

RTE - IVB

Session Monitor

User's Student Course Book
Volume II



22999-90220

September 1, 1979
updated August 1, 1980

22999-90220 Session Monitor User Student Workbook

The following pages were updated in this manual Aug.80:

1-28	2-35	3-2	4-2	5-10 thru 14	6-3
-32		-7	-6	-16	-12
-34		-9	-32	-18	-13
-35		-14		-21	-26
		-18 thru -45		-22	

7-6	8-9	9-4	10-12	12-20	Title pg.
-7	-13	-6	-32		Chpt. 17

20-2
-3
-6 thru -20

Total=80 pages

TABLE OF CONTENTS — VOLUME 2

CHAPTER

- 11. DISC CARTRIDGES
 - A. More on Using Disc Cartridges
 - B. Saving/Restoring Cartridges
- 12. SPOOLING
 - A. What is Spooling?
 - B. Using Spooling Interactively
 - C. How Spooling Works
 - D. Monitoring Spooling — GASP
- 13. BATCH PROCESSING OF JOBS
 - A. FMGR and Batch Jobs
 - B. Using Batch Processing
 - C. Monitoring Jobs — GASP
- 14. SYSTEM CONSOLE
- 15. TIME-SCHEDULED PROGRAMS
- 16. PROGRAMS SCHEDULING OTHER PROGRAMS
 - A. EXEC Scheduling Calls
 - B. Passing and Returning Information
 - C. Program Termination

TABLE OF CONTENTS — VOLUME 2

CHAPTER

- 17. CLASS I/O
 - A. Program to Program Communication
 - B. CLASS I/O for Program to Program Communication
 - C. CLASS I/O — a Summary of Features
 - D. CLASS I/O for Device I/O and Control
 - E. Variations with CLASS I/O
 - F. Terminal Handlers

- 18. MORE RTE SERVICES
 - A. Resource Numbers and LU Locks
 - B. EXEC Calls to the Disc
 - C. Large Programs

- 19. EXTENDED MEMORY AREA (EMA)
 - A. What is EMA?
 - B. Using EMA from FORTRAN
 - C. How EMA Works

- 20. LIBRARIES
 - A. System Library
 - B. Relocatable Library
 - C. Decimal String Arithmetic Library

APPENDIX

- A. LAB EXERCISES
 - Labs 11 to 20

HP 1000 RTE-IVB/SESSION MONITOR USER'S COURSE STUDENT WORKBOOK — VOLUME 2

This volume of the Student Workbook is for use during week 2 of the 2 week HP 1000 RTE-IVB/Session Monitor User's Course.

The schedule below indicates the chapters of the Student Workbook to be used during the week and the corresponding lab exercises. The topics to be discussed in each chapter are summarized in the Table of Contents.

	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY
8	Review 11. DISC CARTRIDGES	Review 14. SYSTEM CONSOLE	Review 17. CLASS I/O	Review 18. MORE RTE SERVICES	Review 20. LIBRARIES
9		15. TIME-SCHEDULED PROGRAMS	• program to program communication		
10	12. SPOOLING	LAB 14,15	LAB 17a	LAB 18	LAB 20
11					
12					
1	LAB 11, 12	Review 16. PROGRAMS SCHEDULING OTHER PROGRAMS	Review (MORE CLASS I/O)	Review 19. EXTENDED MEMORY AREA (EMA)	COURSE SUMMARY — OPEN LAB
2			•		
3	Review 13. BATCH PROCESSING OF JOBS	LAB 16	LAB 17b	LAB 19	
4	LAB 13				
5					

11 USING DISC CARTRIDGES



SECTION

- | | | |
|----------|------------------------------------|--------------|
| A | USING CARTRIDGES | 11-3 |
| B | SAVING/RESTORING CARTRIDGES | 11-14 |

11A. USING CARTRIDGES

Before you can access a cartridge, the cartridge must be entered in two lists.

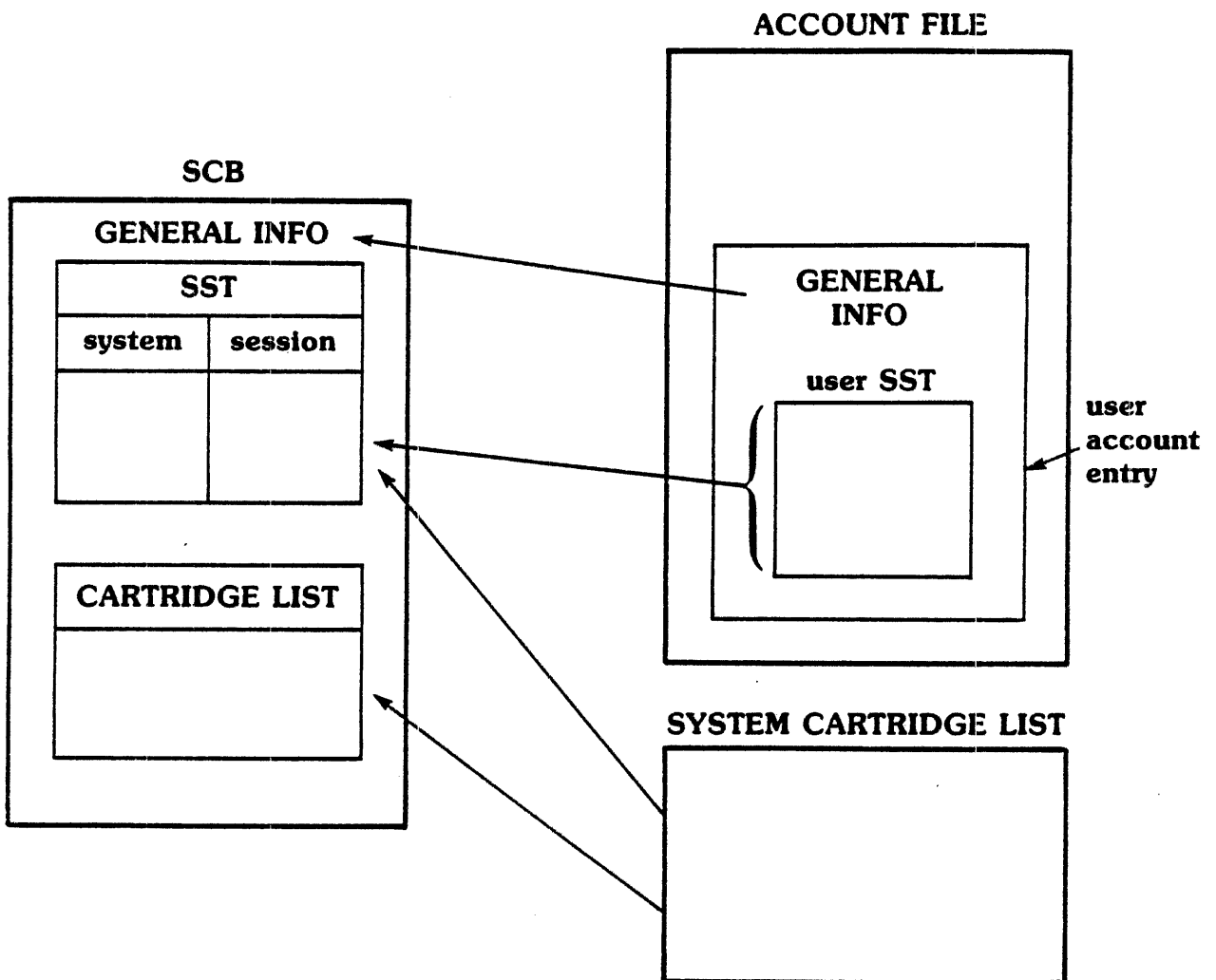
SYSTEM CARTRIDGE LIST — a permanent list (on LU 2) of all cartridges currently mounted in the system.

SCB CARTRIDGE LIST — a temporary list (in your SCB) of the cartridges currently mounted to your account (P) or to your group (G).

- **When you allocate a cartridge (AC command), the File Management System adds the cartridge to both lists.**
- **When you log-off and save private and/or group cartridges, the cartridges are *not* removed from the System Cartridge List.**

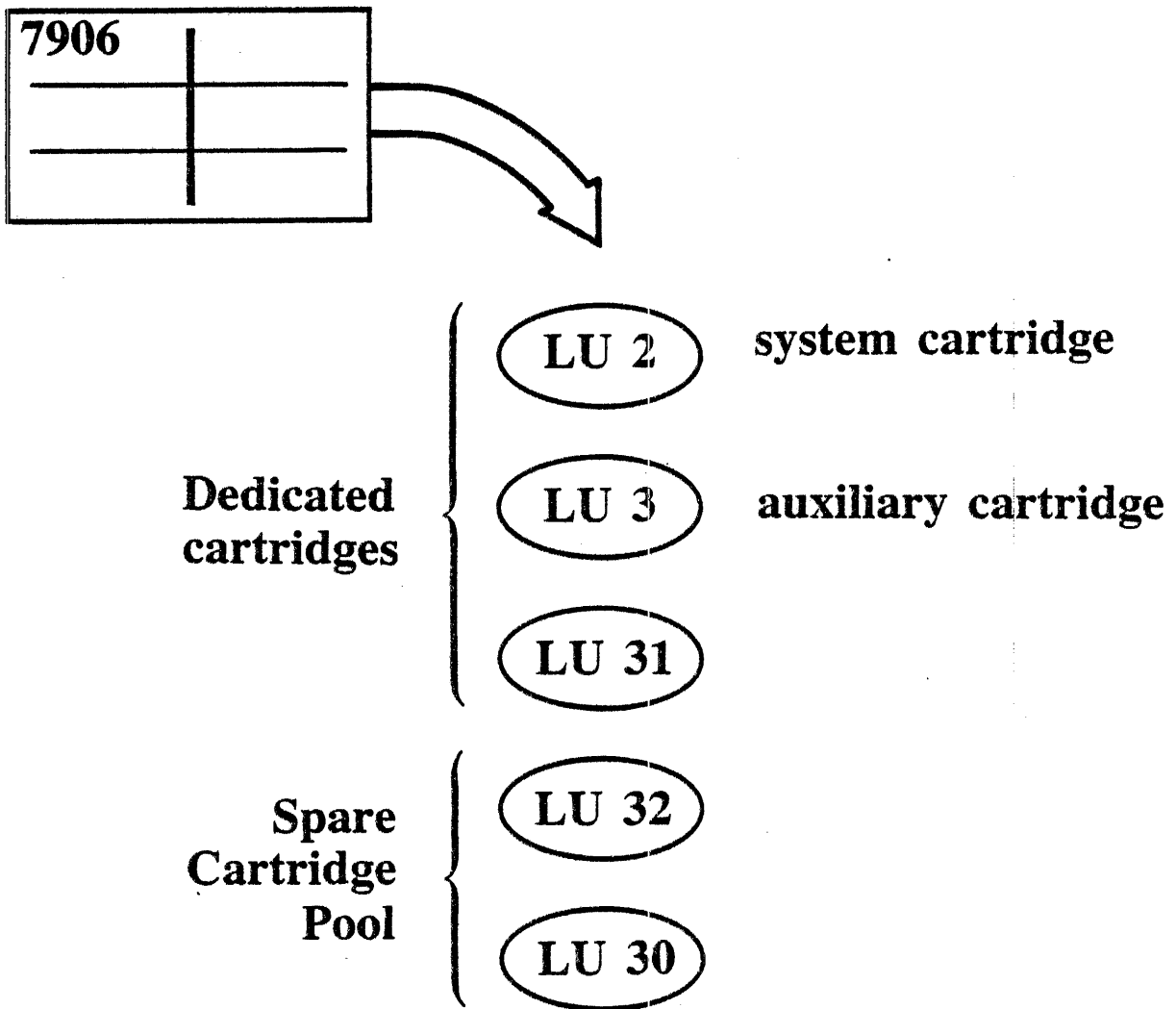
LOG ON

When you log-on, the system builds an SCB in SAM for your session. The SCB describes your session's capabilities and limitations.



WHAT TYPE OF CARTRIDGES?

Disc cartridges can be either dedicated cartridges or spare pool cartridges. For example,



ALLOCATING CARTRIDGES

: AC

You request the use of a disc cartridge from the spare cartridge pool with the FMGR AC command.

```
:AC,crn [ ,P [ ,G [ ,size [ ,id [ ,#directory tracks ] ] ] ] ] ] ]
```

```
:CL
LU  LAST TRACK  CR  LOCK  P/G/S
02  00255  00002  .  S
03  00255  00003  .  S
31  00400  00031  .  S
```

```
:AC,SS
```

```
:CL
LU  LAST TRACK  CR  LOCK  P/G/S
32  00140  SS  .  P
02  00255  00002  .  S
03  00255  00003  .  S
31  00400  00031  .  S
```

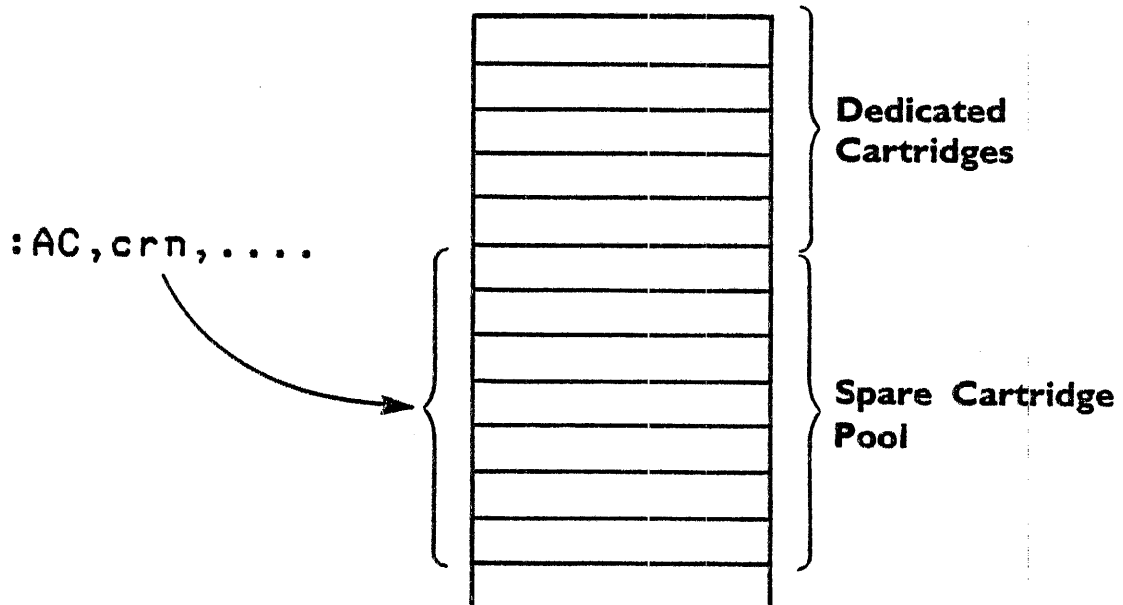
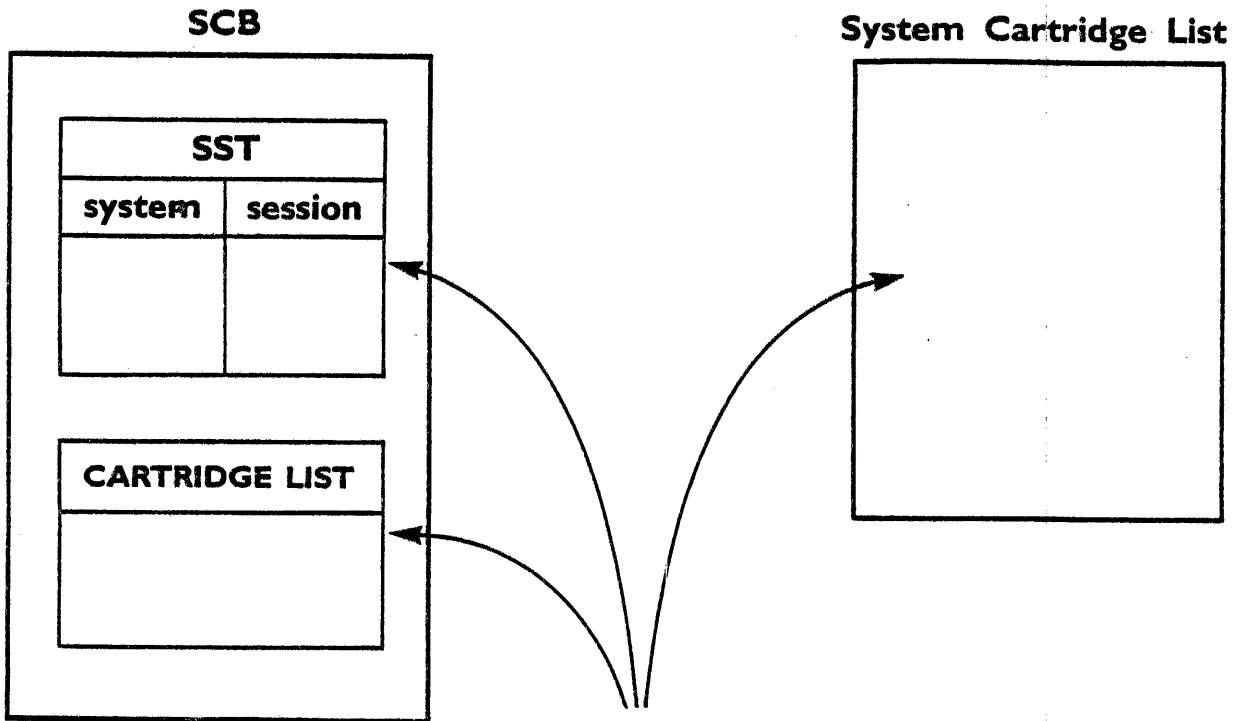
```
:DL,SS
```

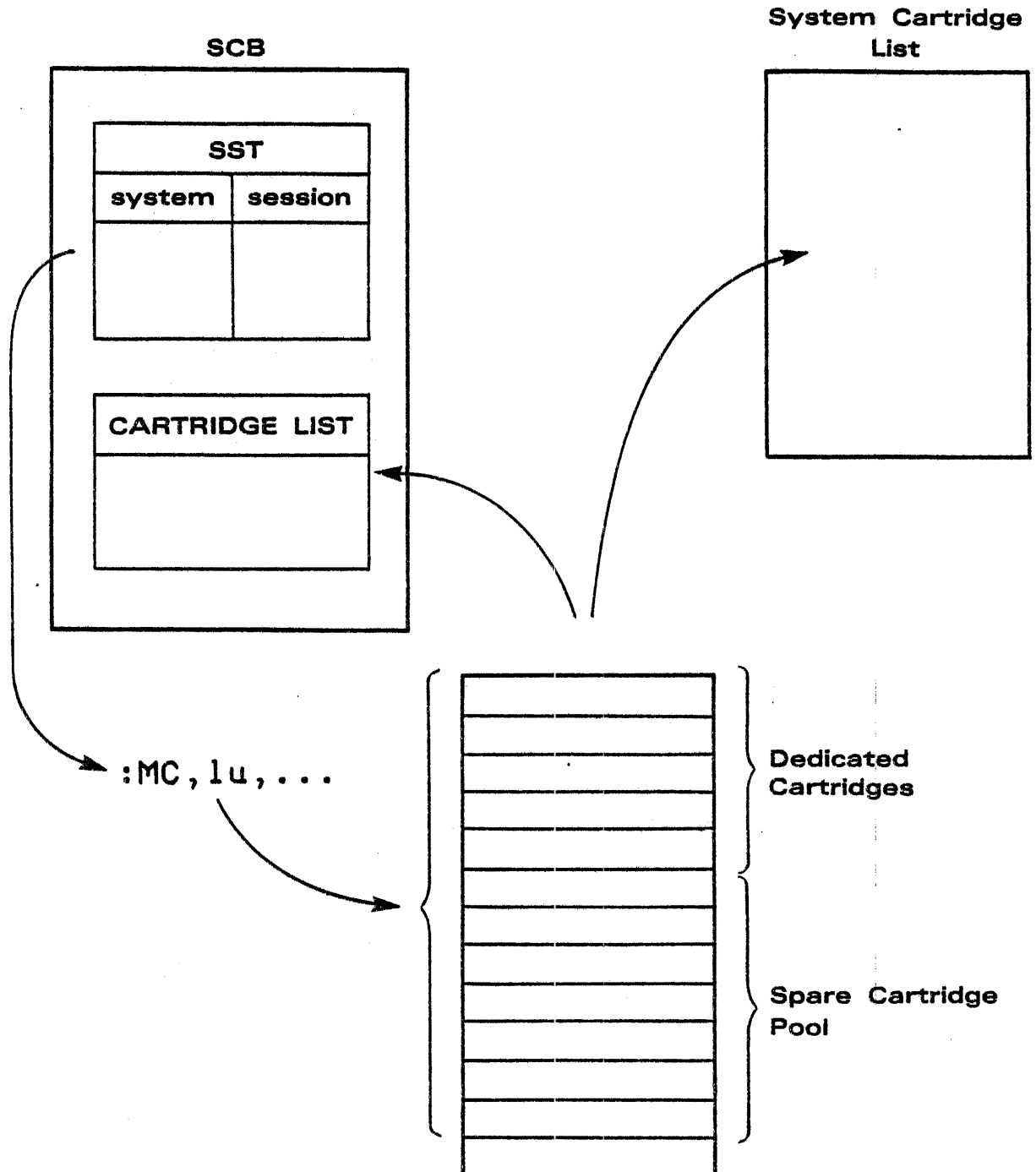
```
CR= SS
```

```
ILAB=DC0032 NXTR= 00000 NXSEC=000 #SEC/TR=096 LAST TR=00140 #DR TR=01
```

```
NAME  TYPE  SIZE/LU  OPEN TO
```

```
:
```





DISMOUNTING CARTRIDGES

:DC

Cartridges may be dismounted in two ways:

- ① Setting a cartridge inactive

:DC, cartridge

↑
positive crn
negative LU

This marks the cartridge as inactive in your SCB Cartridge List.

- the cartridge remains in the System Cartridge List and in your SCB Cartridge List.
- the cartridge is omitted in any file searches

② **Actually *dismounting* a cartridge**

:DC, cartridge, RR

**↑
Release Resources**

This deletes the cartridge from

- **your SCB's cartridge list**
- **the system cartridge list**

and removes the disc LU from your SCB's SST.

✱ PACKING CARTRIDGES ✱

Users can pack their private or group cartridges to reclaim space released by purged files and close any gaps left between files.

:PK, cartridge

↑
**positive CRN,
negative LU**

- **if no cartridge is specified, all of the user's private and group cartridges are packed.**
- **all files on the cartridge must be closed.**

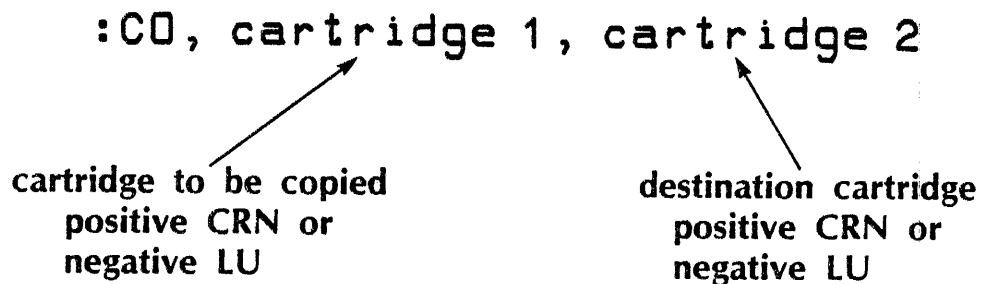
COPYING CARTRIDGES TO OTHER CARTRIDGES

You can copy all files on a mounted cartridge to another mounted cartridge with the FMGR CO command.

`:CO, cartridge 1, cartridge 2`

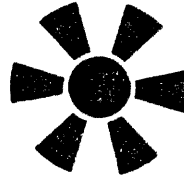
cartridge to be copied
positive CRN or
negative LU

destination cartridge
positive CRN or
negative LU

The diagram shows the command `:CO, cartridge 1, cartridge 2` in a monospaced font. Below the command, there are two blocks of text. The first block, 'cartridge to be copied positive CRN or negative LU', has an arrow pointing from it to the text 'cartridge 1' in the command. The second block, 'destination cartridge positive CRN or negative LU', has an arrow pointing from it to the text 'cartridge 2' in the command.

- Files of the same name on both cartridges are not transferred.
- The original contents of cartridge 2 are not affected.
- Any records >128 words are truncated as they are transferred to cartridge 2.

11B. SAVING/RESTORING CARTRIDGES



Users can *save* the contents of their private or group cartridges on magnetic tape with the WRITT utility.

```
:RU, WRITT [ ,cartridge [ ,mag tape lu ] ]
```

↑
positive crn
negative LU of disc cartridge

- WRITT will rewind the mag tape before and after the save operation.
- if the tape already contains a save of a different CRN, WRITT will ask if you want to overlay the cartridge already on the tape.

RESTORING CARTRIDGES

Users can restore cartridges from magnetic tape back to private or group disc cartridges with the READT utility.

```
:RU,READT [ , cartridge [ , mag tape lu [ ,  $\begin{matrix} P \\ G \end{matrix}$  [ , size ] ] ] ] ]
```

↑
positive crn
negative LU of disc cartridge

- if a CRN is specified and the cartridge is not already mounted, READT will mount a cartridge from the spare cartridge pool and restore it from tape.
- if a disc LU is specified, the specified cartridge is restored.

12

SPOOLING



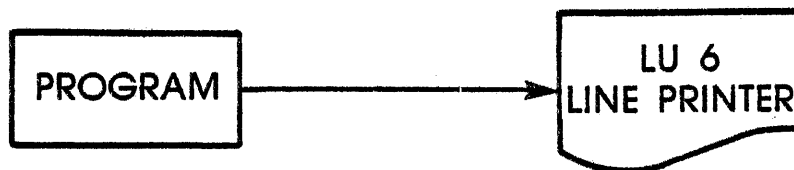
SECTION

A	WHAT IS SPOOLING?	12-3
B	USING SPOOLING INTERACTIVELY	12-6
C	HOW SPOOLING WORKS	12-14
D	MONITORING SPOOLING — GASP	12-18

12A. WHAT IS SPOOLING?



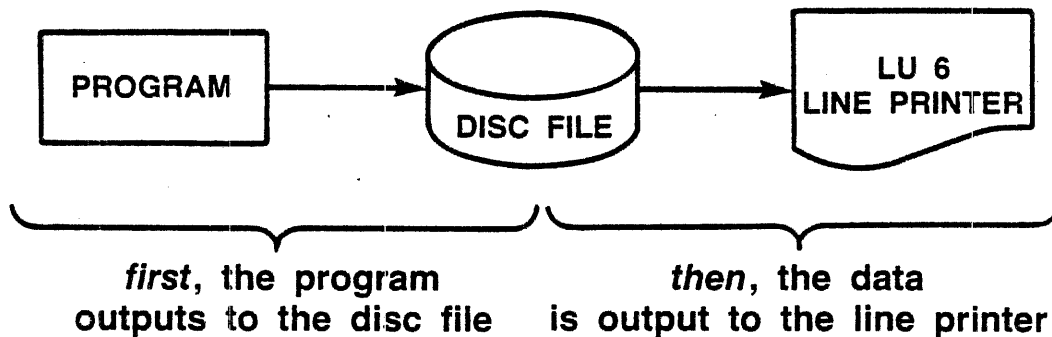
Consider a program outputting data to the line printer —



1. If the program outputs a line of data every few minutes, how do you prevent interleaved listings from other programs?
2. If the program outputs large amounts of data, how do you insure that the program will execute as quickly as possible?

A SOLUTION

Use an intermediate disc file in the output process.



1. The program may execute faster since disc accesses are faster than line printer speeds.
2. The line printer may operate as fast as it can, rather than waiting for output requests to be made by a program.



THE SPOOL SYSTEM

- **allows both INPUT and OUTPUT spooling**
- **must be initialized by the System Manager before you can use spooling. In the initialization process, the System Manager sets up:**

SPOOL FILE POOL — the System Manager defines a group of disc files available for use as spool files. Individual spool files in the pool are called SPOOL POOL FILES.

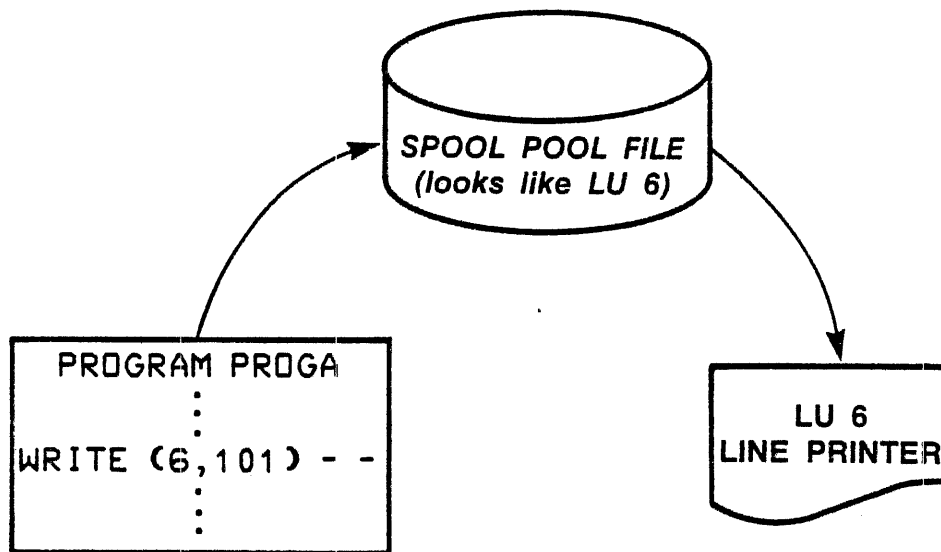
OUTSPOOL LU's — the System Manager identifies which devices (LU's) can be the destination of an output spooling operation.

- **can be used**

**INTERACTIVELY
PROGRAMMATICALLY
FROM BATCH JOBS**

12B. USING SPOOLING INTERACTIVELY

Using SPOOLING involves three simple steps. For example, consider a program outputting to the line printer (LU 6) via a `WRITE (6,101)` statement.



Step 1 Set up a spool file by — allocating a spool pool file
— associating it with LU 6

Step 2 Run the program

Step 3 Close the spool file by — reassociating LU 6 with the line printer
— letting the Spool System dump the spool file to the line printer and then return the spool file to the spool file pool.

NEW FMGR COMMANDS

STEP 1 — another version of the SL command sets up spooling

:SL,6,,WR,6

OUTSPOOL LU — the lu where the output will eventually be dumped

ATTRIBUTE — says the spool file will be used for output

SPOOL FILE — defaulted parameter says use a spool pool file

SPOOL LU — the lu to be associated with the spool file and to be referenced by programs or FMGR commands

STEP 2 — run the program (or use FMGR commands)

STEP 3 — the CS command completes the spooling operation

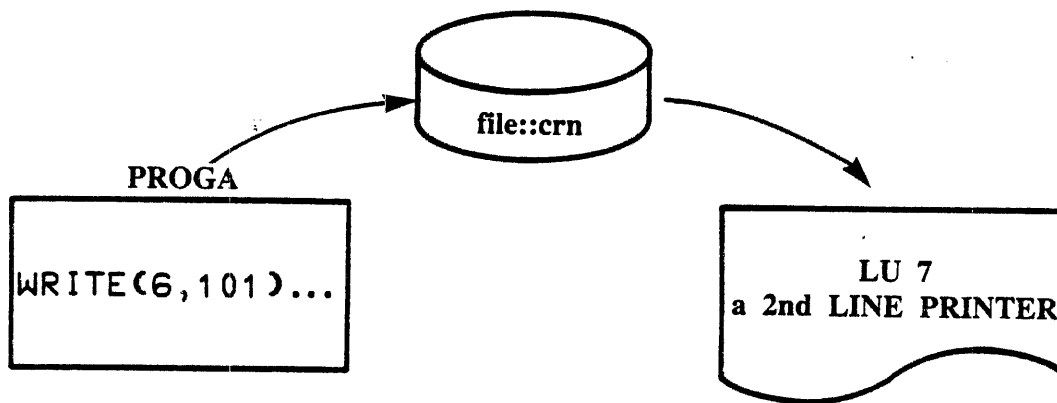
:CS,6

↑
closes the spool file associated with this LU by — reassociating LU 6 with its former device
— queuing the spool file to be dumped to the specified outspool lu.

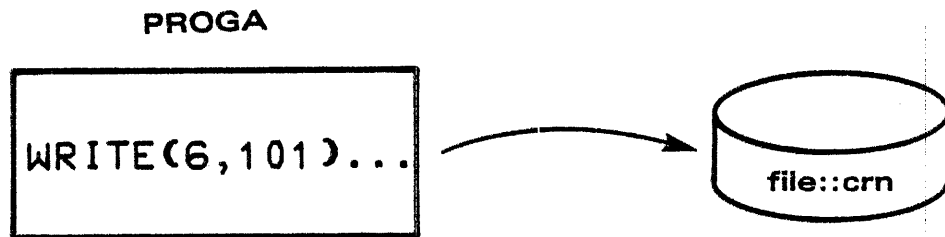
EXAMPLE 2 Spooled output via a user defined spool file.

Instead of using a file from the spool file pool, you can use one of your own disc files as a spool file.

```
:CR,file::crn:3:24  
:SL,6,file::crn,WR,7  
:RU,PROGA  
:CS,6
```

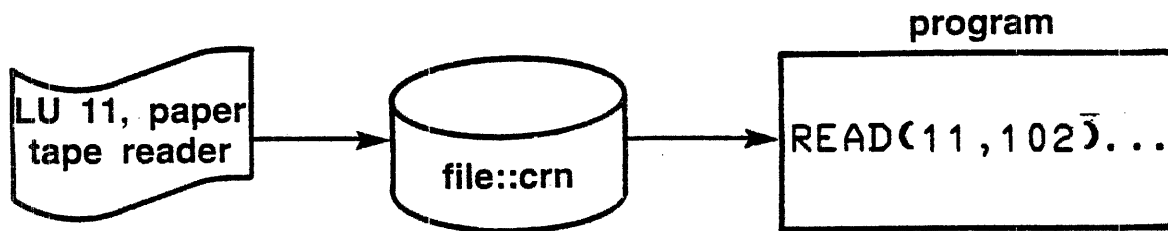


**EXAMPLE 3 Formatted output to disc files using
FORTRAN WRITES or EXEC WRITES
to non-disc LU's**



```
:CR,file::crn:3:24  
:SL,6,file::crn,WRST  
:RU,PROGA  
:CS,6
```

EXAMPLE 4 Input spooling



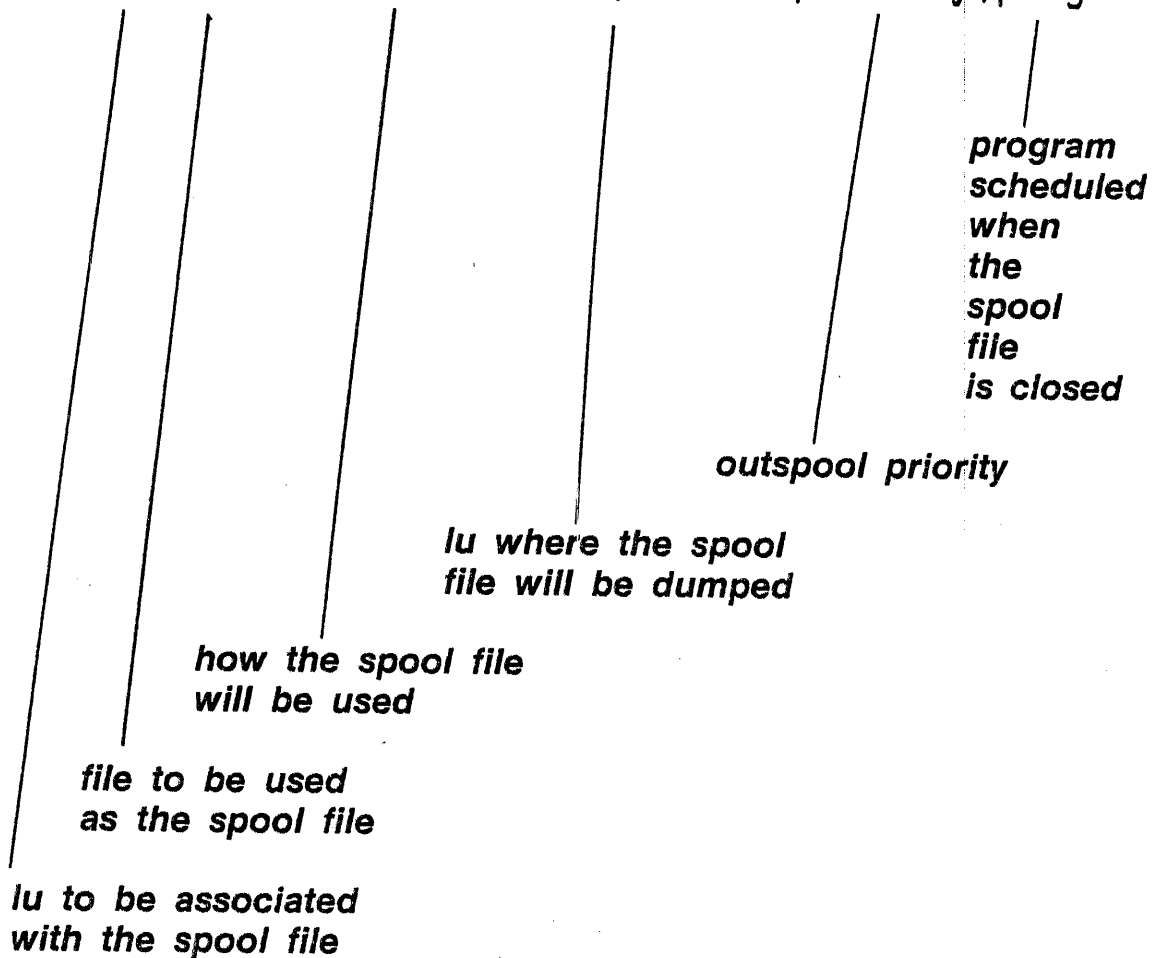
```
:ST,11,file::crn:3:24  
:SL,11,file::crn,RE  
:RU,program  
:CS,11
```

SPOOL FILE SETUP



The SL command associates an LU with a spool file; the spool file can be either a spool pool file or a user defined file.

`:SL,lu,namr,attributes,outspool lu,priority,prog`



CHANGING SPOOL FILE ATTRIBUTES

In addition to closing spool files, the FMGR CS command can be used to change a spool file's attributes.

```
:CS, lu [ ,EN  
          ,RW  
          ,PU  
          ,SA  
          ,PA  
          ,NB  
          ,BU  
          ,NP,outlu,priority ]
```


SPOOLING



- EASY TO USE
- TRANSPARENT TO YOUR PROGRAMS
- ALLOWS INCREASED UTILIZATION OF SYSTEM RESOURCES
- PERMITS I/O TO DISC FILES VIA READS AND WRITES TO NON-DISC LU's



12C. HOW SPOOLING WORKS

The Spool System consists of:

SPOOL FILE POOL — a set of disc files which the Spool System lets you use as spool files

JOBFIL — a disc file containing information about the availability of the spool pool files

SPLCON — a disc file containing information about the spool files currently in use

SPOOL SYSTEM MODULES — programs to manage spooling

SPOOL SYSTEM MODULES

Three of the Spool System Modules are

SMP — assigns and manages spool files.

SPOUT — dumps spool files in the outspool queues to their appropriate outspool LU's. SMP schedules SPOUT as needed.

GASP — allows the System Manager to initialize the Spool System.

— allows users to interactively enter commands to examine and modify the status of their spool files.

SYSTEM SPOOL LU's

When generating RTE, the System Manager defines a set of LU numbers to be used by the Spool System.

Table of System Spool LU's

Spool file	Used?	System Spool LU
—	no	16
—	no	17
—	no	18
—	no	19
SPOL01	yes	20

SPOOLING AND YOUR SST

When you set up a spool file, the Spool System adds an entry to the SST in your Session Control Block.

SCB	
KAREN.PROGDEV	
30	
SST	
system LU	session LU
9	1
2	2
3	3
32	4
33	5
6	6
10	7
.	.
.	.
.	.

```

:SL
SLU 1=LU # 9 = E 9
SLU 2=LU # 2 = E 1
SLU 3=LU # 3 = E 1 S 6
SLU 4=LU # 32 = E 9 S 1
SLU 5=LU # 33 = E 9 S 2
SLU 6=LU # 6 = E 6
SLU 7=LU # 10 = E10
SLU 8=LU # 8 = E 8
SLU 25=LU # 25 = E 1 S16
SLU 28=LU # 28 = E 1 S 1
SLU 38=LU # 38 = E 1 S 2
SLU 47=LU # 47 = E 1 S11
SLU 50=LU # 50 = E 1 S14
:SL,6,,6
:SL
SLU 1=LU # 9 = E 9
SLU 2=LU # 2 = E 1
SLU 3=LU # 3 = E 1 S 6
SLU 4=LU # 32 = E 9 S 1
SLU 5=LU # 33 = E 9 S 2
SLU 6=LU # 19 = E19
SLU 7=LU # 10 = E10
SLU 8=LU # 8 = E 8
SLU 25=LU # 25 = E 1 S16
SLU 28=LU # 28 = E 1 S 1
SLU 38=LU # 38 = E 1 S 2
SLU 47=LU # 47 = E 1 S11
SLU 50=LU # 50 = E 1 S14
:
    
```

12D. MONITORING SPOOLING — GASP

GASP allows you to interactively enter commands to examine and modify the status of your spool files.

You run GASP by

```
:RU,GASP[,lu]  
↑
```

GASP responds with a
prompt for a command

or

```
:RU,GASP,command
```

GASP processes the
command and terminates

Two useful commands are

```
↑??      error explanation  
↑EX      exit GASP
```

DISPLAYING SPOOL STATUS

```
:RU,GASP,DS
  SESLU SYSLU NAME  PRIORITY JOB# STATUS
      6      6 SPOL03    99    --   W
      --    -- IFILE    99    --   W
END GASP
:RU,GASP,DSAL
  SESLU SYSLU NAME  PRIORITY JOB# STATUS USER.GROUP
      6      6 SPOL01    99    --   AH   MANAGER.SYS
      6      6 SPOL02    99    --   W    YIT.HP
      6      6 SPOL03    99    --   W    COREY.HP
      --    -- IFILE    99    --   W    COREY.HP
END GASP
:
```

The status of a spool file is represented by 1 of 4 states.

- W WAITING** — the spool file has been setup but not closed.
- the spool file has been closed and is in an outspool queue.
- A ACTIVE** — the spool file is being dumped to an outspool LU.
- AH ACTIVE HOLD** — the spool file was being dumped but was held by a GASP operator request or a down device.
- H HOLD** — the spool file was in state W but was held by a GASP operator request.

MODIFYING SPOOL STATUS

- CHANGE SPOOL FILE STATUS

↑CS,spool file,H
 outspool priority
 R

- RESTART AN ACTIVE SPOOL FILE

↑RS,spool file[,outspool lu]

- KILLING SPOOL FILES

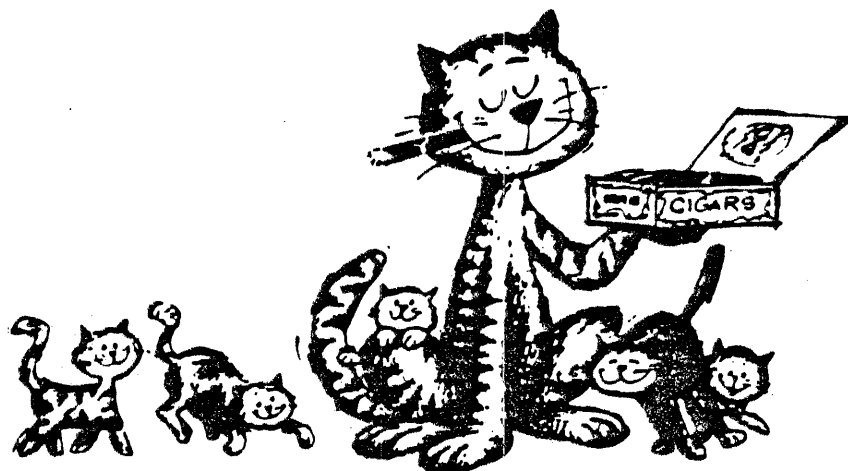
↑KS,spool file

- UP AN OUTSPOOL DEVICE

↑UP[,RS]

13

BATCH PROCESSING OF JOBS



Section

A	FMGR AND BATCH JOBS	13-3
B	USING BATCH PROCESSING	13-6
C	MONITORING JOBS – GASP	13-14

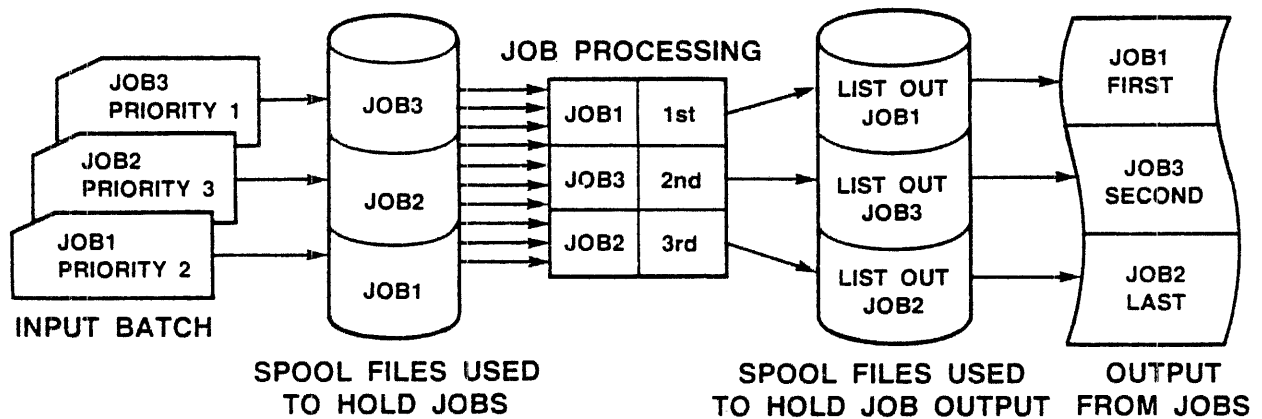
13A. FMGR AND BATCH JOBS

FMGR performs three main functions:

1. Interfacing users to RTE
2. Allowing interactive file manipulation
3. Processing jobs in batch mode

Only FMGR (not FMGxx) can process jobs!

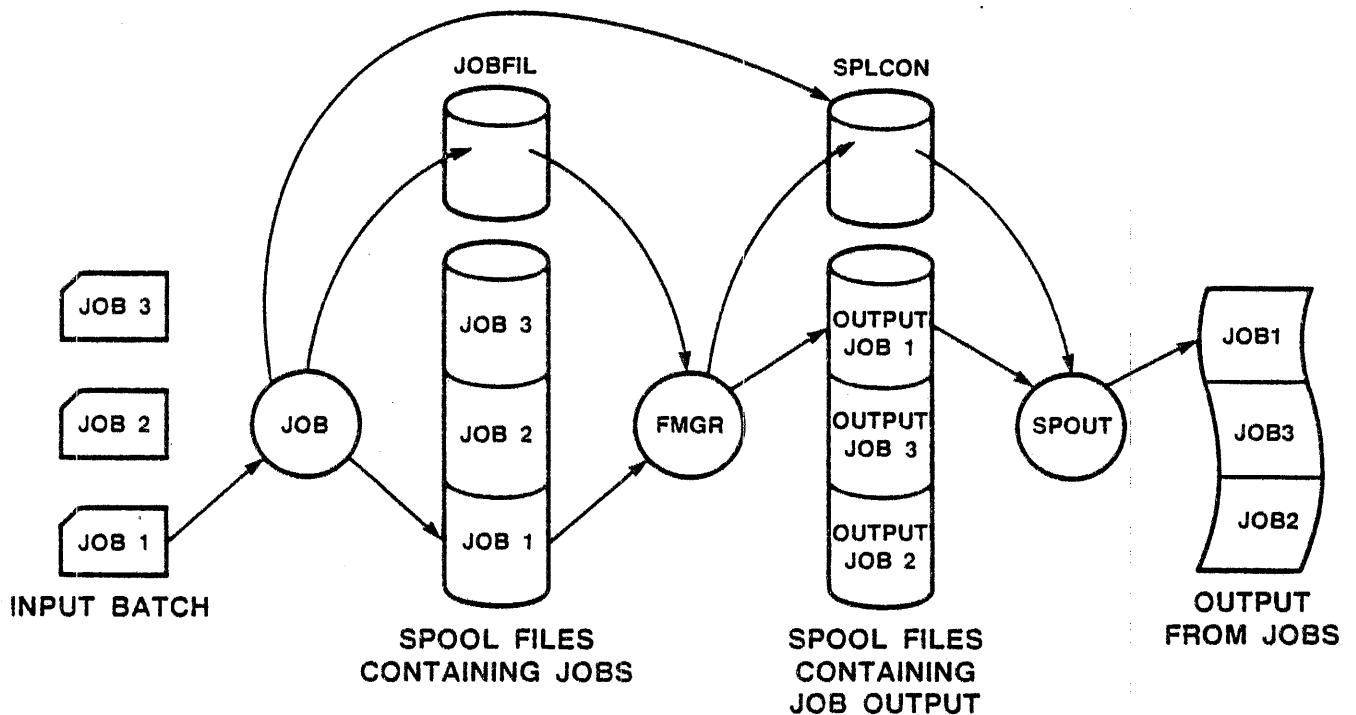
BATCH PROCESSING USES SPOOLING



- Jobs execute on a priority basis, except for the first job which goes into execution as soon as it is entered.
- Job output is printed on a priority basis, except for the first job, whose output is printed as soon as the job ends.

HOW JOBS ARE PROCESSED

You run program **JOB** to submit a batch job. Program **JOB** will initiate batch processing, scheduling the necessary programs automatically.



13B. USING BATCH PROCESSING

3 EASY STEPS

- **Define the job**

2 new FMGR commands

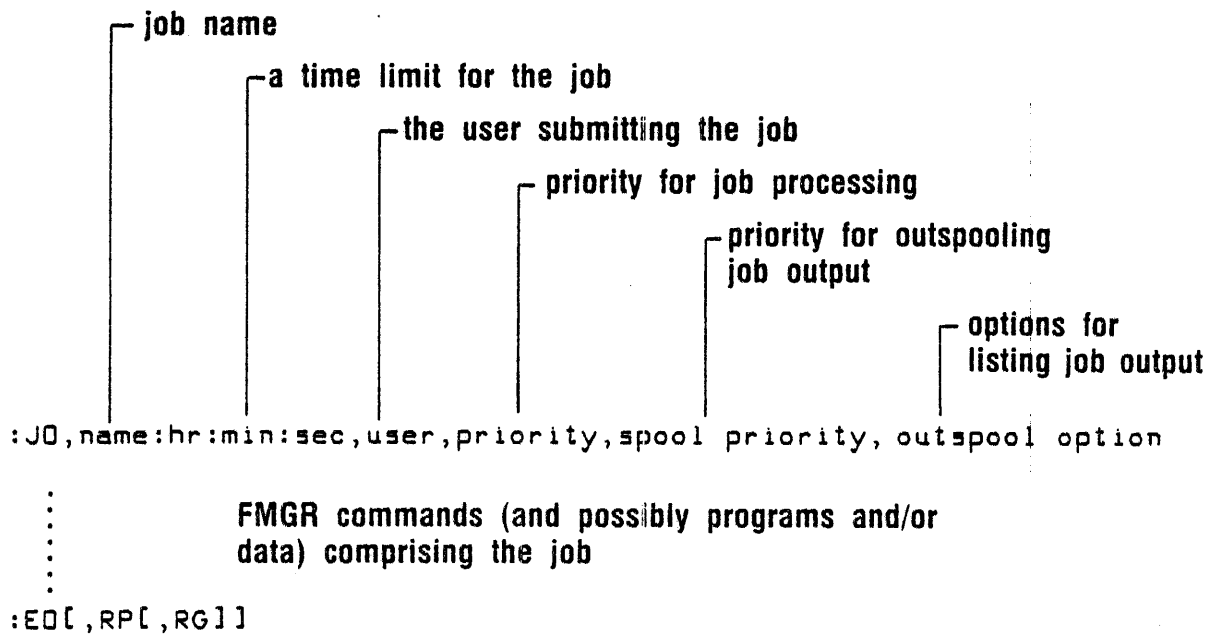
- **Submit the job**

Program "JOB"

- **Get the answers!**

DEFINING JOBS

The format of any batch job is:



For example,

```
:JQ,JOBS:0:5:30,VIC.HP/%VO
:RU,FTN4,&PROGF,6,-
:RU,LOADR,,%PROGF,6
:RU,10G
:EO
```

A JOB'S SESSION

When a job executes, it runs in a session, just as if the user specified in the JO command had logged on at a terminal.

If no user was specified in the JO command, the user who submitted the job is used.

- The job has access to the same LU's as does that user, with three exceptions:
 - LU 1 is the system console
 - LU 5 is the job's inspool file (the job itself)
 - LU 6 is the job's outspool file
- The job has access to the same disc cartridges as does that user.
- FMGR's devices are initially set as follows:
 - input device LU 5
 - list device LU 6
 - log device LU 1
- FMGR's severity code is initially set to 0.

When a job terminates, it ends its session, just as if the user had logged off at a terminal. Private and group cartridges may be saved (default) or released when the job ends.

SUBMITTING JOBS

Jobs are submitted by running program JOB.

```
:RU, JOB, namr [ , priority ]
```

If namr is

Ⓢ a file, program JOB inputs a single job from the file.

```
:RU, JOB, JOBE
```

Ⓢ a non-interactive lu, program JOB inputs one or more jobs from the device specified.

```
:RU, JOB, 8
```

Ⓢ an interactive lu, program JOB prompts for a command.

```
:RU, JOB, 1  
; :XE, JOBF  
; :XE, JOBG  
; (control D)  
:
```

The :XE
command
inputs a single
job stored in the
specified file.

A JOB INPUT PROBLEM

Assume the card reader (LU 5) contains a FORTRAN source program. Will this job compile, load, and run the FORTRAN program?

```
: JO, JCLR, KAREN. PROGDEV  
: RU, FTN4, 5, 6, %TEMP  
: RU, LOADR, , %TEMP, 6  
: RU, 10G  
: EO
```

SOLUTION 1

In a job, LU 5 references the spool file containing the job itself; you could put the FORTRAN program in the "job deck".

```
:JO,JCLR,KAREN.PROGDEV
:RU,FTN4,5,6,%TEMP
FTN4,L
PROGRAM APROG

——
——
——
——
——
——

END
END$
:RU,LOADR,,%TEMP,6
:RU,10G
:EO
```

SOLUTION 2

Suppose the card reader is session LU 5 but system LU 15. Use the SL command to define a new session LU pointing to the same system LU.

```
: JO , JCLR , KAREN . PROGDEV  
: SL , 11 , 15  
: RU , FTN4 , 11 , 6 , %TEMP  
: RU , LOADR , , %TEMP , 6  
: RU , 10G  
: EO
```

MORE COMMANDS TO USE WITH JOBS

- **Setting a program execution time limit**

:TL:hr:min:sec

- **Aborting a job**

:AB

13C. MONITORING JOBS — GASP

GASP lets you enter commands to examine and modify the status of your jobs.

DISPLAYING JOB STATUS

:RU,GASP,DJ

JO#	NAME	STATUS	SPOOLS
-----	------	--------	--------

1	JOB1	S 99 A	1
---	------	--------	---

END GASP

:RU,GASP,DJAL

JO#	NAME	STATUS	USER.GROUP	SPOOLS
-----	------	--------	------------	--------

1	JOB1	S 99 A	COREY.HP	1
2	JOB2	S 99 R	YIT.HP	
3	JOB3	S 99 R	DENNIS.HP	

END GASP

:

A job's status is represented by 1 of 5 states.

- I** the job is in the process of being **INSPOOLED**
- IH** the job was being **INSPOOLED** but was **HELD** by a **GASP** operator request
- IA** the job was being **INSPOOLED** but was **ABORTED** by a **GASP** operator request
- R** the job is **READY** to be processed
- RH** the job was **READY** but was **HELD** by a **GASP** operator request
- A** the job is being processed by **FMGR (ACTIVE)**
- CS** the job has **COMPLETED** and is ready for **OUTSPOOLING**

MODIFYING JOB STATUS

- Changing a job's status

↑ CJ, job number, job priority
H
R

- Removing pending jobs

↑ AB, job number

ABORTING THE CURRENTLY EXECUTING BATCH JOB

The system AB command

- aborts the currently executing batch job
- must be entered from the system console

*AB $\left[\begin{array}{c} 0 \\ , 1 \end{array} \right]$

SESSION MODE

The System Console is enabled as a “session terminal” by using the following command.

```
*EN, master security code[,option]
```

The System Console then operates like any other terminal.

```
PLEASE LOG ON : KAREN.PROGDEV/%KRP
-
-
-
-
-
-
:
```

The System Console remains enabled as a session terminal until

- the system is re-booted
- the System Console is disabled via the program ACCTS

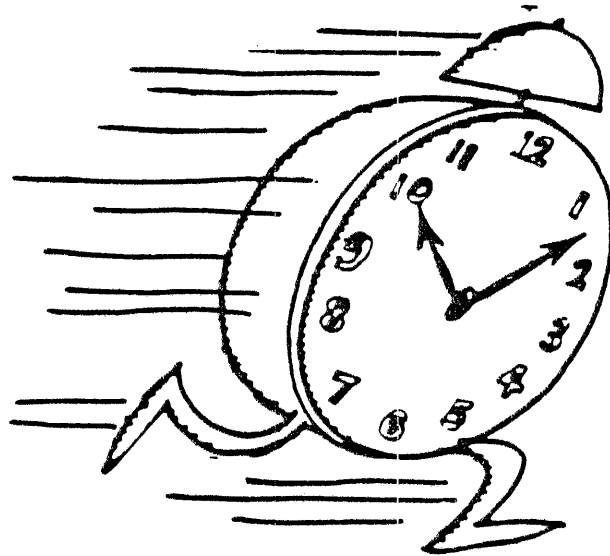
SYSTEM LEVEL OPERATOR COMMANDS

The system OP command lets the operator enter a single system command without having the command subject to command capability checking.

```
S = 01 COMMAND?OP, master security code, command
```

(Note: the command is still processed in session; only command capability checking is bypassed)

15 TIME-SCHEDULED PROGRAMS



TIME-SCHEDULED PROGRAMS

RTE, the REAL-TIME EXECUTIVE, allows you to schedule programs to execute at specific times during the day.

- RTE keeps a list (the TIME LIST) of those programs which are time-scheduled.
- RTE keeps a REAL-TIME SYSTEM CLOCK which is used to time-schedule programs.
 - * the system TI command displays the current system time

```
S = xx COMMAND?TI
1978 345 13 17 43
```

- * the system TM command lets the operator set the System Clock

```
S = xx COMMAND?TM,1979,150,11,45,30
```

TIME BASE GENERATOR

RTE uses the TIME BASE GENERATOR (TBG) to update its System Clock. The TBG interrupts the CPU every 10 milliseconds. While processing the interrupt, RTE

- *updates the time in its System Clock*
- *checks the time-list to see if any programs need to be scheduled*



TYPES OF TIME-SCHEDULING

RTE allows you to schedule programs to execute:

- ◆ at an absolute time
 - ◆ at a specified offset from the present
- } “starting times”
-
- ◆ only once
 - ◆ repeatedly, at specific intervals
- } “interval time”

A program’s “starting time” and “interval time” are referred to as its time parameters.

WAYS TO TIME-SCHEDULE A PROGRAM

INTERACTIVELY

- **define the program's time parameters in its PROGRAM (or NAM) statement**
issue the system ON command to put the program in the time list
- **use the system IT command to set the program's time parameters**
issue the system ON command

PROGRAMMATICALLY

- **have the program make an EXEC 12 call to define a program's (or its own) time parameters and place that program (or itself) in the time list**

DEFINING TIME PARAMETERS

- PROGRAM (or NAM) statement

PROGRAM name (type,priority, res,mpt, hr,min,sec,ms)

defines interval time defines starting time

- System IT command

:SYIT,program, res,mpt, hr,min,sec,ms

defines interval time defines starting time

For interval time {

- res — 1 10's of milliseconds
- 2 seconds
- 3 minutes
- 4 hours

mpt — number of units (in "res") to wait between repeated time scheduling

For starting time {

- hr — hour
- min — minutes
- sec — seconds
- ms — 10's of milliseconds

TIME PARAMETER EXAMPLES

:SYIT,APROG,3,5,12,0,0,0

:SYIT,BPROG,1,10,3,15,45,10

:SYIT,CPROG,1,0,18,0,0,0



PLACING THE PROGRAM IN THE TIME LIST

The system ON command places the specified program in the time list, using previously defined time parameters.

:SYDN,program,NOW,p1,p2,p3,p4,p5

optional parameters to be passed to the program

if specified, ignore starting time and schedule the program immediately

if omitted, place the program in the time list according to its time parameters

For example,

```
:SYIT, YOURP, 4, 1, 17, 0, 0, 0  
:SYDN, YOURP
```

```
:SYIT, MYPRG, 2, 100, 18, 0, 0, 0  
:SYDN, MYPRG, NOW
```

PROGRAMMATIC TIME SCHEDULING

The EXEC 12 call allows a program to put a specified program or itself into the time list. The program may be scheduled to initially execute:

at a specified offset from the present time

CALL EXEC (12, INAME, IRESL, IMULT, IOFST)

resolution code,
defining units of time

execution multiple,
defining interval time

offset starting time,
a negative value

at an absolute time

CALL EXEC (12, INAME, IRESL, IMULT, IHRS, IMIN, ISEC, IMSEC)

resolution code

execution multiple

absolute starting time

Examples

1. IRESL = 4
IMULT = 1
IHRS = 15
IMIN = 45
ISEC = 0
IMSEC = 0
CALL EXEC (12, INAME, IRESL, IMULT, IHRS, IMIN, ISEC, IMSEC)

2. IRESL = 2
IMULT = 0
IOFST = -600
CALL EXEC (12, INAME, IRESL, IMULT, IOFST)

3. IRESL = 3
IMULT = 15
IOFST = -30
CALL EXEC (12, INAME, IRESL, IMULT, IOFST)

4. IRESL = 3
IMULT = 0
IHRS = -5
IMIN = 0
ISEC = 0
IMSEC = 0
CALL EXEC (12, INAME, IRESL, IMULT, IHRS, IMIN, ISEC, IMSEC)



A QUESTION

When you run a program
(:RU, program), FMGR waits for the
program to complete and then issues
another prompt.

When you time schedule a program
(:SYDN, program), FMGR requests RTE to
place the program in the time list and
immediately issues another prompt.

Can you log-off before the time-scheduled
program executes?

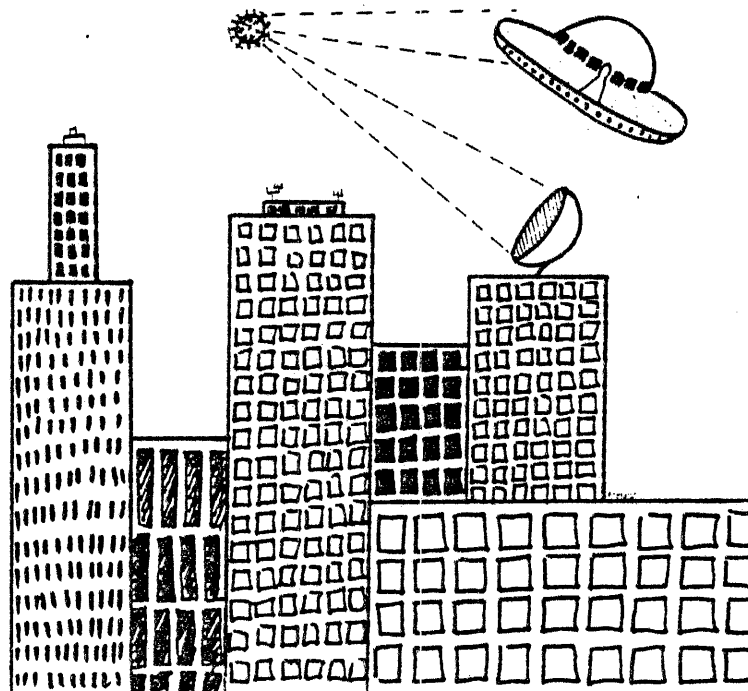


16

PROGRAMS

SCHEDULING OTHER

PROGRAMS



SECTION

A	EXEC SCHEDULING CALLS	16-3
B	PASSING AND RETURNING INFORMATION	16-7
C	PROGRAM TERMINATION	16-12

16A. EXEC SCHEDULING CALLS

The EXEC 9, 10, 23 and 24 calls allow a program to schedule another program.

A program that schedules another program is called a FATHER; a program that is scheduled by another program is called a SON.

To schedule a Son use:

```
CALL EXEC (ICCODE, INAME)
```

|
array containing the
name of the program to
be scheduled

|
type of scheduling:

EXEC 9 — Immediate Schedule, wait for
completion

EXEC 10 — Immediate Schedule, no wait

EXEC 23 — Queue Schedule, wait for completion

EXEC 24 — Queue Schedule, no wait

IMMEDIATE SCHEDULE vs. QUEUE SCHEDULE



Immediate Schedule — The Son program should be scheduled immediately.

- **If the Son is dormant, it is immediately scheduled and runs at its own priority.**
- **If the Son is not dormant it is not scheduled and its state is returned in the A-register.**



Queue Schedule — The Son program should be scheduled whenever it is dormant.

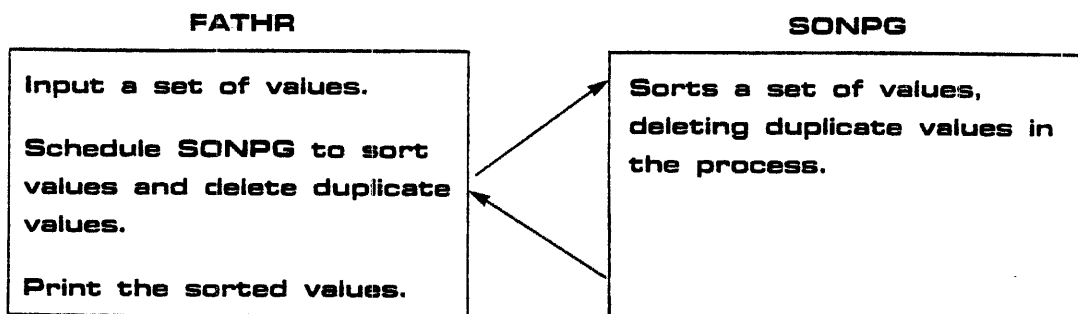
- **If the Son is dormant, it is immediately scheduled and runs at its own priority.**
- **If the Son is not dormant, the schedule request is placed in a queue but the Son's state is not returned in the A-register.**

WAIT SCHEDULE vs. NO WAIT SCHEDULE

Schedule with wait implies that the Father waits for the Son to complete before resuming execution.

Schedule without wait implies that the Father does not wait for the Son to complete; rather, the Father and the Son will compete for execution time on a priority basis.

AN EXAMPLE TO PROGRAM



16B. PASSING AND RETURNING INFORMATION

Consider running a program from FMGR —

- you can pass information to the program

```
:RU,PROGA,6,AF,IL,E
```

PROGA can then retrieve the information.

- the program can also return information to FMGR

```
PROGRAM PROGA  
INTEGER PARM(5)  
.  
.  
.  
.  
CALL PRTN (PARM)  
END
```

PROGA returns the values in PARM to FMGR.

FATHER TO SON COMMUNICATION

A Father can send information to a Son via the EXEC scheduling call.

IOP1 to IOP5 are optional parameters, whose values are passed to the Son. The Son uses RMPAR to retrieve the values.

CALL EXEC (ICODE, INAME, IOP1, IOP2, IOP3, IOP4, IOP5, Ibuff, ILEN)



Ibuff is an array of data to be passed to the Son. ILEN specifies the positive number of words or negative number of characters to be transferred.

The Son uses an EXEC 14 call to retrieve the data.

SON TO FATHER COMMUNICATION

* * * * *

If the Father scheduled the Son with wait, the Son can return information to the Father.

Via PRTN and RMPAR

- the Son calls PRTN to pass up to 5 values back to the "waiting" Father.
- the "waiting" Father retrieves the values with a call to RMPAR.

Via EXEC 14

- the Son uses an EXEC 14 call to pass a buffer of data back to the "waiting" Father.
- the "waiting" Father retrieves the buffer with another EXEC 14 call.

THE FATHER

GFATHR T=00004 IS ON CR01500 USING 00005 BLKS R=0000

```
0001  PTN4,L
0002      PROGRAM FATHR
0003      INTEGER NSON(3), MTDATA(100), PARM(5)
0004      DATA NSON/2HSO,2HNP,2HG /
0005  C
0006  C      INPUT DATA VALUES FROM THE MAG TAPE (LU 8),
0007  C      FIRST VALUE IS A HEADER VALUE.
0008  C
0009      READ(8,*) IVALS, (MTDATA(I),I=1,IVALS)
0010  C
0011  C      SCHEDULE THE SON WITH WAIT, PASSING THE NUMBER
0012  C      OF DATA VALUES AND THE ARRAY CONTAINING THE VALUES..
0013  C
0014      ICODE = 23
0015      CALL EXEC(ICODE,NSON,IVALS,J,K,L,M,MTDATA,IVALS)
0016  C
0017  C      LET THE SON SORT THE VALUES AND DELETE DUPLICATE VALUES.
0018  C
0019  C      WHEN THE SON COMPLETES, RETRIEVE THE NEW NUMBER OF VALUES
0020  C
0021      CALL RMPAR(PARM)
0022      IVALS = PARM(1)
0023  C
0024  C      AND RETRIEVE THE SORTED DATA
0025  C
0026      CALL EXEC(14,1,MTDATA, IVALS)
0027      CALL ABREG(IA,IB)
0028      IF(IA .EQ. 0) GOTO 75
0029      WRITE(1,102)
0030 102      FORMAT(" NO SORTED DATA RECEIVED FROM SON".)
0031      STOP
0032  C
0033  C      PRINT THE RESULTS
0034  C
0035 75      WRITE(1,103) (MTDATA(I),I=1,IVALS)
0036 103      FORMAT(10(10(I5,1X)/))
0037  C
0038      END
```


AND THE SON

&SONPG T=00004 IS ON CR01500 USING 00004 BLKS R=0000

```
0001  FTN4,L
0002      PROGRAM SONPG
0003      INTEGER VALUES(100), PARM(5)
0004  C
0005  C      RETRIEVE THE NUMBER OF VALUES TO BE SORTED
0006  C
0007      CALL RMPAR(PARM)
0008      NVALS = PARM(1)
0009  C
0010  C      AND THE ARRAY OF DATA VALUES.
0011  C-
0012      CALL EXEC(14,1,VALUES,NVALS)
0013      CALL ABREG(IA,IB)
0014      IF(IA .EQ. 0) GOTO 50
0015      WRITE(1,101)
0016  101      FORMAT("NO DATA BUFFER FROM FATHER FOUND".)
0017      STOP
0018  C
0019  C      SORT THE VALUES, DELETING DUPLICATE VALUES IF ANY
0020  C
0021  50      CONTINUE
0022  C
0023  C          (VALUES WILL THEN CONTAIN THE SORTED DATA,
0024  C          NVALS WILL CONTAIN THE NEW NUMBER OF VALUES.)
0025  C
0026  C
0027  C      RETURN THE SORTED VALUES TO THE FATHER
0028  C
0029      CALL EXEC(14,2,VALUES,NVALS)
0030      CALL ABREG(IA,IB)
0031      IF(IA .EQ. 0) GOTO 60
0032      WRITE(1,102)
0033  102      FORMAT("NO FATHER FOUND TO ACCEPT RESULTS".)
0034      STOP
0035  C
0036  C      AND THE NEW NUMBER OF VALUES
0037  C
0038  60      CONTINUE
0039      PARM(1) = NVALS
0040      CALL PRTN(PARM)
0041  C
0042      END
```

16C. PROGRAM TERMINATION

A Father can use an EXEC 6 call to terminate itself or a Son.

```
CALL EXEC (6, INAME, ICMCD, IOP1, IOP2, IOP3, IOP4, IOP5)
```

0 — terminate self
or an array containing
the name of a Son

if a program is terminating
itself, up to 5 parameters
may be stored in its ID segment

Completion code, specifies the type of termination

- 0 — normal termination
- 1 — terminate serially reusable
- 1 — terminate saving resources and point of suspension
- 2 — terminate immediately after current I/O operation completes, remove from time list (same as :SYOF,program,0)
- 3 — terminate immediately (clear any I/O), remove from time list (same as :SYOF,program,1).

EXAMPLE

Program COUNT is used to print a message each time program UPDAT is run.

```
0001 FTN4,L
0002     PROGRAM UPDAT
0003 C
0004     INTEGER NAME(3)
0005     DATA NAME/2HCO,2HUN,2HT /
0006 C
0007 C     SCHEDULE COUNT TO PRINT MESSAGE THAT
0008 C     WE ARE EXECUTING AGAIN.
0009 C
0010     CALL EXEC(24,NAME)
0011 C
0012     END
0013
```

```
0001 FTN4,L
0002     PROGRAM COUNT
0003 C
0004     INTEGER ICNT(5)
0005     DATA ICNT/0/
0006 C
0007 C     RETRIEVE PREVIOUS COUNT STORED IN MY ID SEGMENT
0008 C
0009     CALL RMPAR(ICNT)
0010 C
0011 C     INCREMENT COUNT AND PRINT MESSAGE
0012 C
0013     ICNT(1) = ICNT(1) + 1
0014     WRITE(1,101) ICNT(1)
0015 101  FORMAT(/"PROGRAM UPDAT WAS RUN AGAIN, NUMBER ",I5)
0016 C
0017 C     TERMINATE, STORING CURRENT COUNT IN MY ID SEGMENT
0018 C
0019     CALL EXEC(6,0,0,ICNT)
0020     END
0021
```

TERMINATE SERIALY REUSABLE

When a program executes an EXEC 6 with a "terminate serially reusable" completion code, RTE marks the program as being serially reusable.

When the terminated program is scheduled again, RTE checks to see if it is still resident in memory.

If so, the program in memory is executed.

If not, the program is loaded from disc and then executed.

Program resumes →
at main entry point

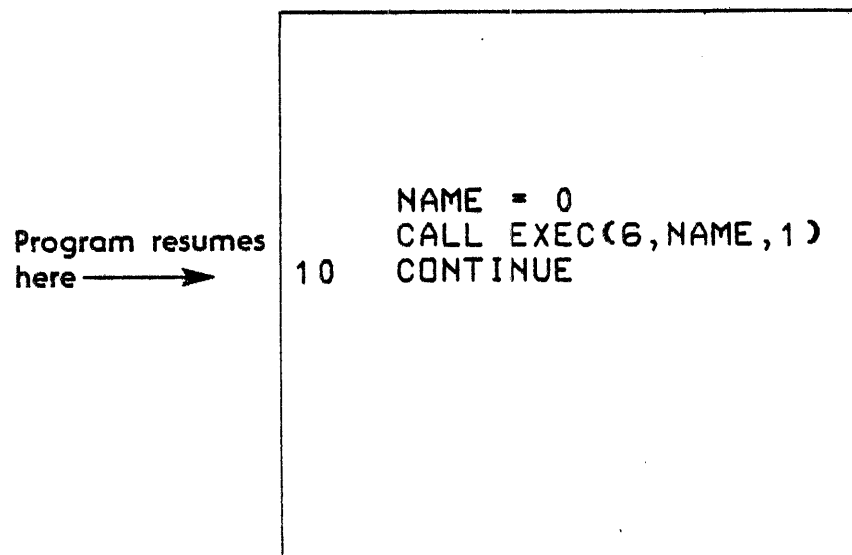
```
NAME = 0  
CALL EXEC(6,NAME,-1)  
10 CONTINUE
```

TERMINATE SAVING RESOURCES

When a program executes an EXEC 6 with a "terminate savings resources" completion code, RTE will not deallocate any resources allocated to the program being terminated.

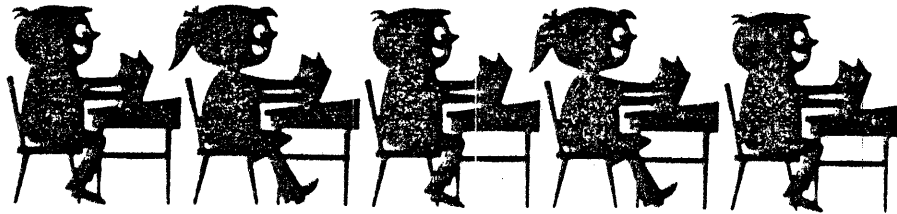
When the terminated program is rescheduled, it will resume execution at its "point of suspension."

If a program terminates itself saving resources, any run stimulus will reschedule it; if a Father terminates a Son saving resources, then only the Father or the RU or ON commands can reschedule it.



17

CLASS I/O



SECTION

A	PROGRAM TO PROGRAM COMMUNICATION	17-3
B	CLASS I/O FOR PROGRAM TO PROGRAM COMMUNICATION	17-5
C	CLASS I/O — A SUMMARY OF FEATURES	17-21
D	CLASS I/O FOR DEVICE I/O AND CONTROL	17-23
E	VARIATIONS WITH CLASS I/O	17-35
F	TERMINAL HANDLERS	17-39

17A. PROGRAM TO PROGRAM COMMUNICATION

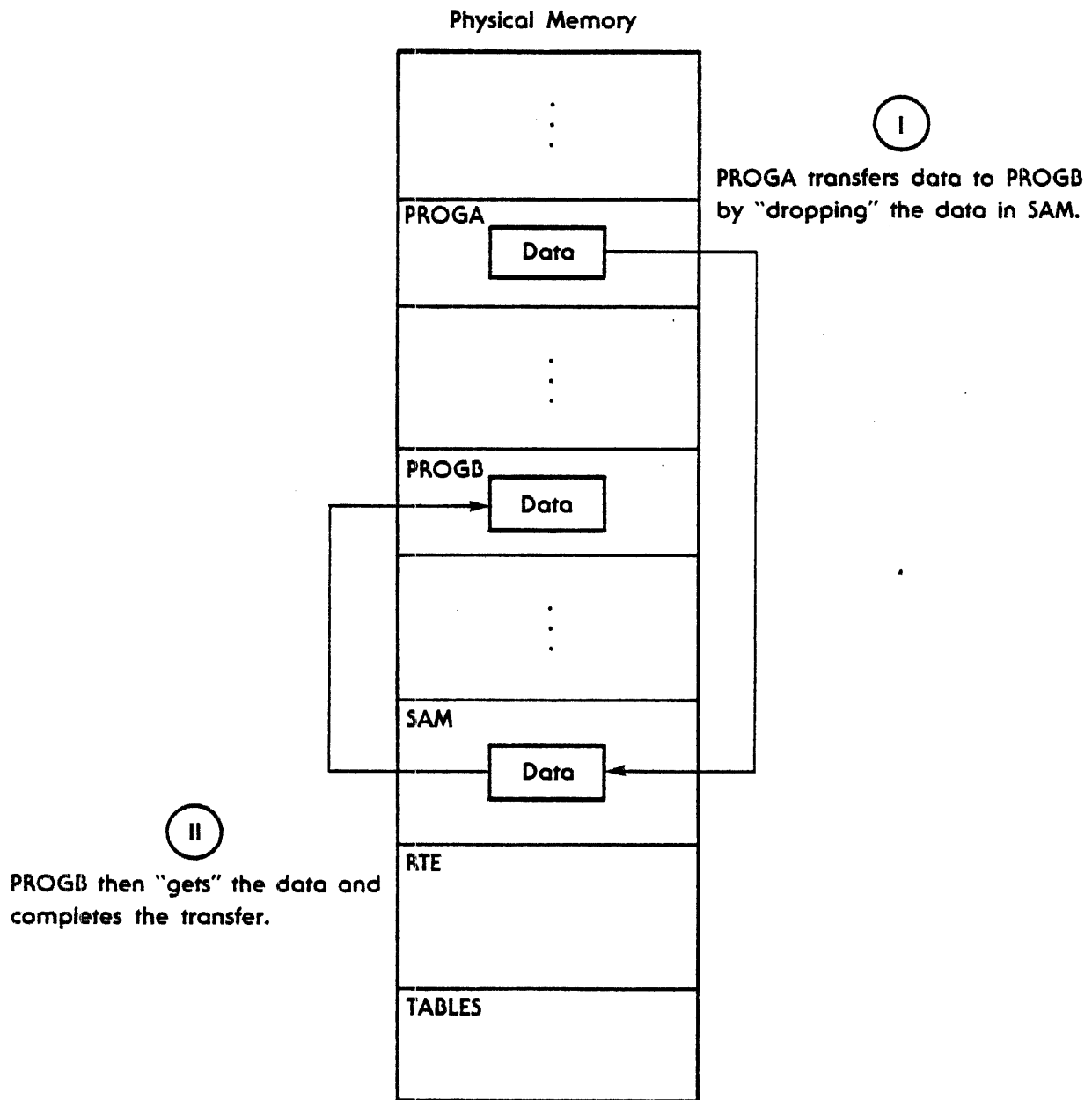
- RMPAR and PRTN
- EXEC 14
- SYSTEM COMMON

17B. CLASS I/O FOR PROGRAM TO PROGRAM COMMUNICATION

CLASS I/O (or MAILBOX I/O) is another means of letting programs talk to each other. With CLASS I/O –

- the size of the data blocks passed between programs is limited only by the size of SAM.
- any number of programs can communicate with each other and share data.
- a program can send many data buffers before another program accepts one.
- if a program asks for a data buffer before one is sent, RTE can put that program “to sleep” until a buffer is “sent” to the program.
- only programs knowing a special “key” can access data being passed.

DATA TRANSFERS BY CLASS I/O ARE DONE THROUGH SAM





MANUFACTURERS and CONSUMERS

EXEC 20 (CLASS WRITE/READ)

A program initiates a CLASS I/O program to program data transfer by making an EXEC 20 call. The CLASS WRITE/READ call will "manufacture" a buffer in SAM and fill it with data from the calling program.

EXEC 21 (CLASS GET)

The receiving program retrieves the data in SAM by making an EXEC 21 call. The CLASS GET will "consume" the buffer in SAM by storing the data in the program and releasing the SAM buffer for use by other programs.

*Every call that "manufactures" a buffer in SAM must have a corresponding call that "consumes" the buffer.

CLASS I/O is said to be "double call."

*Programs can execute independently of the data transfers, which are handled by RTE.



WHO GETS WHICH BUFFER? THE CLASS NUMBER IS THE KEY!

When generating an RTE system, the System Manager specifies how many *class numbers* are to be in the system.

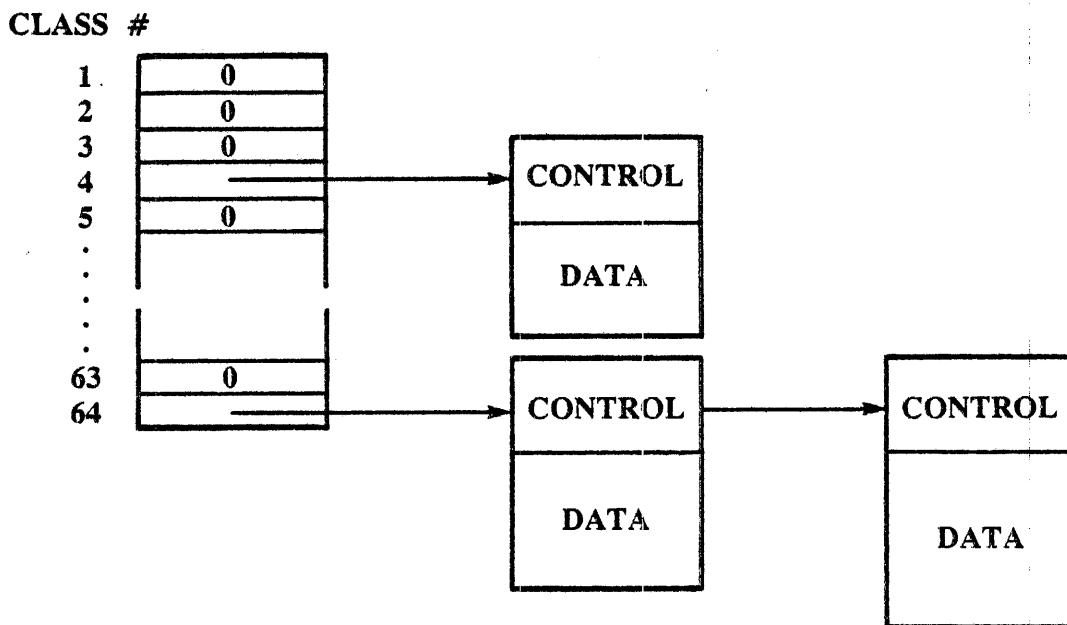
Class numbers are used to protect the CLASS I/O data buffers in SAM —

- when a program uses CLASS I/O for program to program communication, it must
 - request a class number from RTE
 - make the CLASS WRITE/READ request, specifying the class number
- the program retrieving the data makes a CLASS GET call, specifying the appropriate class number



COMPLETED CLASS QUEUE

RTE keeps lists of the data buffers in SAM which were manufactured by a CLASS WRITE/READ and are waiting to be consumed by a CLASS GET. These lists are referred to as COMPLETED CLASS QUEUES.



The buffer in SAM manufactured by a CLASS WRITE/READ is called a CLASS BUFFER. There are two parts to each class buffer:

CONTROL INFORMATION includes

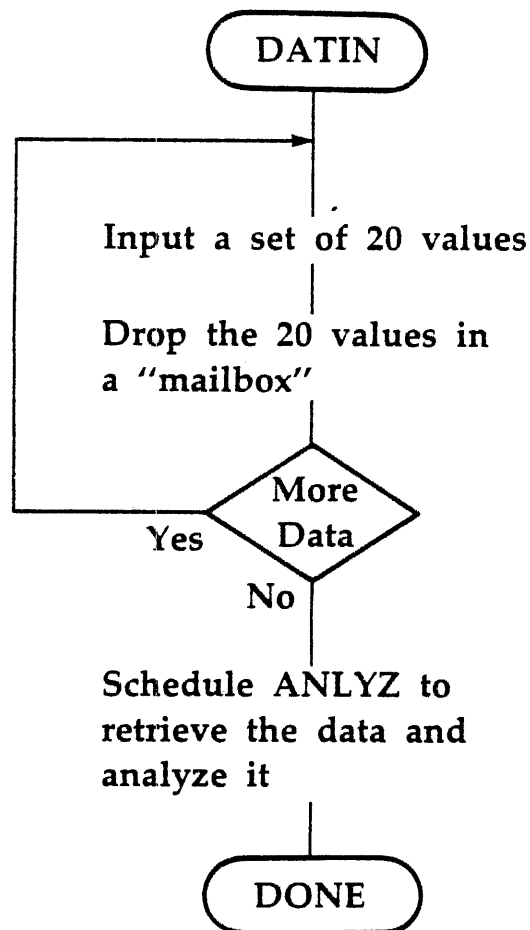
- the class number
- the size of the data area
- other information

DATA BUFFER which contains the data being transferred.

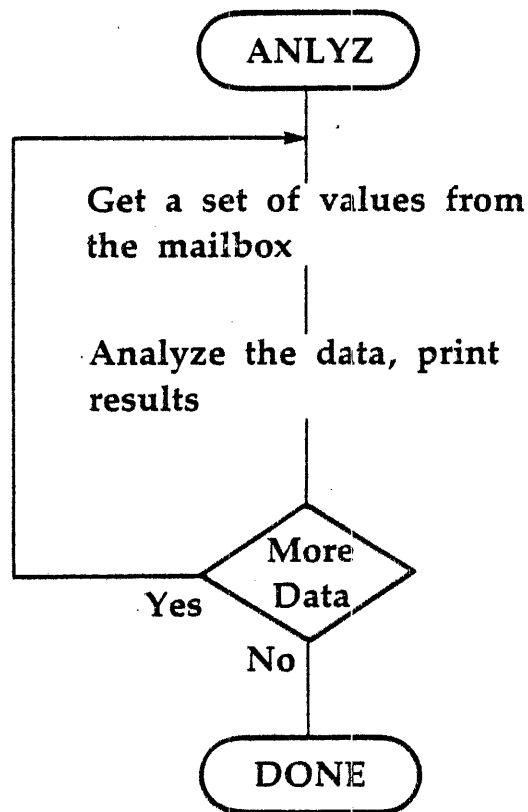
A SAMPLE PROBLEM

Suppose you are conducting an experiment which will produce 20 data values every minute for 10 minutes. You might design two programs to input and analyze each set of 20 values.

Program DATIN is to be scheduled when the experiment begins —



Program ANLYZ will be scheduled by DATIN after all of the data sets have been input and dropped in the mailbox. ANLYZ will then retrieve and analyze each set of data values.





EXEC 20 — CLASS WRITE/READ

An EXEC 20 call manufactures a buffer in SAM, fills it with data from the program and links it to the appropriate completed class queue.

```
CALL EXEC (20, ICNWD, IBUFF, ILEN, IOP1, IOP2, ICLAS)
```

where

ICNWD = 0	the request is issued to LU 0
IBUFF	array containing the data to be transferred
ILEN	positive number of words or negative number of characters to be transferred
IOP1, IOP2	optional parameters; their values are placed in the control part of the manufactured class buffer and can be retrieved by the CLASS GET which consumes the buffer
ICLAS	the class number allocated by RTE

ALLOCATING A CLASS NUMBER

You request a class number by making an EXEC 20 call with ICLAS having a value of 0.

```
.  
.  
LU = 0  
ICLAS = 0  
CALL EXEC (20,LU,IAR1,NWD1,IPA,IPB,ICLAS)
```

↑
RTE will return the class number allocated to you

- You can then use ICLAS to tell another program what class number to use when retrieving data sent to it.
- You can also use ICLAS to do additional CLASS WRITE/READS on the already allocated class number.

```
.  
.  
CALL EXEC (20,LU,IAR2,NWD2,IPC,IPD,ICLAS)
```

↑
RTE uses a previously allocated class number

EXEC 21 – CLASS GET

An EXEC 21 consumes one buffer in the completed class queue of the specified class number.

```
CALL EXEC (21, ICLAS, IBUFF, ILEN, IOP1, IOP2)
```

where

ICLAS	= previously allocated class number
IBUFF	= array where the retrieved data is to be stored
ILEN	= positive number of words, negative number of characters to be retrieved
IOP1 IOP2	= two optional parameters; if specified, they are assigned the values passed when the buffer was manufactured

DEALLOCATING THE CLASS NUMBER

After RTE consumes a buffer in SAM in response to an EXEC 21 call, RTE checks to see if there are more buffers queued on the class number.

If the last buffer was just consumed, RTE deallocates the class number and makes it available for use by other programs.





THE MANUFACTURER

```
0001  FTN4,L
0002      PROGRAM DATIN
0003  C
0004      INTEGER DATA(20), SPROG(3)
0005      DATA SPROG/2HAN,2HLY,2HZ /
0006  C
0007  C      INPUT THE 10 SETS OF 20 VALUES
0008  C
0009      ICLASS = 0
0010      DO 20 I = 1,10
0011  C
0012  C      INPUT A SET OF VALUES
0013  C
0014      CALL EXEC(1,45,DATA,20)
0015      CALL ABREG(IA,IB)
0016  C
0017  C      DROP IN THE MAILBOX
0018  C
0019      CALL EXEC(20,0,DATA,IB,K,L,ICLASS)
0020  C
0021  20  CONTINUE
0022  C
0023  C      AFTER THE DATA HAS BEEN PUT IN THE MAILBOX,
0024  C      SCHEDULE THE ANALYSIS PROGRAM.
0025  C
0026      CALL EXEC(10,SPROG,ICLASS)
0027  C
0028      END
```

.....AND THE CONSUMER 

```
0001  FTN4,L
0002      PROGRAM ANLYZ
0003  C
0004      INTEGER DATA(20), PARM(5)
0005  C
0006  C      RETRIEVE THE CLASS NUMBER
0007  C
0008      CALL RMPAR(PARM)
0009      ICLASS = PARM(1)
0010  C
0011  C      GET EACH SET OF VALUES AND ANALYZE
0012  C
0013      DO 30 I = 1,10
0014  C
0015  C          GET ONE SET OF VALUES
0016  C
0017          CALL EXEC(21,ICLASS,DATA,20)
0018          CALL ABREG(IA,IB)
0019  C
0020  C          ANALYZE THE SET OF VALUES
0021  C
0022          WRITE(1,101) (DATA(J),J=1,IB)
0023 101      FORMAT(20A2)
0024  C
0025 30      CONTINUE
0026  C
0027      END
```

ANOTHER WAY TO PROGRAM OUR EXAMPLE

Why not let DATIN schedule ANLYZ to process the data as it is input rather than waiting until all of the data has been received?

```
0001 FTN4,L
0002     PROGRAM DATIN
0003 C
0004     INTEGER DATA(20), SPROG(3)
0005     DATA SPROG/2HAN,2HLY,2HZ /
0006 C
0007 C     INPUT THE FIRST SET OF 10 VALUES,
0008 C     DROP THEM IN THE MAILBOX, AND
0009 C     SCHEDULE THE ANALYSIS PROGRAM.
0010 C
0011     CALL EXEC(1,45,DATA,20)
0012     CALL ABREG(IA,IB)
0013 C
0014     ICLASS = 0
0015     CALL EXEC(20,0,DATA,IB,K,L,ICLASS)
0016 C
0017     CALL EXEC(10,SPROG,ICLASS)
0018 C
0019 C     NOW INPUT AND SEND THE REMAINING SETS OF VALUES.
0020 C
0021     DO 20 I = 1,9
0022 C
0023 C         INPUT A SET OF VALUES
0024 C
0025     CALL EXEC(1,45,DATA,20)
0026     CALL ABREG(IA,IB)
0027 C
0028 C         DROP IN THE MAILBOX
0029 C
0030     CALL EXEC(20,0,DATA,IB,K,L,ICLASS)
0031 C
0032 20    CONTINUE
0033 C
0034     END
```



RETAINING THE CLASS NUMBER

When a program makes an EXEC 21 call (CLASS GET), the specified class number will be deallocated if the last buffer queued on that class number is consumed by the GET.

Set bit 13 of ICLAS for a "no deallocate" option; the class number will not be returned to the system when the last buffer is consumed.

```
CALL EXEC (21, ICLAS+20000B, IBUFF, ILEN, IOP1, IOP2)
```


..... AND THE CONSUMER — VERSION 2



```
0001 FTN4,L
0002     PROGRAM ANALYZ
0003 C
0004     INTEGER DATA(20), PARM(5)
0005 C
0006 C     RETRIEVE THE CLASS NUMBER
0007 C
0008     CALL RMPAR(PARM)
0009     ICLASS = PARM(1)
0010 C
0011 C     GET EACH SET OF VALUES AND ANALYZE
0012 C
0013     DO 30 I = 1,10
0014 C
0015 C         GET ONE SET OF VALUES,
0016 C         BUT DON'T DEALLOCATE THE CLASS NUMBER.
0017 C
0018         CALL EXEC(21,ICLASS+20000B,DATA,20)
0019         CALL ABREG(IA,IB)
0020 C
0021 C         ANALYZE THE SET OF VALUES
0022 C
0023         WRITE(1,101) (DATA(J),J=1,IB)
0024 101     FORMAT(20A2)
0025 C
0026 30     CONTINUE
0027 C
0028 C     NOW THAT ALL THE DATA SETS HAVE BEEN ANALYZED,
0029 C     USE AN EXTRA GET CALL TO DEALLOCATE THE CLASS NUMBER.
0030 C
0031     CALL EXEC(21,ICLASS,DATA,20)
0032 C
0033     END
```

17C. CLASS I/O — A SUMMARY OF FEATURES

Programs may use class I/O for —

- program to program communication
- input/output requests to peripheral devices
- control requests to peripheral devices

All types of class I/O share these features —

- data transfers are done via buffers in SAM
- CLASS I/O is “double call”
 - one call manufactures a buffer in SAM
 - a second call consumes a buffer in SAM
- buffers are queued on class numbers, the “keys” to accessing data
- buffers may be manufactured and consumed asynchronously

CLASS I/O vs OTHER I/O

All types of I/O must specify the:

- LU of the device
- buffer containing or receiving the data
- the number of words or characters to be transferred

Various forms of I/O differ by:

	Number of EXEC calls	Location of buffer used by driver	Program swappable ?	Program waits ?
Normal I/O (unbuffered)				
Automatic output buffering				
REIO, input				
REIO, output				
CLASS I/O				

17D. CLASS I/O FOR DEVICE I/O AND CONTROL

The class I/O EXEC requests are:

EXEC 17 — CLASS READ

EXEC 18 — CLASS WRITE

EXEC 19 — CLASS CONTROL

EXEC 20 — CLASS WRITE/READ

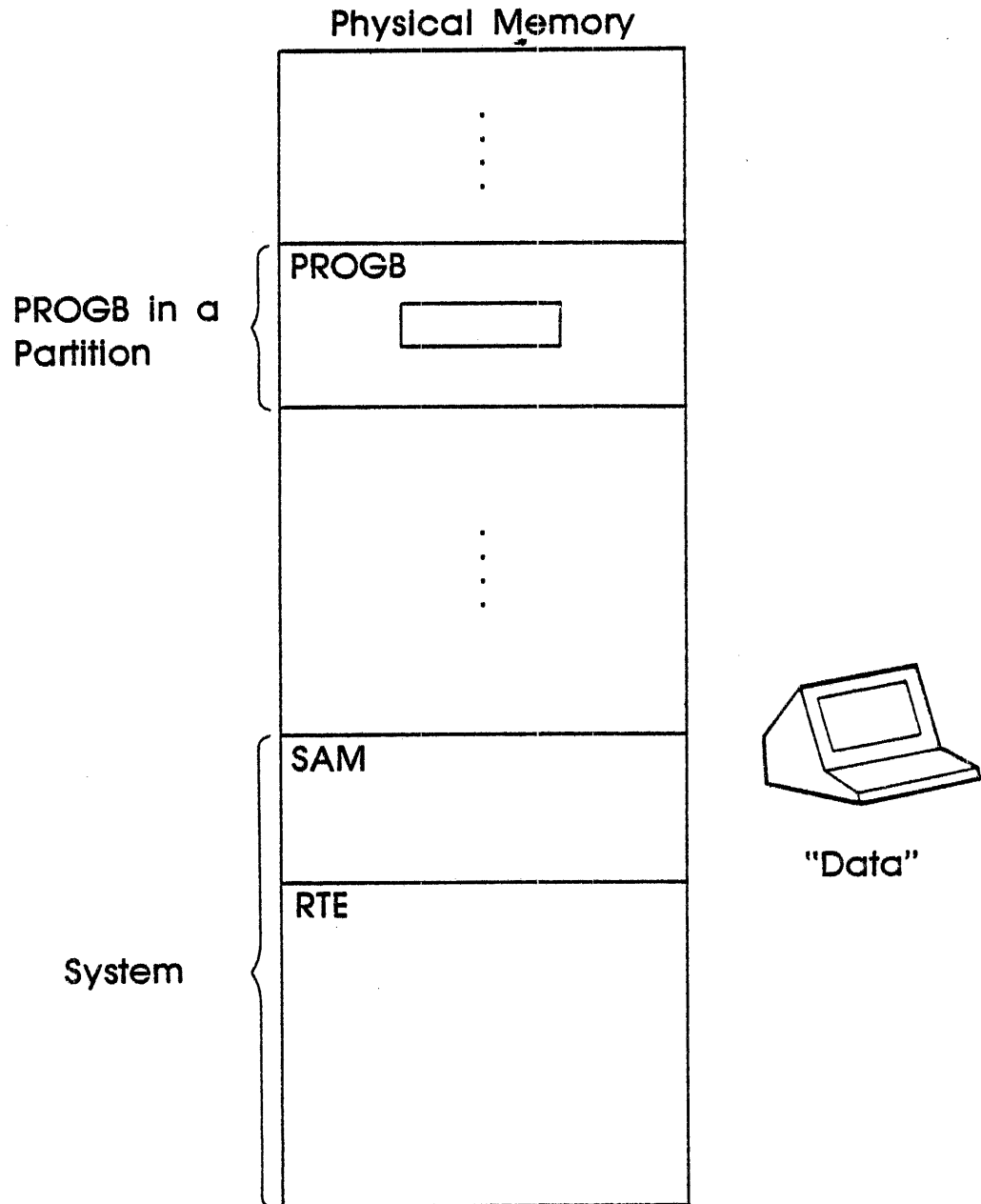
EXEC 21 — CLASS GET

CLASS I/O FOR INPUT

Suppose PROGB wants to input some values from a terminal into an array. Using CLASS I/O, the program might look like this —

```
PROGRAM PROGB
  :
  :
C   REQUEST INPUT OF DATA - CLASS READ
C   :
C   CALL EXEC(17,.....)
C   :
C   CONTINUE EXECUTION WHILE RTE DOES THE I/O
C   :
C   :
C   RETRIEVE THE DATA THAT WAS INPUT - CLASS GET
C   :
C   CALL EXEC(21,.....)
  :
  :
```

CLASS READ OPERATION





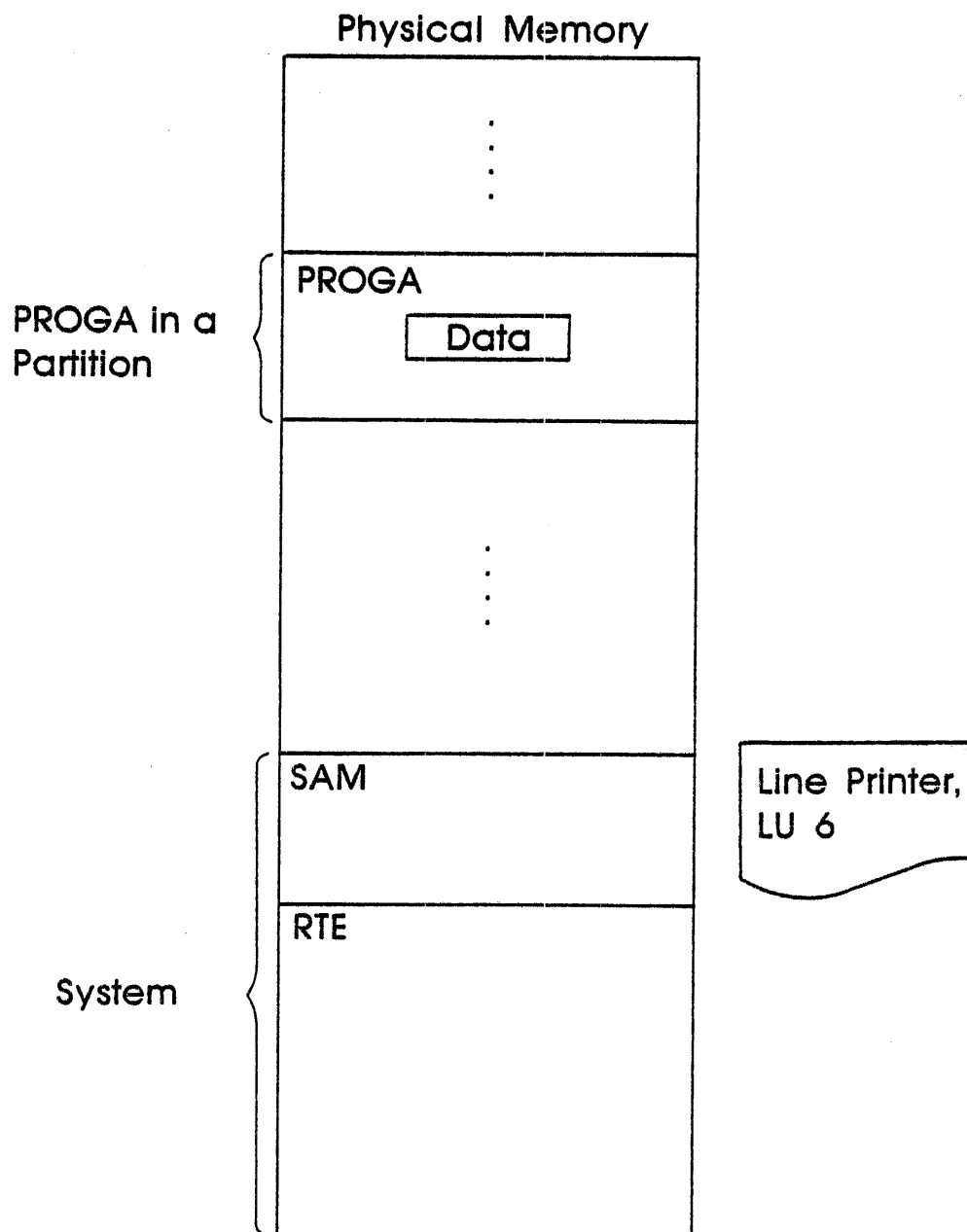
CLASS I/O FOR OUTPUT

Suppose PROGA wants to output a buffer to the line printer (LU 6). Using CLASS I/O, the program might be structured like this —

```
PROGRAM PROGA
  .
  .
  .
C   OUTPUT DATA - CLASS WRITE
C   CALL EXEC(18,.....)
C   CONTINUE EXECUTION WHILE RTE DOES THE I/O
C   .
C   .
C   .
C   COMPLETE THE OPERATION - CLASS GET
C   CALL EXEC(21,.....)
  .
  .
  .
```



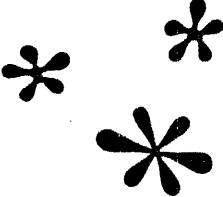
CLASS WRITE OPERATION



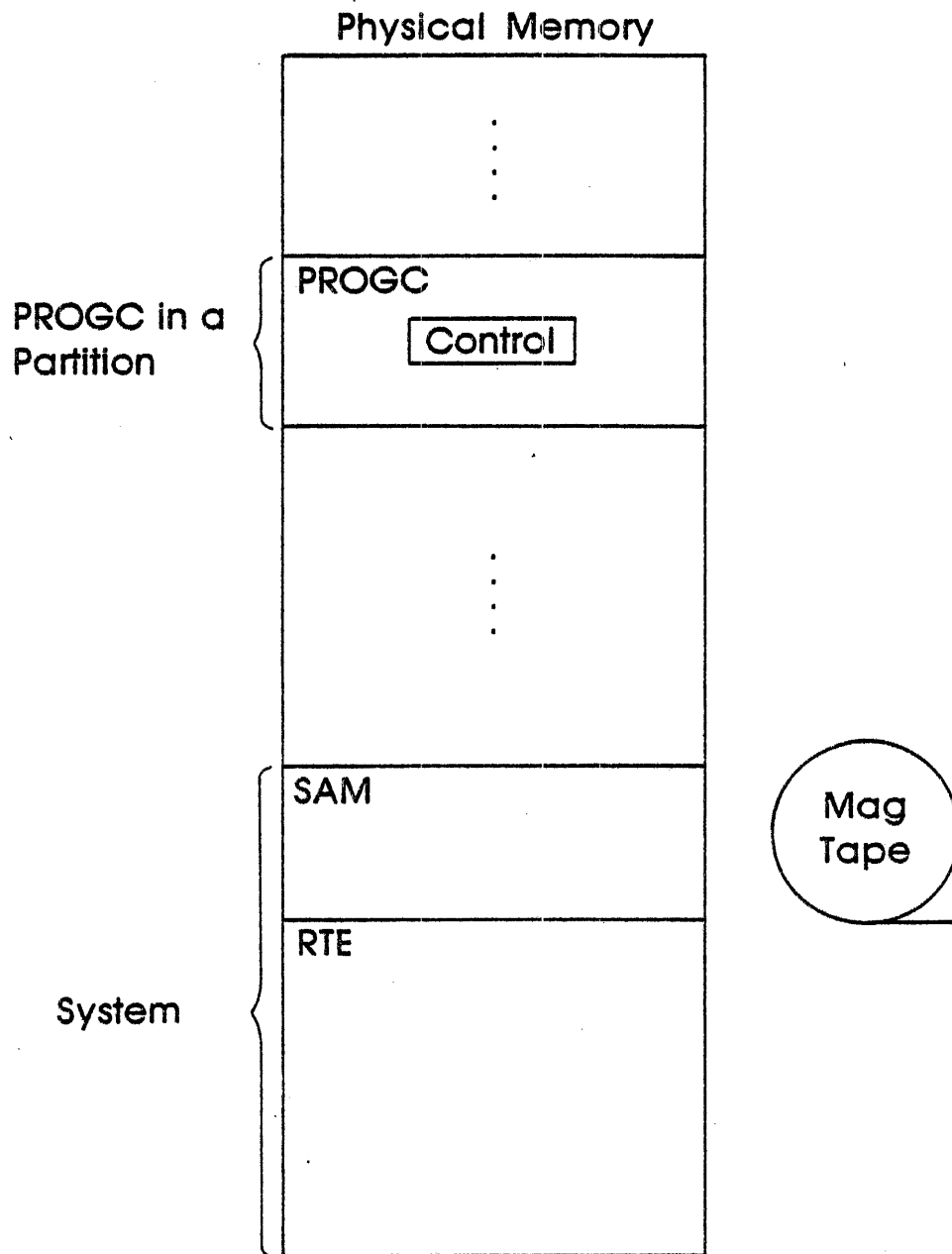
CLASS I/O FOR DEVICE CONTROL

Suppose PROGC wants to rewind the tape on the mag tape drive. Using CLASS I/O the program might be —

```
PROGRAM PROGC
  .
  .
  .
C   REQUEST RTE REWIND THE TAPE - CLASS CONTROL
C   CALL EXEC(19,.....)
C   CONTINUE EXECUTION WHILE RTE REWINDS THE TAPE
C   .
C   .
C   .
C   COMPLETE THE OPERATION - CLASS GET
C   CALL EXEC(21,.....)
  .
  .
  .
  .
```

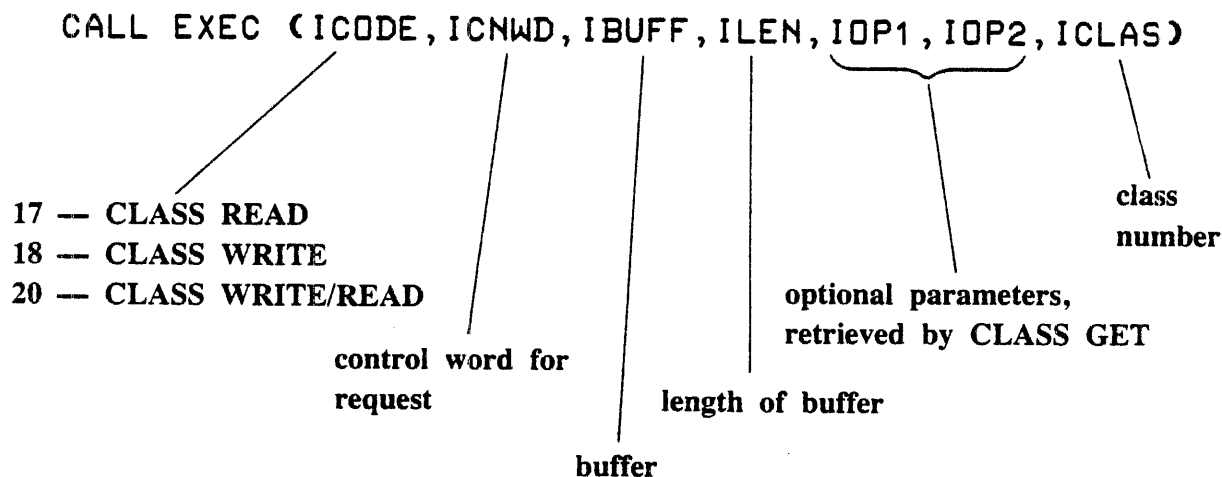


CLASS CONTROL OPERATION



• CLASS WRITE/READ •
• CLASS WRITE • CLASS READ •

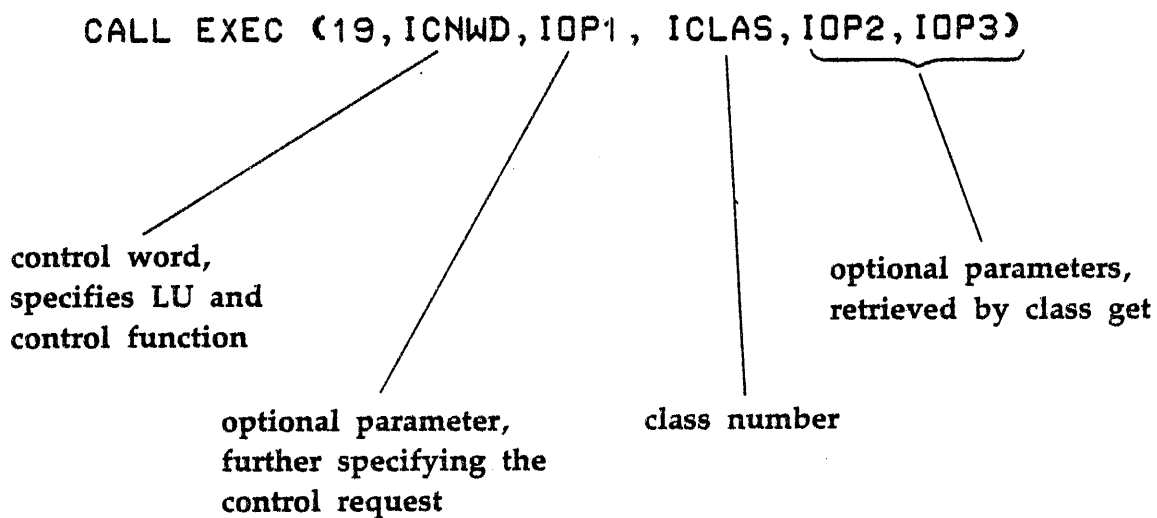
The CLASS WRITE/READ, CLASS WRITE, and CLASS READ all have the same call format.



These calls manufacture one buffer in SAM.

• CLASS CONTROL •

The CLASS CONTROL call format is:

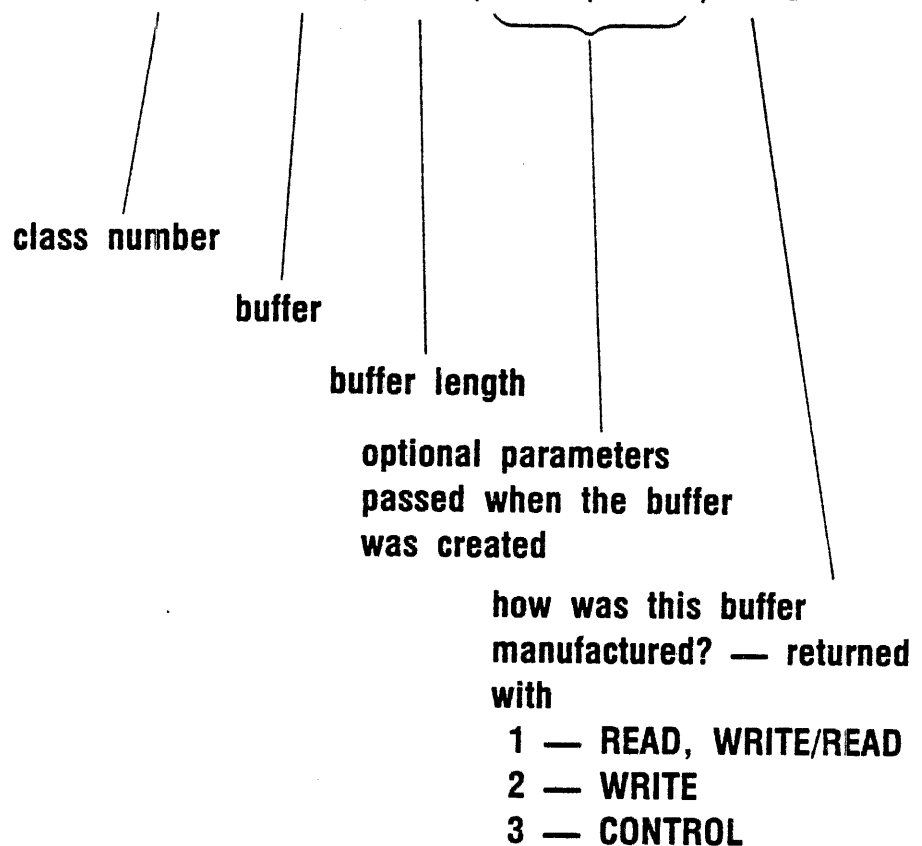


This call manufactures one buffer in SAM.

• CLASS GET •

A CLASS GET call consumes one buffer in SAM, regardless of how the buffer was manufactured.

CALL EXEC (21, ICLAS, IBUFF, ILEN, IOP1, IOP2, IOP3)



A QUICK EXAMPLE

These FORTRAN statements prompt for an input of 2 characters:

```
INTEGER IPRMT(4)
DATA IPRMT/2HIN,2HPU,2HT?,2H_b/
.
.
.
CALL EXEC (2,1,IPRMT,4)
CALL EXEC (1,1+400B,IANS,1)
.
.
.
```

Using CLASS I/O for the I/O operations:

```
INTEGER IPRMT(4)
DATA IPRMT/2HIN,2HPU,2HT?,2H_b/
.
.
.
a CLASS WRITE, CLASS READ
and 2 CLASS GETS
.
.
.
```



GET A BUFFER IN WHAT ORDER?

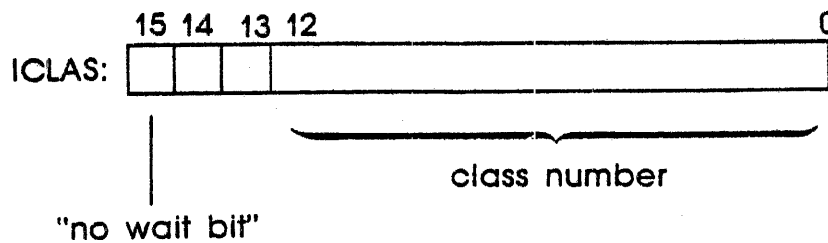
- * SAM buffers manufactured by CLASS WRITE/READS are retrieved in the order in which they were created.
- * SAM buffers manufactured by CLASS READS, CLASS WRITES, or CLASS CONTROLS are retrieved in the order in which they were completed.



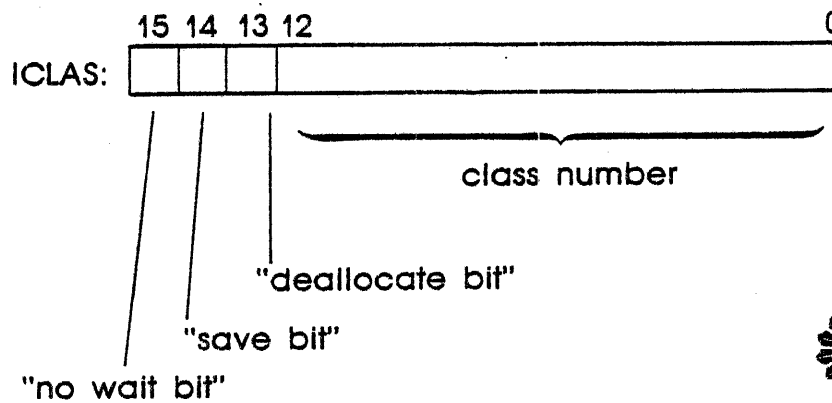
17E. VARIATIONS WITH CLASS I/O

The ICLAS parameter of the CLASS I/O calls allows for several options in using CLASS I/O.

CLASS WRITE/READ, CLASS WRITE, CLASS READ, CLASS CONTROL



CLASS GET





WAITING FOR A CLASS NUMBER OR SAM?

When a program makes an EXEC 17, 18, 19 or 20 call, it will be suspended if:

- there were no available class numbers and one was requested
- there was not enough SAM to hold the class buffer

Set bit 15 of ICLAS for a "no wait" option. The program will not be suspended if a resource is unavailable and the A-register will be set to:

- -1 if no available class number
- -2 if insufficient SAM



WHY WAIT FOR YOUR DATA?

If a program does a CLASS GET on a class number before a buffer has been manufactured by a CLASS READ, CLASS WRITE or CLASS CONTROL, RTE will suspend the program until a buffer is available.

When a buffer is placed in the appropriate class queue, RTE will "wake up" the suspended program.

***This feature allows class I/O to be used for
PROGRAM SYNCHRONIZATION**

To avoid being "put to sleep" until the data is available, set bit 15 of ICLAS to select the "no wait" option. Upon return from the CLASS GET call, the A register will be

positive — if data was available and consumed
negative — if no data was available



GETTING YOUR DATA TWICE

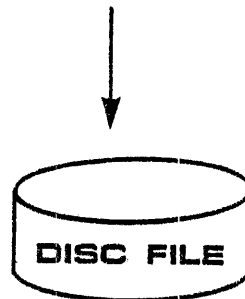
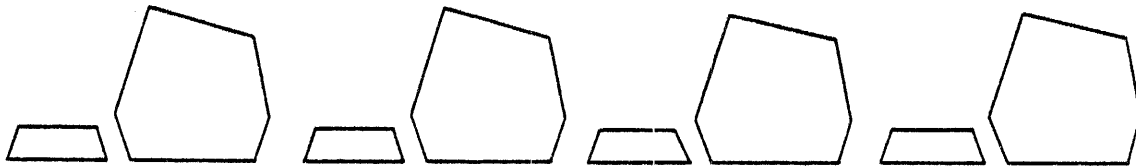
When a program makes a CLASS GET call, the first data buffer in the appropriate class queue is consumed. A subsequent CLASS GET will consume the next data buffer.

Set bit 14 of ICLAS to retrieve the contents of a data buffer but leave the data buffer in the class queue. A subsequent CLASS GET will get the same data buffer.



17F. TERMINAL HANDLERS

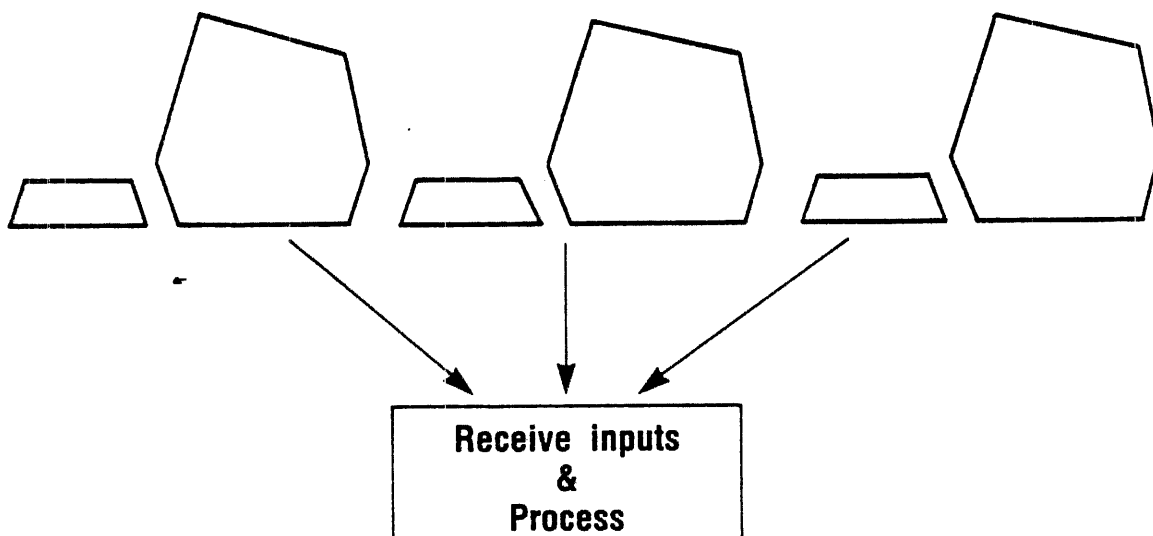
Suppose operators at several terminals are entering data which is used to update a disc file.



TERMINAL HANDLERS

Since CLASS I/O allows input without wait, one program can easily handle inputs from several terminals “simultaneously.”

- the program can issue CLASS READS to several terminals without waiting for completion.
- the program (or another program) uses CLASS GETS to retrieve the inputs from the terminals as they are completed.



A VERY SIMPLE EXAMPLE

Consider a program which will

- prompt 3 terminals for a string of 10 characters
- process the input by printing each string on the line printer, along with the LU of the terminal which supplied the string

```
PROGRAM TERMS
INTEGER LUS(3)
DATA LUS/15,16,17/
.
.
.
C
C   ISSUE 3 PROMPTS AND READS
C
DO 10 I=1,3
  LU=LUS(I)
  WRITE(LU,101)
101  FORMAT(/"INPUT 10 CHARACTERS:_" )
      issue CLASS READ
10  CONTINUE
.
.
C
C   RETRIEVE INPUTS
C
DO 20 I=1,3
      issue CLASS GET
      print string and LU
20  CONTINUE
.
.
.
```

18

MORE RTE SERVICES

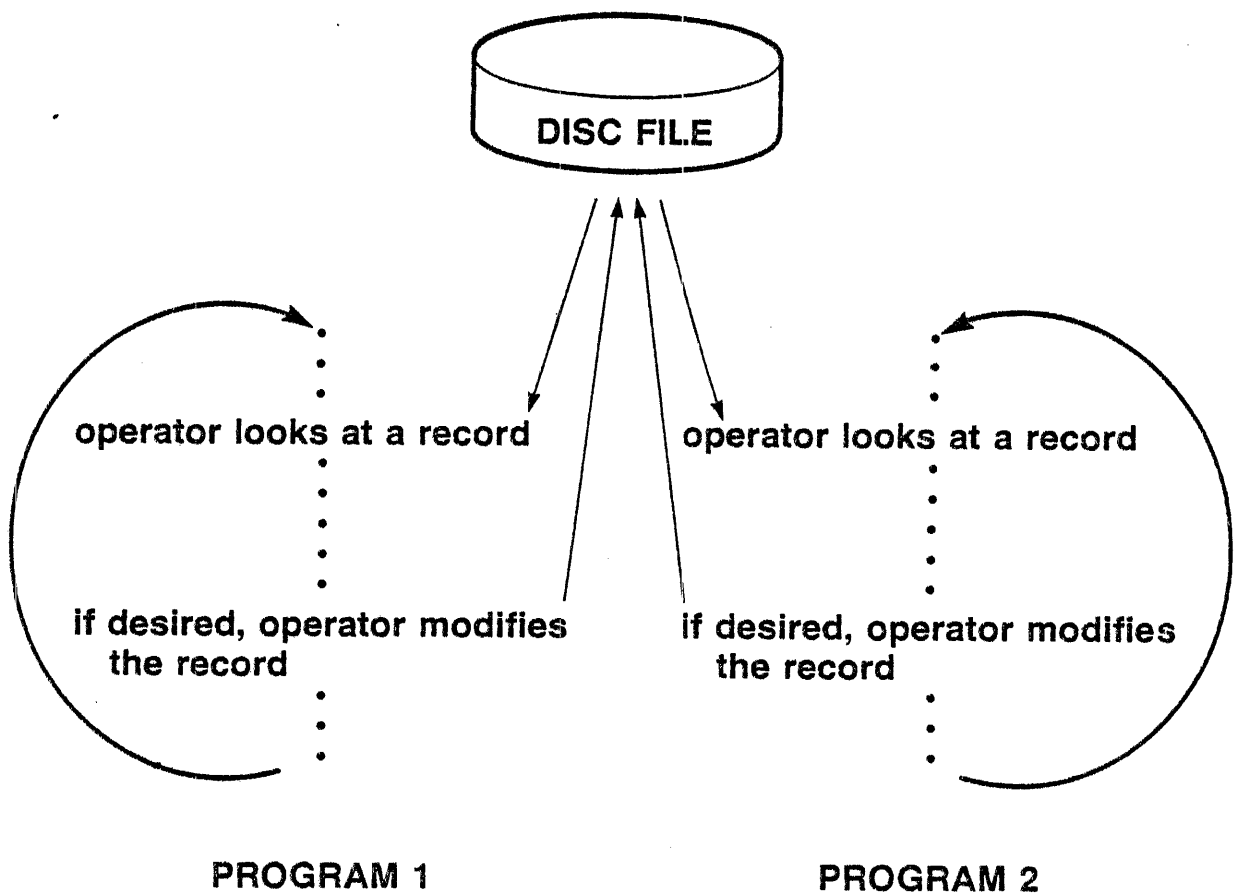


SECTION

A	RESOURCE NUMBERS AND LU LOCKS	18-3
B	EXEC CALLS TO THE DISC	18-14
C	LARGE PROGRAMS	18-19

18A. RESOURCE NUMBERS AND LU LOCKS

Suppose you have two programs, each accepting operator requests to examine and possibly modify the records in one disc file.



CRITICAL SECTIONS

If you have several programs trying to use the same resource,

- **a disc file**
- **a memory location**
- **a peripheral device**
- **a program**

and only one program should use the resource at a time, then the segment of code in each program which manipulates the resource can be considered to be a CRITICAL SECTION.

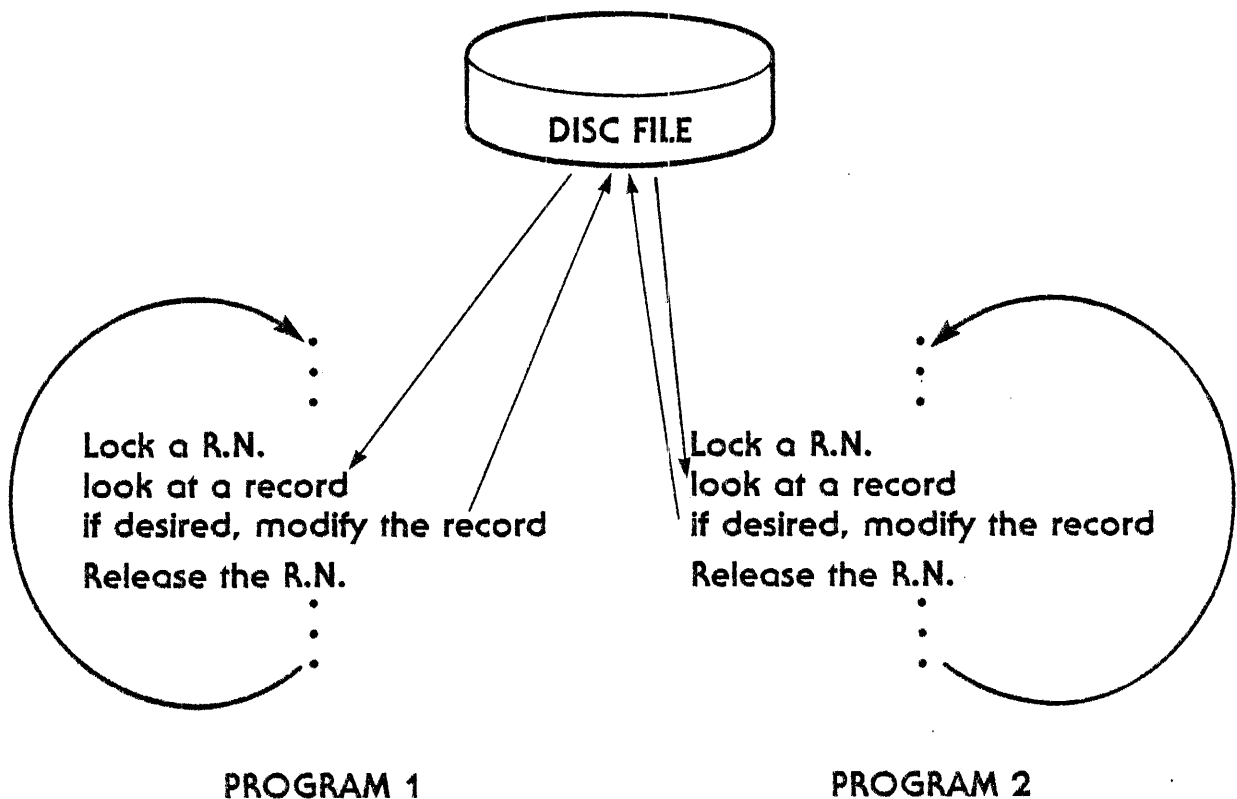
Only one program at a time should be executing in its critical section!

RESOURCE NUMBERS

One of RTE's services is the management of RESOURCE NUMBERS.

Your programs can use Resource Numbers to protect critical sections of code so that only one program will use a protected resource at a time.

The number of Resource Numbers in an RTE system is specified when the system is generated.



RNRQ

RESOURCE NUMBER MANAGEMENT

The library routine RNRQ lets you allocate, deallocate, lock and clear RESOURCE NUMBERS (RN's). Cooperating programs can use RNRQ to coordinate their use of shared resources.

`CALL RNRQ(ICODE, IRN, ISTAT)`

specifies operation
desired

- allocate
- deallocate
- lock
- unlock

resource number,
allocated or
manipulated

returned status
of request

TYPES OF REQUESTS

In a call to RNRQ,

- the first parameter indicates the request —

	15	14	5	4	3	2	1	0
	WAIT OPTION		ALLOCATE OPTION			LOCK OPTION		
ICODE=	NO W A I T	NO A B O R T	D E A L L O C A T E	G L O B A L	L O C A L	U N L O C K	G L O B A L	L O C A L

- the last parameter indicates the status of the request —

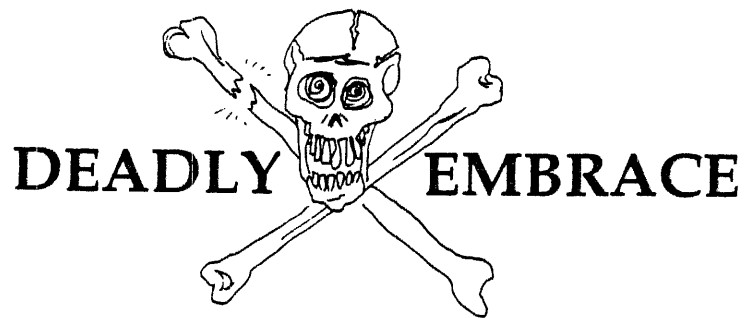
- ISTAT= 0 NORMAL DEALLOCATE
(RETURNED)
- 1 RN IS CLEAR (UNLOCKED)
 - 2 RN IS LOCKED LOCALLY TO CALLER
 - 3 RN IS LOCKED GLOBALLY
 - 4 NO RN AVAILABLE NOW
 - 5 —
 - 6 RN IS LOCKED LOCALLY TO ANOTHER PROGRAM
 - 7 RN WAS LOCKED GLOBALLY WHEN REQUEST WAS MADE

NOTE: STATUS 4, 6, AND 7 ARE RETURNED ONLY IF "NO WAIT" BIT IS SET.

TO ABORT OR NOT TO ABORT

If a call to RNRQ causes an error, RTE will abort the calling program. By setting bit 14 of ICODE, a program can process its own errors (i.e., continue execution).

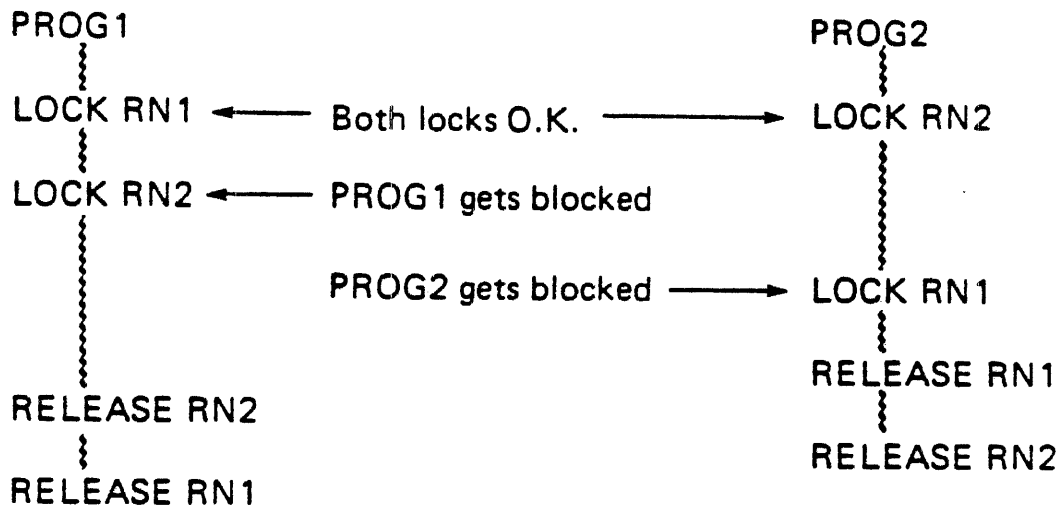
```
error →          CALL RNRQ(ICODE+40000B, IRN, ISTAT)
no error → 44     GO TO 99
                  CONTINUE
                  .
                  .
                  . } continue processing
                  .
                  .
                  .
99                CONTINUE
                  CALL ABREG(IA, IB)
                  WRITE(1, 101) IA, IB
101              FORMAT(/"RN ERROR", 2A2)
                  STOP
                  .
                  .
                  .
```



Programs using RN's can become deadlocked in a "deadly embrace".

This occurs when each program has what the other one wants, so neither program can proceed.

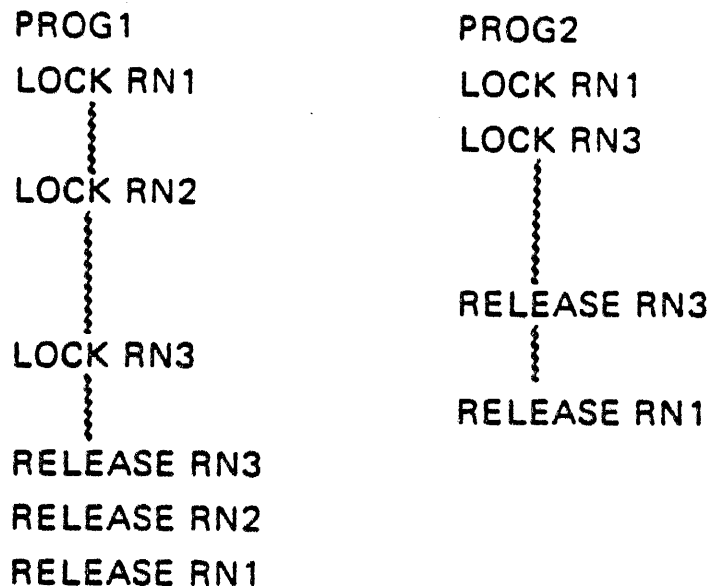
Example:



AVOIDING DEADLY EMBRACE

1. All programs lock RN's in same order.

Example:



2. Logically associate the resources to be protected with one RN instead of several RN's.

LU LOCK

- Use the library subroutine **LURQ** to exclusively dominate (lock) a group of LUs.
- **LURQ** uses one RN to lock all the specified LU's to the program.

CALL LURQ(ICON, LUARY, NUM)

control option,
specifies type
of request

- lock
- unlock all
- unlock some

array containing
LU's to be
locked/unlocked

number of
LU's to be
locked/unlocked

CONTROL OPTIONS

- Let ICON have these values to:

0 0 0 0 0 0 B — unlock the specified LU's

1 0 0 0 0 0 B — unlock all LU's currently locked

0 0 0 0 0 1 B — lock (with wait) the specified LU's

1 0 0 0 0 1 B — lock (without wait) the specified LU's

- If the lock without wait option is selected, the A-register is returned with:

0 lock successful

-1 no RN available

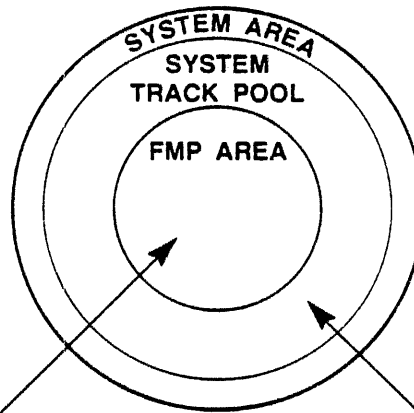
1 one or more LU's are already locked

- Bit 14 is the "no abort bit"

```
error    →      CALL LURQ (ICON+40000B,LUARY,NUM)
no error →45     GO TO 99
          CONTINUE
```

18B. EXEC CALLS TO THE DISC

LU 2 — the system cartridge

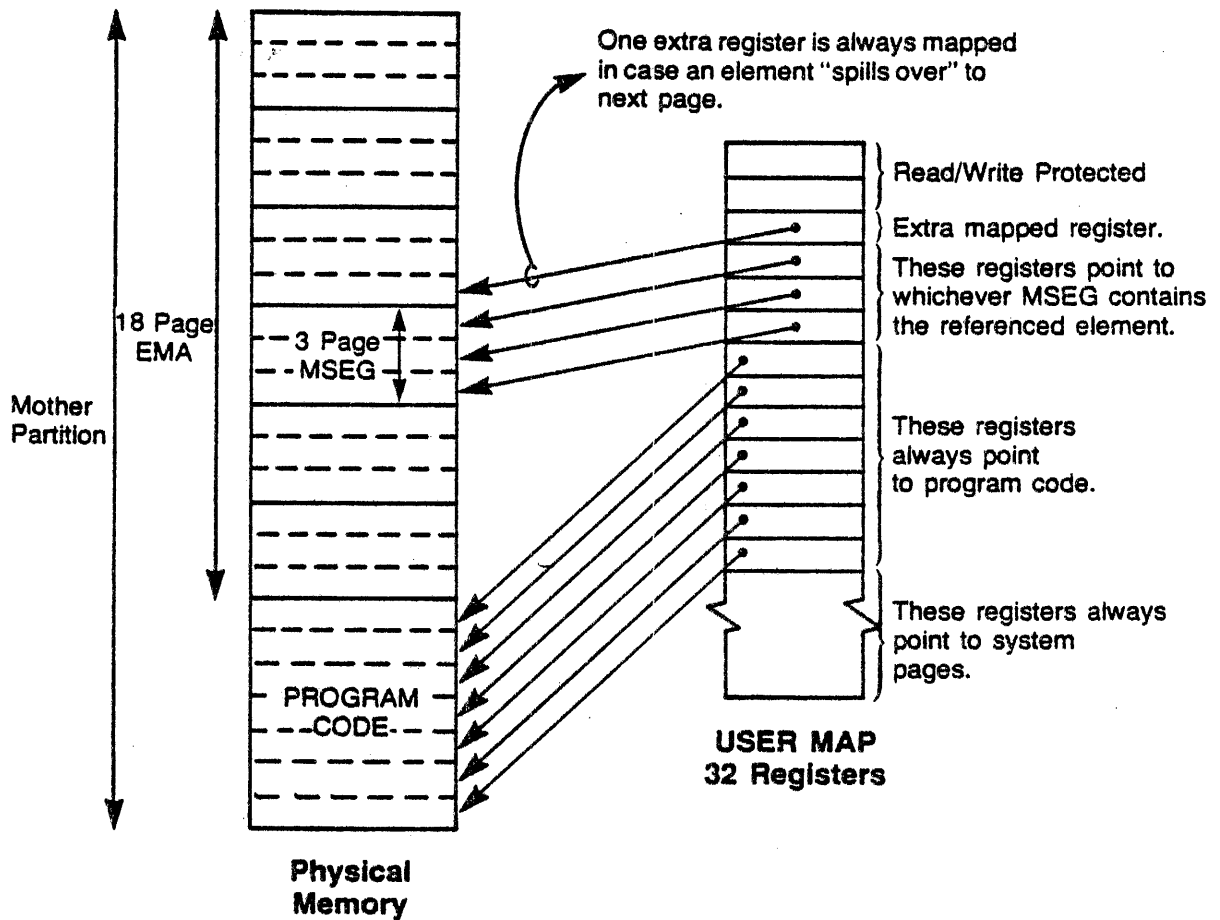


The File Management
System manages this area, letting you access the disc via named files.

Programs can use tracks in the system track pool as temporary storage space.

EMA IN A NUTSHELL

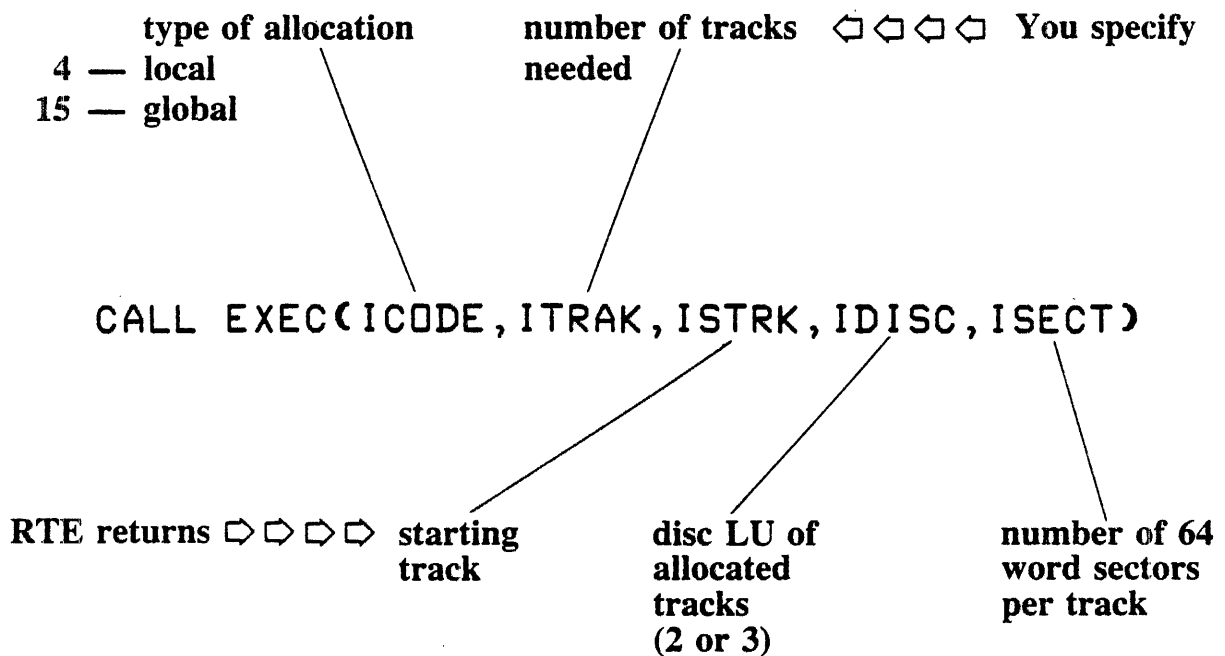
- User map registers change to point to different physical pages as needed to reference data.



$$(\text{System Pages} + \text{Program Size} + \text{MSEG size} + \text{EXTRA PAGE}) \leq 32.$$

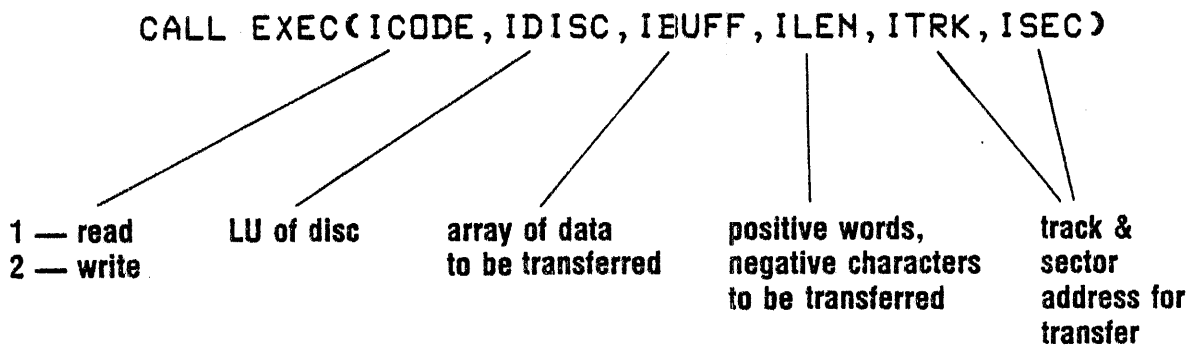
ALLOCATING TRACKS

Use an EXEC 4 or 15 request to allocate space from the system track pool:



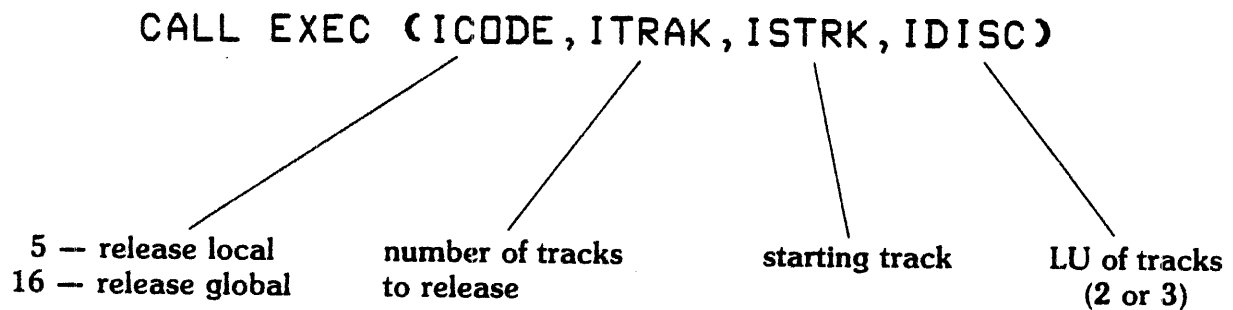
ACCESSING TRACKS

The normal EXEC I/O calls are used to access allocated tracks.



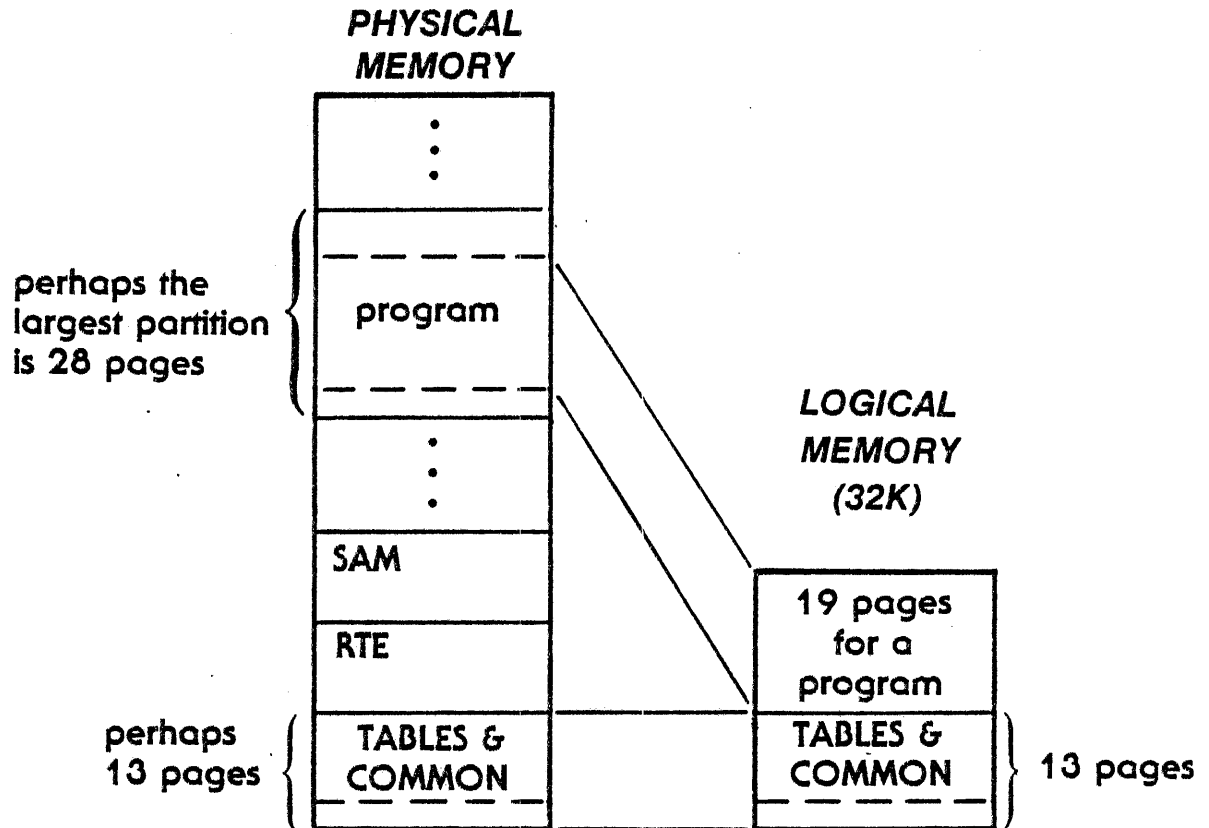
DEALLOCATING TRACKS

The EXEC 5 or 16 calls deallocate tracks from the system track pool.



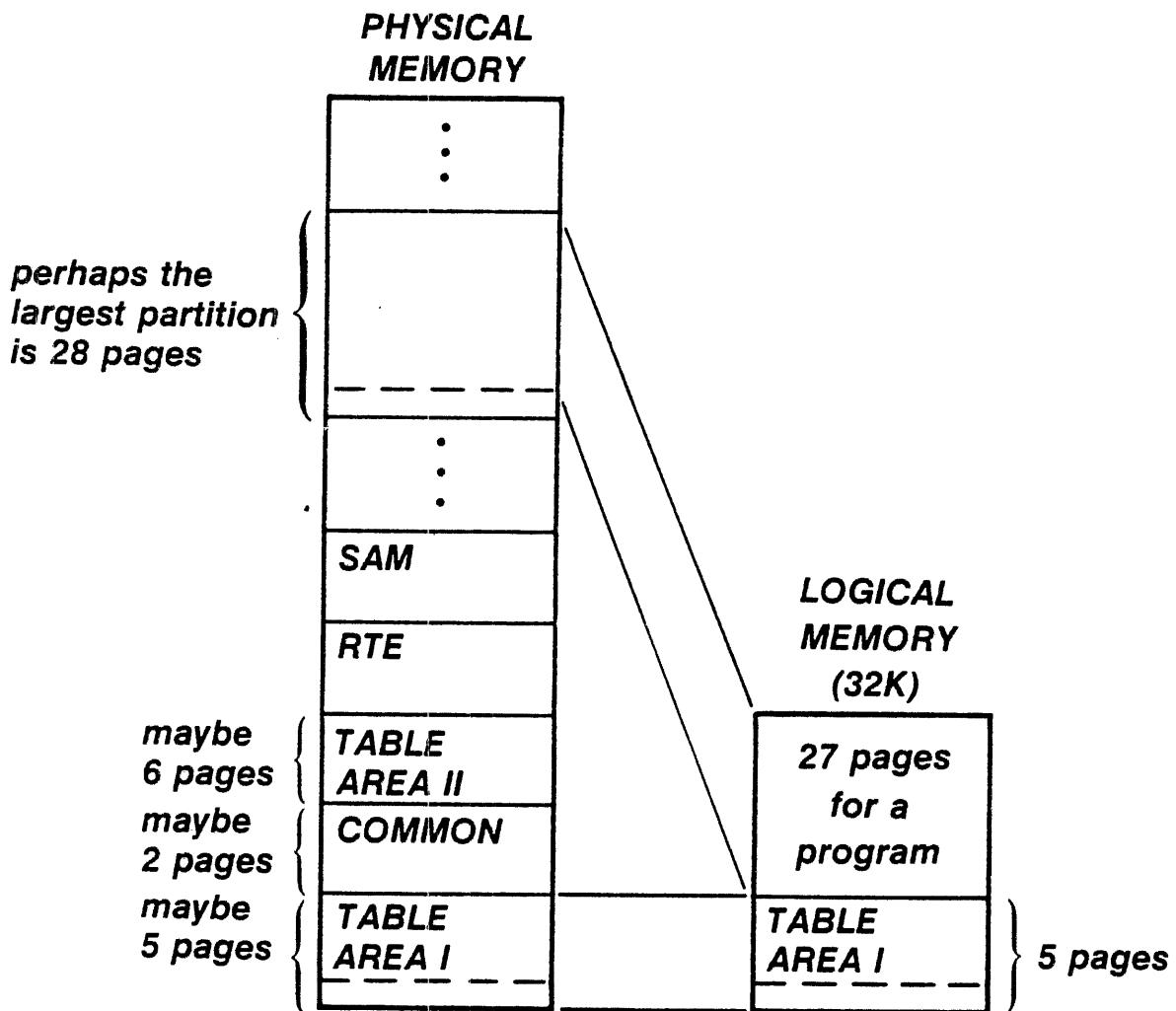
18C. LARGE PROGRAMS

A sample memory configuration might be —



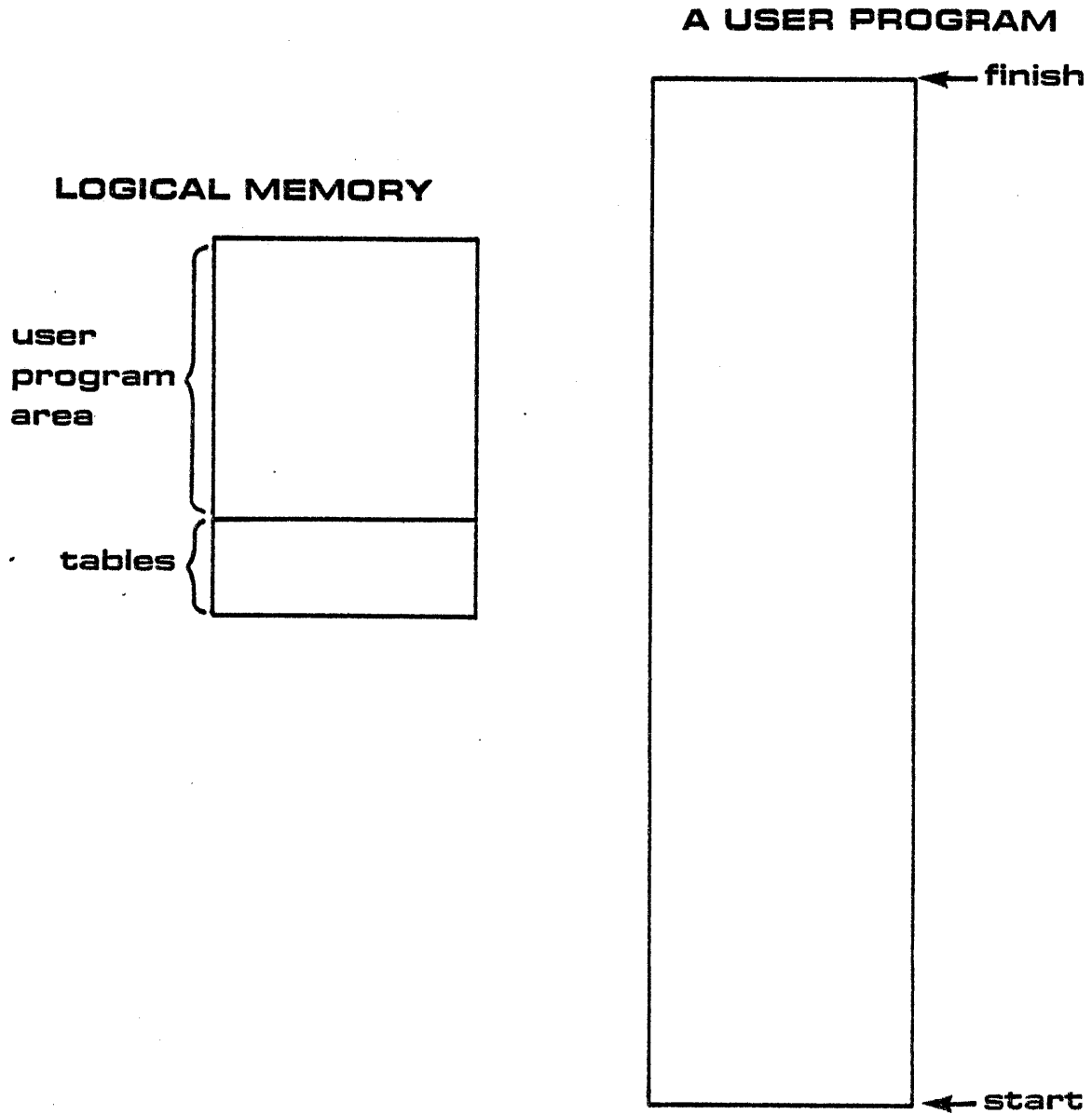
LARGE BACKGROUND PROGRAMS

Many programs do not need access to all of the information in RTE's "tables area". Such programs may be loaded as **LARGE BACKGROUND PROGRAMS** or **TYPE 4 PROGRAMS**.



Use the "OP, LB" op code when using **LOADR** to load a **LARGE BACKGROUND PROGRAM**.

STILL TOO BIG?



Program segmentation involves breaking up a large program into small pieces so that the pieces fit into logical memory.

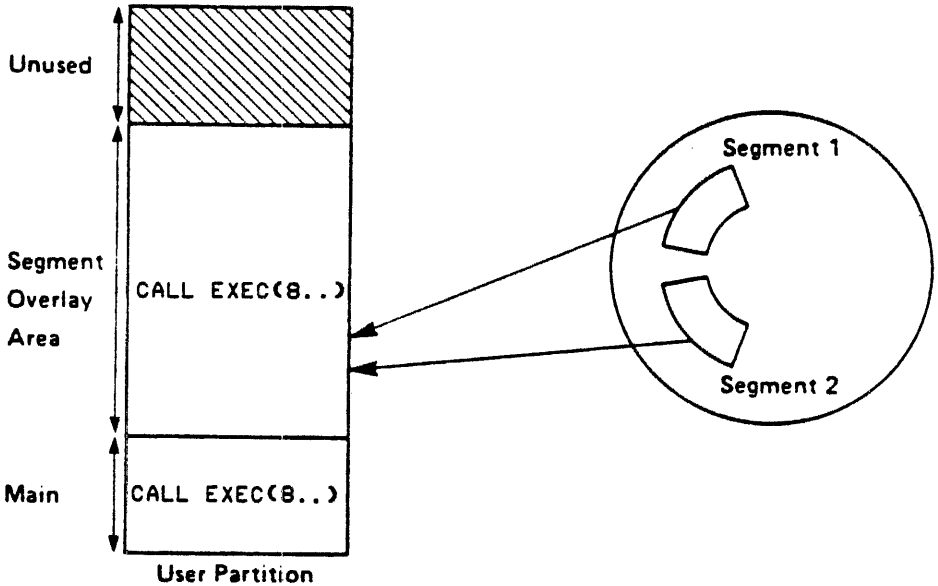
EXEC SEGMENT LOADS

An EXEC 8 request tells RTE to load a segment from disc into memory and transfer control to it.

```
CALL EXEC(8, INAME, IOP1, IOP2, IOP3, IOP4, IOP5)
```

array containing the name of a segment

optional parameters passed to the segment



EXAMPLE

FTN4,L

```
PROGRAM HOW(3)
DIMENSION NAME(3)
DATA NAME/2HHO,2HW1,2H /
WRITE(1,10)
10  FORMAT("HOW ")
CALL EXEC(8,NAME)
END
```

```
PROGRAM HOW1(5)
DIMENSION NAME(3)
DATA NAME/2HHO,2HW2,2H /
WRITE(1,10)
10  FORMAT("ARE ")
CALL EXEC(8,NAME)
END
```

```
PROGRAM HOW2(5)
WRITE(1,10)
10  FORMAT("YOU?")
CALL EXEC(6)
END
END$
```

LOADING SEGMENTED PROGRAMS

Since LOADR must process segmented programs differently than ordinary programs, some pre-load preparation will help reduce loading time.

The modules associated with a segmented program can be grouped as —

- **the main**
- **the segments**
- **subprograms referenced by the main and/or one or more of the segments**

LOADR COMMAND FILES FOR SEGMENTED PROGRAMS

A LOADR command file for a segmented program might be organized like this:

```
RE, main
SE, library
RE, segment 1
SE, library
RE, segment 2
SE, library
.
.
.
RE, segment n
SE, library
EN
```

LONG vs SHORT ID SEGMENTS

Main programs are identified by "long ID Segments." Program segments are identified by "short ID Segments."

When a segmented program is loaded, LOADR fills in a blank long ID Segment for the main and one blank short ID Segment for each program segment.

To remove a segmented program from the system, you must release

- the long ID Segment
- each short ID Segment

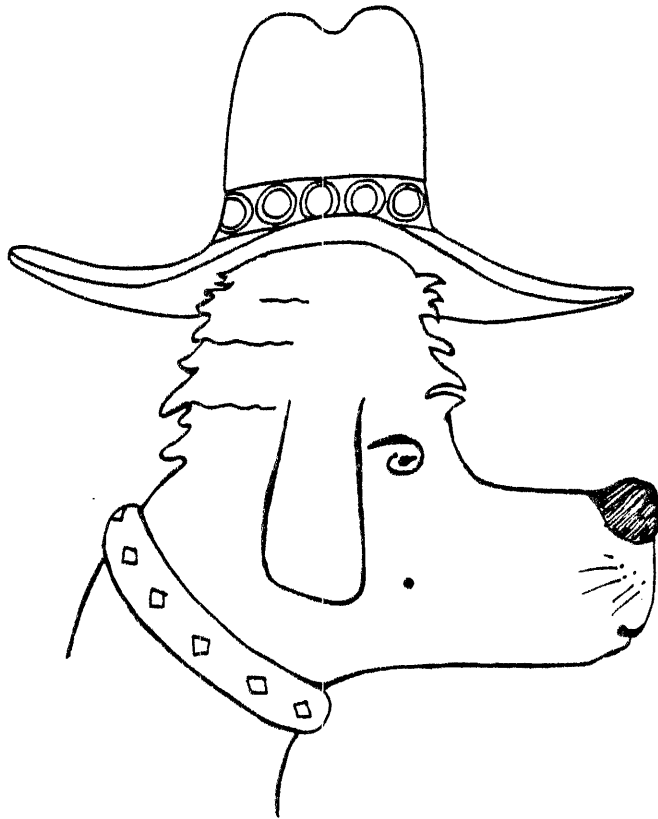
SEGMENTED PROGRAMS and TYPE 6 FILES

To save a segmented program as a type 6 file, the main and each of the program segments need to be "SP'd".

```
: SP, MAIN
: SP, SEG01
: SP, SEG02
: SP, SEG03
: SP, SEG04
: OF, MAIN
: OF, SEG01
: OF, SEG02
: OF, SEG03
: OF, SEG04
: RP, SEG01
: RP, SEG02
: RP, SEG03
: RP, SEG04
: RU, MAIN ← runs the segmented
              program from the
              type 6 file
```

19

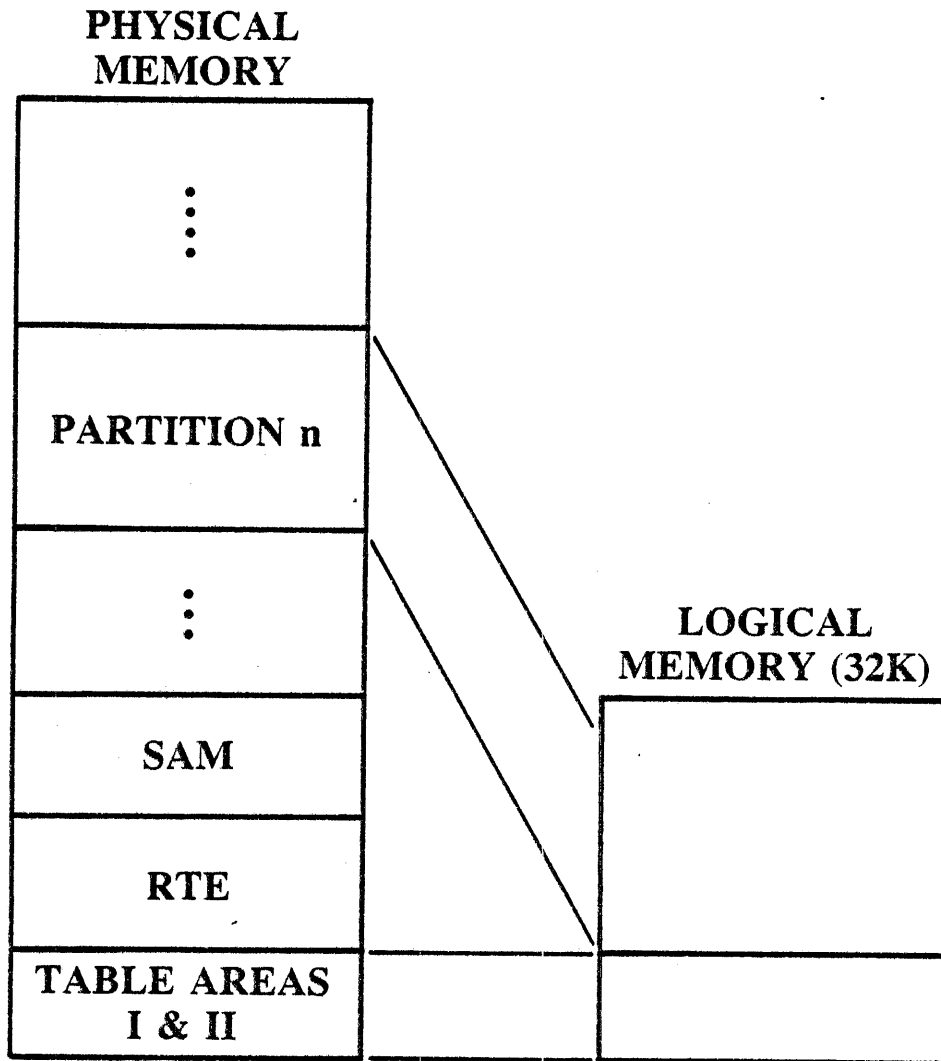
EXTENDED MEMORY AREA (EMA)



SECTION

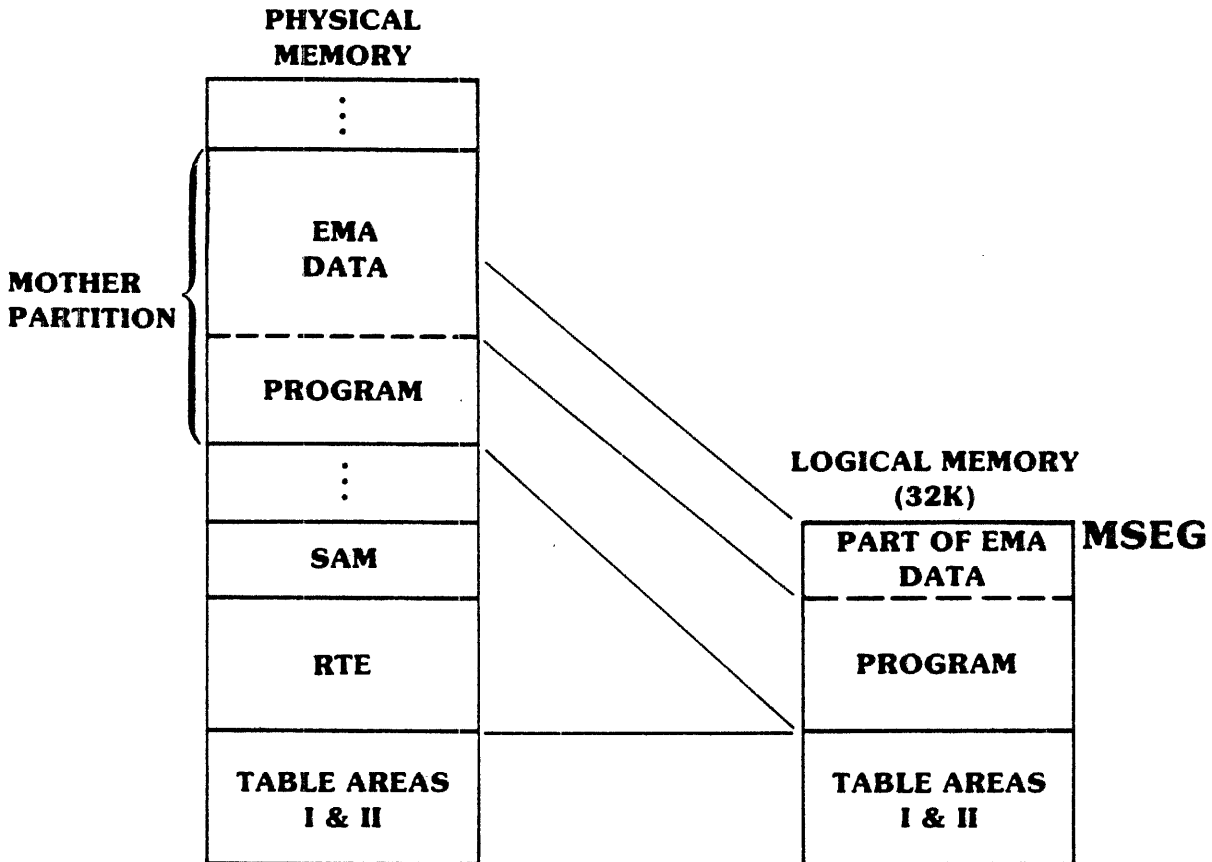
A	WHAT IS EMA?	19-3
B	USING EMA FROM FORTRAN	19-7
C	HOW EMA WORKS	19-14

19A. WHAT IS EMA?



A user's program (code & data) must fit into 32K words (minus some space for RTE tables, etc.)

MOTHER PARTITIONS



REAL-TIME and BACKGROUND partitions may be combined to form a MOTHER PARTITION containing

- a program
- an EMA data area

The size of a Mother Partition is limited only by the amount of physical memory available for partitions.

◀ WHAT IS EMA? ▶

- A means to store and access large amounts of data **IN MEMORY**.
 - data is stored beyond a program's logical address space.
 - the size of the EMA data area may extend to all available physical memory.

- RTE dynamically "maps in" that part of the EMA data area which is being referenced by a program.
 - EMA operation is "transparent" to the FORTRAN programmer.
 - customized mapping schemes can be implemented in Assembly Language.

WHEN TO USE EMA?



- Linear Programming or Matrix Manipulation with many large data arrays.

```
DIMENSION A(10000), B(10000), C(10000)..., Z(10000)
DO 10 J = 1,10000
A(J) = B(J) * B(J) + C(J) ... + Z(J) * G(J)
10 CONTINUE
```

Virtual schemes off the disc will page fault for each array access leading to thrashing. EMA only has to modify user map registers to access the elements needed.

- Random access of large amounts of data (e.g., hashing)

```
INTEGER HASH
DIMENSION IA(20000), IB(20000), IC(20000), IDATA(10), ISYM(10)
DO 10 L = 1,10
IKEY = HASH(ISYM(L))
IDATA(L) = IB(IKEY)*IC(IKEY)
10 CONTINUE
```

Subroutine HASH returns a key used to index into arrays IB and IC.

- When fast coding is necessary.

When writing FORTRAN programs, the program developer can let the system resolve references to EMA elements.

- When real-time acquisition and processing of large amounts of data is needed.

Using customized mapping schemes written in Assembly Language, large amounts of data may be stored and processed in real-time.

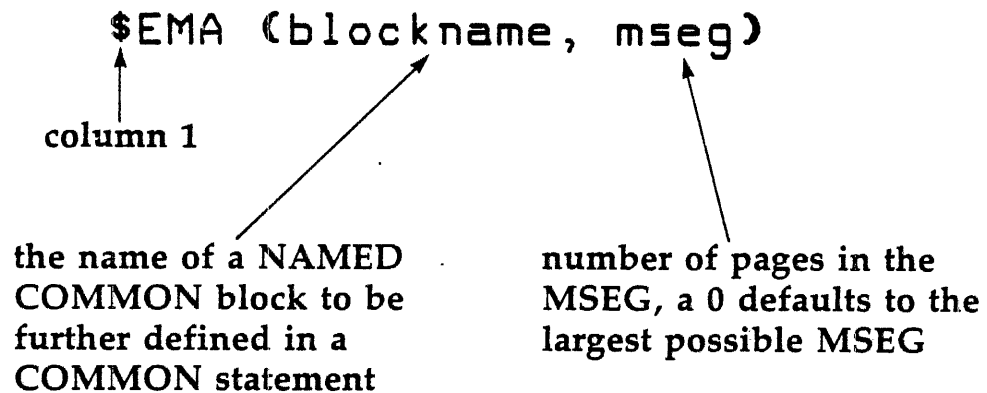
19B. USING EMA FROM FORTRAN

FORTRAN thinks of the EMA data area as a NAMED COMMON block. To use EMA in a FORTRAN program:

- **Use a COMMON statement to identify your variables in a NAMED COMMON block.**
- **Identify this NAMED COMMON block as the EMA data area by using the \$EMA statement.**
- **Manipulate the EMA variables just like any other variables in your program.**

DECLARING AN EMA DATA AREA

The FORTRAN \$EMA statement declares that the variables in the specified NAMED COMMON block are to be EMA variables.



The \$EMA statement must be the first non-comment statement in a program module.

EXAMPLE EMA DECLARATION

```
FTN4,L
$EMA(XYZ,0)
  PROGRAM EXMPL
  COMMON/XYZ/ IA(1000,100), IB(32767), REAL(10000)
  .
  .
  .
  .
  CALL SUBR (I,J)
  .
  .
  .
  END
$EMA(XYZ,0)
  SUBROUTINE SUBR (I,J)
  COMMON/XYZ/ IA(1000,100), IB(32767), REAL(10000)
  .
  .
  .
  .
  RETURN
  END
```

INPUTTING AND OUTPUTTING EMA VARIABLES



- Use FORTRAN READ or WRITE statements. The READ or WRITE can be formatted or unformatted.
- EMA variables CANNOT be used with FMP calls or EXEC calls.



EMA VARIABLES AND FORTRAN SUBPROGRAMS

A FORTRAN subprogram may be written to

- accept ordinary variables as parameters

```
                SUBROUTINE SUMAR (IARRAY,NELE,ISUM)
                DIMENSION IARRAY (NELE)
                ISUM=0
                DO 20 I=1,NELE
                   ISUM = ISUM+IARRAY(I)
20             CONTINUE
                RETURN
                END
```

- accept EMA variables as parameters

```
                SUBROUTINE SUMAR (IARRAY,NELE,ISUM)
                EMA IARRAY(NELE)
                ISUM=0
                DO 20 I=1, NELE
                   ISUM = ISUM+IARRAY(I)
20             CONTINUE
                RETURN
                END
```

PASSING EMA VARIABLES TO SUBPROGRAMS — CALL BY VALUE —

If a subprogram is written to accept ordinary variables as parameters, a calling program (or subprogram) can pass an EMA variable if the program uses “call by value” when invoking the subprogram.

```
FTN4,L
$EMA(XYZ,0)
  PROGRAM EXMPL
  COMMON/XYZ/ IA(1000,100), IB(32767), REAL(10000)
  .
  .
  .
  .
  CALL ADD( IB(15)+Ø, (IB(30000)), ISUM)
  .
  .
  .
  .
  END
  SUBROUTINE ADD (I,J,ISUM)
  ISUM=I+J
  RETURN
  END
```

PASSING EMA VARIABLES TO SUBPROGRAMS — CALL BY REFERENCE —

If a subprogram is written to accept EMA variables as parameters, the calling program (or subprogram) must pass EMA variables to the EMA parameters using “call by reference” when invoking the subprogram.

```
FTN4,L
$EMA(XYZ,Ø)
  PROGRAM EXMPL
  COMMON/XYZ/ IA(1000,100), IB(32767), REAL(10000)
  .
  .
  .
  .
  CALL ADD(IA(15), IB(30000), ISUM)
  .
  .
  .
  .
  END
  SUBROUTINE ADD (I,J,ISUM)
  EMA I,J
  ISUM=I+J
  RETURN
  END
```

19C. HOW EMA WORKS

The "load map" of an EMA program might be:

```
:RU,LOADR,,&TEMA
TEMA  32042 55552
SUM   55553 55631
```

```
FMTIO  55632 57125   24998-16002 REV.1901 781107
FMT.E  57126 57126   24998-16002 REV.1901 781107
PNAME  57127 57174   771121  24998-16001
REIO   57175 57321  92067-16268 REV.1903 790316
ERRO   57322 57411   771122  24998-16001
ERO.E  57412 57412   750701  24998-16001
FRMTR  57413 63037   24998-16002 REV.1901 781107
.CFER  63040 63115   750701  24998-16001
```

```
14 PAGES RELOCATED      32 PAGES REQ'D      18 PAGES EMA      3 PAGES MSEG
/LOADR:TEMA  READY AT  1:41 PM  SUN., 24  JUNE, 1979
/LOADR:$END
```

USING TRACKS FROM THE SYSTEM TRACK POOL

Programs using tracks from the system track pool need to —

- **allocate tracks**
EXEC 4 or 15
- **access the tracks**
EXEC 1 or 2
- **deallocate the tracks**
EXEC 5 or 16

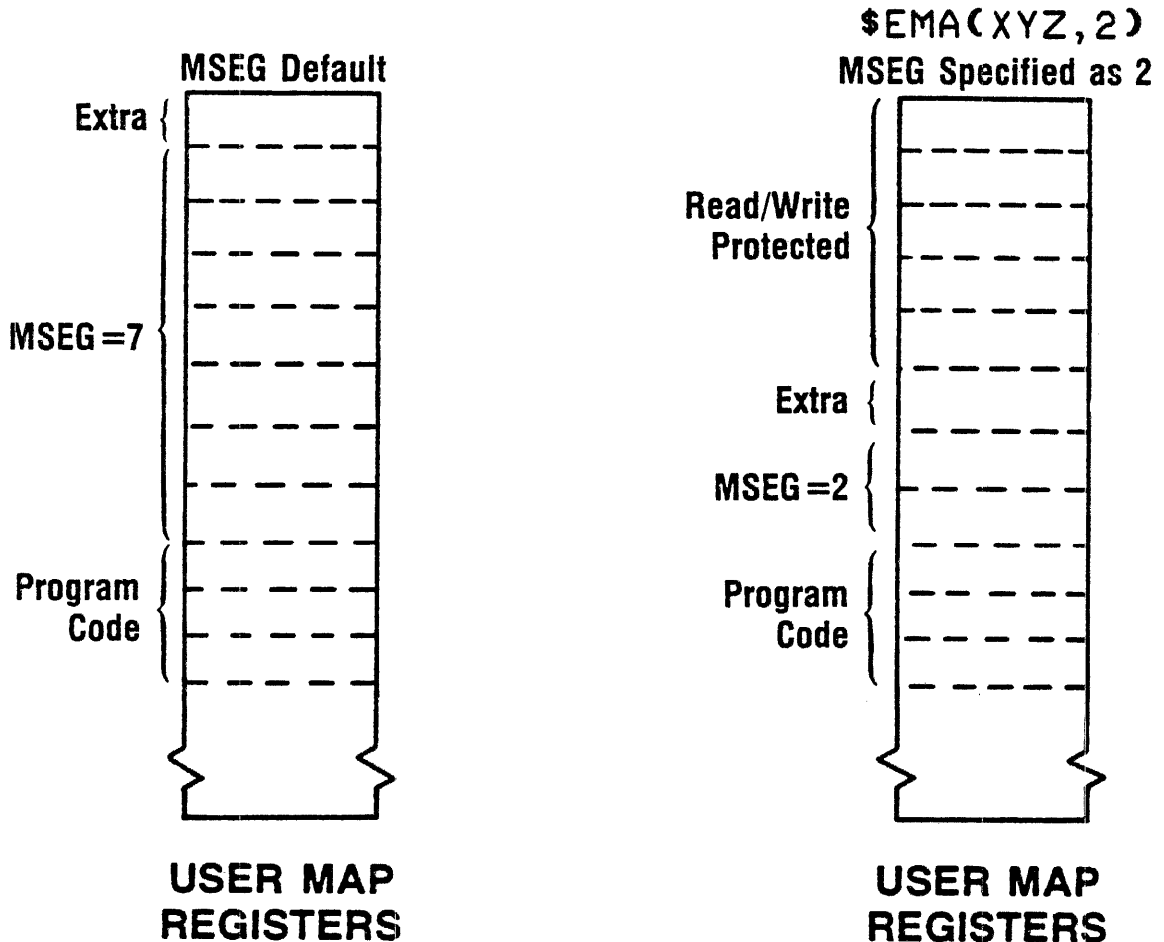
DEFAULTING MSEG

Largest MSEG possible is used.

All user map registers not pointing to program and system are used as MSEG.

Default MSEG size is set at load time; the MSEG size can be modified on-line with SZ command.

Default value = (32 - #mapped system pages - program size - 1).



CONSIDERATIONS OF MSEG SIZE

- * In an all FORTRAN program executing with EMA firmware, MSEG size makes absolutely no difference. The EMA firmware always maps two pages (one page containing the data and one extra page).
- * The Assembly Language programmer may want to specify MSEG size since subroutines .EMIO and MMAP use the MSEG size.

DISPATCHING EMA PROGRAMS

- RTE will dispatch an EMA program to any mother partition large enough to hold it. An EMA program can be dispatched to a subpartition if it is assigned to it.
- If a Mother partition is chosen, RTE will dispatch an EMA program when:
 - 1) All programs in the subpartitions are swappable.
 - 2) The EMA program has a higher priority than all the programs in the subpartitions.
- RTE then swaps all the programs out of the subpartitions. They will vie for other available partitions and subpartitions.
- RTE then loads the EMA program into the Mother partition where it gets CPU time just like any other program.

SWAPPING OUT AN EMA PROGRAM

- RTE will not swap out an EMA program that has been loaded into a Mother partition. This avoids thrashing.

EXCEPTION: RTE will swap out an EMA program in a Mother partition if a higher-priority program has been assigned to a subpartition within the Mother partition.

EMA vs. DISC

		DATA IN EMA		DATA ON DISC
		NON-TRANSPARENT (USER MAPPING)	TRANSPARENT (FORTRAN EMA)	NON-TRANSPARENT (USER SEGMENTS)
S P E E D	LOCAL ACCESS (SEQUENTIAL)	BEST	3rd BEST	2nd BEST
	RANDOM ACCESS	BEST	2nd BEST	TERRIBLE
EASE OF CODING		HARD	EASY	HARD

LOCAL ACCESS EXAMPLES: SOME LINEAR SORTS; ACCESSING SINGLE
ARRAY; FOURIER TRANSFORMS

RANDOM ACCESS EXAMPLES: QUICKSORTS; ACCESSING MULTIPLE
ARRAYS; HASHING

20 LIBRARIES



SECTION

A	SYSTEM LIBRARY	20-3
B	RELOCATABLE LIBRARY	20-14
C	DECIMAL STRING ARITHMETIC LIBRARY	20-17

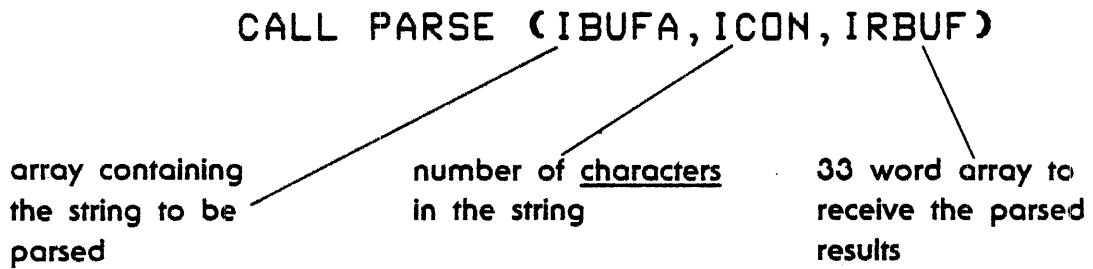
20A. SYSTEM LIBRARY

General purpose subprograms for using RTE services, including —

- **RMPAR** — retrieve parameters for a program
- **PRTN** — return parameters to a “waiting” Father
- **ABREG** — retrieve the contents of the A and B registers
- **IFBRK** — checks a program’s break bit
- **GETST** — retrieve parameter strings from interactive commands to schedule programs or buffers of data from programmatic schedule requests
- **REIO** — allows programs doing I/O to be swappable
- **RNRQ** — allows programs to use RTE resource numbers
- **LURQ** — allows programs to lock devices
- **AND MORE . . .**

PARSE

This routine parses an ASCII string containing up to 8 parameters, separated by commas.



PARSE returns the parsed string in blocks of 4 words, one per parameter.

		type of parameter parsed		
		null	numeric	ASCII
a 4 word block in IRBUF	word 1	0	1	2
	word 2	0	value	2 characters
	word 3	0	0	2 characters
	word 4	0	0	2 characters

IRBUF (33) contains the number of parameters parsed.

PARSE EXAMPLE

```
FTN4,L
PROGRAM PARSR
DIMENSION IBUFA(40),IRBUF(33)
C
C READ AN ASCII STRING INTO IBUFA
C
CALL EXEC(1,1+400B,IBUFA,-80)
CALL ABREG(IA,IB)
C
C AND PARSE IT
C
CALL PARSE(IBUFA,IB,IRBUF)
C
C DISPLAY RESULTS
C
WRITE(6,100)IBUFA
100 FORMAT(" ASCII STRING:  ",40A2)
WRITE(6,101)IRBUF
101 FORMAT("0 TYPE      VALUE      VALUE      VALUE"/
C      8(3X,12,4X,06,2X,06,2X,06/)
C      "0",12," PARAMETERS RECEIVED.")
END
```

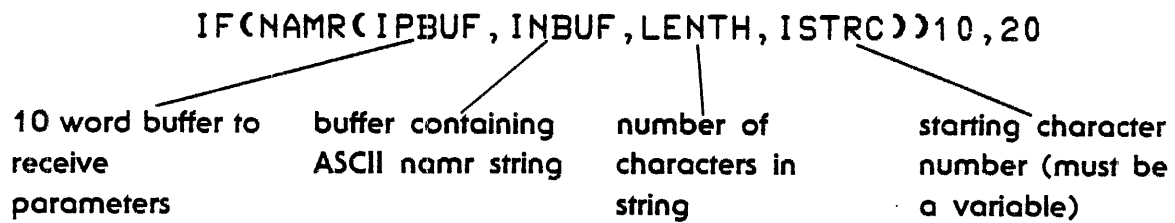
ASCII STRING: HELLO,14B,57,CARL,7

TYPE	VALUE	VALUE	VALUE
2	044105	046114	047440
1	000014	000000	000000
1	000071	000000	000000
2	041501	051114	020040
1	000007	000000	000000
0	000000	000000	000000
0	000000	000000	000000
0	000000	000000	000000

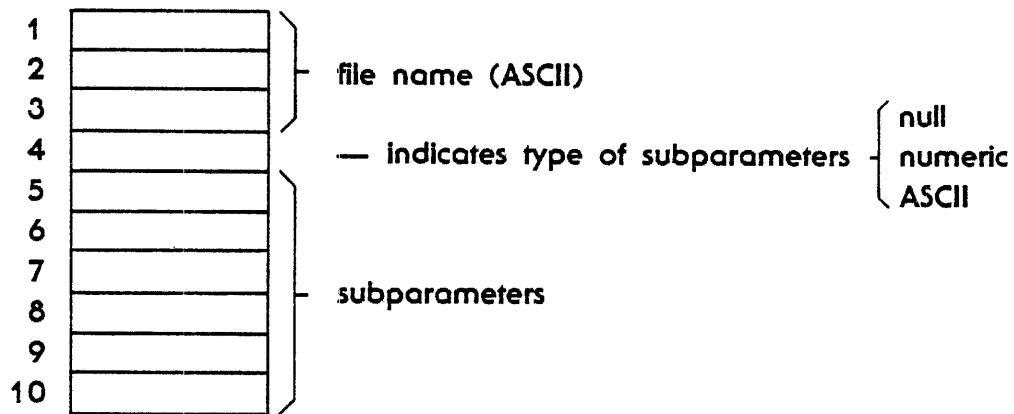
5 PARAMETERS RECEIVED.

NAMR

Parse a FMGR namr parameter string.



The returned buffer IPBUF is structured as:



The value of the function NAMR is returned as

- 0 if a string has been parsed
- 1 if no characters are in INBUF

NAMR EXAMPLE

```
FTN4,L
      PROGRAM N
      DIMENSION IPBUF(10),INBUF(40)
      LU=LOGLU(ISES)
5      WRITE(LU,100)
100    FORMAT("ENTER A NAMR.")
      CALL EXEC(1,LU+400B,INBUF,-80)
      CALL ABREG(IA,IB)
      ISTRC=1
      IF(NAMR(IPBUF,INBUF,IB,ISTRC))99,10
10     WRITE(LU,101)INBUF,IPBUF
      WRITE(6,101)INBUF,IPBUF
101    FORMAT(" INBUF NAMR : ",40A2/
C       /" I P B U F  PARAMETER BUFFER"/
C       " *****"/
C       /" WORD  1 = ",A2/
C       " WORD  2 = ",A2/
C       " WORD  3 = ",A2/
C       " WORD  4 = ",@6,"B"/
C       " WORD  5 = ",A2/
C       " WORD  6 = ",@6,"B"/
C       " WORD  7 = ",@6,"B"/
C       " WORD  8 = ",@6,"B"/
C       " WORD  9 = ",@6,"B"/
C       " WORD 10 = ",@6,"B")
      GO TO 5
99     STOP
      END
```

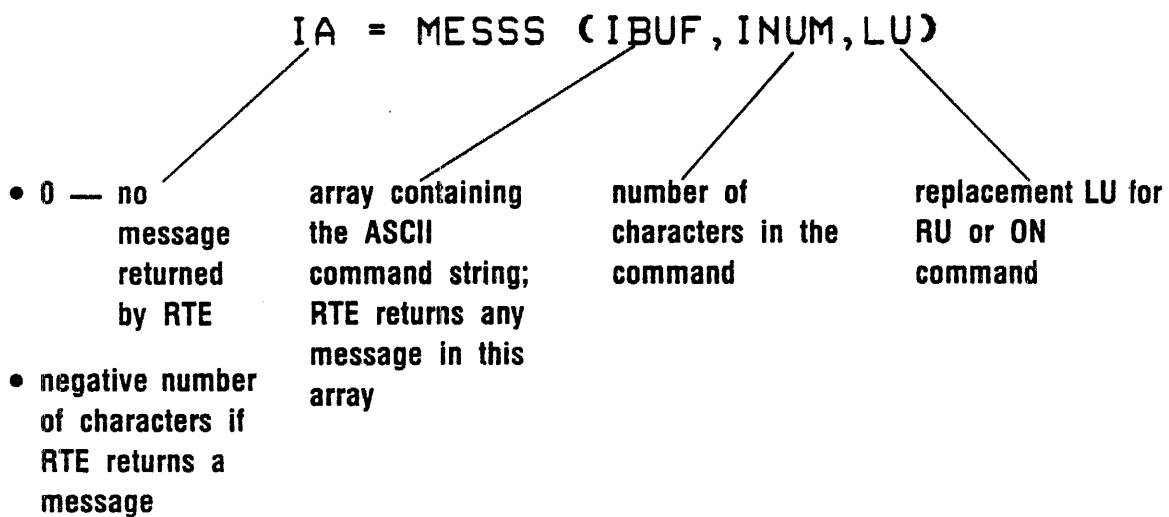
INBUF NAMR = NEWFIL:SC:12:3:10

I P B U F PARAMETER BUFFER

WORD 1 = NE
WORD 2 = WF
WORD 3 = IL
WORD 4 = 000537B
WORD 5 = SC
WORD 6 = 000014B
WORD 7 = 000003B
WORD 8 = 000012B
WORD 9 = 000000B
WORD 10 = 000000B

MESSS

Allows programs to issue interactive system commands.



IBUF should be at least 14 words long to allow for the largest possible system return.

EXAMPLE

Program LABIL calls MESSS to change its priority during execution.

```
FTN4,L
PROGRAM LABIL(3,99)
DIMENSION IPR50(20),IPR99(20)
DATA IPR50/2HPR,2H,L,2HAB,2HIL,2H,5,2H0 /
DATA IPR99/2HPR,2H,L,2HAB,2HIL,2H,9,2H9 /
C
C SET PRIORITY UP TO 50 FOR CRITICAL SECTION
C
    I = MESSS(IPR50,11)
    IF(I.NE.0)GO TO 99
    .
    . CRITICAL CODE
C
C SET PRIORITY BACK TO 99
C
    I = MESSS(IPR99,11)
    IF(I.NE.0)GO TO 999
    .
    .
    STOP
99 WRITE(1,100)
100 FORMAT("MESSAGE RETURNED WHEN SETTING PR TO 50:")
    CALL EXEC(2,1,IPR50,I)
    GO TO 1000
999 WRITE(1,101)
101 FORMAT("MESSAGE RETURNED WHEN SETTING PR TO 99:")
    CALL EXEC(2,1,IPR99,I)
1000 STOP
    .
    .
    .
```

ASCII EQUIVALENTS OF INTEGERS

integer → ASCII decimal

CALL CNUMD (IVAL, IBUF)

positive integer ———

3 word array to receive the 6 character ASCII representation, leading zeros suppressed

integer → ASCII octal

CALL CNUMD (IVAL, IBUF)

integer → ASCII decimal, only 2 digits

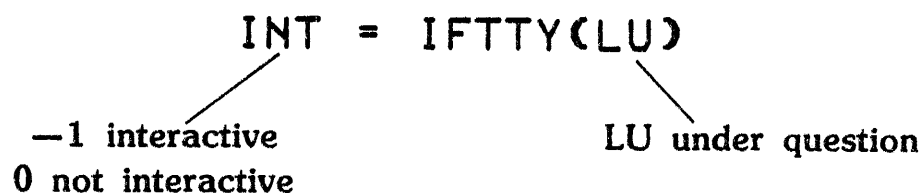
I = KCVT (IVAL)

ASCII decimal equivalent ———

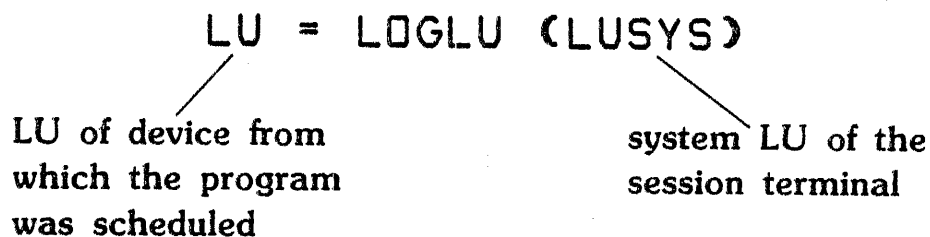
integer value, 0 → 99

CONCERNING LU'S

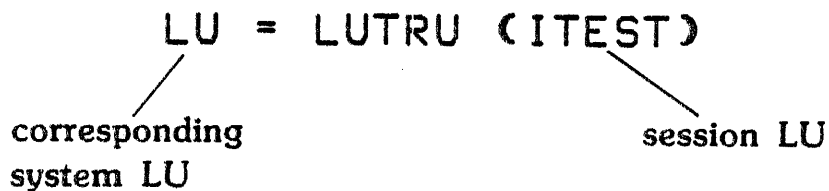
Is this LU an interactive device?



Who scheduled me?



What is the system LU?



FTIME

Supplies the system time in formatted mode.

CALL FTIME (IBUF)

a 15 word array where the formatted time is returned, for example

2:09 PM THU., 5 JAN., 1978

WHERE IS MY ID SEGMENT?

IDGET returns the address of a specified program's ID segment.

IDSEG = IDGET(INAM)

- address of ID segment
- 0 if program has no ID segment

3 word array containing a program name

AND MORE



- **Retrieve memory contents**

IMVAL = IGET (IADDR)

contents of memory address in memory

- **Set the S register**

CALL ISSR (N)

value to be set into the
S register

- **Retrieve a program's name (possibly renamed)**

CALL PNAME (NAME)

3 word array returned with
program's name

- **Check and then clear the overflow register**

IF (OVF(IDUMMY)) 10, 20

is set is clear

20B. RELOCATABLE LIBRARY

Math and general purpose utility routines, including,

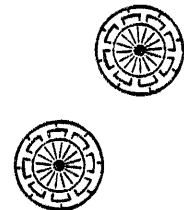
- .ENTR — retrieve parameters for a subprogram
- AND OTHERS

MATH ROUTINES

Square root	A = SQRT(X)		
Logs	A = ALOG(X)	A = ALOGT(X)	
Mods	A = AMOD(X, Y)	J = MOD(I, K)	
Trig functions	A = TAN(X)	A = SIN(X)	etc.
Complex values	A = CSQRT(X)	A = CEXP(X)	
Exponentials	A = EXP(X)		
Absolute values	A = ABS(X)	J = IABS(I)	

LOGICAL FUNCTIONS

And	K = IAND(I, J)
Or (Inclusive)	K = IOR(I, J)
Or (exclusive)	K = IXOR(I, J)



MANIPULATING TWO WORD INTEGERS

Callable from FORTRAN:

FIXDR converts a real value to a double length record number
FLTDR converts a double length record number to a real value

Callable from Assembler:

.DADS double integer addition/subtraction
.DMP multiplication
.DDI division
.DNG negation
.DCO comparison
.DIN increment
.DIS increment and skip if zero
.DDE decrement
.DDS decrement and skip if zero

.FIXD converts real to double integer
.XFXD extended real to double integer
.TFXD double real to double integer

.FLTD converts double integer to real
.XFTD to extended real
.TFTD to double real

20C. DECIMAL STRING ARITHMETIC LIBRARY

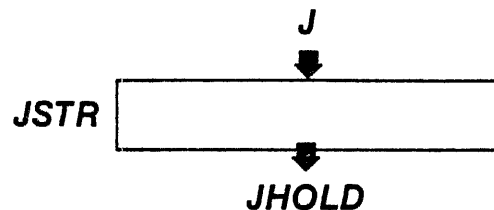
Routines for handling large character (decimal) strings.

- math functions
- string code conversions
- string editing

GETTING A CHARACTER FROM A STRING

SGET will extract a character from a string:

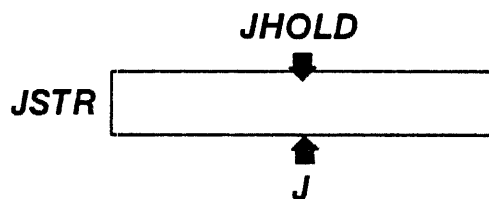
CALL SGET (JSTR, J, JHOLD)



PUTTING A CHARACTER INTO A STRING

SPUT will place a character into a string:

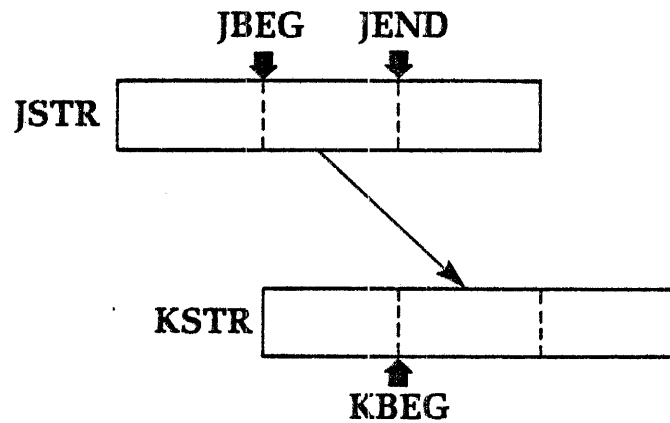
CALL SPUT (JSTR, J, JHOLD)



MOVING CHARACTERS BETWEEN STRINGS

Routine SMOVE moves a string of characters from one string array to another.

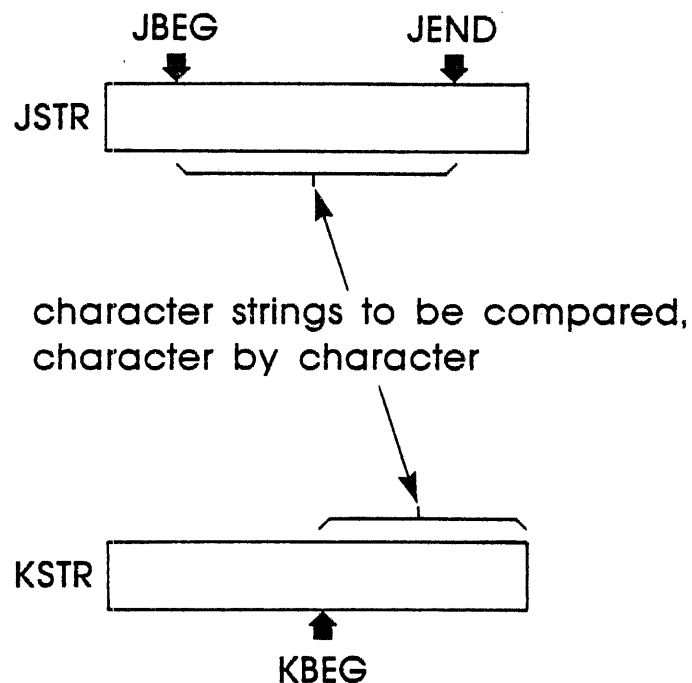
```
CALL SMOVE (JSTR,JBEG,JEND,KSTR,KBEG)
```



COMPARING CHARACTER STRINGS

JSCOM will compare two character strings according to the ASCII collating sequence.

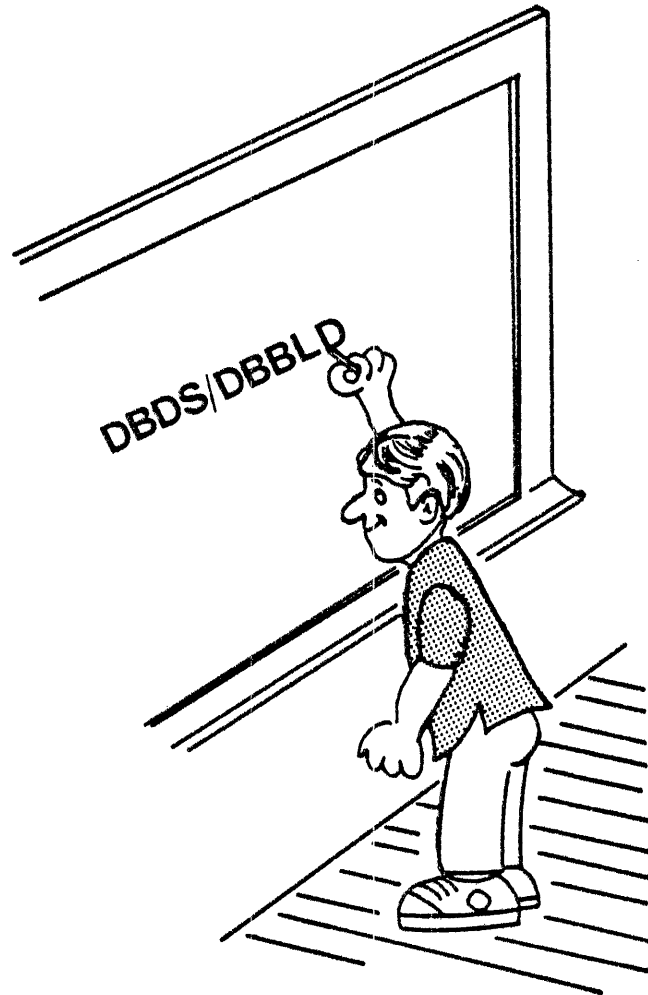
```
IRES = JSCOM (JSTR,JBEG,JEND,KSTR,KBEG,IERR)
```



JSCOM returns { <0 , string in JSTR $<$ string in KSTR
0, strings are equal
 >0 , string in JSTR $>$ string in KSTR

IERR indicates an invalid character in a character string.

APPENDIX A LAB EXERCISES



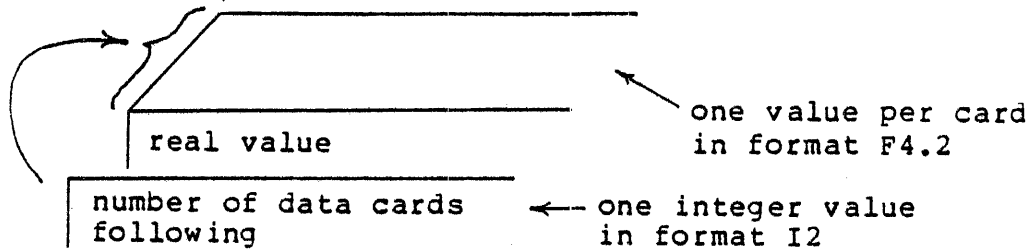
LAB 11 - DISC CARTRIDGES

1. If you "accidentally" dismount your private cartridge, how can you remount the cartridge without losing any data? Can you guarantee that this process will prevent the loss of any data?

If you try this exercise, you might want to back up your cartridge on tape first.

LAB 12 - SPOOLING

1. Using spooling, list several files to the line printer, making sure that no interleaved listing will occur.
2. File &LB122 contains a FORTRAN source program designed to input and process a data deck (cards) which has the following structure:



The program calculates and prints the average of the real values. The integer value acts as a "header" value, specifying how many real values are following. The program inputs from LU 5.

Create a disc file with these records:

```
b5
1154
1132
1167      ('b' represents a blank)
1159
1173
```

Make a copy of the FORTRAN program for your use (with a new program name) and then compile and load the program.

Using spooling, run the program to input and process the data in the disc file (without having to modify the program's code!).

3. Write a program which will

- write several records on a tape
(minicartridge or magnetic tape).
- rewind the tape
- read the records just output to the tape

Use ordinary FORTRAN READS and WRITES (and a REWIND call).

Create a type 4 file, 5 blocks in length. Repeat the exercise but use spooling so that the program you just wrote will manipulate the disc file.

4. Turn the line printer off line, then use spooling to list a file on the printer. What happens when you close the spool file? How do you recover?

LAB 13 - BATCH PROCESSING

1. Create and run a batch job which consists of

```
:JO,.....  
:CL  
:EO
```

Notice where the output is printed. Modify the job so that the output appears at your terminal.

LAB 14 - SYSTEM CONSOLE

1. Create and run a batch job which consists of

```
:JO,.....  
:IF,,EQ,,-1  
:EO
```

Submit the job and then abort it while

- * the System Console is in Non-session mode
- * the System Console is in Session mode

2. File &LB142 contains a FORTRAN source program which

- * prompts for an LU
- * prints a message on that device

Make a copy of the source file, giving the program a new name. Then compile and load the program.

Run the program from the System Console so that the message appears at your (peripheral) terminal. Try this with

- * the System Console in Non-session mode
- * the System Console in Session mode
(or is this possible?)

LAB 15 - TIME-SCHEDULED PROGRAMS

1. File &LB151 contains a FORTRAN source program which prints a message at your terminal. Make a copy of the source file, giving the program a new name and compile and load the program.

Using the system IT and ON commands, time schedule the program to execute

- every 5 seconds, starting immediately.
- every 5 seconds, starting 20 seconds from the time you schedule it. (Or is this possible to do?)

2. Write a program which will time schedule the program you used in problem 1 to execute every 5 seconds, starting 5 seconds from the time of the scheduling request.
3. Modify the program used in problem 1 so that it will place itself in the time list and print out the message
 - * every 5 seconds, starting 5 seconds from the time when the program is initially run.
 - * every 5 seconds, starting 20 seconds from the time when the program is initially run.

You might include an EXEC time request in your program to be sure that the message is being printed at the correct intervals.

4. Place program WHZAT into the time list to execute every 10 seconds. What happens when another user enters a WH command?

LAB 16 - PROGRAMS SCHEDULING OTHER PROGRAMS

1. Write two programs which will be a Father and a Son. The Father should prompt the operator for a string of characters and an LU (where the string will eventually be printed) and then schedule the Son program. The Son should print the string of characters on the device specified by the operator.

2. Files &LB161 and &LB162 contain FORTRAN source programs.

- * Program LB161 writes a message and suspends itself

- * Program LB162 tries to schedule program LB161 "queue with wait."

Make a copy of each source file and then compile and load the programs.

- a) Follow these steps:

- use the FMGR RU command to run program LB162
- from breakmode, run WHZAT to see what is happening
- from breakmode, run program LB162 again.

Explain what happens next.

- b) Reschedule program LB161 and then try these steps:

- run program LB162 from FMGR
- from breakmode, run WHZAT
- from breakmode, restart your FMGR
- from FMGR, run program LB162 again
- from breakmode, run WHZAT.

Explain the WHZAT display.

LAB 17 - CLASS I/O

* PART A - PROGRAM TO PROGRAM COMMUNICATION

1. Write two programs (a Father and a Son). The Father will
 - prompt the user to enter a string of characters
 - place the string in a "mailbox"
 - schedule the Son to retrieve and print the string.
 - a) Schedule the Son "without wait" and let the Son deallocate the class number.
 - b) Schedule the Son "with wait." When the son completes, the Father should deallocate the class number.

2. Write two programs (a Father and a Son). The Father should
 - go into a loop, prompting the user for a string of characters and placing the string in a "mailbox"
 - terminate the loop when the string "XX" is entered, then
 - schedule the Son.

The Son should then retrieve and print the strings, terminating after all the strings have been retrieved and printed.

3. Modify the programs you wrote for problem 2 so that the Son is scheduled after the first string of characters is entered, rather than waiting until an "XX" (the last string) is entered.

4. In problem 3, if the Son "hangs up" for some reason (the printer is down perhaps), the Father could end up having many buffers in SAM, all waiting to be retrieved by the Son.

Arrange your programs so that there will be a maximum of 4 buffers in SAM at any one time.

(Perhaps use a second class number and let the Son tell the Father when a buffer is consumed. This way, the Father can keep a running count of the number of buffers in SAM.)

* PART B - DEVICE I/O AND CONTROL

5. Write a program which prompts you for a string of characters. Let the program repeatedly print a message on the line printer until you respond with your input.
6. Write a program which goes into a loop, inputting strings of characters from your terminal (CLASS I/Q) and outputting them to the line printer. Terminate the loop when the string "XX" is entered.

Modify the program to accept input from several terminals, until an "XX" is entered at each one. When the string is output to the line printer, also identify the terminal which input the string.

LAB 18 - MORE RTE SERVICES

1. Write a program which insures itself exclusive access to the line printer by using an LU Lock. Have the program PAUSE before unlocking the printer. While the program is suspended, restart your FMGR and try to list a file on the printer.
2. Write two programs which will compete for use of the line printer.

Program 1 should write the message "I'M PROGRAM 1" 25 times on the printer.

Program 2 should write the message "I'M PROGRAM 2" 25 times on the printer.

Program 1 should schedule program 2 without wait before starting to print its messages. This way, the two programs will be competing for the same resource.

Consider these questions:

- * If the line printer is BUFFERED, what will the output look like?
- * If the line printer is UNBUFFERED, what will the output look like?

Modify the programs to use a Resource Number so that the output will alternate between 5 lines from program 1 and 5 lines from program 2.

Some suggestions:

- use an unbuffered line printer.
- after a program unlocks the RN, have the program output a message to the line printer. (Perhaps print "PROGRAM x'S TURN OVER").
- which program should deallocate the RN?

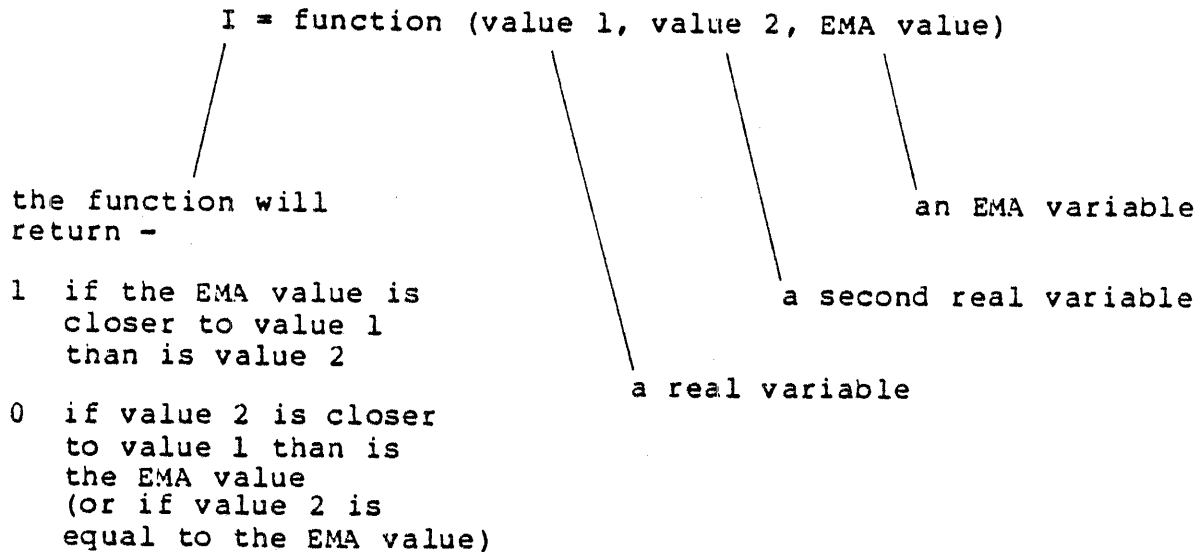
3. File &LB183 contains a FORTRAN source program which will
 - request a local allocation of 5 tracks
 - print out the location of the tracks allocated
 - PAUSE
 - when rescheduled, deallocate the tracks and terminate.
 - a) Make a copy of the source file, giving the program a new name. Compile, load and run the program. While the program is suspended, run LGTAT and identify the allocated tracks. Reschedule the program and run LGTAT after the program terminates.
 - b) Run the program again. While it is suspended, abort the program (OF,program,1). Then run LGTAT.
 - c) Remove the EXEC track deallocation call and repeat part a.

4. For the system you are using, how many words (locations) are gained by making a program "large-background"?

LAB 19 - EXTENDED MEMORY AREA (EMA)

1. RNDFIL is a type 1 disc file containing 16384 real values. Write a program which will -
 - * input the values and store them in EMA.
 - * compute and output the average of the EMA values.
2. Modify the program you wrote for problem 1 to determine the average value and the value closest to the average value for the values in EMA. Write two subprograms as follows to implement this modification.

Subprogram 1: Write a FUNCTION subprogram to be invoked as follows -



Subprogram 2: Write a SUBROUTINE subprogram which uses the above FUNCTION subprogram to determine which EMA value is closest to the average.

The SUBROUTINE is to be invoked by -

CALL subroutine (average, index)

the subroutine is passed the average of the real values

the subroutine is to return the index of the value closest to the average

LAB 20 - LIBRARIES

1. Write a program which will accept a 5 character program name and "rename" it for the user's Session. Print out the new name.
2. Write a program which places itself into the time list, waking up every second to "up the line printer."
3. Write a program which looks at its ID Segment and prints out its type: real-time, background or large-background.

Can your program be a large-background program?

4. Write a program to flash the display register in a clever pattern.

* PART B *

5. Is "random access" using POSNT really slower than "true random access" using LOCF and APOSN (not counting set up time) for a sequential (type 3 and above) file?

File TLB105 contains a source program which you can use to test out this idea. You should write two programs:

Program 1. This program should use POSNT to randomly access the file. The program should go into a loop, prompting for a record number and then displaying the contents of that record. Terminate the program when a record number of 0 is entered.

Program 2. This program should operate similarly to the first program except that it should access the file using LOCF and APOSN for "true random access."

Notice any speed difference?

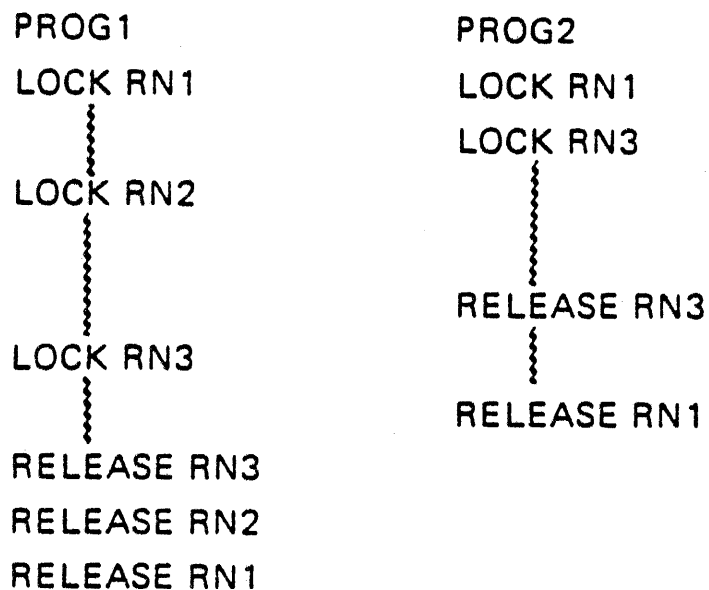
SECTION

A	SYSTEM LIBRARY	20-3
B	RELOCATABLE LIBRARY	20-14
C	DECIMAL STRING ARITHMETIC LIBRARY	20-17

AVOIDING DEADLY EMBRACE

1. All programs lock RN's in same order.

Example:



2. Logically associate the resources to be protected with one RN instead of several RN's.

RESTORING CARTRIDGES

Users can restore cartridges from magnetic tape back to private or group disc cartridges with the READT utility.

```
:RU,READT [ , cartridge [ , mag tape lu [ , PG [ , size ] ] ] ] ]
```

↑
positive crn
negative LU of disc cartridge

- if a CRN is specified and the cartridge is not already mounted, READT will mount a cartridge from the spare cartridge pool and restore it from tape.
- if a disc LU is specified, the specified cartridge is restored.

ALLOCATING A CLASS NUMBER

You request a class number by making an EXEC 20 call with ICLAS having a value of 0.

```
.  
.  
LU = 0  
ICLAS = 0  
CALL EXEC (20,LU,IAR1,NWD1,IPA,IPB,ICLAS)
```

↑
RTE will return the class number allocated to you

- You can then use ICLAS to tell another program what class number to use when retrieving data sent to it.
- You can also use ICLAS to do additional CLASS WRITE/READS on the already allocated class number.

```
.  
.  
CALL EXEC (20,LU,IAR2,NWD2,IPC,IPD,ICLAS)
```

↑
RTE uses a previously allocated class number

DISPATCHING EMA PROGRAMS

- RTE will dispatch an EMA program to any mother partition large enough to hold it. An EMA program can be dispatched to a subpartition if it is assigned to it.
- If a Mother partition is chosen, RTE will dispatch an EMA program when:
 - 1) All programs in the subpartitions are swappable.
 - 2) The EMA program has a higher priority than all the programs in the subpartitions.
- RTE then swaps all the programs out of the subpartitions. They will vie for other available partitions and subpartitions.
- RTE then loads the EMA program into the Mother partition where it gets CPU time just like any other program.



MANUFACTURERS and CONSUMERS

EXEC 20 (CLASS WRITE/READ)

A program initiates a CLASS I/O program to program data transfer by making an EXEC 20 call. The CLASS WRITE/READ call will "manufacture" a buffer in SAM and fill it with data from the calling program.

EXEC 21 (CLASS GET)

The receiving program retrieves the data in SAM by making an EXEC 21 call. The CLASS GET will "consume" the buffer in SAM by storing the data in the program and releasing the SAM buffer for use by other programs.

*Every call that "manufactures" a buffer in SAM must have a corresponding call that "consumes" the buffer. CLASS I/O is said to be "double call."

*Programs can execute independently of the data transfers, which are handled by RTE.

READER COMMENT SHEET

Manual Name: _____
(Please Print)

Part Number: _____

We welcome your evaluation of this publication. Your comments and suggestions will help us improve our training materials. Please use additional pages if necessary.

Is this book technically accurate?

Did it meet your expectations?

Was it complete?

Is it easy to read and use?

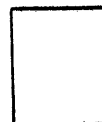
Other comments?

FROM:

Name _____

Company _____

Address _____



**Training Coordinator/Technical Marketing
Hewlett-Packard Co.
11000 Wolfe Road
Cupertino, California 95014**