HEWLETT PACKARD

RTE-A PROGRAMMER & SYSTEM MANAGER

VOLUME I

data systems training center

STUDENT WORKBOOK

22999-90546

FEB. 1984

INTRODUCTION TO RTE-A

CHAPTER 1

Table of Contents

Chapter 1
INTRODUCTION TO RTE-A

MODULE OBJECTIVES

1. Learn the special aspects of a real-time computer system.

2. Know how to log on and use simple commands through the Command Interpreter.

3. Learn the relationship between directories, subdirectories and files.

4. Understand the additional features offered by the VC+ enhancement to the RTE-A operating system.

5. Be familar with the hardware components on the system.

SELF-EVALUATION QUESTIONS

1-1. If you were given a "generic" computer system, what would you look for to determine it's usefulness as a real-time system?

1-2. Classify the following as either: (I) Hardware
(II) Operating System
(III) Utility Program

a. Pascal compiler
b. Load A-Register (machine instruction)
c. EDIT (text editor)
d. Terminal device driver
e. Time base generator (generates 10ms interrupts)
f. System clock
g. EXEC(9) (subroutine to schedule a program)
h. WH command
i. FmpOpen (file system subroutine)
j. Select code on an HP-IB I/O card

1-3. What happens when you type in the following command:

```
CI> co myfile yourfile
```

followed by 4 backarrow (<==) keys and a carriage return?

1-4. What is the difference between:

```
CI> wh
```

and:

```
CI> ru wh
```

1-5. What is the effect of the following command?

```
CI> /////
```

1-6. When you type in your user name and password, the LOGON program creates a session for you then runs a program that was specified by the System Manager when he set up your account (this is normally CI). What would happen if he had specified WH as the program to run when you logged on?

1-7. Explain the three features of the spooling system. What are the advantages to spooling?

1-8. What is the difference between an LU and a select code? How can you find your terminal LU? ...your terminal select code?

1-9. What is the size of physical memory in the system you're using? How does this differ from logical memory?

1-10. What is the ID segment list? How can you create more than one ID segment for a program? Name two ways in which ID segments are purged.

1.1 What is an Operating System?

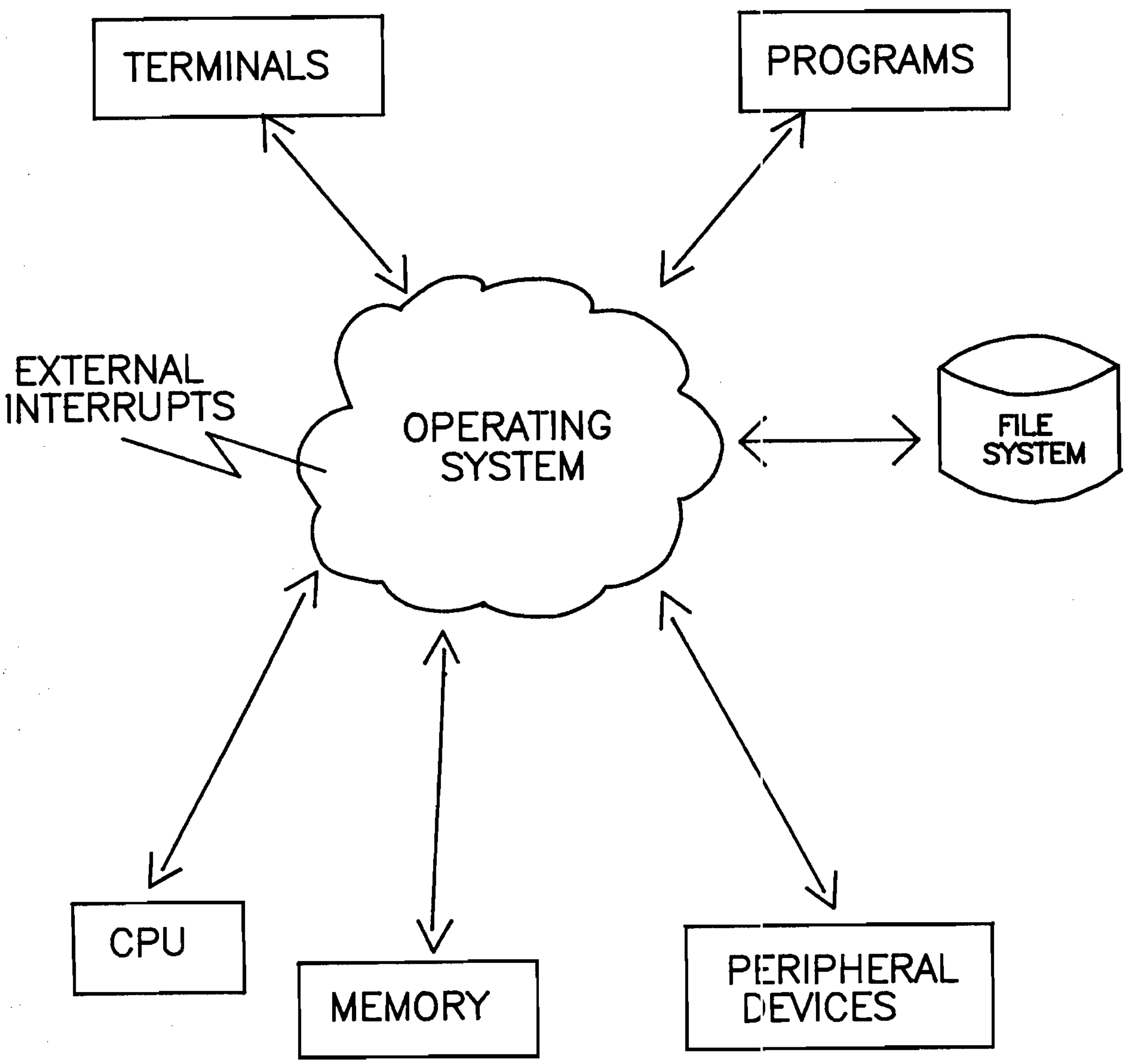
Interface to users -- interactive users (terminals) and programs are isolated from the low-level details of the hardware and I/O communication.

Controls resources -- the CPU, memory, and peripheral devices must be shared by users and programs. The operating system enforces a policy that determines who gets what, when and how much.

Interrupt handling -- the operating system must respond to interrupts in a timely manner. Interrupts may be caused by a user striking a key, a signal from a disc drive after completing an I/O operation, a pressure sensor detecting a critical condition, etc.

File system -- the operating system provides a simple method for accessing disc files and removes the user from the physical aspects of the various disc devices.

What is an Operating System?



system Resources

1.2 HP 1000 -- A Real-Time System

Interrupts -- External events interrupt the CPU. The operating system handles the interrupt (e.g., ignore it, schedule a program, execute a task immediately) before returning to the interrupted program. The primary feature of a real-time system is the fast response to these interrupts.

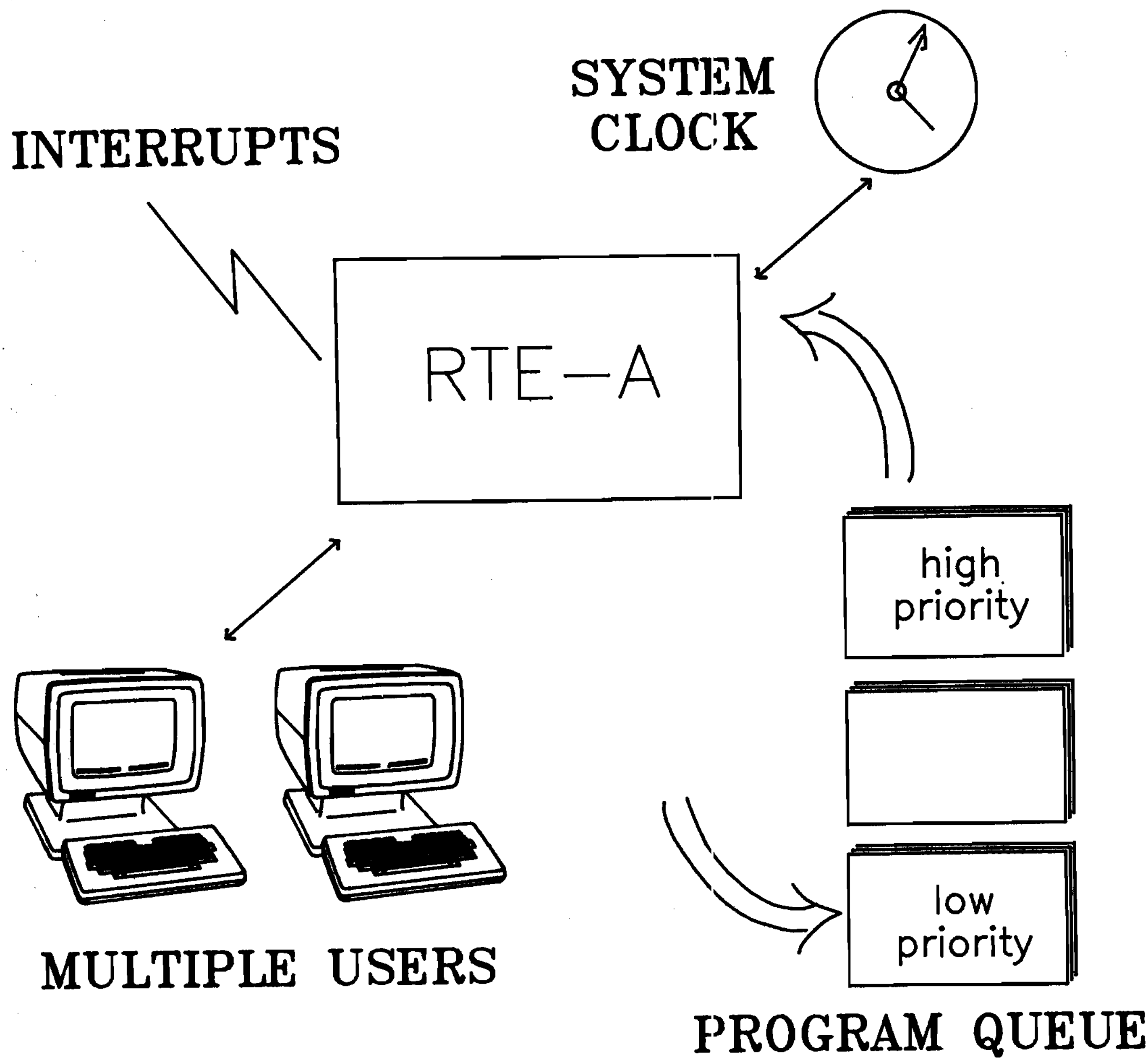
System clock -- Maintains the time of day and date. A time base generator (TBG) generates an interrupt every 10 milliseconds from which the system increments the clock. In a real-time system programs may be scheduled to run at a specific time with reference to the system clock.

Program queue -- Programs are run in order of priority. If the top program must wait (e.g., for I/O), other programs can run concurrently. Programs with equal priority may take turns using the CPU which is known as "time-slicing".

Multiple Users -- Each user interacts with a separate program. These programs are typically time-sliced to give each user the impression of having a personal computer.

HP/1000

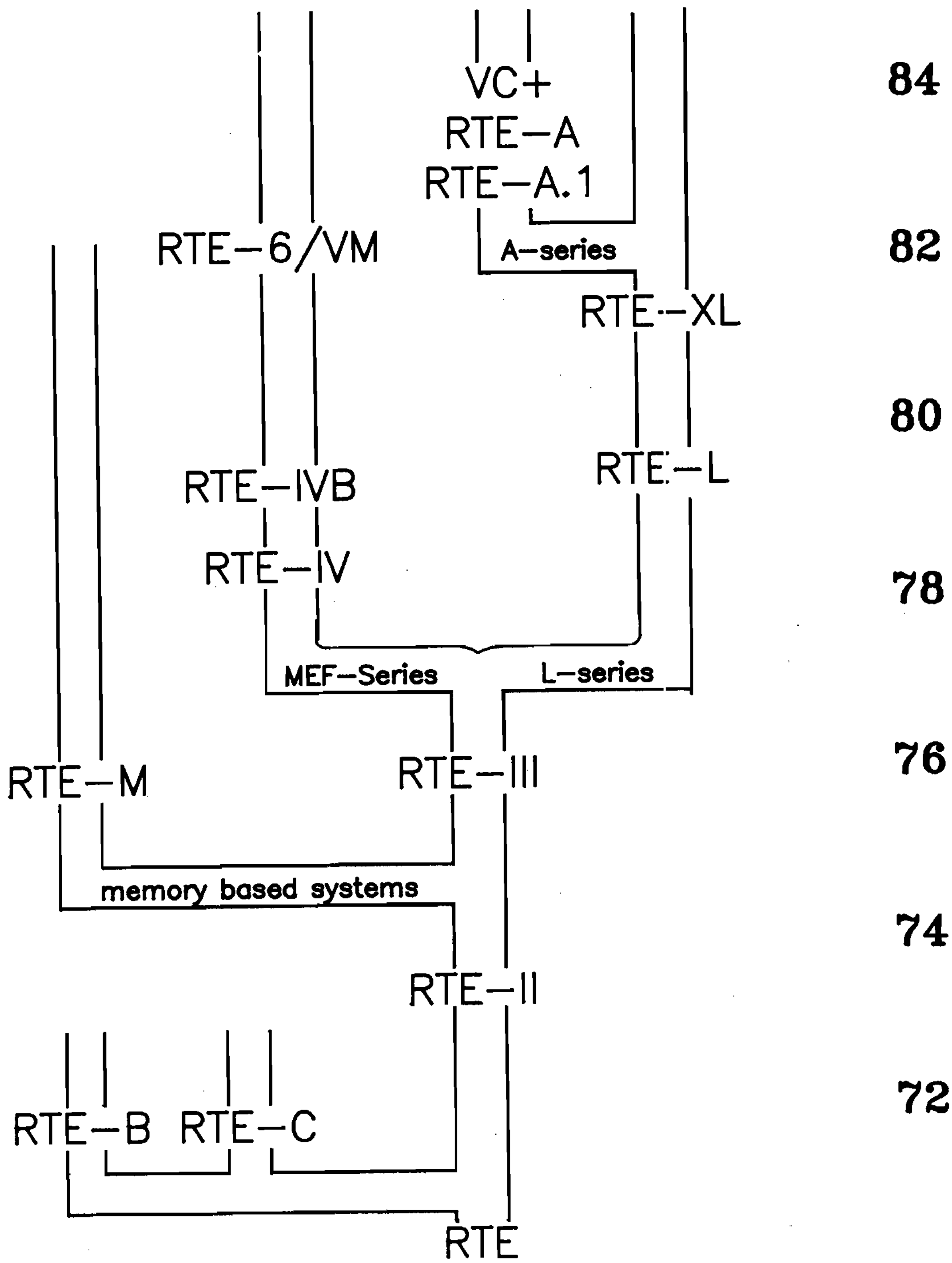
A REAL-TIME SYSTEM



1.3 RTE Origins

```
RTE        -- No file system
              2 memory partitions
              64 kb memory
RTE-B      -- BASIC command interpreter
RTE-C      -- Memory based RTE
RTE-II     -- FMGR file system
              I/O spooling
              Command files
RTE-III    -- Multiple partitions
              Multiple users
              2 mb memory
RTE-M      -- Memory based
              2 mb memory

------------------------------------------------

RTE-IV     -- EMA
              Class I/O
              FORTRAN IV
RTE-IVB    -- Session monitor
              Full screen editor
              Pascal
RTE-6/VM   -- VMA
              MLS/LOC
              Macro assembler

------------------------------------------------

RTE-L      -- Disc or memory based
              Fast system generation
              Fast I/O
              64 kb memory
RTE-XL     -- Multi-tasking
              512 kb memory
RTE-A.1    -- EMA
              VMA
              32 mb memory
RTE-A      -- Command Interpreter
              Hierarchical files
RTE-A/VC+     Code and data separation
              Auto-segmentation
              Multi-user sessions
              Out-spooling
```

RTE ORIGINS


VC+
RTE–A
RTE–A.1
RTE–6/VM
A–series
RTE–XL
RTE–IVB
RTE–L
RTE–IV
MEF–Series
L–series
RTE–M
RTE–III
memory based systems
RTE–II
RTE–B
RTE–C
RTE
84
82
80
78
76
74
72

1.4 RTE-A System

Operating system -- Manages resources such as memory, cpu, I/O devices, etc. Utilizes time base generator to provide date, time, and scheduling.

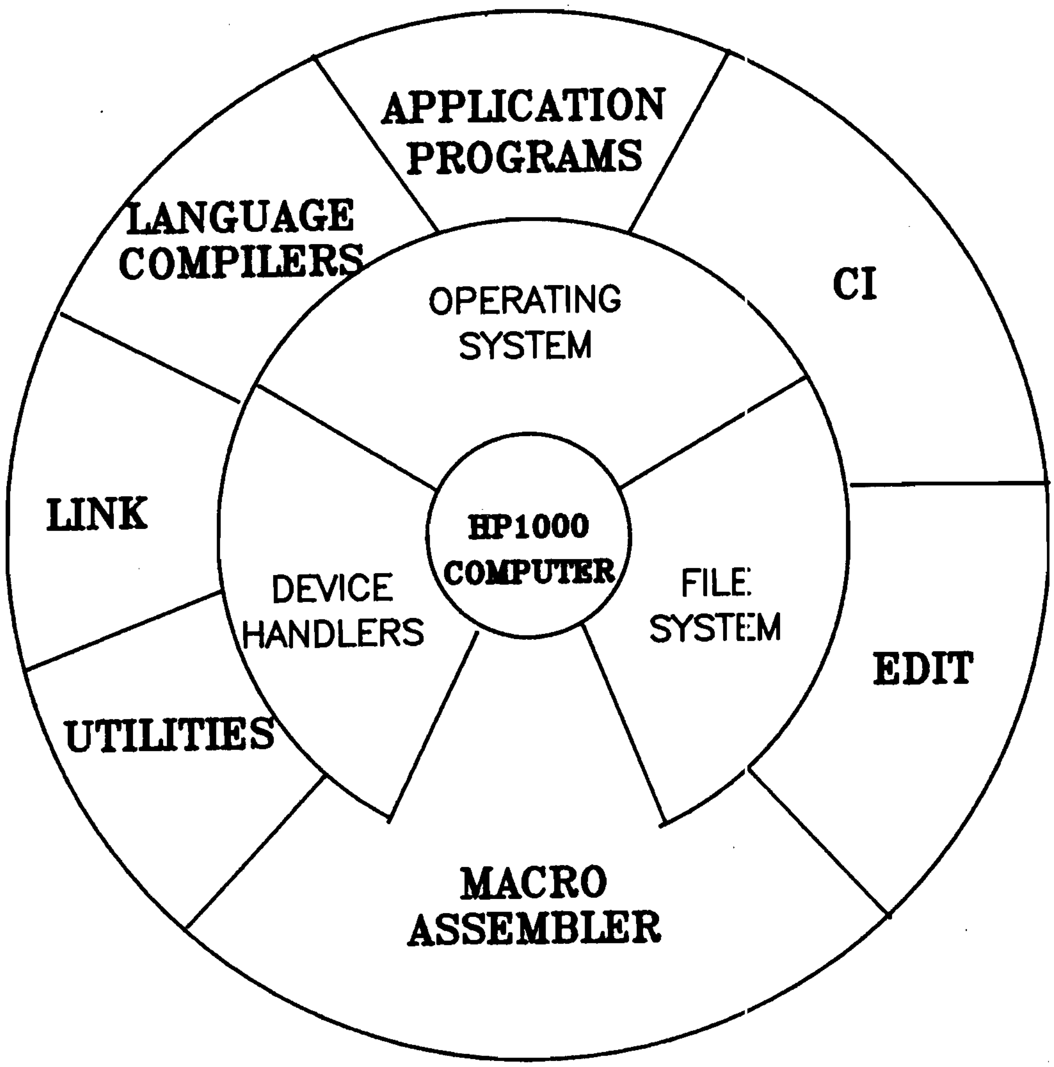
Device Handlers -- Provides an easy method for communicating with different I/O devices.

File system -- Provides convenient access to files on disc devices.

CI -- Command interpreter is the user level interface to the facilities of the RTE-A operating system.

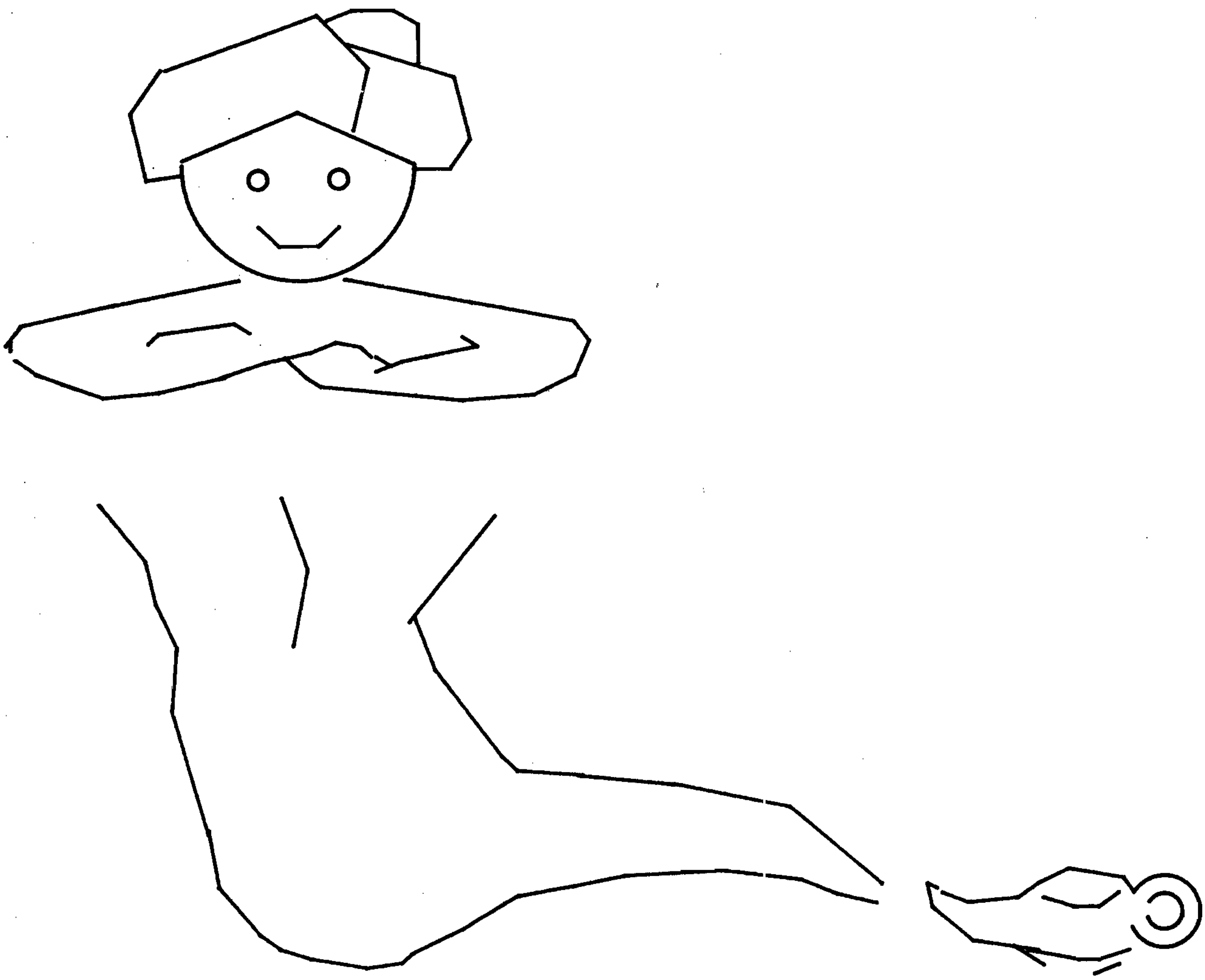
All other programs -- May share the hardware and operating system facilities via access through RTE-A.

RTE-A SYSTEM


APPLICATION PROGRAMS
LANGUAGE COMPILERS
CI
OPERATING SYSTEM
LINK
HP1000 COMPUTER
DEVICE HANDLERS
FILE SYSTEM
EDIT
UTILITIES
MACRO ASSEMBLER

1.5 THE COMMAND INTERPRETER

The Command Interpreter



1.6 Using the 2621 Terminal

Command Entry:

RETURN -- All commands are terminated with this key.

BACK SPACE -- Used to correct commands before hitting RETURN.

Note: Do not use the left arrow key in place of the backspace key.

DEL -- Deletes the entire command entered so far.

Screen Editing:

ROLL SCREEN -- Scrolls thru display memory; cursor does not move.

MOVE CURSOR -- Moves cursor around screen display; cursor wraps around.

HOME CURSOR -- Displays first page of display memory with cursor at the top left.

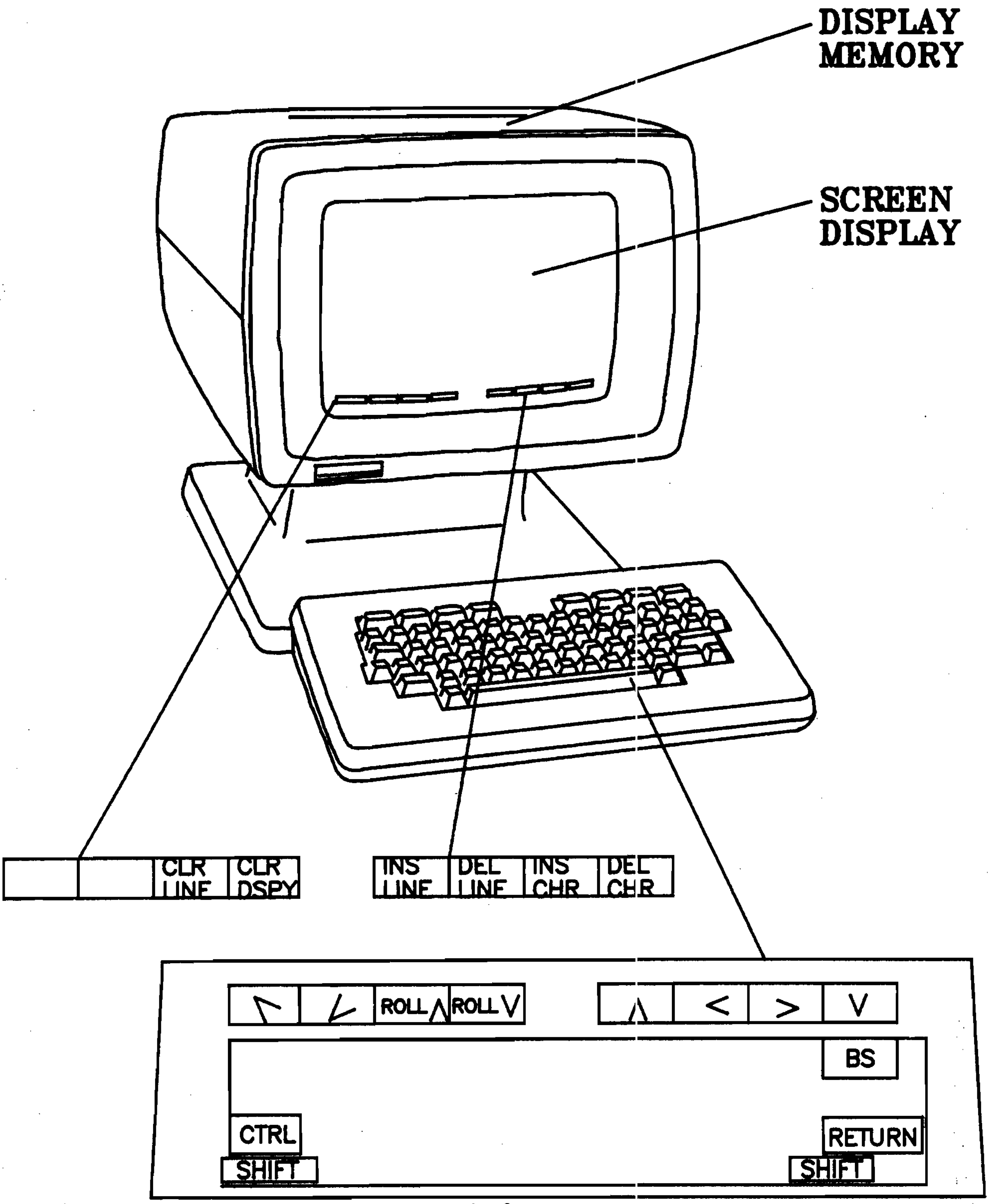
CLEAR LINE -- Clears line from cursor to the right.

CLEAR DISPLAY -- Clears display memory from cursor position down.

INSERT/DELETE KEYS -- Used for editing characters and lines.

References: Terminal User's Manual

USING THE 2621 TERMINAL



R1.6

1.7 Using the 2622 Terminal

Command Entry:

RETURN -- All commands are terminated with this key.

BACK SPACE -- Used to correct commands before hitting RETURN.

Note: Do not use the left arrow key in place of the back space key.

DEL -- Deletes the entire command entered so far.

Screen Editing:

ROLL SCREEN -- Scrolls thru display memory; cursor does not move.

MOVE CURSOR -- Moves cursor around screen display; cursor wraps around.

HOME CURSOR -- Displays first page of display memory with cursor at the top left.

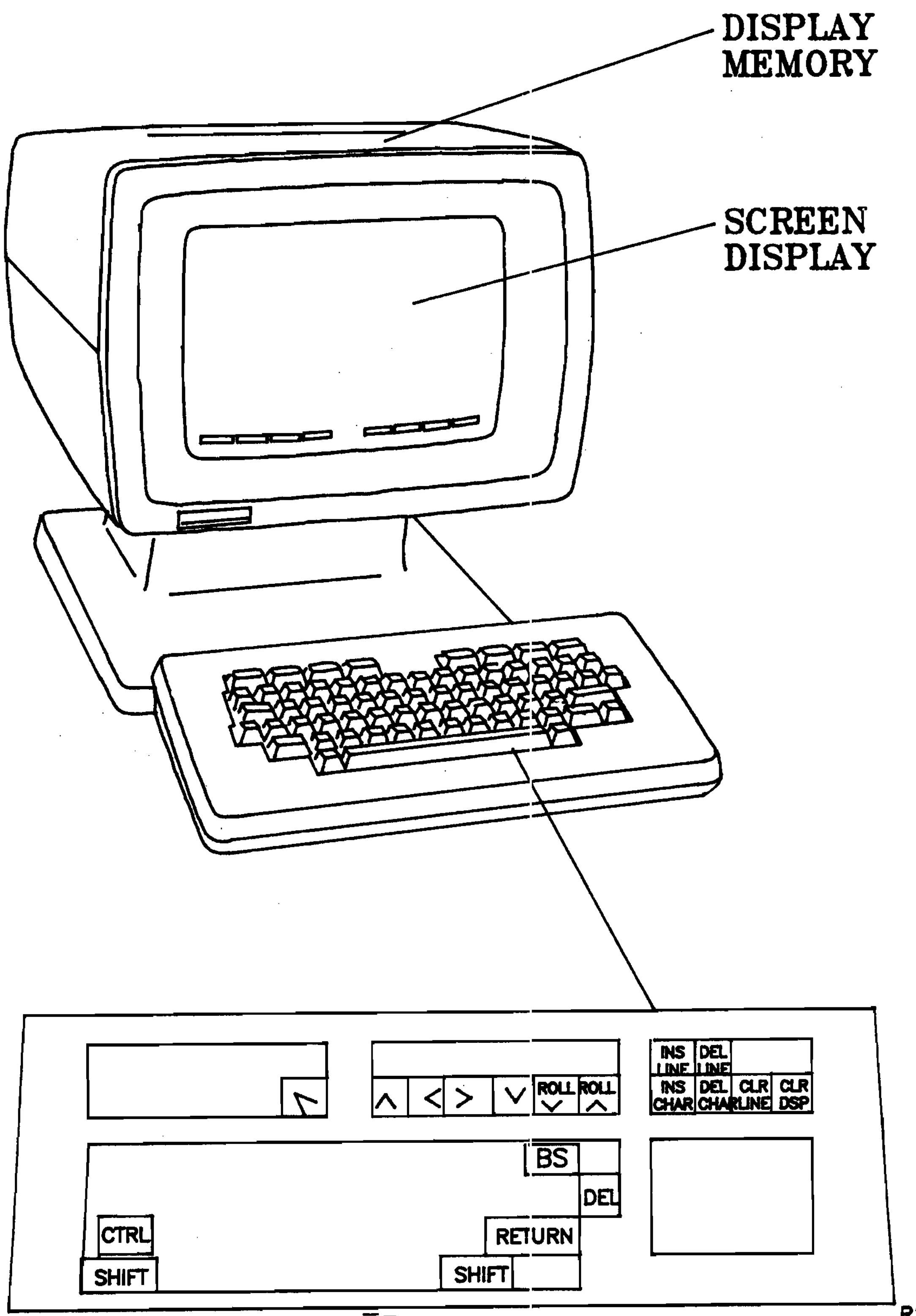
CLEAR LINE -- Clears line from cursor to the right.

CLEAR DISPLAY -- Clears display memory from cursor position down.

INSERT/DELETE KEYS -- Used for editing characters and lines.

References: Terminal User's Manual

USING THE 2622 TERMINAL



R1.6A

1.8 CI Basics

Commands can only be entered after the prompt. The command is terminated with a carriage return.

Commands may be upper or lower case (all are converted to upper case internally).

Delimiters may either be spaces or a comma (spaces are converted to a comma internally).

References: User's Manual

CI BASICS

```
CI>_
CI> tm
Sun Jan 1, 1984 12:01:23 am

CI> LI MyFile.txt
This sure is easy using CI to
list the contents of my file.

.
.
.
More...('A' to abort)


CI> co,myfile.txt,MiscComments.txt
Copying MYFILE.TXT to MISCCOMMENTS.TXT... [ok ]

CI> ?
```

help facility

1.9 Running Programs

All of the examples shown are equivalent.

Implied Run -- If the command cannot be found, CI puts "RU" in front of the command line and tries again. The "RU" is only explicitely needed if the program name matches that of a CI command. In this case "print" is not a CI command so the "RU" is not needed to run the PRINT program.

Program Parameters=-- Any parameters following the program name are passed to the program, in this case PRINT, to be dealt with by the program. Sometimes parameters can be defaulted. In this case the output device defaults to "6", and therefore does not need to be specified.

References: User's Manual

RUNNING PROGRAMS

```
CI> ru print lab1.txt 6
Print job supervised by PRIN1
CI> print lab1.txt 6
Print job supervised by PRIN1
CI> print, lab1.txt
Print job supervised by PRIN1
CI> _
```

1.10 Command Stack

Use cursor control and local edit keys to modify a command in the stack. RETURN executes the command.

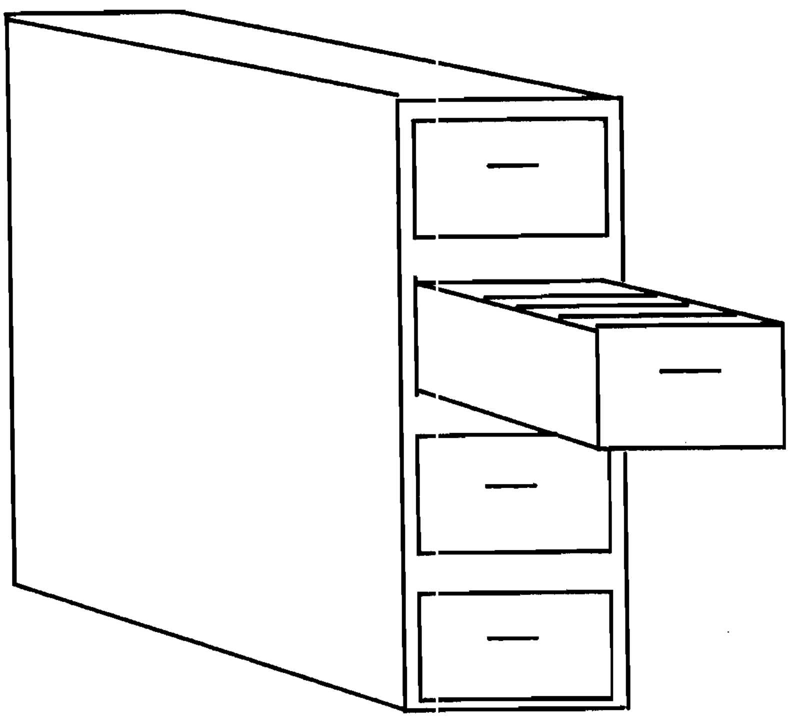
References: User's Manual

COMMAND STACK

```
CI> /
---Commands---
tm
LI MyFile.txt
co,myfile.txt,MiscComments.txt
ru print lab1.txt 6
print lab1.txt 6
print,lab1.txt
_
```

1.11 THE FILE SYSTEM

The File System



1.12 Files and Records

(blank)

References: User's Manual

FILES AND RECORDS

A file is a collection of related pieces of information:

* names and addresses of all employess
* a Pascal source program
* a binary memory image of a runnable program

A Record is an individual piece of information in the file:

* the name and address of one employee
* a single Pascal statement
* a standard size "chunk" of a memory-image file (eg. 128 bytes)

RECORD 1	RECORD 2	...	RECORD n	EOF

A file which might reside on disc or mag tape

1.13 The Disc File

Disc file -- recorded on one or more tracks of the disc volume.

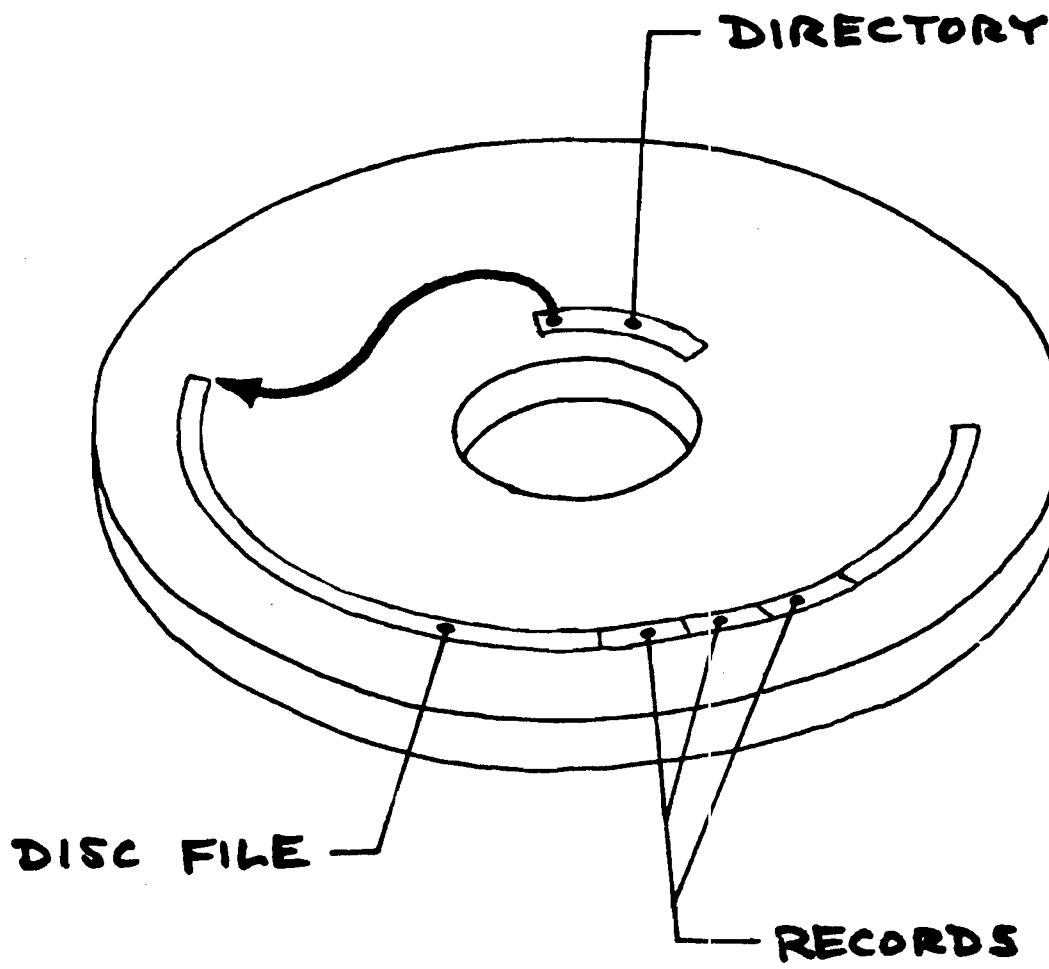
Records -- basic units of the disc file.

Directory -- a disc file with the following special properties:

1. Found at a known location on the disc.
2. Contains the names and addresses of other files on the disc.

References: User's Manual

THE DISC FILE



1.14 Directories

Directories -- Are files that contain information about other files such as file name, size, and its location on disc. Directories are extendable; that is they can reference any number of files (limited only by the space on the disc volume).

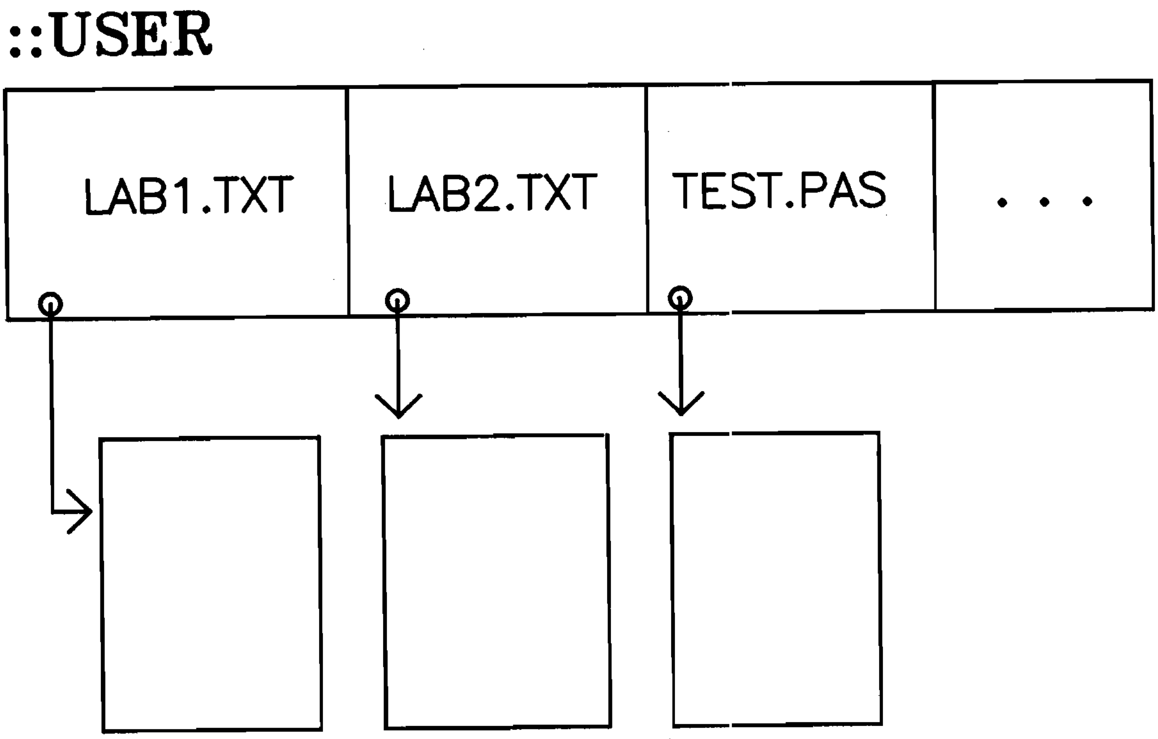
Directory Names -- May be found in one of two formats as shown here. /user/ refers to the same directory as ::user and either may be displayed by RTE-A in its messages.

DL command -- is used to list the contents of a directory. Note that the command shown could also have been specified:

```
CI> dl ::users
```

References: User's Manual

DIRECTORIES




```
CI> dl /user/
directory   ::USER
LAB1.TXT   LAB2.TXT   TEST.PAS

CI> _
```

1.15 Hierarchical Structure

Global Directories -- are at the top of the hierarchy and are referenced as /user/ or ::user

Sub-directories -- may nest to any level and are referenced as /user/tests/ and /user/tests/more/

Files -- can be found at any level and are referenced as /user/doc.txt and /user/tests/more/t8.ftn

ADVANTAGES

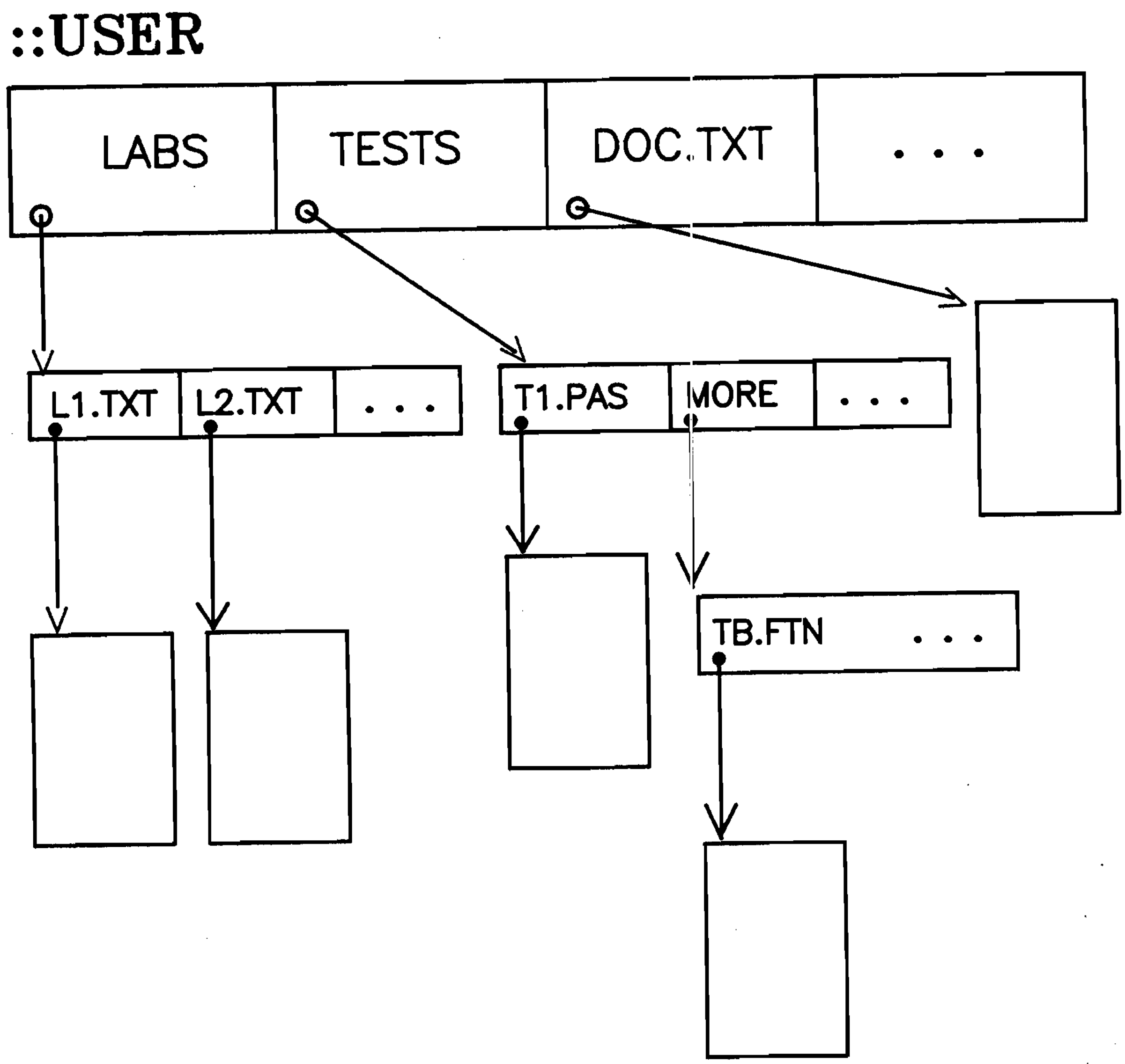
1. Files can be catagorized to any hierarchical level to keep the number of files in each directory to a manageable number.

2. Unrelated files can be kept logically separate.

3. Disc space for files within a directory is allocated dynamically from the space on a volume. Unused space can be given to any directory that needs it. (In the previous FMGR file system, each directory was given a fixed amount of disc space that could not be used by any other directory.)

References: User's Manual

HIERARCHICAL STRUCTURE



1.16 Program Files vs. Programs

Skeleton ID Segment -- A byte-for-byte copy of the ID segment that will be put into the system partition. It contains information such as the program name, priority, segmentation information, and space that will be used by the system to store temporary information about the program. In addition, the skeleton ID segment contains disc related information such as entry points and checksums for the program file.

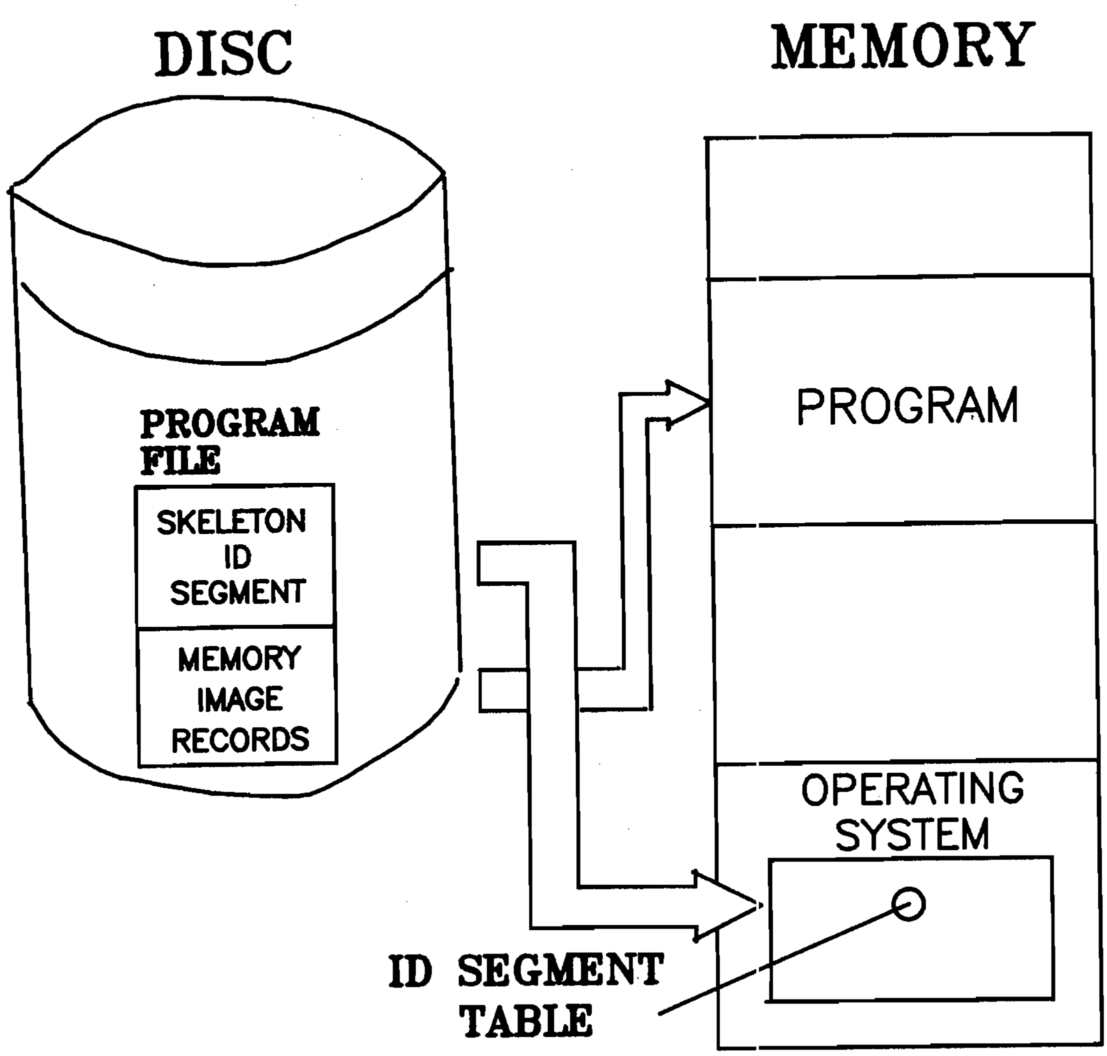
ID Segment Table -- Contains space for as many program's ID segments as may be running concurrently. This number is determined at system generation time. Each ID segment is used to keep track of the current status and location of a program running in the system.

RP Command -- (Restore program) Is used to create an ID segment for the program file. Program names can be only five characters long. The default program name is the first five characters of the program file name. The ID segment remains until explicitly removed.

RU Command -- Begins execution of the program. If no ID segment exists for the program, an implicit RP will be performed before executing the program and the ID segment will be removed upon termination.

References: User's Manual

PROGRAM FILES
VS.
PROGRAMS



CI> rp program_file name
CI> ru name

1.17 P E R I P H E R A L D E V I C E S

PERIPHERAL DEVICES

1.18 Addressing Peripheral Devices

Logical Unit Number -- Or "LU" tells the device drivers within RTE-A with which interface card and device to communicate. The actual LU of each device is specified during system generation.

Device Drivers -- Part of the operating system software that contains the protocol required to communicate with a particular device.

Terminals -- Reference to LU 1, whether interactively or programmatically, always refers to the user's terminal. The actual LU of the terminal may be found by use of the WH command.

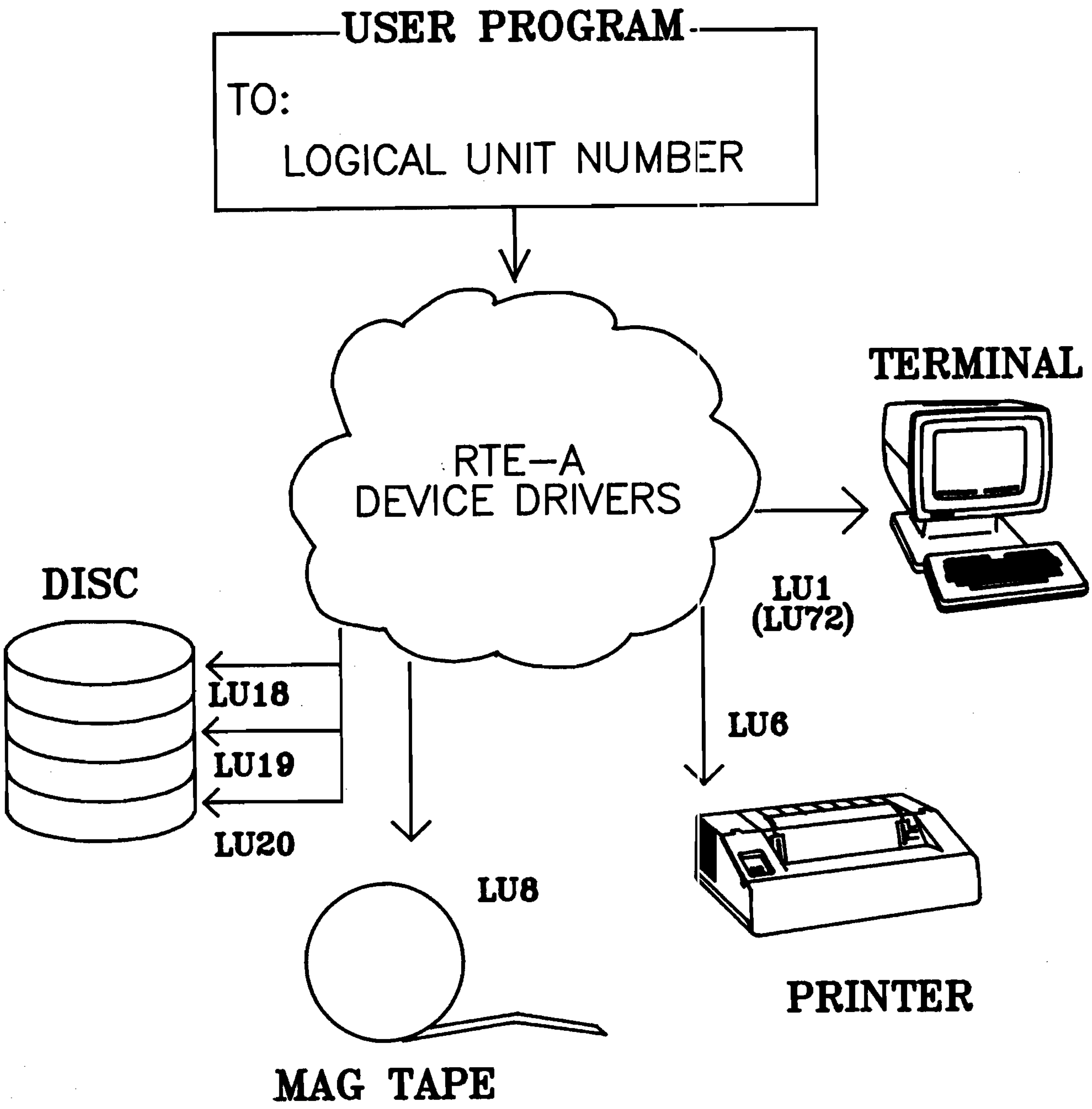
Disc -- Each physical disc may be broken into separate logical LUs of various sizes. Each disc LU is then treated as an independent unit.

Printer -- Usually LU 6. Many utilities that use the printer default to LU 6.

Mag tape -- Usually LU 8. Cartridge tape drives (integral to CS/80 disc units) are usually LU 24.

References: User's Manual

ADDRESSING PERIPHERAL DEVICES



1.19 LUs and Select Codes

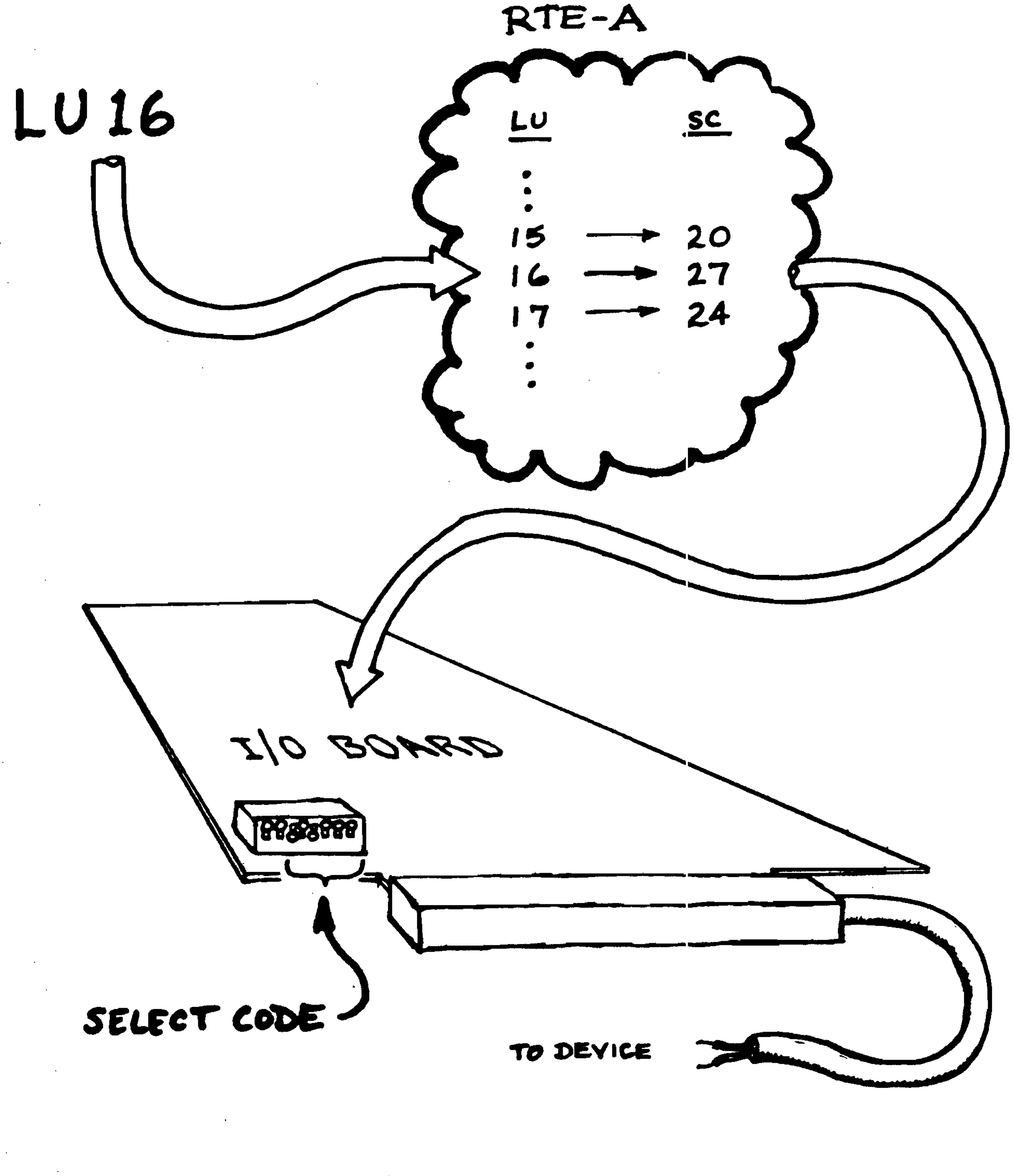
LU -- Logical Unit number. The user or program always refers to a device by it's LU.

LU Table -- Translates the LU reference to a select code.

Select Code -- Is physically set on each I/O board to assign it a physical address. The I/O board is connected directly to a device.

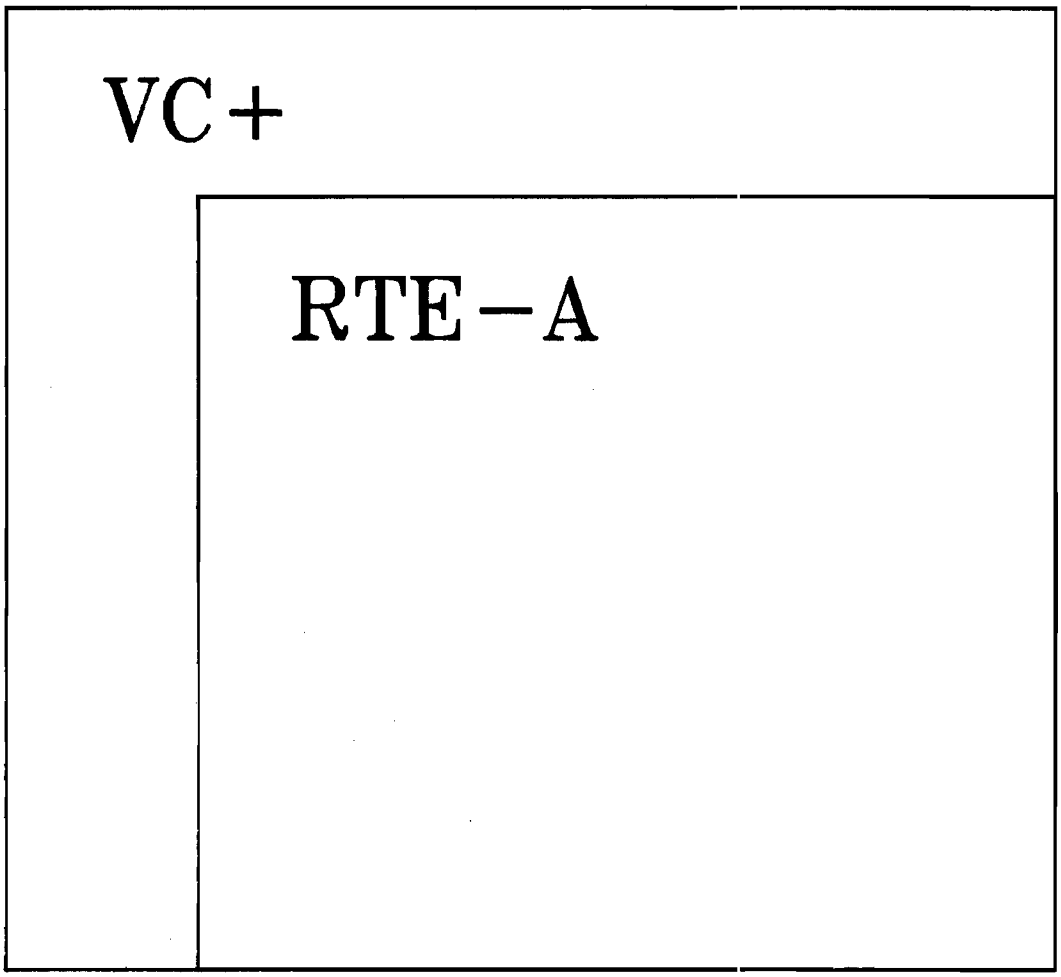
References: System Installation Manual, System Generation Manual

LUs AND SELECT CODES



1.20 V I R T U A L C O D E +

VIRTUAL CODE +



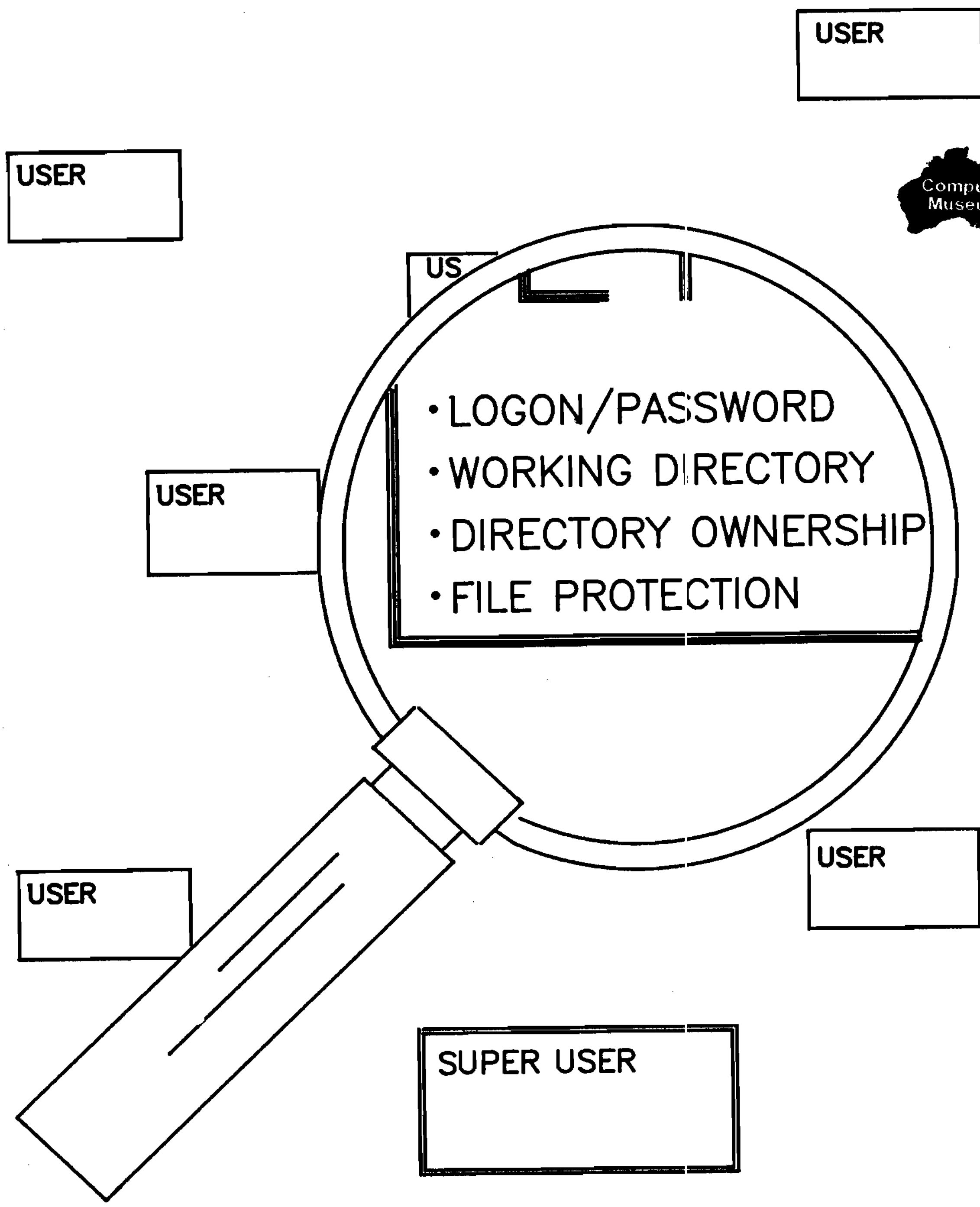
1.21 Multi-user Environment

User -- Each user is assigned a session. The session is maintained through a system table to provide logical separation between users.

Super-user -- Is assigned a session also. The super-user has the capability to create user accounts, modify system programs, set the system clock, initialize disc volumes, and override the file protection of a general user.

References: User's Manual

MULTI–USER ENVIRONMENT (VC+)



1.22 CDS -- Shareable Programs VC+ only

CDS -- Code and Data Separation.

Physical Memory -- Contains two logically separate data partitions but only one copy of the program code.

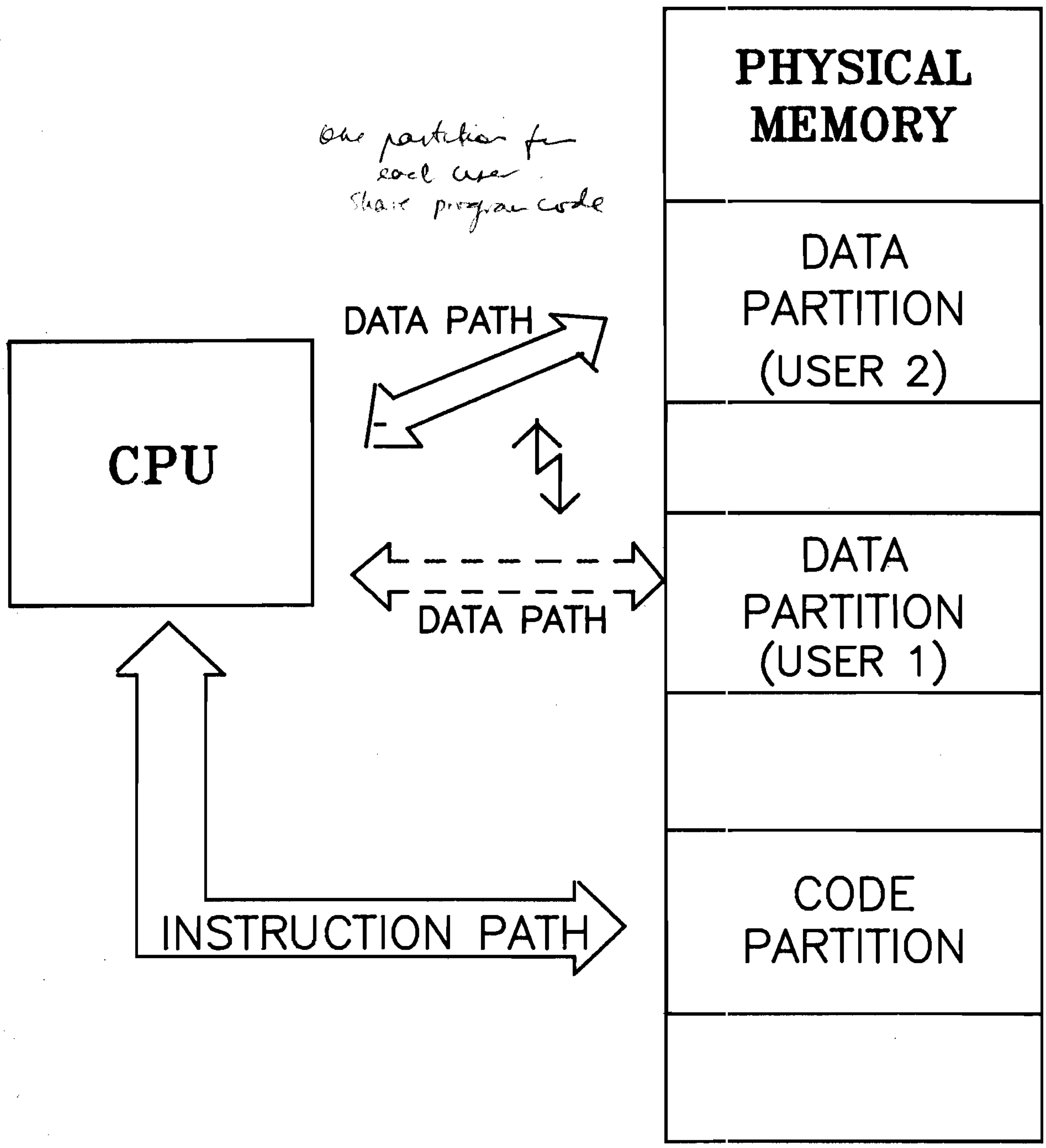
Data Path -- Can be rapidly switched from one data partition to another using a dynamic mapping system.

CPU -- The CPU "sees" only the data relevent to the current user of the program.

Instruction Path -- Is the same for both users of the program.

References: User's Manual, System Design Manual

CDS (VC+)
SHAREABLE PROGRAMS



1.23 CDS -- Transparent Segmentation

CDS -- code and data separation.

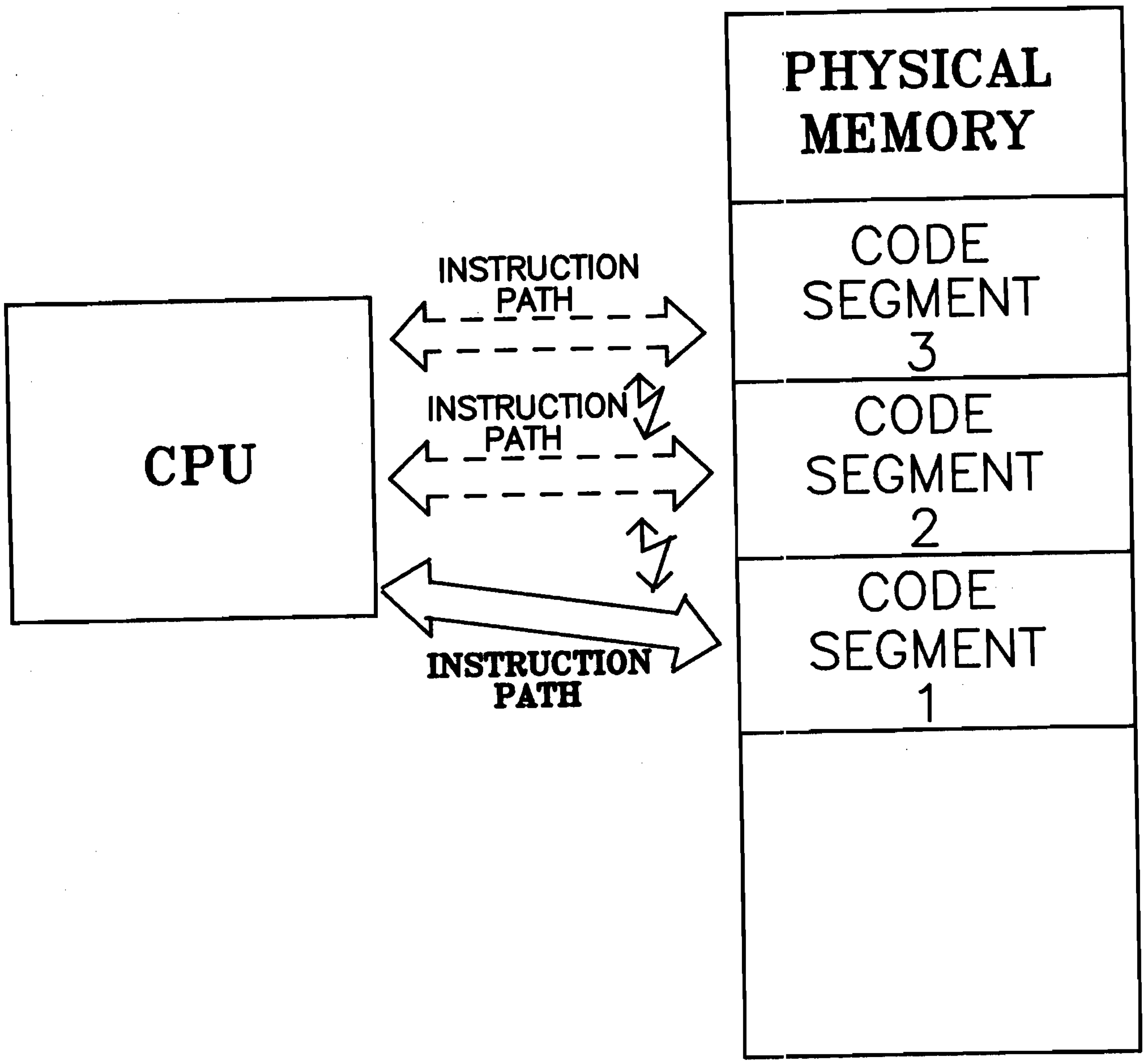
Physical Memory -- Contains logically dependent code segments. Each segment contains one or more subprograms. Up to 128 code segments allowed.

CPU -- The CPU "sees" just one code segment at a time which is entirely within the 15-bit address space of the CPU.

Instruction Path -- Can be rapidly switched from one code segment to another using a dynamic mapping system.

References: Link User's Manual, System Design Manual

CDS TRANSPARENT SEGMENTATION



1.24 Spooling System (VC+)

Out-spooling -- Data goes to a spooling file until released to printer (or other device).

LU Redirection -- All data sent to an LU (device) is redirected to another device.

Error-Logging -- logon, logoff and system-wide errors are sent to an error log file.

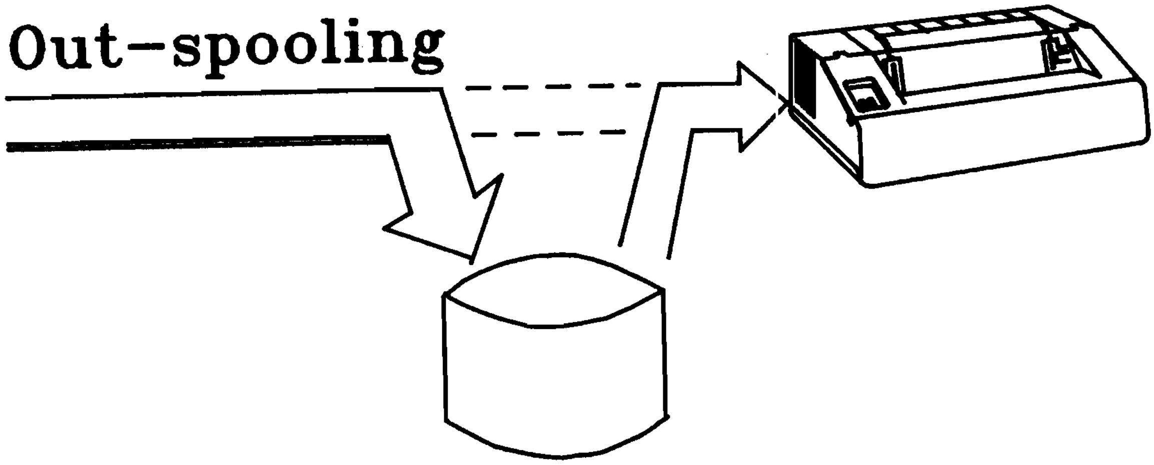
BENEFITS

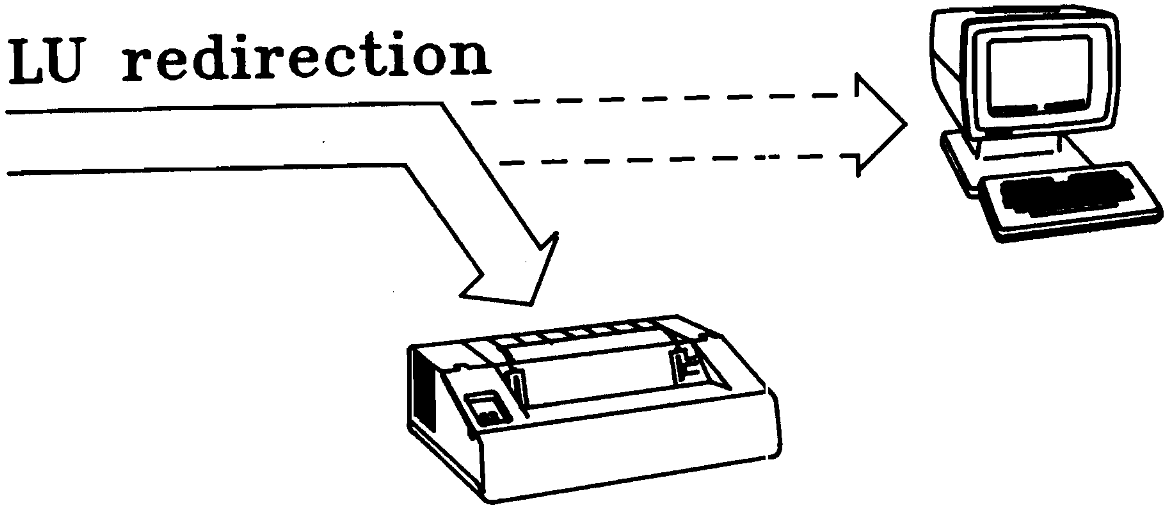
1. Provides an "unlimited" I/O buffer so the program need not wait for lengthy I/O.
2. Provides a user-independent way of sharing output devices.
3. Allows a program to complete I/O requests even if the I/O device is busy.
4. Allows redirection of the output from a program without changing the program itself.
5. Provides a log file to document user activity and system errors that are normally listed on the system console.

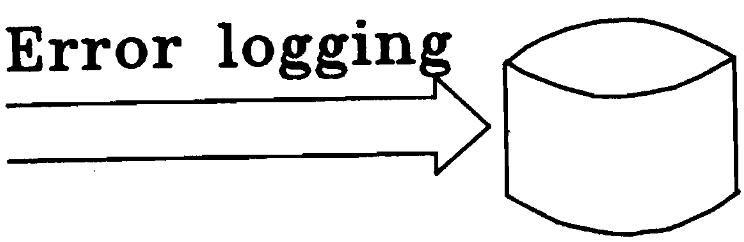
References: User's Manual

SPOOLING SYSTEM

(VC+)







1.25 LOGON Program

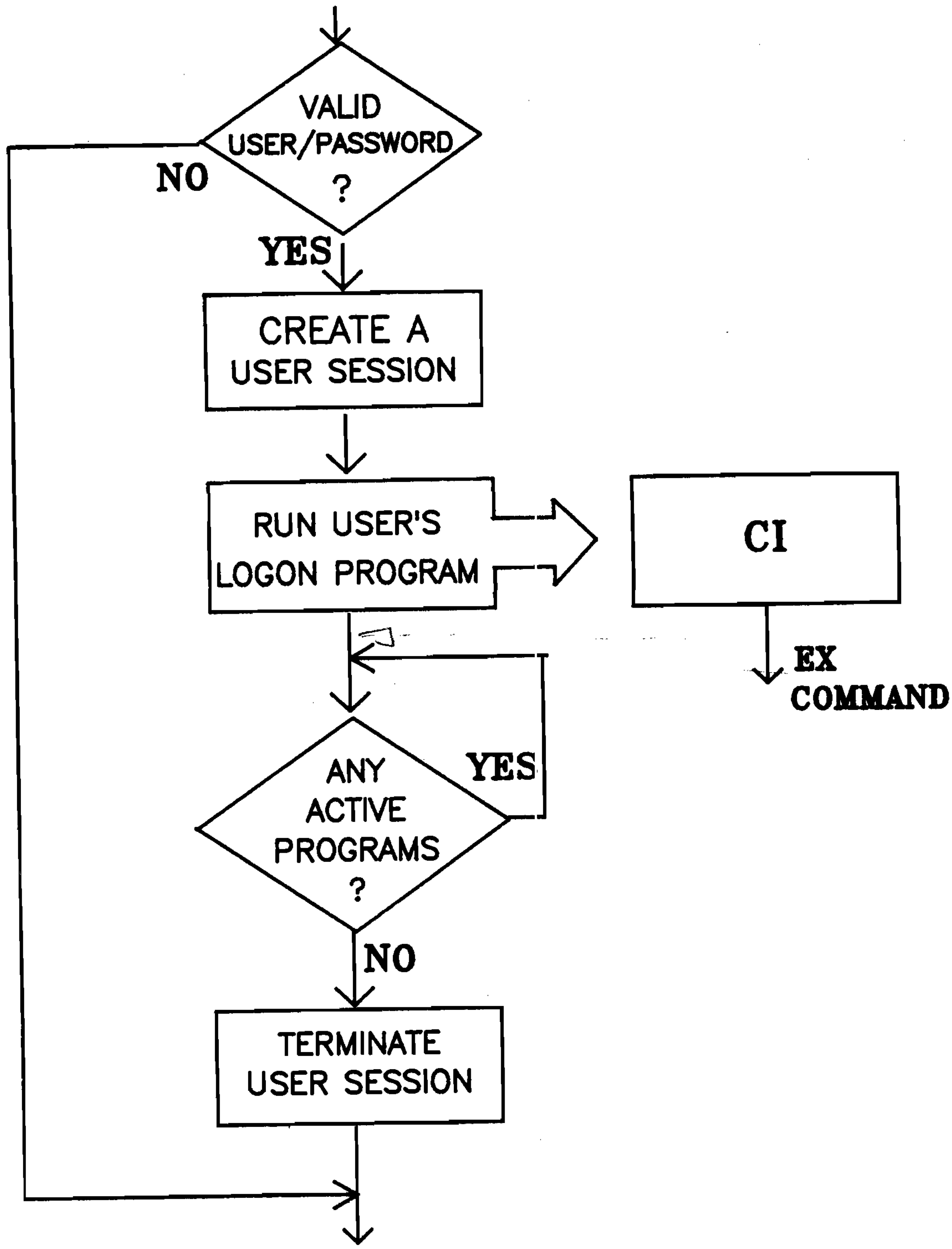
User Session -- Is created for each user that logs on. All programs run by the user are associated with the session. When all programs for the session terminate (including the user's logon program) the session goes away.

CI -- Is normally the program that is run when the user logs on. CI then issues the "CI>" prompt and executes commands issued by the user. The EX command exits CI which may or may not be the last program associated with the session.

References: User's Manual

LOGGING ON (VC+)

Please log in: William/casper



Please log in:

1.26 Logging Off

EX Command -- This terminates the CI program and it's associated session. If there are any other active programs, the user has the option to create a background session in which they can continue to run. When the last program terminates, the background session will go away.

LOGGING OFF (VC+)

```
CI> ex
Your programs:
 DLOG
 PRINT
 Continue, Logoff, Background or ? [C]
```

C = CONTINUE, IGNORE EX COMMAND

L = LOG OFF, TERMINATE PROGRAMS

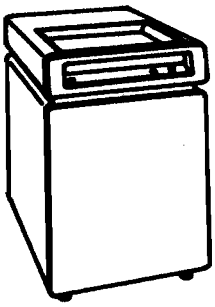
B = LOG OFF, CREATE A BACKGROUND SESSION IN WHICH TO CONTINUE THE ACTIVE PROGRAMS

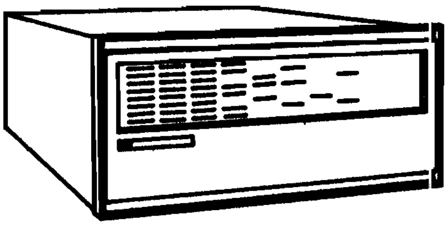
? = GET ADDITIONAL HELP WITH THIS COMMAND

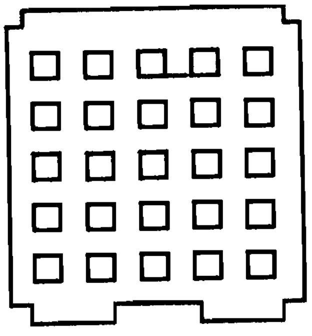
```
CI> ex
FINISHED
```

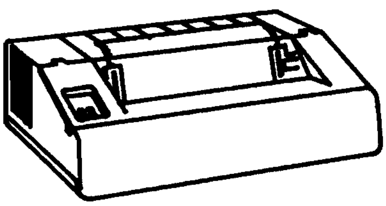
1.27 H A R D W A R E   O V E R V I E W

HARDWARE OVERVIEW









1.28 Block Diagram

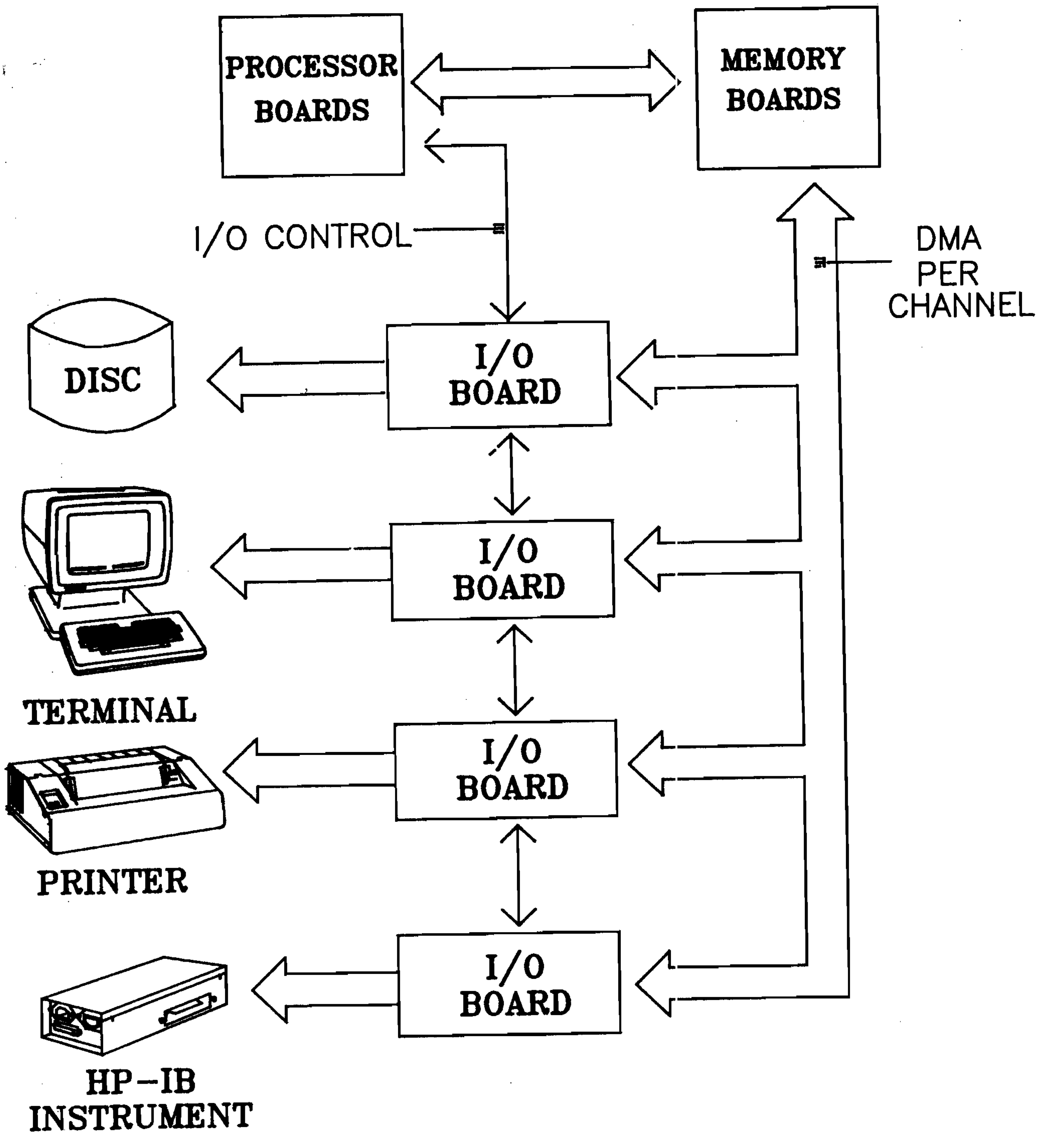
A600 - 1 processor board
1 memory controller with memory
0 to 4 additional memory boards (128 kb to 4 mb total)

A700 - 2 processor boards + 1 optional for floating point
1 memory controller
1 to 4 memory boards (128 kb to 4 mb total)

A900 - 3 processor boards
1 memory controller board
1 to 8 memory boards (768 kb to 6 mb total)

I/O Boards -- Async serial interface
Async serial fiber optic interface
HP-IB interface
8 channel multiplexer
300/1200 baud modem
Data link slave interface
Data link master interface
DS HDLC interface
DSN/MRJE interface
DSN/X.25 interface
Analog I/O
Digital I/O
PROM storage
A700 Writable control store
A700 PROM control store
A900 control store
Integrated disc controller
.
.
.

BLOCK DIAGRAM



1.29 Power-Up

There are 2 selftests executed when the machine is powered up or reset. The first is the microcoded selftest which tests the logic on the processor and memory controller boards. The execution time for this test is negligible. The second test is the assembly language test residing in the VCP ROMs.

This test checks:

- basic instruction set
- several internal flags
- all of memory (non-destructively if battery backup is used)

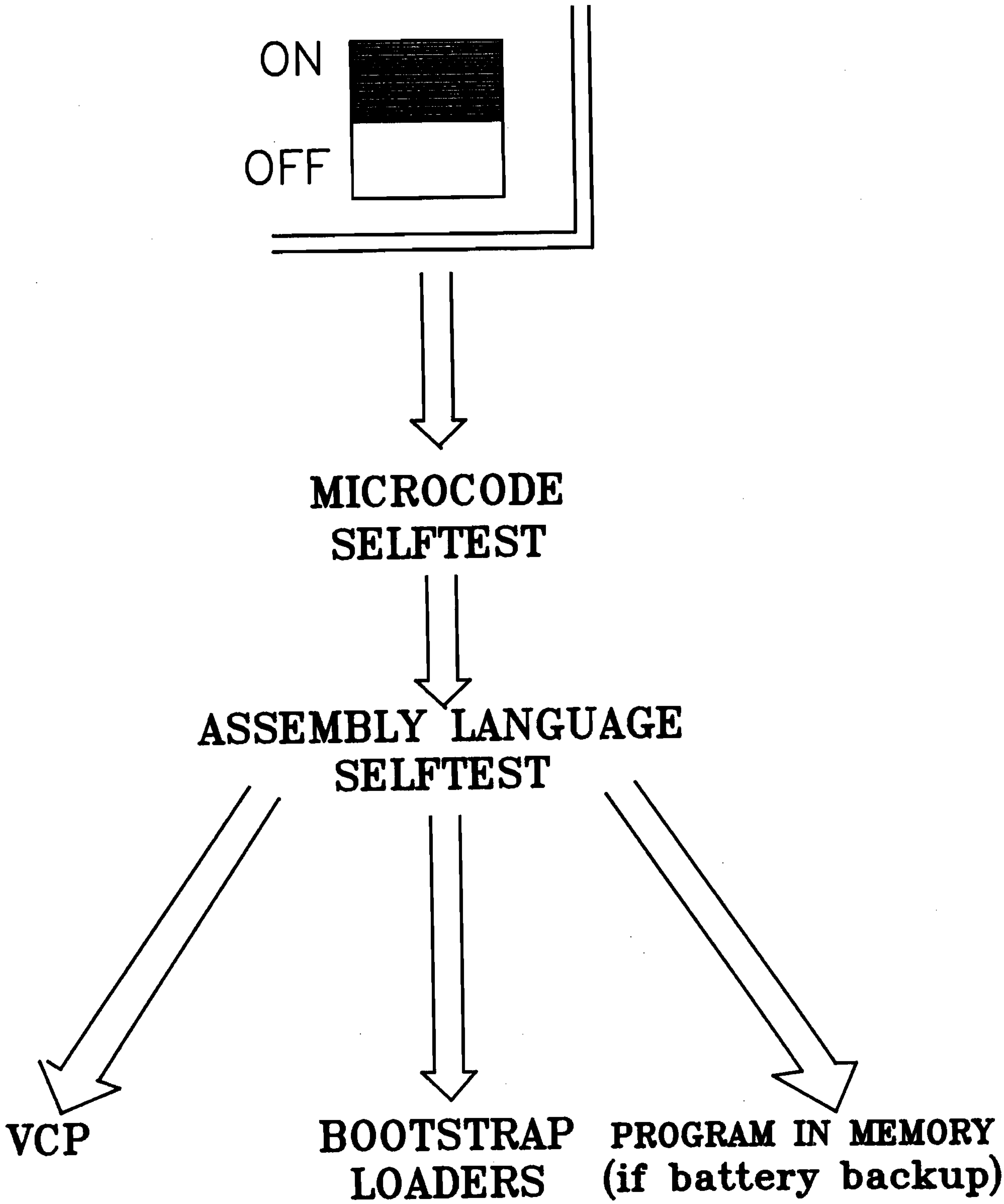
The execution time is less than 10 seconds.

Both test display pass/fail information on the LEDs which reside on the frontplane or processor card.

The path taken after the selftests complete is determined by switches on the frontplane or processor card.

References: System Installation Manual

POWER UP


ON
OFF
MICROCODE SELFTEST
ASSEMBLY LANGUAGE SELFTEST
VCP
BOOTSTRAP LOADERS
PROGRAM IN MEMORY (if battery backup)

1.30 VCP -- Virtual Control Panel

MEMORY -- Amount of standard memory available.

ECA -- Amount of error-correcting memory available.

P -- Program counter register.

A, B -- A- and B-Register.

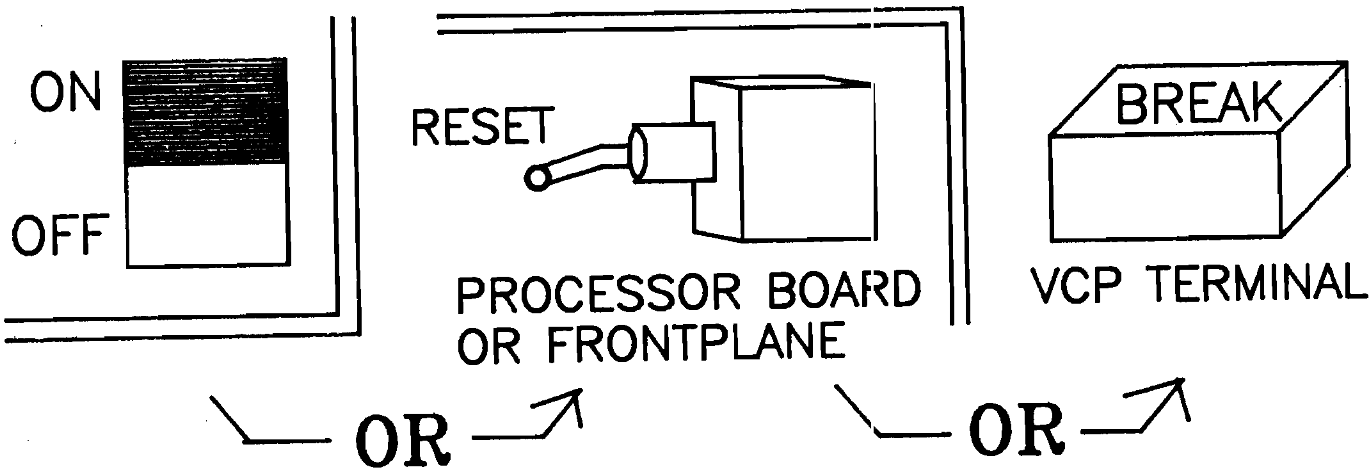
RW -- working map set used with dynamic mapping system.

M -- last memory location accessed.

T -- contents of location M.

References: System Installation Manual

V C P
VIRTUAL CONTROL PANEL



HP1000 A–SERIES ? FOR HELP

512KB MEMORY OKB ECA

P000000 A000003 B002011 RW000000 M000000 T000000

VCP> _

1.31 VCP Commands

References: Computer Reference Manual

VCP COMMANDS

- VIEW/ALTER HARDWARE REGISTERS
- VIEW/ALTER MAIN MEMORY
- BOOT
- LOAD
- RUN
- EXECUTE
- CLEAR MEMORY
- EXECUTE SELFTEST

USING YOUR RTE–A SYSTEM

CHAPTER 2

Table of Contents

Chapter 2
USING YOUR RTE-A SYSTEM

MODULE OBJECTIVES

1. Understand the relationship between CI, CM and SYSTEM (RTE) prompts and capabilities available from each.

2. Create a Pascal or FORTRAN source file using the editor.

3. Compile and load a simple Pascal or FORTRAN program.

4. Learn the concepts of real-time, background and time-sliced programs in relation to program priority.

5. Know the relationship of physical to logical memory and types of memory partitions to be found therein.

SELF-EVALUATION QUESTIONS

2-1. What is the program responsible for creating foreground sessions in a VC+ environment. Are multiple sessions available on a non-VC+ system?

2-2. With the CM> prompt on the screen, you type in a command. Do your keystrokes generate solicited or unsolicited interrupts to the terminal driver?

2-3. What causes a program to go from:

a. Dormant list to scheduled program list?

b. Scheduled list to execute state?

c. Execute state to wait list?

d. Execute state to dormant list?

2-4. The time-slice fence is at priority 50. Two programs of priority 70 are started simultaneously and their execution profile looks like (use your imagination):

```
"A"       |||---|||---|||---------------|||---|||---|||

"B"       ---|||---|||---|||||||||||||||---|||---|||---|||

time --> 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16
```

a. What can you say about the time-slice quantum?

b. What might have happened at time 5?

c. What could you change to make program A run to completion before program B started?

2-5. What is the difference between real-time and background programs?

2-6. There is a system program called "D.RTR" which does the communication with the disc drives for file calls, swapping, etc. It typically runs at priority 1. Is there any situation under which it might be swapped out for another program?

2.1 Properties of a Session (VC+)

Session Numbers -- When a user at terminal LU nn logs on to the system, the session created will be session number nn.

Attributes -- CI and all programs started by the user share the attributes of the session. If any of the attributes are changed (such as the working directory), they are changed for all those programs running under the session.

Programs -- Belong to the session under which they were invoked. This includes the copy of CI which was scheduled at logon.

Interactive Sessions -- Are created by the program LOGON and are always associated with an active terminal.

Background Sessions -- Are the system session, which is always active, and sessions created programmatically. Programs such as CI create background sessions (as when logging off with active programs) by use of session management subroutines described in the Relocatable Library Reference Manual.

References: System Design Manual

PROPERTIES OF A SESSION (VC+)

* **SESSION NUMBER SAME AS TERMINAL LU**

* **ATTRIBUTES:** User Name
  Working Directory
  Normal/Super–User

* **PROGRAMS BELONG TO THE SESSION**

* **FOREGROUND SESSIONS:**
  are interactive

* **BACKGROUND SESSIONS:**
  System Sessions
  DS Sessions
  "Logoff" Sessions

2.2 When CI is Busy

Non-VC+ systems -- Terminal driver selects the "primary" program or (if CI is busy), the "secondary" program CM. CM is a special copy of CI that will execute any of the CI commands but will exit after the completion of one command. If all else fails (CM is also busy), the RTE operating system issues a prompt and executes one of the base set commands.

Unsolicited Interrupt -- Is easier to define in terms of a solicited interrupt. When a READ statement is issued from a program, the terminal driver knows where to send data received from the keyboard. Each keystroke from the keyboard generates an interrupt which is said to be solicited and the terminal driver sends the received character to the soliciting program. If the terminal driver is not expecting data from the keyboard, each keystroke generates an interrupt which is said to be unsolicited and the terminal driver schedules a pre-defined program to handle the keyboard input.

Primary/Secondary Programs -- Are defined at system generation time. These are programs that will handle the keyboard input when the terminal driver does not otherwise know where to send the data from the keyboard.

RTE Base Set Commands -- Are a limited subset of the CI commands. The most used base set command is OF,CM which will abort the CM program and thus make it available to the terminal driver.

The following commands are available from the RTE> prompt

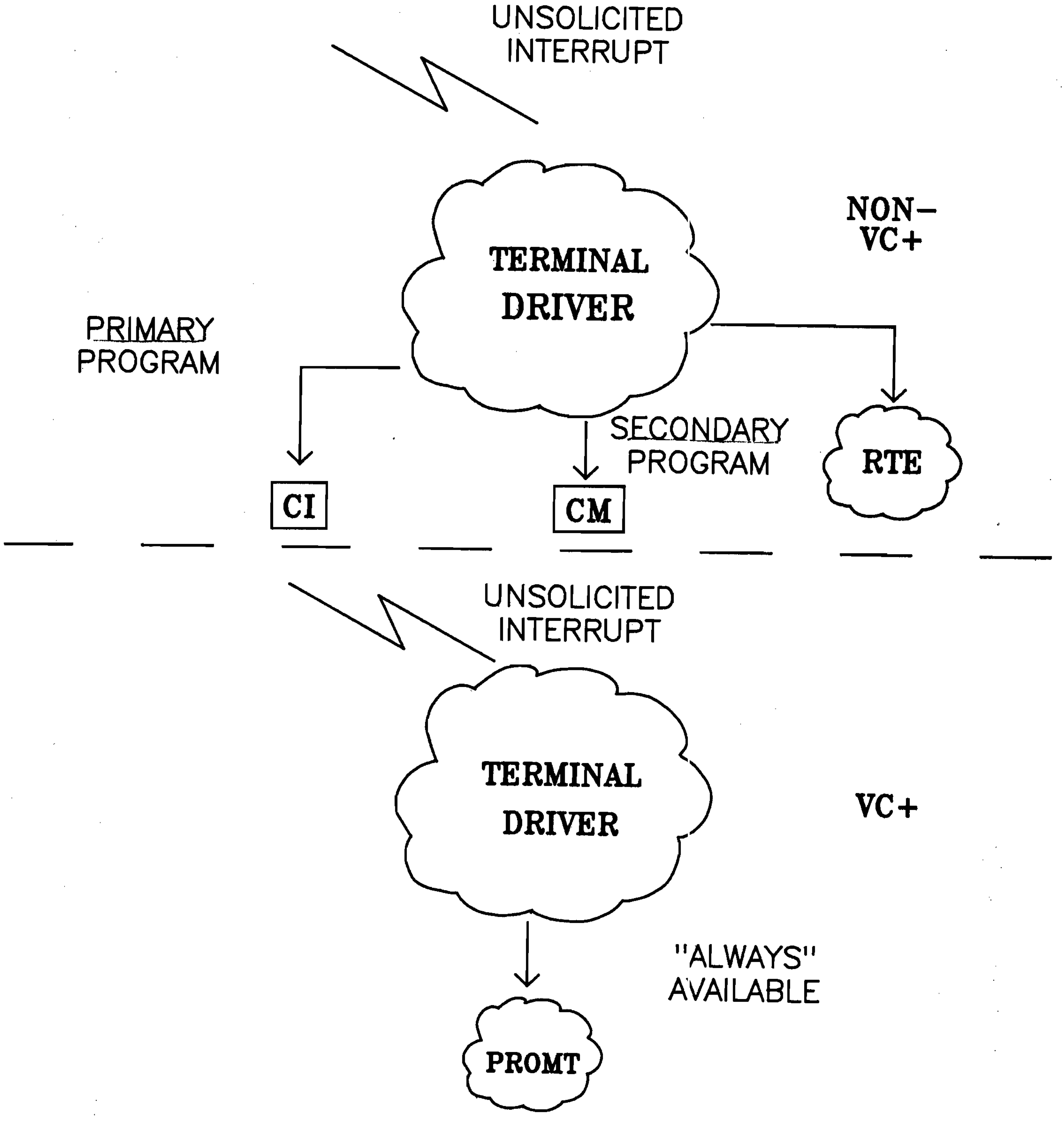
AS, BR, CD, DS, DN, DT, GO, OF, PR, PS,
RU, SS, SZ, TM, VS, UP, UL, WS, XQ

VC+ Systems -- When the terminal driver receives an unsolicited interrupt, the program PROMT is scheduled. PROMT checks to see if there is an session active for the terminal and schedules an appropriate program to handle the user input. PROMT is designed to execute very quickly and is effectively "always" available. For this reason, no secondary program is defined for VC+ systems.

PROMT -- NOTE: The program name is purposely spelled this way. It is left as an exersize to the reader to figure out why.

References: User's Manual, Driver Reference Manual

WHEN CI IS BUSY



R2.2

2.3 PROMT Program (VC+)

The terminal driver always schedules PROMT upon an unsolicited interrupt. PROMT does very little processing itself (it schedules other programs to do whatever is necessary), and therefore should never be busy.

PROMT first checks to see if a session is enabled for the terminal and if not, issues the logon prompt and schedules LOGON. Upon successful logon, LOGON will normally schedule CI.

If a key is pressed while CI is busy, PROMT will issue the CM> prompt and schedule CM. CM is a special copy of CI that executes a single CI command and then exits. Note that rapidly pressing a terminal key after sending a command to CM may result in another CM> prompt before the first command has been processed. Remember, the CM> prompt comes from the program PROMT and not from CM itself. PROMT will queue up such commands to CM, and will eventually schedule CM to process each of them.

If a terminal key is pressed when CM is busy, PROMT will issue a SYSTEM> prompt and take an RTE system level command similar to the RTE commands available in a non-VC+ system. The recommended action is to issue the command OF,CM to allow access to the CM program.

The following commands are available from the SYSTEM> prompt:

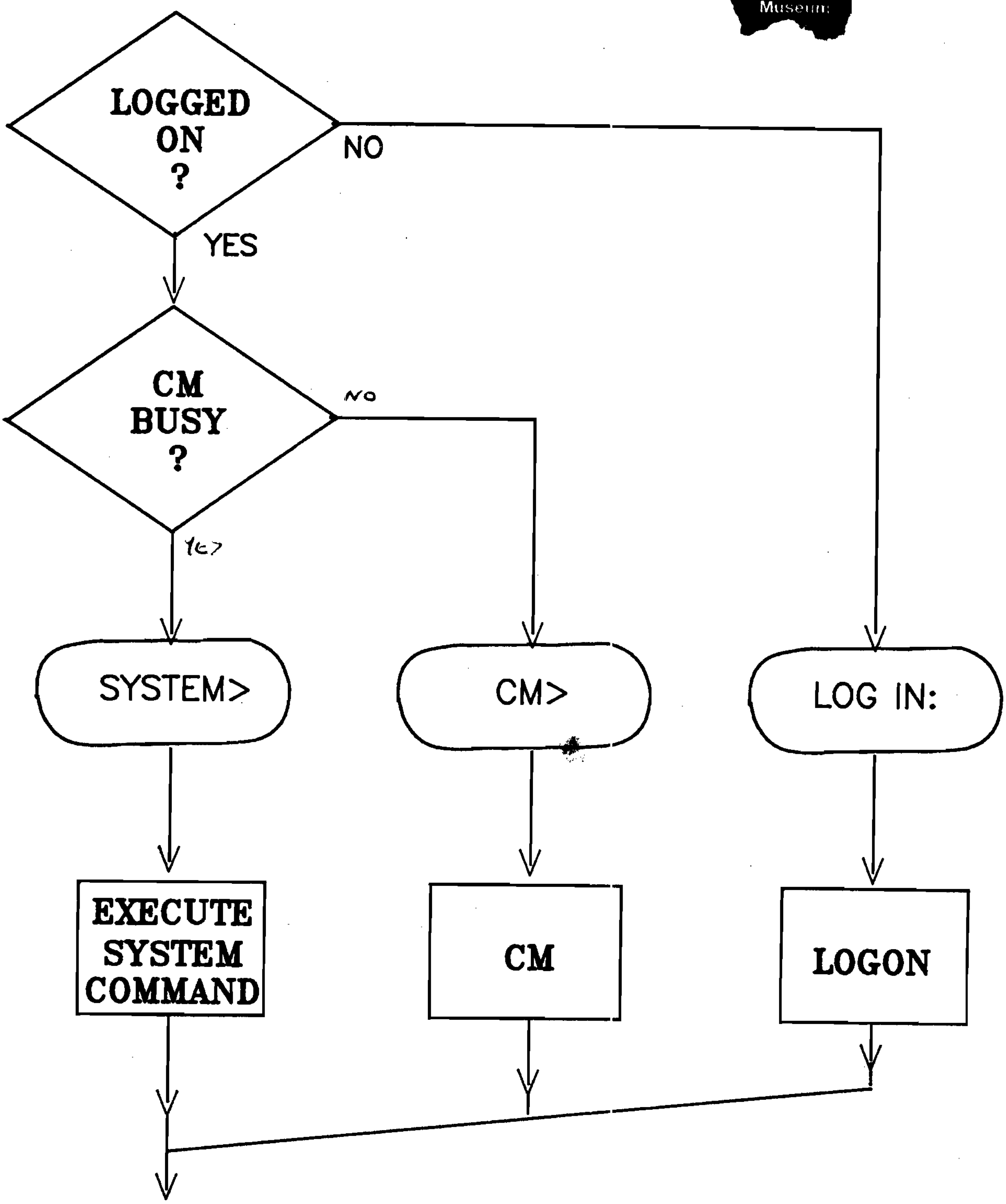
AS, BR, CD, DS, DT, GO, OF, PR, PS,
RU, SS, SZ, VS, UP, UL, WS, XQ

Note that these are the same as with RTE> with the absence of the commands TM and DN.

Note: When using the RU command from SYSTEM or RTE prompts, the program must already have an ID segment established (RP'ed).

References: User's Manual

PROMT PROGRAM (VC+)



2.4 Non-VC+ Considerations

The non-VC+ environment can be thought of as having one session with the name "SYSTEM". All users are part of the same session and the following characteristics apply:

Working Directory -- Changing the working directory changes it for all users. Application programs that reference files should always use full path names to avoid working directory problems.

File Protection -- No protection is available without the VC+ option. There is no counterpart to the FMGR security code.

Super-User -- All users are effectively super-users. There is no protection against file access, offing programs, changing system time or initializing disc volumes.

Spooling -- None of the facilities of the spooling system are available: out-spooling, LU redirection and error logging. The PRINT utility allows spooling to the printer (see the Utilities Manual).

References: User's Manual

NON–VC+ CONSIDERATIONS

* WORKING DIRECTORY

  same for all users

* FILE PROTECTION

  not available

* SUPER–USER

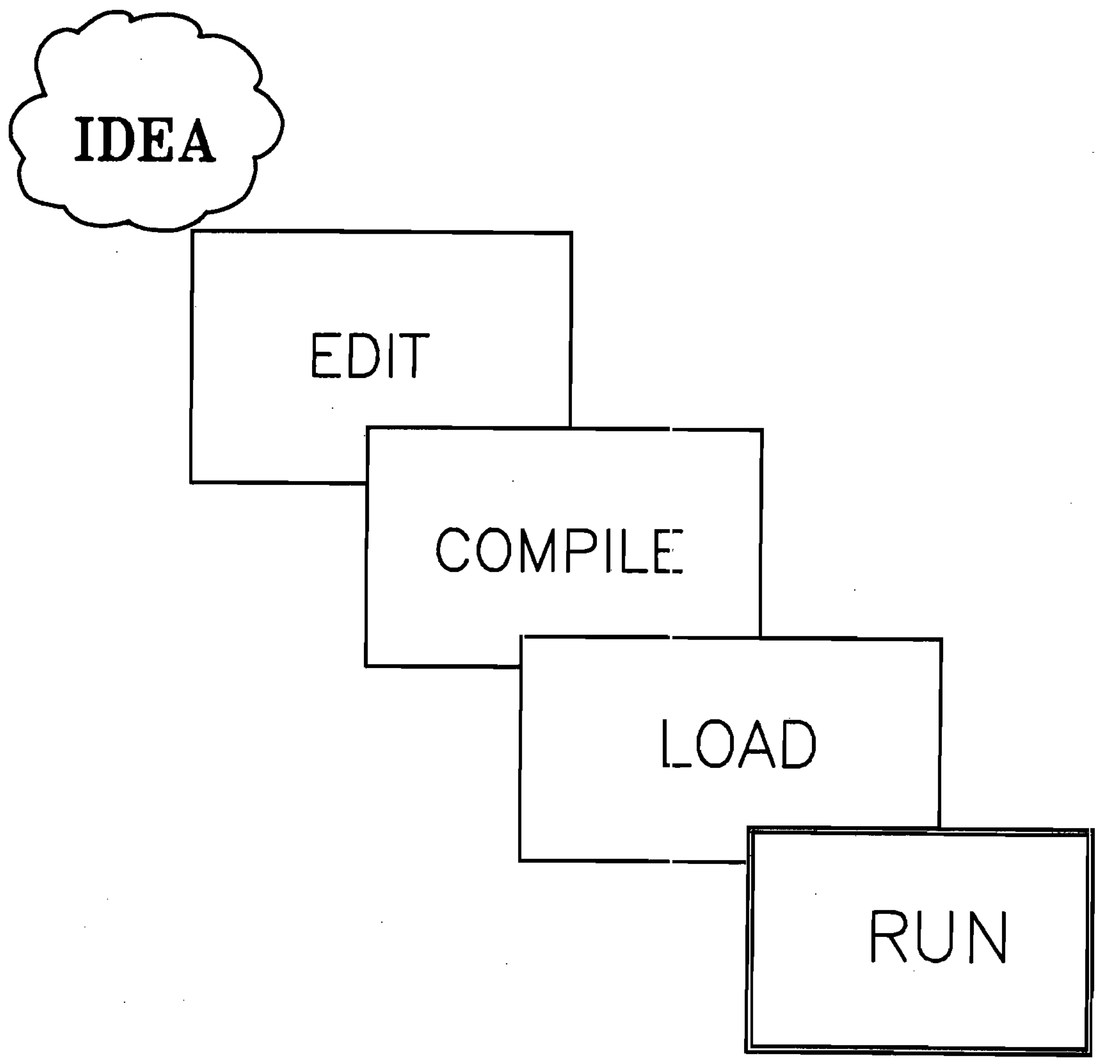
  all users have super–user capability

* SPOOLING

  printer spooling only (PRINT utility)

2.5 PROGRAM DEVELOPEMENT

PROGRAM DEVELOPMENT



2.6 EDIT/1000

EDIT runstring:

edit <sourcefile>

Screen commands:

^F -- forward screen
^P -- previous screen
^S -- screen from cursor position
^Q -- quit screen mode
^^... -- same as above but don't save screen

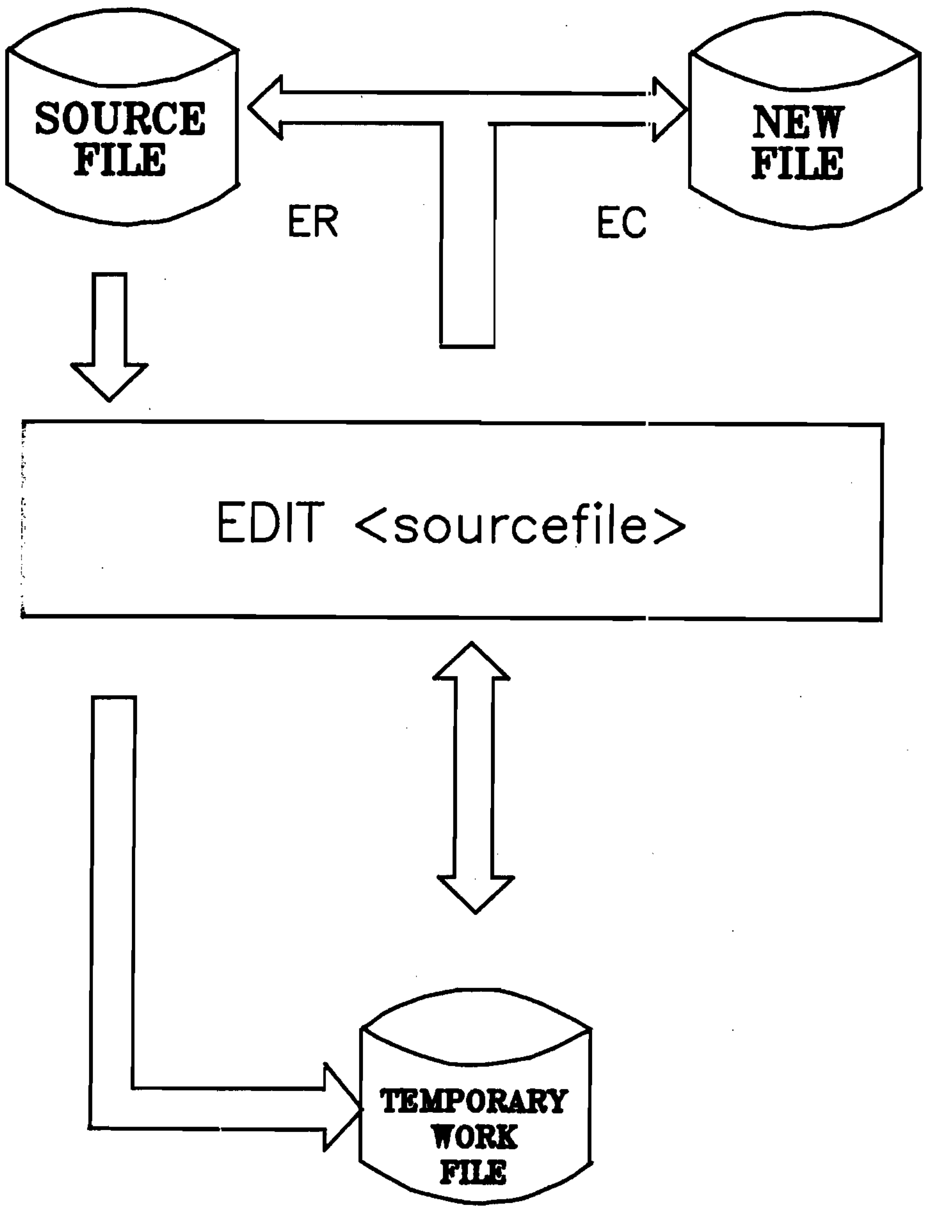
Exiting EDIT:

EC -- exit create
ER -- exit replace
A -- abort

^ = the CNTL key. Hold this key down while typing the lettered key.

References: EDIT Reference Manual

EDIT/1000


SOURCE FILE
NEW FILE
ER
EC
EDIT <sourcefile>
TEMPORARY WORK FILE

2.7 A Simple Pascal Program

Computes area of circle given the radius.

Note HP Pascal extension:

* Use of underscore as the last character in a writeln statement leaves cursor on same line.

References: Pascal Ref Manual

A SIMPLE PASCAL PROGRAM

```
program area (input, output);
var radius, area:real;
begin
    writeln ('area of circle program');
    repeat
        writeln ('radius:_');
        read (radius);
        area:=3.14159*radius*radius;
        if radius > 0
            then writeln ('area=', area:4:2)
            else writeln ('finished')
    until radius <=0
end.
```

2.8 Compiling a Pascal Program

PASCA -- Five character program name for PASCAL program file.

<filename> -- Pascal source file created by EDIT. Name should have .PAS type extension.

<list> -- Can be a device LU or a filename to which is sent the source listing and compiler errors.

"-" -- Relocatable filename defaulted to same as source file except with a .REL type extension.

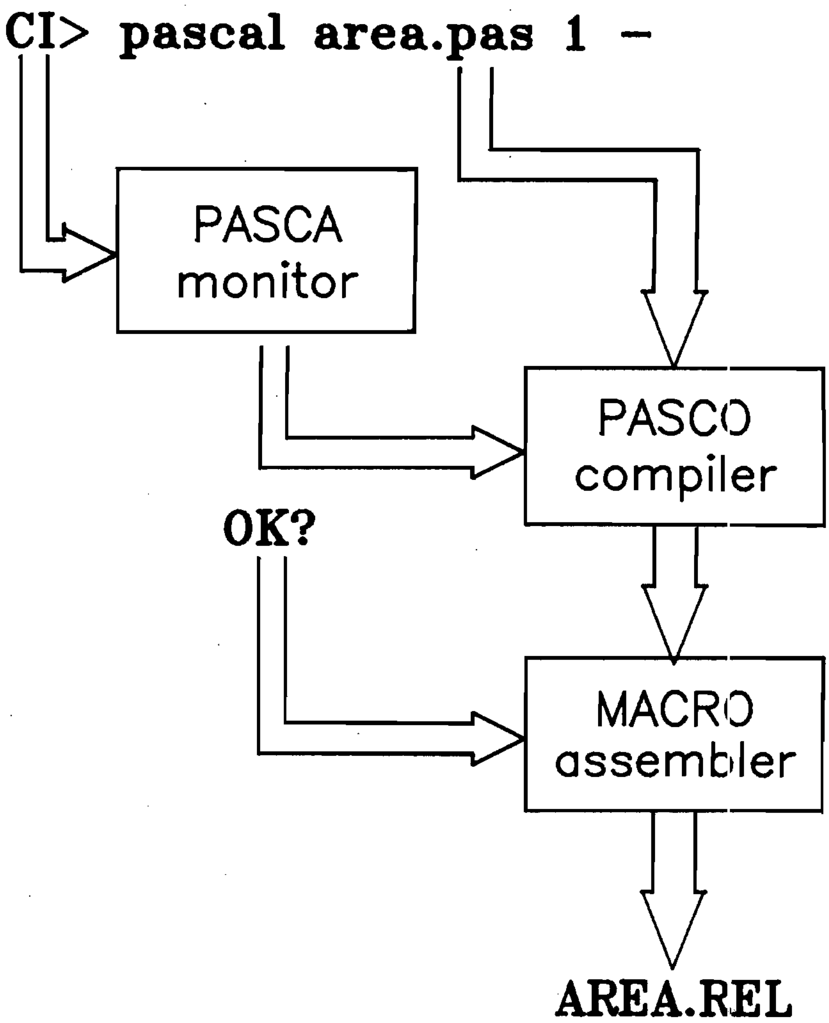
PASCO -- Pascal compiler, accepts Wirth standard Pascal, ANSI standard, HP Pascal, and HP 1000 Pascal. Produces a temporary file to pass to the macroassembler.

MACRO -- Macroassembler scheduled if compiler finishes with no errors.

AREA.REL -- Relocatable file created by MACRO.

References: Pascal Ref Manual

COMPILING A PASCAL PROGRAM



2.9 A Simple FORTRAN Program

Computes area of circle given the radius.

Note HP FORTRAN extension:

* Use of underscore as the last character in a write statement leaves cursor on same line.

References: FORTRAN Ref Manual

A SIMPLE FORTRAN PROGRAM

```
program area
real radius, area
write (1,'("area of circle program")')
radius = 1
do while (radius.GT.0)
    write (1,'("radius:_")')
    read (1,*) radius                inhibit CRLF
    area=3.14159*radius*radius
    if (radius.GT.0) then
        write (1,'("area="F4.2)') area
    else
        write (1,'("finished")')
    end if
end do
end
```

2.10 Compiling a FORTRAN Program

FTN7X -- Name of FORTRAN program file. Compiler accepts ANSI standard FORTRAN 77 and MIL-STD-1753 FORTRAN plus a number of HP extensions to the language.

<filename> -- FORTRAN source file created by EDIT. Name should have .FTN type extension.

<list> -- Can be a device LU or a filename to which is sent the source listing and compiler errors.

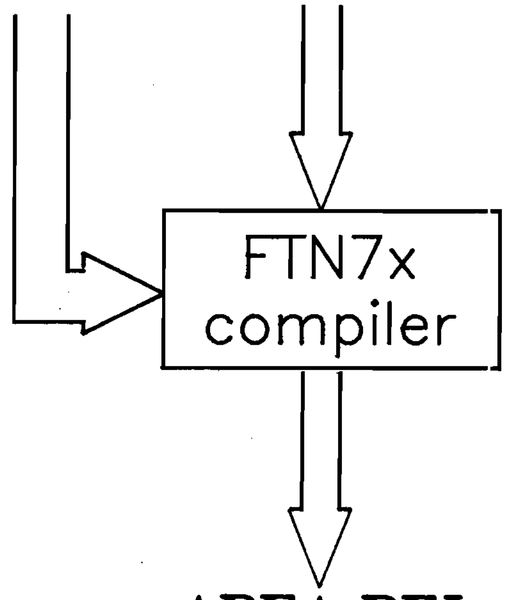
[illegible] -- Relocatable filename defaulted to same as source file except with a .REL type extension. Source file must have .FTN extension

AREA.REL -- Relocatable file created by FTN7X.

References: FORTRAN Ref Manual

COMPILING A FORTRAN PROGRAM

CI> ftn7x area.ftn 1 -



AREA.REL

2.11 LINK Relocating Loader

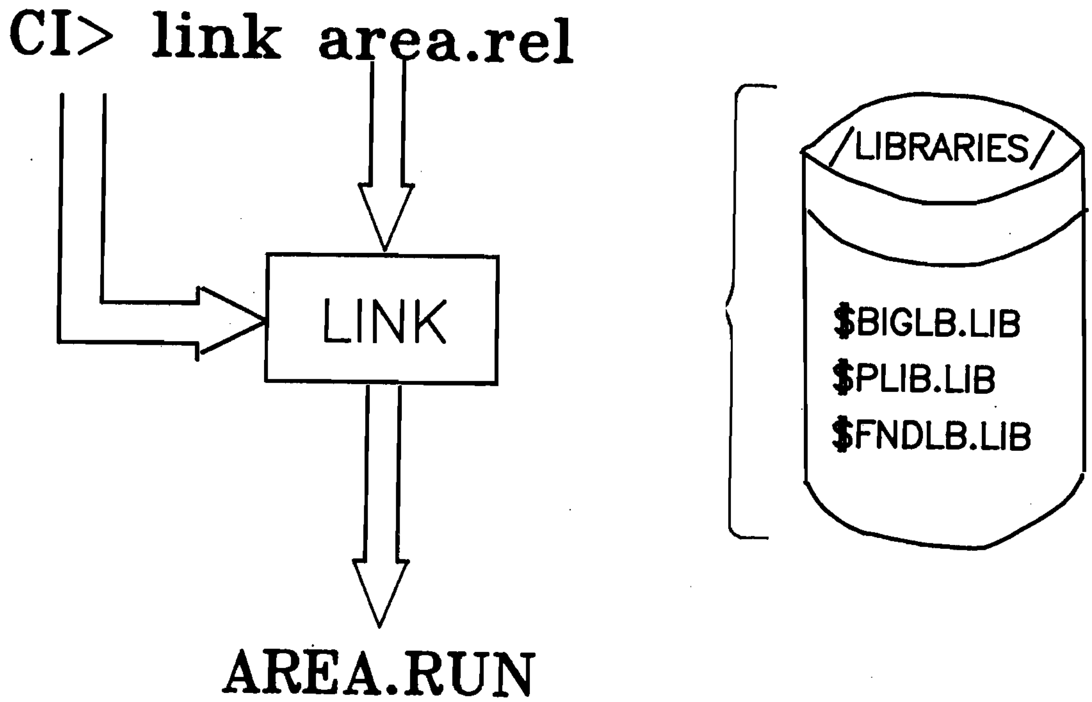
AREA.REL -- Relocatable file produced by compiler

/LIBRARIES/ -- Directory that contains system and user library routines. These libraries may or may not be searched automatically by LINK. The default search libraries are defined at system generation time.

AREA.RUN -- Runnable memory image code.

References: Link User's Manual

LINK
RELOCATING LOADER



2.12 PROGRAM EXECUTION ENVIRONMENT

PROGRAM EXECUTION ENVIRONMENT

2.13 Program Scheduling

RP Command -- (Restore Program) Initializes an ID segment. This command is normally not necessary because of the action of the RU command.

RU Command -- Moves the program from the dormant state into the scheduled program list. Note that the RU command will implicitly RP the program if it is not found in the dormant program list (i.e., if it does not have an ID segment).

Dormant Program List -- This contains all programs that have been explicitly RP'ed but are not yet scheduled to run.

Scheduled Program List -- This contains all programs that have been scheduled via the RU command or other means. The programs are dispatched in order of priority.

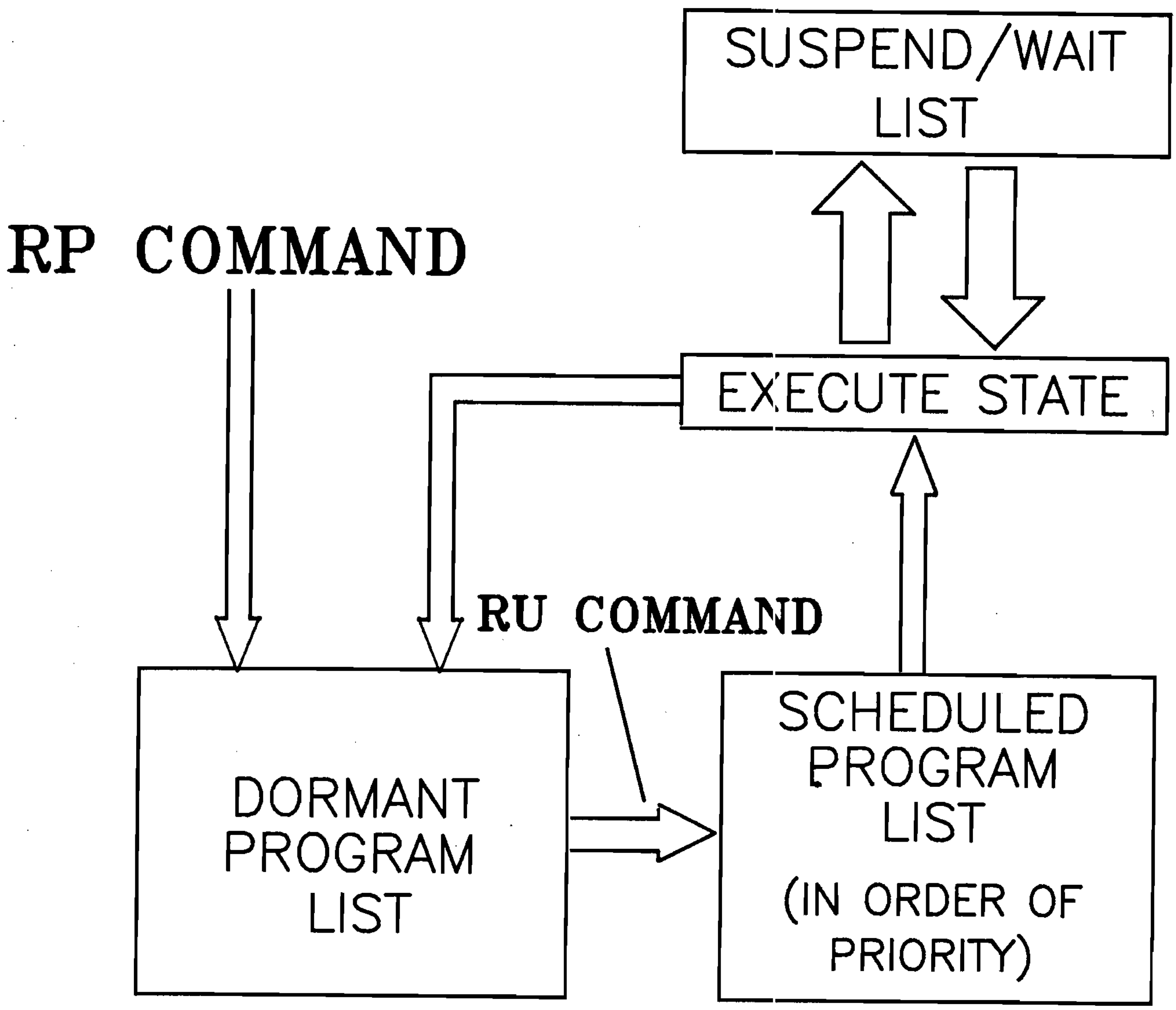
Priority -- Order of importance represented by integers in the range 1 to 32767, 1 being the highest priority.

Execute State -- The program is currently executing.

Suspend/Wait List -- The program is active but suspended for some reason (e.g., requested a resource that is not immediately available, a higher priority program was ready to run).

References: System Design Manual

PROGRAM SCHEDULING



2.14 Priorities

Time-Slice Fence -- Programs of priority lower than the boundary are subject to time-slicing, but only if the programs have the same priority. The value of the time-slice fence is set at system generation time and may be changed in the boot-up command file.

Program Priority Boundary -- Also called the background fence. Programs of priority lower than the boundary are considered background programs. All others are real-time programs. Background programs get chosen first as candidates to be swapped to disc to make room for programs requiring memory from the system. The value of the program priority boundary is set at system generation time and may be changed in the boot-up command file.

References: System Design Manual

PRIORITIES

1 — HIGHEST PRIORITY

execute to completion

30 — — — — — — TIME–SLICE FENCE

50 — — — — — — PROGRAM PRIORITY BOUNDARY

32767 — LOWEST PRIORITY

2.15 Multi-Programming

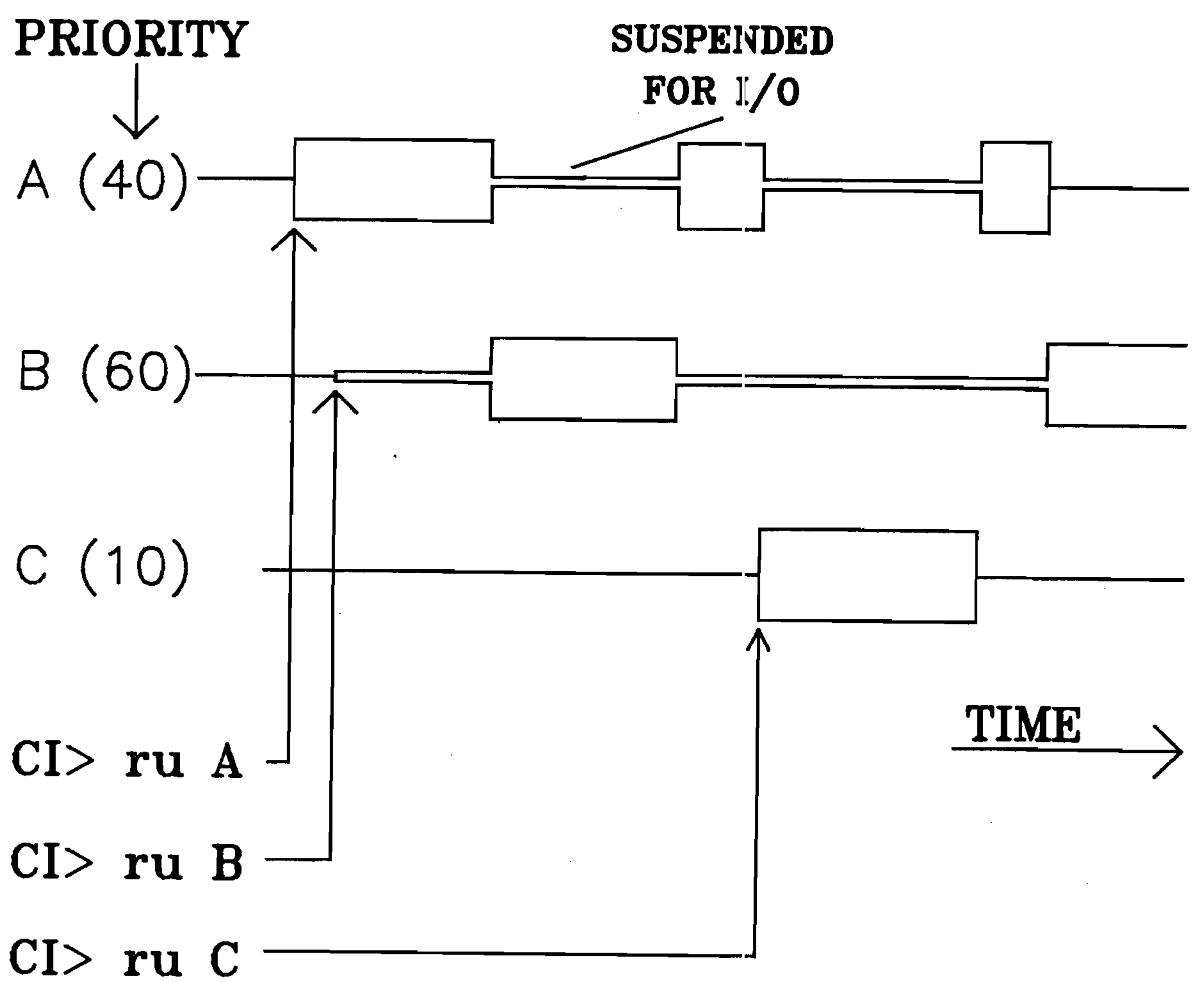
Program A -- An I/O intensive program.

Program B -- A compute intensive program of lower priority than program A.

Program C -- A high priority program.

References: System Design Manual

MULTI–PROGRAMMING



2.16 Time-Slicing

Time-Slice Fence -- Programs with priorities lower than the fence will time-slice with programs of equal priority. Programs with priorities equal to or higher than the fence will run to completion (or suspension) before another program of equal priority can run. The time-slice fence is set at system generation and may be changed in the boot-up command file.

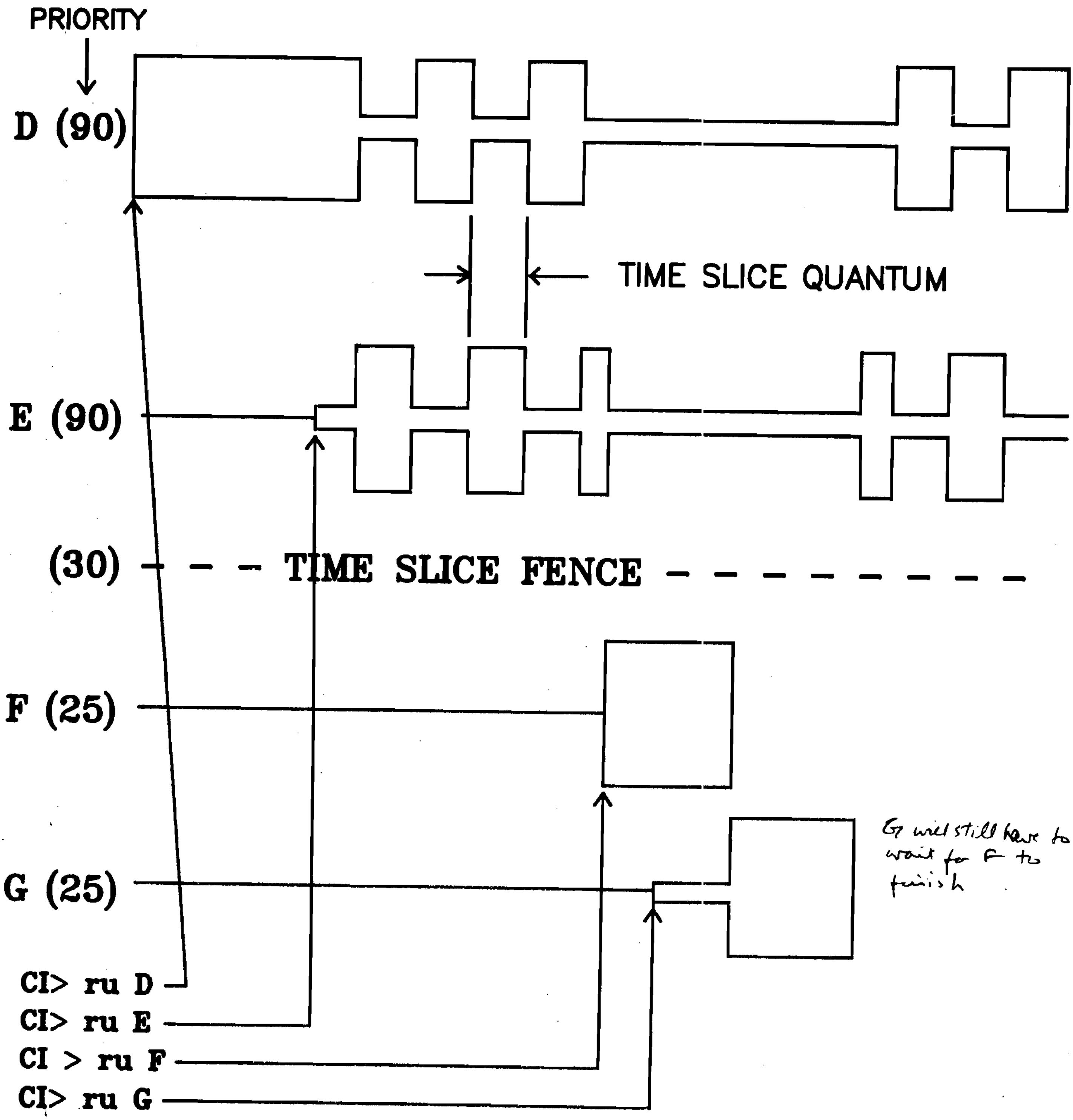
Time-Slice Quantum -- Is the length of time a program will run before being suspended to let another program (of equal priority) run. The time-slice quantum is set at system generation and may be changed in the boot-up command file.

Programs D, E -- Compute intensive programs, same priority, time-slice side of fence.

Programs F, G -- Compute intensive programs, same priority, other side of fence.

References: System Design Manual

TIME–SLICING



R2.16

2.17 Swap File

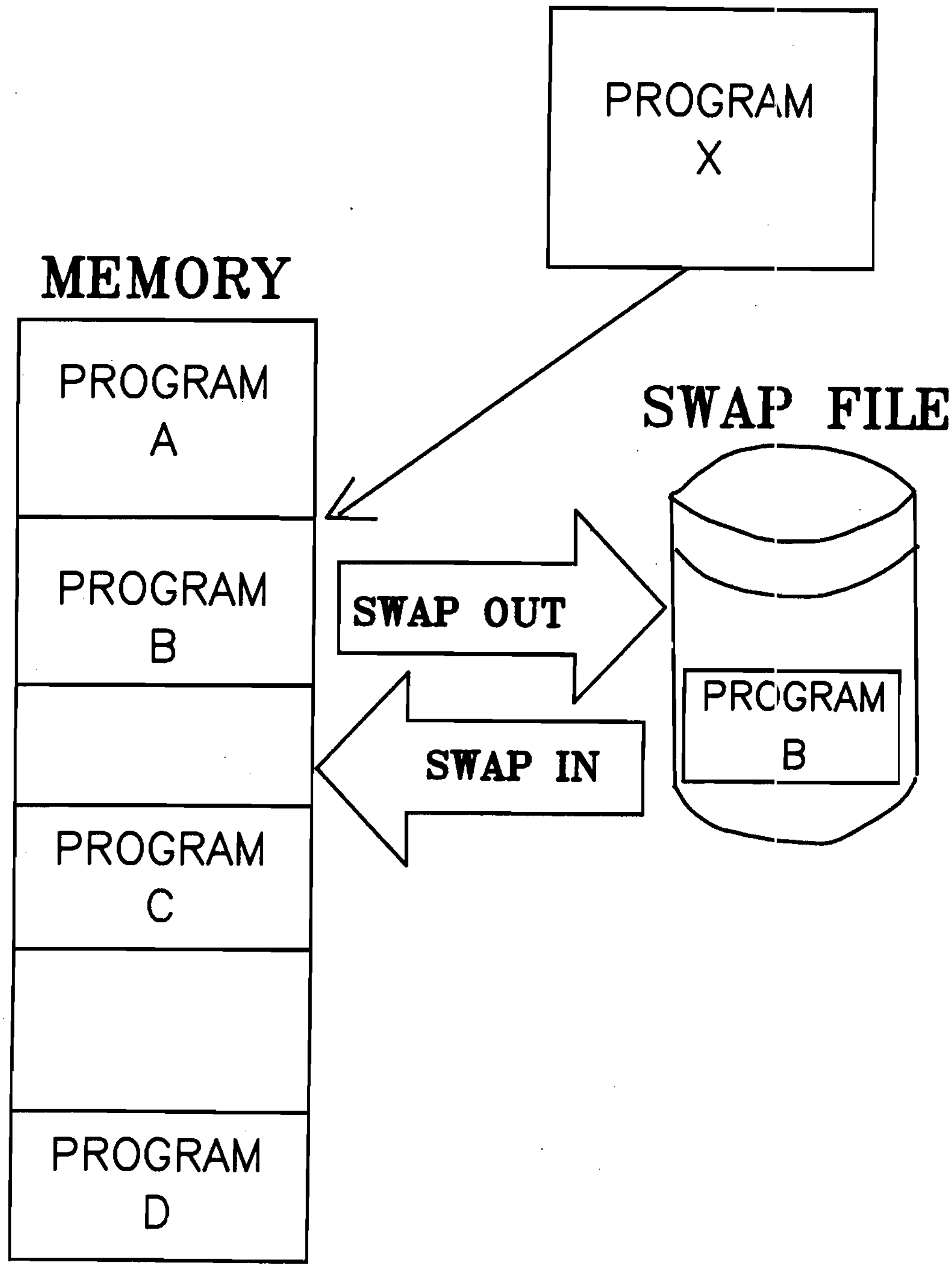
Swap file -- The name and size of the swap file are defined in the bootup command file.

Swap Out -- If no partitions are available for a scheduled program, space is made available by swapping some other program (or programs) out to the swap file. The program to be swapped out is chosen for the least impact on the system and is typically not running at the time (i.e., waiting or suspended).

Swap In -- When the swapped out program is again ready to run (i.e., at the top of the scheduled list) it is brought back into memory from the swap file.

References: System Design Manual

SWAP FILE


PROGRAM
X
MEMORY
PROGRAM
A
PROGRAM
B
PROGRAM
C
PROGRAM
D
SWAP FILE
SWAP OUT
SWAP IN
PROGRAM
B

2.18 Program Swapping

Program Priority Boundary -- Defines the distinction between real-time and background programs. The default value is set during system generation and can be changed in the boot-up command file.

Background Program -- Has priority set lower or equal to the program priority boundary.

Real-Time Program -- Has priority set higher than the program priority boundary.

Free Memory -- Unoccupied memory partitions.

Suspended Real-Time Program -- Waiting for I/O.

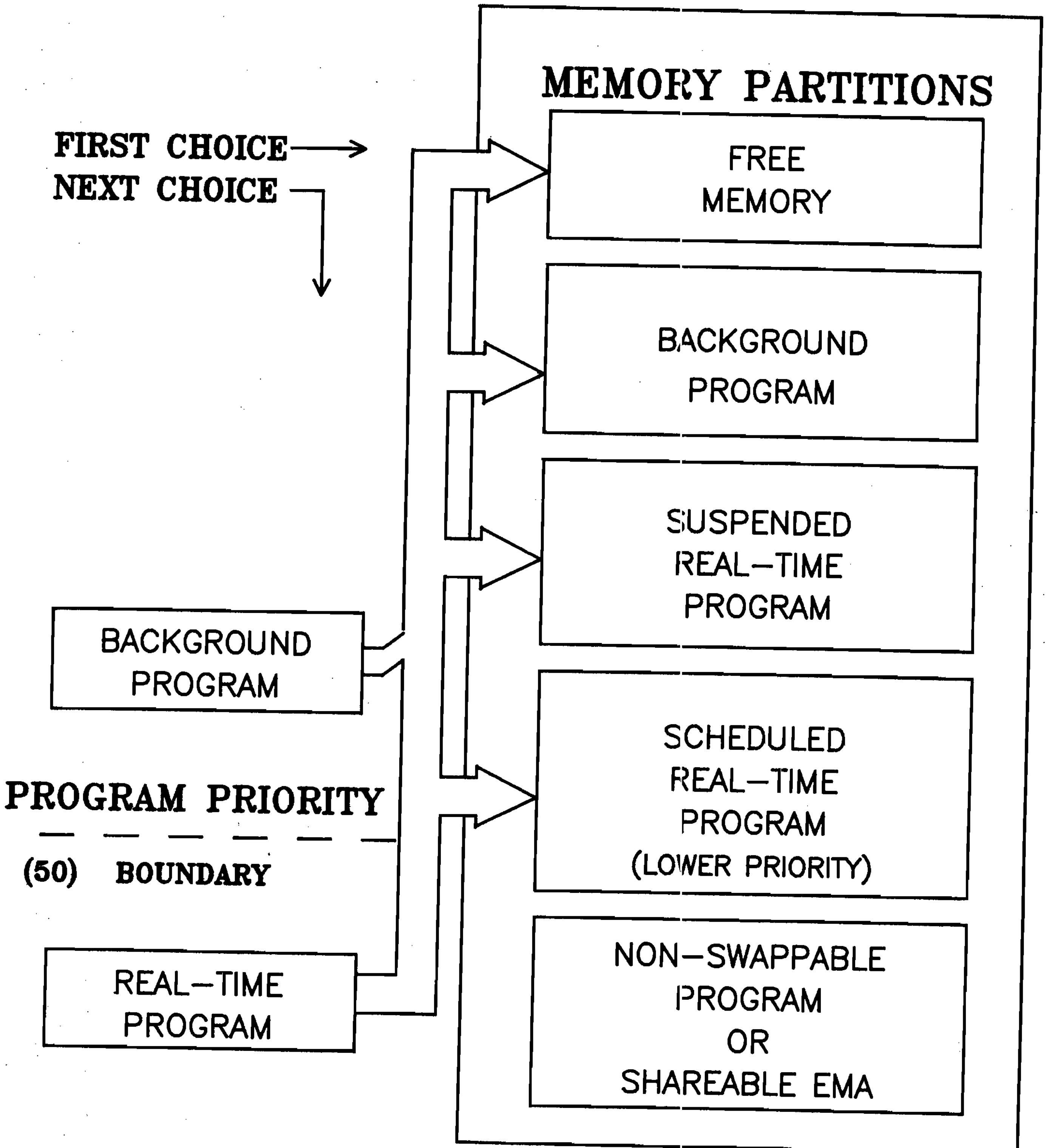
Scheduled Real-Time Program -- Currently running or waiting on higher priority programs to run.

Non-Swappable Program -- A program that executed an EXEC 22 request or is I/O suspended with a buffer in the program partition.

Shareable EMA -- Extended Memory Area (for large amounts of data) that multiple programs use to share data.

References:

PROGRAM SWAPPING



R2.18

2.19 User Memory Partitions

CDS data partition (VC+) -- Contains all data for a CDS program including heap area (e.g., Pascal dynamic variables), stack used for saving local variables upon procedure entry, and global data area. This also contains the EMA area or VMA working set if these facilities are used.

CDS Code Partition (VC+) -- Contains the code portion of a CDS program. This portion may be shared among many users since user specific data is contained in the data partition.

Non-CDS Program Partition -- Contains the program and data for a non-CDS program.

References: System Design Manual

USER MEMORY PARTITIONS

CDS DATA PARTITION (VC+)

EMA/VMA
HEAP/STACK
GLOBALS

CDS CODE PARTITION (VC+)

CODE
SEGMENT
BLOCKS

NON-CDS PROGRAM

HEAP AREA
SEGMENT OVERLAY AREA
MAIN PROGRAM

2.20 Physical Memory

User memory

Dynamic Partitions -- Variable sized partitions allocated from free memory and, if necessary, background or suspended program partitions. This is where normal programs are run which would include CI, LINK, Pascal, EDIT, user application programs, etc.

Reserved Partitions -- Fixed size partitons that are used by programs specifically assigned to them. These partitions must be specified during system generation.

System memory

Operating System -- This includes program scheduling, resource management, I/O requests, memory management, system clock, spooling (VC+), basic system commands and error handling routines.

Drivers -- These routines take generic I/O requests from the system program and convert them to the necessary format for the I/O board and device addressed.

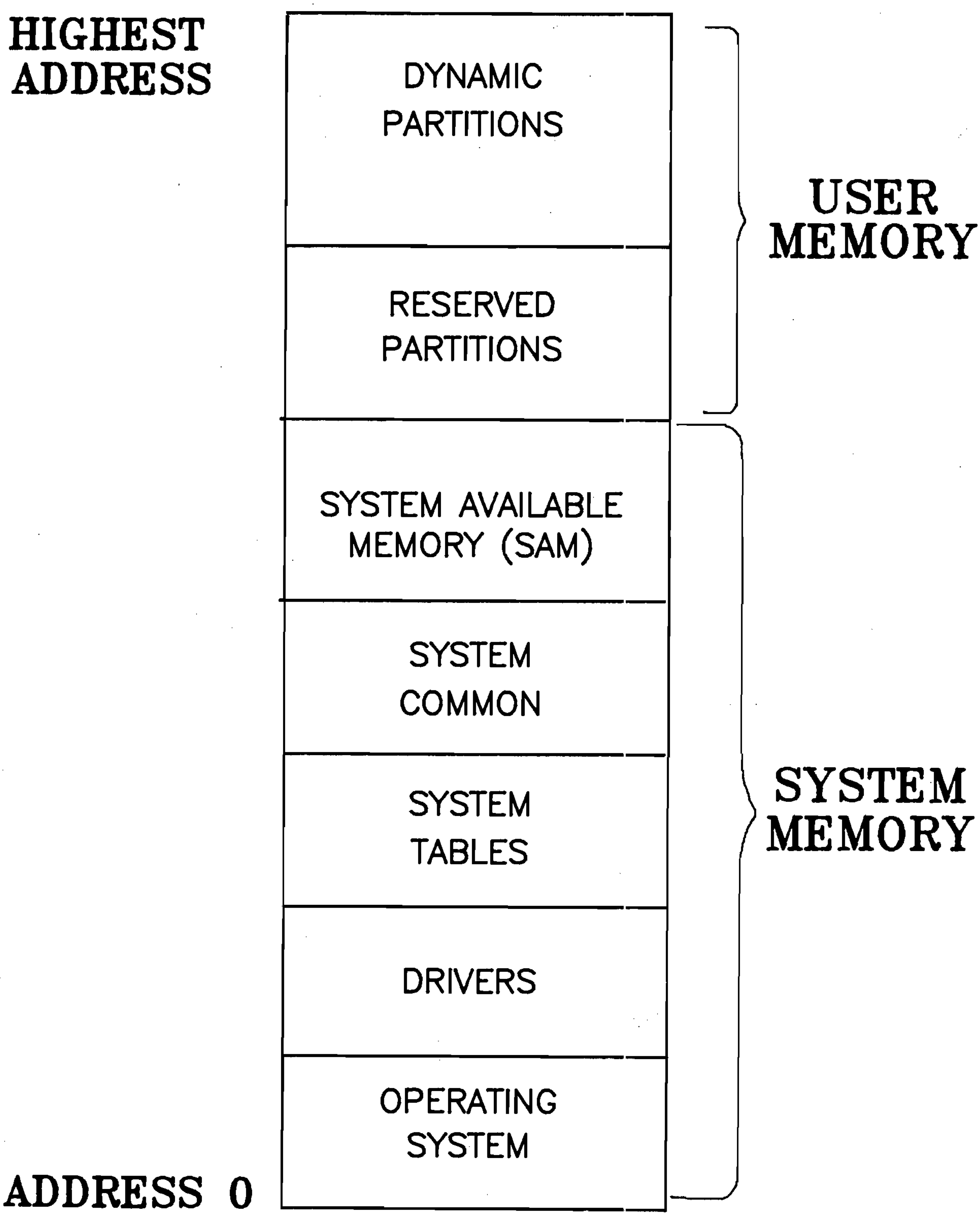
System Tables -- These provide flexibility in the number and type of devices the operating system can control and allow efficient operation for either minimally or maximally configured systems.

System Common -- This area may be mapped into multiple program partitions to allow common data areas between programs.

System Available Memory (SAM) -- This is a memory area used by the system for I/O buffering, class I/O (mailbox I/O), and string passage.

References: System Design Manual

PHYSICAL MEMORY



2.21 Logical Memory

A user program partition is shown as an example.

Physical Memory -- The portions that make up a logical memory partition may come from distinctly separate physical areas.

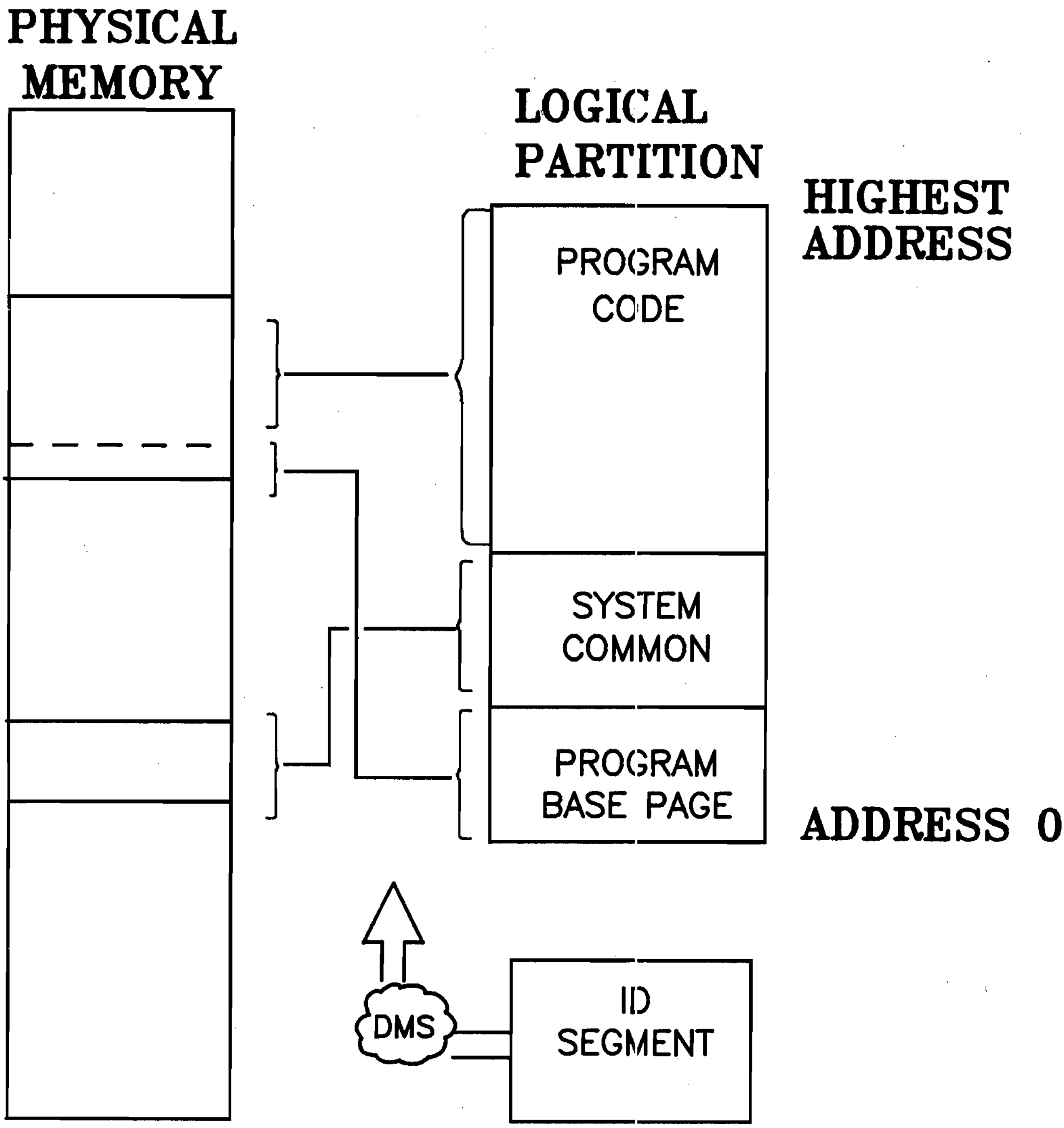
Logical Partition -- Is seen by the program as one contiguous logical unit. References to logical memory locations are sequential from 0 to the top of the logical partition.

ID Segment -- Contains the information necessary to set up the mapping between physical memory and the logical partition for the program.

DMS -- Dynamic Mapping System, to be described later in this chapter.

References: System Design Manual

LOGICAL MEMORY



R2.21

2.22 Dynamic Mapping System DMS.

Logical Address -- Obtained from the program instruction. The logical address space is 32k words (from 15 bits). Ten bits are used to select one word from a 1024 word page in physical memory. Five bits are used to select one of 32 page mapping registers. The 16th bit is used for indirect addressing.

$2^{10} = 1024$

	Memory register		page offset
15	14	11 10 9	0

select from bit 11-14 of logical address

Page Mapping Registers -- Contain a 14-bit address that selects one of the available pages in physical memory. The address in the PMR is set up by the operating system when a partition is allocated. The remaining two bits in the register are used for read and/or write protection. $2^{14} =$

	page address		word address
15	14	10 9	0

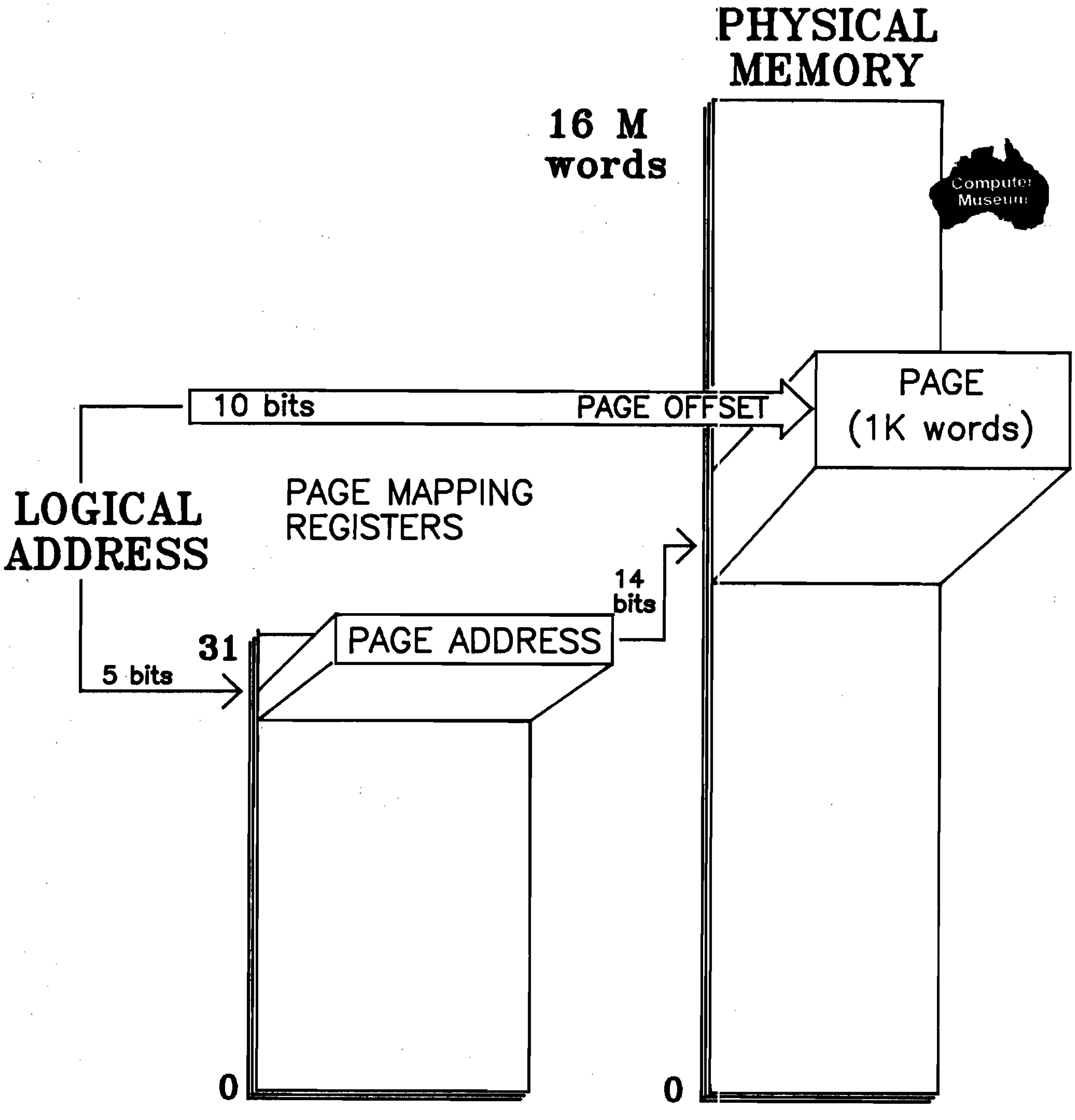
Physical Memory -- Up to 16 megawords (16384 pages) although the actual size is limited by current memory board density and the size of the customer's budget.

Bit 0-9 of logical address ~~indicate~~ indicate which word in the page
Bit 0-13 of Page Mapping register " " page of memory

1 page = 1024 words = 1 k words
1 block = 128 word
1 word = 2 bytes
1 block = 128 words = 256 bytes

References: System Design Manual, Computer Ref Manual

DYNAMIC MAPPING SYSTEM



FILE SYSTEM

CHAPTER 3

Table of Contents

MODULE OBJECTIVES

1. Be able to use file manipulation commands with various path lengths and working directories.

2. Use source and destination file masking with wildcard characters, time stamps, directory paths, and file types.

3. Use file protection and directory ownership features.

4. Use the spooling feature to outspool files and redirect LUs.

5. Use DS transparency to copy files from one system to another.

SELF-EVALUATION QUESTIONS

3-1. What is the root directory? How do you list it's contents?

3-2. What is the difference between a global directory, a sub-directory, and a working directory?

3-3. What type of file does EDIT create? How could you specify a different type? What would happen if you specified a type 99 file?

3-4. What type of file requires a record length specification when you create it? Why?

3-5. Change the following file descriptors to hierarchical format:

a. memo::henry

b. orders/july_12::produce_dept

c. ::henry.dir:2:48:32

Change the following file descriptors to combined format:

d. /programs/ci.run

e. /henry/docs/zap.txt

f. /friday:::4:30

3-6. A file is created at 1:00 pm. At 1:05 the editor opens the file, some changes are made and the file is closed at 1:10 pm. What are the create, update and access time stamps for the file?

3-7. Directory /PROGRAMS has protection "rw/r" and the owner is SYSTEM. Can you as a general user:

a. Copy a file from /PROGRAMS to your own directory?

b. Copy a file from your directory into /PROGRAMS?

c. Run a program contained in /PROGRAMS?

d. Edit a file in /PROGRAMS?

3-8. What do the following CI commands do?

a. dl /

b. co 1 temp.txt (what terminates this command?)

b. pu /mary/@.@

c. unpu memos/@.@.c-830812

d. rn /henry/@.text /henry/@.txt

e. mo /elmer/programs.dir.d /george/programs/@.@

f. co @b@.---.s /b_files/@.@

g. dl /doris/@.@.os

h. dl @swap@.run.e

i. co /lester/@pr-.-----.pxsc83u8306-830615 @.ohno

3-9. What is the difference between the following spooling system commands?

```
- sp on 6
Spooling started from LU 6 to OUTSPOOL22.SPL::SPOOL

- sp on 6 outspool22.spl::spool
Spooling started from LU 6 to OUTSPOOL22.SPL::SPOOL
```

3-10. What does the following command do?

```
CI> co henry/boggle.run>15 /george/@.@>15[george]
```

3-11. What is a FMGR cartridge? If a CL command produced the following output:

```
File System disc LUs: 17  19
FMGR Disc LUs (CRN):  16(16) 18(AL)
```

How would you get a list of the files on LU 18? What are two ways to specify the "crn" in the following command if %quark is on LU 18?

```
CI> dl %quark::crn
```

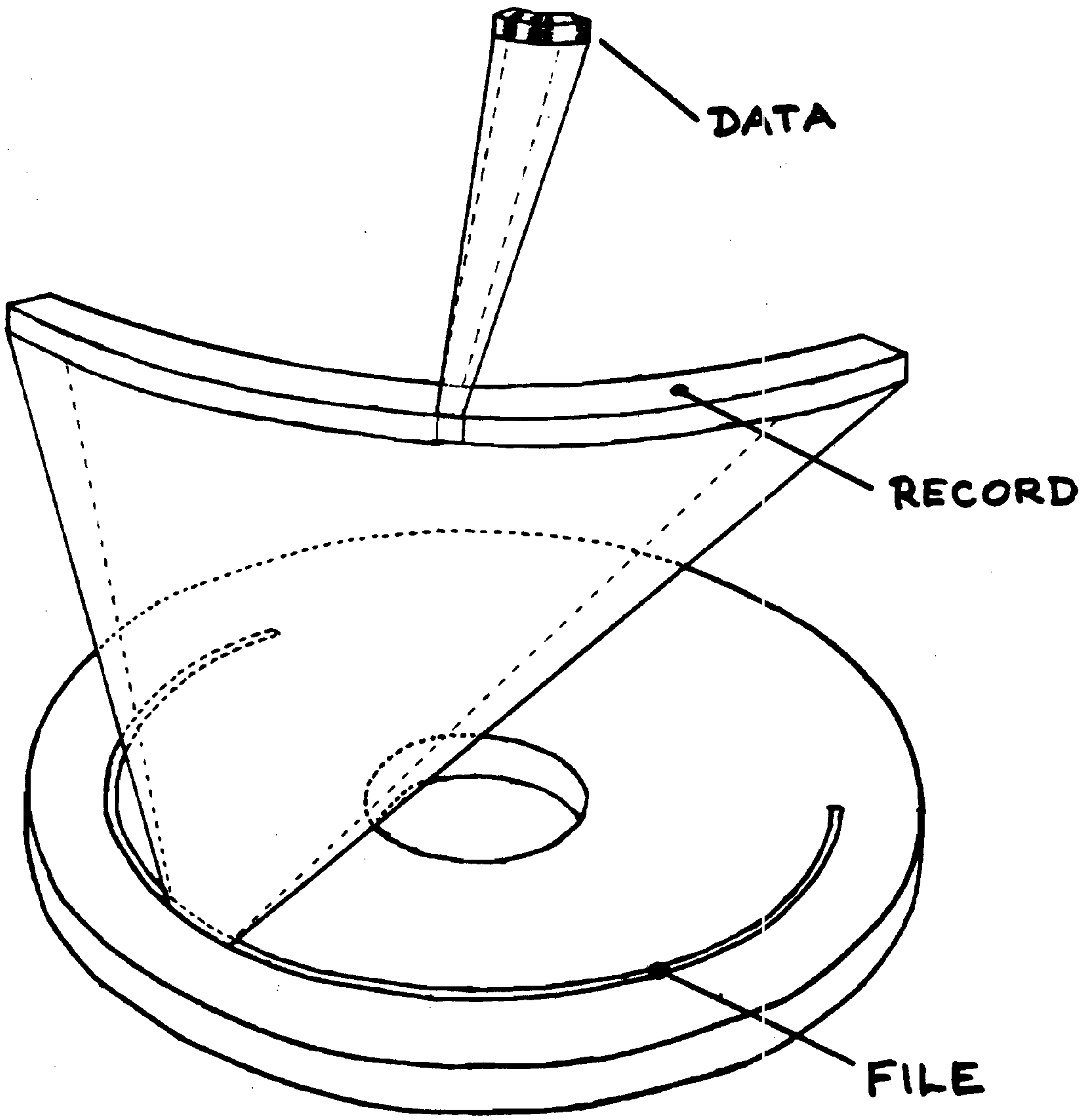
3.1 The Disc File

Data -- Basic elements of information such as digits in an address, characters in a Pascal keyword, or bits in a memory image file.

Record -- Logical grouping of data such as one line in a text file or a standard size "chunk" of a memory image file.

File -- Is a named collection of records on disc and is the means by which a program stores data for use at some later time.

THE DISC FILE

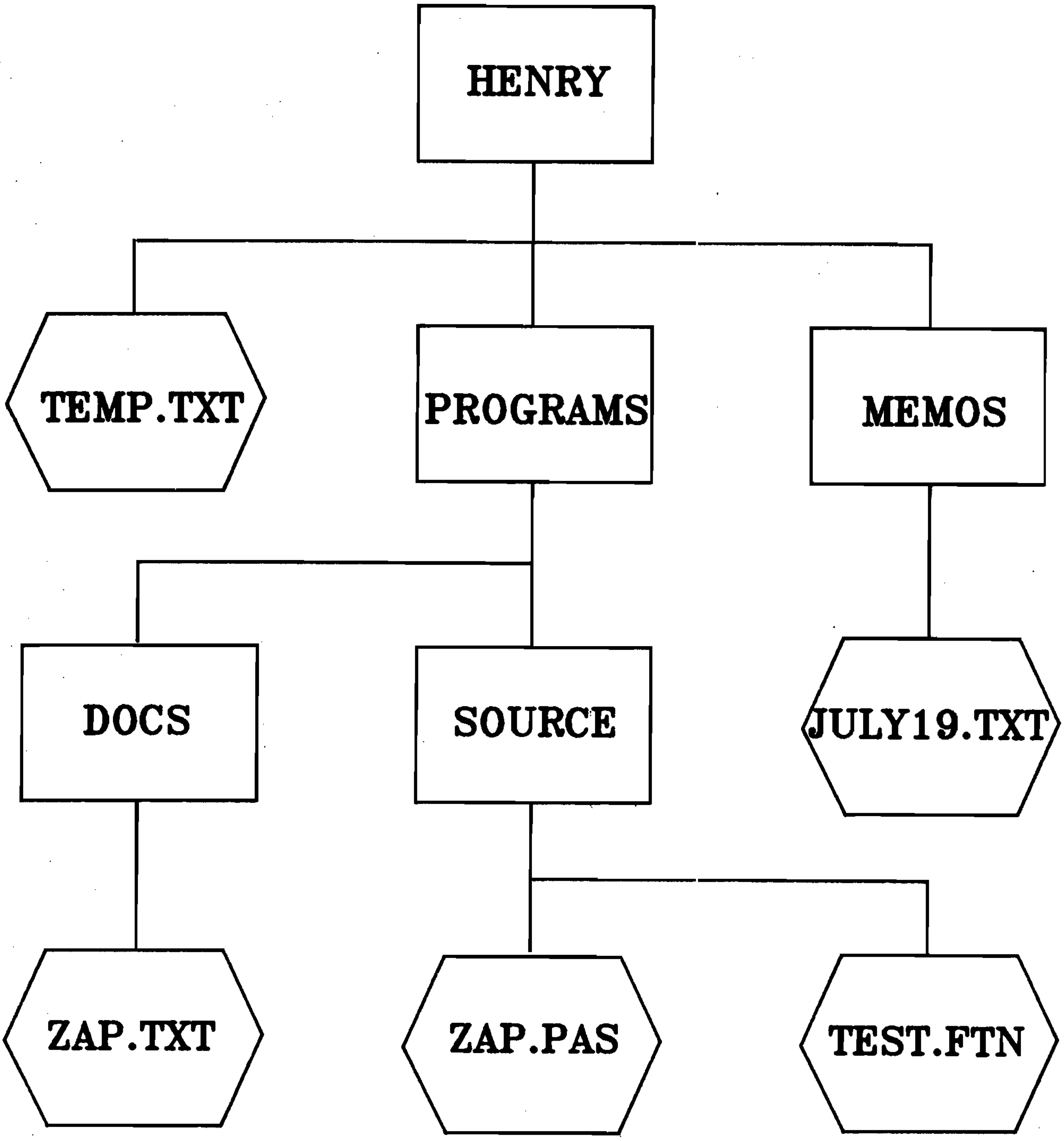


3.2 Directories

Hierarchical Structure -- Allows a directory to have entries for sub-directories as well as files. "Directory" is the generic name for either global or sub-directories.

References: System Design Manual

DIRECTORIES



3.3 Global Directories

Disc Volume -- A logically independent disc volume. This may be a flexible disc, a part of a hard disc, or an entire hard disc unit. Each volume is treated as a separate logical unit (LU).

Volume Header -- Contains information on disc free space and a pointer to the root directory. The volume header is the only information on a disc volume that is a fixed size and has a fixed location (last track of volume).

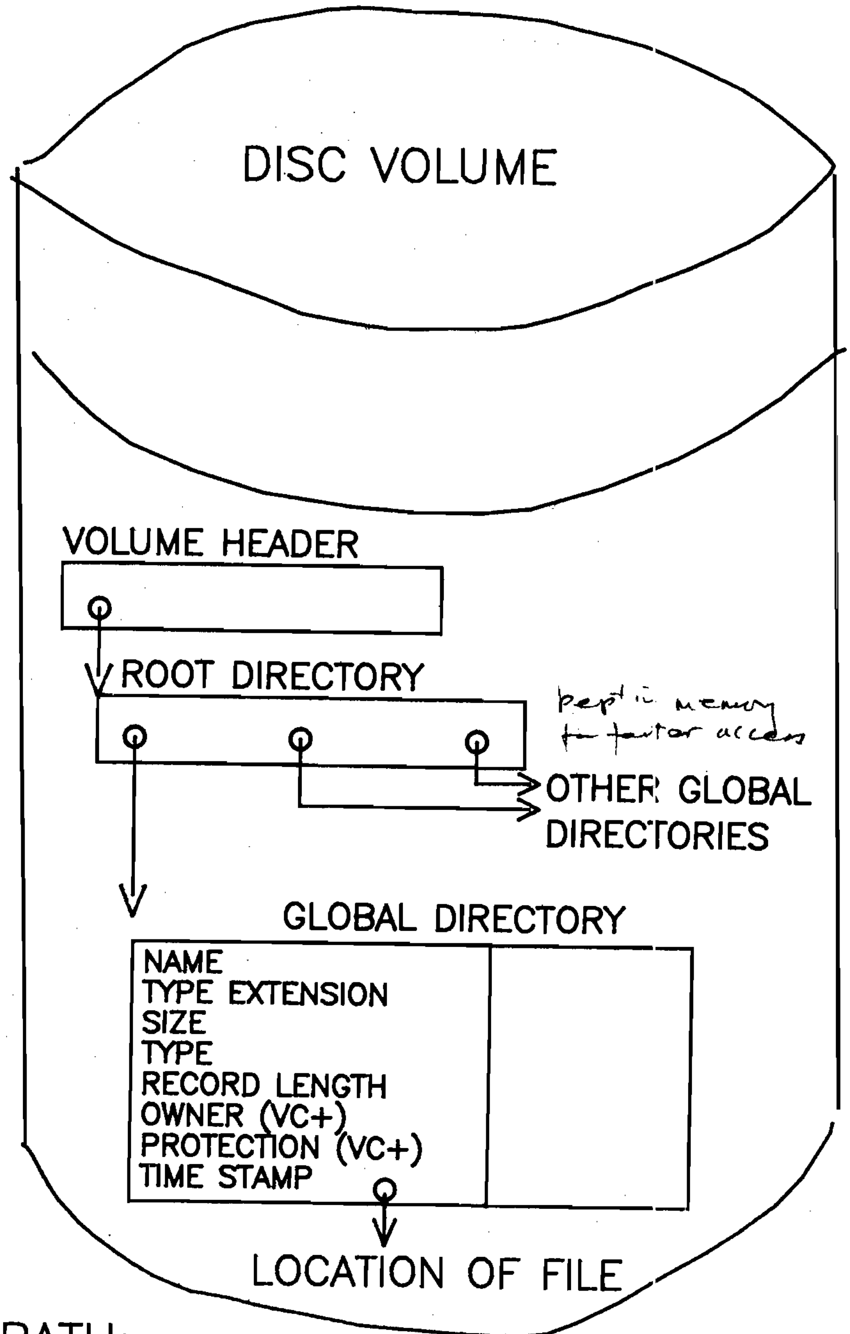
Root Directory -- Contains the name and location of each global directory on a volume. An unlimited number of global directories are allowed although all global directory names must be unique across volumes.

Global Directory -- Contains information about an unlimited number of files. The information includes the file name, various file properties, and the file's location on the disc. Global directories may also contain information about other directories.

Path -- Files are specified by the path the system must take to find them. The path starts at the root directory which is indicated by a leading slash.

References: System Design Manual

GLOBAL DIRECTORIES



PATH:

/GlobalDirectory/File

- / — SPECIFIES ROOT DIRECTORY
- GlobalDirectory — GLOBAL DIRECTORY NAME
- File — FILENAME

3.4 Sub-directories

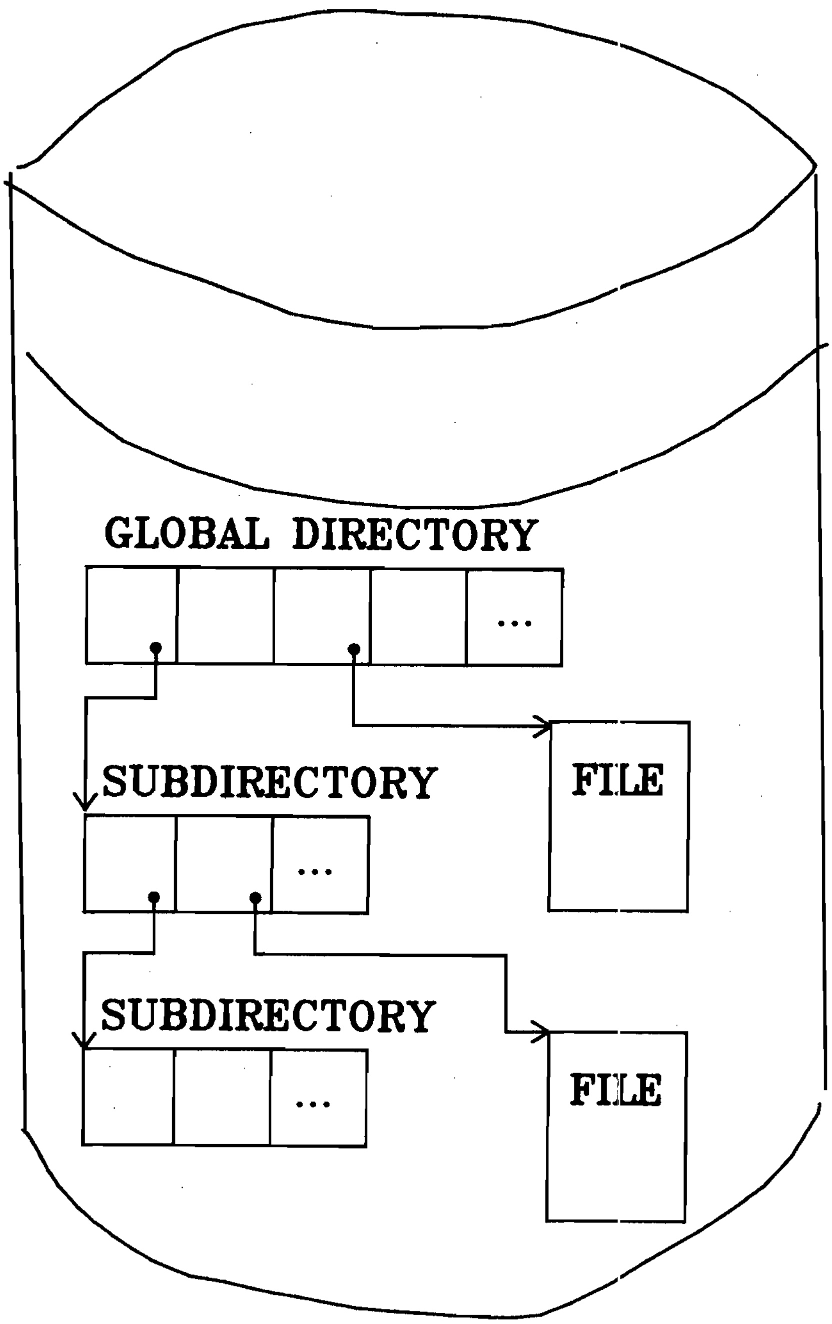
Global Directory -- The only real distinction to global directories is that they are at the top of the directory path.

Sub-directory -- These contain identical information to global directories, which may include information about still lower sub-directories.

Path -- May contain any number of directories but is limited to 64 characters in the path name including slashes, colons and file descriptor information.

References: System Design Manual

SUB-DIRECTORIES



PATH:

/GLOBALDIRECTORY/SUBDIRECTORY/FILE
/DIRECTORY/DIRECTORY/DIRECTORY/...
← $\leq$ 64 →

3.5 The Hierarchical Structure

SYSTEM

System Table -- Called the cartridge directory keeps track of "mounted" disc volumes. Mounted volumes are available for use. Unmounted volumes must be mounted with the MC command before they are accessible:

```
CI> mc 23          (mounts the volume with LU 23)
```

Root Directories -- Contain the names of all global directories whose names must be unique in the system.

VISIBLE

Global Directories -- This is the highest level in the file structure that can be manipulated by the user. All global directories must have unique names.

Sub-directories -- Can be nested to any depth. Sub-directory names may be the same as global directories names. Sub-directories contained in different parent directories may also use the same name.

Files -- Can be found at any level below the global directories. File names must only be unique within their parent directory.

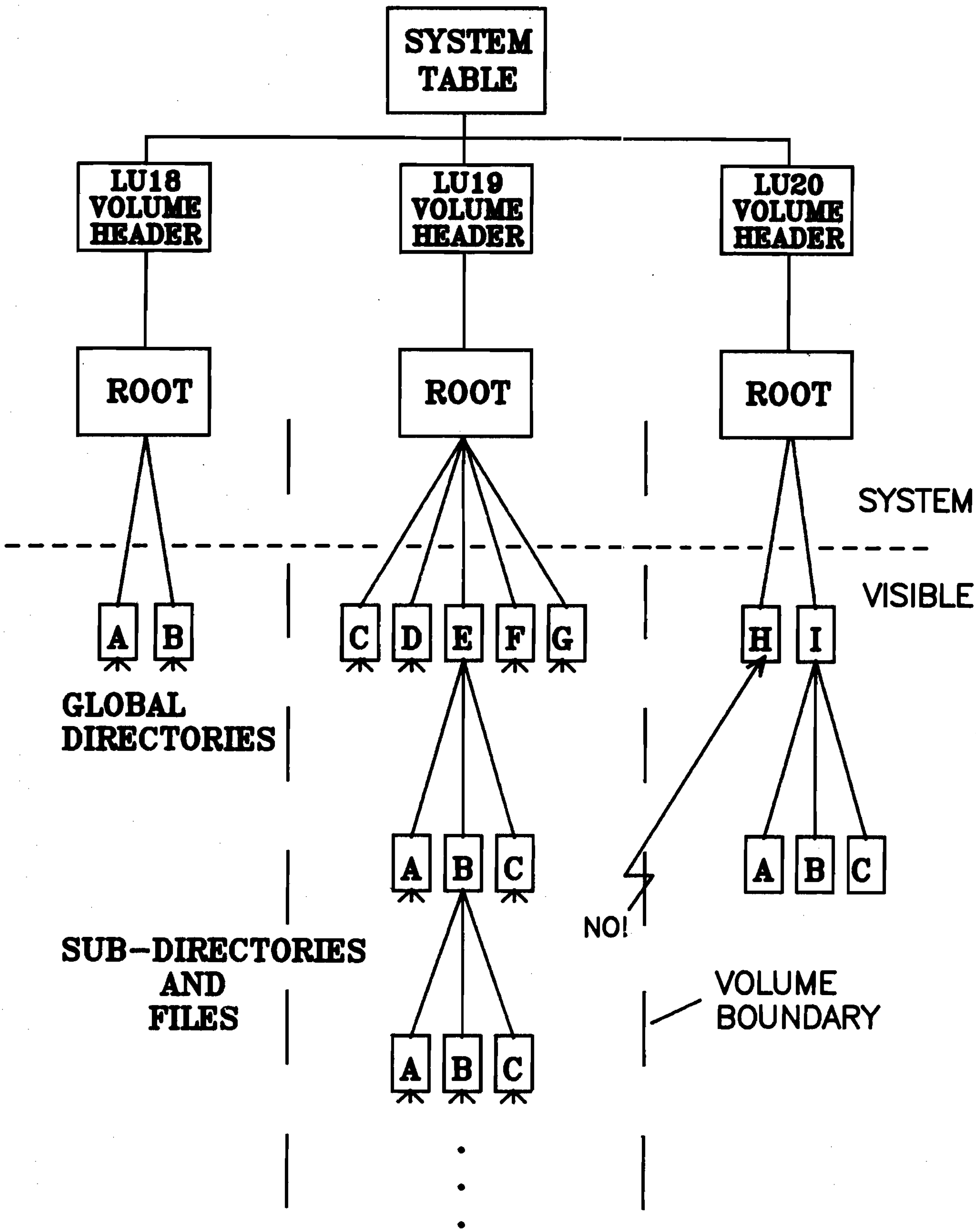
Volume Boundary -- the file system hierarchy may not cross the volume boundary. This has effects on two types of commands:

* Create Commands -- Files and directories will always be created on the same volume as their parent directory.

* Move Command -- Files may not be moved across volume boundaries since a move changes only directory information and does not physically move data. The copy command must be used to cross volume boundaries.

References: User's Manual

HIERARCHICAL STRUCTURE



3.6 Creating Directories

References: User's Manual

CREATING DIRECTORIES

crdir <name> <lu>

CI> crdir /mike

CI> crdir /sale 23

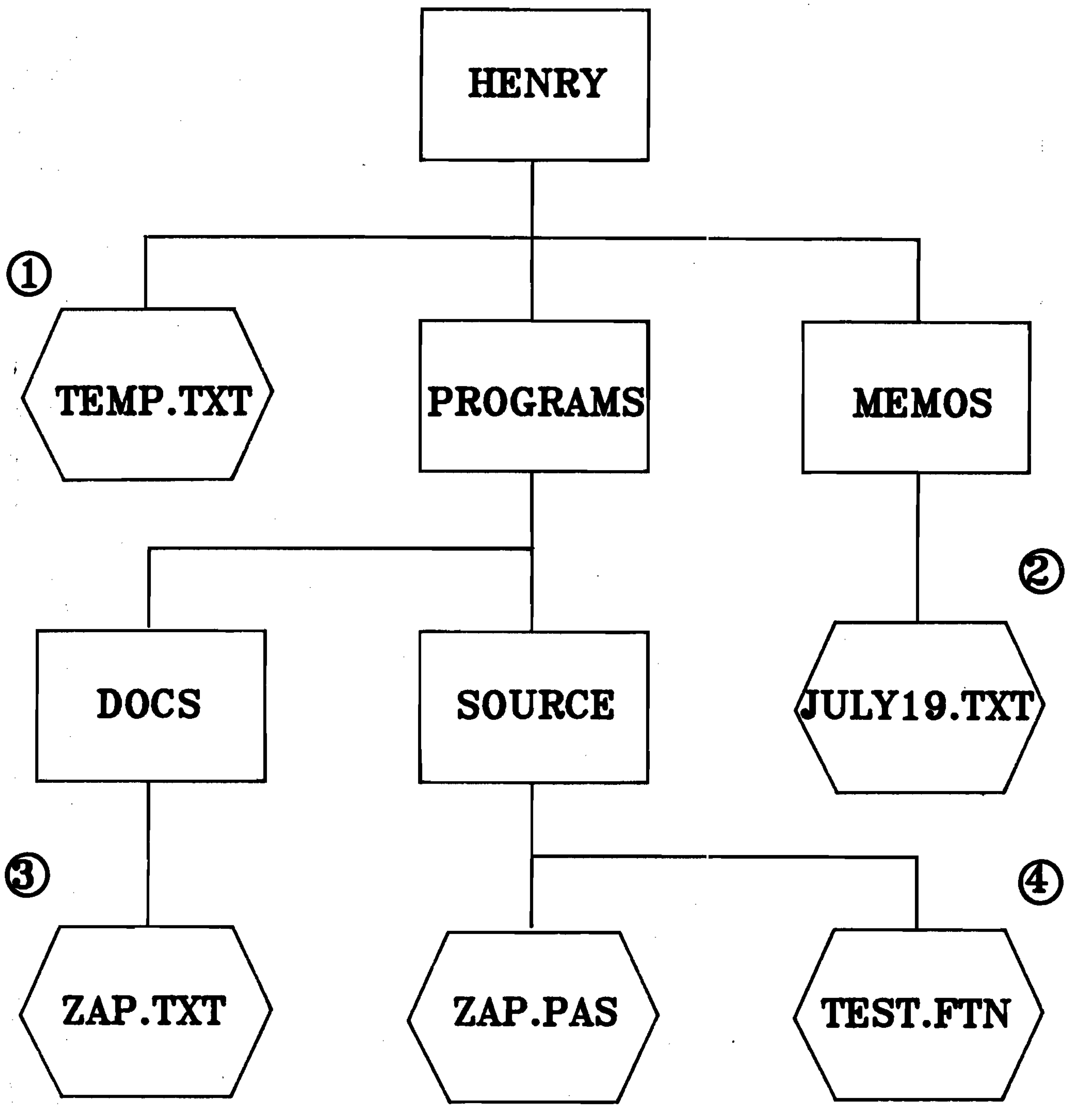
CI> crdir /sale/prices

3.7 Path Specification

Fill in 2, 3 and 4.

References: User's Manual

PATH SPECIFICATIONS



① /HENRY/TEMP.TXT

② /

③ /

④ /

3.8 Working Directory

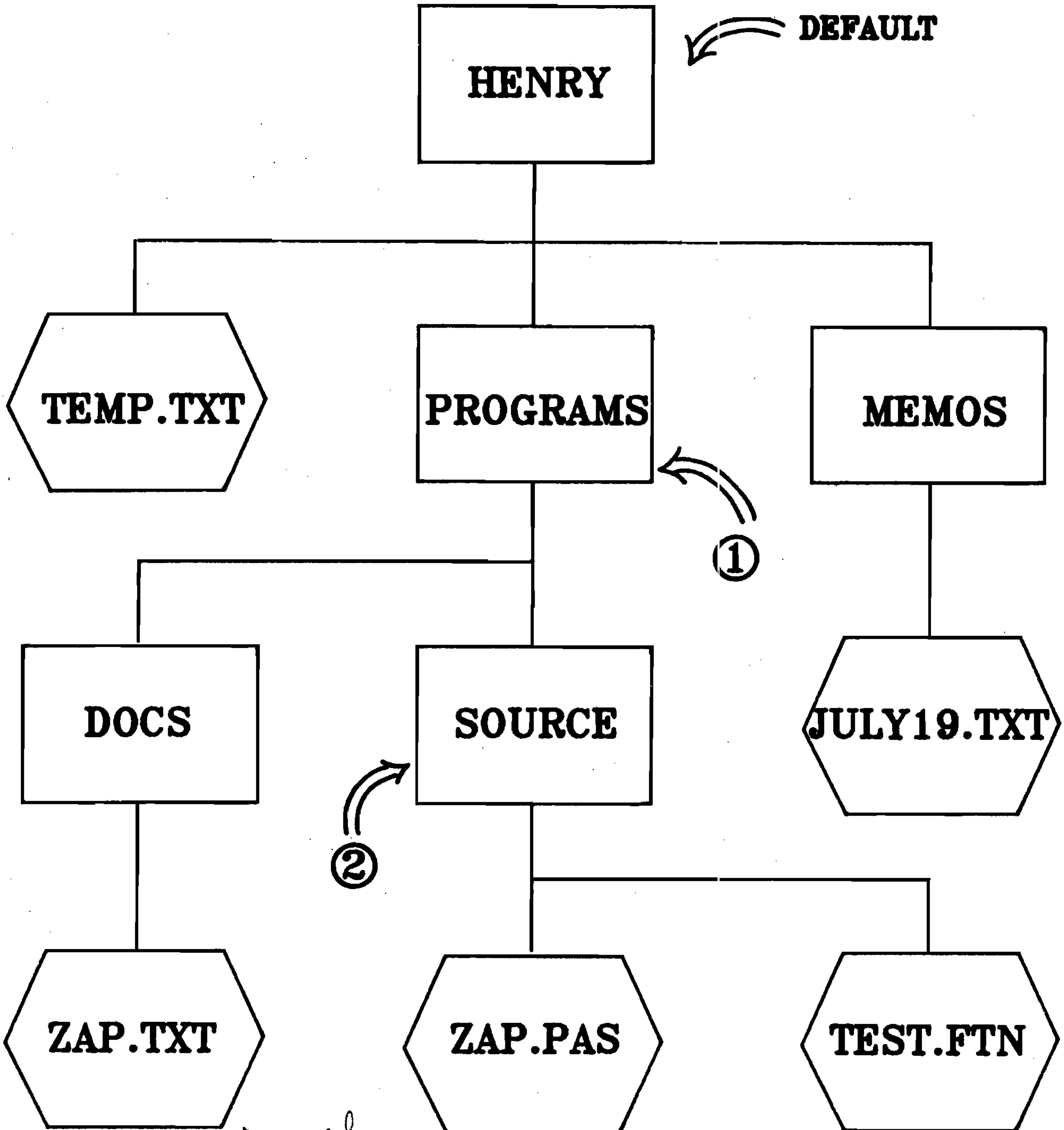
Typically, your working directory at logon is the global directory by your logon name.

Working directories are specified by their path names with the WD command.

Fill in 2.

References: User's Manual

WORKING DIRECTORY



root global

① CI> WD/HENRY/PROGRAMS
OR
CI> WD PROGRAMS

② CI> WD/
OR
CI> WD

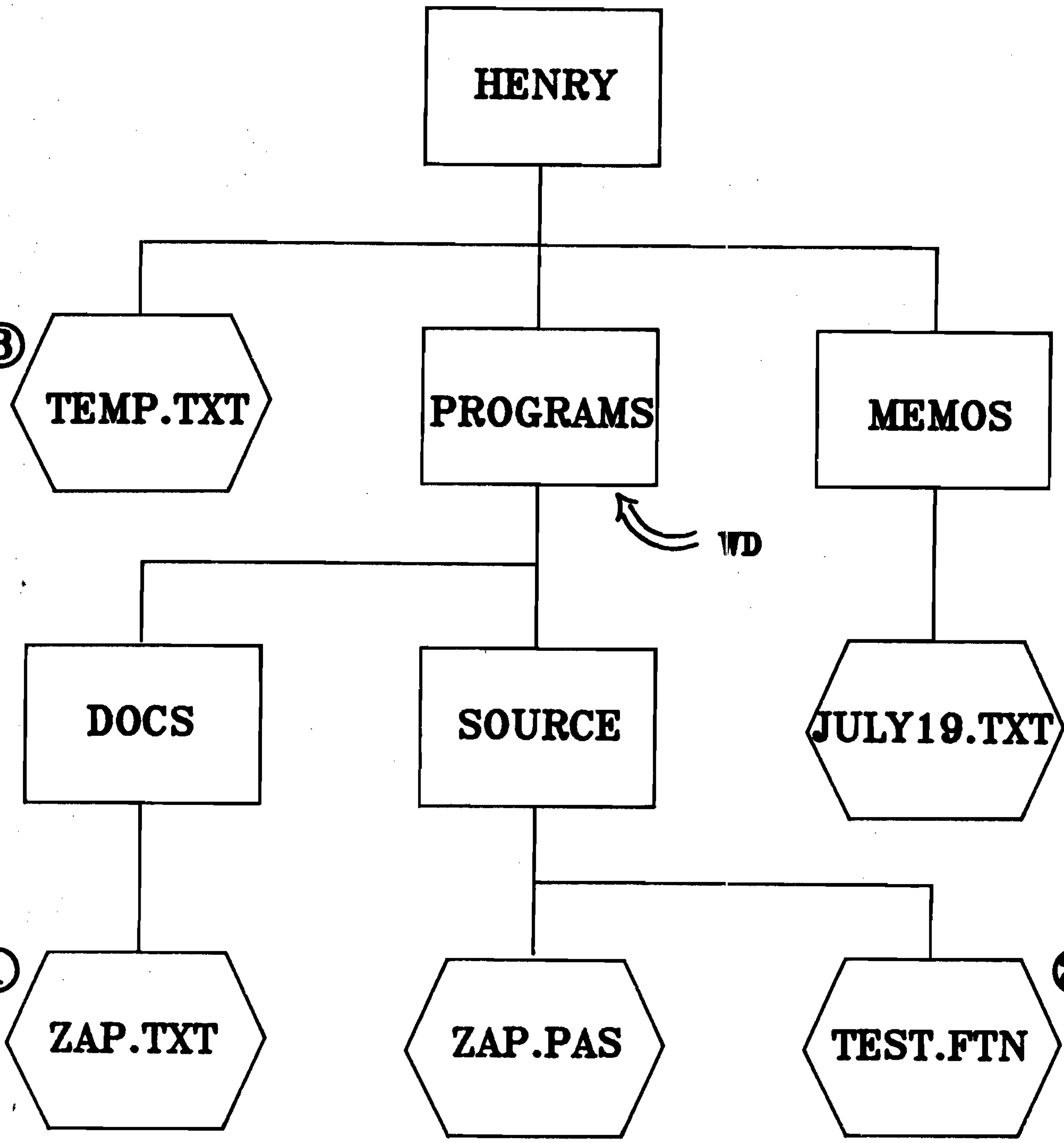
3.9 Using the Working Directory

Note for UNIX hacks:

There is no way to specify the parent directory in a path name as UNIX does with "..".

References: User's Manual

USING THE WORKING DIRECTORIES



① DOCS/ZAP.TXT

② source / Test.Ftn

③ / Henry / Temp.Txt.

3.10 U S I N G F I L E S

USING FILES

u
n
p
u

rn

MO

cl

DL

CR

owner

crdir

prot

pu

WD

li

3.11 Filenames

Restricted Characters -- These have special meaning to CI. Other punctuation may be used in filenames although HP "officially" recommends using only alpha characters and numbers. This allows HP to use other punctuation for something special in the future.

Filename -- 16 characters allow you to use meaningful filenames.

Type Extension -- Provides standard type classifications to aid readability. Also provides protection with those programs that will not accept a file with the wrong type extension.

Standard type extensions --

.cmd	CI command file
.cop	compiler options
.dbg	debug file
.dat	data file
.dir	directory
.ftn	FORTRAN source file
.ftni	FORTRAN include file
.lib	library of relocatable files
.lod	LINK command file
.lst	listing from compiler
.mac	Macro source file
.maci	Macro include file
.map	loader map listing
.pas	Pascal source file
.pasi	Pascal include file
.rel	relocatable file
.run	runnable program
.snp	system snapshot file
.spl	spooling system file
.txt	text file
.sys	system file

References: User's Manual

FILENAMES

RESTRICTED CHARACTERS:

/ : . @ - [>

FILENAME:

A_LONG_FILE_NAME

alpha

$\leq$ 16 CHARACTERS

TYPE EXTENSION:

FILE_NAME.TYPE

SEPARATOR — $\leq$ 4 CHARACTERS

STANDARD TYPE EXTENSIONS:

.CMD	COMMAND FILE	**.PAS**	PASCAL
.DIR	DIRECTORY	**.REL**	RELOCATABLE
.FTN	FORTRAN	**.RUN**	PROGRAM
.LST	LISTING	**.TXT**	TEXT

3.12 File Attributes

Type -- Not to be confused with type extension.

Other types available are:

0 -- used with programmatic file calls to access an I/O device as a file.

7 -- absolute binary files.

8 thru 32767 -- user defined.

The F option for the DL command will list the file type for the specified files.

Size -- Space is allocated on disc whether its used or not. If more space is subsequently required, extents are allocated automatically.

If more than 16383 blocks are required, space is allocated in 128 block "chunks". The number of chunks is specified by a negative size parameter (e.g., -130 = 16640 blocks).

The S option for the DL command will list the size of the files.

Record Length -- Automatic record lengths are computed for the following type files:

1 -- 128 words

3 and above -- variable record length.

The R option for the DL command will list the record length. For type 3 and above files, the length of the longest record in the file will be listed.

References: User's Manual

FILE ATTRIBUTES

TYPE

1	RANDOM ACCESS 128 WORD RECORD LENGTH
2	RANDOM ACCESS FIXED RECORD LENGTH
3,4	TEXT FILES VARIABLE RECORD LENGTH
5	RELOCATABLE CODE
6	RUNNABLE PROGRAMS

SIZE

IN BLOCKS, 1 BLOCK =128 WORDS

RANGE: 1 TO 16383 BLOCKS
DEFAULT: 24 BLOCKS

RECORD LENGTH

IN WORDS, FOR TYPE 2 FILES ONLY
ALL OTHER TYPES ARE AUTOMATIC

3.13 The File Descriptor

Hierarchical Format -- Preferred method of specifying files.

Combined Format -- Is used sometimes by the file system so programmatic calls can be made from either CI or FMGR based programs. The output from the DL command is a good example.

FMGR Format -- Included here for historical perspective. "sc" refers to security code. "crn" refers to cartridge reference number, which is similar in function to a global directory.

References: User's Manual

THE FILE DESCRIPTOR

HIERARCHICAL FORMAT:

/GLOBAL/SUB/FILE:::TYPE:SIZE:RECLEN

optional

COMBINED FORMAT:

SUB/FILE::GLOBAL:TYPE:SIZE:RECLEN

FMGR FORMAT:

FILE:SC:CRN:TYPE:SIZE:RECLEN

optional

3.14 File Commands

CR -- Create a file. <name> may include full file-descriptor information.

PU -- Purge a file. If <name> includes file-descriptor information, the file will be purged only if all fields match.

UNPU -- Unpurge a file. Purged files are not really deleted but only flagged as being purged. If the space has not yet been reclaimed, the file may be unpurged. There is no guarantee as to how long a purged file may exist and still be unpurgeable.

RN -- Rename a file.

CO -- Copy a file. A copy of <source> file is made under the <destination> name. The source file remains unchanged. The copy command will not overwrite an existing file unless the "D" option is included. If the "P" option is specified, the <source> file will be purged after a successful copy. The copy command does not verify.

MO -- Move a file. The move command changes only directory information for the file and is therefore faster than copy. The only restriction is that files may not be moved across LU boundaries.

References: User's Manual

FILE COMMANDS

cr <name>

pu <name>

unpu <name>

rn <old name> <new name>

co <source> <destination> [DP]

mo <source> <destination>

3.15 Time Stamps

All files are time-stamped, directories are not. Times stamps include date and time to 1 second resolution. Times are posted at:

Creation -- When file is created programmatically, with the CR (create) command or with the CO (copy) command.

Update -- Posted when the file is closed after being changed. Also posted at creation. This is the only time stamp that is not posted during a copy operation

Access -- Posted when the file is opened for reading or update. Also posted at creation. This time stamp is not effected by examining file attributes in the directory such as size, type, protection, etc.

References: User's Manual

TIME STAMPS

CREATED
JUN 28 1983
3:05:21 PM

UPDATED
JUL 7 1983
10:18:52 PM

ACCESSED
JUL 19 1983
9:24:14 AM

3.16 Protection (VC+)

Protection applies to all general users; superusers override all file protection.

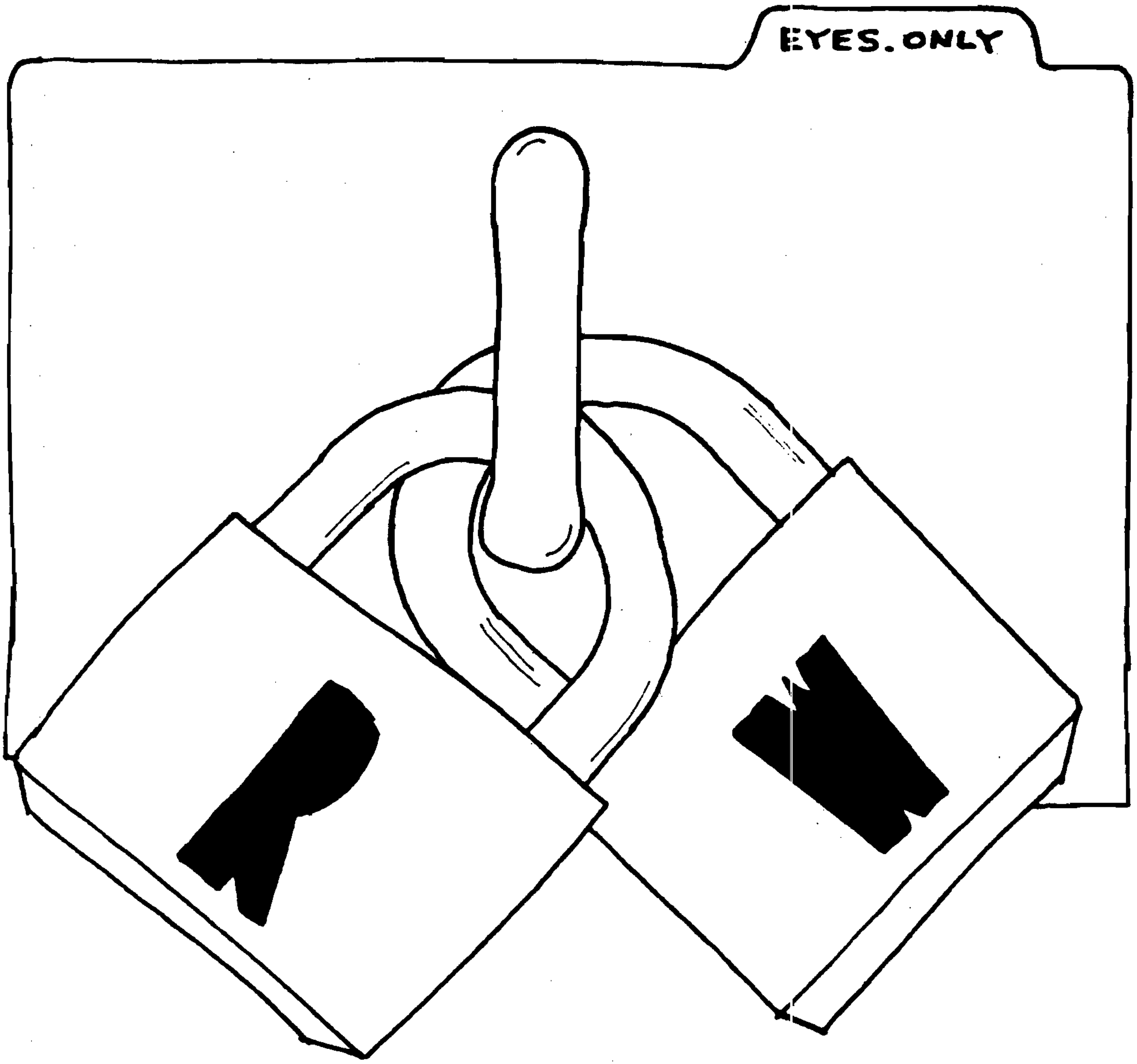
To change protection, the user must own the file (explained on the next slide).

Examples:

```
CI> prot eyes.only rw/rw   -- everyone has access

CI> prot eyes.only rw/r    -- owner has full access
                              others can only read

CI> prot eyes.only r/      -- owner can read
                              no one can write

CI> prot eyes.only /rw     -- owner cannot access
                              others can read or write

CI> prot eyes.only /       -- no one can read or write
                              only superusers can access

CI> prot eyes.only         -- lists current protection
    directory ::HENRY
        name    prot

    EYES.ONLY   rw/r
```

References: User's Manual

PROTECTION (VC+)



CI> prot eyes.only rw/rw

owner
other users

r = read access

w = write access

3.17 Directory Ownership (VC+)

The owner of a directory has two privileges:

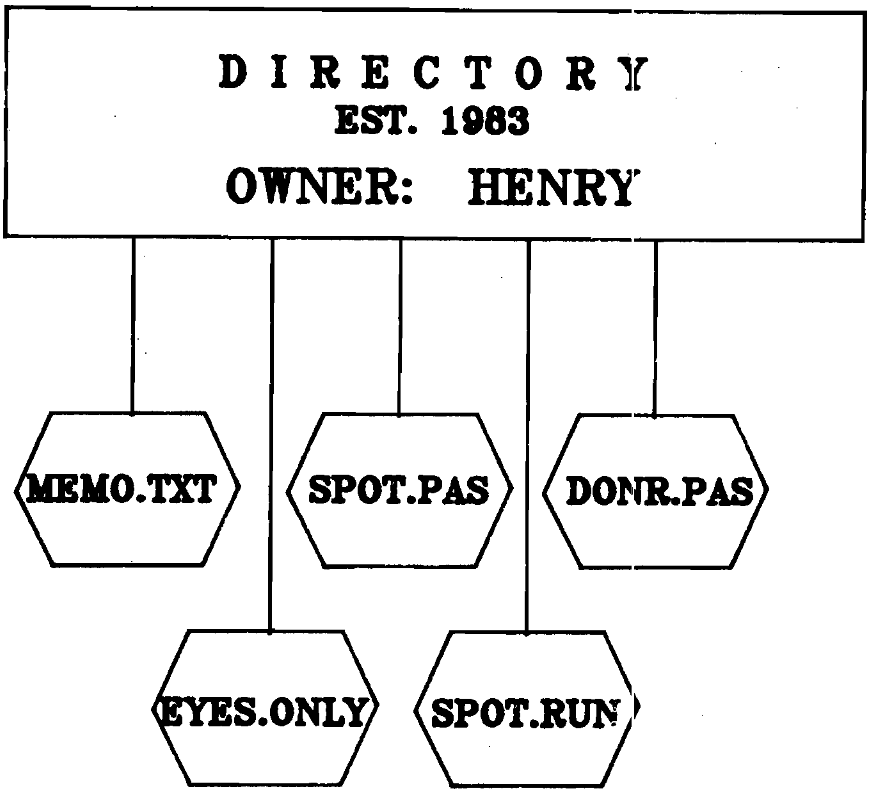
1. Access to files is allowed as assigned by the owner field of the protection command.

2. The owner may transfer ownership to another user. Note, however, that all privileges are transferred to the new owner and are then unavailable to the previous owner.

A sub-directory may be owned by someone other than it's parent directory.

References: User's Manual

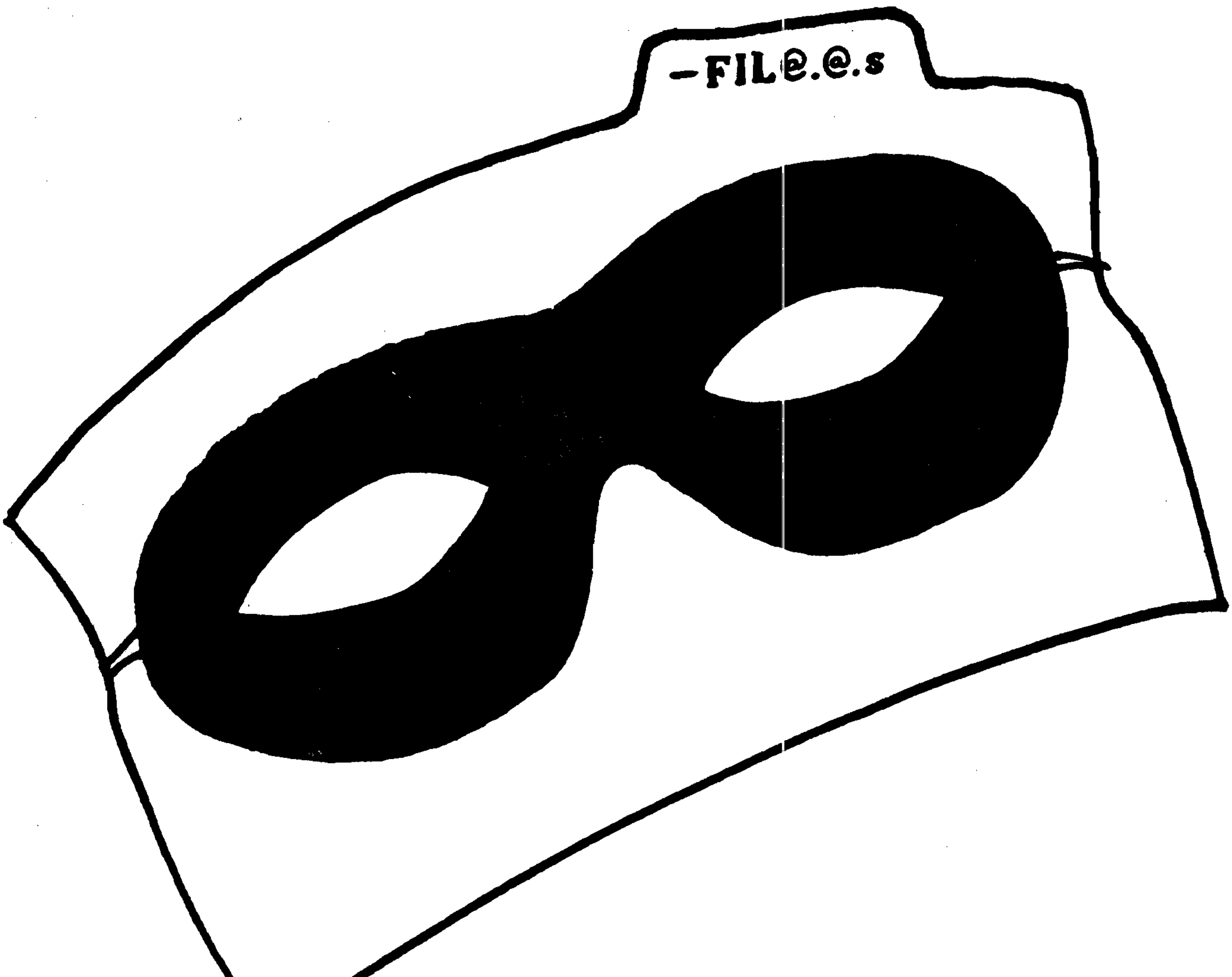
DIRECTORY OWNERSHIP (VC+)



CI> Owner /Henry William
CI> Owner /Henry
Owner of /HENRY is WILLIAM

3.18 FILE MASKING

FILE MASKING



3.19 Wildcard Characters

"-" -- Will not match the "." between the file name and type extension. For example, "joker-wild" will not match "joker.wild".

"@" -- Does not match the "." either but... when the mask ends with "@", the system assumes you meant "@.@" (unless you have explicitly used "." somewhere in the mask). For example, "jok@" will translate to "jok@.@" and match "joker.wild" as well as "joker" (blank type extension).

Note that the file system will make some default assumptions about the mask intended in certain circumstances. For example:

When the mask is:	The file system will assume:	
/name/	/name/@.@	
/name/@	/name/@.@	
/name/@.	/name/@.	(blank type extension)

References: User's Manual

WILDCARD CHARACTERS

botch jo job joke joker joker.wild

– MATCHES ANY NON–BLANK CHARACTER

```
jo–      ——>job
jo––     ——>joke
–––––    ——>botch joker
```

@ MATCHES ZERO OR MORE CHARACTERS

```
jo@      ——>jo job joke joker joker.wild
jok@     ——>joke joker joker.wild
jok@.    ——>joke joker

–0@      ——>
@b@      ——>
@.wild   ——>All files with wild type extension
@.-@     ——>All files with non–blank type extension
```

3.20 Mask Qualifier

File Characteristics -- Files that match the name and type extension mask are then selected to meet the qualifier specification. The "b" qualifier refers to files that have their backup flag set. This is used by the TF utility during incremental backup. The "t" qualifier refers to files that were flagged as temporary when they were opened.

Search Directives -- The system normally searches through only the directory specified in the file descriptor (or the working directory if none was specified). The search directives expand the search to include additional directories. The "d" directive selects all files whose directory matches the mask.

Time stamps -- Files may be selected whose time stamp falls within a range of dates. This may be done with either the creation date, access date, or update date.

Examples:

* List all files created during the first six months of 1983:
  CI> dl @.@.c830101-830630

* List all FORTRAN source files updated since July 15, 1983:
  CI> dl @.ftn.u830716-

* Purge all files not accessed since December 31, 1982:
  CI> pu @.@.a-821231

References: User's Manual

MASK QUALIFIER

<filename>.<ext>.<qualifier>
/HENRY/@.@.S

FILE CHARACTERISTICS:

- **B – SELECT FILES THAT HAVE BACKUP BIT SET**
- **O – SELECT OPEN FILES**
- **P – SELECT PURGED FILES**
- **T – SELECT TEMPORARY FILES**
- **X – SELECT FILES WITH EXTENTS**

SEARCH DIRECTIVES:

- **S – SEARCH DIRECTORY AND IT'S SUB-DIRECTORIES**
- 4 **E – SEARCH EVERYWHERE**
- **D – DIRECTORY MATCH** Default
- **N – NEGATE DIRECTORY MATCH** must use in directory copying to stop recursive copying

TIME STAMPS:

- **C, A, U – SELECT BY TIME STAMP**

3.21 Destination Masks

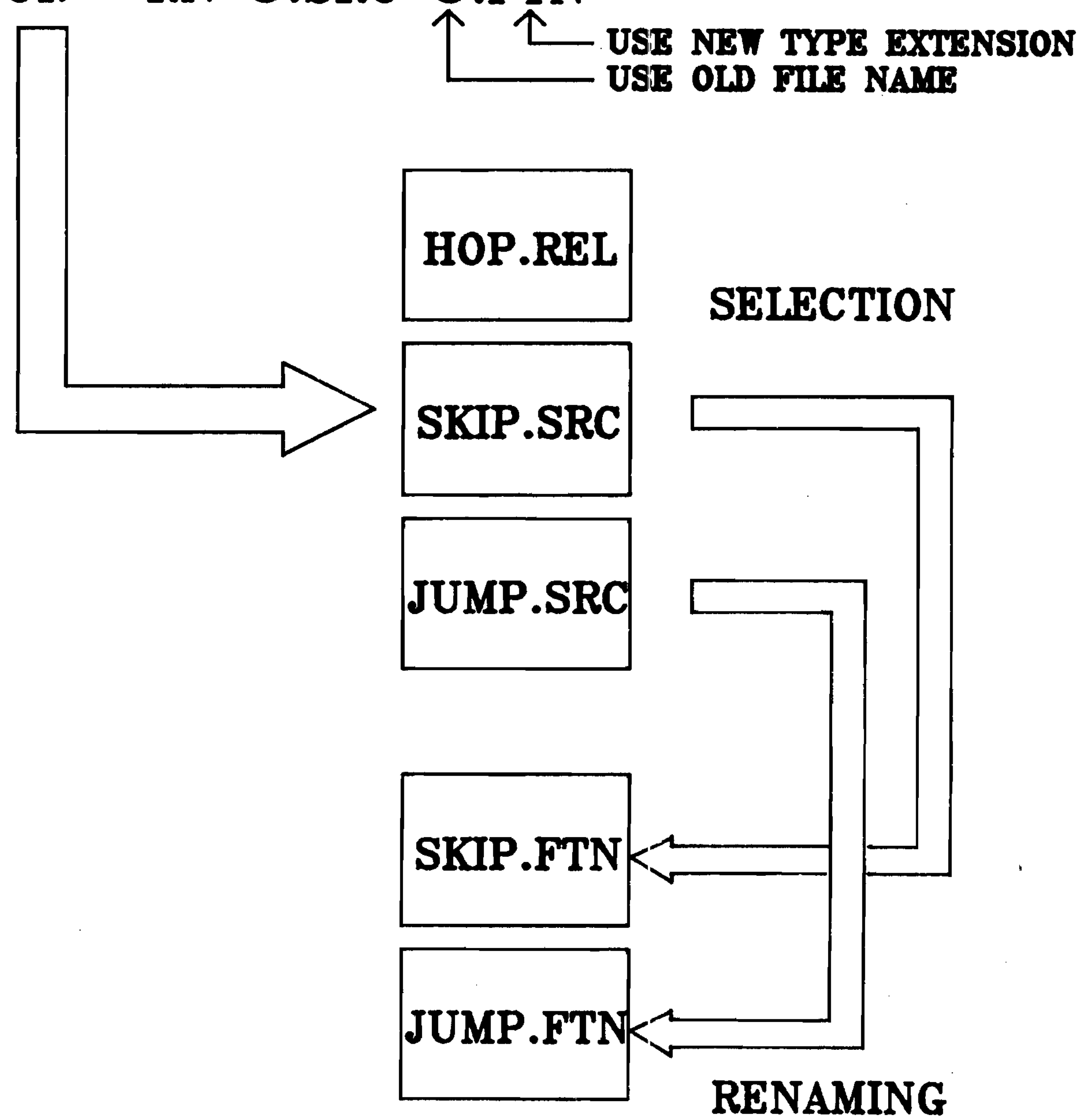
If the destination mask uses "@" for the filename or type extension, the source filename or type extension is used for the destination filename. Otherwise, the destination mask defines the filename or type extension.

References: User's Manual

DESTINATION MASKS

**@ MASKS ENTIRE FILE NAME
OR ENTIRE TYPE EXTENSION**

CI> RN @.SRC @.FTN



R3.19

3.22 Mask Examples

What do these commands do?

References: User's Manual

MASK EXAMPLES

```
CI> MO MAIN.FTN SUB1.@
CI> DL @.TXT.S
CI> PU @.PAS.U-82
CI> RN P---.@ @.TXT
CI> DL PROGRAMS.@.D
CI> UNPU @.@.E
CI> DL /WILLIAM.@.X
CI> CO @.@.N SUBDIR/@.@
```

3.23 Owner Accounting (VC+)

FOWN -- Displays the total disc space owned by each user for the files specified by <mask>. The default is for all files in the system.

<mask> -- Specifies which files to consider for ownership accounting. In general, the mask "/<global.dir>/@.@.s" will provide ownership accounting for all files under global directory <global.dir>.

SYSTEM -- Are those files created under a non-VC+ system and have no owner.

Unknown -- Files whose owners have been removed from the system or files accessed through DS/1000 remote file access.

FMGR Files -- Old file system files have no owner and therefore are not scanned.

References: User's Manual

OWNER ACCOUNTING (VC+)

FOWN <mask>

```
CI> FOWN
Scanning...                    Mask = /@.@.S
    Owner                        Disc Blocks
SYSTEM                                 9041
MANAGER                               62594
WILLIAM                                6115
SHELLY                                  585
LESLIE                                 7685
Unknown (#10)                            96
                                    -------
Total                                 86114

FMGR files not scanned
```

```
CI> FOWN /WILLIAM/@.@.S
Scanning...         Mask = /WILLIAM/@.@.S
    Owner                        Disc Blocks
WILLIAM                                4126
MANAGER                                  82
                                    -------
Total                                  4208
```

3.24 USING THE FILE SYSTEM

USING THE FILE SYSTEM

3.25 Device Status

I/O Command -- Shows assignment of logical unit numbers and the status of each associated device. The list of LUs may be up to 255 long. The first and last LU of interest (e.g., 6 and 10) may be included in the I/O command to limit the amount of output.

LU -- The logical unit number of the device.

Device Name -- Name assigned to the device during system generation.

Select Code -- This is a physical switch setting on the I/O card and must match the number shown here which was defined at system generation.

HPIB Address -- This is a physical switch setting on the device and must match the number shown here which was defined at system generation.

Device Status -- This is the current status of the device as found in a device table associated with it's LU. Typical statuses are:

* up -- The device is ready for I/O.

* down -- The device is not responding to system requests because either it is off-line (as a printer might be when it runs out of paper) or it may have a hardware problem.

* Locked to <program> -- The device has been locked for exclusive use by the named program.

* Busy with class request -- Often seen when a program (such as CI) is waiting for input from a terminal.

References: User's Manual

DEVICE STATUS

CI> IO 6 10

LU	Device Name	Select Code	HPIB Address	Device Status
6	Printer	30	6	Up
7	Not Assigned			
8	Tape Drive	27	4	Locked to TF
9	Instrument	27	36	Down
10	Floppy Disc	27	5	Up

3.26 Commands Using LUs

DL Command -- The third parameter is the LU to which the directory listing is sent. This normally defaults to LU 1.

CO Command -- Either the source or destination may be the LU of a device.

Bit Bucket -- LU 0 may be used to send data to a non-existant device when that data is not wanted.

References: User's Manual

COMMANDS USING LUs

```
CI> DL /WILLIAM/,,6
CI> CO MEMO.TXT 6
CI> CO 1 68
CI> FTN7X AREA.FTN 1 0
```

3.27 Spooling System (VC+)

SP Command -- The spooling system is a program that is initialized by the System Manager and runs under the system session. The SP command gives each user access to the spooling system. Each user may spool output to LUs or redirect output from one LU to another independent of other users. The SP command may also be given with a command parameter. The command will then execute and return immediately to CI, for example:

```
CI> sp st
    (spooling status here)
CI> _
```

ST Command -- Gives the status of the spooling system. LU redirection is shown for the user's session. This is independent of other user's redirection status. Spool file status shows all active spooling on the system. Each spool file is associated with an LU. An LU is currently being spooled for a particular user if "Owner" shows that user's logon name, "Spool LU" indicates the LU, and "File Status" shows "Actively spooling".

EX Command -- Exits interactive spool mode.

References: User's Manual

SPOOLING SYSTEM (VC+)

```
CI> SP
RTE-A Spooling System
Type ? for help
- ST

Redirection Status:

    6 =>68

Spool File Status:

    Maximum Spool Files = 21

Filename      = OUTSPOOL 22.SPL::SPOOL
Owner         = HENRY    Terminal LU = 68
Spool LU      = 8   File Status = Actively Spooling
Approximate Line Count   = 0

- EX

CI>
```

3.28 Output Spooling

ON Command -- The spooling system directs all output that would have gone to the specified LU to a file created by the spooling system. This file is kept in directory ::SPOOL and has a name created by the spooling system. Files in this directory must not be modified except through the spooling system to avoid confusing it. These files are read/write protected from the general user for this reason.

OF Command -- Terminates spooling to the spool file. The file is then put into a spooling queue for output to the specified LU, after which it is purged. The LU is locked by the spooling system during output so no other output to the LU can interfere.

LI Command -- The named file is put into a spooling queue for output to the specified LU. The LU is locked by the spooling system during output so no other output to the LU can interfere. The third parameter "nc" specifies no carriage control (a space is padded at the beginning of each record), which is normal for most text files.

References: User's Manual

OUTPUT SPOOLING (VC+)

```
on <lu>
of <lu>
li <file> <lu> n


- on 6
Spooling started from LU 6
   To OUTSPOOL22.SPL::SPOOL

- of 6
Spooling terminated from LU 6
   To OUTSPOOL22.SPL::SPOOL

- li area.pas 6 nc
File AREA.PAS::HENRY
   Queued for output to LU 6
```

3.29 LU Redirection

ON Command -- Subsequent output to the LU is redirected into the named file or LU. This does not protect the file or redirected LU from being written to by other programs.

OF Command -- Terminates redirection from the specified LU. If redirection was to a file, the file is closed and not written to the LU as with output spooling.

References: User's Manual

LU REDIRECTION (VC+)

on <lu> <file>

on <lu> <lu>

```
- on 8 MYFILE
Spooling started from LU 8
   To MYFILE::HENRY

- on 6 67
LU redirection started
   from LU 6 to LU 67

- of 6
Spooling terminated
   from LU 6 to LU 67
```

3.30 DS Transparency

RTE-A systems which use the DS/1000-IV Distributed Systems Network can access files which reside on other RTE-A systems within the network. Any command or program that specifies a file, directory or mask may be used with DS transparency (e.g., co, pu, dl, edit, link).

If the remote node is running with VC+ multiple sessions, a general user session will be created for you on the remote system (even if you are a super-user on your system). You then have the file access rights of a general user. You may optionally log on as a specific user (which may be a super-user) by specifying a logon name and password in the file descriptor. You will be logged off when the file you are accessing is closed.

Copy, purge, and directory list are typical commands utilizing DS transparency. Remote files may be used as the input to programs such as EDIT or the Pascal compiler.

The following are limitations of the DS system:

1. You cannot access I/O devices on a remote system (such as a printer or tape drive).

2. You cannot run a program on a remote system (although you may copy the file to your system and run it).

3. Working directories are not available on a remote system. You must specify full path names.

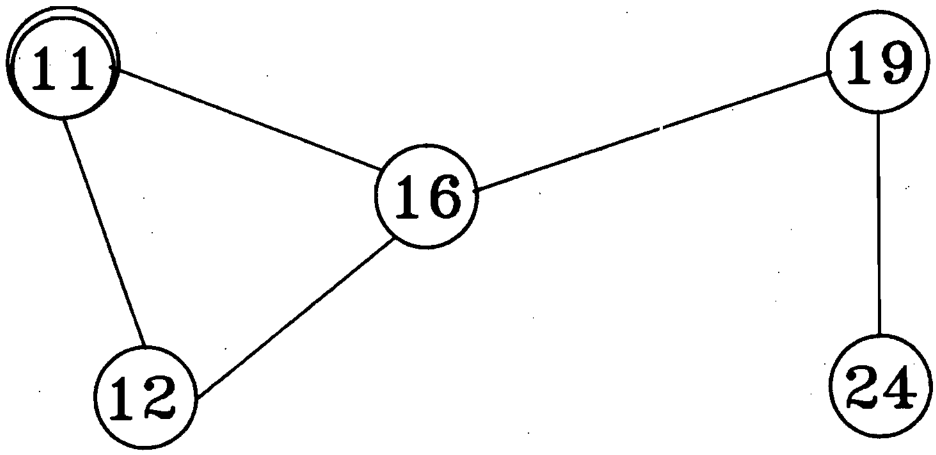
4. CI commands are not available through your remote logon.

5. Note that the last copy command, if issued from node 11, will transfer the file via the route:

   24 => 19 => 16 => 11 => 16 => 19

References: User's Manual

DS TRANSPARENCY



FILENAME>NODENAME [LOGON]

CO AREA.PAS /GEORGE/@>24

EDIT /GEORGE/MEMO12.TXT>24

DL /PROGRAMS>16 [MANAGER/ZYX]

CO /AMY/DOCS>24 /ALICE/DOCS>19

3.31 FMGR File System

Filename -- Up to 6 characters, must start with a letter, cannot use the characters : , - +, names with / are unusable in CI as are names with periods as the first or last character.

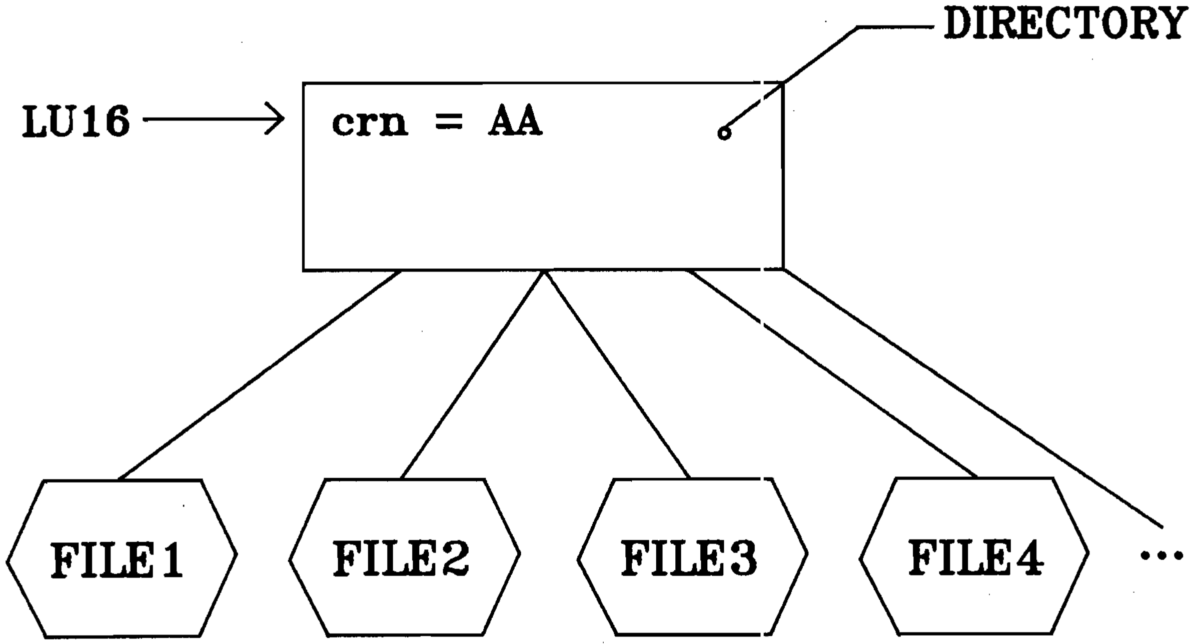
SC -- Security code, two characters or a decimal number in the range -32768 thru 32767. Zero specifies no security code.

CRN -- Cartridge Reference Number, two characters or a decimal number in the range -64 thru 32767. When the number is negative, it refers to the LU of the disc cartridge (volume).

Type, size, reclen -- Same as in CI.

References: Utilities Manual

FMGR FILE SYSTEM



filename:sc:crn:type:size:reclen

- ONE DIRECTORY PER LU
- 6 CHARACTER FILE NAMES
- NO TYPE EXTENSIONS, TIME STAMPS
- NO UNPURGE

3.32 Using FMGR

Subsystems -- Such as Image/1000 or Graphics/1000 cannot use hierarchical files.

Master Relocatables -- Are supplied with FMGR filenames in FC format.

BL Command -- Change buffer limits (UN = unbuffered):

```
+---------+        +---------------------+         +----------+
| program | ===>   |        buffer       |  ====>  | terminal |
+---------+        +---------------------+         +----------+
                             ^             ^
                             |             |
                           lower         upper
                         (eg. 100)     (eg. 300)
```

IN Command -- Initialize a FMGR cartridge.

PK Command -- Pack a FMGR cartridge.

Copying FMGR Files -- The CI copy command can be used to put FMGR files into the hierarchical file system. In fact, any CI command will accept a FMGR filename within the naming restrictions imposed by CI.

References: Utilities Manual

USING FMGR

CI> FMGR
FMGR: EX
CI>

- SUBSYSTEMS
- MASTER RELOCATABLES
- BL, lu, BU/UN, low, high
- IN, msc, oldcrn, newcrn label,, dirtracks
- PK,crn

CI> CO <FMGR file> <CI file>

3.33 FMGR Cartridges

File System Disc LUs -- These are the mounted hierarchical file system volumes.

FMGR disc LUs -- These are the mounted FMGR cartridges.

CRN -- (Cartridge Reference Name) is a name by which the FMGR cartidges may be referred that is similar to a directory name. The CRN is stored as a 16 bit integer and may take on the form of two ASCII characters or an integer in the range -64 to 32767. When the CRN specified is negative, it refers to the LU assigned to the FMGR cartridge. LU 17 may be referred to as "DB" or "-17" in the CRN specification. LU 16 may be referred to as "16" or "-16". Note that the DL command requires a positive LU number.

References: Utilities Manual

FMGR CARTRIDGES

CI> CL

File System Disc LUs: 18 19 23

FMGR Disc LUs (CRN): 16(16) 17(DB)

CI> LI &AREA::DB

CI> LINK %AREA::-17

CI> DL 17

PROGRAM DEVELOPMENT

CHAPTER 4

Table of Contents

Chapter 4
PROGRAM DEVELOPMENT

MODULE OBJECTIVES

1. Be able to use EDIT/1000 to create a Pascal or FORTRAN program with compiler options. Compile, link and run the program.

2. Debug a program using Debug/1000.

3. Be able to create a private, indexed library.

4. Use command files with variable parameters and nesting.

SELF-EVALUATION QUESTIONS

4-1. The first line of the Pascal source file CRASH.PAS contains the option specification:

```
$LIST OFF, TABLES OFF, KEEPASMB, MIX ON
```

A file called CRASH.COP contains the line:

```
$LIST ON, MIX OFF
```

The runstring to compile the program is:

```
CI> pascal CRASH.PAS 6 0 - LIST,TABLES
```

What are the option settings during the compilation? What will appear at LU 6? Why is LU 0 specified for the relocatable file?

4-2. The first few lines of source file CRASH.FTN look like:

```
1       program crash
2
3       dimension node(20)
4       data pi/3.14159/
```

The runstring used to compile the program is:

```
CI> ftn7x CRASH.FTN 1 -,,sm
```

What compiler options are in effect during the compilation? What will appear at the terminal (LU 1)? Where is the relocatable file put? Why are two commas used in the runstring? How many files are produced by the compiler assuming a successful compilation?

4-3. What is the difference between a private library and a library like $BIGLB? What would happen if you named your private library (in your working directory) $BIGLB? Would there be any way to resolve references to routines contained in $BIGLB.LIB::LIBRARIES?

4-4. When you linked your program, the load map showed that your main program started at location 2000 octal (1024 decimal). What is in locations 0 through 1777 octal?

4.1 The Pascal/1000 Compiler

Defaults for <list>, <relocatable> and <option> files:

* If nothing is specified, no file is produced.

* If "-" is specified, the source filename is used with the default type extension.

  For example:

  ```
  CI> pascal area.pas 1 -
  ```

  sends the source listing to LU 1 (the terminal) and creates a relocatable file called AREA.REL.

  ```
  CI> pascal area.pas - 0 -
  ```

  sends the source listing a file called AREA.LST, sends the relocatable file to the bit bucket, and looks for an option file called AREA.COP. Using LU 0 for the relocatable file saves time when debugging syntax errors in large programs since the compiler does not waste time creating relocatable code for the correct parts of the program.

References: Pascal Ref Manual

4.2 Pascal Compiler Options

Options are separated from each other with commas. Parameters to options (e.g., ON or OFF) are separated from the option with a blank. Where options have an ON or OFF parameter, the default will be ON if no parameter is specified.

Runstring -- Options here can contain no blanks, therefore options with parameters (e.g., LIST OFF) are unusable in the runstring. Remember that options like LIST can be used with no parameter and will default to ON.

Option File -- Each line containing options begins with a $ sign. Options here override those in the runstring.

Source File -- Each line containing options begins with a $ sign. Options here override those in the runstring or option file.

References: Pascal Ref Manual

4.2 Pascal Compiler Options

Options are separated from each other with commas. Parameters to options (e.g., ON or OFF) are separated from the option with a blank. Where options have an ON or OFF parameter, the default will be ON if no parameter is specified.

Runstring -- Options here can contain no blanks, therefore options with parameters (e.g., LIST OFF) are unusable in the runstring. Remember that options like LIST can be used with no parameter and will default to ON.

Option File -- Each line containing options begins with a $ sign. Options here override those in the runstring.

Source File -- Each line containing options begins with a $ sign. Options here override those in the runstring or option file.

References: Pascal Ref Manual

PASCAL COMPILER OPTIONS

$tables on,list

LIST [ON or OFF] – turn on/off source listing (errors always listed)

LIST_CODE [ON or OFF] – mix Pascal source with assembly code

TABLES [ON or OFF] – list relocatable addresses and symbol table

XREF – produce cross-reference listing of all variables within block

4.3 Pascal Extensions

Extensions to the language are discussed in Chapter 1 of the Pascal Reference Manual.

References: Pascal Ref Manual

PASCAL EXTENSIONS

WIRTH "STANDARD" PASCAL

PLUS:

1. Additional I/O routines
2. Functions may return records, arrays and sets
3. CASE has subranges and OTHERWISE
4. Constant expressions and structured constants
5. External routines
6. Separate compilation

.

.

.

4.4 The FORTRAN/77 Compiler

Defaults for <list> and <relocatable> files:

* If nothing is specified, no file is produced.

* If "-" is specified, the source filename is used with the default type extension.

  For example:

  ```
  CI> ftn7x area.ftn 1 -
  ```

  sends the source listing to LU 1 (the terminal) and creates a relocatable file called AREA.REL.

  ```
  CI> ftn7x area.ftn 6 0 80
  ```

  sends the source listing to LU 6 (the printer), sends the relocatable file to LU 0 (the bit bucket) and sets the lines per page to 80. Setting the relocatable to 0 allows the compiler to run faster, which may be helpful while debugging syntax errors in large programs.

References: FORTRAN Ref Manual

THE FORTRAN/77 COMPILER

FTN7X <source> <list> <relocatable>
<line count> <options>

<source>	.FTN
<list>	.LST
<relocatable>	.REL
<line count>	59 default, <10=no pagination
<options>	no delimiters

4.5 FORTRAN Compiler Options

Source File -- Single character options must be specified in the first line of the program and are separated by commas. The options must be preceded by FTN77 (which specifies FORTRAN 77 compatibility). If nothing is specified, FTN77,L is assumed.

Runstring -- Any single character options may be specified in the runstring. Runstring options are not separated by commas.

References: FORTRAN Ref Manual

FORTRAN COMPILER OPTIONS

(line 1) FTN7X,T,L,Q

C – produce cross-reference listing of all variables and labels.

L – produce a source listing

$LIST [ON OR OFF] – a program statement to turn on/off source listing.

M – produce a mixed listing of FORTRAN source and assembly code.

Q – include the relocatable address with the source listing.

S – insert information for Symbolic **Debug into relocatable file.**

T – **produce a symbol table.**

4.6 FORTRAN Extensions

The extensions and their backward compatibility are discussed in the FORTRAN 77 Reference Manual in Chapter 8 and Appendix E.

References: FORTRAN Ref Manual

FORTRAN EXTENSIONS

FORTRAN 66
+
FORTRAN 77
+
MIL-STD-1753

* DO WHILE
* block DO
* IMPLICIT NONE

+
HP/1000 FORTRAN

* bit manipulation
* include files
* EMA common areas
* character concatenation
* > 6 character names
* recursion
*
*

4.7 LINKING RELOCATABLE FILES

LINKING RELOCATABLE FILES

4.8 Using LINK Interactively

? Command -- Lists all the available commands. Note that there is no help with individual commands as there is in CI.

RE Command -- Relocates a relocatable module into the current program file being linked.

EN Command -- Ends the linking process. The system libraries are searched to resolve external references (I/O routines, math routines, etc.) and a runnable program file is created.

Modules used to resolve all external references are listed, then the load map is listed which indicates the starting address and size of each module.

References: Link User's Manual

USING LINK INTERACTIVELY

CI> link

```
link Rev.2326  Use ? for help
link: re area.rel
    AREA
link: en
    PNAME XREIO REIO LOGLU
    •
    •
    •
Load Map:
    AREA   2000   126.
    PNAME   2210   24.
    •
    •
    •
Program AREA.RUN ready; 6 Pages
Runnable only on an RTE–A system
CI>_
```

• signify number in decimal otherwise it is octal

4.9 LINK Commands

PR Command -- Set program priority. This defaults to 99 if not otherwise specified in the source file.

DE Command -- Set debug mode. A file with type extension of DBG is created which contains information required by Symbolic Debug/1000.

DI Command -- Displays as yet undefined external references. These must be resolved using the SE command if the references are not in the standard system libraries.

SE Command -- Search a library. LINK will assume a file type extension of .LIB and will look for the file in both your working directory and the directory /LIBRARIES if not otherwise specified.

A Command -- Aborts LINK.

References: Link User's Manual

LINK COMMANDS

link: PR 65

link: DE

link: DI

Undefined symbols:

.NFEX .EIO .FION .FIO .RIO .DTA

link: SE $BIGLB

PNAME XREIO REIO LOGLU

.

.

.

link: A

aborting link

CI>_

4.10 Libraries

$BIGLB -- Contains I/O routines, file handling routines, math routines, and system calls.

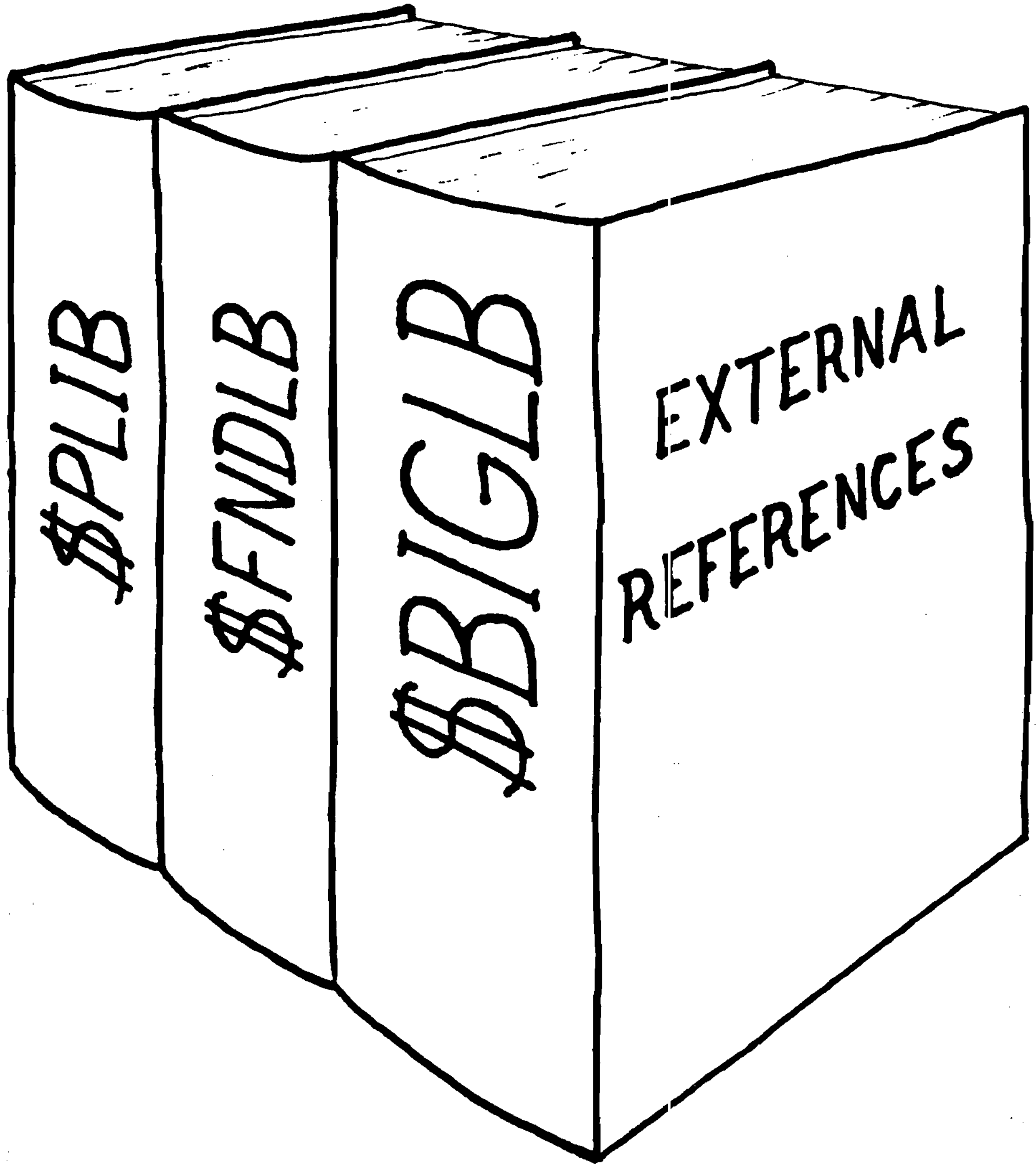
$FNDLB -- Contains FORTRAN routines.

$PLIB -- Contains Pascal routines.

These libraries may or may not be searched automatically when LINKing your programs. The libraries to be searched are defined during system generation.

References: System Generation and Installation Manual

LIBRARIES

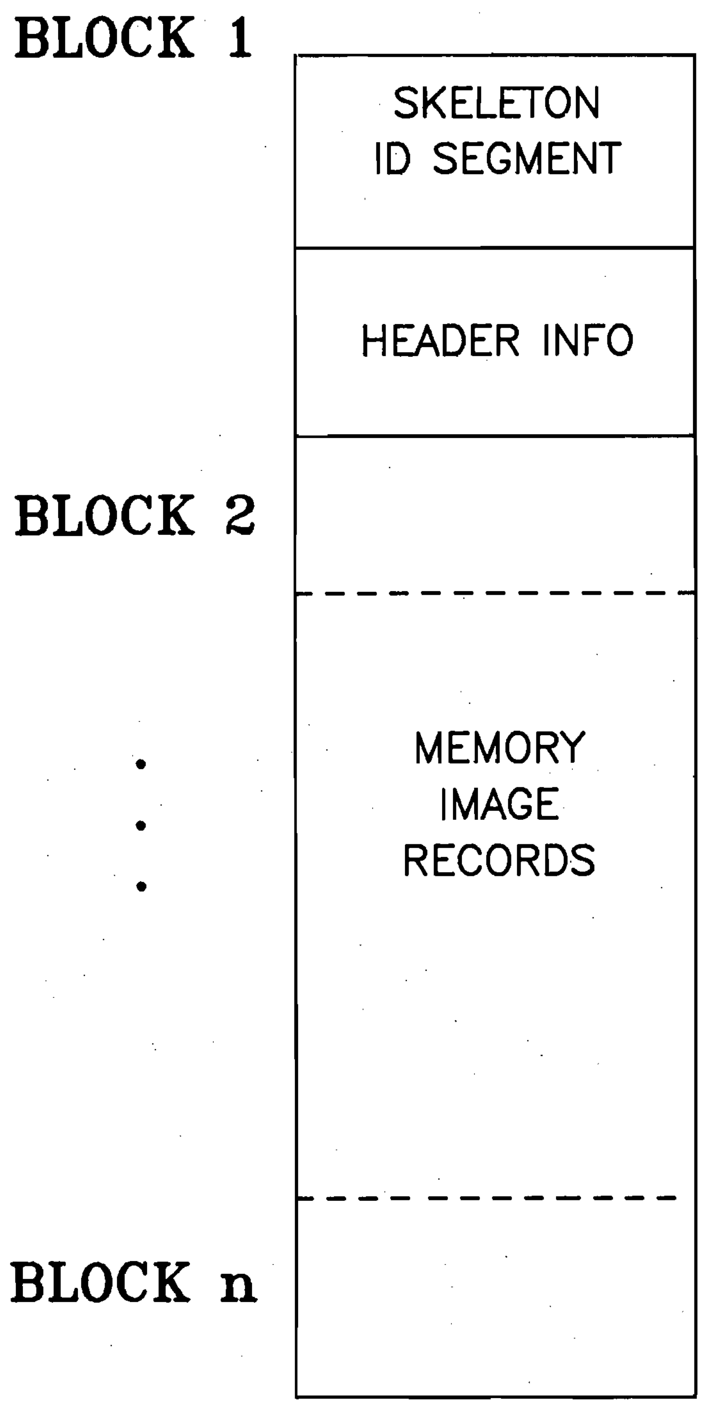


4.11 Type 6 File

Contains 256 byte blocks. The first block contains housekeeping information. The remaining blocks contain the program memory image records.

References: none

TYPE 6 FILE


BLOCK 1
SKELETON
ID SEGMENT
HEADER INFO
BLOCK 2
MEMORY
IMAGE
RECORDS
BLOCK n

4.12 Private Libraries

Any set of relocatable files can be made into a library using the MERGE and LINDX utilities.

MERGE -- The first parameter is shown as LU 1 which causes the program to prompt for a list of relocatable files to be merged. This parameter can also be a filename which contains the list of relocatable files to be merged.

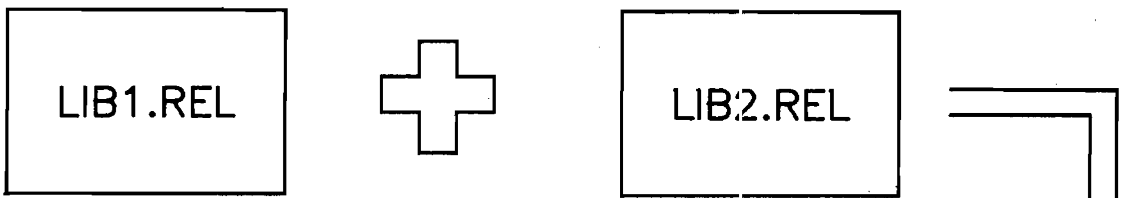
The second parameter must be the name of the file which will contain the merged files.

LINDX -- Creates an index to all external references in the library file.

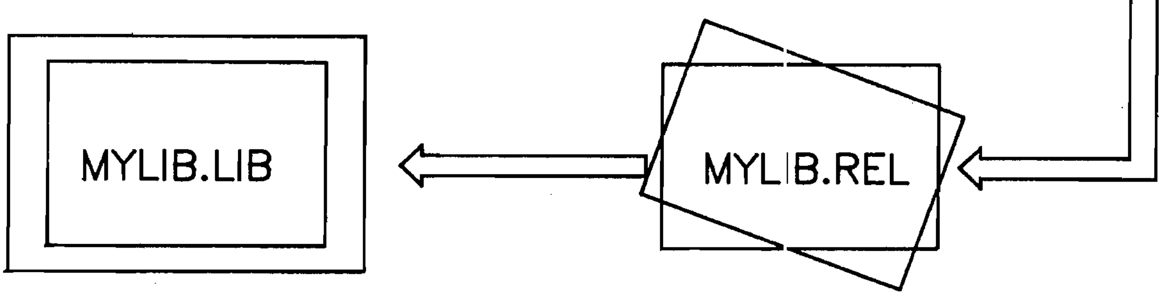
References: Utilities Manual, Link User's Manual

PRIVATE LIBRARIES

CI> merge 1 mylib.rel
Enter filename lib1.rel
Enter filename lib2.rel
Enter filename <cr>



CI> lindx mylib.rel mylib.lib
Sorting entries
A_ENTRY
B_ENTRY
.
.
.



4.13 SYMBOLIC DEBUG/1000

SYMBOLIC DEBUG/1000

4.14 Using Debug

Debug may only be used if the program was compiled with the debug compiler option in effect and if the relocatable file was LINKed using the DE command with LINK. LINK will create a file with the same name as the program but with a type extension of DBG. Note that Debug will appear to work if an old copy of the DBG file is used with a newer version of the program, but with misleading results.

Debug displays a portion of the source file and has a pointer to the line that is about to be executed. Help with commands is available by using the ? command.

P Command -- Proceed to a specified line number. If no line number is specified, the program runs to the next breakpoint or to completion if no breakpoint is encountered.

E Command -- Exit debug. The symbol table is saved in the file with type extension DBG.

References: Debug Ref Manual

USING DEBUG

CI> debug area

```
  1   program area
  2
  3   real radius, area
  4
> 5   write (1,'("area of circle program")')
  6   radius = 1
  7   do while (radius .GT. 0)
  8       write (1,'("radius:  ")')
  9       read (1,*)radius

  2326 Version
  DEBUG>
```

DEBUG> p 7
area of circle program

DEBUG> e
Saving symbol table

CI> _

4.15 Debug Commands

B Command -- Sets a breakpoint at the line specified. Any number of breakpoints may be set in a program. Breakpoints may be specified within a different module (source file) by specifying the module name after the line number, separating them with a slash. Breakpoints may also be set to break after a certain number of iterations thru the breakpoint (as in a loop). They may also break contingent on the value of a variable. See the Debug/1000 Reference Manual for more information.

P Command -- Proceed to a breakpoint or a specified line number.

D Command -- Display the value of a variable.

C Command -- Clear a breakpoint.

M Command -- Modify the value of a variable.

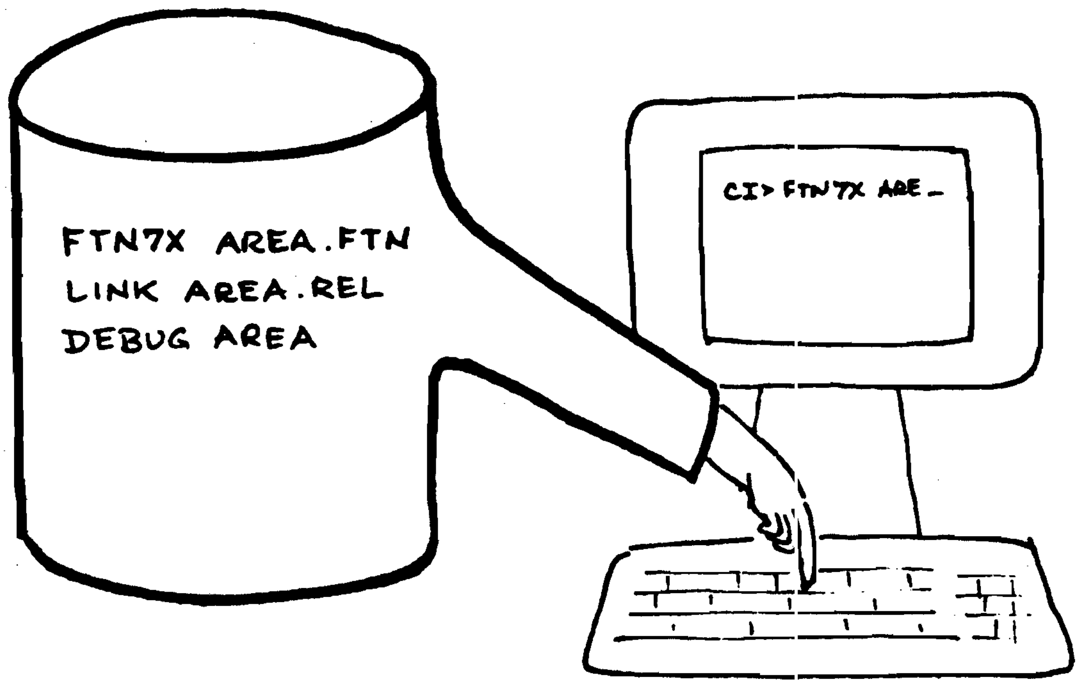
References: Debug Ref Manual

DEBUG COMMANDS


```
DEBUG> b 7
Breakpoint set at 7/AREA

DEBUG> p
area of circle program

DEBUG> d radius
RADIUS = 1

DEBUG> c 7
Cleared 7/AREA

DEBUG> m radius 0
RADIUS:1 => 0

DEBUG> p
Program ran to completion

CI> _
```

4.16 COMMAND FILES

COMMAND FILES

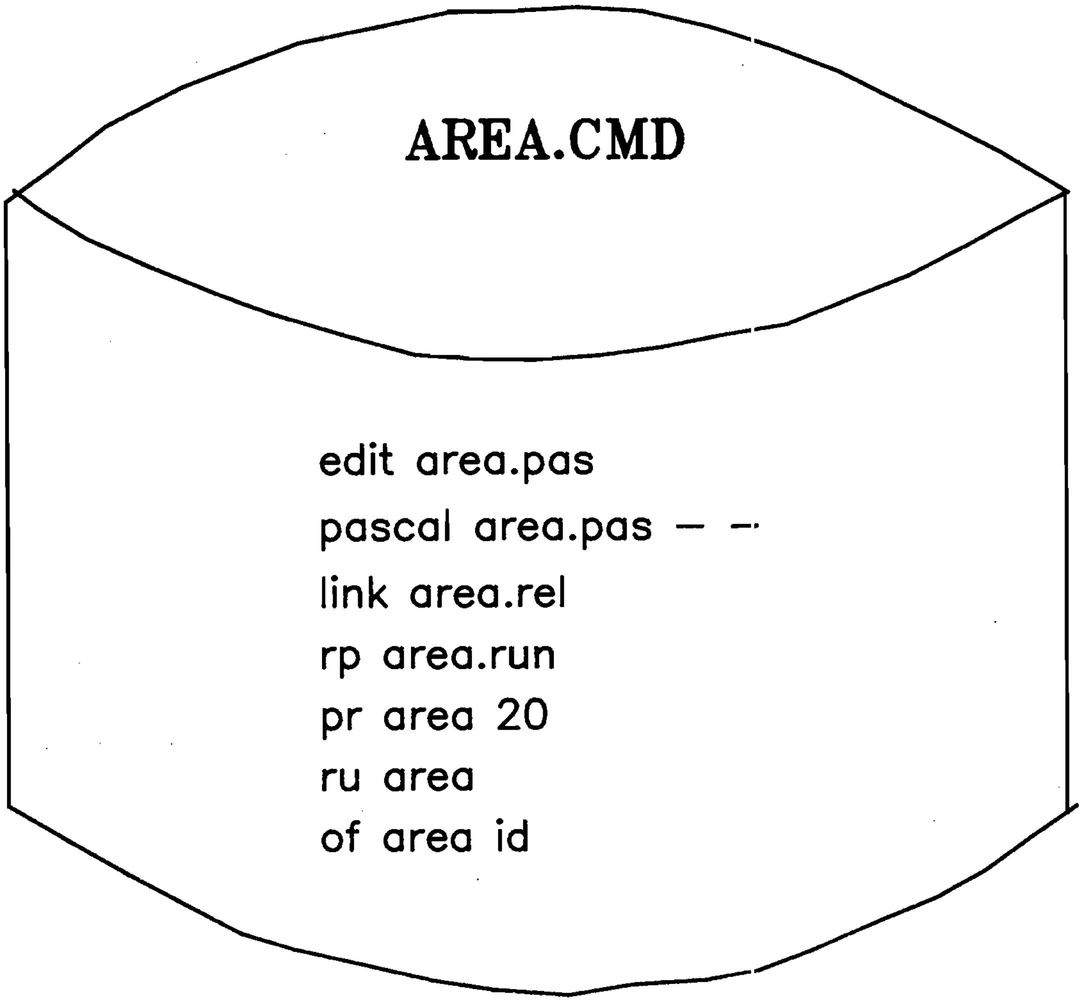


4.17 Using Command Files

Command files contain a list of commands to be executed sequentially. The TR command transfers control to the command file, the commands are executed, and control is transferred back to CI.

References: User's Manual

USING COMMAND FILES



CI> tr area.cmd

4.18 $ Parameters

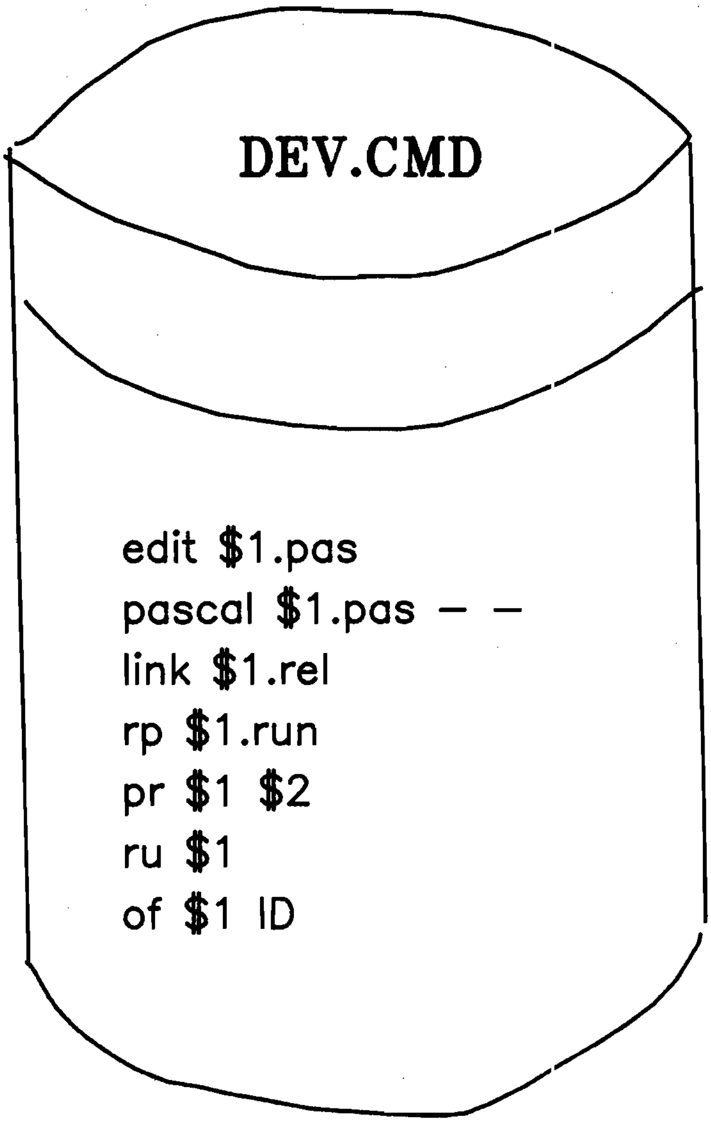
The parameters included in the TR command are automatically given the names $1, $2, ..., $9. These names can then be used in the command file to directly substitute the characters from the runstring.

References: User's Manual

$ PARAMETERS

$1 $2 $9

CI> tr <cmdfile><param1><param2>...<param9>



CI> tr dev.cmd area 20

4.19 Command File as a CI Parameter

The Command Interpreter CI executes the program file CI which requires the creation of a clone name.

The first parameter to the program CI is assumed to be a command file. The commands in this file are executed by the CI clone and the clone is terminated. The third through eleventh parameters are the $ parameters to the command file.

References: User's Manual

COMMAND FILE AS A CI PARAMETER

CI> ci <cmdfile> <parameters>

* CI clone: CI..A

* execute commands

* exit CI..A

CI>_

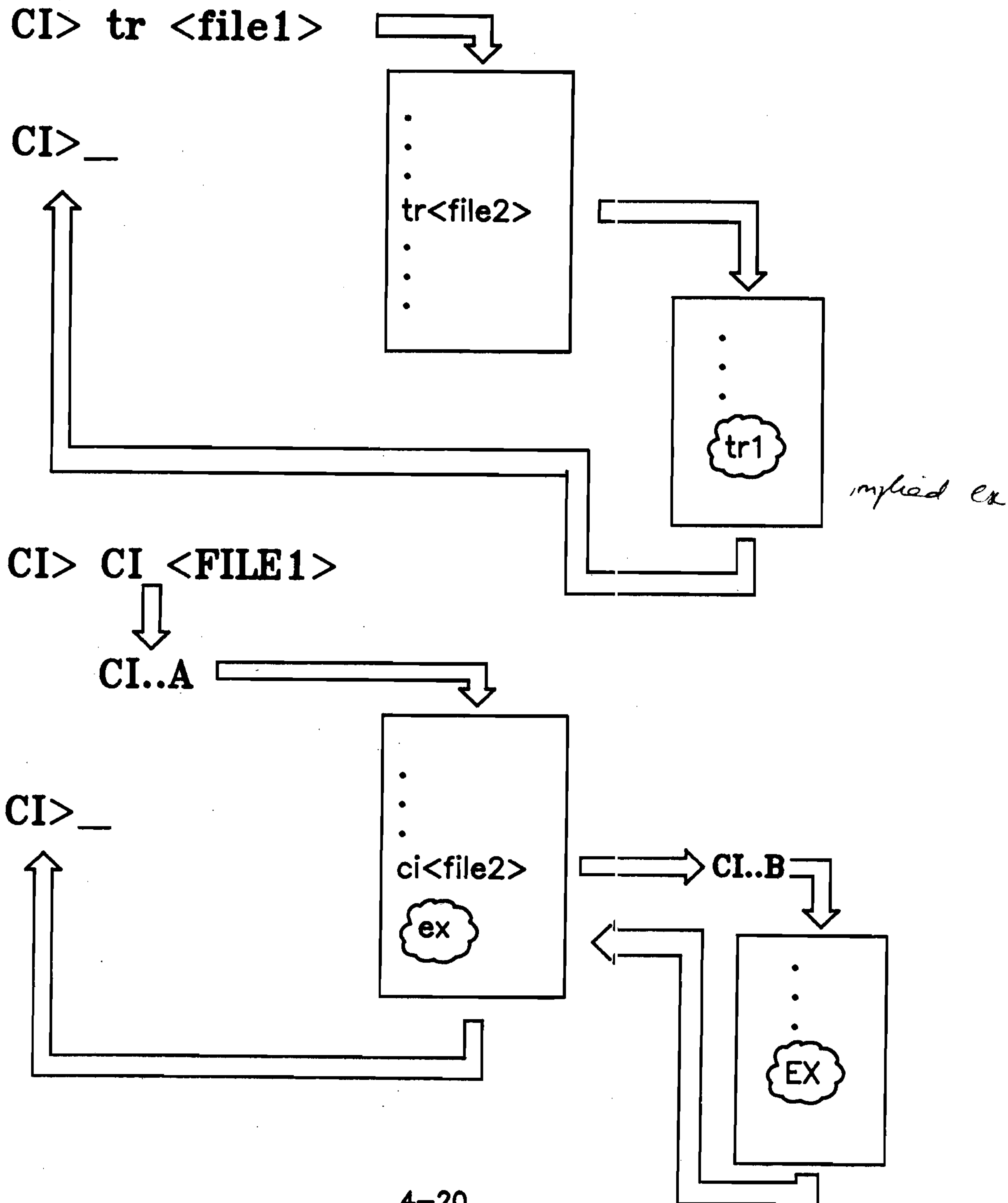
4.20 Exiting Command Files

When a command file is executed with a TR command, there is an implied TR,1 command at the end of the file. This effectively terminates the command sequence even if the command files were nested.

When a command file is executed as a parameter to CI, there is an implied EX command at the end of the file. This exits the CI clone that was executing the command file and returns control to the previous copy of CI that created the clone. Notice the effect on nested command files.

References: User's Manual

EXITING COMMAND FILES

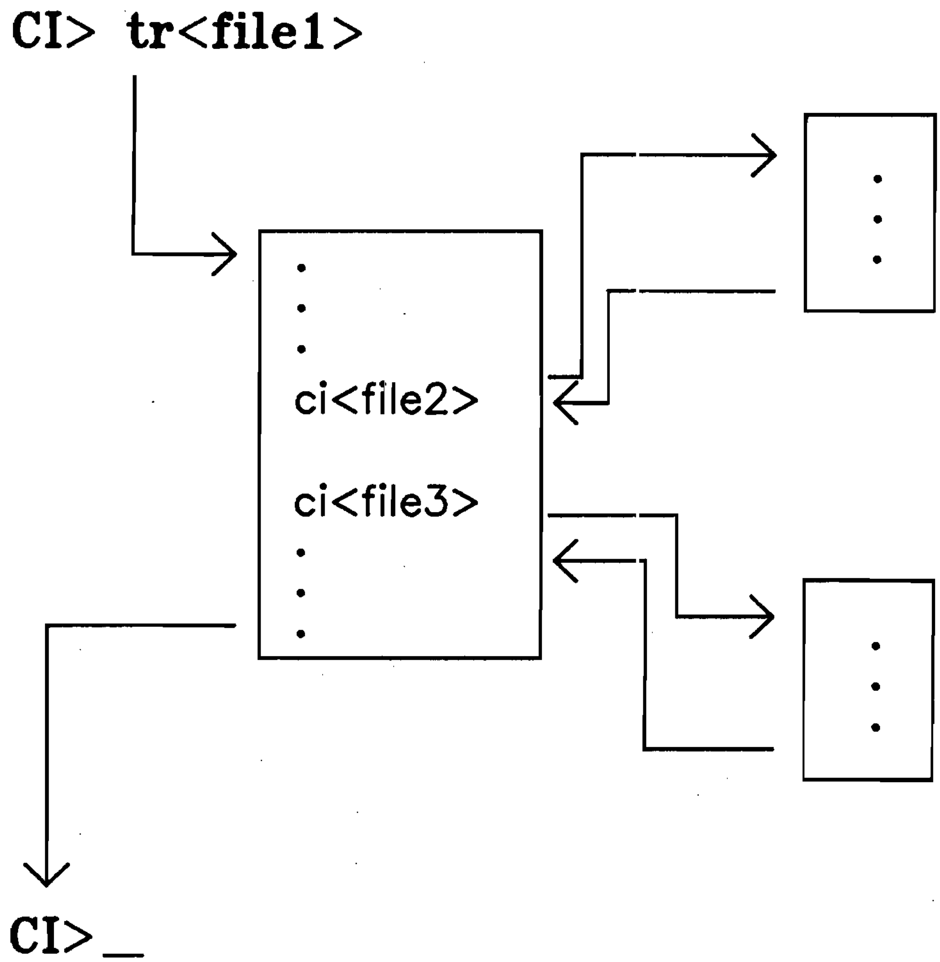


4.21 Nesting Command Files

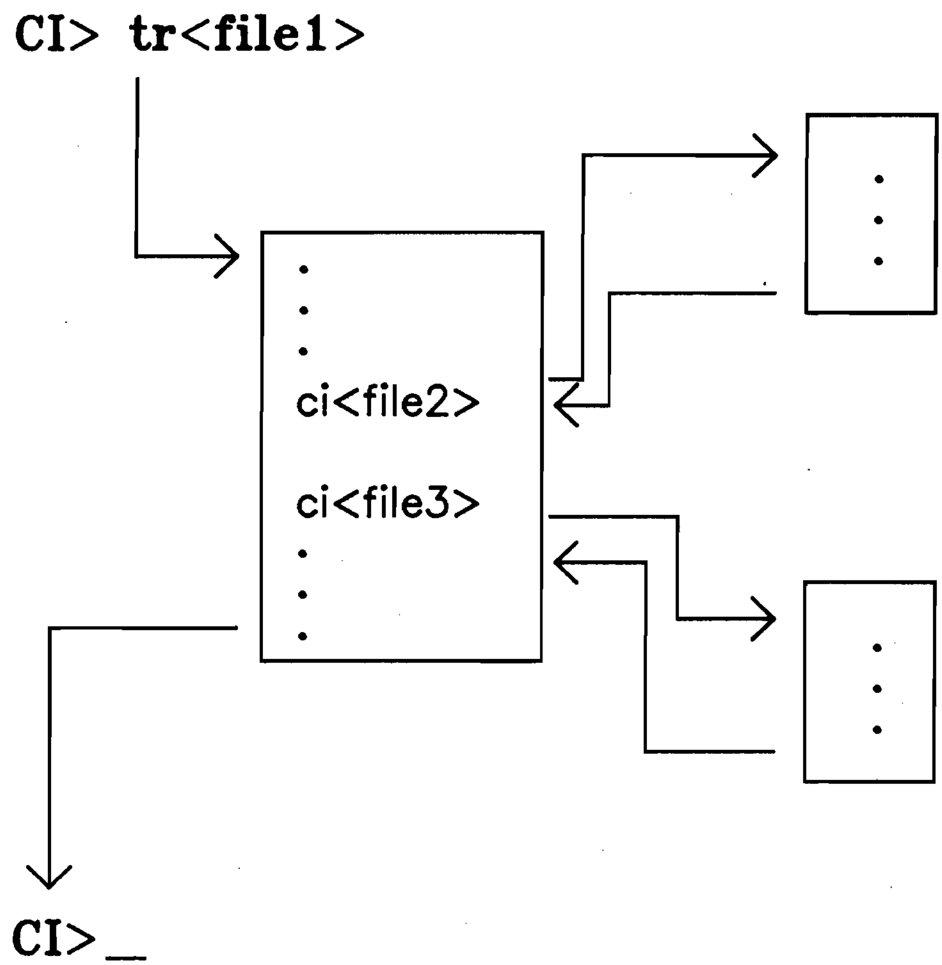
Nesting is accomplished by cloning copies of CI with command file parameters. Nesting can be to any level to the limit of the number of ID segments available for the cloned copies of CI. Note that although parameters can be passed from each copy of CI to it's clone, no information can be passed back.

References: User's Manual

NESTING COMMAND FILES



NESTING COMMAND FILES



USING RTE-A PROGRAMMATICALLY

CHAPTER 5

Table of Contents

MODULE OBJECTIVES

1. Understand the control path from an I/O request through the LU table, device table, and interface table to the device itself.

2. Know the advantages and disadvantages of using EXEC I/O as opposed to using the I/O routines of a higher level language.

3. Be able to write a program using EXEC I/O with programmatic error recovery and the various forms of I/O buffering.

SELF-EVALUATION QUESTIONS

5-1. What is the function of the device driver?

5-2. What is the function of the interface driver?

5-3. Where does the device status information come from?

5-4. What are the disadvantages of using EXEC services?

5-5. What are the advantages of using EXEC services?

5-6. What happens in an EXEC 2 (write) call with no option bits set if the device to which you are writing is off-line?

5-7. What are the advantages of using a buffered read or write? What are the disadvantages?

5-8. When reading from a device, how can your program detect the "end of file"?

5-9. What is the advantage of always using XLUEX and XREIO? Can you think of any disadvantages?

5.1 LUT, DVT and Device Driver

I/O Request -- References an LU. The system uses the LU number as an index into the LU table.

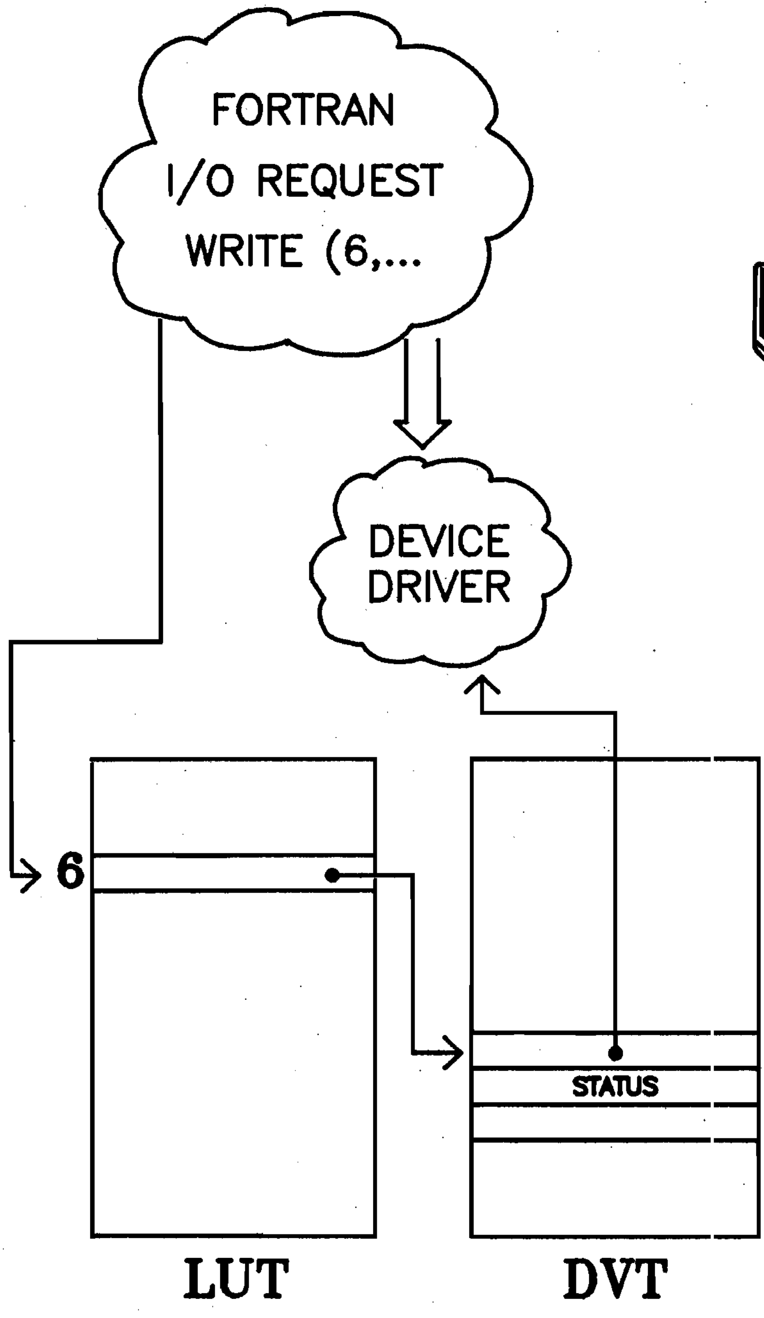
LU Table (LUT) -- Maps each LU to an entry in the device table associated with the requested device.

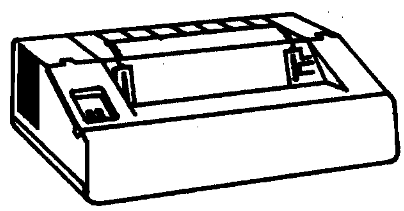
Device Table (DVT) -- Contains the latest status information for the device and the entry point for the device driver routine.

Device Driver -- Makes the communication protocol required for the device transparent to the user.

References: System Design Manual

LUT, DVT AND DEVICE DRIVER





5.2 IFT and Interface Driver

Device Table (DVT) -- Also contains a pointer into the interface table to an entry associated with the type of I/O card to which the device is connected.

Interface Table (IFT) -- Contains the select code which physically identifies the I/O card. It also contains the entry point for the interface driver routine associated with the I/O card.

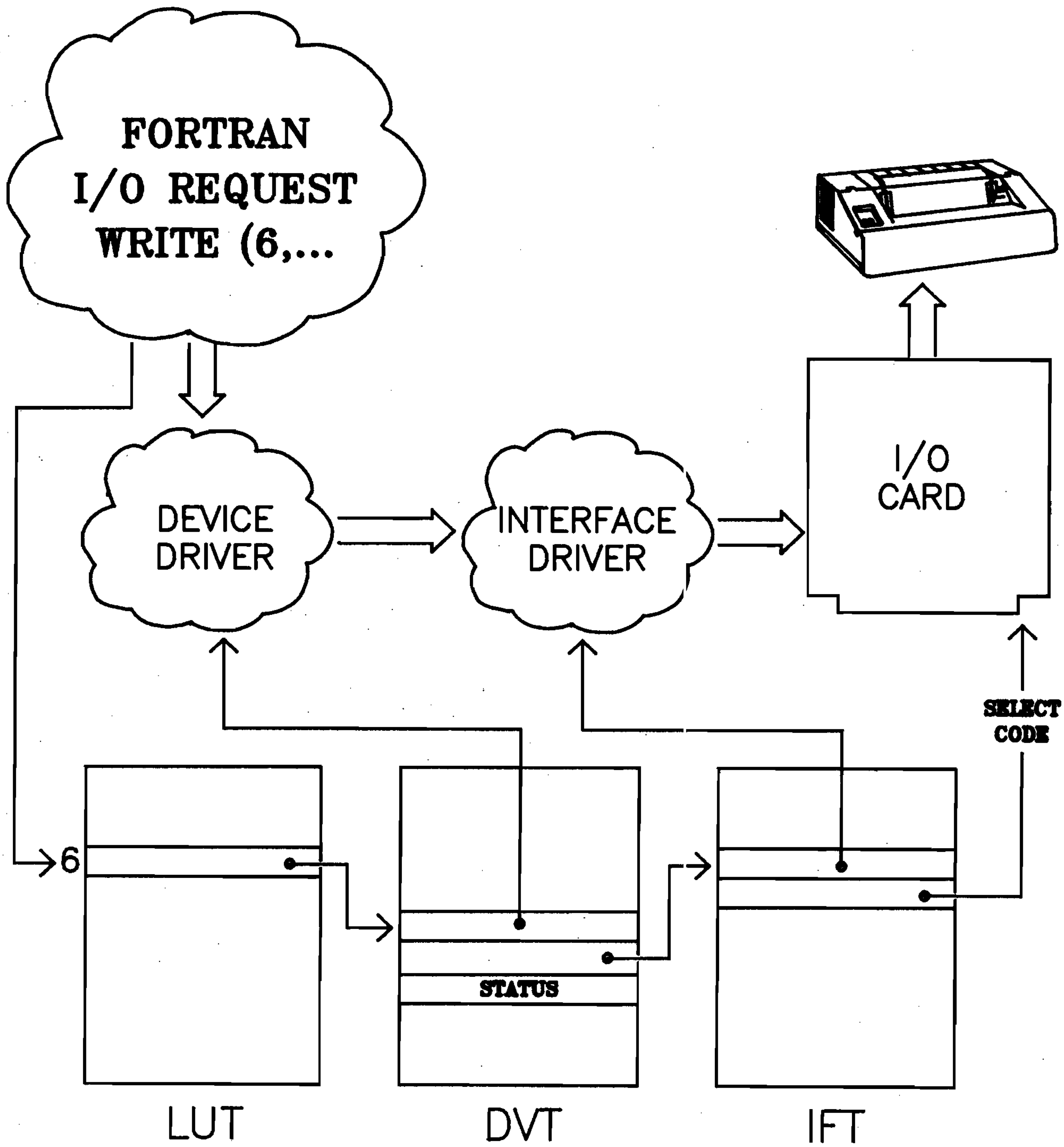
Interface Driver -- Communicates with the device driver and the I/O card to provide the necessary protocol for the I/O card.

I/O Card -- Is physically connected to the device.

one IFT / interface
one DVT / device (LU)

References: System Design Manual

IFT AND INTERFACE DRIVER



5.3 E X E C C A L L S

EXEC

CALLS

5.4 Intro to EXEC Calls

ADVANTAGES

Less Code -- This means the program will use less memory and, in general, be faster than the Pascal or FORTRAN counterpart.

Unavailable Services -- Services not available from high-level languages include communication to other programs and scheduling other programs. EXEC calls also allow much more control over standard I/O services such as testing a device to see if its busy before initiating data transfer.

DISADVANTAGES

Not Portable -- Programs using EXEC calls will not be portable to other types of computers and may be marginally portable to previous RTE operating systems.

Less Readable -- EXEC calls are very criptic. The main difference between EXEC calls is the first parameter, for example EXEC(1,...) is a read request and EXEC(11,...) is a time request.

More Difficult -- EXEC calls have two or (usually) more parameters. Their are many flag bits that may be set in various combinations to control the way the request is handled. Formatting and error returns must be handled by the program.

EXEC calls were originally written to interface with FORTRAN programs. The use of EXEC calls with Pascal is sometimes more difficult than FORTRAN because of differing data formats (e.g., EXEC often expects strings to be passed in integer arrays) and because of Pascal's strong type checking (e.g., EXEC(1,...) and EXEC(11,...) have to be declared as separate procedures and aliased to reference the same external name because of differing parameter types).

References: Prog Ref Manual

INTRO TO EXEC CALLS

RTE-A EXEC Services:

* I/O Communication
* Program to Program Communication
* Program Control
* System Time Requests

Advantages:

* Produce less code than
  Pascal/FORTRAN
* Perform services unavailable
  from high-level languages

Disadvantages:

* Programs not as portable
* Programs are less readable
* More difficult to use than
  high-level language services

5.5 The Generic EXEC Call

May be called as a procedure (Pascal), a subroutine (FORTRAN), or a function (Pascal or FORTRAN). In Pascal, the EXEC call must be declared external in the procedure declaration section.

ECODE -- is a one-word integer that identifies the specific service requested from EXEC. Bit 14 and 15 in this word are used to change the form of error handling and are not used to identify the service request. Therefore, the ECODES 2, 16386, -16382, and -32766 (in decimal) are all for the same service request.

Parameters -- Are usually one-word integers or integer arrays. Parameters are assumed to be passed by reference. This presents no problems in FORTRAN/77 or Pascal/1000 since values are always passed to subroutines by reference (in Pascal/1000, it is the procedure itself that de-references the non-VAR parameters).

A- and B-Registers -- Return error information or other data related to the specific request. If the EXEC call is made as a function, the returned value is the A- and B-Register for two-word function types or just the A-Register for one-word types.

This info will be useful when setting control bits:

Control bit:	Decimal:	Octal:	Hexidecimal:
0	1	1	1
1	2	2	2
2	4	4	4
3	8	10	8
4	16	20	10
5	32	40	20
6	64	100	40
7	128	200	80
8	256	400	100
9	512	1000	200
10	1024	2000	400
11	2048	4000	800
12	4096	10000	1000
13	8192	20000	2000
14	16384	40000	4000
15	-32768	100000	8000

References: Prog Ref Manual

THE GENERIC

EXEC CALL

THIS CALL OPERATES DIFFERENTLY WITH DIFFERENT ECODES AND IS SUITABLE FOR EVERYDAY USE.

EXEC (ECODE,P1,P2,...,Pn)

IDENTIFIES THE SPECIFIC SERVICE REQUESTED (ECODE)

FURTHER SPECIFIES THE REQUEST (P1,P2,...,Pn)

RETURNS:

P1,P2,...Pn	RETURN PARAMETERS CONTAIN THE INFORMATION REQUESTED
A AND B REGISTERS	

ERRORS:

RTE HANDLER	OPERATING SYSTEM ABORTS OR SUSPENDS THE PROGRAM.
PROGRAMMATIC	NO ACTION IS TAKEN BY RTE. THE PROGRAM MUST DETECT AND CORRECT THE ERROR.

5.6 Read and Write -- EXEC 1, 2

FORTRAN example:

```
      program example

      integer bufr(40), prompt(13),  !NOTE: must be integer buffers
     +        bufln, promptln, cntwd, lu, ec
      data prompt/'ENTER UP TO 80 CHARACTERS:'/
     +     bufln/40/, promptln/13/, ec/400b/

      lu = 1                     !send the prompt to the terminal (LU 1)
      cntwd = lu
      call exec(2,cntwd,prompt,promptln)

      cntwd = lu + ec            !read and echo input
      call exec(1,cntwd,bufr,bufln)

      cntwd = lu                 !write input back to terminal
      call exec(2,cntwd,bufr,bufln)
      end
```

Pascal Example:

```
program example (input, output);

   type buffer = packed array [1..80] of char;
        int    = -32768..32767;
   var bufr, prompt : buffer;
       bufln, promptln, cntwd, lu, ec : int;
   procedure exec (ecode, cntwd: int; bufr: buffer, bufln: int);
      external;

   begin
      prompt := 'ENTER UP TO 80 CHARACTERS';
      bufln := -80;
      promptln := -25;
      ec := 256;
      lu := 1;

      cntwd := lu;        {send the prompt to the terminal (LU 1)}
      exec (2, cntwd, prompt, promptln);

      cntwd := lu + ec;   {read and echo input}
      exec (1, cntwd, bufr, bufln);

      cntwd := lu;        {write input back to terminal}
      exec (2, cntwd, bufr, bufln);
   end.
```

References: Prog Ref Manual

READ AND WRITE

EXEC (ECODE,CNTWD,BUFR,BUFLN)

ECODE **1=READ** / **2=WRITE** — ASCII

CNTWD:

	NB		Z		EC		LU
	14		12		8		5 4 3 2 1 0

LU LU OF DEVICE
EC ECHO TO TERMINAL (TERMINAL READ)
Z LOOK FOR ADDITIONAL PARAMETERS
NB NON-BUFFERED I/O

BUFR INTEGER ARRAY

BUFLN LENGTH OF BUFR (+WORDS OR −CHARACTERS)

A-REGISTER RETURNS DEVICE STATUS (UNBUFFERED I/O ONLY)

B-REGISTER RETURNS ACTUAL NUMBER OF WORDS (CHARACTERS) READ/WRITTEN (ALWAYS POSITIVE)

5.7 Automatic Output Buffering

SAM -- System Available Memory

BL command -- (buffer limits) can be changed interactively through FMGR:

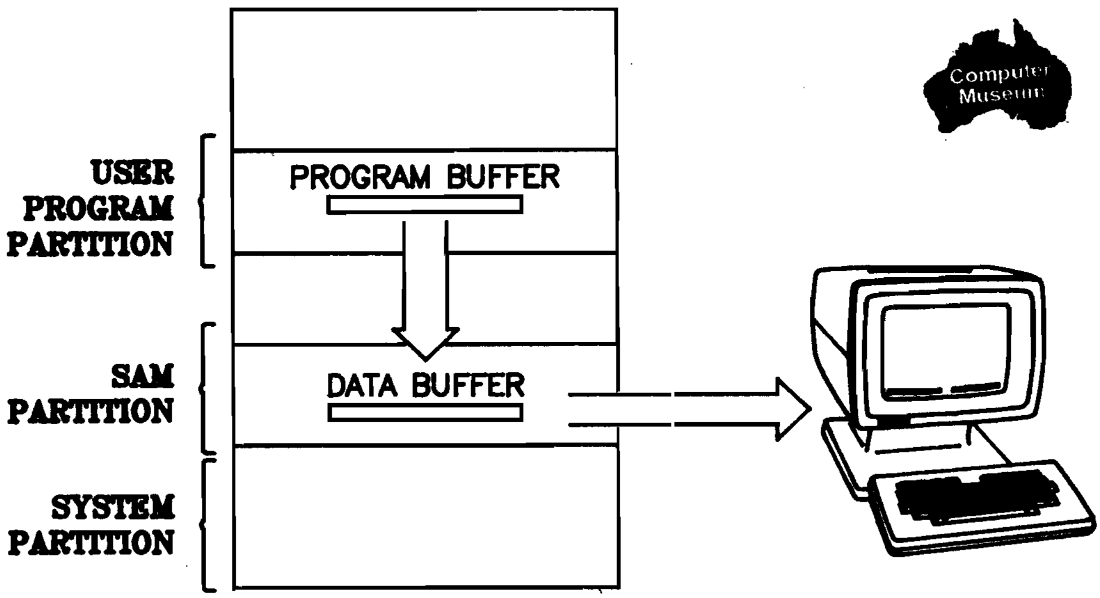
```
     CI> fmgr

     FMGR: bl,1,bu,100,300             sets buffer limits to 100,300

     FMGR: bl,1,un                     sets unbuffered operation

     FMGR: bl,6                        displays current buffer limits
     LU# 6  BU  BL= 96, 384  AC= 85       and accumulated characters

     FMGR: ex

     CI> _
```

References: Prog Ref Manual

AUTOMATIC OUTPUT BUFFERING

* TERMINALS AND PRINTERS ARE USUALLY BUFFERED FOR OUTPUT

* FMGR: bl,<lu>,bu,<low>,<high>



* PROGRAM CONTINUES AFTER DATA IS TRANSFERRED TO SAM

* PROGRAM IS SWAPPABLE WHILE DOING I/O

5.8 Unbuffered Write

FORTRAN example:

```
      integer nb
      data nb/40000b/
          .
          .
          .
      cntwd = lu + nb
```

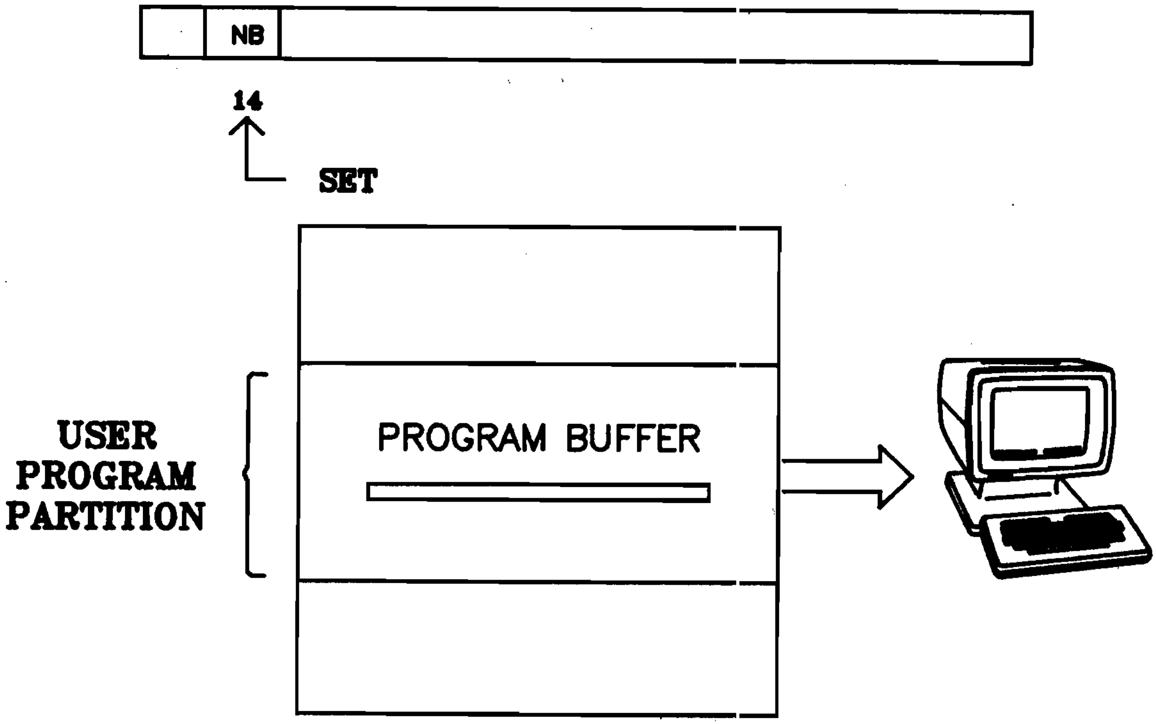
Pascal example:

```
   var nb : int
       .
       .
       .
   nb:= 16384;
       .
       .
   cntwd := lu + nb;
```

References: Prog Ref Manual

UNBUFFERED WRITE

CNTWD:



PROGRAM MUST WAIT FOR I/O

PROGRAM NON-SWAPPABLE WHILE DOING I/O

MUST BE USED TO OBTAIN DEVICE STATUS OR FOR USER ERROR HANDLING

5.9 Device Status From the A-Register -- ABREG

DB -- Device busy, such as a tape rewind preventing any other operation from starting.

EOM -- End of medium, such as attempting to write past the end of the tape.

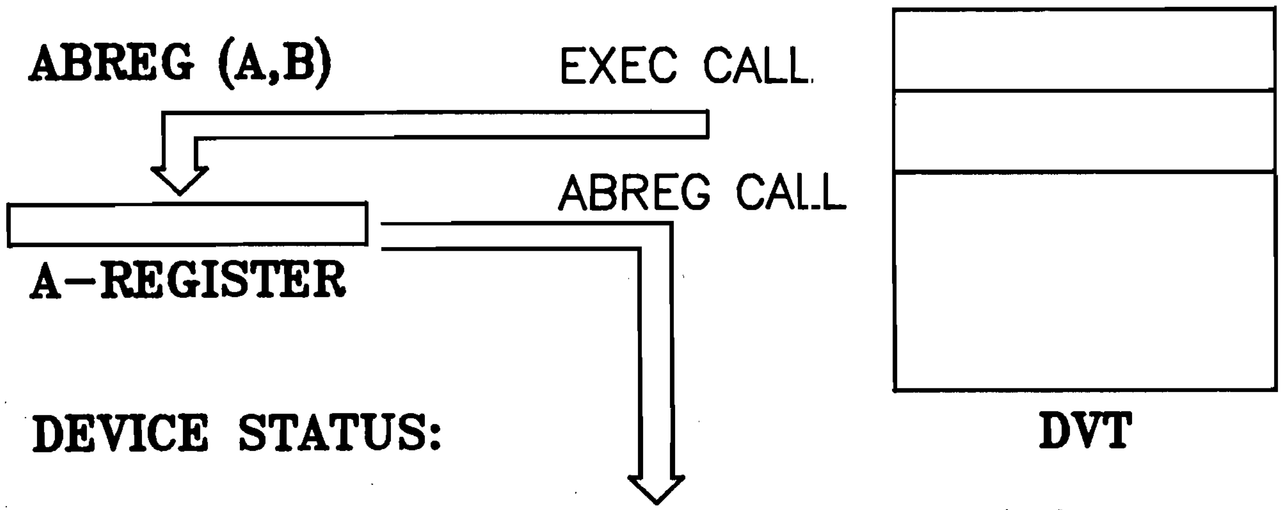
SOM -- Start of medium, might be found after issuing a rewind request.

SE -- Soft error would be found if the read/write was successful, but not without some error recovery attempts such as a re-read after an initial parity error.

E -- Hard error is found when the I/O operation was unsuccessful such as for device time-out, down device, or write-protection. For any error of this nature, the appropriate error message is displayed (unless the UE bit was set for the request).

References: Prog Ref Manual

DEVICE STATUS FROM THE A-REGISTER



AV	DEVICE TYPE	EOF	DB	EOM	SOM	SE		E
15 14	13 12 11 10 9 8	7	6	5	4	3		0

AV – AVAILABILITY: 0=AVAILABLE, 1=DOWN

DEVICE TYPE – 0–7=TERMINAL, 23=MAG TAPE
12 =PRINTER, 37=HPIB

EOF – END OF FILE DETECTED

DB – DEVICE BUSY

EOM – END OF MEDIUM

SOM – START OF MEDIUM

SE – SOFT ERROR

E – HARD ERROR

5.10 User Error Handling

FORTRAN example:

```
      integer areg, breg, nb, na, ns
      data nb/40000b/, na/100000b/, ns/40000/
         .
         .
         .
      cntwd = lu + nb
      call exec(2+na+ns,cntwd,bufr,bufln)
      goto 777            !return here if error occurs
      continue            !return here if all is ok
         .
         .
         .
777   call abreg(areg,breg)
      write(1,'("Error on write = ",a2,a2)') areg,breg
```

Pascal example:

```
   label 777;

   var areg, breg, nb, na, ns : int;

   procedure abreg (areg, breg: int);
      external;

   begin
      nb := 16384;
      na := -32768;
      ns := 16384;
         .
         .
      cntwd := lu + nb;
      exec (2+na+ns, cntwd, bufr, bufln);
      goto 777;       {return here if error occurs}
      ;               {return here if all is ok}
         .
         .
777   abreg (areg, breg);
      writeln ('Error on write = ', areg, breg);
```

References: Prog Ref Manual, Pascal & FORTRAN Ref Manuals

USER ERROR HANDLING

ECODE:

NA	NS	
15	14	

SET

NA – NO ABORT

NS – NO SUSPEND (LU DOWN, LU LOCKED, ETC)

```
CALL EXEC (...)
GOTO 777
CONTINUE
```

ERROR

NORMAL RETURN

A-REGISTER:

SC SCHEDULING ERROR
RN RESOURCE NUMBER ERROR
LU LU ERROR
CL CLASS I/O ERROR
IO I/O ERROR

UNCONDITIONAL ABORT:

EXEC ERROR – TOO MANY PARAMETERS, ILLEGAL ECODE

CPU ERROR – UNIMPLEMENTED INSTRUCTION, MEMORY PROTECT VIOLATION, ETC

5.11 Prompting for Input

FORTRAN example:

```
      integer ec, z
      data ec/400b/, z/10000b/

      cntwd = lu + ec + z      !prompt for data and read it
      call exec(1,cntwd,bufr,bufln,prompt,promptln)
        .
        .
        .
```

Pascal example:

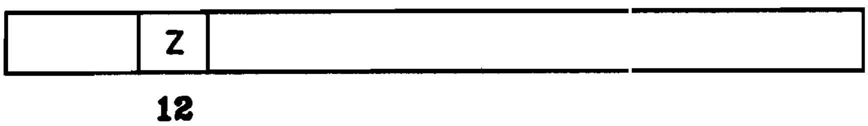
```
   var ec, z : int;

   begin
      ec := 256;
      z := 4096;
        .
        .
      cntwd := lu + ec + z;      {prompt for data and read it}
      exec (1, cntwd, bufr, bufln, prompt, promptln);
```

References: Prog Ref Manual

PROMPTING FOR INPUT

EXEC 1 READ/WRITE

exec (1,cntwd,bufr,bufln,prompt,promptln)



Z – look for additional parameters

*** PROGRAM IS <u>NOT</u> SWAPPABLE**

5.12 Buffered Input -- REIO

FORTRAN example:

```
call reio (1, cntwd, bufr, bufln)
```

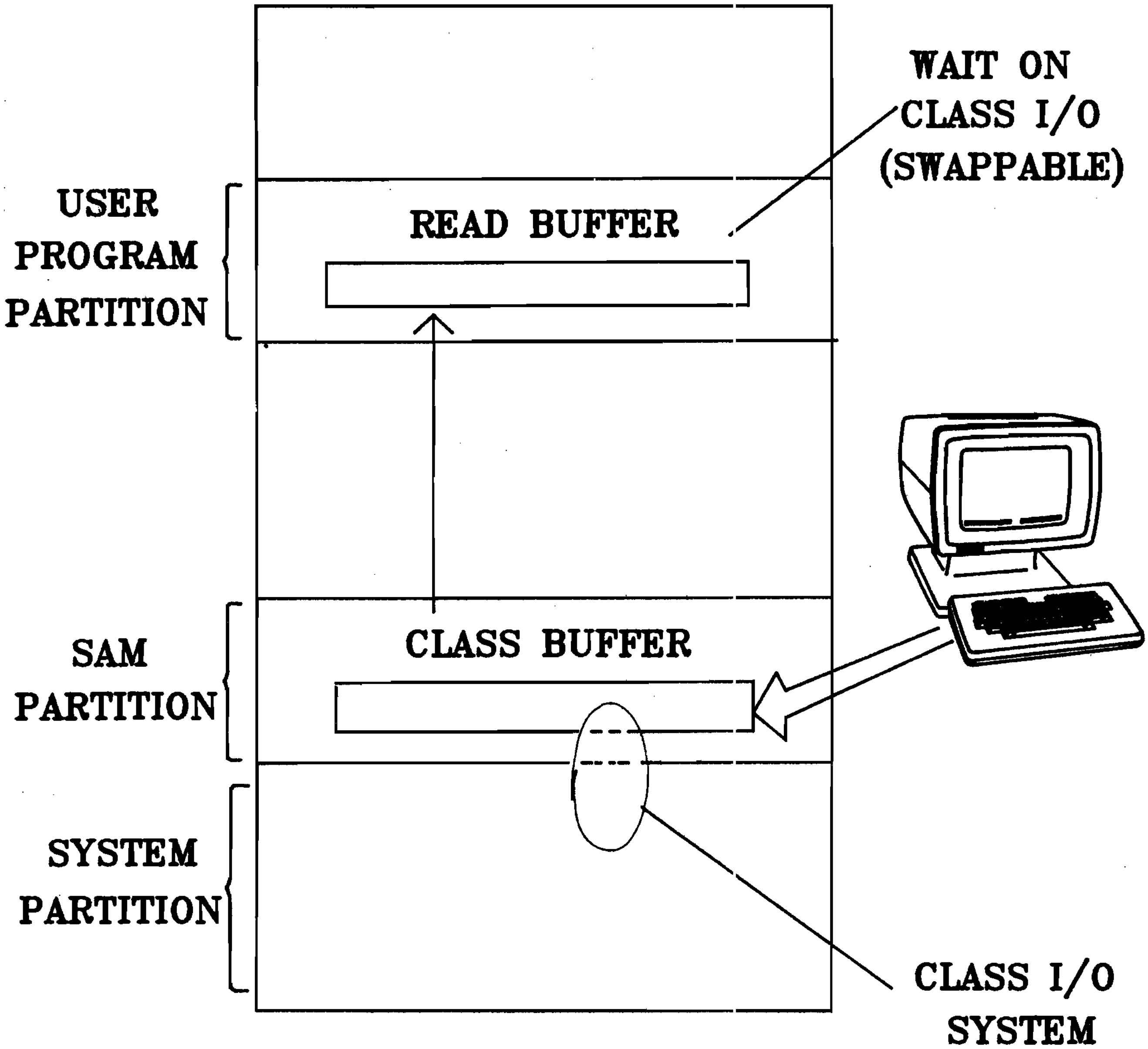
Pascal example:

```
reio (1, cntwd, bufr, bufln);
```

References: Prog Ref Manual

BUFFERED INPUT

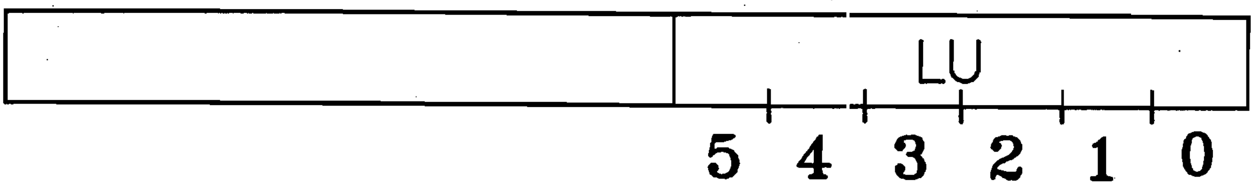
reio (ecode, cntwd, bufr, bufln)
ecode = 1



5.13 Standard LU Addressing

STANDARD LU ADDRESSING

CNTWD:



HOW MANY LU'S CAN WE ADDRESS ?

5.14 Extended EXEC Calls -- XLUEX, XREIO

FORTRAN example:

```
      integer cntwd(2), ec, nb, ov, os
      data ec/400b/, nb/40000b/, ov/100000b/, os/40000b/
         .
         .
         .
      cntwd(1) = lu + ov + os
      cntwd(2) = ec + nb
      call xluex(2,cntwd,prompt,promptln)
      call xreio(1,cntwd,bufr,bufln)
```

Pascal example:

```
   type intarray = array [1..2] of int;

   var cntwd : intarray;
       ec, nb, ov, os : int;

   begin
      ec := 256;
      nb := 16384;
      ov := -32767;
      os := 16384;
         .
         .
      cntwd[1] := lu + ov + os;
      cntwd[2] := ec + nb;
      xluex (2, cntwd, prompt, promptln);
      xreio (1, cntwd, bufr, bufln);
```

References: Prog Ref Manual

EXTENDED EXEC CALLS

LU'S > 63

XLUEX ⟺ **EXEC**

XREIO ⟺ **REIO**

CNTWD:

<table>
<tr><td>OV</td><td>OS</td><td colspan="6"></td><td colspan="8">LOGICAL UNIT</td></tr>
<tr><td colspan="10">SAME AS BEFORE</td><td colspan="6"></td></tr>
<tr><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
</table>

OV – OVERRIDE LU MAPPING (VC+)

OS – OVERRIDE SPOOLING (VC+)

NOTE: You should normally use XLUEX and XREIO. XLUEX will also replace EXEC 3, 13 ,17 ,18 ,19 ,20.

5.15 Buffering Review

(fill in)

References: Prog Ref Manual

BUFFERING REVIEW

EXEC CALL	BUFFERED YES	BUFFERED NO	SWAPPABLE YES	SWAPPABLE NO
EXEC 1 XLUEX 1				
EXEC 2 XLUEX 2				
REIO 1 XREIO 1				
REIO 2 XREIO 2				

5.16 SYSTEM SUBROUTINES

SYSTEM SUBROUTINES

5.17 Op System Command -- MESSS

FORTRAN example:

```
      program messscall

      integer bufr(40), lu, count, ic
      data bufr/'RU,WH'/, count/80/

      lu = 71
      ic = messs(bufr,count,lu)
      if (ic.ne.0) write (1,'(80a)') bufr
      end
```

Pascal example:

```
program messscall (input, output);

   type buffer = packed array [1..80] of char;
        int = -32768..32767;

   var bufr : buffer;
       lu, count, ic : int;

   function messs (bufr: buffer, count, lu: int) : int;
      external;

   begin
      count := 80;
      bufr := 'RU,WH';
      lu := 71;
      ic := messs (bufr, count, lu);
      if ic <> 0
         then writeln (bufr)
   end.
```

References: Prog Ref Manual

OP SYSTEM COMMAND

ic = MESSS (bufr, count[,lu])

ic – NEGATIVE CHARACTER COUNT OF RETURNED MESSAGE OR 0 IF NO MESSAGE.

bufr – CONTAINS SYSTEM COMMAND ON ENTRY, RETURNED MESSAGE ON RETURN.

count – NUMBER OF CHARACTERS IN bufr.

lu – FOR RU AND XQ REQUESTS ONLY. RUNS THE PROGRAM AS IF FROM THE LU SPECIFIED.

NOTE: only system level commands are available from MESSS.

5.18 Get Program Name -- PNAME

FORTRAN example:

```
      program pnamecall

      integer prog(3)

      call pname(prog)
      write (1,'(6a)') prog
      end
```

Pascal example:

```
program pnamecall (input, output);

   type buffer = packed array [1..6] of char;

   var prog : buffer;

   procedure pname (prog: buffer);
      external;

   begin
      pname (prog);
      writeln (prog)
   end.
```

References: Prog Ref Manual

GET PROGRAM NAME

PNAME (prog)

prog – THREE WORD INTEGER BUFFER
RETURNS PROGRAM'S CLONED NAME

5.19 Get System Time -- FTIME

FORTRAN example:

```
      program ftimecall

      integer bufr(15)

      call ftime(bufr)
      write (1,'(30a)') bufr
      end
```

Pascal example:

```
program ftimecall (input, output);

   type buffer = packed array [1..30] of char;

   var bufr : buffer;

   procedure ftime (bufr: buffer);
      external;

   begin
      ftime (bufr);
      writeln (bufr)
   end.
```

References: Prog Ref Manual

GET SYSTEM TIME

FTIME (bufr)

bufr – 15 WORD INTEGER BUFFER
RETURNS A STRING IN THE
FORM:

2:35 PM WED., 20 JUL., 1983

5.20 Send Logging Message -- LOGIT (VC+)

FORTRAN example:

```
      program logitcall

      integer string(40), strln
      data string/'LOGGING MESSAGE'/, strln/40/

      call logit(string,strln)
      end
```

Pascal example:

```
program logitcall (input, output);

   type buffer = packed array [1..80] of char;
        int = -32768..32767;

   var string : buffer;
       strln : int;

   procedure logit (string: buffer, strln: int);
      external;

   begin
      string := 'LOGGING MESSAGE';
      strln := 40;

      logit (string, strln)
   end.
```

References: Prog Ref Manual

SEND
LOGGING MESSAGE (VC+)

LOGIT (string, strln)

string	– INTEGER BUFFER
strln	– NUMBER OF WORDS IN STRING (CHARACTERS/2)

5.21 Other System Subroutines

NOTE: This is only a small portion of the system subroutines available. See the references noted at the bottom of the page for more information.

References: Relocatable Libraries Ref Manual, Prog Ref Manual

OTHER SYSTEM SUBROUTINES

Loglu –	get LU of scheduling terminal
Casefold –	convert lower to upper case
DecimalToInt –	convert ASCII to integer
IntToDecimal –	convert integer to ASCII
ElapsedTime –	milliseconds since ResetTimer
ResetTimer –	resets elapsed time counter
GetSN –	get a unique session number (VC+)
CLgOn –	programmatic logon (VC+)
CLgOF –	programmatic logoff (VC+)

PROGRAM SCHEDULING

CHAPTER 6

Table of Contents

Chapter 6
PROGRAM SCHEDULING

MODULE OBJECTIVES

1. Understand the process that the RU command goes through in finding and running a program.

2. Be able to use interactive and programmatic commands to schedule, suspend, resume, and terminate a program.

3. Differentiate between immediate and queued scheduling with and without wait.

4. Know how to use time scheduling.

SELF-EVALUATION QUESTIONS

6-1. Under what circumstances are clone names produced?

6-2. A program is run interactively and it's ID segment remains in existance after the program terminates. What can be said about the history of the ID segment? What can be done (interactively) to destroy the ID segment?

6-3. George has a program file called BOZO.RUN::GEORGE. Name two additional file descriptors under which the CI command "ru,bozo" will work properly. Is there any way in which the previous CI command will work properly if the file descriptor is BOZO.RUN::MARTHA?

6-4. What effect does the BR command have on a user program?

6-5. What is the maximum interval at which a program can be rescheduled using the AT command?

6-6. Which EXEC calls can you use to schedule a child program if you expect to receive parameters back from the child?

6-7. What is the difference between queued and non-queued scheduling?

6-8. What happens to the parent program if a child is scheduled with EXEC 9 and the child happens to be busy?

6-9. Under what conditions (using EXEC scheduling) can a parent program terminate before the child is run?

6-10. What does it mean to terminate "serially reuseable"? Under what circumstances would this be no different than normal termination?

6-11. What does it mean to terminate "saving resources"? What happens to the program saving resources if another program needs it's partition?

6-12. When a program is suspended, it is normally resumed using the GO command from the terminal. Can you think of any way to resume the program programmatically? Hint: review the system subroutines from last chapter.

6.1 Cloning

A clone name is generated any time a second copy of the program needs to be run. In a VC+ session environment, the session number is implicitely included as part of the program name. Therefore, programs may have the same name if they are run from two different sessions.

CLONING

CI> ru prog

PROGRAM NAME:

PROG

PRO.A

PRO.B

•

•

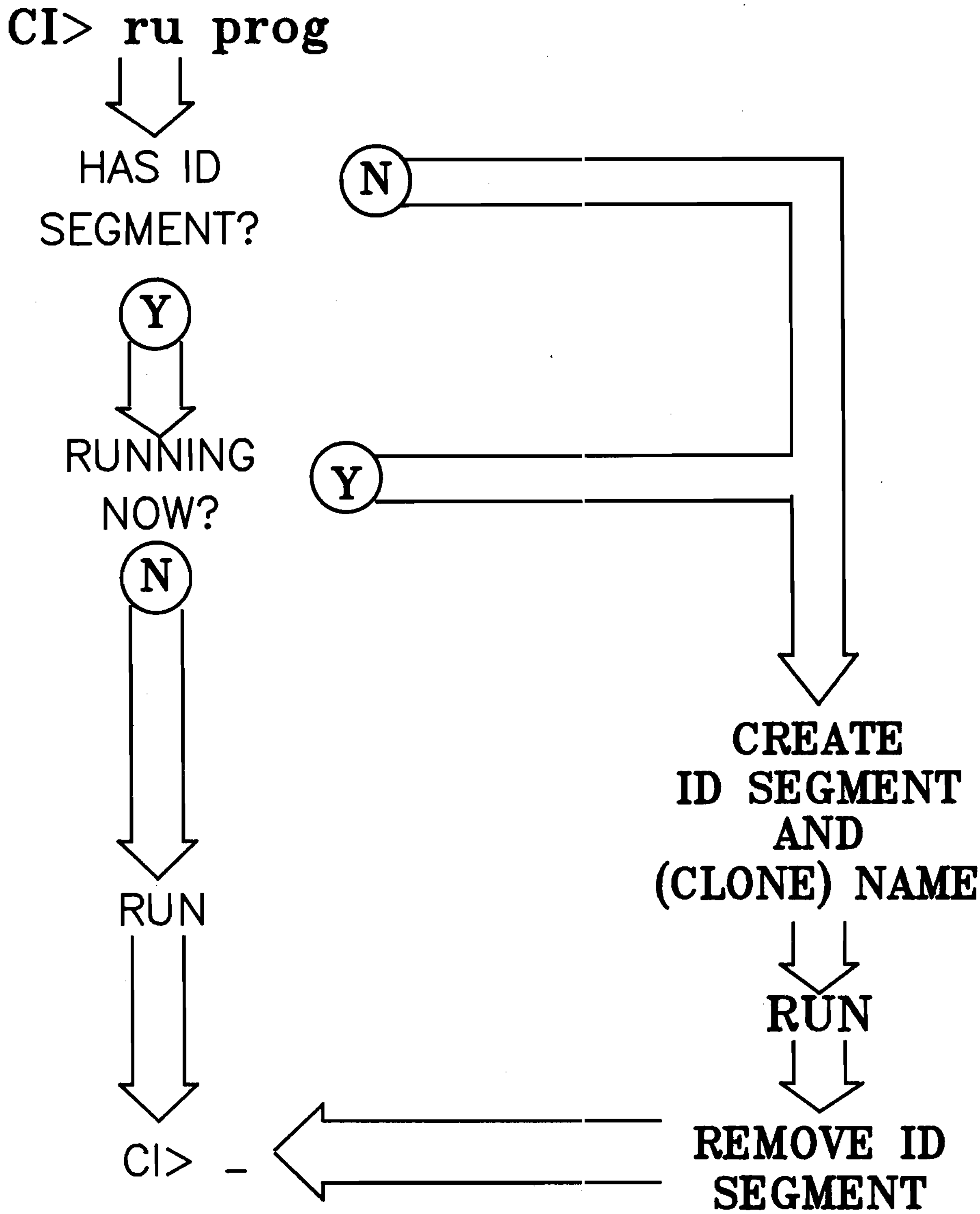
•

6.2 ID Segment Disposition

If an ID segment was created by an RP command, it is not removed upon termination of the program.

If the ID segment was created implicitly with the RU command, it is removed upon termination of the program.

ID SEGMENT DISPOSITION



6.3 Finding the Program

A user can specify <u>no</u> working directory by using the command:

```
CI> wd 0
```

This is useful if you are dealing with FMGR files from CI. If you run a program from CI, the FMGR cartridges are searched first for the program file.

FINDING THE PROGRAM

CI>ru prog

WITH WORKING DIRECTORY:

1. Search ID segment list
2. Search for prog in working directory
3. Search for prog.run in working directory
4. Search for prog.run in directory /PROGRAMS

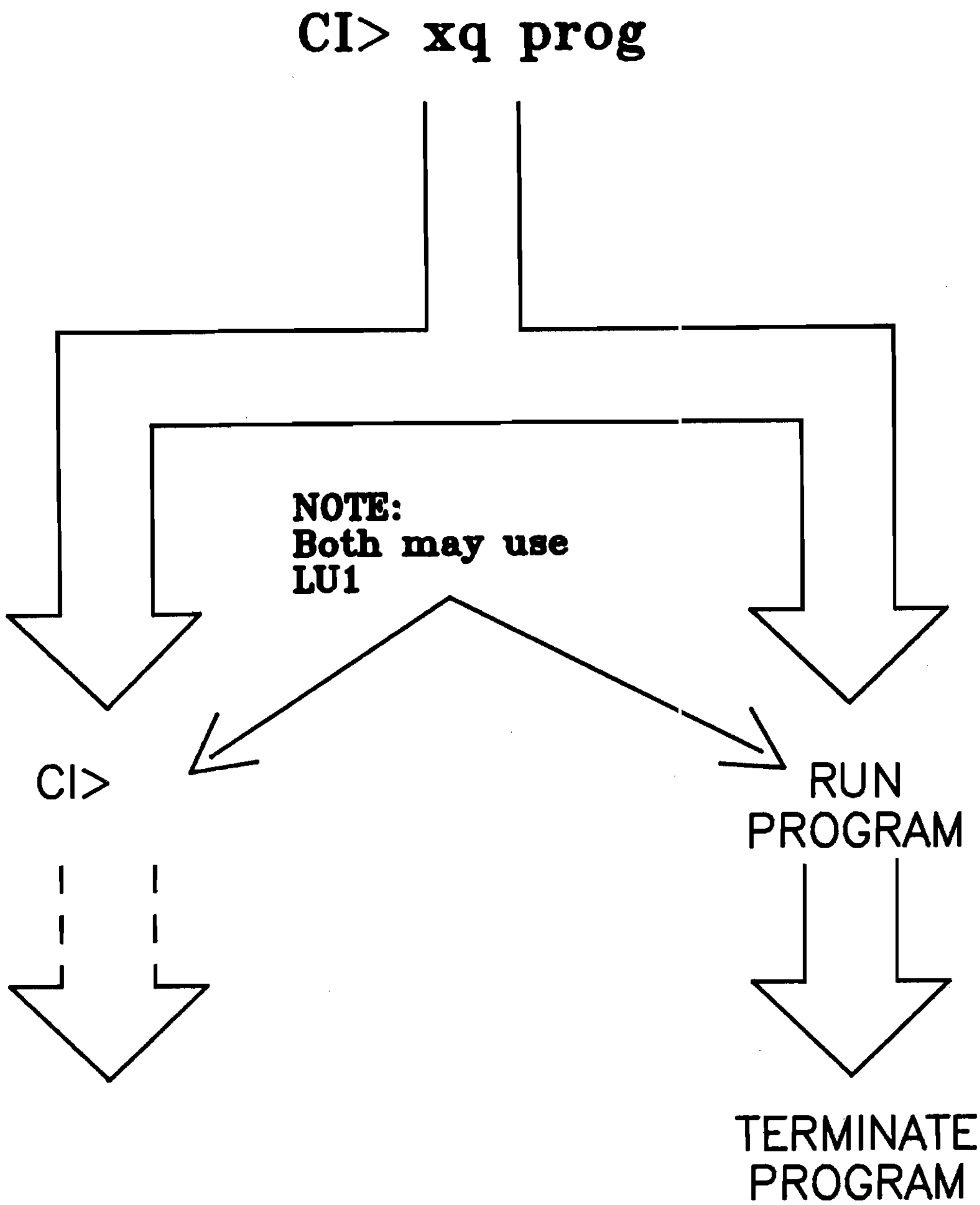
WITHOUT WORKING DIRECTORY: (WD 0)

1. Search ID segment list
2. Search for prog in FMGR cartridge list
3. Search for prog.run in directory /PROGRAMS

6.4 Concurrent Programs

A user may have any number of concurrent programs attached to his session. This is only limited to the maximum number of ID segments allocated in the system (set at system generation time).

CONCURRENT PROGRAMS



6.5 Program Suspension

The BR command only sets the break flag in the program's ID segment. It is up to the user program to check the break flag, otherwise the BR command will have no effect.

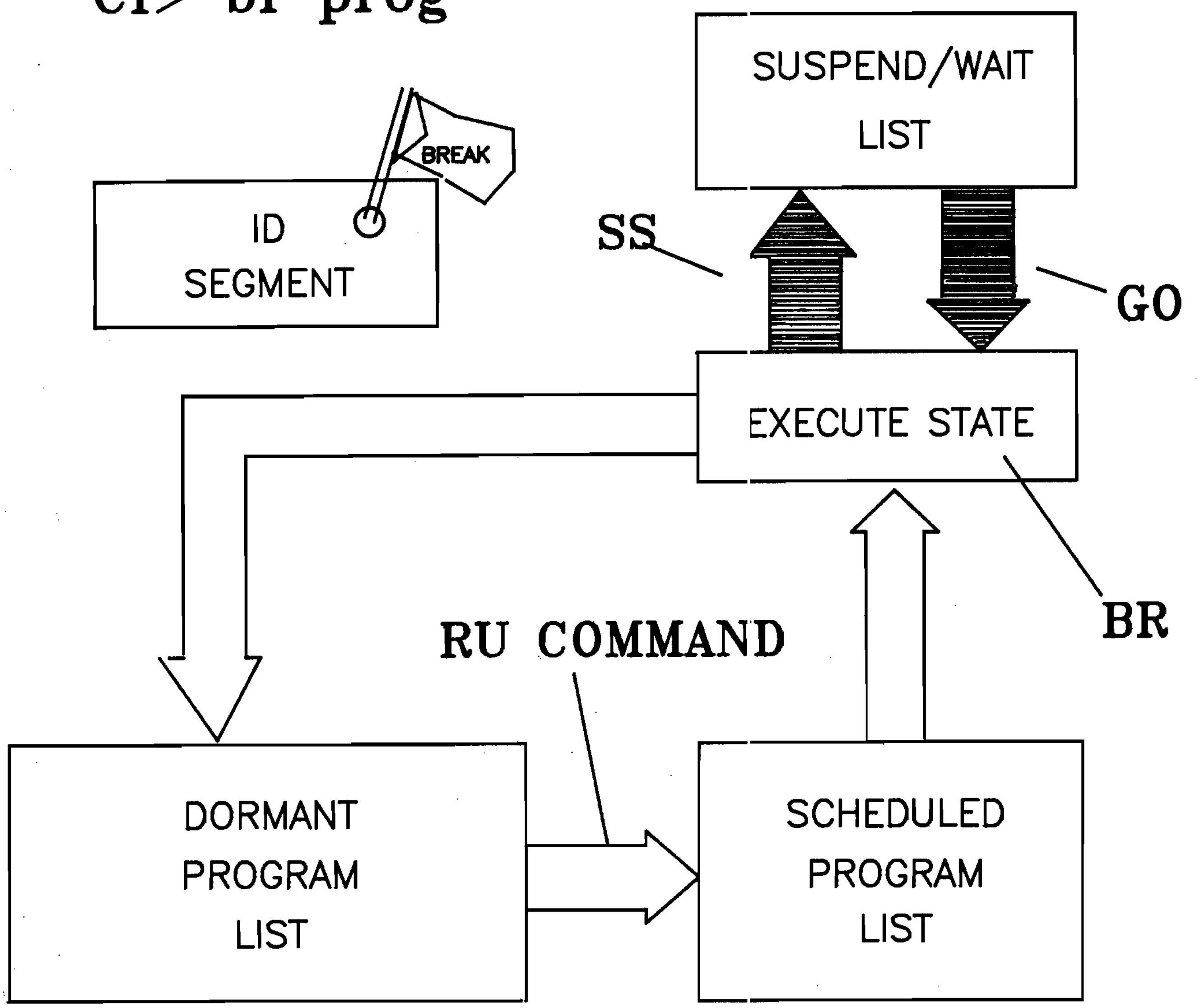
The break flag is checked by use of the IFBRK function. This function is described in the Programmer's Reference Manual.

PROGRAM SUSPENSION

CI> ss prog

CI> go prog

CI> br prog



6.6 Program Termination

If an ID segment has been created using the RP command, the ID segment can be removed using the third parameter to the OF command:

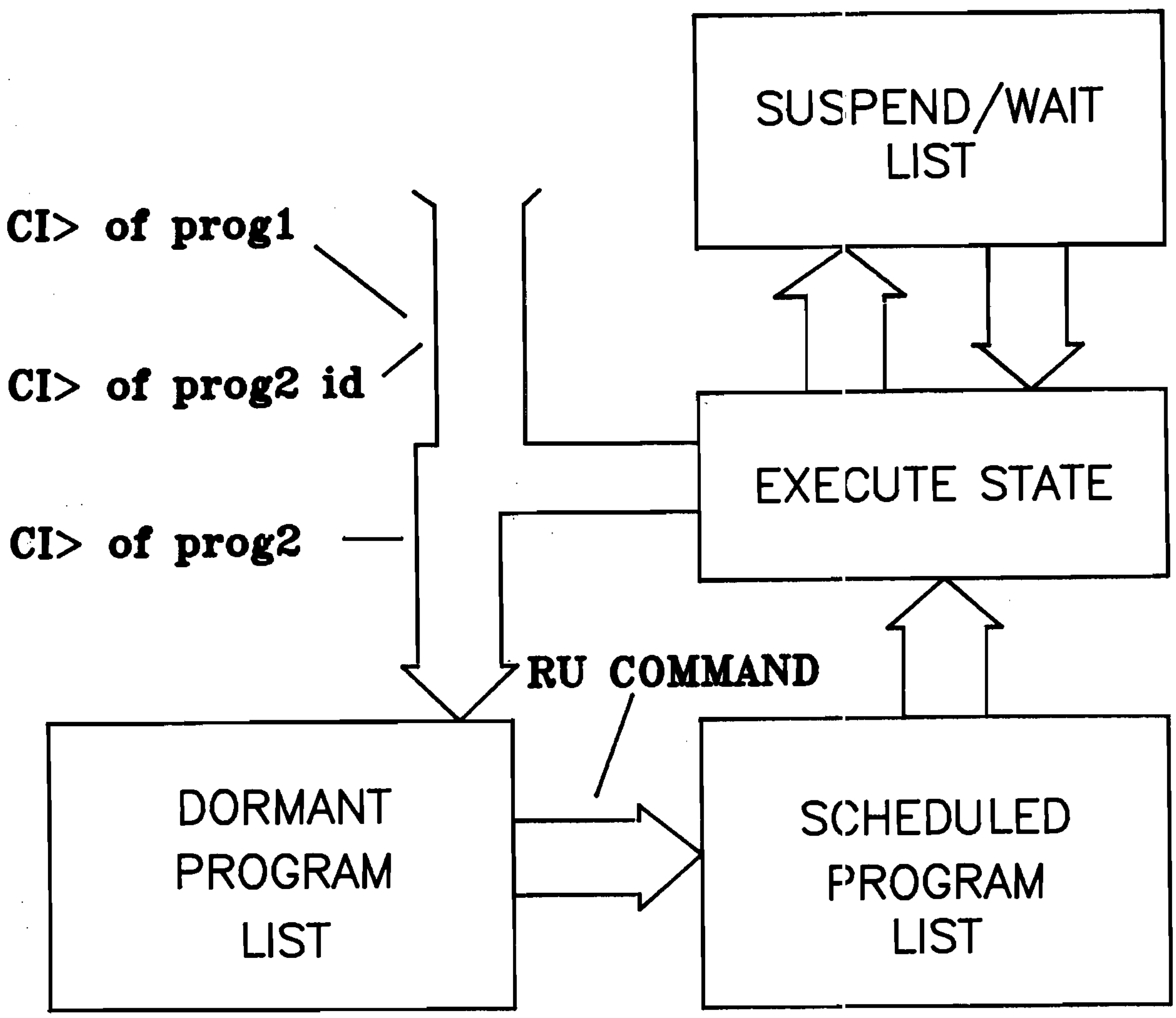
```
CI> of prog2 id
```

PROGRAM TERMINATION

CI> ru prog1 (implicitly RP'ed)

CI> rp prog2

CI> ru prog2

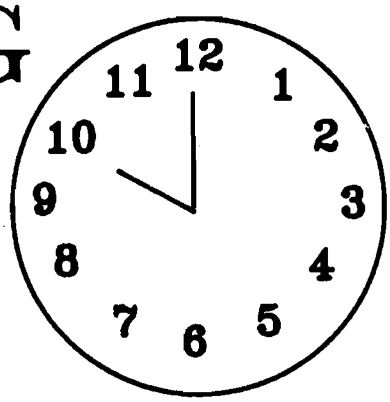


6.7 Time Scheduling

<time> -- Is in 24 hour format unless "am" or "pm" is specified.

<intvl> -- Is in the range 0 to 4095. A maximum of 24 hours may be specified for <intvl>. Greater intervals will be reduced modulo 24 so that 27 hours results in an interval of 3 hours.

TIME SCHEDULING



CI> at <time> prog

CI> at 1:30:00 pm afternoonlog

CI> at 13:30 afternoonlog

CI> at 9 morninglog

CI> at <time> <intvl> prog

CI> at 1:00 pm 1 min timer

CI> at 13 60 sec timer

CI> at 1 27 hour slowtimer

CI> at 10:01 am 150 mil fasttimer

6.8 PROGRAMMATIC SCHEDULING

PROGRAMMATIC SCHEDULING

6.9 Immediate Scheduling -- EXEC 9, 10

Note:

The program must be RPed before for programmatic scheduling to work. The EXEC system has no file handling capabilities.

FORTRAN example:

```
      program Immed_sched

      integer prog(3)
      data prog/'WH'/

      call exec(9,prog)
      end
```

Pascal example:

```
program Immed_sched (input, output);

   type buffer = packed array[1..6] of char;
        int = -32768..32767;

   var prog : buffer;

   procedure exec (ecode: int; prog: buffer);
      external;

   begin
      prog := 'WH';
      exec (9, prog)
   end.
```

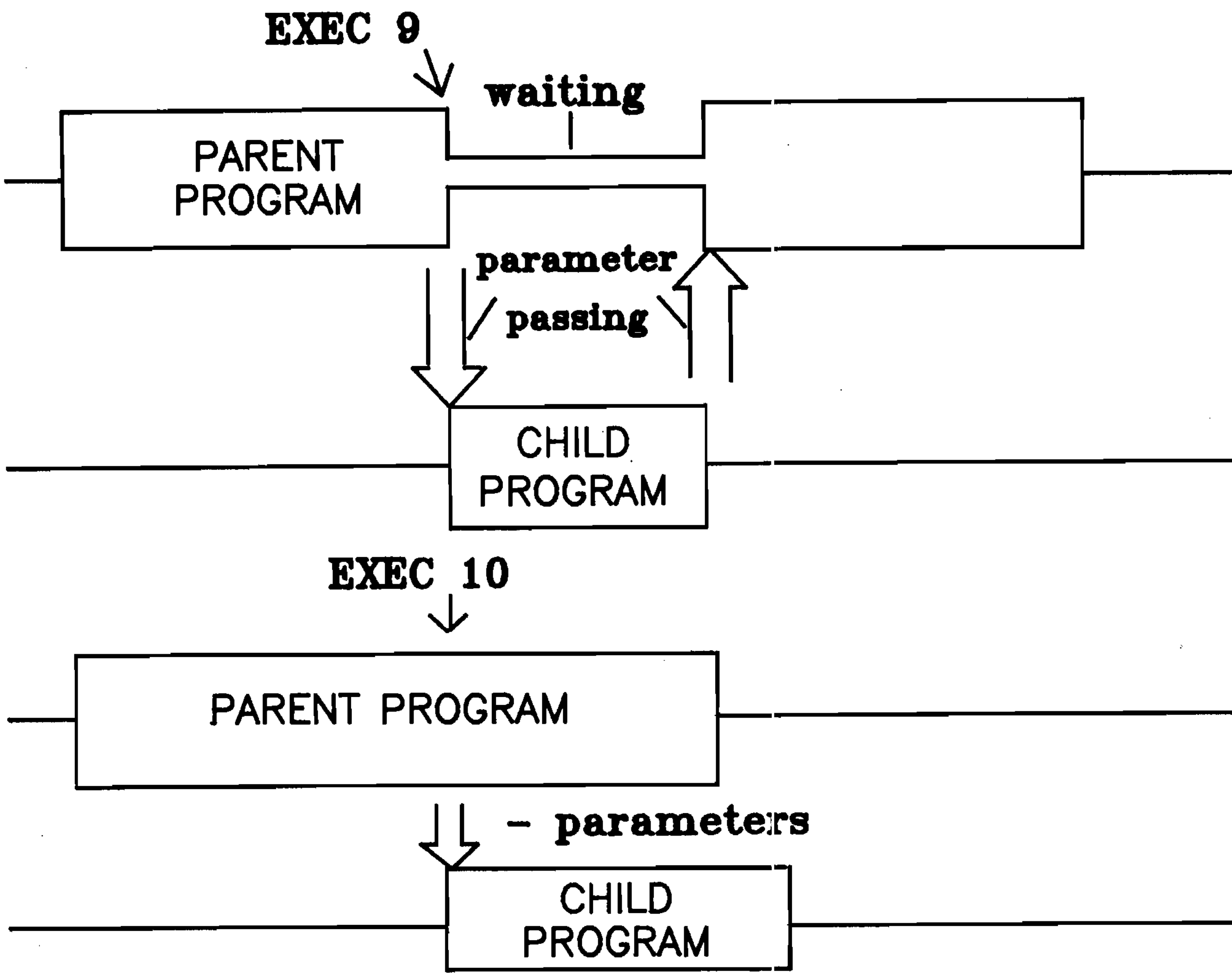
IMMEDIATE SCHEDULING

EXEC (ecode, prog, param*5)

ecode – 9 SCHEDULE WITH WAIT
10 SCHEDULE WITHOUT WAIT

prog – PROGRAM NAME – must be caps

param*5 – UP TO 5 PARAMETERS THAT MAY BE PASSED TO THE SON OR RETURNED FROM THE SON



6.10 Queued Scheduling -- EXEC 23, 24

FORTRAN example:

```
      program Queue_sched

      integer prog(3)
      data prog/'WH'/

      call exec(23,prog)
      end
```

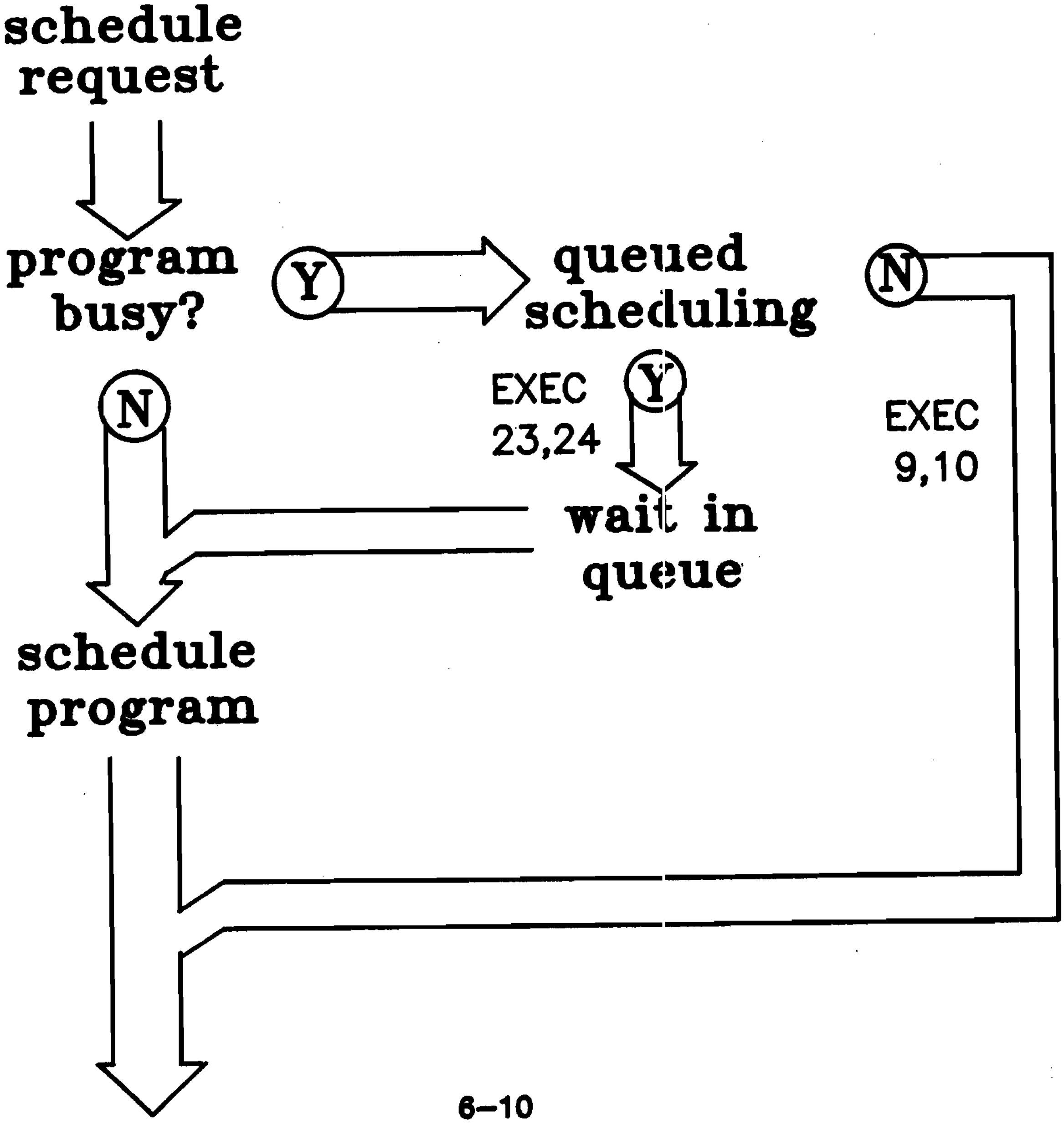
Pascal example:

```
program Queue_sched (input, output);

   type buffer = packed array[1..6] of char;
        int = -32768..32767;

   var prog : buffer;

   procedure exec (ecode: int; prog: buffer);
      external;

   begin
      prog := 'WH';
      exec (23, prog)
   end.
```

QUEUED SCHEDULING

EXEC (ecode, prog, param*5)

ecode – 23 SCHEDULE WITH WAIT
24 SCHEDULE WITHOUT WAIT



6.11 Program Termination -- EXEC 6

FORTRAN example:

```
      program Termination

      integer prog(3)
      data prog/'WH'/

      call exec(10,prog)
      call exec(6,prog,3)
      end
```

Pascal example:

```
program Termination (input, output);

   type buffer = packed array[1..6] of char;
        int = -32768..32767;

   var prog : buffer;

   procedure sched (ecode: int; prog: buffer); $ alias 'exec' $
      external;

   procedure term (ecode: int; prog: buffer, type: int); $ alias 'exec'
      external;

   begin
      prog := 'WH';
      sched (10, prog);
      term (6, prog, 3)
   end.
```

PROGRAM TERMINATION

EXEC (6, prog, type)

prog – 0 FOR CALLING PROGRAM OR PROGRAM NAME OF A CHILD

type – TYPE OF TERMINATION:

- **0** NORMAL TERMINATION
- **1** SAVE RESOURCES
- **−1** SERIALLY REUSABLE
- **2** NORMAL TERMINATION
REMOVE FROM TIME LIST
- **3** NORMAL TERMINATION
REMOVE FROM TIME LIST
REMOVE ID SEGMENT

6.12 Program Suspension -- EXEC 7

FORTRAN example:

```
      program Suspension

      call exec(7)
      end
```

Pascal example:

```
program Suspension (input, output);

   type int = -32768..32767;

   procedure exec (ecode: int);
      external;

   begin
      exec (7)
   end.
```

PROGRAM SUSPENSION

EXEC (7)

same as:

Cl> ss prog

FORTRAN "PAUSE" STATEMENT

continued with:

Cl> go prog

6.13 Time Scheduling -- EXEC 12

FORTRAN example:

```
      program Time_sched

      integer prog(3)
      data prog/'WH'/

      call exec(12,prog,1,0,14,30,0,0)   !at 2:30 pm
      end
```

Pascal example:

```
program Time_sched (input, output);

   type buffer = packed array[1..6] of char;
        int = -32768..32767;

   var prog : buffer;

   procedure exec (ecode: int; prog: buffer; units, intvl,
                   hour, min, sec, msec: int);
      external;

   begin
      prog := 'WH';
      sched (12, prog, 1, 0, 14, 30, 0, 0)  {at 2:30 pm}
   end.
```

TIME SCHEDULING

EXEC (12, prog, 1, 0, hour,
min, sec, msec)

prog– PROGRAM NAME

hour **min** **sec** **msec**	TIME AT WHICH TO START THE PROGRAM

6.14 Interval Scheduling -- EXEC 12

FORTRAN example:

```
      program Intvl_sched

      integer prog(3)
      data prog/'WH'/

      call exec(12,prog,3,10,14,30,0,0)   !at 2:30 pm ...
      call exec(12,prog,3,10,-10)         !10 min from now ....
      end                                 !... then every 10 min
```

Pascal example:

```
program Intvl_sched (input, output);

   type buffer = packed array[1..6] of char;
        int = -32768..32767;

   var prog : buffer;

   procedure exec (ecode: int; prog: buffer; units, intvl,
                   hour, min, sec, msec: int);
      external;

   begin
      prog := 'WH';
      sched (12, prog, 3, 10, 14, 30, 0, 0) {at 2:30 pm ...}
      sched (12, prog, 3, 10, -10, 0, 0, 0) {10 min from now ...}
   end.                                     {... then every 10 min}
```

INTERVAL SCHEDULING

EXEC (12, prog, units, interval,
hour, min, sec, msec)

prog– PROGRAM NAME

units–
- **0** REMOVE FROM TIME LIST
- **1** TENS OF MILLISECONDS
- **2** SECONDS
- **3** MINUTES
- **4** HOURS

interval– **0 TO 4095**
THE INTERVAL OF "units" AFTER WHICH THE PROGRAM WILL BE REPEATED.
0 INDICATES NO REPEAT

hour
min
sec
msec
TIME AT WHICH TO START THE PROGRAM

–hour NUMBER OF "units" TO DELAY STARTING THE PROGRAM. min, sec, msec NOT SPECIFIED

6.15 TIME FUNCTIONS

TIME

FUNCTIONS

6.16 Time Retrieval -- EXEC 11

FORTRAN example:

```
      program Time_retrieval

      integer time(5), year

      call exec (11,time,year)
      write (1,'(i5)') year
      do i = 5,1,-1
         write (1,'(i5)') time(i)
      end do
      end
```

Pascal example:

```
program Time_retrieval

   type int = -32768..32767;
        int_array = array [1..5] of int;

   var time : int_array;
       year, i : int;

   procedure exec (ecode: int; time: int_array; year: int);
      external;

   begin
      exec (11, time, year);
      writeln (year);
      for i := 5 downto 1 do
         writeln (time[i])
   end.
```

TIME RETRIEVAL

EXEC (11,time, year)

time (1)	TENS OF MILLISECONDS
time (2)	SECONDS
time (3)	MINUTES
time (4)	HOURS
time (5)	DAY OF THE YEAR

year – OPTIONAL PARAMETER

6.17 Setting System Time -- SETTM

FORTRAN example:

```
      program Set_time

c     set time to 2:30 pm, April 1, 1984
      call settm (14,30,0,4,1,84)
      end
```

Pascal example:

```
program Set_time

   type int = -32768..32767;

   procedure settm (hr, min, sec, mo, day, yr: int);
      external;

   begin
      {set time to 2:30 pm, April 1, 1984}
      settm (14, 30, 0, 4, 1, 84);
   end
```

SETTING SYSTEM TIME

error = SETTM (hr, min, sec,
mo, day, yr)

error:	0	NO ERRORS
	− 1	ILLEGAL PARAMETER TIME NOT CHANGED

hr	**0 to 23**
min	**0 to 59**
sec	**0 to 59**
mo	**1 to 12**
day	**1 to 31**
yr	**1976 to 2144**

SETTING SYSTEM TIME

error = SETTM (hr, min, sec,
mo, day, yr)

error:	0	NO ERRORS
	– 1	ILLEGAL PARAMETER TIME NOT CHANGED

hr	**0 to 23**
min	**0 to 59**
sec	**0 to 59**
mo	**1 to 12**
day	**1 to 31**
yr	**1976 to 2144**

PROGRAM COMMUNICATION

CHAPTER 7

Table of Contents

Chapter 7
Program Communication

MODULE OBJECTIVES

1. Ability to explain the methods available for program communication under RTE-A when passing information and when sharing data.

2. Ability to use the routines that are available for parameter passing for interactive communication.

3. Ability to understand and use communication techniques of Parent-Child programs -- sending and receiving parameters.

4. Ability to use one method -- System Common Area -- while being aware of different methods for using shared data.

SELF-EVALUATION QUESTIONS

7-1. What are the differences between scheduling with WAIT and without WAIT with reference to parameter passing?

7-2. How does a parent program send parameters to a child program?

7-3. What are the different ways a child can return information to the parent program?

7-4. Can parameters be passed to a program via an EXEC 6?

7-5. What routine is used to pick up parameter values passed in the run command? In FORTRAN? In Pascal?

7-6. What is the difference between an EXEC 14 call and the GETST subroutine?

7-7. How does a user specify that the program should use System Common Area?

7-8. When would you use labeled system common area as opposed to blank SCA?

7-9. What are System Common Area limitations and restrictions when used for shared data?

7-10. What methods were discussed in this chapter for passing information for shared data from program to program and from CI to program?

7.1 Communication Considerations

When selecting the appropriate communication routines, the programmer must consider the following: the kind of transfer -- interactive, or from program to program, the method of synchronization of programs, the amount of information being transferred. Next, the programmer must concern himself/herself with such questions as: what resources are available -- SAM, class numbers and resource numbers (see chapters 8,12). Can the programs be swapped? Furthermore, the programmer must analyze the methods for sharing data before he/she finally chooses communication routines best suited for his/her application.

This chapter will cover interactive program communication and one method of passing small amounts of information between parent and child programs. Note that program to device I/O communication has been partially discussed in Chapter 5 with EXEC calls. This chapter also explains the first method of sharing data - system common area. The topic of data will be discussed again in later chapters.

COMMUNICATION CONSIDERATIONS

* amount of information in transfer

* synchronization of transfer

* swappability of communicating programs

* resources available

* sharing data

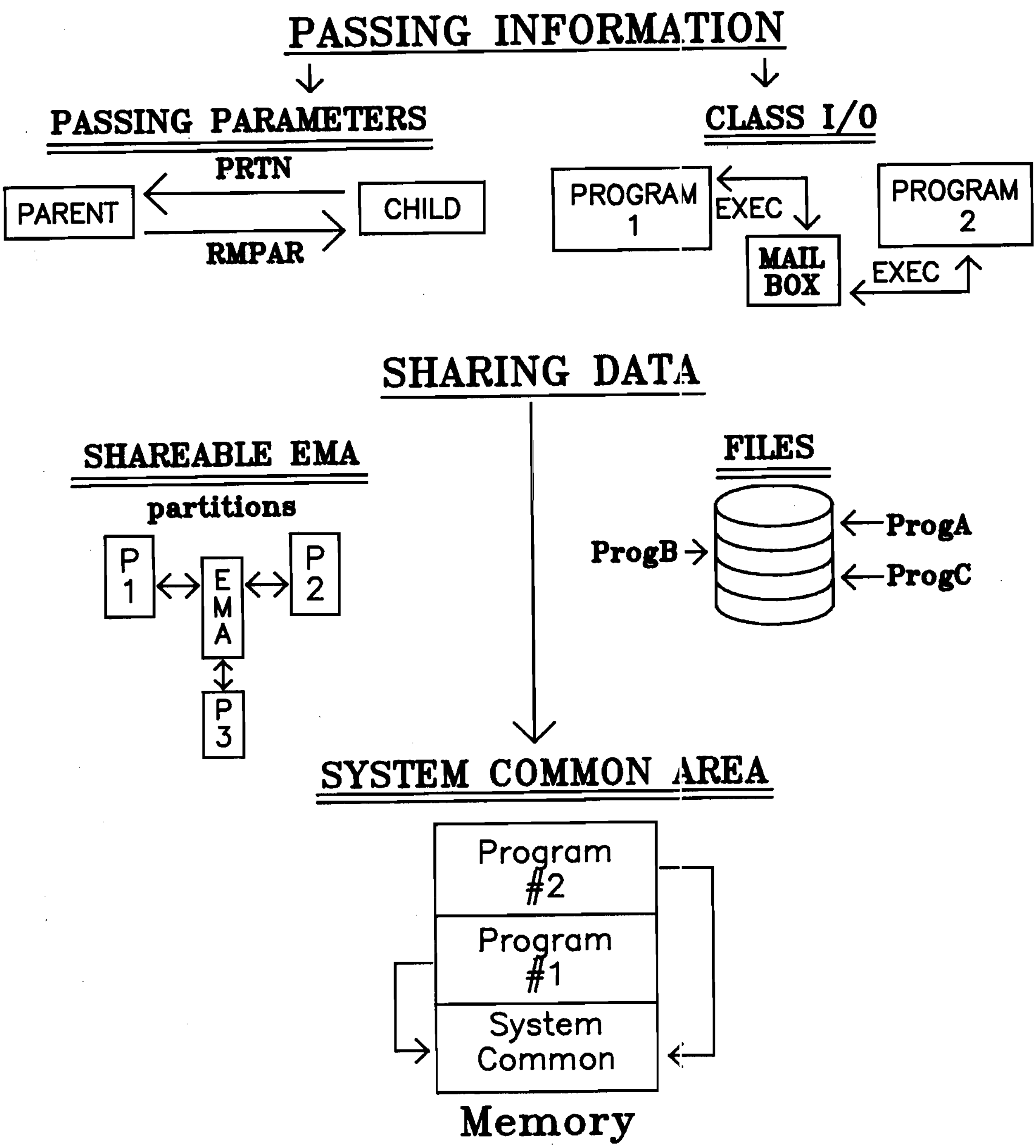
7.2 Communication Methods

There are many ways to communicate among programs. One method, Parameter Passing, will be described in this chapter and is useful for a small number of parameters passed interactively, or small arrays of data passed between certain programs (i.e., Parent-Child).

Class I/O, or mailbox I/O, passes information between programs and aids in synchronizing program communication. It will be discussed in the next Chapter.

Sharing of data can be accomplished through various techniques. One, shareable EMA involves sharing an entire partition and is presented in Chapter 11. Another, files, allows multiple to be shared by many programs. System Common Area is the method that will be described in this chapter.

COMMUNICATION METHODS



7.3 PASSING INFORMATION

There are many ways to pass values in programs. In this course we will be describing passing information from programs to other programs, programs to devices, and CI to programs.

PASSING INFORMATION

Values may be passed to:

* **Subroutines**

  CALL SUBR (I,J,K)
  SUBR (I,J,K : INTEGER);

* **Programs when run**

  CI> PROG.RUN 1 2 3 4

* **Programs from other programs**

7.4 Passing Information Interactively

The routines used when passing parameters interactively depend upon the amount of data to be transferred. Up to five integer values or pairs of ASCII characters can be passed interactively to a program and retrieved by a call to RMPAR. Parameter strings may be passed to a program and be retrieved by GETST. The entire runstring may be retrieved by an EXEC 14 call.

PASSING INFORMATION INTERACTIVELY

HOW MUCH DATA IS TO BE TRANSFERRED?

* **Up to 5 integer values or pairs of ASCII characters**

* **Parameter string**

* **Runstring**

7.5 Passing Parameters - RMPAR

RMPAR is a general purpose request. It recovers parameters which have been passed to the calling program and which were stored in temporary words of the ID segment. These parameters may have originated by operator run commands (interactive), program scheduling request or by some drivers.

Only integer or pairs of ASCII characters can be passed with RMPAR. The size of the array for RMPAR must be 5. Also, it is very important that the first executable statement in the program be RMPAR since other system routines use this area of the ID segment and therefore the data stored there may change.

Notice, that from CI a program can only receive data interactively. It cannot send parameters back to CI. With FMGR, however, a program can send back information to FMGR with the routine PRTN. RMPAR can also be used to retrieve five parameters from the GO command if it has been suspended (EXEC 7).

References: Programmer's Reference Manual

RMPAR

UP TO **5** VALUES INCLUDED AS PARAMETERS IN THE **RU** COMMAND

CI> **RU,LISTR.RUN,6,TE,XT,F4**

PROG's
ID SEGMENT

6
TE
XT
F4

PROGRAM PROG
INTEGER PARM (5)
CALL RMPAR (PARM)

7.6 Passing Parameters - FORTRAN 77

RMPAR may be easily called from FORTRAN, however, FORTRAN also provides a routine FPARM which will copy the parameters into character variables, array elements or substrings for you. It must be the first executable statement. The first time FPARM is called it makes an EXEC 14 call (next slide) to get the runstring and to store it into an internal buffer.

Subsequent FPARM calls access this internal buffer. Thus multiple calls to FPARM may be made, while only one RMPAR call is possible. (Notice, that a call to EXEC 14 or GETST after a call to FPARM will not return the runstring, and that FPARM will not work after an explicit call to EXEC 14 or GETST since the runstring would already by consumed.)

Syntax:

```
Call FPARM (vl,...,vn)
```

where:

vl..vn — are character, character array, or string variables and are positioned to the parameter location in the runstring.

NOTE: CI takes blank or commas as variable separaters and shifts all characters to upper case.

References: FORTRAN 77 Reference Manual

PASSING PARAMETERS – FORTRAN 77

FPARM – TO COPY RUNSTRING PARAMETERS

```
PROGRAM param
CHARACTER*10 str,str1,str2
CALL FPARM (str,str1,str2)
WRITE (1,*) str,str1,str2
END
```

```
CI> param.run here there everywhere
HERE THERE EVERYWHERE
CI> param.run 1 2 3 4
1   2   3
```

7.7 Passing Parameters Pascal

Pascal provides a library, Pas.NumericParms, which returns RMPAR parameters. It is provided since Pascal run-time startup code makes the values of the parameters stored in the ID segment unreliable. For integer parameters, the routine takes one VAR parameter, a 5-element array of one-word integers, and returns RMPAR parameters. For character parameters, it is better to use Pas.Parameters as will be shown later. Pas.NumericParms should only be used to pick up the initial parameters, thus a program terminating serially reusable should use RMPAR directly. As will be shown, RMPAR, not Pas.NumericParms, can be used for parent-child communication.

References: Pascal/1000 Reference Manual

PASSING PARAMETERS – PASCAL

Pas.NumericParms – RETURNS RMPAR PARAMETERS

```
PROGRAM param(INPUT,OUTPUT);
TYPE int = -32768..32767;
     index = 1..10;
     parms = PACKED ARRAY [1..5] OF int;
VAR i : index;
    p : parms;
PROCEDURE Pascal_Rmpar $ALIAS 'Pas.NumericParms'$
   (VAR p : parms);
  EXTERNAL;
BEGIN
  Pascal_Rmpar(p);
  WRITE ('The parameters are: ');
  FOR i := 1 to 5 DO
     WRITE(p [i] :3);
  WRITELN;
END.
```

```
CI> param.run 1 1 2 3
the parameters are: 1 1 2 3
CI> param.run 1 1 5 6 7
the parameters are: 1 1 5 6 7
```

7.8 Passing Strings - EXEC 14/GETST

The EXEC 14 will retrieve the command that scheduled the program:

```
                              CALL EXEC (14,1,BUFR,BUFLN)
                                         ^   ^    ^
    indicator to retrieve the runstring--+   |    |
  array to receive the runstring-------------+    |
number of words or negative number of characters--+
```

FORTRAN Example:

```
      PROGRAM GRSTR
      INTEGER IBUF(35)
C       Retrieve the runstring via EXEC 14.  Specify
C       the maximum number of words to be retrieved.
      CALL EXEC (14,1,IBUF,35)
      CALL ABREG (IA,IB)
      ILOG = IB
C       Print the runstring, using the actual
C       number of words retrieved.
      WRITE(1,*)'THE RUNSTRING IS: '
      WRITE(1,'(35A2)')(IBUF(J),J=1,ILOG)
      END
```

```
CI> GRSTR.RUN  This is a string
    THE RUNSTRING IS:
    RU,GRSTR,THIS,IS,A,STRING
CI>
```

Calls to EXEC 14 from Pascal may be made. Not advised for interactive use.

A call to GETST will retrieve the parameter string part of the command that scheduled the program.

```
                                    CALL GETST (BUFR,BUFLN,TLOG)
                                                ^    ^     ^
     array to receive the parameters string--+     |     |
   positive # of words or negative # of characters--+     |
number of words or characters actually retrieved---------+
```

Again, both of these routines should be the first executable statement in the program.

References: Programmer's Reference Manual

PASSING STRINGS

STRINGS OF CHARACTERS MAY BE
PASSED TO A PROGRAM –

EXEC 14 –
RETRIEVE THE
"RUNSTRING"

CI> RU GRSTR THIS IS A STRING

GETST –
RETRIEVE THE
"PARAMETER STRING"

7.9 Passing Strings - FORTRAN 77

From FORTRAN, GETST and EXEC 14 may by called. The FORTRAN libraries RCPAR and RHPAR may be used if desired. They return the specific parameter desired (as FPARM, although only one parmeter at a time may be retrieved). If the parameter number is 0, the program name will be retrieved.

For Example:

```
              PROGRAM RCP
              CHARACTER FILE1*20
              CALL RCPAR(0,FILE1)
              WRITE(1,*) FILE1
              END
        CI>   RCP.RUN 1 2 3
              RCP.RUN
        CI>
```

References: Programmer's Reference Manual

PASSING STRINGS – FORTRAN 77

```
      PROGRAM GPSTR
      INTEGER IBUF(35)
C
C     Retrieve the runstring via GETST. Specify
C     the maximum number of words to be retrieved.
C
      CALL GETST (IBUF,35,ILOG)
C
C     Print the runstring, using the actual number
C     of words retrieved.
C
      WRITE(1,*) 'THE PARAMETER STRING IS: '
      WRITE (1,'(35A2)') (IBUF(J),J=1,ILOG)
C
      END
```

```
CI> GPSTR.RUN This is a string
    THE PARAMETER STRING IS:
    THIS,IS,A,STRING

CI>
```

7.10 Passing Strings - Pascal

In Pascal, EXEC 14 and GETST may be called. To insure the Pascal initialization code will not change the ID segment information, the programmer may use the RUN_STRING 0 option to turn off Pascal initialization code. This method is NOT recommended because the standard Pascal I/O system will not work. Thus, either Pas.NumericParms or Pas.Parameters is provided by Pascal and is the recommended method. Pas.NumericParms picks up 5 integer or pairs of ASCII data (RMPAR) and Pas.Parameters picks up character strings (EXEC14).

Pascal GETST Example:

```
$run_string 0$
PROGRAM GETPAS:
TYPE
    IBUFFY = PACKED ARRAY[1..70] OF CHAR;
    INT = -32768..32767;
VAR IBUF : IBUFFY;
    I ,
    ILEN,ILOG : INT;
    OUT : TEXT;
 PROCEDURE GET_STRING $ALIAS 'GETST'$
   (VAR IBUF : IBUFFY;
        ILEN : INT;
    VAR ILOG : INT);
 EXTERNAL;
 BEGIN
 get_string (ibuf, -70, ilog);
 REWRITE(OUT,'1');
 WRITELN(OUT, ibuf);
 END.

 CI>  GETPAS.RUN  HI THERE CUTEY PIE HOW ARE YOU?
      HI,THERE,CUTEY,PIE,HOW,ARE,YOU?
 CI>
```

Other system routines that are provided for sending and retrieving and manipulating information are:

```
PARSE  - parse input buffer from ASCII representation
INPRS  - inverse parse - parse back to ASCII
LOGLU  - get logical unit of invoking terminal
MESSS  - message processor interface
LOGIT  - send logging message
PNAME  - retrieve program name
IDGET  - retrieve program ID segment address
```

References: Pascal/1000 Reference Manual

PASSING STRINGS – PASCAL

```
PROGRAM pas;
TYPE
    index = 1..10;
    int = -32768..32767;
    parms = PACKED ARRAY [1..80] OF CHAR;
VAR
    out : TEXT;
    p : parms;
    i, length, position : int;
FUNCTION Pascal_Parms $ALIAS 'Pas.Parameters'$
    (  position : int;
    VAR p : parms;
       length : int) : int;
    EXTERNAL;
BEGIN
position :=-1;   (*-1 gives the entire runstring*)
length :=80;
    i :=Pascal_Parms(position,p,length);
REWRITE(out,'1');
WRITELN(out,'the parameters are: ',p);
END.
```

```
CI> Pas.run THIS IS FUN
    the parameters are: RU,PAS.RUN,THIS,IS,FUN
CI>
```

7.11 Passing Information Programmatically

The EXEC 9,10,23 and 24 calls allow a program to schedule another program. The scheduler is called the parent and the program which is scheduled is called the child. As a review:

EXEC 9 - immediate schedule, wait for completion
EXEC 10 - immediate schedule, no wait
EXEC 23 - queue schedule, wait
EXEC 24 - queue schedule, no wait

Schedule with wait implies the parent waits for the child to complete before resuming execution. The child can return information to the parent. Schedule without wait implies that the parent does not wait for the child to complete. Thus the parent will continue and will compete for execution time with the child on a priority basis and the child will not be able to send information back to the parent.

PASSING INFORMATION
PROGRAMMATICALLY

PARENT

CHILD

parent can pass information
to its child

-----OR-----

PARENT

CHILD

parent can pass information
to its child

-AND-

the child can pass information
back to its parent

7.12 Parent-Child Communication

The parent can send information to the child via EXEC scheduling calls.

```
     CALL EXEC (ECODE,NAME,PRAM,PRAM2,PRAM3,PRAM4,PRAM5,BUFR,BUFLN)
                  ^    ^    |                             | |     |
     9,10,23,24---+    |    +-----------v-------------+  +---v----+
                       |    PRAM1 to PRAM5 are          An array of data
  array name of program-+   optional parameters         can be passed to
  to be scheduled           whose values are            the child via
                            passed to the child.        BUFR.  The child
                            The child uses RMPAR        can use EXEC 14
                            to retrieve the value.      or GETST to re-
                                                        trieve the data.
```

If the parent schedules the child with wait, the child can return information to the parent as follows:

Via PRTN and RMPAR

- child calls PRTN to pass 5 values back to 'waiting' parent.

- parent retrieves the values with a RMPAR call.

Via EXEC 14

- child uses EXEC 14 call to pass a buffer of data back to "waiting" parent.

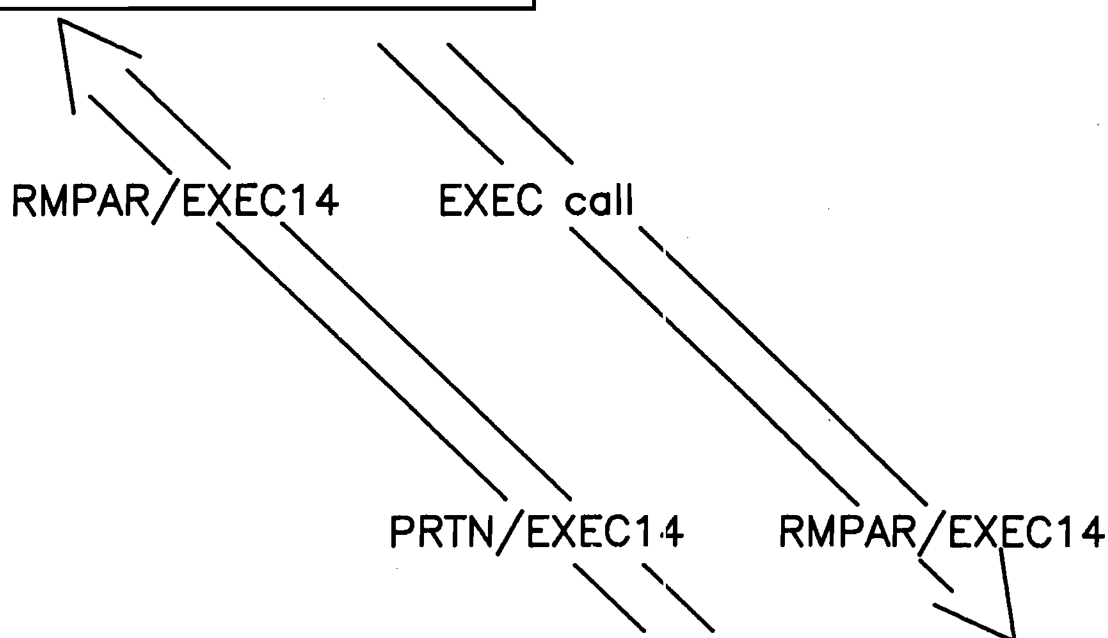
- parent retrieves the buffer with another EXEC 14 call.

References: Programmer's Reference Manual

PARENT ↔ CHILD COMMUNICATION

PARENT WITH WAIT

- input a set of values
- **schedule child to sort** and delete duplicate values
- print sorted values



sort values and
delete duplicate values

CHILD

7.13 Pascal Example - Parent

```
Program mom (input,output);
   TYPE
      int = -32768..32767;
      typ100 = array[1..100] of int;
      typ6   = packed array[1..6] of char;
      ptype = packed array [1..5] of int;
   VAR
      child : typ6;
      tdata : typ100;
      i,ivals,a,b,w,x,y,z : int;
      parms : ptype;
   PROCEDURE exec23 $alias 'xluex' $
      (ecode:int ; child:typ6 ; lu,w,x,y,z:int ; msg:typ100; len:int);
      external;
   PROCEDURE rmpar $ alias 'rmpar' $
      ( var parms: ptype ); external;
   PROCEDURE exec14 $ alias 'exec' $
      (ecode,rcode:int ; var tdata:typ100 ; var ivals:int); external;
   PROCEDURE abreg $ alias 'abreg' $
      (var a,b:int); external;
   BEGIN

     { INPUT DATA FROM THE TERMINAL  }
      writeln ('How many integer values do you wish to input: ');
      read ( ivals );
      writeln ('Input ', ivals, ' values: ');
      for i := 1 to ivals do
         read ( tdata[i] );

     { SCHEDULE THE CHILD PROGRAM WITH WAIT }
      child := 'sonpg ';
      exec23(23,child,ivals,w,x,y,z,tdata,ivals);

  {    LET SON SORT VALUES AND RETURN TO MOM NEW NUMBER OF VALUES }
      rmpar ( parms );
      ivals := parms[1];
      writeln(ivals);

     { RETRIVE SORTED DATA FROM THE CHILD }
      exec14(14,1,tdata,ivals);
      abreg ( a, b );
      if a = 0 then begin
         for i := 1 to ivals do
            writeln ( tdata[i] );
      end else writeln ('no sorted data received from child');
   END.
```

THE PARENT

```
      program mom
      integer nson(3), tdata(100), parm(5)
      data nson/6HSONPG /
C
C INPUT DATA FROM TERMINAL
C
      read (1,*) ivals, (tdata(i),i=1,ivals)
C
C SCHEDULE SON WITH WAIT
C
      Call exec(23,nson,ivals,j,k,l,m,tdata,ivals)
C
C                                        TO SON EXEC 14 -->
C
C                                TO SON RMPAR -->
C
C
C LET SON SORT VALUES AND RETURN TO MOM NEW
C NUMBER OF VALUES
C
C                            FROM SON PRTN <--
      Call rmpar(parm)
      ivals=parm(1)
C
C RETRIEVE THE SORTED DATA
C                                    FROM SON EXEC 14 <--
      Call exec(14,1,tdata,ivals)
      Call abreg (ia,ib)
      if (ia .eq. 0) then
          write (1,50)(tdata(j),j=1,ivals)
50        format (10(10(i5,1x)/))
      else
          write (1,'(" No sorted data received from son")')
      end if
      end
```

7.14 Pascal Example - Child

```
$run_string 0$
Program sonpg;
   TYPE
      int = -32768..32767;
      ptype = packed array[1..5] of int;
      typ100 = array[1..100] of int;
   VAR
      i, ivals, a, b : int;
      parms : ptype;
      tdata : typ100;
      out : text;

   PROCEDURE rmpar $ alias 'rmpar' $ (var parms:ptype); external;
   PROCEDURE prtn $ alias 'prtn' $
          ( var parms:ptype); external;
   PROCEDURE exec14 $alias 'exec' $
      (ecode:int ; rcode:int ; var tdata:typ100 ; var ivals:int);
      external;
   PROCEDURE abreg $ alias 'abreg' $
      (var a,b : int);  external;
   BEGIN

    { RETRIEVE THE NUMBER OF VALUES TO BE SORTED }
      rmpar(parms);
      ivals := parms[1];
      rewrite(out,'1');

    { PICK UP THE ARRAY OF DATA VALUES }
      exec14(14,1,tdata,ivals);
      abreg(a,b);
      if a = 0 then begin

    { SORT THE VALUES, DELETE DUPLICATE VALUES }
         exec14(14,2,tdata,ivals);
         abreg(a,b);
         if a = 0 then begin

    { RETURN THE SORTED VALUES TO THE MOM }
            parms[1] := ivals;
            prtn (parms);
         end else
            writeln(out,'No mom found to accept results');
      end else
         writeln(out,'No data buffer from mom found');
   END.
```

THE CHILD

```
      program sonpg
      integer values(100), parm(5)
C
C RETRIEVE THE NUMBER OF VALUES TO BE SORTED
C
C  ------------------------------> FROM MOM EXEC 23
C
      call rmpar(parm) <-----------
      nvals = parm(1)
C
C PICK UP THE ARRAY OF DATA VALUES
C
C  ------------------------------> FROM MOM EXEC 23
C
      call exec (14,1,values,nvals)
      call abreg (ia,ib)
      if (ia .eq. 0) then
C
C SORT THE VALUES, DELETING DUPLICATE VALUES
C (Values will then contain the sorted data,
C  nvals will contain the new number of values)
C <-------------------------------TO MOM EXEC 14
      call exec (14,2,values,nvals)
      call abreg (ia,ib)
      if (ia .eq. 0) then
C
C RETURN THE SORTED VALUES TO THE MOM
C
C <---------------------BACK TO MOM'S RMPAR
C
            parm(1) = nvals
            call prtn(parm) >-----
      else
            write (1,'("No mom found to accept results")')
      else
            write (1,'("No data buffer from mom found")')
      end if
      end if
      end
```

7.15 Program Termination

A parent can terminate itself or a child with an EXEC 6 call. Parameters may be stored in a program's ID segment when terminated with an EXEC 6 call. If its ID segment is not cleared, the parameters PRAM1 and PRAM5 are passed back to the program when it is next scheduled. They can be picked up by RMPAR when the program executes next. Thus, a program in the timelist can pass parameters to itself. Parameters cannot be passed to a child terminated by an EXEC 6 call.

PROGRAM TERMINATION

PARENT EXEC6 → CHILD

EXEC6

CALL EXEC(6,PROG,TYPE,P1,...,P5)

ITSELF – 0
CHILD – PROGRAM NAME

0 NORMAL TERMINATION
–1 SERIALLY REUSABLE
1 SAVING RESOURCES
2 REMOVE TIME LIST
3 REMOVE TIME LIST AND
REMOVE ID SEGMENT

IF TERMINATE SELF,
VALUES STORED IN ID SEGMENT

7.16 Communication Review

The following is a summary of the key points of parameter passing and Parent to Child communication.

* Parmeters can be passed from CI to a program by:

  -- RMPAR - up to five values

    * FORTRAN can use RMPAR or the FORTRAN libraries FPARM, RHPAR, and RCPAR.

    * Pascal should use Pascal library Pas.NumericParms since calling RMPAR for interactive communication is not supported.

  -- GETST/EXEC 14 - parameter string/ entire runstring passage

    * FORTRAN can use GETST/EXEC 14 directly or a combination of FPARM, RCPAR, and RHPAR routines.

    * Pascal should use Pas.Parameters. GETST/EXEC 14 can be used directly, although this is not recommended.

* For Parent to Child communication:

  -- A combination of EXEC 14/GETST and RMPAR/PRTN and parameters in the EXEC scheduling calls (i.e., 9,10,23,24) are used from FORTRAN and Pascal.

Of the many methods of sharing large amounts of data, System Common Area will be discussed now.

COMMUNICATION REVIEW

PASSING INFORMATION

Parameter passing – RMPAR
EXEC 14
GETST
PRTN

CLASS I/O

SHARING DATA

Shareable EMA

Files

System COMMON Area

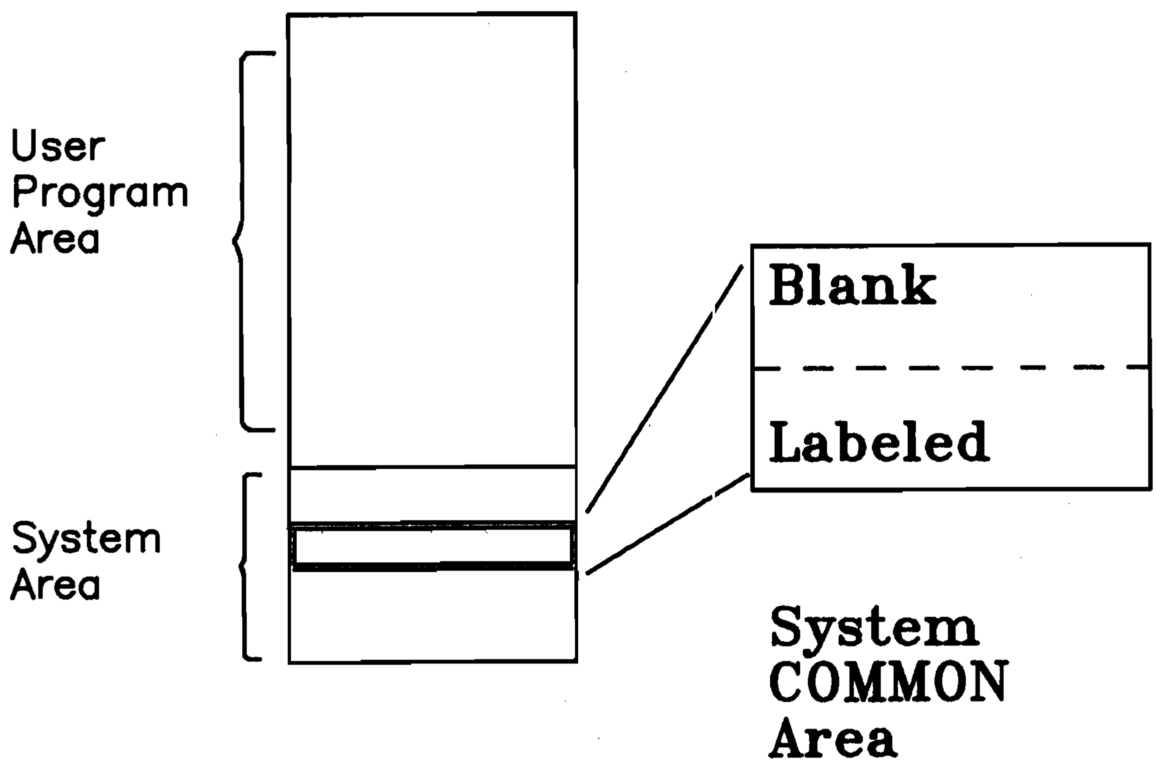
7.17 SYSTEM COMMON AREA

System Common Area is an area in memory used to share data. It can be shared by two or more programs. Blank common can be used by any program by using the SC link option when loading. Labeled Common can only be accessed by specifying the correct entry point to access the data. Labeled common is set up at generation time only, and many subsystems use this area. A drawback in using System Common Area is that whenever the size or content of system common area is changed by the generator, all programs that access the area should be reloaded and checked to see if they need modification. Also, shareable EMA can be larger and there can be more shareable EMA partitions as compared with only one System Common Area. (System Common Area should not be confused with FORTRAN common which is for communication within one program.)

References: System Design and Programmer's Reference Manuals

SYSTEM COMMON AREA

* External to the program
* Always resident in memory
* Set aside at system generation
* Shared by all programs that need it



7.18 Accessing System Common Area

To access System Common Area, link the program with the SC option. Both mains and subroutines can access the System Common Area and so can any other programs that have been loaded to access System Common Area.

For FORTRAN, the blank local common block which is set up for communication within the program's modules now accesses the System Common Area (because of SC loader command).

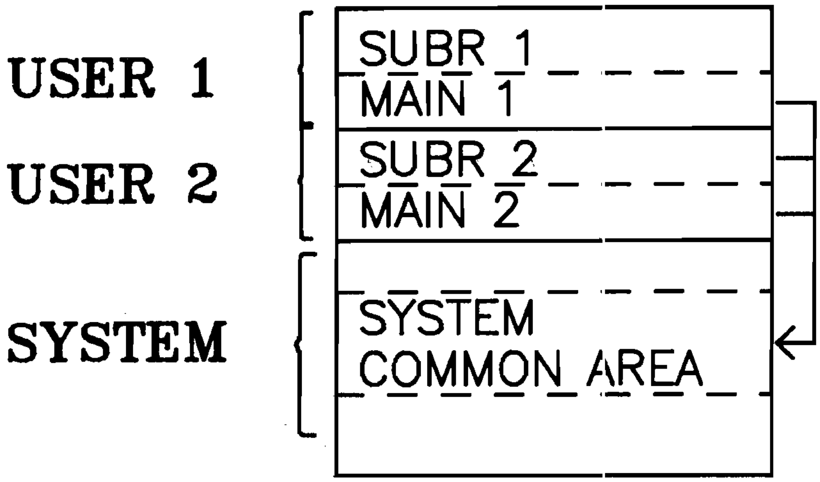
If Pascal is to access the System Common Area, the program must be linked with the SC option, and the programmer must use the Pascal libraries:

Pas.BlankCom1 (&2) & Pas.BlankSize for Blank Common access
and
Pas.LabelCom1 (&2) & Pas.LabelSize for Labeled Common access

References: Link Manual

ACCESSING SYSTEM COMMON AREA

```
CI> link
link Rev.2326    Use ? for help
link:    sc  <------------ System Common Area
link: re lin.rel
  MAIN SUBR
link: en
  LOGLU  $CVT1  $CVT3  .FION  .UFMP  PAU.E  ERO.E
Load Map:
  MAIN   6000  11.
  SUBR   6013   8.
  LOGLU  6023  20. 92077-1X205 REV.2326 <830718.1751>
  $CVT1  6046   7. 92071-1X321 REV.2041 800530
  $CVT3  6056  46. 92071-1X322 REV. 2041 800530
  .FION  6134  24. 24998-1X355 REV. 2326 830406
  .UFMP  6164  15. 24998-1X296 REV. 2326 830406
  PAU.E  6203   1. 24998-1X254 REV. 2001 750701
  ERO.E  6204   1. 24988-1X249 REV. 2001 750701
Main         6000-6204          133. words
Program MAIN.RUN:::6:17 ready; 2 pages
Runnable only on an RTE-A system
```



MEMORY

7.19 System Common Example - Pascal

```
$HEAP 1$

Program SystemCommonOne ( input, output );

type
   int = -32768 .. 32767;
   com = array [1..5] of int;
   comptr = ^com;

var
   i, sizeblank : int;
   systempointer : comptr;

function common_blank $ alias 'Pas.BlankCom1' $
       : comptr; external;

function blank_size $ alias 'Pas.BlankSize' $
       : int; external;

begin

   sizeblank := blank_size;
   if sizeblank <> 0 then begin
      systempointer := common_blank;
      writeln ('Input five INTEGERS!!!');
      for i := 1 to 5 do
         read ( systempointer^[i] );
   end else writeln ('This program has no access to system common');

end.
```

```
      program sycom1
C
C     DECLARE BLANK SYSTEM COMMON AREA
C
      common //inum
C
      write(1,*) 'input an integer value'
      read(1,*) inum
C
C     VALUE NUM IS IN SYSTEM COMMON AREA
C
      end
```

```
      program sycom2
C
C     DECLARE BLANK SYSTEM COMMON AREA
C
      common //iber
C
      write(1,*) 'the number from sycom1 is ', (iber)
C
C     VALUE BER IS PICK UP FROM SYSTEM COMMON AREA
C
      end
```

```
$HEAP 1$

Program SystemCommonTwo ( input, output );

type
   int = -32768 .. 32767;
   com = array [1..5] of int;
   comptr = ^com;

var
   i, sizeblank : int;
   systempointer : comptr;

function common_blank $ alias 'Pas.BlankCom1' $
       : comptr; external;

function blank_size $ alias 'Pas.BlankSize' $
       : int; external;

begin

   sizeblank := blank_size;
   if sizeblank <> 0 then begin
      systempointer := common_blank;
      writeln ('The output from SYS1.PAS is: ');
      for i := 1 to 5 do
         write ( systempointer^[i] );
   end else writeln ('This program has no access to system common');

end.
```

CLASS I/O

CHAPTER 8

Table of Contents

Chapter 8
CLASS I/O

MODULE OBJECTIVES

1. Describe the various methods of program to program communication.

2. Understand the advantages and disadvantages of using CLASS I/O for program-to-program communication.

3. Discuss CLASS I/O operations - two call procedure and Class number usage.

4. Understand the advantages of using Class I/O for program to device communication.

SELF-EVALUATION QUESTIONS

8-1. What are the various methods discussed in this course,of program-to-program communication?

8-2. Where does a Class Write/Read call create the data buffer?

8-3. How does the receiving program retrieve the data after a class Write/Read?

8-4. What are Class Numbers used for?

8-5. What is the difference between the two methods of allocating numbers and which method is preferred?

8-6. What must occur for every CLASS I/O request call, in order to complete the operation?

8-7. What happens when a Class request is made and there are no Class numbers available? No buffers in the complete class queue? Not enough SAM available?

8-8. When is a Class number deallocated by default?

8-9. Why should ownership of Class numbers be assigned?

8-10. How can a CLASS I/O buffer be "consumed" more than once?

8-11. What are the three functions of CLRQ?

8-12. What is rethreading?

8-13. How is the Completed Class Queue ordered?

8-14. Why is CLASS I/O considered a "double call" process? How does the process flow?

8-15. What are the advantages of CLASS I/O over other program-to-program and other program to device communications?

8-16. Must all I/O request to the device use the same Class number?

8.1 C L A S S I / O

CLASS I/O

* PROGRAM TO PROGRAM COMMUNICATION

* I/O FOR DEVICES

8.2 PROG - PROG COMMUNICATION .

Program-to-program communication can be implemented via CLASS I/O. CLASS I/O is itself implemented by a special set of EXEC I/O and system library calls. It provides programs with extra I/O and program communication capabilities such as:

For program-to-program:

Mailbox I/O - allows cooperating programs to communicate via controlled access to a buffer and synchronizes the data transfers.

For I/O to devices:

I/O Without Wait - Allows programs to continue executing concurrently with its own I/O operation to a device.

Some considerations when choosing CLASS I/O are: Is the program swappable? Will the program go into I/O suspend? How much data is to be transferred (buffers)? Is synchronization required? How much SAM is available?

References: Programmer's Reference Manual

PROGRAM TO PROGRAM COMMUNICATION

* RMPAR/PRTN
* EXEC 14/GETST
* SYSTEM COMMON AREA
* SHAREABLE EMA
* FILES
* CLASS I/O

MAIL BOX I/O $\Longrightarrow$ SYNCHRONIZATION

8.3 Mailbox I/O

Mailbox I/O prevents communicating programs from processing incomplete or non-updated data. One program can send multiple data buffers to another even though the other program has not accepted any of them yet. Also, a program can suspend if it asks for data that has not been sent or that is not yet valid. Multiple programs can access the same data buffers if they all know the same 'key' or class number. This is the mailbox address or mail key. CLASS I/O uses SAM for its data buffer.

MAILBOX I/O

* Any number of programs can communicate and share data

* Can send multiple data buffers before accepting any data buffers

* A program requesting a data buffer before one is available, is suspended by RTE until a buffer is available

* A special key controls access to data buffers

* Size of data buffers is limited only by the size of SAM

8.4 Data Transfers Thru SAM

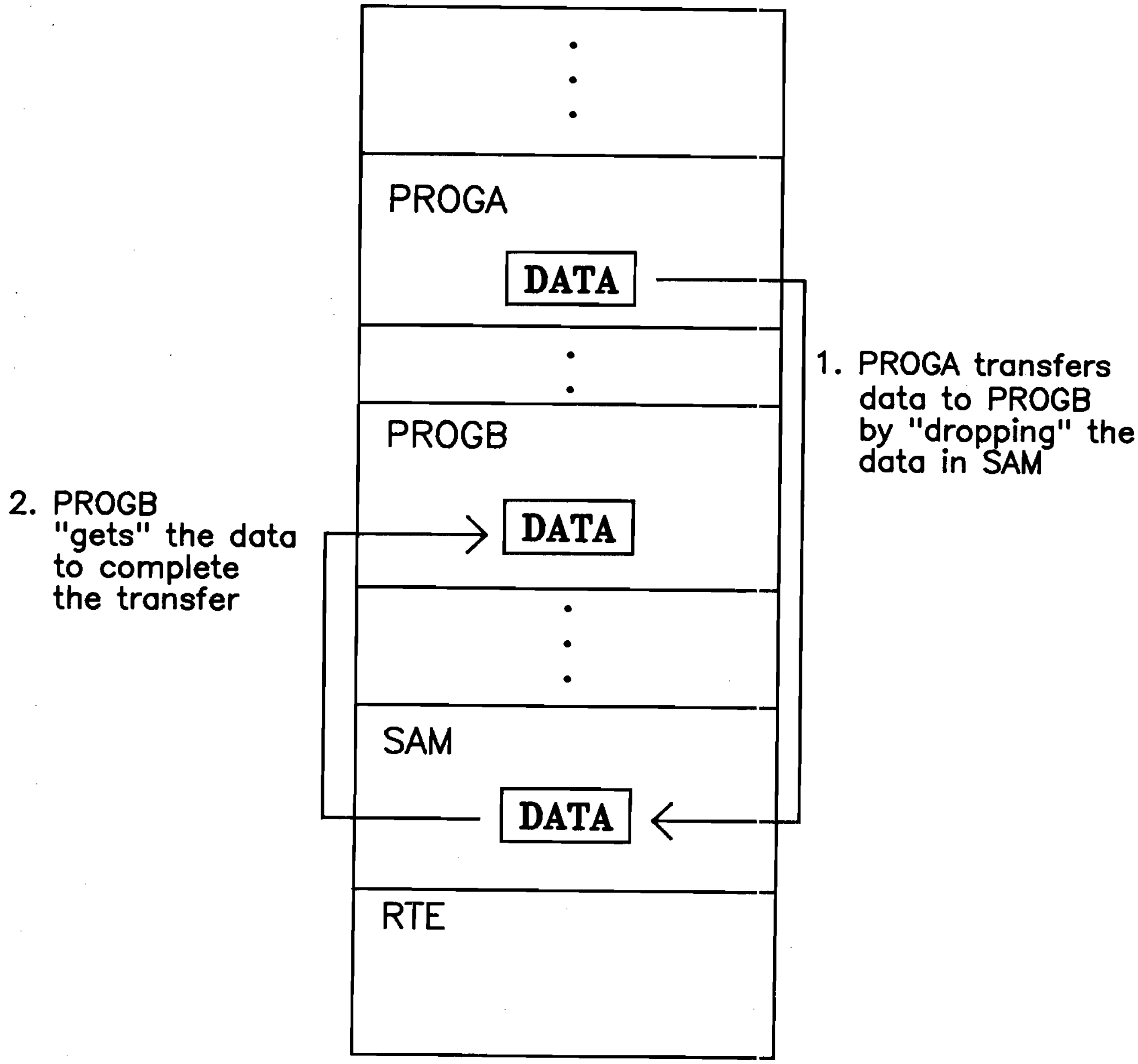
CLASS I/O uses SAM to pass data between programs. By placing the data buffers in SAM, the program can be swapped out if necessary (i.e., the data buffer is not in the program user space).

Multiple data buffers may be placed in SAM and synchronously retrieved by other programs if they have the correct class number, which may be passed via the System Common Area, or the EXEC scheduling call.

Because SAM limits the buffer size and the number of buffers, care must be taken not to use up all of SAM. Class numbers (1-255) which are set up at generation are another resource for Class I/O.

DATA TRANSFERS THRU SAM

PHYSICAL MEMORY



8.5 Manufacturers and Consumers

A program initiates a program-to-program data transfer with an EXEC 20 (LU=0). This call manufactures a buffer in SAM and fills it with data from the calling program, while the other program retrieves the data and consumes the buffer in SAM by calling EXEC 21 - a Class Get. Thus, every CLASS I/O operation is a dual call, an initiation request and a completion request. The programs can execute independently of the data transfers, which are handled by RTE. When generating RTE, the number of class numbers to be used in the system is specified. Class numbers are used to protect the CLASS I/O data buffers in SAM in the following ways. One, when a program uses CLASS I/O for program-to-program communication it must request a class number from RTE and make the Class Write/Read request specify the class number. Two, the program retrieving the data must make a Class Get call specifying the appropriate Class number. (The automatic teller machines have a card and a secret number for communication protection between the customer and the bank account as an example.)

MANUFACTURERS
and
CONSUMERS

INITIATED BY **EXEC 20** (*CLASS WRITE/READ)

- **MANUFACTURES** A BUFFER IN SAM
- FILLS BUFFER WITH DATA FROM PROGRAM

COMPLETED BY **EXEC 21** (*CLASS GET*)

- **CONSUMES** DATA IN SAM BUFFER
- RELEASES SAM BUFFER

⟶ EVERY CLASS I/O TRANSFER IS A

DOUBLE CALL

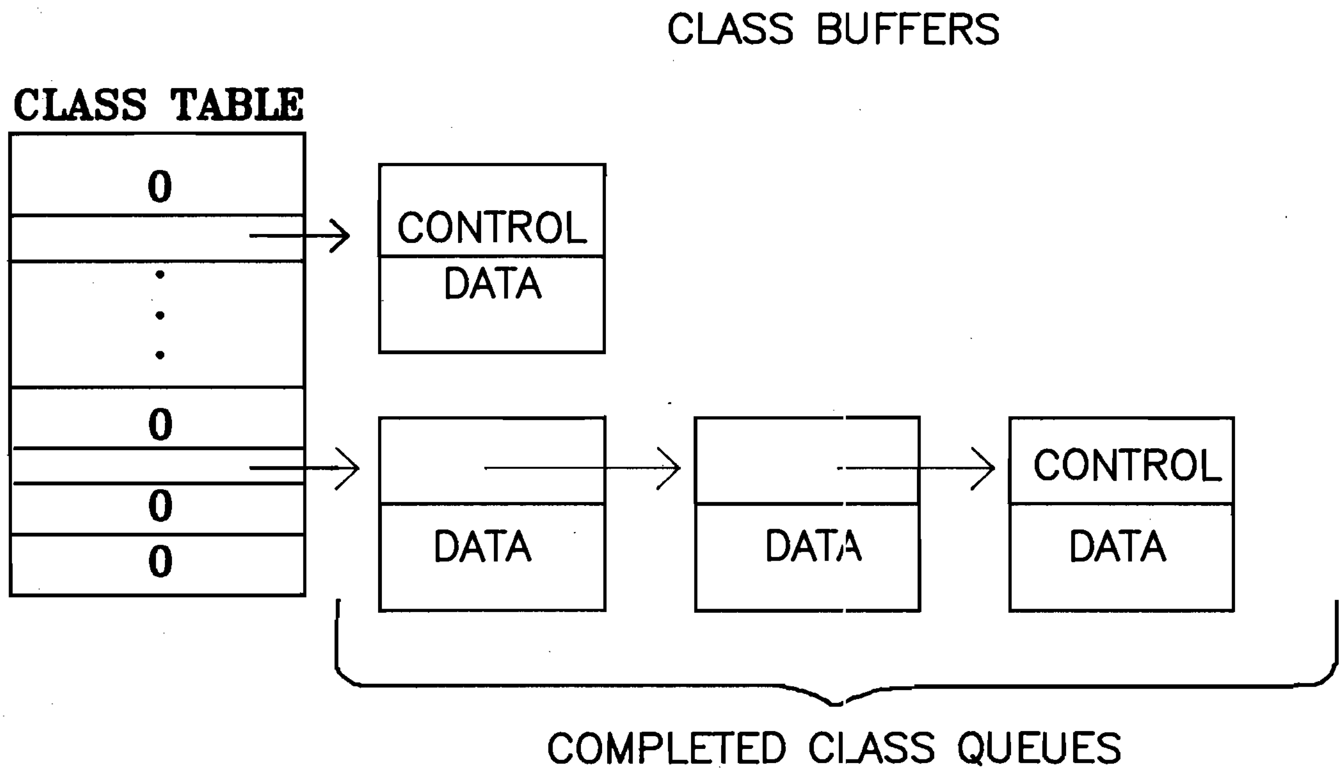
⟶ EVERY CLASS I/O TRANSFER USES

CLASS NUMBERS

8.6 Completed Class Queue

RTE keeps a list in SAM of the data buffers which were manufactured by a Class call and are waiting to be consumed by a Class Get. These lists are called Completed Class Queues and are linked off the appropriate Class number with which they were called (i.e., manufactured). A program must specify the proper class number to access the buffers linked off that Class number. The buffer in SAM is called the class buffer. It has two parts: the control information, which specifies the class call, and the data buffer which contains the data being transferred. Class numbers can be allocated by CLRQ or by the CLASS I/O request call itself. The Completed Class Queue linking is ordered in the FIFO method.

COMPLETED CLASS QUEUE



EACH CLASS BUFFER HAS TWO PARTS –

CONTROL INFORMATION

- CLASS NUMBER
- SIZE OF DATA AREA
- ADDITIONAL INFORMATION

DATA BUFFER

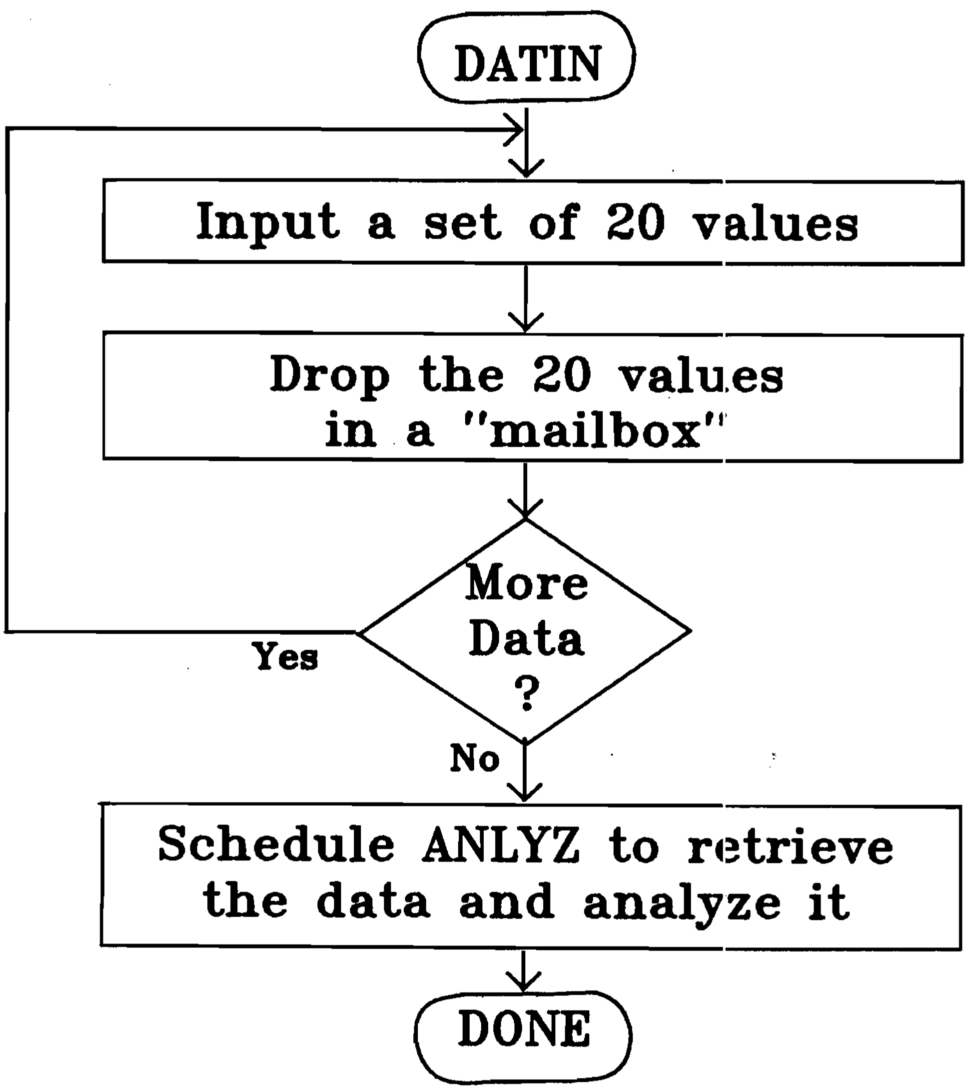
8.7 A Sample Program

No Text

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.

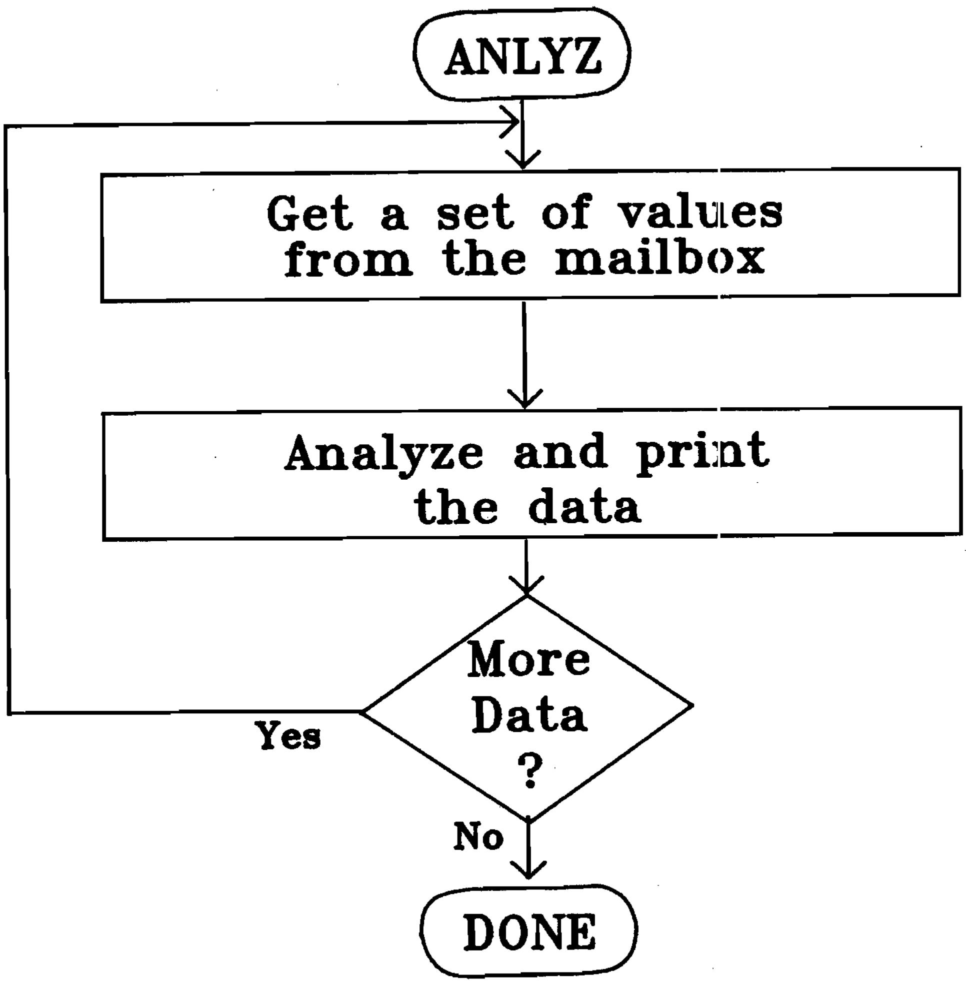
Program DATIN is to be scheduled when the experiment begins –



8.8 ANLYZ

No Text

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.



8.9 Allocating a Class Number

Allocating a class number can be done in two ways. The preferred method is with the CLRQ subroutine. CLRQ allows the Class number to be allocated and assigned an owner. The CLRQ routine is important since if the program aborts or terminates without making a Class Get (i.e., releasing the Class number or SAM buffers used), the Class number could be lost to the system along with the SAM buffer. Potentially, no Class numbers would be available in the system, or all of SAM could be used up and lost. However, if CLRQ is used (i.e., class ownership is known), RTE will automatically deallocate the Class number and return the data buffers to SAM if the owning program aborts or terminates without "cleaning up".

FUNC is the Class Management control function which, when set to 1, will:

1. assign ownership to the program name in PARM1,
2. will <u>not</u> assign ownership if PARM1=0, and
3. will assign class ownership to the calling program if there is no PARM1 parameter given (default).

CLASS now contains the Class number returned by RTE. Now, CLASS can be sent to other programs to tell the program what Class number to use when retrieving data from the Completed Class Queue. The calling program can do additional Class WRITE/READs on the already allocated Class number also.

The alternative method, which does <u>not</u> allow automatic clean up and does not provide for assigning ownership, is with the EXEC 20 (& 17, 18, 19) calls. If the Class parameter has a value of zero, RTE will return a Class number into CLASS. CLASS can then be used as stated above. The only means for deallocating the Class number and buffer is when a Class Get is made and there are no more buffers on the Completed Class Queue and no pending requests for that Class number, unless some option bits are set. The dis-advantages of this method are that all the Class numbers (and SAM) could be used and not deallocated. This could crash the system.

ALLOCATING A CLASS NUMBER

* ALLOCATE AND ASSIGN OWNERSHIP

CLRQ(FUNC,CLASS [,PARM1])

1=class ownership assigned

program name, 0,defaulted

* ALLOCATE ONLY

CLASS=0
EXEC(20,0,BUFR,BUFLN,P1,P2,CLASS)

RTE returns allocated class number

* IF THE PROGRAM ABORTS OR IS TERMINATED WITHOUT EXPLICITLY DEALLOCATING THE CLASS NUMBER

CLRQ – class number released for clean–up

EXEC 20 – class number NOT released for clean–up

8.10 EXEC 20/EXEC 21

EXEC 20 - Write/Read - manufactures a buffer in SAM, fills it with data from the program and links it to the appropriate Completed Class Queue for program-to-program.

The control word (CNTWD) is set to zero for program-to-program communication. The control word contains device driver information and the LU of the device, thus for program-to-program we set the LU to zero.

CLASS Is the Class number.

UV is a user-defined variable retrieved by the Class Get and used in rethreading as the old Class number.

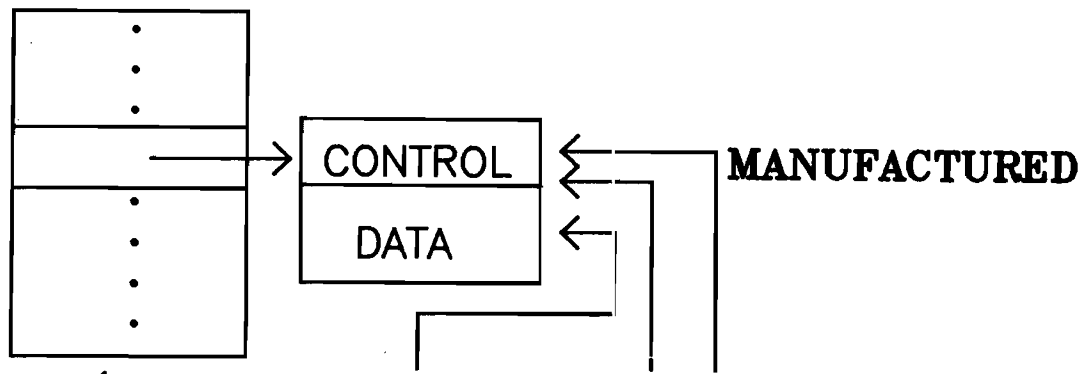
KEY is the key number for a locked LU (See Chapter 12, i.e., two programs can share a locked LU). PARM1 and PARM2 are optional parameters that can be retrieved by the Class Get.

EXEC 21 - Class Get consumes one buffer in the Completed Class Queue off the specified Class number. The completed class queue is ordered by FIFO and thus the EXEC 21 "gets" the first buffer in the queue. It is therefore important that the programmer knows what he/she is getting. CLASS is the previously allocated Class number. RTN1 and RTN2 are the optional parameters which were passed when the buffer was manufactured. RTN3 tells how the buffer was manufactured (i.e., 1=R,W/R 2=W 3=C). The Class Get completes the data transfer. It is the second call of the "double call". This calling sequence synchronizes the data access. The receiving program will not be able to do the Class Get call if there is no buffer in the Completed Class Queue to get.

EXEC 20/EXEC 21

EXEC 20 CLASS WRITE/READ

CLASS TABLE

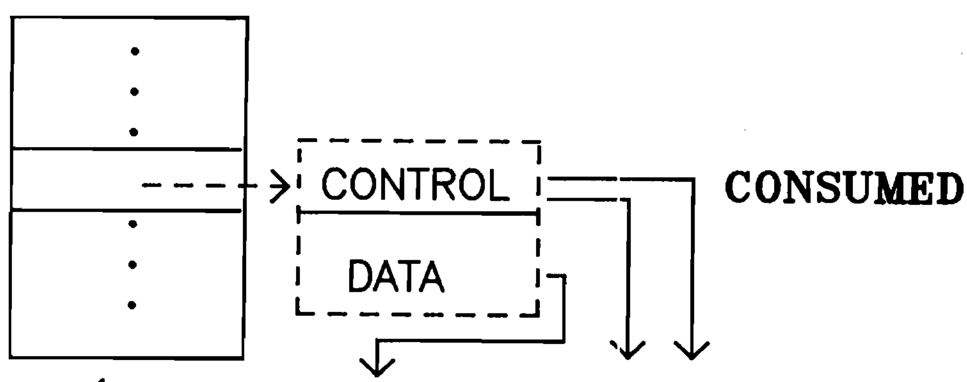


EXEC(20,CNTWD,BUFR,BUFLN,P1,P2,CLASS,UV,KEY)

0 indicates program to program communication

EXEC 21 CLASS GET

CLASS TABLE



EXEC(21,CLASS,BUFR,BUFLN,R1,R2,R3,UV)

HOW BUFFER WAS MANUFACTURED

8.11 Clean-Up

CLRQ has functions other than allocating Class numbers. If FUNC=2 then the class requests on that Class number will be flushed and the Class number deallocated. If FUNC=3, then all class requests to a particular device (LU) will be flushed, but the Class number is <u>not</u> deallocated.

Clean up can also occur by making a Class Get call if certain specifications are met. If there are no pending requests (for device I/O) and no buffers in the Completed Class Queue, the Class number will be deallocated with the Class Get call. (Remember, if CLRQ was used to allocate the Class number then, if the program fails to do a Class Get or aborts, the Class number will be deallocated and all buffers cleaned up.) If CLASS parameter has bit 13 set in the Class Get call then the Class number will <u>not</u> be returned to the system when the last buffer is consumed.

CLEAN-UP

CLRQ (1,CLASS) ⟶ ALLOCATE AND ASSIGN OWNERSHIP

CLRQ (2,CLASS) ⟶ flush class requests, deallocate class number

CLRQ (3,CLASS,LU) ⟶ flush class requests on specified LU number

Class GET ⟶ deallocates class number, if last buffer

⟶ no deallocation if set bit 13 of CLASS

8.12 Example - The Manufacturer

The next four slides show the solution to the example problem. Both solutions use CLRQ and assign ownership to either the calling program or the Child program. Recall EXEC 10 and RMPAR.

```
program datin (input,output);
type int = -32768..32767;
     protype = packed array [1..6] of char;
     valuetype = packed array [1..40] of char;
var  a,b,i,func,class,len,error: int;
     progname: protype;
     numbers: valuetype;
procedure readex $ alias 'exec' $
     ( ecode, cntwd : int; var numbers : valuetype; length : int );
     external;
procedure clrqone $ alias 'clrq' $
     ( func, class : int; progname : protype );
     external;
procedure clrq $ alias 'clrq' $
     ( func, class : int );
     external;
procedure schedule $ alias 'exec' $
     ( ecode : int; progname : protype; class : int );
     external;
procedure abreg $ alias 'abreg' $
     ( var a, b : int );
     external;
procedure classio $alias 'exec'$
     (ecode,lu: int; numbers: valuetype; len,pl,p2: int;
      class: int);  external;
begin
 { ASSIGN CLASS NUMBER OWNERSHIP TO DATIN }
  class := 0;
  func := 1;
  clrq(func,class);
 { INPUT THE 10 SETS OF 20 VALUES }
  writeln ('Input two sets of twenty values: ');
  for i := 1 to 10 do begin
     readex ( 1, 257, numbers, -40 );
     abreg ( a, b );
 { DROP IT IN THE MAILBOX }
     classio ( 20, 0, numbers, b, 0, 0, class );
  end;
 { SCHEDULE ANALYSIS PROGRAM AND ASSIGN IT CLASS OWNERSHIP }
  progname := 'ANLYZ ';
  clrqone ( func, class, progname );
  schedule ( 10, progname, class );
end.
```

EXAMPLE OF PROGRAM TO PROGRAM COMMUNICATION

THE MANUFACTURER . . .

```
      PROGRAM DATIN
C
      INTEGER DATA(20), SPROG(3), EC
      DATA SPROG/6HANLYZ /
      DATA EC/400B/

C  Assign class number ownership to DATIN
C
      IFUNC=1
      ICLAS=0
      CALL CLRQ (IFUNC,ICLAS)
C
C  Input the 10 sets of 20 values.
C
      DO 20 I = 1,10
C
           CALL EXEC (1,1+EC,DATA,20)
           CALL ABREG (IA,IB)
C
C          Drop in the mailbox.
C
           CALL EXEC (20,0,DATA,IB,K,L,ICLAS)
20    CONTINUE
C
C  After the data is in the mailbox, schedule the
C  analysis program assigning it the class ownership.
C
      CALL CLRQ (IFUNC,ICLAS,SPROG)
      CALL EXEC (10,SPROG,ICLAS)
C
      END
```

8.13 The Consumer

Pascal example:

```
$run_string 0$
Program Anlyz;

type
    int = -32768 .. 32767;
    valuetype = packed array [1..40] of char;
    ptype = array [1..5] of int;
var
    class, a, b, i, j : int;
    buffer : valuetype;
    parms : ptype;
    out : text;

procedure rmpar $ alias 'rmpar' $
        ( var parm : ptype );
        external;
procedure class_get $ alias 'exec' $
        ( ecode, class : int; buffer : valuetype; length : int );
        external;
procedure abreg $ alias 'abreg' $
        ( var a, b : int ); external;

begin

  { RETRIEVE THE CLASS NUMBER }

    rmpar ( parms );
    class := parms[1];
    rewrite ( out, '1');

  { GET EACH SET OF VALUES AND ANALYZE }

    for i := 1 to 10 do begin
       class_get ( 21, class, buffer, -40 );
       abreg ( a, b );
       writeln (out, 'The values passed by DATIN are: ');
       for j := 1 to b do
          write ( out, buffer[j] );
       writeln(out);
    end;

end.
```

THE CONSUMER . . .

```
      PROGRAM ANLYZ
C
      INTEGER DATA(20),PARM(5)
C
C     Retrieve the class number.
C
      CALL RMPAR (PARM)
      ICLAS = PARM(1)
C
C     Get each set of values and analyze.
C
      DO 20 I = 1,10
C
         CALL EXEC (21,ICLAS,DATA,20)
         CALL ABREG (IA,IB)

         WRITE(1,'(20A2)') (DATA(J),J=1,IB)
C
20    CONTINUE
C
      END
```

8.14 Another Way to Program Our Example

```
program datin (input,output);

type int = -32768..32767;
     protype = packed array [1..6] of char;
     valuetype = packed array [1..40] of char;
var  a,b,i,func,class,len,error: int;
     progname: protype;
     numbers: valuetype;
procedure readex $ alias 'exec' $
     ( ecode, cntwd : int; var numbers : valuetype; length : int );
     external;
procedure clrqone $ alias 'clrq' $
     ( func, class : int; progname : protype );
     external;
procedure schedule $ alias 'exec' $
     ( ecode : int; progname : protype; class : int );
     external;
procedure abreg $ alias 'abreg' $
     ( var a, b : int );
     external;
procedure classio $alias 'exec'$
     (ecode,lu: int; numbers: valuetype; len,p1,p2: int;
      class: int);  external;
begin

  { ASSIGN CLASS NUMBER OWNERSHIP TO ANLYZ }
   class := 0;
   func := 1;
   progname := 'ANLYZ ';
   clrqone ( func, class, progname );
  { INPUT THE 10 SETS OF 20 VALUES }
   writeln ('Input ten sets of twenty values: ');
   readex ( 1, 257, numbers, -40 );
   abreg ( a, b );

  { DROP IT IN THE MAILBOX }
   classio ( 20, 0, numbers, b, 0, 0, class );
  { SCHEDULE THE ANALYSIS PROGRAM }
   schedule ( 10, progname, class );

  { NOW INPUT AND SEND THE REMAINING SETS OF VALUES }
   for i := 1 to 9 do begin
      readex ( 1, 257, numbers, -40 );
      abreg ( a, b );
      classio ( 20, 0, numbers, b, 0, 0, class );
   end;
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?

```
      PROGRAM DATIN
C
      INTEGER DATA(20), SPROG(3),EC
      DATA SPROG/6HANLYZ /
      DATA EC/400B/
C
C     assign class ownership to anlyz
C
      IFUNC=1
      ICLAS=0
      CALL CLRQ (IFUNC,ICLAS,SPROG)
C
C     Input the first set of 10 values, drop them in
C     the mailbox, and schedule the analysis program.
C
      CALL EXEC (1,1+EC,DATA,20)
      CALL ABREG (IA,IB)
C
      CALL EXEC (20,0,DATA,IB,K,L,ICLAS)
C
      CALL EXEC (10,SPROG,ICLAS)
C
C     Now input and send the remaining sets of values
C
      DO 20 I=1,9
C
C         Input a set of values.
C
          CALL EXEC (1,1+EC,DATA,20)
          CALL ABREG (IA,IB)
C
C         Drop in the mailbox.
C
          CALL EXEC (20,0,DATA,IB,K,L,ICLAS)
C
20    CONTINUE
C
      END
```

8.15 And The Consumer Version 2

Pascal Example:

```
$run_string 0$
Program Anlyz;

type
   int = -32768 .. 32767;
   valuetype = packed array [1..40] of char;
   ptype = array [1..5] of int;
var
   class, a, b, i, j : int;
   buffer : valuetype;
   parms : ptype;
   out : text;
procedure rmpar $ alias 'rmpar' $
       ( var parm : ptype );
       external;
procedure class_get $ alias 'exec' $
       ( ecode, class : int; buffer : valuetype; length : int );
       external;
procedure abreg $ alias 'abreg' $
       ( var a, b : int ); external;
begin

 { RETRIEVE THE CLASS NUMBER }
   rmpar ( parms );
   class := parms[1] + 8192;
   rewrite ( out, '1');

 { GET ONE SET OF NUMBERS, BUT DON'T }
 { DEALLOCATE THE CLASS NUMBERS      }

   for i := 1 to 10 do begin
      class_get ( 21, class, buffer, -40 );
      abreg ( a, b );
      writeln (out, 'The values passed by DATIN are: ');
      for j := 1 to b do
         write ( out, buffer[j] );
      writeln(out);
   end;

 { NOW THAT ALL THE DATA SETS HAVE BEEN ANALYZED,        }
 { USE AN EXTRA GET CALL TO DEALLOCATE THE CLASS NUMBER }
   class := parms[1];
   class_get ( 21, class, buffer, -40 );
end.
```

... AND THE CONSUMER

VERSION 2

```
      PROGRAM ANLYZ
C
      INTEGER DATA(20),PARM(5),SC
      DATA SC/20000B/
C
C     Retrieve the class number.
C
      CALL RMPAR (PARM)
      ICLAS = PARM(1)
C
C     Get each set of values and analyze.
C
      DO 30 I = 1,10
C
C           Get one set of values, but don't
C           deallocate the class numbers.
C
            CALL EXEC (21,ICLAS + SC,DATA,20)
            CALL ABREG (IA,IB)
C
C           Analyze the set of values.
C
            WRITE (1,'(20A2)') (DATA(J),J=IB)
C
30    CONTINUE
C
C     Now that all the data sets have been analyzed,
C     use an extra GET call to deallocate the class number.
C
      CALL EXEC (21,ICLAS,DATA,20)
C
      END
```

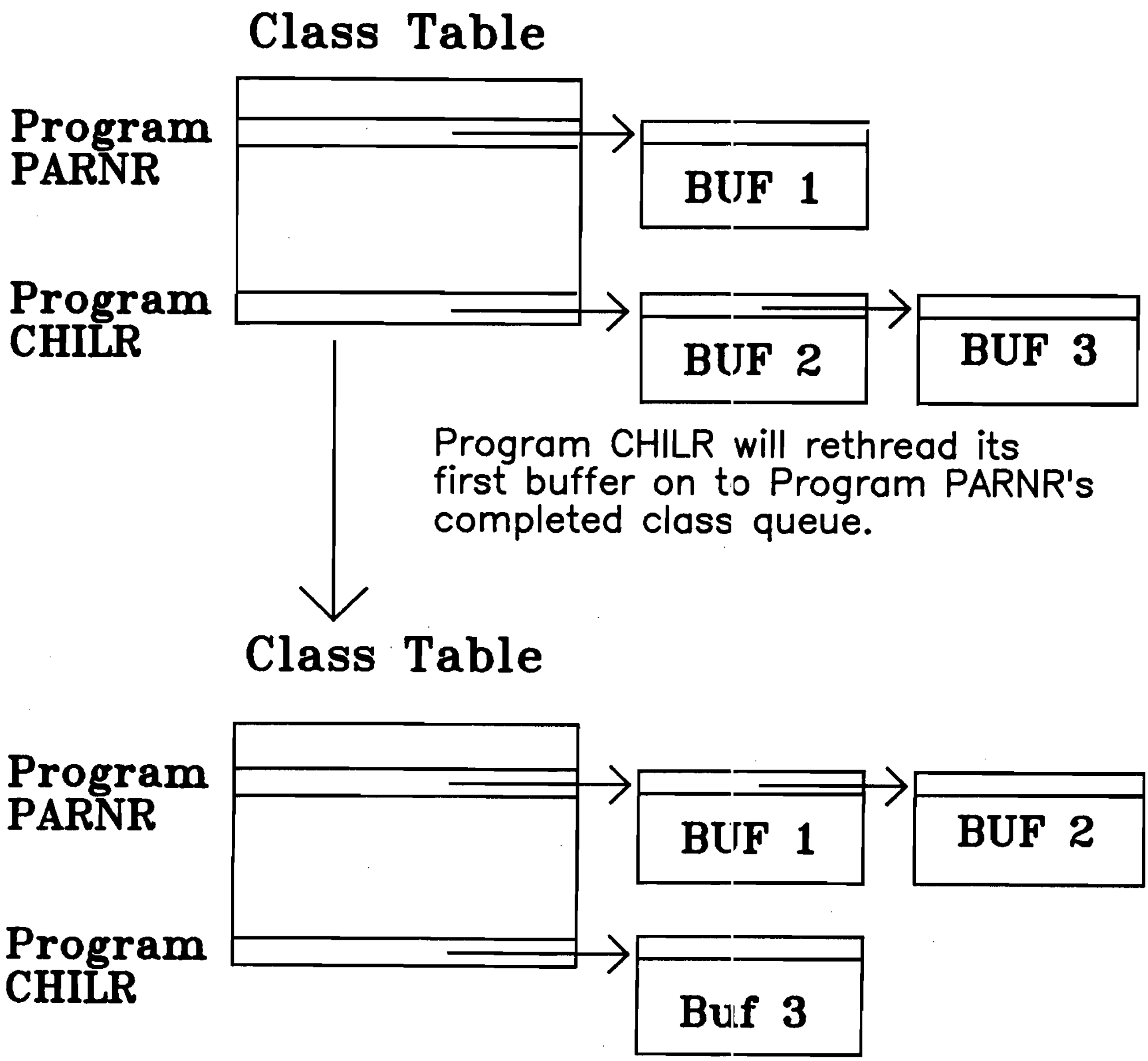
8.16 Class Buffer Rethreading

```
      PROGRAM PARNR
      INTEGER CLASS,IBUF(10),NUM
C** Belongs to CHILR.
      CLASS=0
      CALL CLRQ(1,CLASS,6HCHILR )
      WRITE(1,'("ENTER NUMBER OF SETS TO BE ENTERED_")')
      READ(1,*)NUM
      WRITE(1,*)'ENTER 10 VALUES'
      READ(1,*)(IBUF(I)I=1,10)
      CALL EXEC(20,0,IBUF,10,ID,IE,CLASS)
   50 CONTINUE
      CALL EXEC(9,6HCHILR ,CLASS,NUM)
      END


      PROGRAM CHILR
      INTEGER IPARM(5),CLASS,BUF(10),NUM,CLAS2,BUF2(10)
      DATA BUF2/5,5,5,5,5,5,5,5,5,5/
      CALL RMPAR(IPARM)
      CLASS=IPARM(1)
      NUM=IPARM(2)+1
C** Have the system allocate another class number and link BUF2 on
C** this class number.
      CLAS2=0
      CALL EXEC(20,0,BUF2,10,0,0,CLAS2)
C** Rethread buffer(BUF2) on CLAS2 to class # set up by PARNR(CLASS)
      CALL EXEC(20,0,BUF2,0,0,0,CLASS+20000B,CLAS2)
      DO 50 L=1,NUM
      CALL EXEC(21,CLASS,BUF,10)
      WRITE(1,'("VALUES PASSED ",10(I5,2X))')(BUF(I),I=1,10)
   50 CONTINUE
      END
```

Rethreading can save considerable overhead where request retransmission or broadcasting a request is desired within a program. It is a way to move class buffers without having to allocate more memory or more words. Possible uses include reusing buffers passed via program-to-program communication, recycling through buffers, and broadcasting class-buffered messages to multiple LUs. The UV optional parameter will be set to OCLAS which is the old class number identifying the Completed Class Queue where the rethread buffer will be removed. CLASS is the Class number with the RT bit (bit 13) set, indicating rethreading is desired.

CLASS BUFFER RETHREADING

Rethreading means that the next buffer in a completed class queue is relinked to point to a new class number.

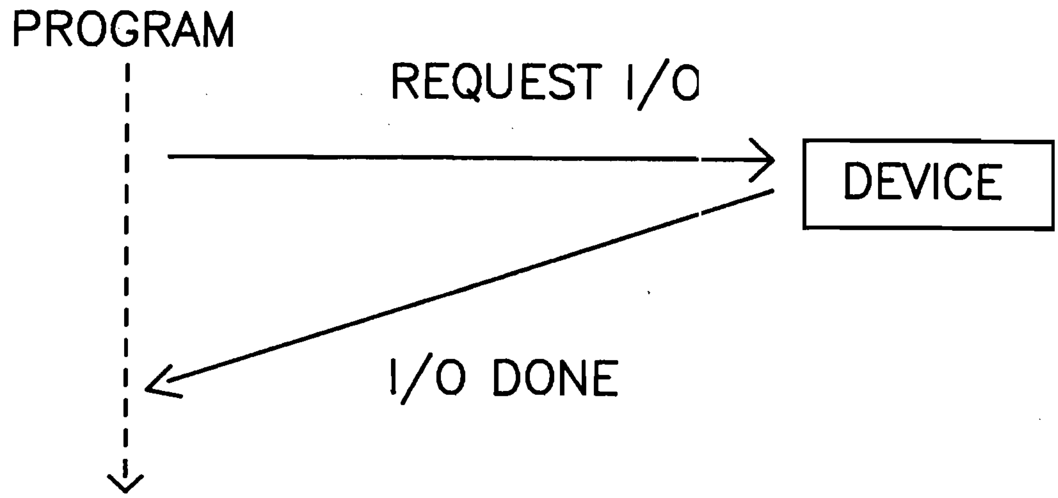


8.17 CLASS I/O FOR DEVICE I/O

CLASS I/O also can be used for device I/O and control. I/O without wait means the program can continue executing concurrently with its own input/output operations. This applies to I/O control to device as well. When a CLASS I/O request is made to a device, it is associated with a specified Class number as before, except this time it is queued off the I/O device table, not the Completed Class Queue.

This list is the pending Class request. The request remains pending until the driver has received and processed it accordingly. When the driver finishes the operation, the request is then linked off the Completed Class Queue (as with program-to-program communication) associated with the Class number. Then the second call of the double call process, the Class Get, is performed to complete the operation. This technique allows more than one buffer to be associated with the same Class number and more than one Class request (of different Class numbers) to be linked off the I/O device. If the device driver is busy, the class request is linked off according to program priority. CLASS I/O for device I/O simulates "buffered I/O devices". The process uses two linked lists, the device I/O request list (by device), and the Completed Class Queue (by Class number).

CLASS I/O FOR DEVICE I/O AND CONTROL



EXEC 17 CLASS READ

EXEC 18 CLASS WRITE

EXEC 19 CLASS CONTROL

EXEC 20 CLASS WRITE/READ

EXEC 21 CLASS GET

8.18 CLASS I/O for Input - EXEC 17

Class Read, Write, and Write/Read requests all have the same call format and all manufacture one buffer in SAM. The Class Read request is executed in the following steps:

* A SAM buffer is created with control words and a buffer.
* The device "reads into" the SAM buffer which is linked off the I/O device table.
* The program can continue execution during this I/O operation.
* After the device has read in the data, it is linked in the Completed Class Queue off the Class number which was used in initiating the request (i.e., in the EXEC 17 call).
* The program can now do a Class Get to consume the buffer.
* The data goes in the program's buffer and the SAM buffer is returned to SAM and Class number deallocated if the previously mentioned criteria exist. With this method, a read to an unbuffered, or buffered device can be made without the program I/O suspending.

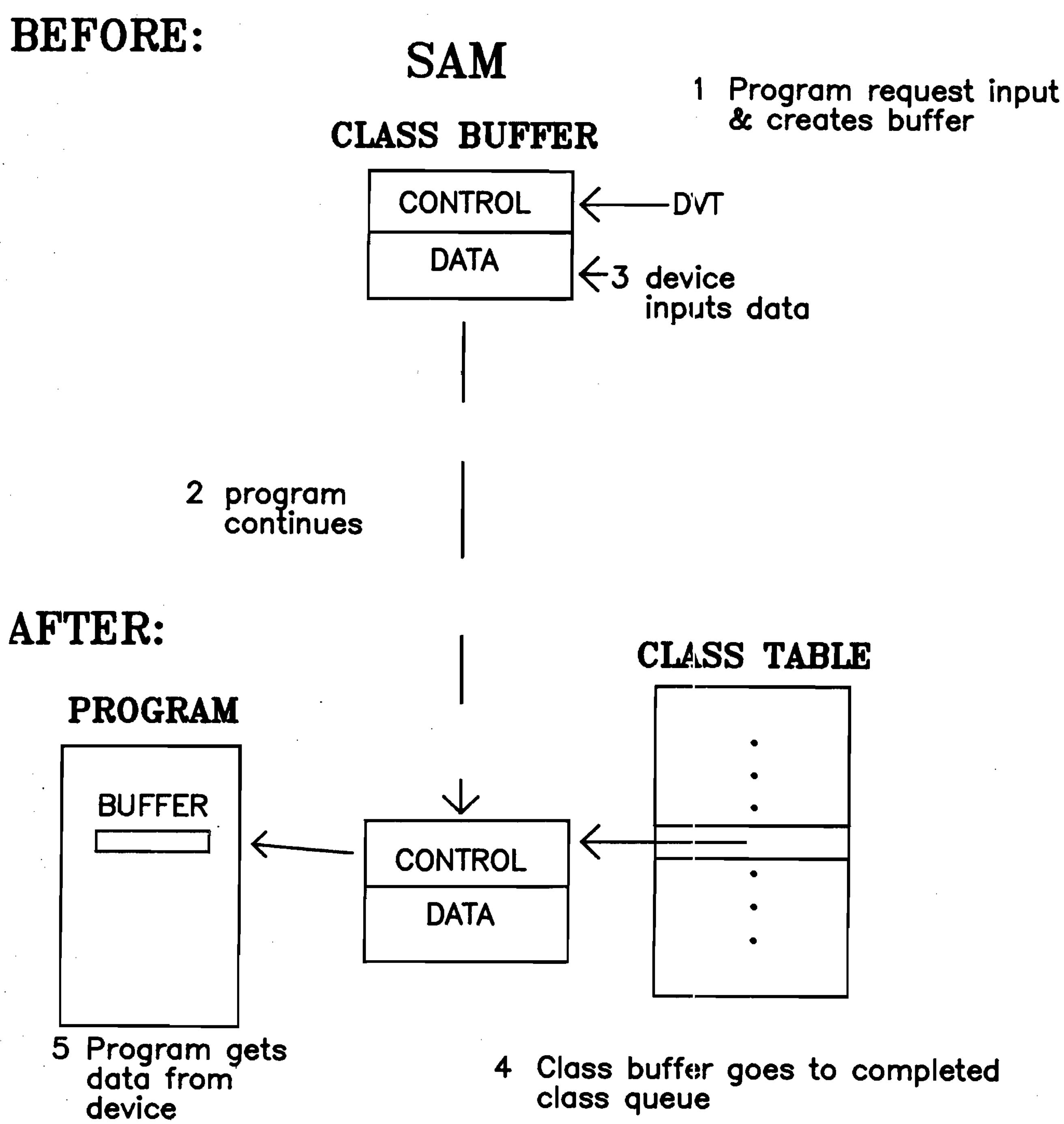
The program can continue execution and can be swapped.

What happens if there are no buffers in the Completed Class Queue when the program does the Class Get?

For example, 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
C     REQUEST INPUT OF DATA - CLASS READ
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
      CALL EXEC (21,...)
```

CLASS I/O FOR INPUT



8.19 CLASS I/O for Output - EXEC 18

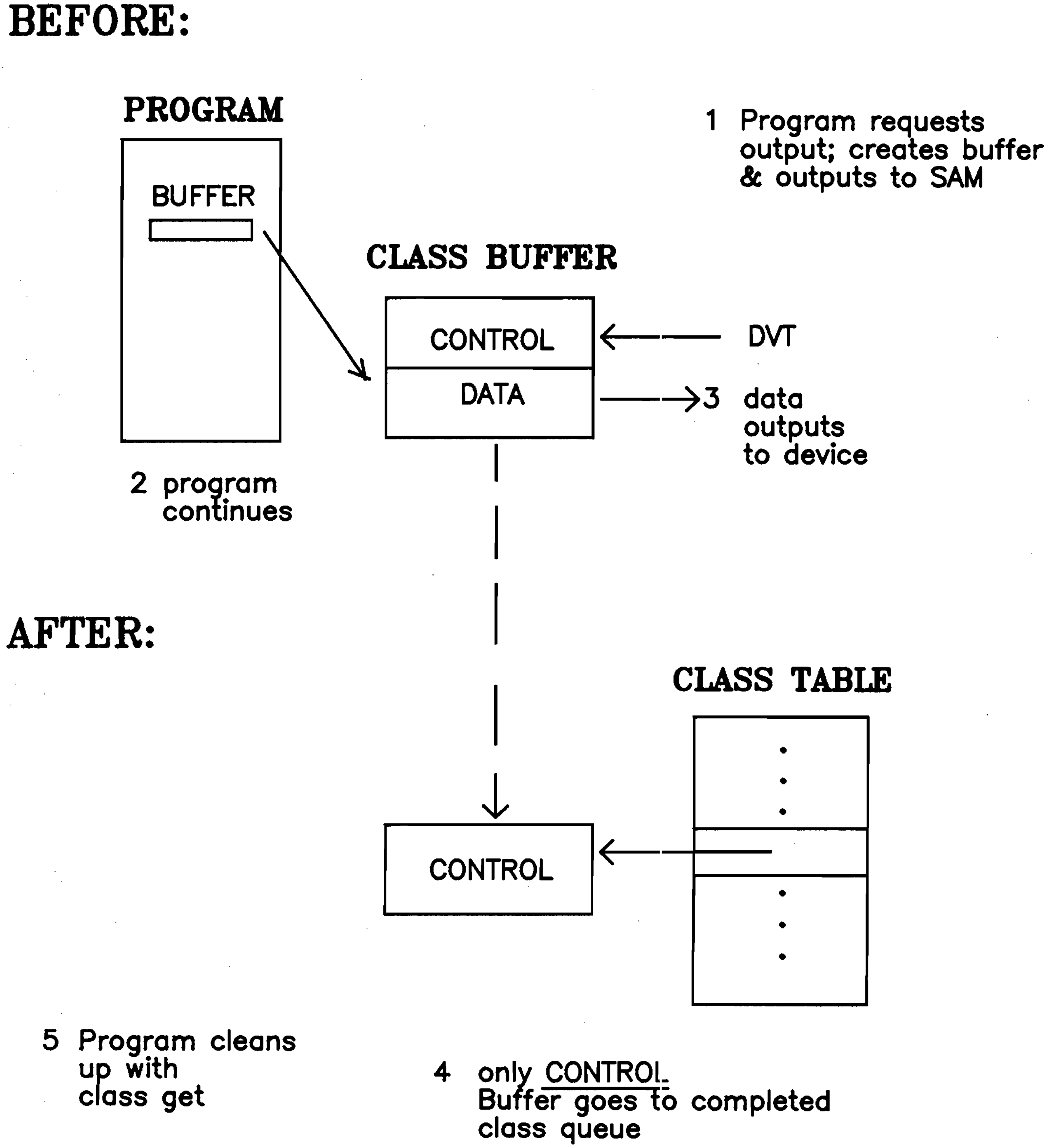
The CLASS I/O for output operation flows as follows:

* The program writes data out to a device by an EXEC 18 call.
* The program buffer is copied to the SAM buffer and the SAM buffer is written out to the device.
* Again, the program can continue without waiting and when the I/O is completed the buffer space will be released to SAM and only the CNTL words will be linked off the Completed Class Queue.
* When the program performs a Class Get, the control words will be released and possibly the Class number.

For example, 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
C     OUTPUT DATA - CLASS WRITE
C
      CALL EXEC (18,...)
C
C     CONTINUE EXECUTION WHILE RTE DOES THE I/O
C

         :
C
C     COMPLETE THE OPERATION - CLASS GET
C
      CALL EXEC (21,...)
         :
```

CLASS I/O FOR OUTPUT



8.20 CLASS I/O for Write/Read - EXEC 20

The CLASS I/O for Write/Read operation for LU <> 0 (i.e., for I/O to devices) is as follows:

* The program initially writes data out to a device by the EXEC 20 call (LU = device Logical Unit number).

* The program buffer is placed in the SAM buffer.

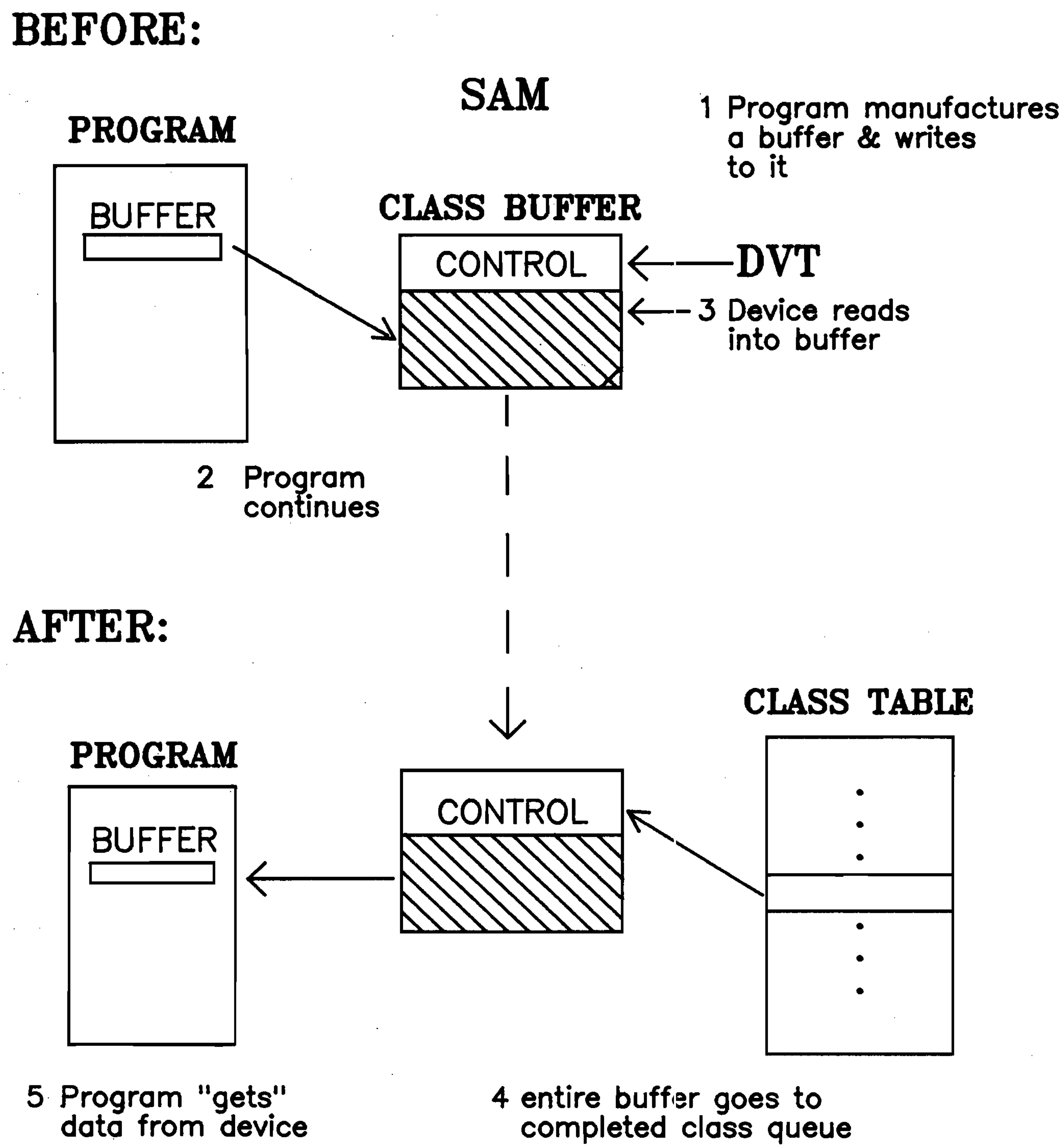
* The device "sees" the request as a read and inputs data into the SAM buffer, overlaying the program's buffer contents already in the SAM buffer.

* Again, the program can continue without waiting and when the device completes input, the buffer is linked off the completed class queue.

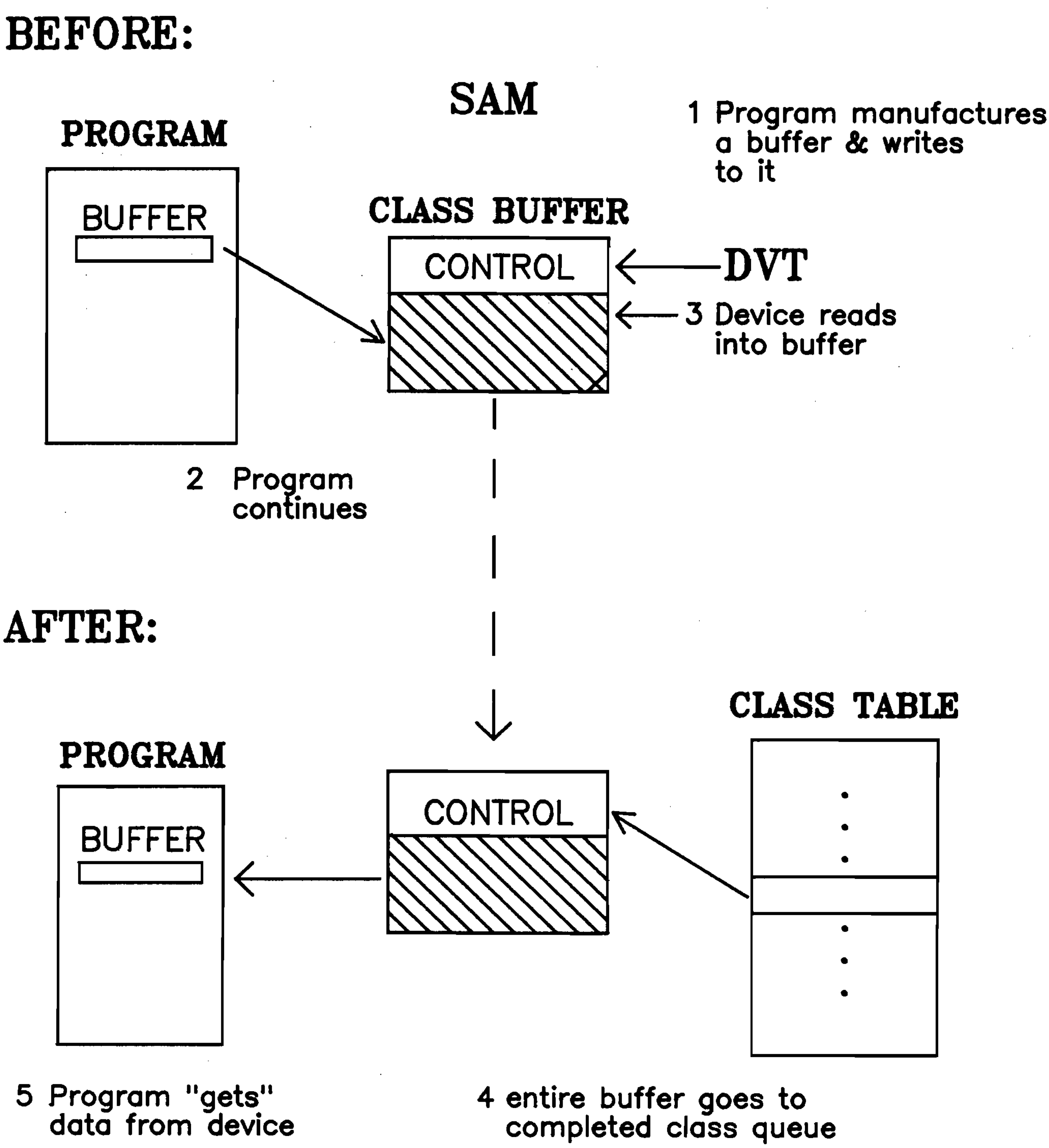
* When the program performs a Class Get, the data is copied into the program's buffer and the SAM buffer is returned to SAM and possibly the Class number is deallocated.

Thus, an EXEC 20 call with LU <> 0 is a means for the program to receive input from a device after the program has first written into or initialized the same buffer.

CLASS I/O FOR OUTPUT/INPUT



CLASS I/O FOR OUTPUT/INPUT



CLASS GET REVISITED

What order are buffers retrieved in?

EXEC (18,.....) CLASS WRITE

.

.

.

EXEC (19,.....) CLASS CONTROL

.

.

.

EXEC (21,.....) CLASS GET

.

.

.

(EXEC 21,.....) CLASS GET

.

.

.

SAM buffers manufactured by CLASS WRITE/READ ⟶ retrieved in order <u>CREATED</u>

SAM buffers manufactured by CLASS READ, CLASS WRITE, CLASS CONTROL ⟶ retrieved in order <u>COMPLETED</u>

8.22 Variations with CLASS I/O

The default use of CLASS I/O can be changed with option bits in the CLASS word. Default usage is:

* class variable contains only a class number (bits 12-0); the class number is deallocated when the last buffer is consumed; the buffer in SAM is not recoverable after it is consumed; and the program will suspend waiting for resources to become available (i.e., Class numbers, SAM, Complete Class buffers, ...).

CLRQ has two option bits in the FUNC parameter. CLRQs CLASS parameter is the same as in EXEC 17,18,19,20 requests.

* Bit 15-(NW) is the no-wait bit. If it is set, the program will not suspend if no class numbers are available when CLRQ request is made.

* Bit 14-(NA) is the no-abort bit. The program will not abort if an error occurs in the CLRQ request. Registers A and B will contain ASCII error message.

The CLASS parameter in EXEC 17,18,19,20 calls has 3 option bits.

* Bit 15-(NW) is the no-wait bit. The program is not suspended if SAM or a class number is not available.

* Bit 14-(SB) is the Save Class Buffer Bit. When it is set, the data buffer (allocated by a Class Write) is saved for future processing. * Bit 13-(RT) is the rethread bit. When set, the request becomes a rethread request rather than a standard call. The UV parameter needs to be set also.

For Class Gets, three option bits are available:

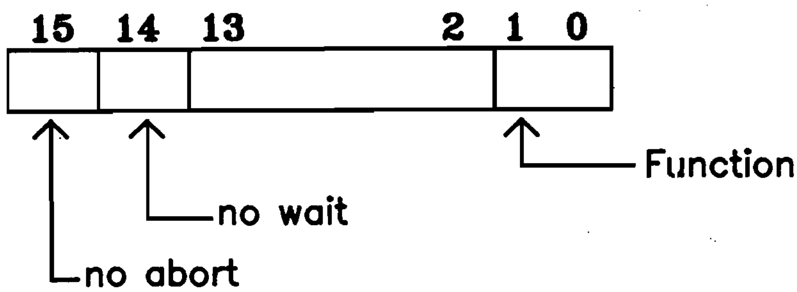
* Bit 15-(NW) is another no-wait bit. The calling program is not suspended if the completed class queue is empty, i.e., there is no SAM buffer to get.

* Bit 14-(SB) is the save class buffer bit. When set, the SAM buffer is saved (i.e., not deallocated by the Class Get) at the head of the list. Thus, on the next Class Get (with same Class number) the same buffer (same data) is consumed.

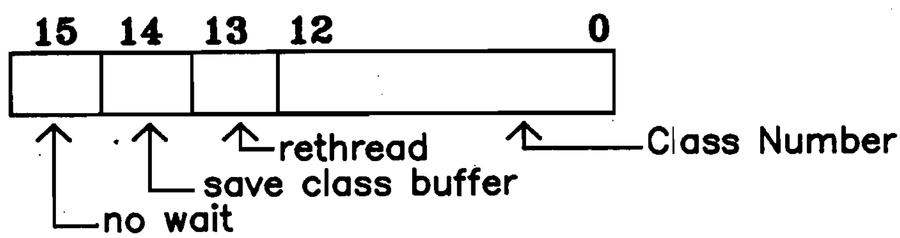
* Bit 13-(SC) is the Save Class Bit or no-deallocate bit. If it is set, the class number is not deallocated when there are no pending class requests and no completed buffers on that class number.

VARIATIONS WITH CLASS I/O

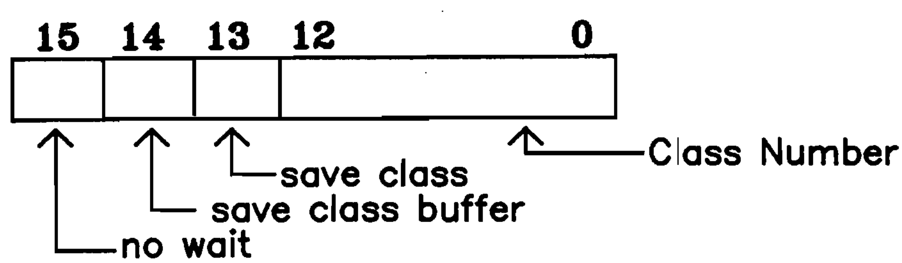
CLRQ – FUNC parameter option bits



CLASS READ, WRITE, CONTROL, WRITE/READ – CLASS parameter option bits



CLASS GET – CLASS parameter option bits



8.23 What Are We Waiting For?

The program will not suspend for the given situations if bit 15 is set. The program must then look at the A register and decide what action to take next.

NOTE: A program will suspend on a Class Get if there is no buffer to get, i.e., no call to a Class Read, Write, Write/Read, or Control has been made previously.

This feature allows CLASS I/O to be used for "synchronization" since the program waits for the buffer to be available. When it is available, RTE "wakes up" the suspended program. The programmer may wish to avoid "being put to sleep" by using bit 15. Beware that the program must then return later to get the buffer, i.e., do its own synchronization.

WHAT ARE WE WAITING FOR?

- **Program will be suspended for –**
  - * NO AVAILABLE CLASS NUMBERS
  - * NOT ENOUGH SAM
  - * EMPTY COMPLETED CLASS QUEUE

- **A "no wait" CLASS request is told what is unavailable**

"no wait" set in	**A–REGISTER**
CLRQ	–1 if no class number available
CLASS R W C W/R	–1 if no class number available –2 if not enough SAM currently
CLASS GET	–n if empty completed class queue n = number of pending requests +1, for that class number

8.24 CLASS I/O - A Summary of Features

Some points to remember about Class I/O:

* Can be used for program-to-program communication with EXEC 20(W/R) with LU = 0 and EXEC 21(GET).
* Allows for synchronization of data transfer for program-to-program communication.
* Allows program to be swappable, since the I/O buffer is in SAM.
* The program does not wait for the I/O transfer, i.e., no I/O suspend.
* CLASS I/O is a "double call". Initiate request with Class W/R,W,R,C and complete with Class Get.
* CLRQ allows "clean-up". Returns SAM to system and deallocates Class number.
* Option bits may override default conditions.

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"
* buffers are queued on class numbers, the "keys" to accessing data
* buffers may be manufactured and consumed asynchronously

8.25 CLASS I/O Vs. Other I/O

No Text

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
* 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				
CLASS I/O				

8.26 TERMINAL HANDLERS EXAMPLE

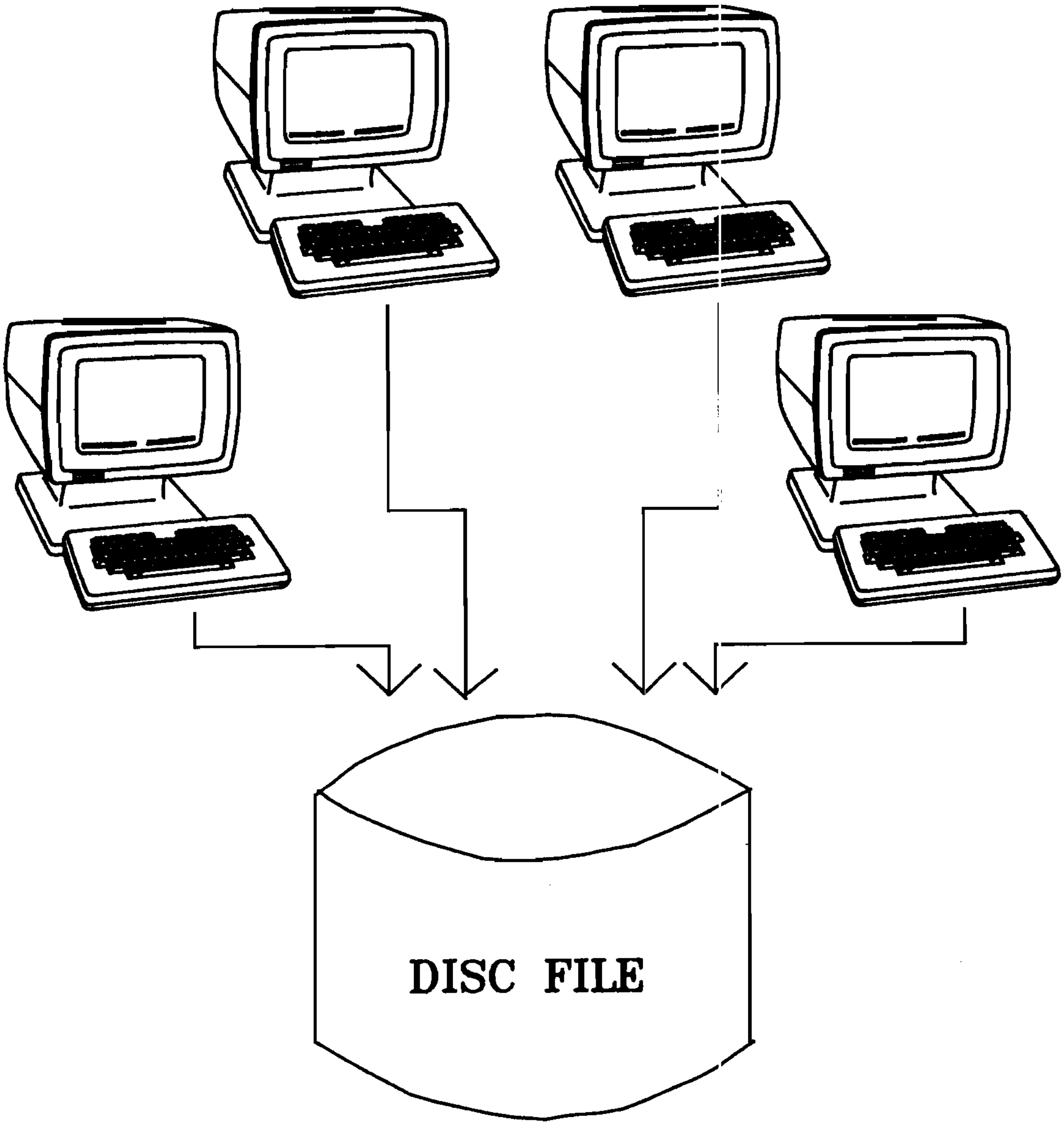
The program can issue CLASS READs to several terminals without waiting for completion.

The program (or another program) uses CLASS GETs to retrieve the inputs from the terminals as they are completed.

TERMINAL HANDLERS

EXAMPLE

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



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

8.27 A Simple Example

Consider a program which will:

* Prompt three 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.

A SIMPLE EXAMPLE –

```
      PROGRAM TERMS
C
      INTEGER LUS(3)
      DATA LUS/15,16,17/
      .
      .
      [allocate a class number]
C
C     Issue 3 prompts and reads.
C
      DO 10 I = 1,3
            LU = LUS(I)
            WRITE (LU,*) 'INPUT 10 CHARACTERS:'
            [issue CLASS READ]
10    CONTINUE
      .
      .
C
C     Retreive inputs.
C
      DO 10 I = 1,3
      [issue CLASS GET]
      [print string and LU]
20    CONTINUE
      .
      .
```