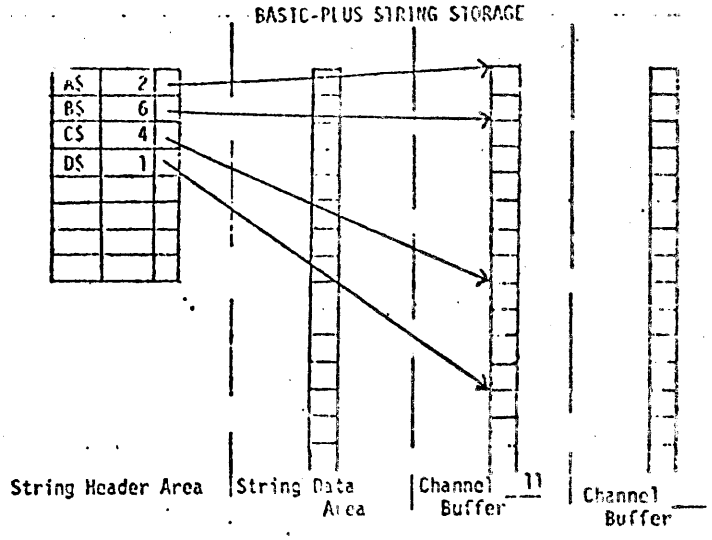


Figure 17

to put the character string "WASHR" into B$ but erroneously use the statement:

```
110 LET B$ = "WASHR"
```

The LET statement causes a new string to be set up in the string data area and not in the channel buffer. Thus we now have the situation shown in figure 18. The B$ string header points into the string data area and the

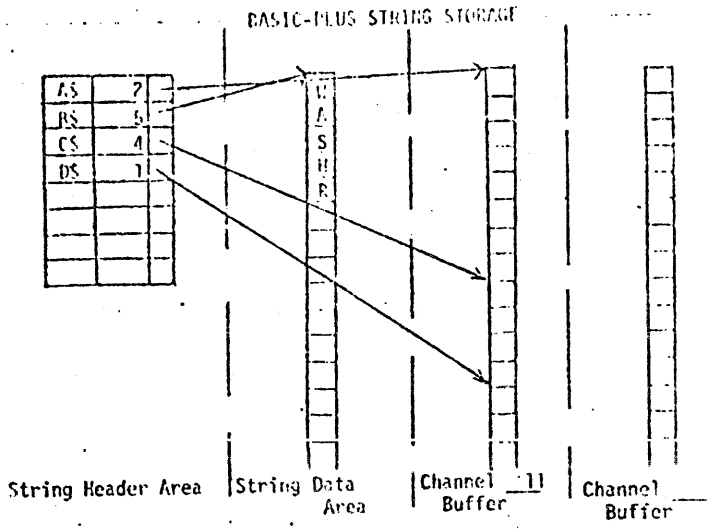


Figure 18

only way to get it to point back into the channel buffer is to execute another FIELD statement. The statement we should have used to put the string "WASHR" into the channel buffer was a LSET statement. If line 110 had been:

```
110 LSET B$ = "WASHR"
```

The string area would have been as shown in figure 19. This would have

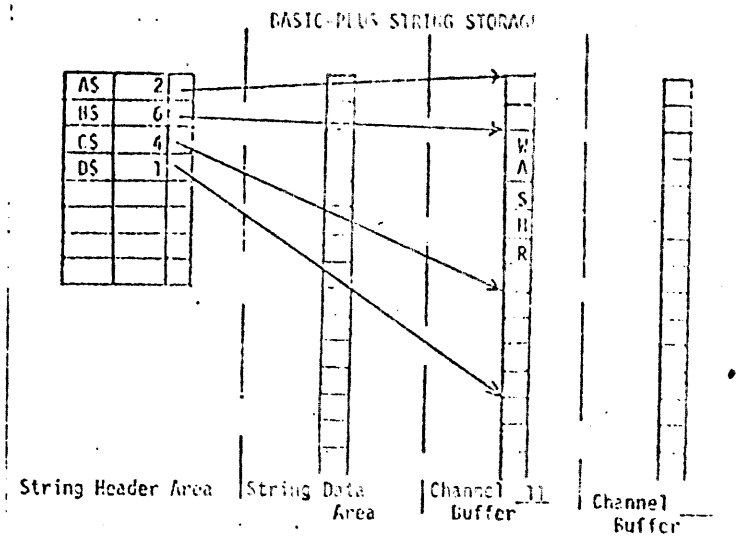


Figure 19

left the B$ pointer pointing into the channel buffer.

The capability to have more than one string header point to the same string is often used where a data item consists of sub-items. For example a six character field could contain the date in ddmmyy format. Sometimes we want to reference it as a single field and move it intact from one place to another. Othertimes we need to look at the specific date and operate on each sub-field. If the date was stored in the first six characters of the file open on channel 1 and it was to be moved to character positions four through nine of the channel buffer for file two and then the day tested to see if it was before the fifteenth of the month we could use the following program segment.

```
-
100 FIELD #1%, 6% AS D1$  :
    FIELD #1%, 2% AS T$, 2% AS M$, 2% AS Y$  :
    FIELD #2%, 3% AS D$, 6% AS D2$
110 GET #1%  :
    LSET D2$ = D1$
    IF VAL(T$)<15 THEN GOTO ....
     -
     -
```

After the three field statements in line 100 were executed the string area would appear as shown in figure 20. Note that D1$ and T$ point to the same place but their length are different. Also M$ and Y$ point into the string which is D1$ and are in fact substrings - (or sub-fields) of D1$.

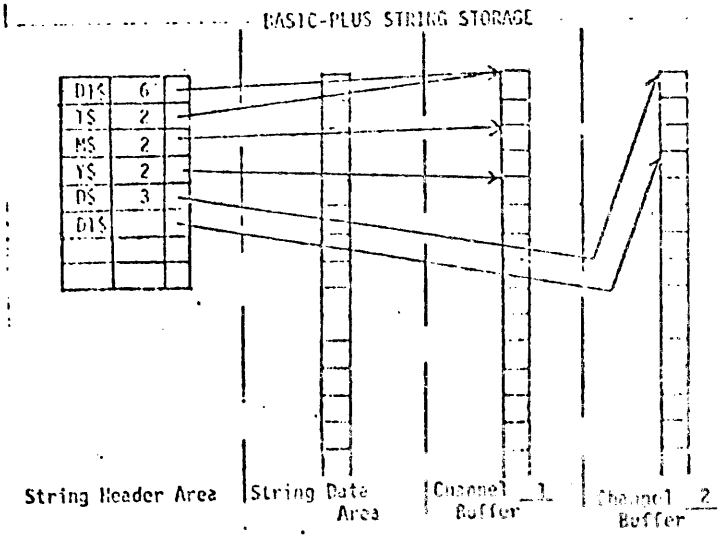


Figure 20

Another common difficulty is the confusion of strings in a channel buffer and strings in the string data area. Look at the Record Definition Table below. Here we have an inventory record where the data is the quantity on hand, the item description and the stock number. The stock number consists of two parts, a two character class code and a three character item code. Note the use of the Sub-item description.

RECORD DEFINITION TABLE

Field Number	Type	Variable Name	Length Item	Length Cumulative	Description
1	F	Q$	4	0	Quantity on hand
2	S	S$	5	4	Stock number
2.1	S	C$	2	4	Class Code
2.2	S	I$	3	6	Item Number
3	S	D$	20	9	Description of item

Table 3. Record Definition Table

The cumulative length column simply shows the number of character positions before each field. Now we want to take this record definition and process a file which is in order by stock number. Basically we are goint to print out a list of items and quantities on hand. We will also maintain two summary totals one for each class and one for the entire inventory. Each time the class changes we will print out the total quantity on hand for that class and then zero the summary total for the class. Such a program might be written as follows:

```
        FIELD #1, 4 AS Q$, 2 AS C$,  3 AS I$
                !WE USE TWO FIELD STATEMENTS TO REDEFINE THE STOCK
                !NUMBER AS THE CLASS AND ITEM NUMBERS
100 S1,S2 = 0   :
        PRINT "STOCK RECORD LISTING", DATE$(0)
                !ZERO THE SUMMARY COUNTERS AND PRINT HEADING
130 GET #1      :
        IF S$ = "99999" THEN GOTO 200                   !STOCK NUMBER 99999
                                                        !IS THE LAST RECORD
140 LET S = CVT$F(Q$)                                   !S IS THE QUANTITY
150 IF C$ <> O$ THEN PRINT "THE TOTAL FOR CLASS "; O$ "IS "; S1
        S1 = 0                                          !PRINT TOTAL FOR PREV
                                                        !CLASS AND ZERO COUNTER

160 PRINT D$, S :
        S1 = S1 + S     :
        S2 = S2 + S                                     !PRINT ITEM RECORD AND
                                                        !UPDATE COUNTERS.
170 LET O$ = C$                                         !O$ HAS THE VALUE OF
                                                        !THE PREVIOUS CLASS

180 GOTO 130 .
200 PRINT "THE TOTAL FOR CLASS "; O$ "IS"; S1   :
        PRINT   :
        PRINT "THE TOTAL FOR ALL CLASSES IS "; S2       !PRINT FINAL
                                                        !TOTALS
210     CLOSE 1 :END
```

The program is fairly straightforward. First we open the file and describe the channel buffer in line 110. We zero the variables which we will use to accumulate the necessary summary totals and then print a heading with the date. We read a record and check for the end of the file. If not we convert the quantity on hand to a floating point number (line 140) and then check to see if we have changed the class of item from the previous item. If we just read an item from a new class then at line 150 we print out the summary total from the previous class and then zero the total to use it with the next class. We print the item record at line 160 and update the totals. At line 170 we try to save the class code of this record for use in making a comparison in the next cycle and then we repeat the operation. At line 220 we print our last class total and the grand total and end the program. Unfortunately this program will not work as it contains a very common error. First let's look at the string storage area after the execution of statement 110. This is shown in figure 21. When the statement in line 170 is executed the string area has a new header, that for O$. This is

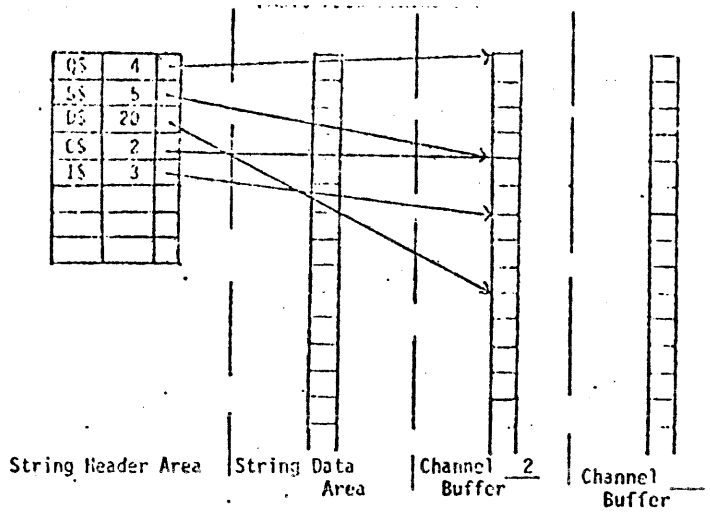


Figure 21

shown in figure 22. Unfortunately the operation simply set up a new

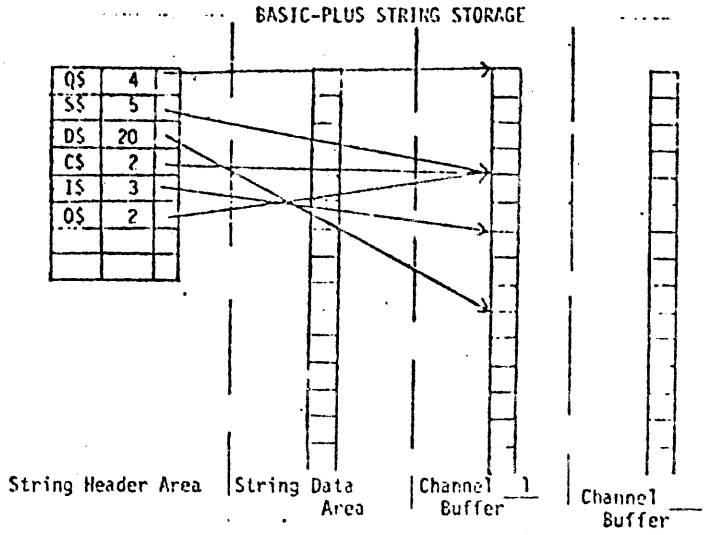


Figure 22

string header but it did not move any data (that is how a string equate is supposed to work). However, the comparison in line 150 will never succeed since C$ and O$ will always be the same since they both point to the same data. When we change the contents of C$ by reading a new record into the channel buffer we also change O$ since O$ and C$ are the same string. We can avoid this by the use of either of two mechanisms. If we simply replace statement 170 by the following:

```
170 LET O$ = C$ + ""
```

then the operation requested will force a new string to be created in the string data area. After this statement is executed the string area will be as shown in figure 23. Now we have the old value O$ stored in the string data area where it will not be affected by GETting the next record into the channel buffer.

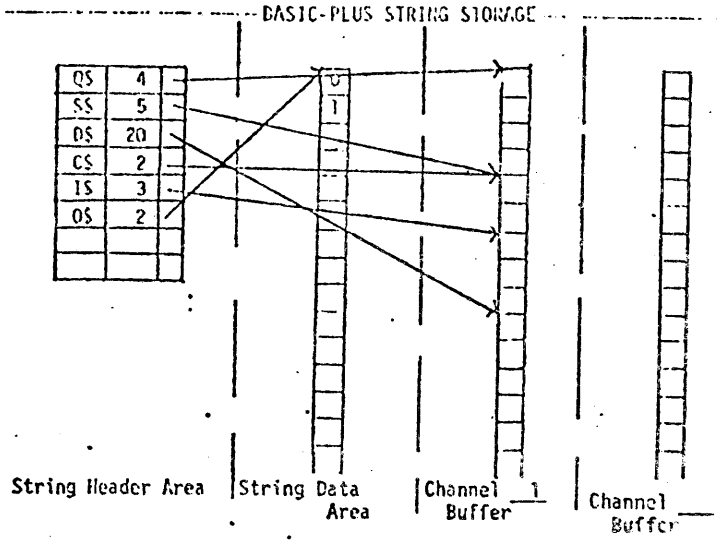


Figure 23

Another more efficient way of accomplishing the same thing would be to establish the string O$ with the correct length in the string data area at some point in the program before it was first referenced. Then we could use a LSET to move the data from the buffer into the string data area. Suppose we added a statement 105 as follows:

```
105 LET O$ = SPACE$(2)
```

Then we could replace line 170 with the following:

```
170 LSET O$ = C$
```

This would have the desired effect of moving the contents of C$ into O$ (remember that LSET can never move a pointer, only string data). Since LSET cannot alter the length of a string it is important to establish the correct length for the destination string before using LSET in this way. The problem described above, that of mistakenly moving a string pointer to point into the channel buffer instead of moving data from the buffer into the string data area is perhaps one of the most common problems encountered in using record I/O.

13 BLOCKING RECORDS

Now that we have described the use of record I/O in file processing we can look at some techniques used to increase efficiency. Earlier we introduced the concept of blocking records. At this point we want to examine the procedures used to block and deblock records. There are four operations we

want to perform: writing blocked records serially, reading blocked records serially, writing blocked records randomly and reading blocked records randomly. We will consider here only records which do not span physical sectors. The fundamental technique used in these cases is to process multiple logical records in one physical block by dynamically executing a FIELD statement. Then the particular record we want to operate on is pointed to by the string headers for the fields of the logical record. That is, if our logical record is 32 characters long and consists of two fields, A$ which is 24 characters long and B$ which is 8 characters long, we would use a FIELD statement such as the following to look at the first logical record in each block:

```
100 FIELD #1%, 24% AS A$, 8% AS B$
```

The effect of this is to describe as A$ and B$ the first 32 characters of the physical record. To examine the second logical record in the block we would use the following FIELD statement.

```
100 FIELD #1%, 32% AS D$, 24% AS A$, 8% AS B$
```

Thus the first logical record is spaced over by the "dummy" field D$ and the second logical record is described by A$ and B$. Since each physical record consists of 512 characters, each will contain exactly 16 logical records. To find the Nth logical record in a block we could set the variable N to a value between 0 and 15 and use the following FIELD statement to set up the headers for A$ and B$ to point to the particular logical record we are interested in:

```
100 FIELD #1, N * 32 AS D$, 24% AS A$, 8% AS B$
```

When this is executed it will space over the previous logical records and set up A$ and B$ as desired. To see how this would work let's see what would happen if we wanted the first logical record in the block. In this case we would set the value of N to be 0. Thus N * 32 would also be zero

(since anything multiplied by zero is zero). Thus the effect would be to execute a FIELD statement like the following:

```
FIELD #1, 0 AS D$, 24% AS A$, 8% AS B$
```

D$ would point to the beginning of the buffer as would A$ and B$ would point to the 25th character in the buffer. To operate on the second logical record in the buffer we would set N to be 1 and execute the FIELD statement described above. This time the formula N * 32 would evaluate as 32 and the effect of the FIELD statement would be to use D$ to space over the first logical record and set up A$ and B$ to point to the second logical record. To operate on the tenth logical record we would set N to be nine and the formula 9 * 32 would evaluate as 288. The effect of this would be to execute the following:

```
FIELD #1, 288 AS D$, 24% AS A$, 8% AS B$
```

The first nine logical records (288 characters in all) would be spaced over by D$ and A$ and B$ would point to the tenth logical record.

Now let's look at how we can make use of this by working with a practical example. We will have a file called "LOCATE.INV" which contains the location of items in a warehouse. Each logical record will consist of sixty-four characters as described in the following record definition table.

RECORD DEFINITION TABLE

Field Number	Type	Variable Name	Length Item	Length Cumulative	Description
1.0	S	L$	5	0	Location of item by floor, bin and rack
1.1	B	F$	1	0	Floor of building
1.2	I	B$	2	1	Bin number for item
1.3	I	R$	2	3	Rack number for item
2.0	S	N$	59	5	Nomenclature of item

Table 4. Record Definition Table for Location File

Thus there would be exactly eight logical records in each physical record and in the buffer at any time. First we must create the file and we do this by serially writing the file using the techniques described above. The following program will create the file. Note that the logical position of the item in the file is its part number. We will signal end of file by placing a count of the number of logical records in the first logical record of the file.

```
100 OPEN "LOCATE.INV" FOR OUTPUT AS FILE 1%      :
        INPUT "HOW MANY ITEMS IN THE FILE";N     :
        FIELD #1%, 5% AS L$      :
        LSET L$ = CVT%$(N)       :C% = 1%
                  !OPEN FILE AND STORE NUMBER OF RECORDS IN IT
110 U% = N%/8% + 1%
                  !U% IS THE NUMBER OF BLOCKS IN THE FILE
120 F% = 1%               !F% IS THE FIRST LOGICAL RECORD TO WRITE
125                       !THE FOR R% LOOP CONTROLS THE WRITING OF THE
                          !BLOCKS INTO THE FILE
130 FOR R% = 1% TO U%    :
        FOR J% = F% TO 7%        :
                  PRINT "LOCATION DATA FOR PART NUMBER",C%          :
                  INPUT "FLOOR, BIN, RACK"; S1%, S2%, S3% :
                  PRINT "NOW THE NOMENCLATURE OF THE ITEM"          :
                  INPUT LINE X$
135                       !GET THE INFORMATION FOR EACH ITEM (OR RECORD)
140               FIELD #1%, J% * 64% AS D$, 1% AS F$, 2% AS B$,
                          2% AS R$, 59% AS N$      :
                  LSET F$  = CHR$(S1%)     :
                  LSET B$ = CVT%$(S2%)     :
                  LSET R$ = CVT%$(S3%)     :
                  LSET N$ = LEFT(X$, LEN(X$)-2%)
145                       !MOVE THE INFORMATION INTO THE BUFFER
150               C% = C% + 1%     :
                  GOTO 200 UNLESS C% <= N :
        NEXT J%                   !UNPDATE RECORD OUT
155                       !THE FOR J% LOOP CONTROL EACH LOGICAL RECORD
160     PUT #1%, RECORD R%        :
        F% = 0% :
    NEXT R%
200 PUT #1%      :
        CLOSE 1%         :
        PRINT "ALL DONE"          :END     !SEE TEXT FOR COMMENTS
```

In this program we first set up the file in line 100 and obtained the total number of logical records to be entered. Then we put that into the file by LSETing it into the first record. Next we set up some useful constants. U% contains the maximum number of blocks we would need for the entire file. (Note that since adding a block to file takes a great deal of time and adding many blocks to a file takes very little more time than adding one block we could make the program run more efficiently by inserting the statement

```
PUT #1%, RECORD U%
```

in line 110). We also set up F% to contain a one. This is used as the first value for the FOR loop which controls the field statement and thus the number of logical records in each buffer. This is done because we will put one fewer logical record in block one since it contains the count of the records as its first logical record. Every time the FOR J% loop is executed except the first it will start at zero since we set F% to 0% at the end of each block. The data is obtained for each logical record and put into the buffer by the FIELD statement at line 140 and the LSETs which follow. We update the record count and check for the last record. If we have just entered it we goto line 200 otherwise we continue to accept records until the buffer is full at which point we goto line 160. At line 160 we write the buffer to the disk and reset F%. At line 200 we write out the last bufferload, close the file and end the program. Another point to observe is that in the statement just above line 150 we remove the last two characters of the input string since the INPUT LINE operation includes the carriage return and line feed characters and we don't want them in the file.

Although the method used above is entirely adequate there is another way of determining the record and position of the desired logical record which is frequently more useful and is essential when dealing with random access to

blocked records. This procedure uses the logical record number to obtain the physical block number and the position of the logical record within the physical record. Given the logical record number we divide it by the number of logical records in one physical record. The quotient is one less than the physical record number in which the logical record is to be found. We then obtain the remainder of the division. This is the number of the logical record within the physical record less one. In the case of the remainder we leave it less one since it is used in that way to multiply by the length of the logical record to determine the length of the dummy field used to space over to the logical record in the buffer. Considering our current example we have eight logical records in each physical record. If we want to access record N% we first divide N% by 8% and add one.

```
LET R% = N%/8% + 1%
```

We now have the actual physical record number we want in R%. We obtain the remainder from the division by multiplying the quotient by the divisor and subtracting it from the dividend. The following statement accomplishes that:

```
LET J% = N% - (R%-1%) * 8%
```

Now suppose we want to look at the location record for part number ten. First we divide ten by eight (Since we are using integer arithmetic there is no fraction when a division is performed. That is essential to the operation of this procedure.) Thus 10 / 8 gives us a quotient of one. To this we add one to obtain a physical record number of two. This makes sense as the first eight logical records are in physical block one and the next eight are in block two. Now we obtain the remainder. First we subtract one from the block number. 2 - 1 gives us 1. Then multiply by eight. 1 * 8 gives an eight. Then subtract this from the original logical record number. 10 - 8 gives us a two. Now we know that we must look at

physical record number two and multiply the length of the logical record by two to obtain the length of the dummy field. Thus we would use a FIELD statement as follows for this example:

FIELD #1%, 2% * 64% AS D$, 1% AS F%

Note that the calculation assumes that the first logical record number is number zero. If the numbering system used in the program starts at one then it will be necessary to subtract one from the logical record number before doing the calculation. This is frequently unnecessary since the technique of putting the number of logical records into the first (or zeroth if you prefer) logical record is very common.

Now we can write a simple program to list all of the location records on the lineprinter. We will use the remainder technique to find each logical record.

```
100 OPEN "LOCATE.INV" FOR INPUT AS FILE 1%          :
        GET #1%, RECORD 1%        :R0% = 1%          :
        FIELD #1%, 5% AS L$        :
        U% = CVT$%(L$)             :
        OPEN "LP:" AS FILE 2%
110     PRINT #2%, "LOCATIONS FILE ON ", DATE$(0)
120 FOR I% = 1% TO U%    :
        R% = I%/8% + 1% :
        J% = I% - (R% - 1%) * 8%          :
        IF R% <> R0% THEN GET #1%, RECORD R%     :R0% = R%
130     FIELD #1%, J% * 64% AS D$, 1% AS F$, 2% AS B$
                ,2% AS R$, 59% AS N$     :
        PRINT #2%, I%, ASCII(F$), CVT$%(B$), CVT$%(R$), N$
    NEXT I%
140 PRINT #2%    :PRINT   :CLOSE 1%, 2%    :END
```

This program is very straightforward. First the files are opened and then the number of records is obtained. The output device is opened and a header printed on it. Then in the main loop starting at line 120 the record number is calculated and the appropriate block read into the buffer if necessary. The FIELD statement sets up the string headers and the data is printed on the printer. At line 140 the files are closed and the program ends.

The only other important point to note is that the variable R0% is used to indicate which block is currently in the buffer so that no more reads from the disk are performed than necessary.

In the example programs in this section you will notice that very few line numbers are used. This is a special feature of BASIC-PLUS and enables us to make our programs more efficient. Also combining the use of only the essential line numbers with indenting the program helps make it more readable and the logic of the program more apparent.

To randomly read a blocked record from a file is similar to the last example. The desired record number is obtained and then the remainder method is used to determine the block and record within the block to read. The block is read with a GET statement and then a FIELD statement is executed which sets a dummy string across the records preceeding the desired record and sets up the string headers to point to the appropriate fields in the buffer. The following simple example requests a part number and then prints on the terminal the item nomenclature. The record is the same as used in the last example.

```
100 OPEN "LOCATE.INV" FOR INPUT AS FILE 2%
        GET #2%, RECORD 1%          :R0% = 1%
        FIELD #2%, 5% AS L$         :
        U% = CVT$%(L$)
110 INPUT "PART NUMBER"; I%         :
        IF I% > U% THEN PRINT "PART NUMBER TOO LARGE"    :
                  GOTO 110                    !CHECK RANGE OF PART NUMBER
120 R% = I%/8% + 1%         :
        J% = I% - (R% - 1%) * 8%            :
        GET #2%, RECORD R% UNLESS R% = R0%          :
        R0% = R%            :
        FIELD #2%, J% * 64% AS D$, 5% AS D$, 59% AS N$
130     PRINT "PART NUMBER"; I%; "IS A "; N$        :
        INPUT "FINISHED"; A$        :
        GOTO 110 UNLESS A$ = "YES"          :
    CLOSE 2%     : END
```

The program is very much like the previous example. Here the files are opened and the number of records is obtained. Then a part number is obtained from the terminal and is checked against the maximum record number. If it is in range then it is divided by the number of logical records in a physical block obtaining the block number as the quotient +1 and the logical record within the block as the remainder of the division. The record is read into the buffer if necessary and a FIELD statement is executed which sets up the desired logical record. Note that the variable D$ is used twice in the same FIELD statement. The first time it is used to select the desired logical record and the second time to space past the characters which we do not need for this particular program.

Randomly writing blocked records is very much like reading them. In fact it is necessary to read the record before it is written. This is done because other records which are in the same block may already have been written. Since we always read and write physical records from or to the buffer if we simply stored our information in the buffer and then did a PUT, anything which was a part of another logical record within the same block as the record we are writing would be destroyed. As an example of a program which writes a random blocked record we will take our previous example and change it so that the operator was to replace the nomenclature of the part after it had been printed out. To do this we would insert the following statements between the third and forth lines from the end of the program.

```
INPUT "NEW NOMENCLATURE": A$  :
LSET N$ = A$ UNLESS LEN(A$) = 0%  :
PUT #2%, RECORD R%  :
```

This would change the specific items nomenclature without altering any other information in the file. Observe that if no information is entered then the old nomenclature is not changed.

To summarize, to randomly read or write blocked records it is necessary to take the logical record number and divide it by the number of logical records contained in one physical block. The quotient of this division plus one is the physical block number which contains the logical record desired. The remainder of the division is the number of logical records within the block which preceed the desired logical record within the block. When it is multiplied by the length of the logical record it is the number of characters within the block which preceed the logical record. When that product is used as the length of a dummy string variable in a FIELD statement then the next character described by the field

statement is the first character of the desired logical record.

In all of our examples so far it has turned out to be very convenient that the length of a logical record is a sub multiple of the length of a block (512 characters). It may appear that this is a little unrealistic. In practical situations the logical record length is determined by the needs of the application. In most applications as many logical records as will fit are put in one physical block and the leftover characters in the block are simply wasted. In many cases the number of characters wasted is very small compared to the size of the file and this wastage is not a burden to the system. If it appears to be unreasonable for a particular application then spanned records can be used. In that case there will be no wastage but the program will be somewhat larger and will execute a little more slowly. Also certain useful functions will not be applicable to such a file. A detailed explanation of the use of spanned records must be deferred until the RECORDSIZE option is described infra. When the records do not span blocks then the method described above is applicable whether or not the records exactly fill the block. We simply divide by the number of logical records which are contained within one physical block.

14. MORE ON OPEN - RECORDSIZE AND CLUSTERSIZE

At this point we can take up several new options which are associated with the OPEN statement. First is the CLUSTERSIZE option. This is used to make access to large files more efficient. A complete discussion of the CLUSTERSIZE option is beyond the scope of these notes, however, a few comments are in order. The information which RSTS keeps on the disk is divided into three parts: the file name, where the file is, and the data within the file. In a simple case the information about where the file is stored is simply a list of the sector numbers of the sectors which make up

the file. Now we are not talking of the relative block numbers within a file which we used earlier but of the actual hardware addresses of the blocks. Each file consists of one or more blocks and the blocks are allocated from the supply of unused blocks maintained by the system. As a result the blocks are randomly arranged and while they seem sequential to the program this is accomplished by the monitor's use of the list of the blocks. In any case when a file is opened seven elements of this list are placed in core in something called a "file window". When any of the seven blocks described by the file window are requested RSTS simply looks up the actual physical address of that block in the window and then accesses the block. If a block which is not part of the window is requested then it is necessary to replace the contents of the file window. This is called a "window turn". A window turn implies reading the location information from the disk which means that before we can do our data transfer we have to wait for the disk to transfer the information about where the block is. To reduce the overhead inherent in this reading of the directory when we want to read data we have defined a structure called a "cluster". A cluster is always a power of two (1,2,4,8,16,32,64,128, or 256). When we define a file to have a clustersize of, say, 8, that means that the file is made up of clusters of eight blocks each. Now the list of locations that the monitor maintains is a list of clusters and the file window describes seven clusters rather than seven sectors. This is of course possible since it's only necessary to know where the first sector of each cluster is since the others will be immediately adjacent. The general rule to remember is that the clustersize of a file should be such that seven times the clustersize is larger than the file and seven times the next smaller

clustersize is smaller than the file. When this is not possible because

the file size is so great that it exceeds 1792 blocks then the clustersize should be 256. The clustersize of a file is established when the file is created and is used when an existing file is accessed. To create a file named "FILE.DAT" with a clustersize of sixteen we would use the following statement:

```
100 OPEN "FILE.DAT" FOR OUTPUT AS FILE 4%, CLUSTERSIZE 16%
```

The best clustersize can be calculated in a program if the size of the file to be created is known. Suppose we had a case where each logial record was to be 64 characters long. Thus exactly eight would fit into a block. The following program segment would find out how large the file was to be, open the file with the optimum clustersize, and then extend the file to its length.

```
100 INPUT "HOW MANY RECORDS IN THE FILE"; I%  :  I% = I%/8% + 1%  :
    J% = -1%
    J% = J% + 1% UNTIL 7% * (2%↑J%)> = I% or J% = 8%:
    OPEN "FILE.DAT" FOR OUTPUT AS FILE 11%, CLUSTERSIZE 2%↑J%  :
    PUT #11%, RECORD I%
```

There is also a "device clustersize." This is the minimum clustersize which any file may have on a given disk. It is set by the system manager and is designed to reduce the extent to which disk accessing slows the system down. In the case of very large disks a device clustersize is necessary simply to make it possible to address such large devices as the RP11 and RJP11 disks.

The RECORDSIZE option, on the other hand, lets us extend the length of the channel buffer and to read more information into it in one request. In our previous discussion we have always assumed that the size of a sector was 512 characters and the size of the buffer was the same as the sector. This is not entirely true. The size of the sector is indeed fixed by the hardware but the size of the buffer can be any integral multiple of the size of

the sector. Thus the buffer could be 512 characters long, or 1024, or 1536, or 2048, etc. Of course if you make the channel buffer bigger than 512 characters then either the program will grow or you will run out of room. When the buffersize has been made larger than the minimum (512 characters) then a GET will cause the entire buffer to be filled by reading as many blocks as necessary. The FIELD statement works as it did before, however, the entire channel buffer can be mapped by the FIELD statement. When a PUT statement is executed the entire buffer is written into as many (logically consecutive) blocks as necessary. A point to be cautious about, however, is that the record specification in the GET and PUT statement still specifies which block is to be the first one read. Thus if we opened a file on channel 1% with a recordsize of 1024 characters as follows:

```
100 OPEN "FILE" FOR INPUT AS FILE 1%, RECORDSIZE 1024%
```

and then we proposed to read each block in the following manner:

```
110 FOR I% = 1% TO N%  :
     GET #1%, RECORD I%  :
     FIELD #1%,.....
    -
    -
    NEXT I%
```

The effect would be that the first GET would read block number one into the first 512 characters of the buffer and then would read block number two into the next 512 characters of the buffer. The next time through the GET statement would start at block number two and would read blocks two and three into the buffer. This is not what was wanted since block two (and every other block except the first) would be read (and processed) twice. Once in the second half of the buffer and once in the first half. To make the program segment above work as desired it is only necessary to change the FOR statement to read:

```
110 FOR I% = 1% TO N% STEP 2%  :
```

Now when the program is run it will GET blocks one and two the first time, then blocks three and four, and so on.

Now we can consider spanned records. To briefly review our discussion of logical record types, a spanned record is one which may be stored in two adjacent physical blocks. This being the case it is necessary to read and write more than one block in order to access a spanned record. There are two techniques which can be used to achieve this. The simplest is described here. This is to simply open the file with a large enough buffersize to contain the desired logical record. The recordsize to use in this case is calculated as follows: When L% is the length of the logical record in characters, the recordsize is,

```
512% * ((L%/512%) + 2%)
```

For most common cases we get the following:

Size of logical record	Buffersize
less than 512	1024
exactly 512	records don't span blocks
513 through 1023	1536
exactly 1024	records only span from odd to even blocks. Use a buffer size of 1024
1025 through 1535	2048
exactly 1536	size of 1536
1537 through 2047	2560
exactly 2048	size of 2048

For the simple cases above (logical record is 1024, 1536, or 2048 characters long) all that is necessary is to multiply the desired record number minus 1 by the number of blocks in each record and then to add one to the product. Thus if the logical recordsize is 1536 (three blocks of 512 characters) and we want to access record N% we would use the following statement:

```
100 GET #4%, RECORD 3%*(N% - 1%) + 1%
```

and then process the record in the normal way remembering that the FIELD statement must map 1536 characters in the buffer if we access all of the data.

When the size of the logical record is not a convenient integral multiple of 512 then we must use a more complex calculation to determine the block where the record starts and the number of characters in the buffer which preceed the record. We use these two numbers in the same way we used them when we used the remainder method with unspanned records. In this case the calculation is as follows:

a. Multiply the record number of the desired logical record by the length of the logical record.
b. Divide that product by 512%. That is physical sect r number -1 one of the blocks which contain the logical record. Use it plus one in a GET statement.
c. Multiply the quotient obtained in step b. above by 512% and subtract it from the product obtained in step a. That difference is the number of characters in the buffer preceeding the desired record.

(NOTE: This valid for files with <=2000 blocks unless double precision arithmetic is used.)

In the following example, the length of each logical record (L%) is forty characters. We want to retrieve the N% logical record.

```
100 OPEN "FILE" FOR INPUT AS FILE 1%, RECORDSIZE 1024%
    L% = 40%
    -
    -
    -

1000 REM This subroutine reads the N% logical record from the
        file. N% is set to the logical record number to read
        and L% contains the length of the logical record.
1010 R1 = L% * N%  :
     R% = R1/512%       :
     C% = R1 - 512% *R%  :
     GET #1%, RECORD R% :
     FIELD #1%, C% AS D$,.....:
     RETURN
```

Our subroutine is called to read a logical record and it returns with the read executed and the buffer set up by the FIELD statement. As with blocked records generally it is necessary to read a logical record before attempting to write it to avoid modifying information in the other logical records in the same block.

The other method of dealing with spanned records involves reading each block in which the desired record is located and copying it into the string data area. Writing is the opposite procedure. The routines to do this are complex and beyond the scope of these notes.

15. FINDING DATA WITHIN A FILE

At this point we have learned how to open and close files and how to read and write data into them both serially and randomly. When making random access to a file we have always specified the record number of the logical record in the file that we wanted to access. In some examples we simply said that the record number would be the stock number or customer number. In practice this is rarely satisfactory as such individual identifications frequently carry more information and also may have letters as well as numbers in them. Even if such problems did not exist there is the problem of re-using numbers when one item is discountinued and another assigned the same number. In most cases it is necessary to establishe a mechanism for retrieving a record from a file based on the information within the record. Let's look at an example. We will take our example of the inventory locations file and we will include in the file an item stock number of eight characters (either numbers or letters). The record definition table for this file is now as follows:

RECORD DEFINITION TABLE

Field Number	Type	Variable Name	Length Item	Length Cumulative	Description
1.	S	L$	5	0	Location of item by floor, bin, or rack
1.1	B	F$	1	0	Floor of building
1.2	I	B$	2	1	Bin number for item
1.3	I	R$	2	3	Rack number for item
2.	S	S$	8	5	Stock Number
3.	S	N$	51	13	Nomenclature of item

Table 5. Record Definition Table for Location by Stock Number File

Our task is to write a program which will look up the location of an item given the stocknumber of the item. This might be part of a larger program which takes orders to be filled and places the items in the sequence on a packing list that will result in the shortest trip through the warehouse. At any rate our current job is to find the record for an item if we know its stock number. The simplest solution would be to simply search through the entire file checking each record until the desired record was found. This is relatively inefficient. Since each logical record is sixty-four characters long, each block will contain eight logical records. If there are 10,000 item records then the file will consist of 1240 blocks and each time we search the file approximately 625 separate disk reads will have to be made. This could take a substantial amount of time. Although this may not be very efficient it is probably still worthwhile to code it as an exercise. The following program will accept the stocknumber from the terminal and print the location of that item on the terminal.

```
100 OPEN "LOCATE.INV" FOR INPUT AS FILE 1%
        GET #1%, RECORD 1%
        FIELD #1%, 5% AS L$
        U% = CVT$%(L$)
        R0% = 1%
110 INPUT "STOCK NUMBER"; X$     : GOTO 150 IF X$ = "DONE"
        IF LEN(X$) <> 8% THEN
                PRINT "NOT A VALID STOCK NUMBER", X$
                GOTO 110
120 FOR I% = 1% TO U%
        R% = I%/8% + 1%
        R1% = I% - 8% * (R% - 1%)
        GET #1%, RECORD R% UNLESS R% = R0%
        R0% = R%
        FIELD #1%, 64% * R1% AS D$, 1% AS F$, 2% AS B$, 2% AS R$,
                8% AS S$, 51% AS N$
130     IF S$ = X$ THEN
                PRINT "FOR STOCK NUMBER "; S$, " THE LOCATION IS ";
                "FLOOR"; ASCII(F$); "BIN"; CVT$%(B$); "RACK"; CVT$%(R$)
                PRINT "THE ITEM IS: ", N$
                GOTO 100
140 NEXT I%
        PRINT "ITEM NOT FOUND"   GOTO 110
150 CLOSE 1%     :END
```

Now we can assume that on the average we will need to go half way through the file before we find the item that we want so we will need to read the file (N/16) times where N is the number of logical records in the file. If there are 10,000 logical records and we assume that each disk access takes around 200 milleseconds (one fifth of a second) then each response would take about two minutes. Not very good response from a computer. Note that the actual amount of time that it takes to access a record on a disk on a timeshared system depends in a large measure on how many other users are trying to use the disks. As a first step toward efficiency let's try to reduce the number of disk reads by setting up a separate file containing just the stock numbers. The term key is used to refer to the field (or fields) within a logical record which identify the record. In this case the stock number is the key and the file which contains the keys is the key file. In this case we will simply set up another file called LOCATE.KEY which will contain the keys in the same sequence as the records are in the main file. All we do is to sequentially read the main file (LOCATE.INV) and transfer each key to another file which we write sequentially. In each case the first logical record contains the number of logical records in the file.

```
100 REM          CREATE KEY FILE

        CREATES AN UNSORTED KEY FILE FORM
        "LOCATE.INV" AS "LOCATE.KEY"



110 OPEN "LOCATE.INV" FOR INPUT AS FILE 1% :
        GET #1%, RECORD 1% :
        FIELD #1%, 5% AS L$ :
        U% = CVT$%(L$) :
        R0% = 1%
                ! INPUT FILE INITIALIZATION
                ! U% IS NUMBER OF RECORDS IN THE FILE
                ! R0% IS THE BLOCK CURRENTLY IN THE INPUT BUFFER
120 OPEN "LOCATE.KEY" FOR OUTPUT AS FILE 2%, CLUSTERSIZE 32% :
        FIELD #2%, 8% AS Y$ :
        LSET Y$=CVT%$(U%) :
        S0% = 1%
                ! OUTPUT FILE INITIALIZATION
                ! PUT RECORD COUNT INTO FIRST RECORD
                ! S0% IS THE BLOCK CURRENTLY IN THE OUTPUT BUFFER
130 FOR I% = 1% TO U% :
        R% = I%/8% :
        R1% = I% - R% * 8%:
        S% = I%/64% :
        S1% = I% - S% * 64%:
        R% = R% + 1% :
        S% = S% + 1% !
                CALCULATE BLOCK NUMBERS AND OFFSETS
                R IS INPUT FILE AND S IS OUTPUT FILE
140     GET #1%, RECORD R% UNLESS R% = R0% :
        R0% = R% :
        FIELD #1%, 64% * R1% AS D$, 5% AS D%, 8% AS S$ !
                RETRIEVE ONE KEY FROM INPUT FILE AS S$
                R% IS THE BLOCK NUMBER AND R1% IS THE BLOCK OFFSET
150     PUT #2%, RECORD S0% UNLESS S0% = S% :
        S0% = S% :
        FIELD #2%, 8% * S1% AS D$, 8% AS Y$
        LSET Y$ = S$ !
                MOVE ONE KEY INTO OUTPUT FILE
                S% IS BLOCK NUMBER AND S1% IS THE BLOCK OFFSET
                WRITE THE BUFFER IF NECESSARY
160 NEXT I% :
        PUT #2, RECORD S0% :
        CLOSE 1%, 2% :
        END !
                WRITE OUT LAST (PARTIAL) BUFFER AND CLOSE FILES
```

the main file then there are 10,000 keys in the key file. Since each key is eight characters long, 64 of them will fit into one disk block. Thus the key file will be 157 blocks and on the average we will need to go through one-half of it to find a given key. Thus we will need to read only 79 blocks which at 200 milleseconds per read is 15.8 seconds. We still need to read one block out of the main file so the total time to retrieve the record is just sixteen seconds which is much better than two minutes. The program is not complex. It uses the remainder method to access both files. First the initialization for the input file obtains the number of records in the input file. Then we create the output file and put the number of records in its first logical record. Then in the main loop (lines 130, 140, 150) we read one logical record from channel one (the main file), extract the key, and write it into the key file. When all the keys have been extracted from the main file and transferred to the key file we write out the last buffer in the key file at line 160, close both channels and stop.

Now that we have created the key file we can use it to access a record in the main file. We will use the same problem as before. We open both files and then obtain the stock number (key) to be searched for from the terminal. Next we search through the key file for the key record which matches the stock number. When we find it we then retrieve the corresponding data record from the data file. We verify that the key in the data record is the one we are looking for. If it isn't that usually means that someone has altered or modified the data file since the last time the key file was created. In this case it is necessary to recreate the key file. If the key's do match then the required information is printed on the terminal and more input is obtained. The program stops when an error occurs or when the operator types either "DONE" or "STOP". Now here is the program which will retrieve data records after searching the key file.

```
100 REM          LOCATE RECORD IN FILE

        THIS PROGRAM FINDS A LOCATION IN THE INVENTORY LOCATIONS
        FILE BY SEARCHING THE KEY FILE FOR THE KEY AND THEN
        ACCESSING THE MAIN FILE.
101!
110 OPEN "LOCATE.INV" FOR INPUT AS FILE 1%       :
    OPEN "LOCATE.KEY" FOR INPUT AS FILE 2%       :
        FIELD #1%, 5% AS L$      :
        FIELD #2%, 8% AS Y$      :
        GET #1%, RECORD 1%       : R0% = 1%      :
        GET #2%, RECORD 1%       : S0% = 1%      !
                OPEN BOTH FILES
120     LET U% = CVT$%(L$)       :
        IF U% <> CVT$%(Y$) THEN
                PRINT "LENGTHS OF FILES DON'T MATCH" :
                CLOSE 1%,2%      : STOP  !

                U% IS THE NUMBER OF RECORDS IN THE FILE
                R0% AND S0% ARE THE BLOCK CURRENTLY IN EACH BUFFER
130 INPUT "STOCK NUMBER"; X$     :
        GOTO 180 IF X$ = "DONE" OR X$ = "STOP"   :
        IF LEN(X$) <> 8% THEN
                PRINT "NOT A VALID STOCK NUMBER"; X$ :
                GOTO 130         !
                CHECK FOR END OF RUN OR VALID STOCK NUMBER
140 FOR I% = 1% TO U%    :
        S% = I%/64%      :
        S1%= I% - S%*64%:
        S% = S% + 1%     :
        GET #2%, RECORD S% UNLESS S% = S0%       :
        S0% = S%         :
        FIELD #2%, 8% * S1% AS D$, 8% AS Y$      :
        GOTO 150 IF X$ = Y$      :
    NEXT I%      :
        PRINT "ITEM NOT FOUND IN FILE"; X$       :
        GOTO 130
141 !           MAIN LOOP.  CALCULATE BLOCK (S%) AND
                OFFSET (S1%) FOR KEY FILE.  READ RECORD AND CHECK
                FOR REQUESTED KEY.  IF NOT FOUND PRINT ERROR MESSAGE
                AND TRY AGAIN.
150 R% = I% / 8%         :
        R1% = I% - R% * 8%       :
        R% = R% + 1%     :
        GET #1%, RECORD R% UNLESS R% = R0%       :
        R0% = R%         :
        FIELD #1%, 64% * R1% AS D$, 1% AS F$, 2% AS B$,
                2% AS R$, 8% AS S$, 51% AS N$
151!
                RETRIEVE DATA RECORD FROM FILE.  R% IS
                BLOCK AND R1% OFFSET.  F$ IS FLOOR, B$ IS BIN,
                R$ IS RACK AND N$ IS DESCRIPTION.  S$ IS THE KEY.
160 IF Y$ <> S$ THEN
                PRINT "FILES ARE INCONSISTENT"   :
                PRINT I%, X$,S$,Y$       :
                PRINT "KEY FILE MUST BE RECONSTRUCTED" :
                CLOSE 1%,2% : STOP  .
161 !
                IF THE KEY IN THE DATA FILE DOES NOT MATCH THE KEY
                IN THE KEY FILE THEN THERE IS A SERIOUS ERROR.  PRINT
                DIAGNOSTIC INFORMATION AND STOP.
170 PRINT "FOR STOCK NUMBER"; S$; "THE LOCATION IS";
        "FLOOR"; ASCII(F$);
        "BIN"; CVT$%(B$);
        "RACK"; CVT$%(R$)        :
        PRINT "NOMENCLATURE"; N$;
        GOTO 130         !
                FOUND MATCH SO PRINT INFORMATION
180 CLOSE 1%, 2% : END
```

Although the above program is much more efficient than the first one it still will need fifteen seconds or more to locate the desired information and this is much longer than the operator should have to wait. We need a more efficient method of searching the key file. We can reduce the time it takes to search the key file by first putting in the file not only the key itself but also the logical record number which it represents. Then we could sort the key file so that the keys were in alphabetical order. Then with the file sorted we could use a binary search to find the requested key and along with the key would be the logical record number of the corresponding record. (if you don't know what a binary search is it will be defined below). With a binary search the number of disk reads required will be somewhat less than the power to which two must be raised to equal or exceed the number of logical records in the file. (In other words, the logarithm to the base two of the number of records in the file). Thus for a file having 10,000 records the number of disk accesses required would be Log_2 (10,000) or 14. This can be reduced by the log_2 of the number of keys stored within one block. Now our keys are somewhat longer than before. Previously we had a key of eight characters. Now we have the same eight characters plus two more used to store the logical record number (in CVT%$ format). Since each key is now ten characters long we can put 51 such keys in one block and the log_2 (51) is 5. Thus we would need to read the disk only nine times which would require only about two seconds to access the file. There are other still faster methods but first we should understand how this one works.

The obvious first question is how is the key file to be placed in order. There are many methods of sorting data and entire books have been written

about the relative advantages of each. Writing an efficient sorting program requires extensive knowledge and much effort. For the RSTS user, however, this is not necessary as an efficient file sorting package is supplied by Digital Equipment Corporation. A manual is provided which explains how the sort package is used. It both extracts the keys from the data file and sorts the keys into alphabetic sequence. As an option it will sort the data file itself but this is hardly ever necessary. The user should read the Sort manual before attempting to use the techniques described below. For our purposes it is sufficient to say that the sort package will create the sorted keyfile which we need to use with a binary search. (Note: If you don't have access to the sort package then use the randomizing or hashing technique described below.)

To make a binary search (also called the half-interval method). We simply take a sequence of values which is in order and search for one of them by seeing if it is in the first or second half of the values. If it is in the first half then we look to see if it is in the first or second half of that half and so on until we have found the value we are looking for. Although we are mostly concerned with record I/O files, we will take as our first example a binary search of a virtual array. We will use the sorted file of names which we created in the program on page [illegible]. Here we have a file of 200 names each 64 long which we have placed in alphabetical order. Now we will search for a particular name. First we open the file and declare it to be a virtual array.

```
100 OPEN "NAMES" FOR INPUT AS FILE 2%  :
    DIM #2%, N$(200)=64%
```

Next we obtain the name to search for in the variable A$.

```
110 INPUT "WHICH NAME";  A$
```

Now we set up the binary search In this case L% is the lower limit of the search and U% is the upper limit of the search. C% is calculated from L% and U% and is the center of the interval between L% and U%. Initially L% is 0% and U% is 200% thus making C% be 100%.

```
110 L% = 0%  :
    U% = 200%
120 C% = (L% + U%)/2%
```

Now we can test whether we have found A$ in the array. If we have, we transfer to line 200 with C% = the index where we found it.

```
130 IF N$(L%) = A$ THEN C% = L%  :  GOTO 200
140 IF N$(U%) = A$ THEN C% = U%  :  GOTO 200
150 IF N$(C%) = A$ THEN GOTO 200
```

Now that we know that we have not exactly found A$ we want to know whether it is above or below N$(C%) in the sorted array. When we determine this we change either L% or U% so that we now have a new interval which is just half as large as the previous one.

```
160 IF A$ < N$(C%) THEN U% = C% ELSE L% = C%
```

Now if U% and L% are the same or differ by one then we know that A$ is not to be found in the virtual array N$ otherwise we check the new interval the same as we did with the previous one.

```
170 GOTO 120 UNLESS U%-L% <= 1%  :
    PRINT "COULD NOT FIND NAME"; A$  :
    GOTO 110
```

When we find the name we could print it out or do whatever we now need to do.

```
200 PRINT "FOUND IT"; A$; "IS NAME NUMBER"; C%; "IN THE FILE":
    CLOSE 2%  :  END
```

Although it is useful for an example, the overhead associated with virtual arrays is so great that this is not a feasible way of writing a production program. In a real situation record I/O should be used when the binary search is being made. What follows is a simple program to make a binary

search of a sorted key file from our previous example; the Inventory Location File.

The method described above is reasonably effective; however, there is still substantial wasted effort. In our example we found it necessary to make nine disk accesses in order to find one data record. This is, in general, not efficient enough for many applications although there are situations in which it is fully adequate. The next increase in efficiency comes from trading off core space (and thus program size) for reduced time in locating a record. We reduce the number of necessary disk accesses to one to the sorted key file and one to the data file by maintaining some information in core which permits us to obtain the block of the sorted key file which contains the key we are seraching for. In our example we had 10,000 data records, each 64 characters long. Thus we also had 10,000 keys each of which was 10 characters long (eight for the actual key and two for the logical record number). Since each block is 512 characters long, each will hold exactly 51 keys. The sorted key file is therefore 196 blocks long. We could now set up an array in core which would contain the first key in each block of the sorted key file. Then our technique to find a specified record would be to first perform a binary search on the in-core array. The outcome of this would be the block number of the block of the sorted key file which contained the key of the record we are looking for. We would read that block and perform a binary search on its contents to determine the actual record number of the desired logical record. Then we would fetch the block from the data file which contained that record. Thus we obtain the record we desire, out of a total of 10,000 records, with two disk accesses, one to the sorted key file, and one to the data file.
If enough core to place the array of keys in core is not available, then it is possible to make that information part of the sorted key file and search the buffer twice. This increases the number of disk accesses by one but

eliminates the need for the array in core.

In our example we had 10,000 keys in 196 blocks of 512 characters. If we use a recordsize of 1024 characters we would read two blocks of the sorted key file at a time. Thus we would need on our preliminary search only 98 keys of 8 characters each. These could be put into two blocks appended to the sorted key file and then searched in the buffer by the same routine which made the binary search on the key file. Using this technique we could expect to use only six-tenths of a second in retrieving a specific record. There are mary variations on this technique which are beyond the scope of these notes.

16. HASHED KEYS

Another technique which is often used to access data is that of hashed or randomized keys. This method is based on the maintenance of a key file in a random sequence which is directly accessed based on the value of the key. The preceeding cryptic remark is best explained by a very simple example. Let us suppose that we have a key consisting of two alphabetic characters. Initially our key file is empty. As entries are made in the data file the keys are posted to the key file. The location in the key file into which the keys are placed is determined by adding the ASCII values of the two characters of the key and dividing by the number of possible entries in the key file. We use the remainder of this division as the position in the key file for that key. For our example we will use a key file having just eleven possible entries. First we will zero the key file and then consider each entry in the data file. Suppose the first key was "EG". The ASCII value of "E" is 69 and that of "G" is 71. Their sum is 140 and the remainder of 140/11 is 8. Thus in logical record 8 of the key file we would put the key "EG" and the logical record number of the corresponding data

record i.e.: 1. Suppose the next key was "FA". The ASCII value "F" is 70 and that of "A" is 65. Their sum is 135 and the remainder of 135/11 is 3. Thus we would post to position 3 of the key file the key "FA" and the logical record number of the second record. The third logical record of the data file we find has a key of "BF". The ASCII values of "B" and "F" are 66 and 70 and their sum is 136. The remainder of 136/11 is 4 so we post the key "BF" and the logical record number 3 to the key file. The key file now looks like this:

Entry Number	Key	Data Record
0	-	0
1	-	0
2	-	0
3	FA	2
4	BF	3
5	-	0
6	-	0
7	-	0
8	EG	1
9	-	0
10	-	0

At this point three keys have been entered. When we want to retrieve the record for which the key is "FA" we perform the same arithmetic operations on it and find that it is to be found in position 3 of the key file. We look there, verify that position three contains the desired key and also that the corresponding data record is in position 2 of the data file.
Now suppose that we found that the key for logical record four was "HD". The ASCII values for "H" and "D" are 72 and 68. Their sum is 140 and the remainder of 140/11 is 8. Thus we post the key "HD" in position eight, or do we? When we try to post "HD" in position eight we discover that position eight is already occupied by EG. These two keys are called synonyms and are delt with in the following way. If we try to post a key and cannot because its position in the key file is already occupied, we try the next position. If it is also occupied we try the next one. Eventually we

will find an unoccupied one and will post it there or the key file will be full. This is not very good. Generally in order to minimize the amount of searching which takes place when synonyms occur it is desirable to have a substantial amount of empty space in the key file. This is usually not a serious defect since the size of the key file is usually small compared to the size of the data file. About half again as many possible keys in the key file as there are data records in the data file is generally adequate.

Let's add two more keys to the key file. The key "AC" yields a hashed value of 1 and the key "BA" a value of 0. Now the key file looks like this:

Entry Number	Key	Data Record
0	BA	6
1	AC	5
2	-	0
3	FA	2
4	BF	3
5	-	0
6	-	0
7	-	0
8	EG	1
9	HO	4
10	-	0

Typically all we do is perform some arithmetic on the key and obtain from that a record number in a key-file. We use the remainder method to obtain that record from the key file and compare the stored value of the key with the one we are searching for. If they match we have found the record we want and can fetch it from the data file. If not we can examine the next key in the key file. Either we will find one that matches or we will find eventually an empty record. In the latter case we know that the desired key is not in the key file. One of the advantages of this technique is that we can post new records to the key file any time we want without elaborate processing. In the case of the sorted key

file adding records implied sorting the new records into the file and then reconstructing the preliminary key table. In the case of the randomized or hashed key file we obtain the record we need in typically one access to the key file and one to the data file. If there are many synonyms then we might have to read another block from the key file but that is unusual if the file is adequately large and the randomizing function is adequate. This latter is not always that easy to determine; however, one which has often been effective is to take the characters of the key from left to right; form a sum by multiplying the previous sum by three and adding the ASCII value of the next character to the sum. When all characters in the key have been summed then divide by a prime number slightly smaller than the maximum number of keys in the key file. In BASIC-PLUS this would be as follows: A$ is the key and N% is the key position. P% is the prime number just less than the maximum position in the key file.

```
100 N% = 0%
    N% = ABS(N% * 3% + ASCII(MID(A$, I%, 1%))) FOR I% = 1% TO LEN(A$) :
    N% = N% - P% * (N% / P%)
```

More complex schemes are available and can be found in many books on file and data structures.

A cautionary note is in order about the importance of not deleting entries in a hashed key file. The reason that keys must not be deleted is simply that they may be in a sequence of synonyms and replacing one by a series of blanks would cause a search to fail when in fact the desired key was simply farther down in the key file. The common solution to this (if deletions are to occur on line) is to reserve a byte in the key file to indicate that the referenced record is deleted and should not be accessed. Periodically the key file is then reconstructed from the data file and the keys corresponding to the deleted records are removed.

Here follow two programs based on our previous example, the inventory locations file. The first one takes the inventory locations file and posts a key file based on the eight character key in position 6 through thirteen. The second retrieves records by using the key file to find the record in the data record file.

```
100 REM                    CREATE HASHED KEY FILE

        THIS PROGRAM CREATES A HASHED OR RANDOMIZED
      . KEY FILE FROM THE LOCATE.INV FILE


110 OPEN "LOCATE.INV" FOR INPUT AS FILE 1%       :
        GET #1%, RECORD 1%       :
        FIELD #1%, 5% AS L$      :
        U% = CVT$%(L$)  :
        R0% = 1%
                !INPUT FILE INITIALIZATION
                 U% IS THE NUMBER OF RECORDS IN THE FILE
                 R0% IS THE BLOCK CURRENTLY IN THE INPUT BUFFER
120 U1% = 1.5 * U%       :
        J% = 1% :
        J% = J% + 1% UNTIL 7% * (2% ^ J%) >= U1% OR J% = 8%      :
        OPEN "LOCATE.KEY" FOR OUTPUT AS FILE 2%, CLUSTERSIZE 2% ^ J%
                !COMPUTE APPROPRIATE CLUSTERSIZE FOR THE
                 KEY FILE AND OPEN IT.
130 PRINT 'PLEASE INPUT A PRIME NUMBER SMALLER THAN '; U1%       :
        INPUT P%         :
        FIELD #2%, 10% AS P$     :
        LSET P$ = CVT%$(P%)
                ! P% IS THE PRIME NUMBER.
                  IT IS STORED IN THE FIRST LOGICAL RECORD OF THE
                  KEY FILE.
140 S0% = 1%     :
        FOR J% = 1% TO U1%       :
                J1% = J% / 51%   :
                J2% = J% - (51% * J1%)   :
                J1% = J1% + 1%   :
                PUT #2%, RECORD S0% UNLESS S0% = J1%     :
                S0% = J1%
                        !S0% IS THE RECORD IN THE OUTPUT BUFFER
150             FIELD #2%, 10% * J2% AS D$, 8% AS Y$, 2% AS R$   :
                LSET Y$ = SPACE$(8%)     :
                LSET R$ = CVT%$(0%)      :
        NEXT J% :
        PUT #2%, RECORD S0%
                !FILL THE KEY FILE WITH BLANKS
160 FOR I% = 1% TO U%    :
        R% = I% / 8%     :
        R1% = I% - 8% * R%       :
        R% = R% + 1%     :
        GET #1%, RECORD R% UNLESS R% = R0%       :
        R0% = R%         :
        FIELD #1%, 64% * R1% AS D$, 5% AS D$, 8% AS S$
                !RETRIEVE EACH RECORD FORM THE DATA FILE
170     J% = 0% :
        J% = ABS(J%*3% + ASCII(MID(S$,K%,1%))) FOR K%=1% TO 8%   :
        J% = J% - P%*(J%/P%)
                !CALCULATE THE HASHED VALUE OF THE KEY
180     J% = 1% IF J% = 0%       :
        J1% = J% / 51%   :
        J2% = J% - 51% * J1%     :
        J1% = J1% + 1%   :
        GET #2%, RECORD J1% UNLESS J1% = S0%     :
        S0% = J1%
                !DETERMINE POSITION IN KEY FILE FOR THIS KEY
190     FIELD #2%, 10% * J2% AS D$, 8% AS Y$, 2% AS R$   :
        IF Y$ <> SPACE$(8%) THEN
                J% = J% + 1%     :
                J% = 1% IF J% > U1%      :
                GOTO 190
                        !SEE IF THE J TH POSITION IN THE KEY FILE
                         IS EMPTY AND IF NOT TRY J%+1%
200     LSET Y$ = S$     :
        LSET R$ = CVT%$(I%)      :
        PUT #2%, RECORD S0%      :
    NEXT I%
                !OK FOUND A HOLE FOR IT SO WRITE IT OUT AND TRY THE
                 NEXT DATA RECORD
210 CLOSE 1%, 2%         :
        PRINT 'ALL DONE'         :END
```

The program to create the hashed key file starts at line 110 by opening the input file and determining the number of records in it. At line 120 the size of the hashed key file is determined by multiplying the number of records in the data file by 1 1/2. Then the cluster size for the key file is determined and the key file is opened for output. At line 130 a prime number which will be used in the hashing algorithm is obtained from the terminal. This could have been calculated by the program. The prime number is stored in the first (i.e.: zeroth) logical record of the key file. This is so that the program(s) which access the key file will know which prime number to use. The FOR loop at line 140 and 150 serve to fill the key file with blanks. These initial values in the key file are necessary so that we can determine, when we post the records to the file, whether or not a particular key file position has been already used and when we retrieve records to determine if a key is not in the file. The main loop in the program, extending from line 160 to line 200 actually posts the keys from the data file to the key file. The code in line 160 serves to sequentially read each record in the data file. At line 170 each key is hashed and its position in the key file (J%) is determined. At line 180 that record is read from the key file and (at 190) if it is blank the key and the logical record number of the corresponding data record is written to the key file. If the position is not blank then a synonym has occurred and we try the next key position. In this code we test for a possible "wrap-around" condition in which we search a chain of synonyms off the end of the file. In this case we simply go to record number ONE. (Remember that record zero contains the prime number.) Finally at 210 we close the files and terminate the program. At this point we have a data file, "LOCATE.INV" and its corresponding key file, "LOCATE.KEY".

We are now ready to randomly access records in the data file using information contained in the key file. The next program does that. It obtains a stock number from the terminal, locates the record in the data file by using the key file and then prints the location information on the terminal.

At line 110 the data and key files are opened and the number of records in the data file as well as the prime number for the hash code computation are read from the files. In line 130 we obtain the stock number from the terminal and check for a valid length and for the end of the job. If neither we calculate the hash code at line 140 and retrieve that key from the key file at line 150. Line 160 checks to see if there is a valid key at that location. If not then there is no record for the specified stock number and the terminal user is so informed. If the key is not blank then a check is made to see if the desired stock number is there or if it is a synonym. If the latter is present then the hashed index is incremented and we try the next key. When the key is found then the record specified by the last two characters of the key (I%) is retrieved from the data file. This occurrs at line 190. A check is made to verify that the key in the key file and the key in the data record are the same. If not then the files are inconsistent (i.e. one has been altered since the key file was created). and the key file must be recreated. At line 200 the requested information is printed on the terminal and the program asks for another stock number.

It is worthwhile to compare this program with the one which did the same job by serially searching the key file to see the reduction in the number of disk accesses. Typically only two disk accesses are required to locate any record in this file.

```
100 REM                    LOOKUP RECORD IN FILE

      THIS PROGRAM FINDS A LOCATION IN THE INVENTORY
      LOCATIONS FILE BY HASHING THE KEY SUPPLIED FROM THE
      TERMINAL AND LOOKING IN THE HASHED KEY FILE FOR THE
      RECORD NUMBER OF THE DATA RECORD
101!
110 OPEN "LOCATE.INV" FOR INPUT AS FILE 1%          :
    OPEN "LOCATE.KEY" FOR INPUT AS FILE 2%          :
        FIELD #1%, 5% AS L$          :
        FIELD #2%, 10% AS P$         :
        GET #1%, RECORD 1%           : R0% = 1%     :
        GET #2%, RECORD 1%           : S0% = 1%
                !OPEN BOTH FILES
120     U% = CVT$%(L$)   :
        P% = CVI$%(P$)
                !U% IS THE NUMBER OF RECORDS IN THE DATA FILE
                 P% IS THE MAGIC PRIME NUMBER USED FOR HASHING THE KEY
                 R0% AND S0% ARE THE RECORDS CURRENTLY IN THE BUFFER
130 INPUT "STOCK NUMBER"; X$     :
        GOTO 210 IF X$="DONE" OR X$ = "STOP"     :
        IF LEN(X$) <> 8% THEN
                PRINT "NOT A VALID STOCK NUMBER"          :
                GOTO 130
                        !GET DESIRED STOCK NUMBER AND CHECK LENGTH
140 J% = 0%      :
        J% = ABS(J%*3% + ASCII(MID(X$,K%,1%)))FOR K% = 1% TO 8% :
        J% = J% - P% * (J%/ P%) :
        J% = 1% IF J% = 0% .
                ! CALCULATE THE HASHED KEY VALUE
150 J1% = J%/51%          :
        J2% = J% - 51% * J1%     :
        J1% = J1% + 1%   :
        GET #2%, RECORD J1% UNLESS J1% = S0%     :
        S0% = J1%
        FIELD #2%, 10% * J2% AS D$, 8% AS Y$, 2% AS P$
                !GET THE INDICATED KEY
160 IF Y$ = SPACE$(8%) THEN
      PRINT "THERE IS NO RECORD CORRESPONDING TO STOCK NUMBER "; X$
        GOTO 130
170 IF Y$ <>  X$ THEN
        J% = J% + 1%     :
        J% = 1% IF J% >1.5 * U% :
        GOTO 150
                ! GOT A SYNONYM FOR X$ SO TRY THE NEXT POSITION
180 I% = CVT$%(R$)       :
        R% = I% /8%      :
        R1% = I% - R% * 8%       :
        R% = R% + 1%
        GET #1%, RECORD R% UNLESS R0% = R%       :
        R0% = R%         :
                !GET THE BLOCK WHICH HAS THE DATA RECORD
190     FIELD #1%, 64% * R1% AS D$, 1% AS F$, 2% AS D%, 2% AS S%,
                8% AS S$, 51% AS N$      :
        IF Y$ <> S$ THEN
                PRINT "FILES ARE INCONSISTENT - KEY FILE MUST BE";
                        " RECONSTRUCTED."        :
                CLOSE 1%, 2%     :STOP
                        !A VERY SERIOUS PROBLEM HERE
200 PRINT "FOR STOCK NUMBER  "; S$; "  THE LOCATION IS:  ";
        PRINT "FLOOR  "; ASCII(F$);
                "BIN  ";CVT$%(D$);
                "RACK  "; CVT$%(R$)      :
        PRINT "NOMENCLATURE  "; N$       :
        GOTO 130
205 !           FOUND THE ITEM SO PRINT THE INFORMATION IN THE FILE.
210 CLOSE 1%, 2%         : END
```

17. VARIABLE LENGTH RECORDS

In all of the previous discussions it has been assumed that the individual records in the files were all of the same length. This is the case in most data processing applications. Usually comparable information is to be maintained pertaining to each item described by the data in the file. There do arise, however, cases in which it is desirable to have records of varying length. There are several ways to handle these. If there is no need for random access then the data can simply be processed using the serial input/output statements, INPUT and PRINT. In this case the records are delimited in the file by carriage-return line-feed characters and the record deblocking is handled by the BASIC-PLUS run time system. Generally the length of each record is stored in the file either explicitly as a character count or implicitly as a character which separates the records. When random access is needed then some kind of key file must be used. After the data file has been created it is necessary to create a key file. It would have within it the key, the block within which the record starts and the character position within the block where the record starts. This file could be arranged in the same sequence as the data file, it could be sorted, or it could be randomized as described above. Whichever procedure is used the technique used to randomly locate a variable length record in a file is to determine the block and starting address of the record from the key file and then to GET the block which contains the record. The length is determined from the record and a FIELD statement is

used to map the record. Then the data is available for processing unless the record spans a block in which case it is necessary to move the partial record into the string data area and GET the next block. The remainder of the record is then concatenated, in the string data area, with the first part and the record can be processed. Rewriting a variable length record in situ is possible as long as the length of the record does not increase. If the length increases then more complicated methods are needed. One which is sometimes used in this case is to provide a field in the record to indicate where the record has been moved when it was rewritten and to provide some space at the end of the file for writing records which no longer fit in their original positions. Thus when a record is altered so that its length is changed it is rewritten at the end of the file. In its old location (the one pointed to by the key file) the record is changed so that a field contains the block number and character offset where the record is now located. When the record is accessed it is first necessary to determine whether the record has moved and if so to access it at its new location. Alternatively the record could be rewritten at the end of the file and the key file updated to point to the new location of the record. In this type of file processing it is necessary to provide programs which re-organize the files so as to reclaim the space lost when a record is rewritten in another location or when a record becomes shorter. Such a program would copy the data file, eliminating the unused records and would reconstruct the key file.

Now let's look at an example of this type of file. We will have a file which contains information about inventory items. In particular it will contain records of purchases of items and the prices paid for the items. This is used to calculate an aged cost of inventory. Thus we can take the number of items on hand and assume that they represent the last purchases

of those items. We take the number received most recently and subtract it from the number on hand. If there are more on hand then we take the next most recent receipt of those items and subtract it from the remainder on hand and so on until all items on hand have been assigned to the receipts. Then costs can be calculated by applying the cost data for each receipt to the number of items on hand from that receipt. When items are received three new fields are added to the record pertaining to that item. Periodically the fields which no longer are applicable (i.e. have been exhausted) are removed from the file. The record layout for the data file is as follows:

RECORD DEFINITION TABLE

Field Number	Type	Name	Length	Description
1	I	L$	2	Length of record in bytes
2	I	Q$	2	Quantity on hand
3	S	S$	8	Stock number (key)
4.	I	B$	2	Block for link record
5.	I	O$	2	Offset for link
6.1	S	J$	9	Date of receipt
6.2	I	N$	2	Quantity received
6.3	F2	C$	8	Unit cost

(NOTE: 6.1-6.3 repeat as many times as needed)

Table 6. Record Definition Table

The first field in each record is the character count for the record, i.e. the number of characters in the record. The next field is the quantity of each item which is on hand. The third field is the key, an eight character alphanumeric stock number. This is what we will use to post to the key file and to access the record randomly. We will maintain the file in stock number sequence for ease in generating reports. The next two fields are used to indicate that the record has been moved in the course of being rewritten. If CVT$%(B$) is 0% then the original record is the valid one and can be used . Otherwise B$ contains the block number of the block where

the record has moved and O$ contains the offset into the block in characters. Even after pursuing the record in this manner it is necessary to check B$ and O$ in the new record to see if it has moved from there. The actual data cannot be used until the record for that stock number has been located with O$ containing a zero. Next in the record we have the variable length elements J$, N$ and C$. These contain the date of the receipt in DD-Mmm-YY format, the quantity received, and the unit cost of the items. These three fields are repeated once for each active receipt. Thus a record which has one active receipt will consist of 35 characters; one which has two active receipts will consist of 54 characters and so on. Let's first consider a program to create a hashed key file. At the beginning of the data file we have some useful information. In the first logical record we have the total number of records in the file, the block and character offset of the next available position in the file (for use in rewritting records which grow). Finally we have the actual length of the file in blocks. After we have the key file established then we can write a program which processes incoming receipts and updates the quantity on hand and adds a field which describes the new receipt. Finally we could print out the aged inventory report. We would maintain the data file in stock number sequence. The record format for the key file is eight characters of key, two of block number (in CVT%$ format) and two for the character offset into the block (also in CVT%$ format). There are thus 42 keys per block and each key is 12 characters long. Now let's look in some detail at the program which creates the hashed key file. It reads the file serially assembling each logical record and posts to the key file the position of each data record and the key. Note that logical records may be allowed to span blocks. In some cases (where the records are short even if variable in length) this can be avoided by simply placing a signal

character (generally a CHR$/255%) in the first character following the last logical record which would fit in a block. Then the program would simply get the next block and would not have to try to assemble a record from two blocks. This does waste some storage space but can result in more efficient programs.

18. FILE DESIGN

These notes are intended to provide some of the technical information about the file handling facilities available in RSTS and some commonly used techniques. At this point you should have a reasonably complete understanding of record I/O and disk files. What follows are a few comments on recommended design practices. This is hardly comprehensive. If you want a discussion of files in general then a textbook on data processing techniques is in order. When designing the files for a system it is wise to keep in mind the following points:

1. The larger a file is (total size) the longer it takes to access a single record. This is somewhat true regardless of the type of access. If the records in a file are large then consideration could be given to dividing the file into two (or more) files each with smaller records. This is particularly effective when only some of the information in a record is used online and the rest is used only for off-line processing.
2. The "natural" sizes of logical records are powers of two characters less than or equal to 512. (i.e. 1, 2, 4, 8, 16, 32, 64, 128, 256, or 512 characters per logical record). If the logical records are equal to or slightly less than these lengths then blocks need not be spanned and little disk space need be wasted. Spanned records cause the program to be larger than otherwise and probably to execute more slowly.