A BATCH OPERATING SYSTEM FOR A MICRODATA 1600/30 :

Intercommunication between processes

A Thesis

presented to

the Faculty of the Department of

Computer Science

University of Houston

In partial fulfillment

of the requirements for the Degree of

Master of Science

By

Jean-Luc Konrat

June, 1975

ACKNOWLEDGEMENTS

I would like to express my gratitude to my friend Xavier Mangin for his obstinacy, understanding and constant help, which made possible the design and implementation of a coherent Operating System. Thanks are extended to Professor J. Rhyne and the department of Computer Science for the financial support provided during the time of my studies.

A BATCH OPERATING SYSTEM FOR A MICRODATA 1600/30 : Intercommunication between processes

---

A Thesis

presented to

the Faculty of the Department of

Computer Science

University of Houston

In partial fulfillment

of the requirements for the Degree of

Master of Science

By

Jean-Luc Konrat

June, 1975

A BATCH OPERATING SYSTEM FOR A MICRODATA 1600/30 :
Intercommunication between processes

ABSTRACT

Part of the implementation of a general purpose batch Operating System for a MICRODATA 1600/30 is presented in this thesis. This Operating System, with a resident of 4K bytes only, works in a minimum configuration of 16K bytes. It may be used both in a batch configuration, with a spooling system, and in stand-alone configuration. In this thesis, the system is described in terms of its decomposition into processes ; a general scheme for intercommunication between processes is presented, and the control language is viewed as a tool for the description of the processes in the system. The memory constraints are solved by a succession of overlays, and the linkage to the context of a user program is demonstrated.

The description of the rest of the system may be found in the thesis of Xavier Mangin [June,1975]. Both theses are needed to get a full understanding of the Operating System.

TABLE OF CONTENTS

CHAPTER I :

INTRODUCTION

The design of operating systems on minicomputers is generally faced with the challenge of providing users with the generality of large operating systems in a limited environment. It is generally considered to be an important task which may require many man-years of work.

Two man-years were spent to design and implement the Operating System described in this thesis. The basic goal was to create a general purpose batch-oriented operating system.

1. AVAILABLE HARDWARE/FIRMWARE

The MICRODATA machine of the Computer Science department of the University of Houston, on which the system was implemented, has the following configuration :

(1) A MICRODATA 1600/30 CPU ; the CPU features include variable precision instructions, character string manipulation, stack processing (256 bytes) with stack overflow interrupt, six operational registers, a set of 110 different instructions, 8 operand addressing modes, a variable word length (1, 2, 3, 4 bytes), I/O facilities including programmed mode transfer, concurrent buffered I/O, direct memory access, detection of power fail, real-time clock.

(2) Two boards with 16K bytes of core each.

(3) The following devices ;

- a dual disk drive with two disks, each of them composed of a fix and of a removable platter.

- a parallel teletype ;
- a card reader ;
- a line printer ;
- an asynchronous communication controller ;
- a synchronous communication controller ;
- an alterable control memory (2K words) ;
- a magnetic tape controller with a 9 tracks transport.

2. AVAILABLE SOFTWARE

The following software was available at the time of the creation of the operating system :

(1) A disk - Teletype Operating System ; the disk version of TOS comprises a main program (formerly the TOS operator input program) and a set of utility programs organized in overlays. The main program contains a main operator control loop, a disk roll-in and roll-out routine and a number of routines handling the console teletype ; the set of utility programs mainly include a memory dump, a disk dump, a disk protect and unprotect, a memory and register display and update programs.

(2) A machine language symbolic assembler ; this assembler is a two pass assembler handling 3 character long labels ; it produces a listing on a teletype or a line printer ; the code generated is punched on a paper tape ;

(3) a relocatable link loader ; the relocatable link loader loads, links and initiates the execution of relocatable programs produced by the assembler on a paper tape.

3. THE REQUIREMENTS

The first step was to replace the paper tape by the disk as an intermediate form of storage. This provisional version was tested during two semesters by students using the MICRODATA machine in a Computer Science course.

The next step was to develop a true batch operating system. The following requirements had to be satisfied :

(1) the system should be available in a 16K configuration ; as much as possible should be actually available to a user program.

(2) A software scheme should be developed to compensate for the lack of any software or hardware protection.

(3) The system should be extensible, in order to meet the research needs of the Computer Science department.

(4) The system should be available under two different configurations :

- a batch configuration, to provide standard users with an efficient and automated use of the computer ;
- a stand-alone configuration, in which a particular user is able to interact with his program during its execution.

(5) The use of the system should be as simple as possible in the standard cases.

4. MAIN RESULTS

The following are felt to be the strong points of the resulting Operating System :

(1) A very general file system ; both random access and sequential access files cna be created, and referenced by symbolic names ; temporary and permanent files can be created ; files are dynamically extended by the use of an allocation map in memory ; several disk operations can be concurrently performed.

(2) A sound scheme for the handling of contexts and the intercommunication between processes ; the control language can be viewed as a tool by which a user defines the particular context of a processor.

(3) Simplicity in the standard usage of the system, by a set of well chosen default options.

(4) The system holds in 16 K and 12 K are available to a user program.

In addition to these main features, the following facilities are provided :

(1) efficient use of the input/output devices in batch configuration, by the implementation of a spooler ;

(2) possibility to dynamically redefine the input/output media of the system ; inparticular, provision for the execution of portion of control streams stored on the disk file system ;

(3) implementation of a reasonable protection of the system against unintentional destruction by a user program.

No real evaluation of the reliability of the programs is possible, since, at the time this thesis is submitted, the system has not been put into use.

5. THE CONTENTS OF THIS THESIS

The operating system was developed in close collaboration with another student, also working in a master thesis. Only part of the Operating System is described in the present thesis. In spite of the fact that a general description of the system is given in chapter III, it is felt that both theses are needed to understand fully the operating system. Frequent references to the thesis of my friend Xavier MANGIN will be found in the following chapters.

The parts of the system specifically described in this thesis include the following :

(1) the intercommunication between processes ;

(2) the monitor and the control language ;

(3) the linkage to a user program.

Two conflicting factors were faced in the writing of the present material, specifically that (1) the size of the material had to be kept within reasonable limits and (2) the thesis should give a sufficient insight of the system.

The program listings were not kept as part of the thesis, and no detailed flow-chart was included. However, an internal approach to the description of the operating system was preferred to a superficial "user manual" approach. This remark holds for the description of the control language as well.

CHAPTER II :

REQUIREMENTS

1. EXTENSIBILITY REQUIREMENTS

The first requirement for the Operating System is that it should be extensible. This Operating System is not an end product in itself, but rather a starting point for future research in the Computer Science Department. The extensibility requirement is two-fold :

(1) Minor extensions in the Operating System.
    This type of extension should be performed without a need for an extensive knowledge of the system. It includes the addition of new processors, adaptation of existing processors to specific uses, and adaptation of the system to new peripherals. These extensions should occur at a high frequency, following the needs of the Computer Science Department.

(2) Major extensions of the Operating System.
    This type of extension should occur in the development of new software. The present Operating System is composed of independant modules ; some of the modules may be kept as part of future software on the Microdata.

2. STORAGE REQUIREMENTS

The Microdata of the Computer Science Department is a machine with 32K bytes of memory. However, the Operating System is required to fit in a 16K machine. An extensive use of the disks is thus done. Disk storage includes two disks, each of them composed of a fix platter and a removable platter, with a capacity of 2.5 million bytes per platter.

The use of the disk must provide the following facilities :

(1) some parts of the disks must be usable out of the context of the operating system ;

(2) the user must be provided with the possibility of coming with his own removable platter.

3. CONFIGURATION REQUIREMENTS

The system is to be used by two categories of users : the batch users and the stand-alone users.

(1) batch users. The system is to be usable by elementary Computer Science courses (FORTRAN and Assembly Language courses) ; this usage is characterized by a high number of I/Os and simple control streams (compile-load-execute) ; this calls for :

. buffering of the I/O's in order to obtain a continuous input and output for the physical devices, independant, as much as possible, of the execution flow ; this may be achieved by a spooling system.

. choosing a system of well organized default options, in order to make the standard use of the system as simple as possible.

(2) stand-alone users. The system must be available, most of the time, to graduate students and faculty members, for them to perform :

. extension and maintenance of the system itself ;

. research involving the development of new software.

These stand-alone users may need to reserve the computer for their exclusive use during large periods of time, create personal files on one of the system disks or on a personal removable disk. They need a more sophisticated system, involving in particular the fol~~lowing facilities~~ :

. debugging tools ;

. facilities to handle personal files on the disks ;

. facilities to create new processors or to adapt existing processors to a particular user needs ;

. possibility of choosing an input or output medium different from the standard ones.

4. PROTECTION REQUIREMENTS

The Microdata machine does not provide any tool to implement protection in an operating system (memory boundaries, supervisor mode). The only protection that may then be provided is a protection against unintentional errors.

(1) Memory protection : the system has to provide a facility to restore the context of the system in the case of an alteration of system memories by a user program.

(2) Disk protection : a reasonable disk protection must be implemented to protect files of a user against access or destruction by another user.

(3) System protection : a recovery processor must be implemented.

5. RELIABILITY REQUIREMENTS

A reasonable reliability must be respected concerning the standard use of the system (compile-load-execute). Nothing more can be achieved since the user may by program zero the whole memory and/or the whole disk at any time.

6. EFFICIENCY REQUIREMENTS

In the spool configuration, the speed of the system is required to be as close as possible to the maximum speed of the card reader (300 cards per minute). In other words, efiiciency does not concern the CPU utilization but rather the card reader servicing rate.

No efficiency is required in the stand-alone configuration case.

CHAPTER III :

SYSTEM OVERVIEW

1. CONTENTS

A general description of the system is given in this chapter, with respects to the requirements given in chapter II. The first part is a description of the system from the point of view of its communications with the outside ; it describes the file system and associated drivers, the input and output of the system in spool and stand-alone configurations. The second part deals with the intercommunication between processes, and includes memory organization, communications via the memory, and samples of the control language. Only the second part will be described in greater details in the subsequent chapters of this thesis ; the first part is fully explained in the thesis of Xavier MANGIN [1] ; however, the short description given in this part should enable one to understand fully the rest of the material in the present thesis. A third part in this chapter gives a list of the processors currently available in the system. Only some of them are described in this thesis.

2. DISK MANAGEMENT

A. THE MICRODATA DISK SYSTEM

A disk platter is divided into 203 cylinders (for a density of 100 tracks/inch) ; each cylinder is composed of two tracks, one on each side of the platter ; each track contains 24 sectors ; the sector is the smallest addressable part of the disk, and contains 256 bytes of information.

B FILE TYPES AND FILE STRUCTURE

The files in the system belong to one of the two following categories : (1) random and (2) sequential access files.

(1) Random access files : they are accessible by sectors and are generally of type 'A' (such as an absolute program).

(2) Sequential access files : they are arranged in the form of records of variable length. A record does not necessarily start at the beginning of a sector ; thus, sequential access files are not normally accessed on a sector basis. Sequential access files are divided into two main types : source file types ('S') contain records of ASCII characters such as card images, source programs or listings. Relocatable files ('R') contain records of binary values such as relocatable programs created by the Assembler.

Each file is characterized by a complete file-name which is unique in the system. A complete file-name is composed of the following items :

(1) a file-id part, normally associated with a user in the system ;

(2) a user-defined name, to differentiate various files of a user ;

(3) a type ('S', 'R', or 'A').

Each platter posseses a track map and a directory. the directory and track map give the physical location of permanent files on the platter.

C. FILE ACCESS

Random access to a file is done through a routine located in the resident part of memory (.DR). Sequential access to a file is done through the Variable Length Record Driver routines ; the Variable Length Record Driver routines use .DR as a subroutine.

A file request block is set by the calling program and the address of this file request block is transmitted to the routine called (.DR or relevant Variable Length Record Driver routine) through the (X) register. The File Request Block contains the following :

(1) an internal name ;

(2) some additional routine-dependant information.

The internal name is different from the file-name. the association between internal name and file name is done by means of a special structure called "Data Chain", which is presented later in this overview. All files required by a processor or user program need to be declared on the control card under the form :

internal name = file-name

The monitor is then able to create the corresponding association in the data chain. The trace in memory of a file contains the informations needed by the disk driver routine to process the request ; this trace is called File Control Block.

D. ASSIGNMENT OF FILES TO A RUN

All files needed by a user have to be assigned to the Run before any I/O may take place. A structure contains the informations relative to all the files currently assigned to a Run ; this structure is called Current Assigned File List. Files cannot be used on a control card before being placed in the Current Assigned File List.

Two categories of files may be found in the Current Assigned File List :

(1) permanent files, the definition of which is found in one of the disk directories ;

I/O MODES

	BYTE I/O		CONCURRENT I/O		D.M.A.
	WITHOUT INTERRUPT	WITH INTERRUPT	WITHOUT INTERRUPT	WITH INTERRUPT	
TELETYPE	*				
CARD READER	*	*	*	*	
LINE PRINTER	*	*	*	*	
DISK					*
MAGNETIC TAPE	*	*	*	*	
AROM	*				
SYNCHRONOUS INTERFACE	*	*	*	*	
ASYNCHRONOUS INTERFACE	*	*			

TABLE 3.1

(2) temporary files, created by a user to store temporary informations ; these temporary files will no longer exist after termination of the current RUN.

3. INPUT/OUTPUT CONFIGURATIONS

A. PERIPHERALS ON THE MICRODATA MACHINE

There are two ways of handling an input or an output on the Microdata ; in the byte I/O mode, a byte is sent or received at each input or output operation ; in concurrent I/O mode, blocks of memory are transferred between execution of the instructions. Interrupts may be enabled in each mode.

Another mode, the Direct Memory Access, also exists in the case of disk transfers. A table (3.1) gives the available modes of transfer for each device.

B. USE OF PERIPHERALS BY THE SYSTEM

In the standard configuration of the system, only the teletype, the card reader, the line printer and the disk are used. The dedicated firmware memories for the other devices (magnetic tape, AROM, Synchronous interface) are left unused by the system and may thus be made available to a particualr processor or user program. The teletype is normally used by the system as a "system teletype" and is used for communications with the console operator, in initialization and error handling cases. However, a "user teletype" is also defined in the system, which is physically the same as the system teletype, but is used as an input/output medium. This particular usage of the teletype is normally restricted to stand-alone configuration only. Thus, the list of input media for the system include the following devices :

(1) the card reader ;

(2) a sequential access disk file ;

(3) the user teletype.

The output media are :

(2) the line-printer ;

(2) a sequential access file ;

(3) the user teletype.

C. THE STAND-ALONE CONFIGURATION

In the stand-alone configuration, the system may be viewed as a computing process, that may request input from an input medium via a call to a READ process, and that may send output to an output medium via a call to a PRINT process.

It should be noted that the "system teletype" and the "user teletype" have two different drivers, and, hence, constitute separate entities in the system. The system teletype has always priority over the user teletype driver. All these drivers wait for competion of the transfer since no buffering is possible (the input or output medium may be changed between two inputs or outputs).

D. THE SPOOL CONFIGURATION

In the spool configuration, the input flow from the card reader and the output flow to the line-printer are buffered through a list of spooled files. Two independant processes, activated by interrupts, perform the following functions :

(1) Read cards from the card reader and store them on a spool input file.

(2) Print the contents of a spool output file onto the line printer.

In this configuration, the READ process described previously normally takes its input from a spool input file ; the PRINT process produces its output on a spooled output file. These READ and PRINT processes may be broken up in two parts :

(1) a resident part ;

(2) an overlay area.

The resident part of the READ/PRINT process is independant of the particular input/output medium used. The overlay part for each input/output medium is constituted of a specific device driver and of a storage area. The input media area:

(1) the card reader ;

(2) the user teletype ;

(3) one of the user sequential access files ;

The output media are :

(1) the line printer ;

(2) the user teletype ;

(3) one of the user sequential access files.

Spooled input and output files are organised in two independant circular lists. When a spooled input file is full, the spool input process requests the next spooled input file on the list. The freed input file is then available for use by the READ process. In the same way, when a spooled input file has been emptied by the READ process, it becomes available again for the spool input process. The same considerations are true in the case of spool output files.

In addition, a change of spool input file is forced in the following cases :

(1) a disk error occurred in writing in a spool file ;

(2) a card, with two double quotes ("") is encountered by the input spooling process.

The READ from the spooled input files and PRINT on the spooled output files are the default options for the READ and PRINT processes. The user may force these processes to be directed, for a portion of his RUN, to/from another input/output media by a stacking discipline. The stackable input media include :

(1) one of the user sequential files ;

(2) the user teletype.

The stackable output media are :

(1) one of the user sequential files ;

(2) the user teletype.

The stacking and unstacking disciplines of input and output media are explained in section 5. In error cases (e.g. devices not ready or swapping out and in), a real time clock program handles the restart of the sleeping spool input or output processes.

E. CHANGE OF INPUT OR OUTPUT CONFIGURATION

The system does not provide the user with a way of passing from the stand-alone configuration to the spool configuration or vice-versa. The configuration is to be chosen at system creation. On the other hand, a special processor, called IO processor, enable the user to direct the READ or PRINT processes to take input from or produce output onto one of the overlay mediums, different from the current one. The old medium overlay is stacked on the disk, and the new medium overlay is loaded in memory. A stacking is done for each call to the IO processor in its stacking function. An unstacking is done when one of the following conditions occur :

(1) reference to the IO processor in its unstacking function by user ;

(2) end of device condition ;

(3) error condition.

An end of device condition is defined for each device in input or in output, and may be one of the following :

(1) end-of-file for an input file ;

(2) special card for the card reader ;

(3) "file full" condition for an output spool file.

A more detailed description of these end of device conditions can be found in [1]. A change of user must occur when the stack is empty. This occurs in teletype-to-teletype configuration (stand-alone case) or spool-input-file-to-spool-output-file configuration (spool case).

4. THE OVERLAY SYSTEM

The memory conflicts are solved by a succession of overlays. A special system file (or scratch file) is used to hold temporary images of memory. The memory is divided into two classes :

(1) the resident part, mostly in low core, contains mainly the swapper ;

(2) the overlay area successively contains the monitor, the processors and the users.

A. MAJOR OVERLAY

The input stream, as encountered by the read process, is composed of a succession of control cards, each of them followed by their input data (if any). Each control card enables a user to call a processor in the system. In addition to the processor name, the control card may specify :

(1) a list of files needed by the processor,

(2) a list of parameters directing the action of the processor.

The monitor is responsible for the processing of the control cards. After the processing of a control card relative to a processor, the monitor is overlayed by the processor. The action takes place via a request to the swapper in low core. The image of the overlayed area is temporarily saved in the scratch file. The processor then executes and upon completion returns control to the swapper. The swapper restores the monitor program in memory for the processing of the next control card.

B. COMMUNICATIONS VIA THE MEMORY

Transmission of information from the monitor to a processor is done via a fixed size area, in low core, constituting the "Communication area", and of a variable size structure, allocated by the monitor in high core, constituting the "processor data chain".

(1) Processor data chain. The processor data chain is defined to be the result of the processing of a control card by the monitor. It includes the definition of the disk files needed by the user, and of the input parameters specified by the user. The specification field on a control card is composed of a list of identities of the form :

internal name = file name(s) or

internal name = value(s)

The internal name is a FORTRAN identifier that is known to the processor. For instance, in the case of the ASSEMBLE processor, we may decide that :

. IN defines the source input file of the assembler ;

. OUT defines the object output file for the assembler ;

. LIST is a boolean value indicating whether a listing is required.

Standard input and output files are normally used, and are the default values of IN and OUT if the user does not specify any particular file. A default value for LIST may have been defined to be YES. The default options can be overpassed on the control card:

"ASM IN = MYFILE, LIST = NO ;

This redefines the standard input file to be the user file MYFILE and specifies that no listing is to be produced.

The processor data chain defines the association between the internal anames known by the processor, and the files and values assigned to them by the user. Several routines in the resident enable a processor to get the value assigned to an internal name or to perform a disk operation on the file corresponding to an internal name ; since several files or values may be associated with a single internal name, a sequence number is added to the internal name to characterize the file or value uniquely. The number of internal names, as well as the number of values or files associated with a given internal name varies from processor to processor and from processor call to processor call. Thus, the processor data chain is a structure of variable size, allocated by the monitor in high core. This structure is not overlayed.

(2) Communication area. A fixed size area is defined in the low resident, and contains some fixed variables in the system, such as :

. statuses and semaphores ;

. user identification and user-related variables (password...) ;

. sector addresses of priviledged information and tables on disk.

It contains in particular a pointer to the top of the current data chain for use by the resident chain manipulation routines.

C. THE RESIDENT

(1) The minimum resident of the system is composed of the following constituents :

. Low core indirections and firmware dedicated memory. This area, mostly in page zero and beginning of page 1, contains the memory boundaries for concurrent I/Os, the interrupt address of the I/O routines and firmware conditional interrupts, the stack pointer and real-time clock counter, the definition of the system files and indirection relays for resident routines.

. Conditional interrupt handling routines.

. A system stack.

. The disk driver routine (.DR) and associated routines and tables.

. The data chain manipulation routines (.FCA, FETCH, LINK, UNLINK, TYPE,...).

. The swapper and its data chain.

. The communication area.

. A system buffer necessary for swapper execution.

(2) The extended resident. The resident area may be extended by addition of some of the following routines :

. The Variable Length Record Driver routines.

. The READ and PRINT processes.

. The spooling processes (in spool configuration only).

. A system file handler.

The READ and PRINT processes and the spool process are normally part of the resident. However, some processors needing a large memory area for storage of programs or tables will try to use the smallest possible resident.

(3) READ and PRINT areas. The READ and PRINT processes, with the system file handler, constitute a particular area of the memory (X' 1000' to X'2000'). Normally a processor needing some input data flow and/or producing some output data flow (such as the ASSEMBLER) would thus begin at memory location X'2000' ; if the processor is too big to hold in the rest of the memory (X'2000' to X'3FFF'), the data following the control cards are copied by the monitor in a special file CR$; the processor then executes from the CR$ file, producing output onto a special file PR$. The monitor, after completion of the processor, then copies the contents of the PR$ file on the current system output medium via requests to the PRINT process.

(4) The spooling process area. In spool configuration, the spooling programs are normally part of the resident. They occupy the high memories (X'3980' to X'3FFF'). However the same considerations as in (3) above may hold, and a processor may execute the following serie of operations :

. halt the spooling process ;

. swap out the spooling programs and buffers ;

. use the freed area for its own storage.

Upon competion, the spooling programs are reloaded and started again. A processor may be declared to be working without the spooling programs, and then the above steps are carried out by the monitor before processor execution.

D. OVERALL MEMORY PICTURE AND EXAMPLE OF PRPGRAM

An overall memory picture may be drawn from the above considerations. The overlay area always excludes the minimum resident. Depending upon the characteristics of the processor (with spooling process or without spooling process), the monitor allocates data chains from the top memory available downwards (X'3980' or X'3FFF'). The processor data chain is always excluded from the overlay and defines an upper boundary for allocation of storage to a processor. The lower boundary of the overlay may be one of the following :

(1) Top of minimum resident (X'0BFF') ;

(2) Top of variable length record driver (X'1000') ;

(3) Top of the EXTENDED resident (X'2000').

An overall memory picture, together with details about components of both minimum and extended residents, is shown in the appendix.

An example of a program may be :

```
"RUN PASS = 1000 ;

"ASM;
        (program to assemble)
"ASM OUT-R = RELA$, LIST = NO ;
        (program to assemble)
"LOAD IN = (REL$, RELA$);

"EXECUTE CHAIN = (IN=CR$, OUT=PR$, NUM-I =
         ( -1,544);
         (data cards of user program)
"FIN
```

The execution of such a program may be described as follows :

(1) Monitor in : the monitor analyses the RUN card and creates the corresponding data chain in high core, indicating in particular that the value of PASS is 1000.

(2) RUN processor in : the RUN processor initializes a user, verifies that the password PASS is valid, creates the standard temporary files associated with the password (in particular CR$,PR$,REL$, RELA$).

(3) Monitor in : the monitor processes the next control card and determines that the assembler is to be called. The processor data chain is created with all default options ; the input of the assembler is set to be the standard input file CR$, and the monitor copies into CR$ the program to assemble following the control card. The output file is set to be REL$. The listing file is set to be PR$, and a listing is effectively required.

(4) Assembler in : the assembler executes, taking input from CR$, producing output into REL$, generating a listing in the PR$ file.

(5) Monitor in : the monitor copies the contents of PR$ to the current system output medium via the PRINT process. It processes the next control card, which is a call to the assebbler processor with CR$ as an input file, RELA$ as an output file ; data following the control card is copied into CR$; no listing is to be produced.

(6) Assembler in : the assembler executes with its input coming from the CR$ file, generating output in RELA$, producing no listing.

(7) Monitor in : the monitor copies the contents of PR$ (which is empty) on the current system output medium. The next control card defines a call to the loader with input files REL$ and RELA$, the output file being given by default (ABS$).

(8) Loader in : the loader executes and combines the two input files REL$ and RELA$ into an absolute program stored on the disk file ABS$. Listing is generated directly on the current system output medium via requests to the PRINT process.

(9) Monitor in : the file PR$ is not in the loader data chain, and thus no copy has to be done by the monitor. The control card defines a call to the EXECUTE processor. The default files and options are generated in the data chain (input from ABS$), the user data are copied into CR$.

(10) Execute processor in : the execute processor prepares the memories before giving control to the user program. It verifies that the program to execute is a valid program, saves all the memory, gives to the user access to the user program data chain (defined on the control card by IN-S = CR$, OUT-S = PR$, NUM-I = (-1, 544)), loads the user into core and gives control to the user program.

(11) User program in : The user has access to CR$ and PR$ for reading input and producing listing. It can retrieve the two values associated with the array NUM by use of the resident chain manipulation routines.

(12) Execute processor in ; The execute processor is able to swap out the whole user memory and to reload the memory saved at the begin-

ning of execution of the EXECUTE processor. It then analyses the effect of the user program (here contents of PR$) and copies the contents of PR$ (listing produced by the user program) on the current output medium.

(13) Monitor in : the monitor processes the next control card, which is a call to the FIN processor, and generates the data chain.

(14) Fin processor in : the Fin processor takes all completion actions necessary to terminate execution of the current run.

(15) Monitor in : the monitor is then able to analyse the next control card.

The above description does not seek to be accurate about the exact action of each processor, and associated control cards, but only wants to give an example of execution of a simple program.

5. PROCESSORS AVAILABLE IN THE SYSTEM

A. PROC processor

The PROC processor is mainly a maintenance processor ; certain options are not available to all users. Its function is to create new processors in the system, or modifying existing processors by freezing some parameters, changing processor names, changing defaults or reordering parameter lists.

B. IO processor

The IO processor handles the change of system input or output medium, as described in previous sections.

C. FILE processor

The FILE processor deals with the assignment, creation, deletion, recall of files (permanent or temporary).

D. ASSEMBLER processor

The ASSEMBLER processor creates a relocatable and absolute output from a source program.

E. LOAD processor

The LOAD processor creates an absolute output, ready to be executed, from several relocatable files, result of the ASSEMBLE or FORTRAN processors.

F. EXECUTE processor

The function of the EXECUTE processor is to protect the system against mistakes occurring during execution of a user program.

Only the PROC and EXECUTE processors are desrcribed in detail in this thesis. The reader is referred to [1] for description of the other processors.

CHAPTER IV

DEFINITION OF DATA CHAINS

1 . PROCESSES AND CONTEXTS

One approach to the description of an Operating System is to view it in terms of its decomposition into processes. Each of the processes constituting the Operating System is associated with a context. The context of a process is the part of the system that the process is able to access and/or modify.

With respects to this definition, the context of a Fortran subroutine consists of the program itself, local memories, parameter list and return address.

The context of a process (such as the Fortran subroutine) clearly contains two classes of elements :

(1) Some elements are strictly bound to the process, i.e they do not have any meaning for any other process (the program, the local memories).
(2) Some elements can be accessed or modified by at least two different processes in the system (parameter list and return address) ; these elements are called OBJECTS in the following discussion ; the effect of a process is completely determined by the set of all objects that the process is able to access ; the effect is measured through the set of objects the process is able to modify.

The idea here is to force each given process in the system to use a standard pattern to communicate with the outside world (by accessing and/or modifying the objects). A structure was designed to accomplish this purpose, and is defined below under the name of DATA CHAIN.

For each given process, a unique Data Chain is defined, which contains pointers to each object that the process is able to access and/or modify. In a uniprocessing system, only one process may be executing at any given time ; thus only one Data Chain may be accessed at any given time ; this particular Data Chain is called CURRENT DATA CHAIN. A fixed memory location in the resident part of the memory defines at each instant the Current Data Chain. It is called the CURRENT DATA CHAIN POINTER. A set of routines in the resident enable the currently executing process to access and/or modify the set of objects accessible through the Current Data Chain Pointer.

2. OBJECTS

An Object is an element of the Operating System which is meaningful for at least two processes in the system. According to this definition, the following items are objects :

(1) Parameters in a subroutine call

(2) Shared buffers

(3) Devices

(4) Files in the Disk File System.

Objects may be classified into three classes :

(1) Memory-bound objects :

(2) File objects ;

(3) "item" objects.

Parameters in the subroutine call, shared buffers are memory-bound objects, devices and files in the Disk File System are file objects ; "item" objects are discussed in the following sections.

Each Object is here formally defined as an association of two constituents :

(1) an ACCESS INDICATOR ;

(2) a VALUE AREA.

The Access Indicator is a simple switch that can be turned on and off ; if the switch is off, the value area cannot be changed. The protection header of a sector may be considered as the Access Indicator of the sector ; in the case of a buffer shared by two processes, a Semaphore may be created as an Access Indicator for the buffer.

A process may then take four types of "actions" with respects to a given object :

(1) READ action (read the current value of an object) ;

(2) PROTECT action (reset the access indicator of the object) ;

(3) UNPROTECT action (set the access indicator of the object) ;

(4) WRITE action (write into the value part of the object if the access indicator is ON, return in error otherwise) .

3. STRUCTURE OF DATA CHAINS

A. ITEMS

A Data Chain consists of a linked list of ITEMS, each of them being able to reference one or several Objects.

Each Item within a Data Chain possesses an identification called INTERNAL NAME. The Internal Name of an Item must be unique within the Data Chain in which it is found.

All the Objects referenced by a given Item have the same nature (MEMORY-BOUND objects, FILE objects, ...). A TYPE represents this nature, and is part of the item definition. This type is associated with the Item as well as with the referenced objects.

Thus, an item is the way for the process to reference a particular set of objects. As seen in the preceding section, four types of actions may be taken by the process with respects to one of the objects (READ, WRITE, PROTECT, UNPROTECT). The Item Definition specifies which actions the process is allowed to perform on any of the Objects referenced by the Item. Four bits, corresponding to each of the actions, are RESET if the process is not allowed to perform the corresponding action upon the referenced Objects.

B. BINDING OBJECTS TO ITEMS

The Items described above are characteristic of a process : the internal names, types and action bits do not change from one execution to another. However, the actual objects referenced by an Item may vary from one execution to another. The action of establishing the linkage from Item to Objects is called BINDING objects to the Item.

The linkage from Item to Objects is done via a linked list of pointers, as shown in fig. 1. VAL points to a linked list ; each element in the list is able to reference a single Object. Binding objects to an item consists in creating this linked list.

We already mentioned in the Object classification that Items could be considered as Objects. A special type is associated with such Objects.

In the case of an Item of type Item, the VAL pointer points to an entire Data Chain (see fig. 2).

The process associated with the data chain containing an item of type item is called PARENT PROCESS. The Data Chain pointed at by the VAL pointer is called DEPENDANT DATA CHAIN ; the associated process is a SUBPROCESS of the parent process. In this definition, an Item is viewed as an Object of a Parent process. The value field of the Item object is defined to be the binding of the item. Perform a WRITE action upon such an Item object consists in binding objects to the item. An additional bit had to be introduced in the Item definition to represent the access indicator of the item as an object of a parent data chain ; if this bit is reset, the current binding of the item cannot be changed.

The binding of objects to items thus becomes the responsibility of the parent process. A process can modify the access indicator or value

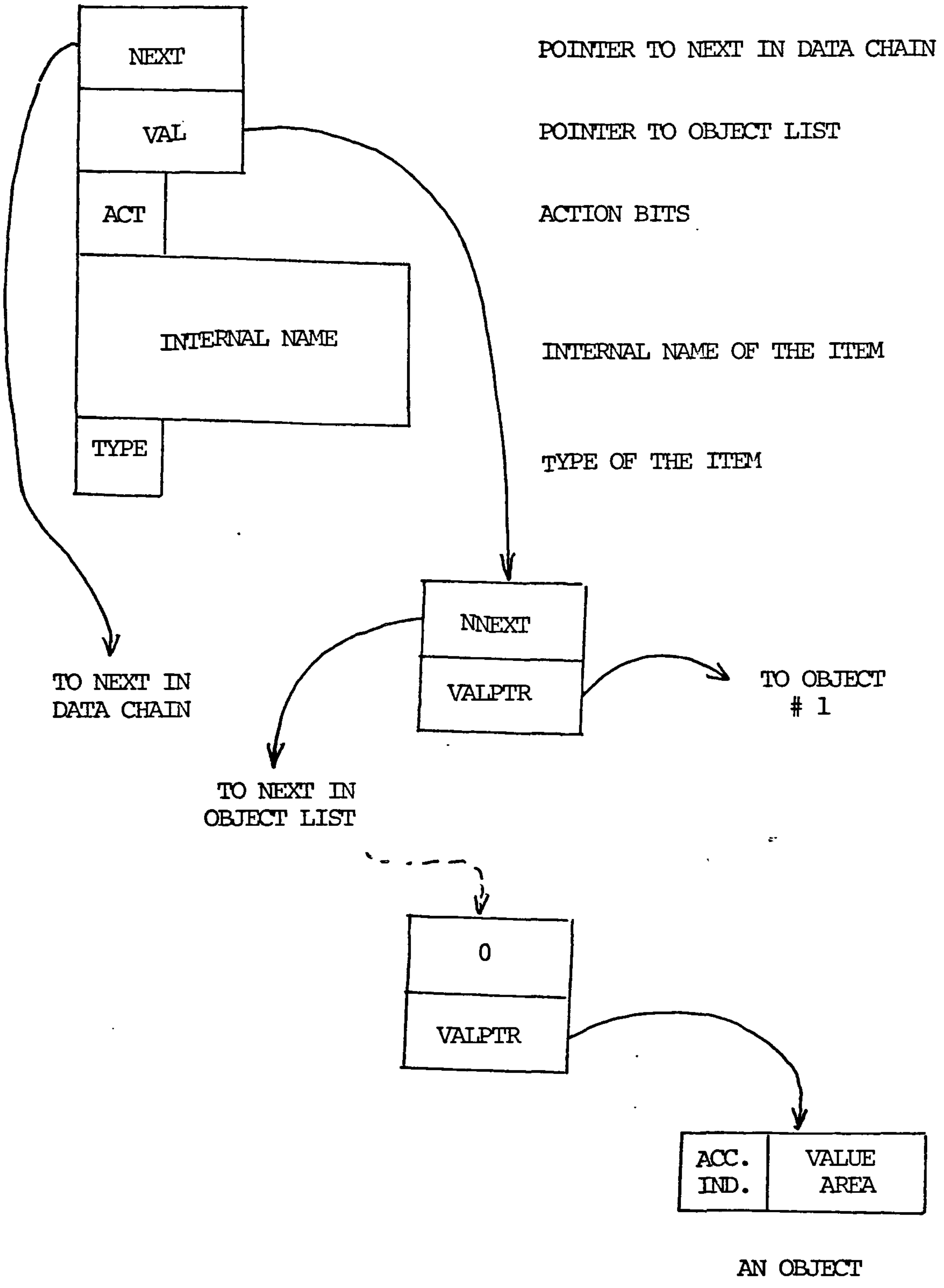


Fig 1: AN ITEM AND LINKAGE TO OBJECTS

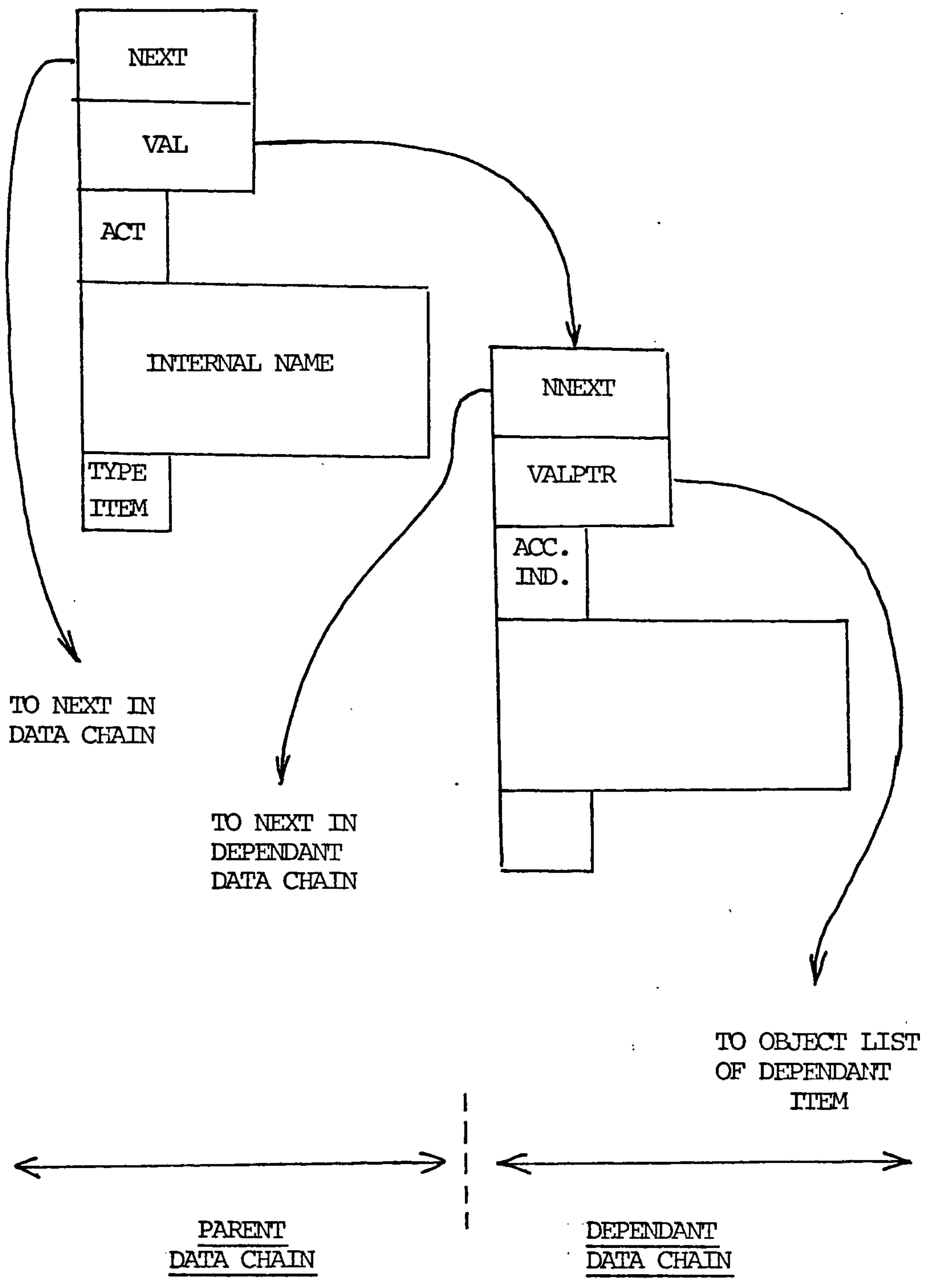


Fig 2 : CASE OF AN ITEM OT TYPE ITEM

field of all objects accessed through its Data Chain ; it cannot change the actual binding of any Item in its own data chain. This situation is very similar to the above example of a Fortran subroutine call. The subroutine can change the value of any parameter in the parameter list. However, it is the responsibility of the main program to associate each parameter in the formal list with a specific address, and this address cannot be changed by the subroutine.

This definition of Items as particular objects adds the power of recursion to the definition of data chains. The number and addresses of the objects associated with the items become dynamic for the parent process. The Internal Name, Type and action bits of the items in a data chain cannot be changed by either the parent or dependant processes. They are characteristic of the process itself, and enable the process to access the objects in a standard fashion. In the Fortran example, they are similar to the formal parameter list in the subroutine definition.

4. ACCESSING OBJECTS

Reference to an object for a READ, WRITE, PROTECT or UNPROTECT action is done through the Current Data Chain.

A set of routines, located in the resident part of the memory, enable a process to use the current data chain. The particular Name and address of the routine may depend upon the TYPE of the Item referenced, and the action required by the process. All routines have a standard calling sequence, specifically :

LDX= Request block address

CAL Subroutine needed

The REQUEST BLOCK is set by the calling process. It always includes the following information :

(1) an IDENTIFICATION of the object within the current data chain ;

(2) some subroutine-dependant information.

The Identification part may be standard between all the routines in the package ; it includes the following :

(1) INTERNAL NAME of the Item ;

(2) a relative OBJECT NUMBER identifying the particular object desired within the list of objects referenced by the Item ;

(3) the TYPE of the Item.

For the routine to be successfully executed, at least five conditions must be met :

(1) the Internal name exists within the current data chain ;

(2) the Type specified in the Request Block matches the type of the Item determined by the Internal Name ;

(3) The Object Number specified in the Request Block is less than or equal to the number of objects associated with the item ;

(4) the action required is valid, i.e the corresponding bit in the item definition is not reset ;

(5) if the particular action requested is a WRITE, the Access Indicator of the object referenced is not reset.

Each of the actions (READ, WRITE, PROTECT, UNPROTECT) is defined below for each object category (MEMORY-BOUND objects, FILE objects, ITEM objects). According to the definitions in Table 1, READ (PARAM(4)) consists in reading the fourth object associated with the Item defined by the Internal Name PARAM ; SEMON (MEM(2)) consists in turning ON the Access Indicator of the second object associated with the Memory-bound item defined by the internal name MEM ; BIND (DATCHN(3)) consists in defining a binding for the third Item of a subchain of the current data chain.

5. CHANGE OF CONTEXT

A change of context in the system merely consists in altering the value of the CURRENT DATA CHAIN POINTER. This can be done by three different disciplines :

(1) Queueing discipline ;

(2) Preemptive discipline ;

(3) Linking discipline.

The Queuing discipline implies an identity of priorities between the processes, and may be used to handle multiprogramming situations.

The Preemptive discipline is used to handle interrupts or interrupt-like situations when a process has to be serviced in priority. The preempting context is fully independant of the preempted context. The chain of the preempting context is generally of a fixed-binding nature.

The Linking discipline makes use of the Items of type Item, as defin-

ACTION \ OBJECT TYPE	MEMORY - BOUND OBJECT	FILE OBJECT	ITEM OBJECT
READ	FETCH The value of a set of memory locations	READ A file on the disk file system	UNBIND get the value of the pointer to object list
WRITE	SET memory locations to a given value	WRITE on the disk file system	BIND an object list to a dependant item
PROTECT	SEMON turn a semaphore on	PROTECT a portion of the disk file system	SECURE an item against ulterior binding
UNPROTECT	SEMOFF turn a semaphore off	UNPROTECT a portion of the disk file system	RELEASE reset the protect. indicator of an item

Table 1 : DATA CHAIN MANIPULATION PRIMITIVES

ed above. The parent context "contains as a value" the entire Data Chain of the dependant context. This scheme corresponds to orderly situations such as a call to a subroutine. It is the responsibility of the parent process to make sure that all items in the dependant data chain are properly bound before the subprocess is enabled. Linking the new context is in this case the responsibility of the parent process, and the scheme makes use of a stack to store successive values of the current data chain pointer.

The preemptive and linking disciplines are sketched in fig 3.a,b.

6. CREATION OF DATA CHAINS

The question arises of when data chains should be created. It was already mentioned that the Items in a data chain and their identification (Internal Name, Type, READ/WRITE/PROTECT/UNPROTECT bits) cannot be changed, since they are characteristic of the process with which the Data Chain is associated. Thus, a permanent image of a data chain may be kept on the Disk for each process defined in the system. The creation of the Data Chain is done merely by mapping the permanent image in memory. The binding for the items of the permanent images is called DEFAULT BINDING. The permanent image is itself called DEFAULT DATA CHAIN. An Access Indicator is defined for each item in the DEFAULT DATA CHAIN, and specifies whether or not the default binding can be modified.

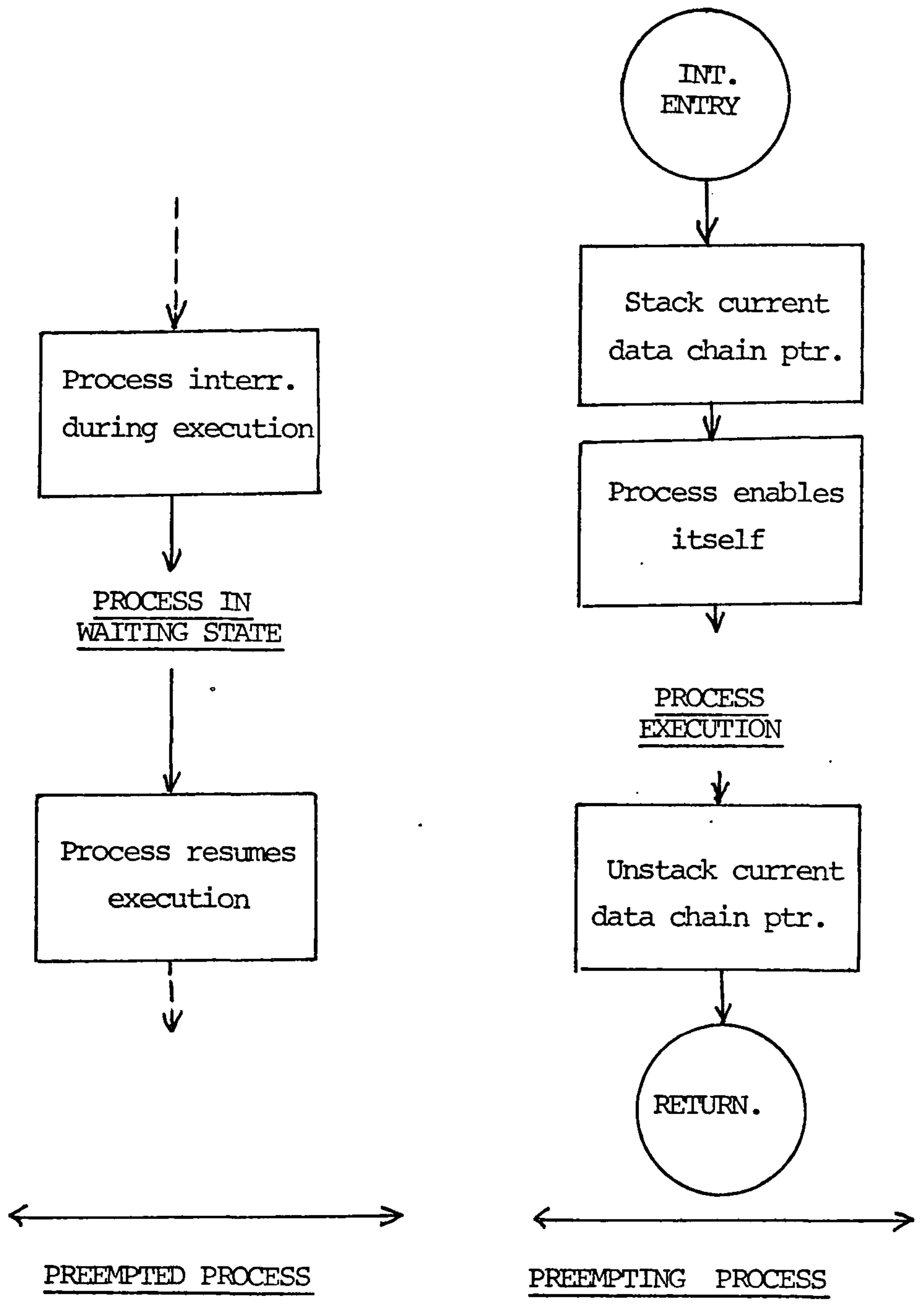
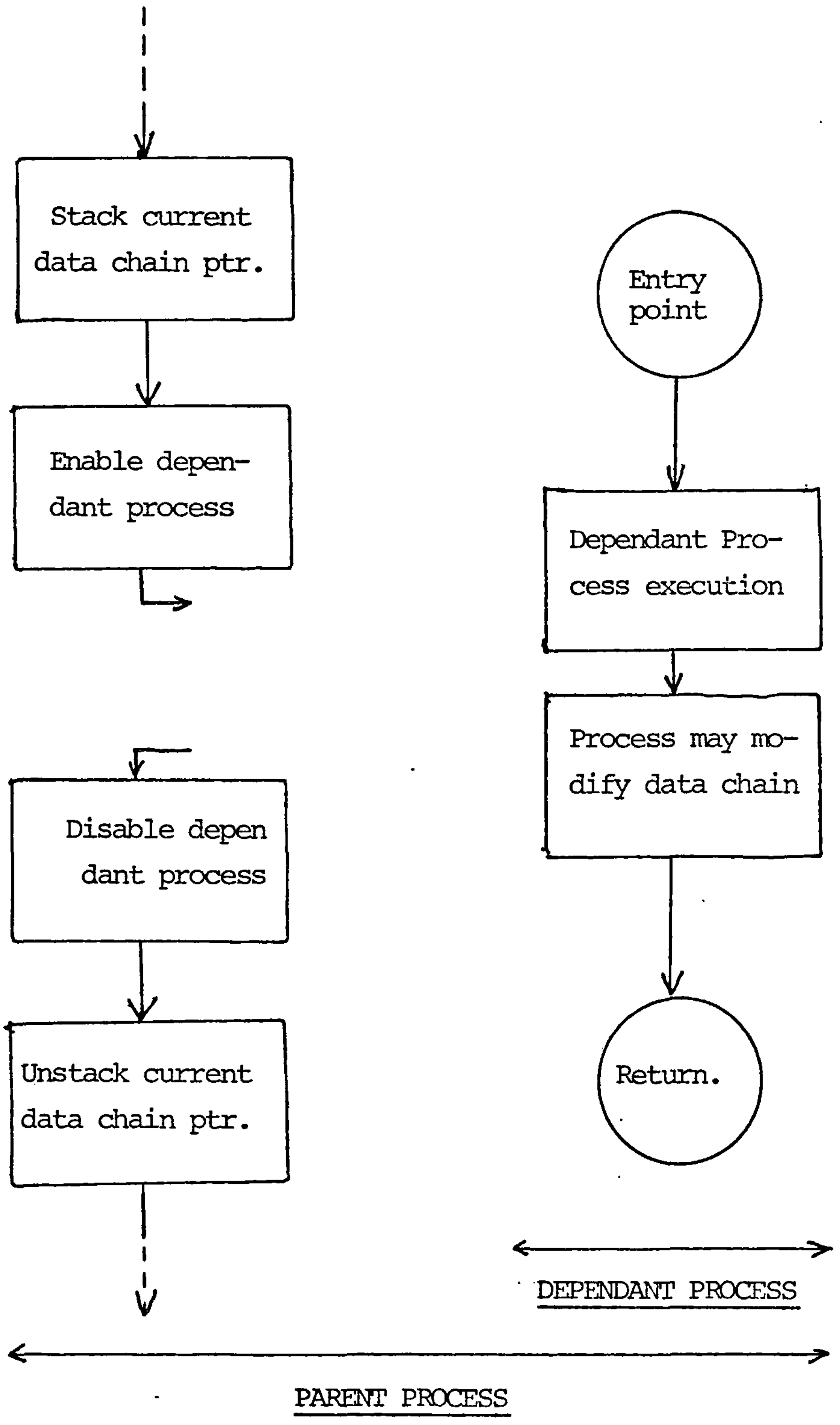

INT. ENTRY
Stack current data chain ptr.
Process enables itself
PROCESS EXECUTION
Unstack current data chain ptr.
RETURN.
Process interr. during execution
PROCESS IN WAITING STATE
Process resumes execution
PREEMPTED PROCESS
PREEMPTING PROCESS

Fig 3.a CHANGE OF CONTEXT, PREEMPTIVE DISCIPLINE



Stack current data chain ptr.
Enable dependant process
Disable dependant process
Unstack current data chain ptr.
Entry point
Dependant Process execution
Process may modify data chain
Return.
DEPENDANT PROCESS
PARENT PROCESS

Fig 3.b CHANGE OF CONTEXT, LINKING DISCIPLINE

A process that has no parent in the system is called MASTER process. Its Data Chain is necessarily of a fixed-binding nature, and is created at the same time that the process itself is brought in memory.

Processes with variable bindings must be enabled by a parent process after their associated Data Chain are created.

CHAPTER V

IMPLEMENTATION OF DATA CHAINS

1. PROCESSES IN THE SYSTEM

In the description of the implemention, we will call process any program associated with a Data Chain. As will soon be seen, the implementation of Data Chains is costly memory-wise. The number of processes in the system was thus reduced to the minimum.

The MASTER processes (i.e the processes having no parent) are defined to be :

(1) the MONITOR process ;

(2) the SPOOLING process ;

(3) the SYSTEM INPUT process ;

(4) the SYSTEM OUTPUT process ;

(5) the SYSTEM FILE HANDLER process ;

(6) the REAL-TIME CLOCK process.

Only the MONITOR and the REAL-TIME CLOCK processes will be described in detail in this thesis. The other processes (SPOOLING, SYST. INPUT, SYST. OUTPUT, SYST. FILE HANDLER) concern the input/output of the system, and are described as such in [MANGIN,(1)].

The other processes in the system are all subprocesses of the MONITOR process. They will be described in this thesis. These DEPENDANT processes include in particular :

(1) the SWAPPER subprocess ;

(2) the PROCESSORS ;

(3) the USER-DEFINED PROGRAMS ..

The creation of the Data Chains for these subprocesses is done by the MONITOR upon user's request ; the mechanism is described together with the MONITOR's description.

2. IMPLEMENTATION OF DATA CHAINS

A. ITEM DEFINITION

As specified in the preceding chapter, a DATA CHAIN consists in a linked list of ITEMS, each of them being able to reference one or several objects. All pointers in Data Chains are absolute ; the last element in a chain has a zero pointer.

The INTERNAL NAME of items is implemented as a FORTRAN identifier, i.e at most 6 letters or digits, beginning by a letter ; characters are stored as ASCII characters (1 byte per character) ; the INTERNAL NAME is completed to 6 characters by trailing blanks on the right.

In the implementation, TYPES consist in a unique ASCII character. The possible types are :

(1)	'I','C','H','F','B'	(MEMORY-BOUND objects)
(2)	' ','S','R','A'	(FILE objects)
(3)	'D'	(ITEM objects)

The ACTION BITS are stored together in a whole byte, though only 5 bits of this byte are used. Each of the lower 4 bits corresponds to a different action, as shown in Fig. 4 ; the bit is set if the process is allowed to perform the action on the objects associated with the item, reset otherwise. The fifth bit is the Access Indicator of the item considered as an object of a parent chain ; it is set if the process can change the binding of the item, reset otherwise.

B. LINKAGE OF OBJECTS TO ITEMS

The linkage of objects to items was defined as a linked list in order to preserve the full recursivity of the structure, in particular with respects to items considered as objects of a parent data chain, via an item of type ITEM.

The linked list was kept in the particular case of items of type ITEM. For cases involving Memory-bound objects and File objects, the pointers to objects (VALPTR in fig. 1) were stored as a linear list starting from the memory location pointed at by the VAL pointer. An additional byte had to be introduced to specify the number of objects in the linear list. Thus, from 0 to 255 objects may be referenced by a single item. For N objects, the savings in terms of memory is :

2 x N bytes of core.

a) another simplification was done in the case of memory-bound items, and is detailed below. In almost all instances, one of the two following conditions was met :

(1) Many processes can read or alter a given object, but this object is of a fixed binding nature, such as a system variable ;

(2) Once bound, the object can only be read by a single process in the system, and cannot be written into (input parameter for a processor) .

Objects in class (1) were simply blocked in a dedicated area in low core, called the COMMUNICATION AREA. The Communication Area includes thus such variables as RUN variables, fixed memory addresses, sector numbers, status bytes, semaphores, a.s.o. A description of the communication area is contained in appendix A. This simplification presents at least two major drawbacks :

(1) it becomes difficult to see which process in the system "possesses" (i.e is able to access and/or modify) a given variable in the communication area ;

(2) processes become sensitive to modifications in the communication area structure.

These drawbacks were partially overcome by :

(1) grouping the communication area objects so that the variables relevant to a given family of processes be stored consecutively in memory ;

(2) forcing all processes to access each object in the communication area in the following standard fashion :

- fetch the address of the top of the relevant group in page zero of memory ;
- access the desired by an offset from this address.

The scheme is sketched in fig. 5 case b.

Since objects in class (2) interest only one process in READ-ONLY fashion, their value is placed within the data chain before activation of the process ; no pointer at all need to be stored

, and this results in non-negligible savings in terms of storage : in the case of an object that can be stored in one byte of core, this simplification cuts the storage needed in a factor of one to three (1 byte needed for storage of object instead of 1 byte for the object and 2 bytes for the pointer) .

b) the same scheme could not be extended to the case of FILE objects. Files are shared objects in the system, and are used extensively in all READ/WRITE/PROTECT/UNPROTECT cases. Two simplifications were made in order to ease the implementation and usage :.

(1) the sharing of the same FILE OBJECT by two different items belonging to the SAME data chain was simply forbidden in the system, as a safeguard against errors ; this is not really a restriction, since DIFFERENT processes can access and/or modify the same file object (see fig. 5, cases d and e).

(2) very few cases were met where the need was felt to use multiple file objects assciated with a single item (however see LOADER data chain example, in [1]) ; the idea was kept but merely implemented as several different items, each of them pointing onto a single file object. The internal names of the items differ only by their OBJECT # byte. The behaviour of the resulting structure is defined to be as if a single item was associated with all the file objects. However, the new scheme is more costly memory-wise, since repeated storage of the internal name is required (see fig. 5,case c).

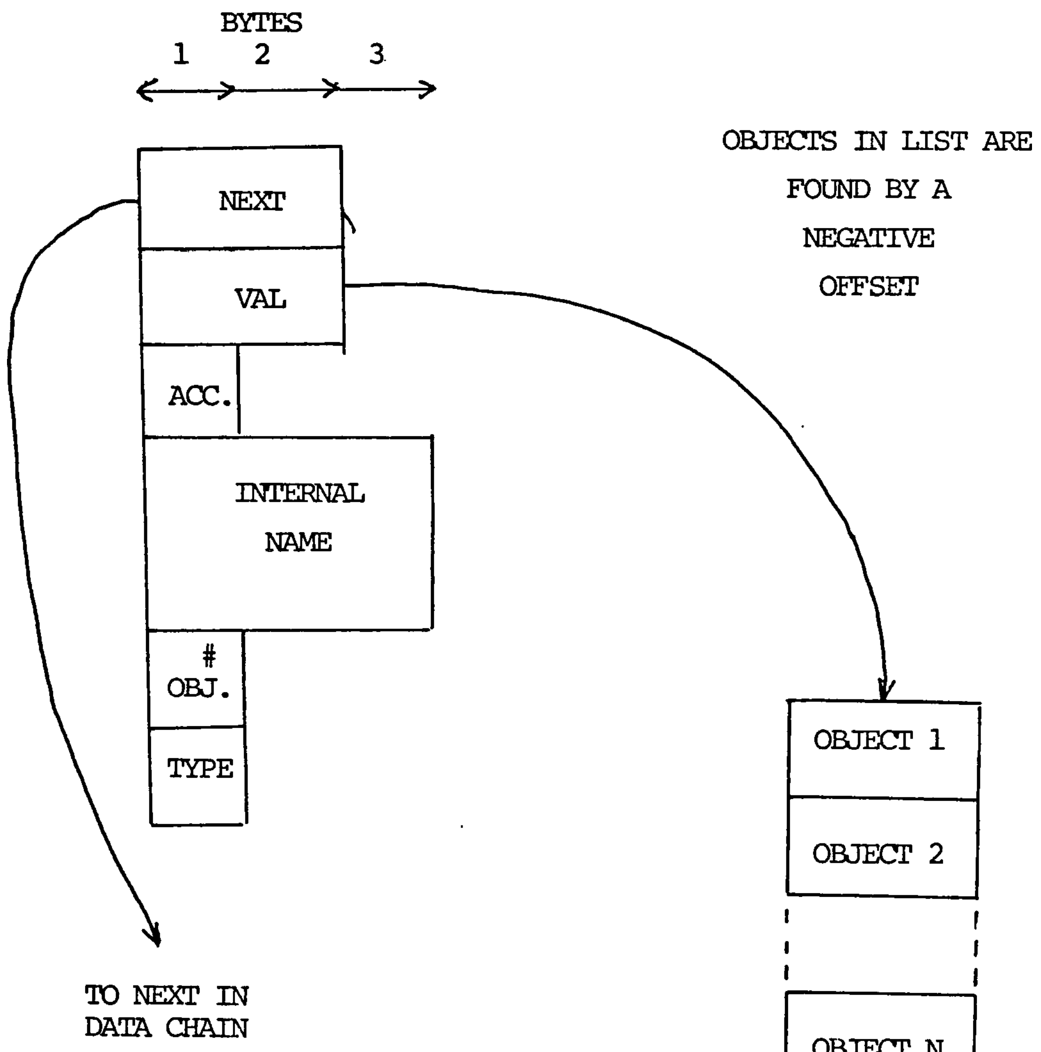


Fig 5 : IMPLEMENTATION OF DATA CHAINS
Case a : MEMORY-BOUND OBJECT, READ_- ONLY, ACCESSIBLE BY ONE PROCESS ONLY

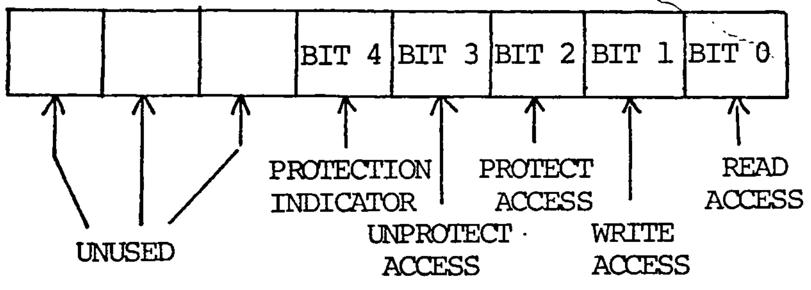


Fig. 4 : ACTION BITS AND PROTECTION INDICATOR

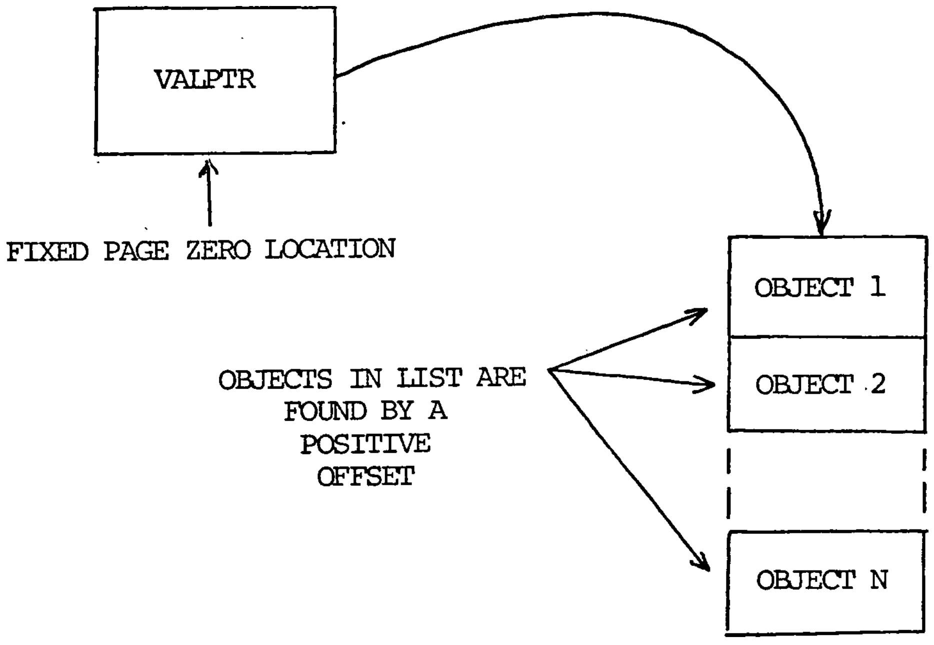


Fig 5, case b : MEMORY-BOUND OBJECT, FIXED-BINDING CASE

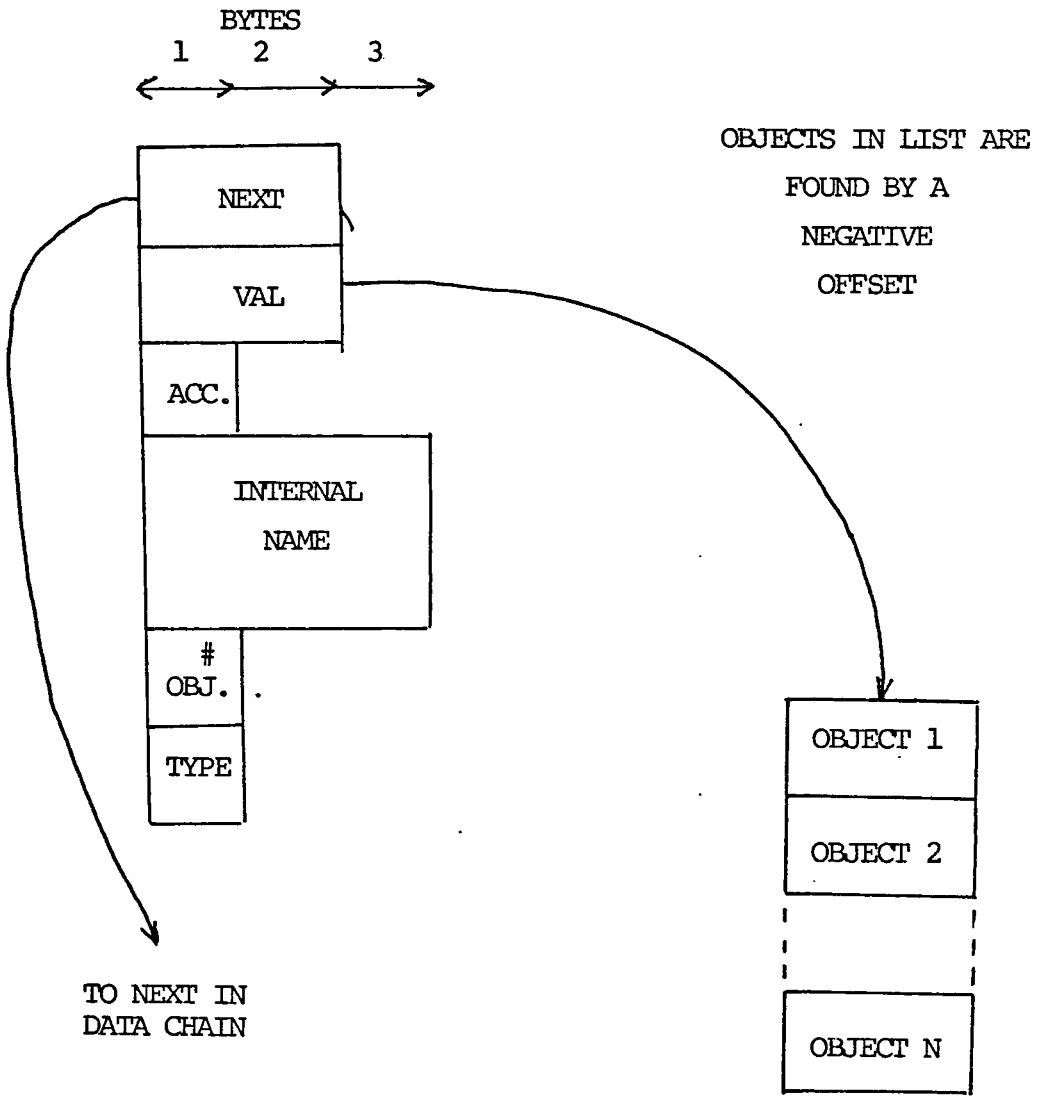


Fig 5 : IMPLEMENTATION OF DATA CHAINS
Case a : MEMORY-BOUND OBJECT, READ_- ONLY, ACCESSIBLE BY ONE PROCESS ONLY

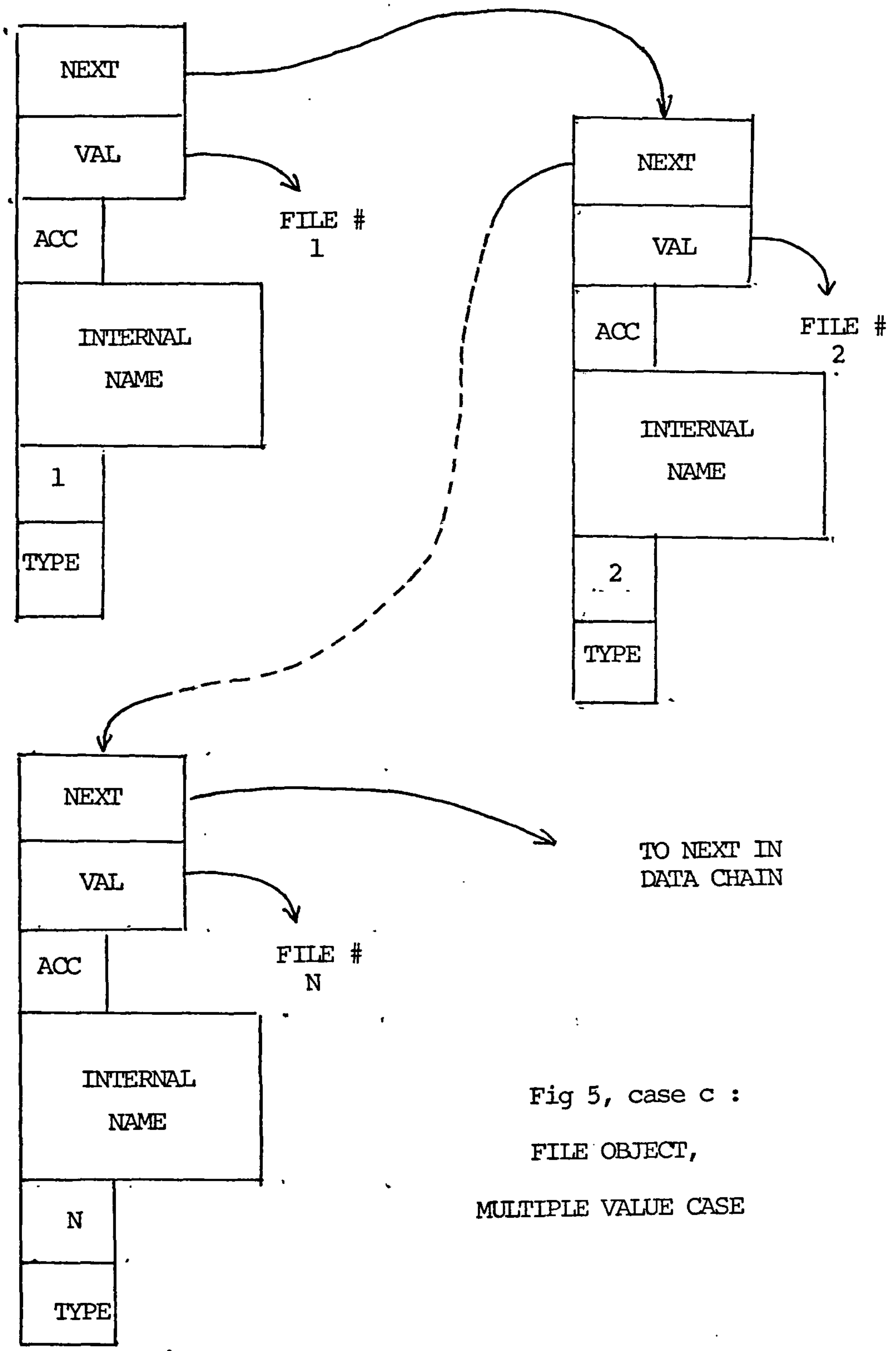


Fig 5, case c :

FILE OBJECT,

MULTIPLE VALUE CASE

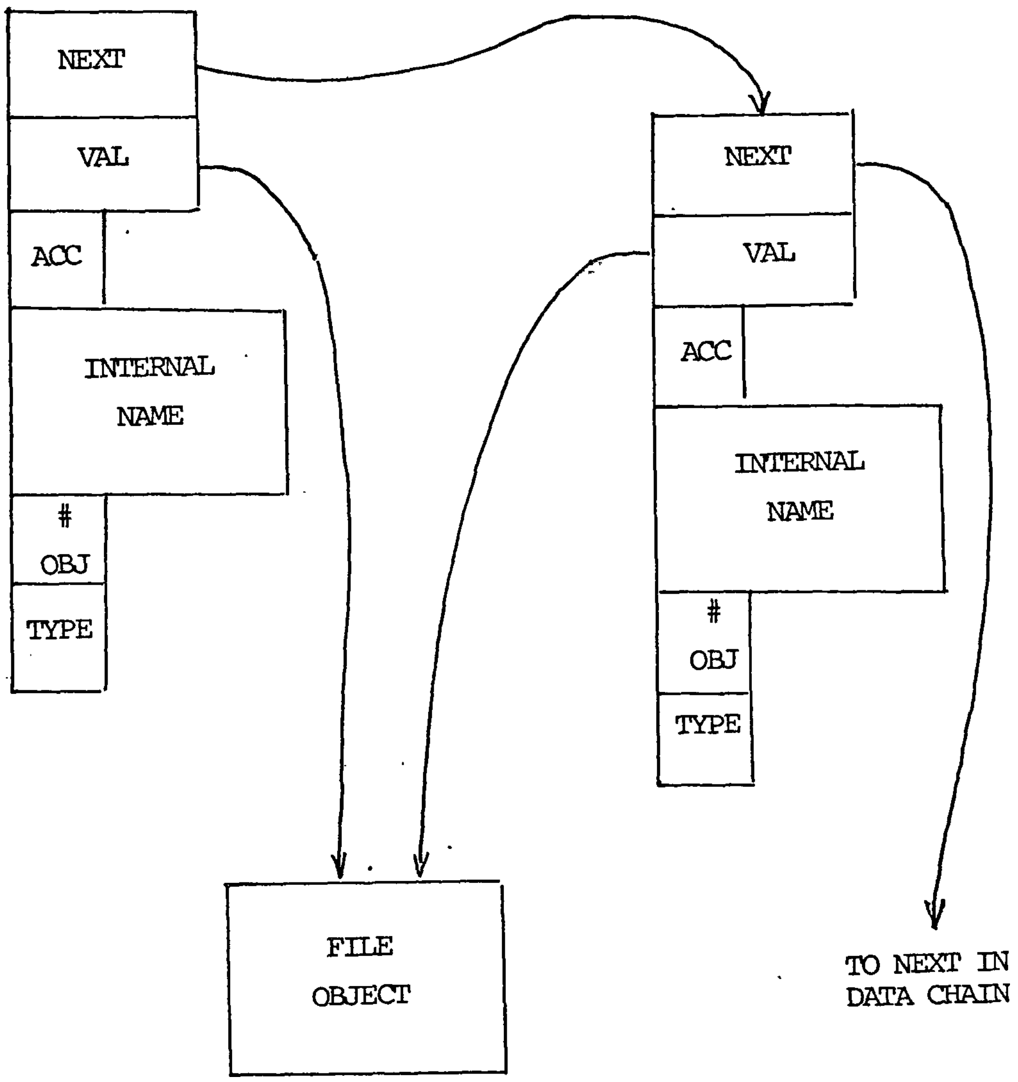


Fig 5, case d :
FORBIDDEN SHARING OF A FILE OBJECT BY
ITEMS IN THE SAME DATA CHAIN

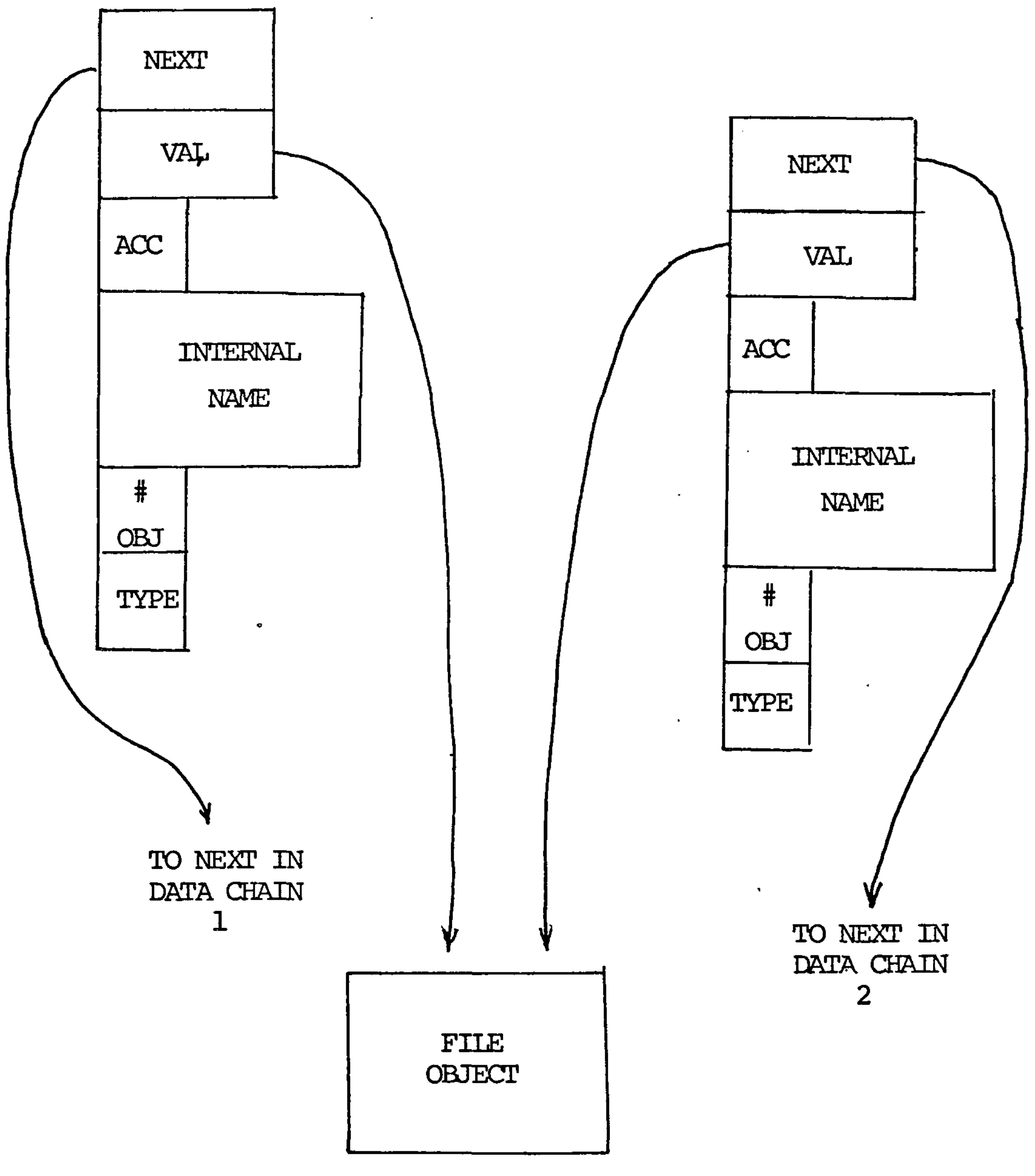


Fig 5; case e

ALLOWED SHARING OF A FILE OBJECT BY

DIFFERENT PROCESSES

3. OBJECTS AND OBJECT TYPES

A. OBJECTS IMPLEMENTED

Among the list of possible objects mentioned in the formal presentation, the following were implemented through use of data chains :

(1) read-only parameters in subroutine calls ;

(2) objects of type item .

The items listed below were implemented by use of the communication area through page zero indirection :

(1) buffer shared between processes (by placing the semaphore in the communication area) ;

(2) devices (by the spooling system) ;

(3) READ/WRITE only memory-bound objects of fixed binding type.

General READ/WRITE/PROTECT/UNPROTECT memory-bound objects of variable binding were not implemented in the 16 K version of the system.

B. PROTECTION MECHANISMS

The formal presentation insisted on a protection mechanism for each object ; the idea was maintained throughout the implementation.

(1) Objects of type ITEM : The additional bit "protecting" each item was implemented as will be seen in the description of the monitor .

(2) File objects : a suitable protection mechanism was implemented by use of the hardware header on each sector of the disks ; an additional software protection, on the basis of a file in the disk file system, was implemented to prevent

a use to destroy another user's or a system file.

(3) Memory-bound objects : no hardware protection is offered by the computer microprograms over any specific memory location. Memory-bound items treated through use of data-chains need no protection indicator since they are READ - ONLY by definition. Protection of objects in low core, and in particular in the communication area, was insured by two mechanisms :

. implementation of a relocating loader "filtering" a process's requests for indirections in page zero, depending on the degree of reliability of the process ;

. implementation of an "execute program" processor to execute a user's program ; the entire memory image is saved before (and restored after) the user's program execution.

A description of the relocating loader can be found in [1], and the EXECUTE processor is described in details in this thesis.

It can be noted that no specific PROTECT/UNPROTECT routine was implemented.

C. ITEM OBJECTS

The associated item type is 'D'. An item object can be shared by several different processes, though no example of this was actually tried. No recursion is possible in the 16 K version of the Operating System, and thus no process can have itself or a parent process as a sub-process.

D. MEMORY-BOUND OBJECTS

The types associated with memory-bound objects treated through use of data chains include :

'I'	INTEGER OBJECT, FULL WORD	Core used : 2 bytes
'H'	HALF-WORD INTEGER OBJECT	Core used : 1 byte
'C'	ASCII CHARACTER OBJECT	Core used : 1 byte
'B'	BOOLEAN OBJECT	Core used : 1 byte
'F'	FILE-NAME OBJECT	Core used : 8 bytes.

No routine to READ/WRITE/PROTECT/UNPROTECT was written, since a copy of the value of the object is done each time the process is activated.

E. FILE-BOUND OBJECTS

The types associated with FILE objects include the following :

' '	ANY FILE IN THE DISK FILE SYSTEM
'S'	FILE OF TYPE 'SOURCE' (TYPE 'S')
'R'	FILE OF TYPE 'RELOCATABLE' (TYPE 'R')
'A'	FILE OF TYPE 'ABSOLUTE' (TYPE 'A').

The trace in memory of a File object is a FILE CONTROL BLOCK, succintly presented in the overview ; the File Control Block is described in details in [1].

4. ACCESSING OBJECTS

Reference to an object for a READ, WRITE, PROTECT or UNPROTECT operation is done by use of the CURRENT DATA CHAIN or the COMMUNICATION AREA.

A. ACCESS VIA THE COMMUNICATION AREA

Accessing an object within the Communication Area is done via an indirection relay in page zero of memory.

READ action :	LDX	Indirection relay
	LDA+	Offset to object required
WRITE action :	LDX	Indirection relay
	STA+	Offset to object required

No explicit Protect or Unprotect can be done by a process, and the objects in the communication area remain constantly unprotected.

B. ACCESS VIA THE CURRENT DATA CHAIN

As defined in the formal presentation, the process sets-up a REQUEST BLOCK, and calls the relevant routine in the resident part of the memory. The beginning of a request block is standard between all the routines, and is shown in fig. 6. It identifies a unique object within the Current data chain. A different appendage may be necessary, depending on the routine called.

A single routine (.FCA) is used by all routines to fetch the address of a particular item within the current data chain. The calling sequence for this routine is :

```
LDX=  Request Block address
CAL   .FCA
```

The output is stored in the registers of the calling program as follows :

(X) remains unchanged

(B) points to the top of the object list or is equal to zero ; a zero value of the (B) register means that no item in the current data chain satisfies all of the following conditions :

(1) Internal name matches internal name specified in R.B.

(2) Type matches type specified in R.B.

(3) # OBJ. byte in item definition is greater than or equal to the OBJ # specified in the R.B.

(A) is undefined if (B) is zero, otherwise (A) contains the address of the matching item.

The routine is flowcharted in fig. 7.

Other chain manipulation routines make use of .FCA as a subroutine, and are succintly described below.

a) Case of FILE objects : A single routine (.DR) may be used to perform all actions (READ/WRITE/PROTECT/UNPROTECT) . The appendage needed to the Request Block is shown in fig. 6.
The OP. REQ. byte must be set to 1,2,3,4 in order for the routine to perform respectively a Read, Write, Protect or Unpro-

tect operation ; no multiple operation can be done at a time. Several routines in the resident constitue the Variable Length Record Driver and make use of .DR. The File Request Block needed in the calling sequence is compatible with the general Request Block format. All these routines are described extensively in [1] .

b) Case of Memory-bound objects : only the READ operation was implemented. The routine FETCH in the resident can be called by a process and is able to match the current value of a single object into the area of the calling process. The calling sequence is :

```
LDX=  Request Block address
LDB=  address to store the object's value
CAL   FETCH
```

The output of the routine is in the (B) register : (B) is zero if the request is in error (bad return from .FCA) , non-zero otherwise. The other registers are unmodified.

c) Case of item objects : The four actions (UNBIND, BIND, SECURE, RELEASE) were formally thought as part of the resident. However, these constitute complex operations that cannot be kept permanently in memory. The solution adopted is that all binding of chains are to be done at the time the Data Chain is created (i.e brought in memory) ; this can be :

. at system generation time for MASTER processes ;

. at control-card processing time for the other processes.

The corresponding actions are thus performed by the MONITOR.

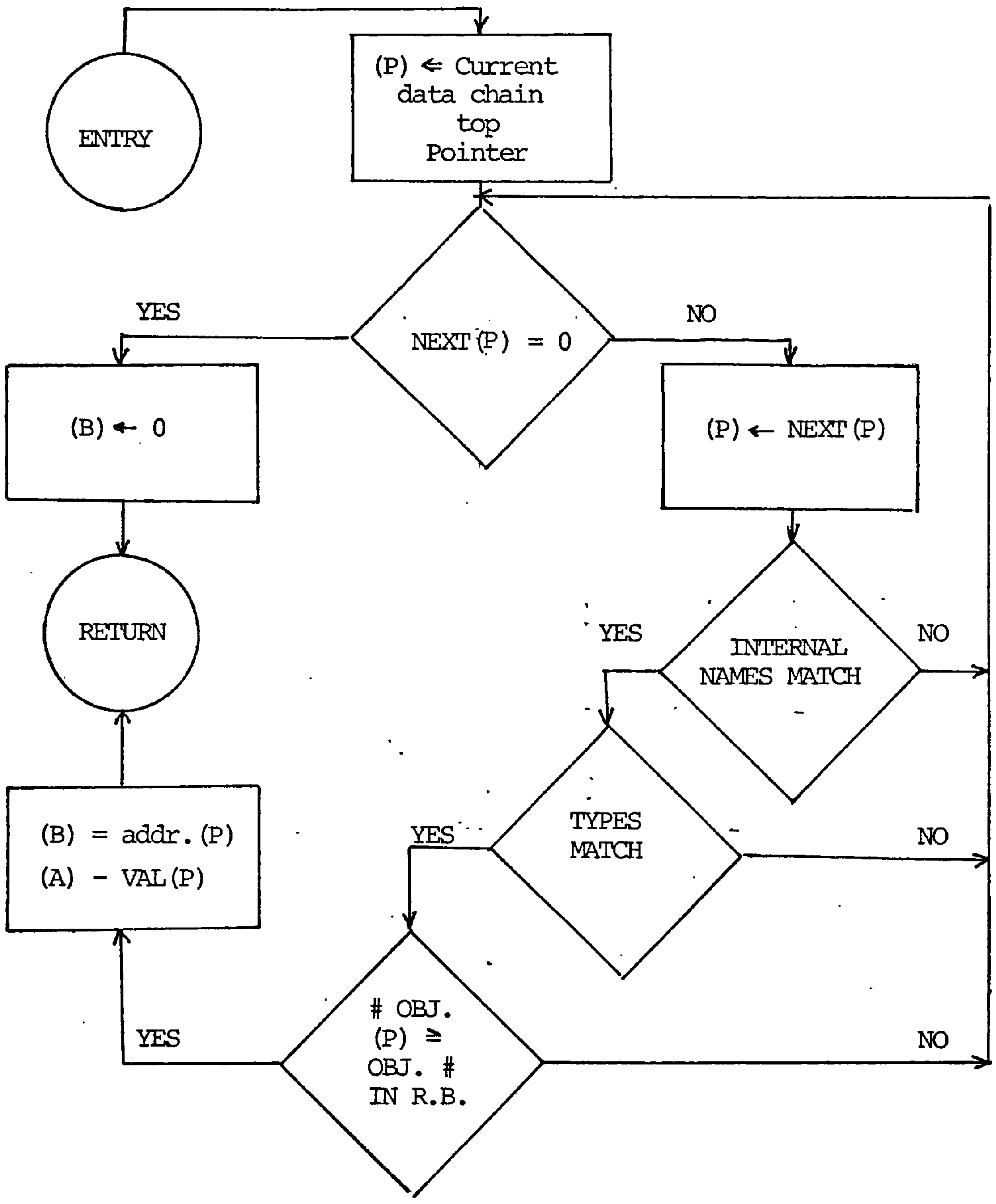


Fig 7

FLOWCHART .FCA

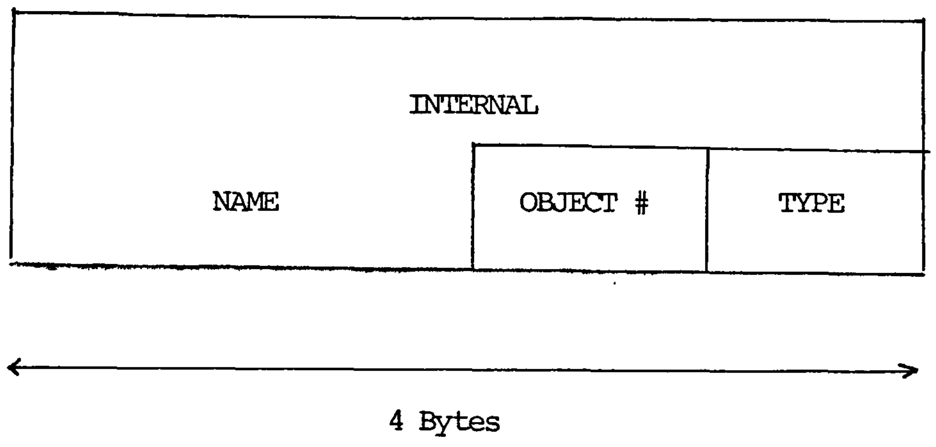


Fig 7

REQUEST BLOCK FORMAT

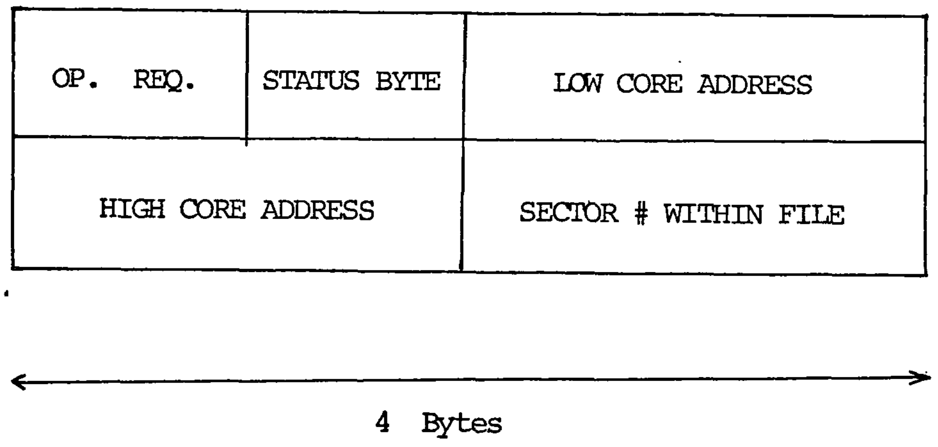


Fig 8

FILE REQUEST BLOCK APPENDAGE FOR

USE BY .DR ROUTINE

OBJECT TYPE / ACTION REQUEST.	MEMORY-BOUND OBJECT THRU COMMUNICATION AREA	MEMORY-BOUND OBJECT THRU DATA CHAIN	FILE-BOUND OBJECT	ITEM OBJECT

READ	LDX Page Zero Indirection
	LDA+ Offset
		LDX= Request Bl. address
		LDB= Core addr.
		CAL FETCH
			LDX= File Req. Block addr.
			CAL .DR
			(OP. REQ. = 1)
				by MONITOR only

WRITE	LDX Page zero Indirection STA+ Offset	not implemented	LDX= File Req. Block addr. CAL .DR (OP. REQ. = 2)	by MONITOR only
PROTECT	not implemented	not implemenyed	LDX= File Req. Block addr. CAL .DR (OP.REQ. = 3)	by MONITOR only
UNPROTECT	not implemented	not implemented	LDX= File Req. Block add. CAL .DR (OP. REQ. = 4)	by MONITOR only

Table 2 : TABLE OF IMPLEMENTED PRIMITIVES AND CALLINGSEQUENCE

5. CHANGE OF CONTEXT

Among the possible change of context described in the formal presentation, two were implemented :

(1) the PREEMPTIVE discipline to handle interrupts ;

(2) the LINKING discipline otherwise.

A. CHANGE OF CONTEXT BY LINKING DISCIPLINE

The data chain to be linked is a subchain of the current data chain. A small stack (for storage of current data chain pointers) and two routines in the resident (LINK and UNLINK) accomplish the purpose.

A Request Block is set-up by the parent process ; its format is given by the general request block format. The parent process then executes the following code :

```
LDX=  Request Block address
CAL   LINK
```

The routine LINK then stacks the current value of the current data chain pointer and updates it to the top of the dependant data chain. The output is indicated in the (B) register of the calling program, and is zero if an error occured ; this error return may be caused by :

(1) bad return from .FCA (i.e no item in the current data chain meets the specifications in the request block)

(2) stack overflow.

The parent process then jumps to the start address of the dependant process, and waits for its completion.

Upon return from the subprocess, the parent process executes a call to the UNLINK routine :

```
                    CAL    UNLINK
```

The UNLINK routine resets the current data chain pointer to the value on the top of the stack, placed by the LINK operation.

B. CHANGE OF CONTEXT BY PREEMPTIVE DISCIPLINE

In this case, the preempting context is fully independant of the preempted context ; the preempting process is a MASTER process (it has no parent in the system) .

Since the data chain of a process is not a subchain of any parent chain, the MASTER process is always assumed to know the address of the top of its own data chain ; two routines in the resident (PREEMP, RELEAS) allow such a MASTER process to preempt the currently executing process without disturbing its execution. An additional difficulty was met in implementing the scheme, as will be seen in the REAL-TIME CLOCK PROCESS description chapter.

The calling sequence for the two resident routines is respectively :

```
                    LDB=   Top of chain pointer address
                    RTX    PREEMP
and :               RTX    RELEAS
```

The routines have no output.

6. CREATION OF DATA CHAINS

Data chains of MASTER processes are of a fixed binding nature. They are mapped in memory together with the process.

Other processes are all subprocesses of the Monitor process. Only the SWAPPER data chain is permanently in memory, since it is a part of the resident. The processor and user program data chains are created by the Monitor at control-card processing time ; the binding for these chains is permanently determined by the Monitor at the same time.

DEFAULT PROCESSOR DATA CHAINS are kept permanently on the disk. They are mapped in memory by the monitor at control-card processing time. The algorithms to create and bind data chains from the processing of a control card and a default processor data chain are described together with the Monitor description.

CHAPTER VI :

APPROACHES TO THE DESCRIPTION OF THE OPERATING SYSTEM

In viewing the system as composed of processes and subprocesses, we can follow two approaches :

1. DESCRIPTION BY PROCESS

The first approach is to describe each process existing in the system. Processes are considered with respects to the following :

(1) initial presentation of what the function of the process is in the overall system.

(2) external behaviour describing a process by looking at all the objects that the process is able to manipulate ; this includes the following :

. behaviour with respects to the process data chain.

. behaviour with respects to the communication area ;

. behaviour with respects to subprocesses ;

. behaviour with respects to system files ;

(3) internal behaviour of the program itself.

In this thesis, the following processes are described with various level of details :

(1) the SWAPPER subprocess (chapter 7);

(2) the REAL-TIME CLOCK process (chapter 8) ;

(3) the MONITOR process (chapter 10), with the control language ;

(4) the PROCESSOR subprocesses general format (chapter 9) ;

(5) the EXECUTE processor, and linkage to a user program (chapt. 11);

(6) the PROC processor (chapter 12).

A description of the other processes in the system (READ and WRITE procesess, SPOOLER process, IO, FILE, ASSEMBLE, LOAD processors) can be found in (1) or (2).

2. DESCRIPTION BY OBJECTS

The second approach is to describe the system by looking at the objects. The description is three-fold :

(1) contents : general description of what the function of the process is as a means of communications between processes.

(2) structure : description of the objects in terms of its components and formatting.

(3) usage : this involves a description of the standard routine(s) accessing the object and a cross reference table of the processes accessing or modifying the object.

A desciption by objects may incluse the following :

A. MEMORY DESCRIPTION

The memory can be described as a serie of invariant areas (residents) and overlay areas. A description of the various overlays of the memory is present in appendix to this thesis.

(1) Data chains. The contents, structure and usage of data chains is described abundantly througout this thesis.

(2) Page zero. The first page of the memory (page zero) contains indirections to access the communication area and routines of the various residents, as well as fixed sector

addresses within system files.

(3) Communication area. The communication area is divided into several sub-areas, each of them being defined by an indirection in page zero. The sub-areas include the following :

. I/O subarea ; this area contains information relative to the current system configuration, and is described in [1].

. RUN subarea ; this area contains information relative to the current user of the system, such as his password, file-id... It contains in particular the RUN byte, giving the current status of the current RUN ; the RUN byte is described in the MONITOR chapter.

. .FC subarea ; this area is a general communication area for such processes as the SPOOLER process, the READ and PRINT processes, and the REAL-TIME CLOCK process. It also contains the current data chain top pointer.

. PROC sub-area ; this area constitutes a provision for transmission of fixed arguments between monitor and processor. A detailed description of this sub-area can be found in the processor general description chapter.

B.DESCRIPTION OF THE FILE OBJECTS

A general description of the disk file system is not done in this thesis, but is present in [1]. A short description can be found in the system overview chapter. Files include the following :

(1) files of type 'S' (normally sequential access, ASCII characters) ;

(2) files of type 'R' (normally sequential access, binary records);

(3) files of type 'A' (normally random access) ; the format of absolute files is described in [1].

A particular attention should be placed onto system files. System files are particular files that are not directly available to a user program. A special handler for these files is part of the extended resident, and constitute the SYSTEM FILE HANDLER process ; it is succintly described in the processor general description chapter. System files are not described in this thesis, but can be found in [2]. Basically, the system files are :

(1) a scratch file (SCR$) ; this file is used by the system as a virtual extension of the memory, as mentioned in the description of the swapper (chapter 7), the EXECUTE processor (chapter 11) and the READ-PRINT processes (in [1]) ;

(2) processor and library files respectively containing the absolute and relocatable programs available in the system.

A particular processor file is PRO$, containing the monitor and the major processors ; PRO$ also contains the particular input-output drivers, as fetched by the IO processor (see [1]) ;

(3) directory files ; these files contain for each disk a permanent track map and a directory of all the files on the disk ;

(4) spooled files ; these files are used as a buffer in the system by the SPOOLER process ;

(5) the RUN$ file, describing :

. the current status of the system (processors, file-ids...)

. and information concerning the current user of the system.

This file contains in particular the CURRENT PROCESSOR TABLE, CURRENT ASSIGNED FILE LIST and PROCESSOR INFORMATION AREA described in this thesis.

CHAPTER VII :

AN EXAMPLE OF CHANGE OF CONTEXT BY LINKING DISCIPLINE : THE SWAPPER PROCESS

1. THE OVERLAY SYSTEM

The system makes an extensive usage of the technique of the change of context by linking discipline. We have seen in the overview that the system involves a continuous overlay pattern, following the alternance :

(1) MONITOR/PROCESSOR/MONITOR or

(2) MONITOR/EXECUTE PROCESSOR/USER/EXECUTE PROCESSOR/MONITOR.

The SWAPPER is the program that actually performs this alternance. It has its own Data Chain, and thus constitutes a particular process. The SWAPPER and its chain are entirely located in the resident part of the memory.

A more accurate description of the overlay pattern would involve in the case (2) :

MONITOR/SWAPPER/EXECUTE PROCESSOR/SWAPPER/USER/SWAPPER/
EXEXUTE PROCESSOR/SWAPPER/MONITOR

At least four contexts would have to be created :

(1) the MONITOR context ;

(2) the SWAPPER context ;

(3) the EXECUTE PROCESSOR context ;

(4) the USER-DEFINED context(s) .

As will be seen in the description of the EXECUTE PROCESSOR, the swapper bringing the user program in core must possess special properties, in order to protect the system against destruction by the user. For this reason, it was decided to use a particular routine different from the general SWAPPER ; this routine does not have a context of its own, but rather is a part of the EXECUTE PROCESSOR program ; it is described as such in this thesis.

Thus, in terms of contexts, the system involves one of the two following patterns :

(1) MONITOR context/SWAPPER context/PROCESSOR context/
SWAPPER context/MONITOR context/...

(2) MONITOR context/SWAPPER context/EXECUTE PROCESSOR context/
USER-DEFINED context(s)/EXECUTE PROCESSOR context/
SWAPPER context/MONITOR context/...

Such a loop is associated with each call to a processor.

Assuming that the monitor is able to create and bind all data chains involved, one execution of the loop can be handled simply by use of the linking mechanism ; the part of the memory that is not overlayed must include at least :

(1) the COMMUNICATION AREA and SWAPPER in low core

(2) the PROCESSOR DATA CHAIN in high core.

Diagrams in Fig 9 and 10 describe one execution of the loop in terms of change of context and show the Data Chains involved.

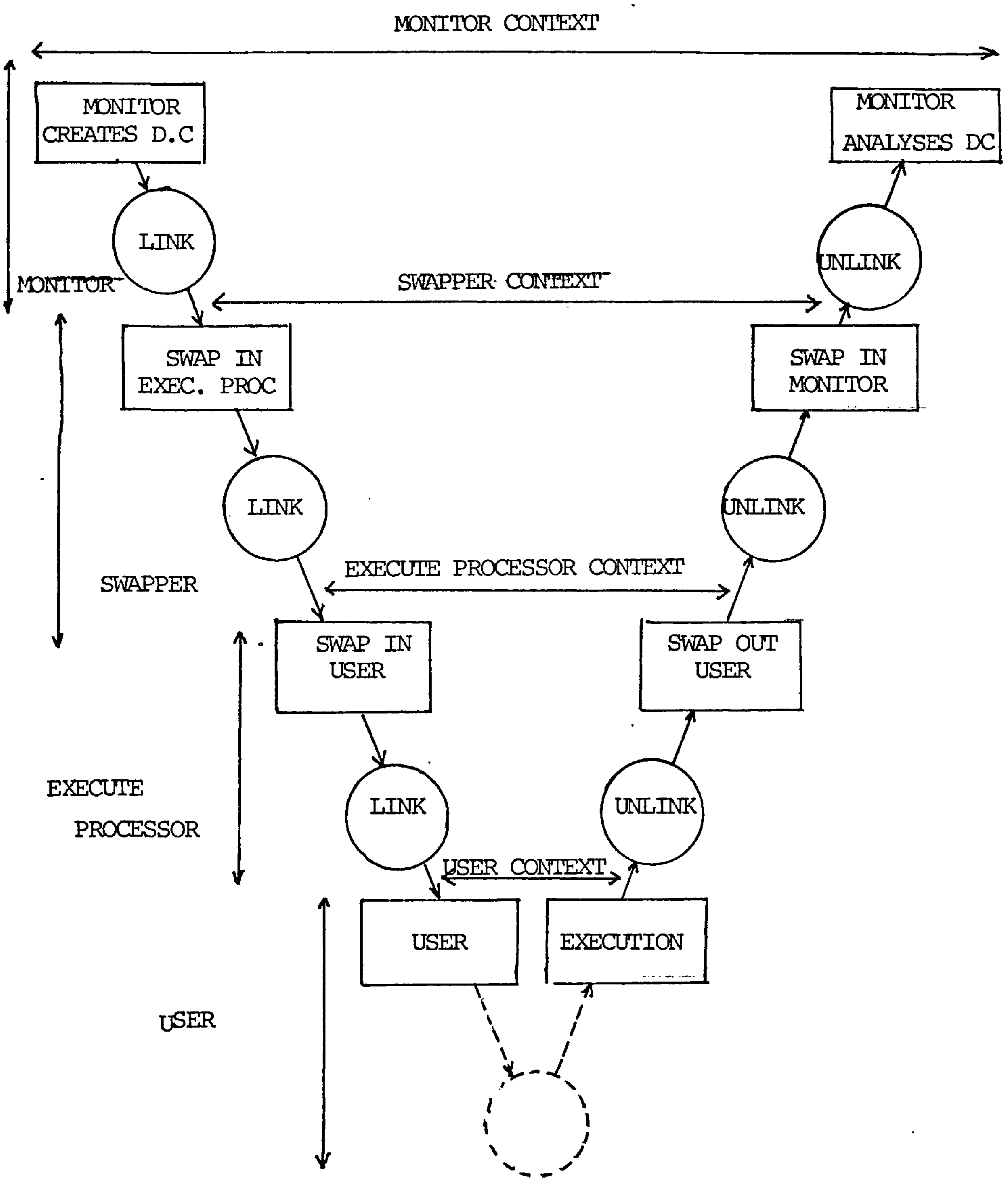


Fig 9 : LINKAGE TO A USER PROGRAM

DIAGRAM SHOWING THE CHANGES OF CONTEXT

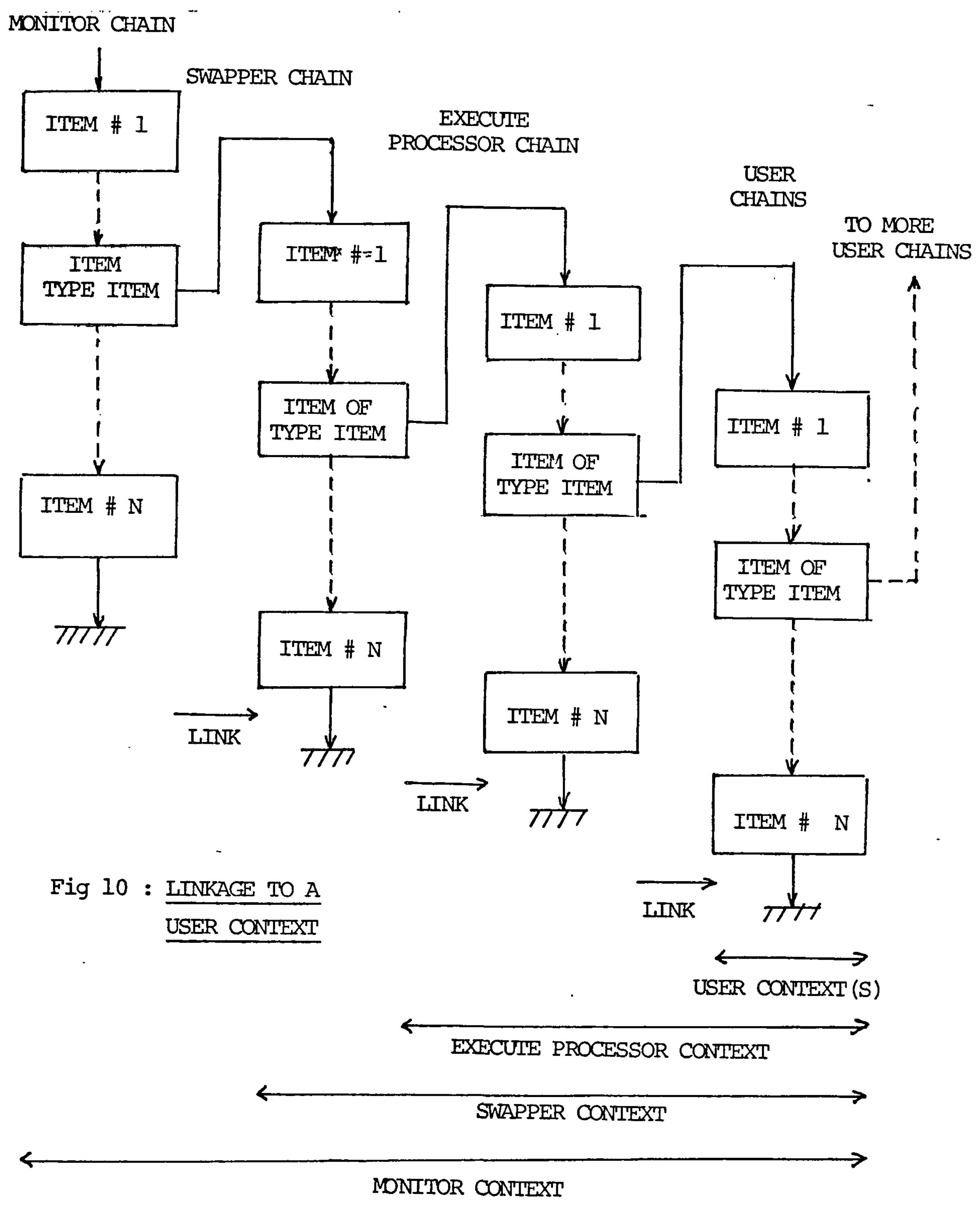


Fig 10 : LINKAGE TO A USER CONTEXT

2. THE SWAPPER

The SWAPPER subprocess is entirely part of the resident. It enables the MONITOR to load a processor in memory, and is able to restore the monitor image upon return from the processor.

The effect of the Swapper program can be described internally by :

(1) save the current monitor image on the SYSTEM SCRATCH FILE ;

(2) load the processor in memory ;

(3) link the processor data chain ;

(4) jump to the processor start address ;

(5) wait for processor completion ;

(6) unlink the processor data chain

(7) reload the last monitor image from the SYSTEM SCRATCH FILE ;

(8) return control to the monitor program .

The SWAPPER DATA CHAIN is entirely in low core. Except for the item of type ITEM defining the PROCESSOR DATA CHAIN for the LINK operation, all bindings of the swapper data chain are also in low core.

The swapper data chain is composed of the following items :

(1) an item of type 'D' (CHAIN) bound by the monitor to the processor data chain in high core ;

(2) a read-only file-bound item, bound by the monitor to the File Control Block of the file where the processor absolute program is to be found ;

(3) a read-write file-bound item of fixed binding (MONI) defining the System Scratch File where the monitor image is to be saved temporarily during the processor execution .

In addition to the objects reached via its data chain, the swapper objects include the following :

(1) the value of the registers upon input :

- (A) contains the start address for the monitor swap ;
- (B) contains the end address for the monitor swap ;
- (X) contains the sector # within file where the processor absolute program is to be found .

(2) a single byte within the PROC section of the communication area. This byte is set to zero by the monitor before execution of the swapper ; the swapper program may change the value of the byte in case of errors, as is shown below :

- SWP = 1 (error in swapping monitor out) ;
- SWP = 2 (error in reading the processor dictionary) ;
- SWP = 3 (error in reading the processor program into core) ;
- SWP = 4 (error in trying to reload the last monitor image) ;
- SWP = 7 (errors 3 and 4 combined) .

(3) a fixed address in page zero, defining the sector address within the System Scratch file where the monitor image is to be temporarily saved.

CHAPTER VIII :

GENERAL SCHEME FOR THE EXECUTION OF A PROCESSOR

As explained in the system overview chapter, each processor constitutes a separate process in the system. A single processor is called for each processing of a control card by the monitor. The processor absolute image file, sector number within file, and PROCESSOR DATA CHAIN are found by the monitor on the disk (PROCESSOR INFORMATION AREA WITHIN RUN$ FILE). The bindings of the processor data chain and the core boundaries for the overlay are determined by the monitor ; the swapper chain is also bound by the monitor program.

Thus, a processor can have effects on the system by the following :

(1) the Processor Data Chain ;

(2) the Communication area ;

(3) requests upon other processes (SYSTEM FILE HANDLER PROCESS, READ, PRINT processes, SWAPPER subprocesses) .

1 EFFECTS THROUGH THE PROCESSOR DATA CHAIN

The Processor Data Chain is the normal way for a processor to :

(1) read input parameters set by the monitor from the processing of the control card;

(2) LINK and UNLINK subprocesses ;

(3) act upon user files on the disk (READ, WRITE, PROTECT, UNPROTECT). The processor data chain typically is used for processor-dependant effects on the system, and thus is not described further in this chapter. A description of processor data chains and meaning of each item is found in the description of each particular processor.

2 EFFECTS THROUGH THE COMMUNICATION AREA

A. VARIABLE EFFECT

Variable effects upon the communication area can be expected from a ~~processor,~~ ~~depending upon~~ the sub-areas that the processor is able to access and/or modify.

B. COMMUNICATION WITH THE MONITOR

A special sub-area (the 'PROC' sub-area) is reserved for communication of special fixed-size information between the monitor and the processors. A detailed description of this sub-area follows ; each object in the sub-area is considered.

(1) MAXMEM : The maximum address actually available to a processor may change from execution to execution, since processor data chains in high core are of variable binding. This memory location contains the high core address available to the processor ; the processor may read this value to optimize buffer sizes, or define upper boudaries for variable-size tables.

(2) PROTIM : This memory location contains at each instant the time remaining before a processor time-out occurs (see REAL-TIME CLOCK process in following chapter) ; this value is initialized by the monitor, and is decremented by the REAL-TIME CLOCK process every time period.

(3) SWP : This byte is used by the swapper to signal abnormal swapper behaviour to the monitor, and has already been described.

(4) ERRFRB : 16 memory locations are reserved to the handling of abnormal disk errors by the processor. When an abnormal return from the disk handler (.DR) occurs, the processor is able to place the File Request Block in this area for subsequent interpretation by the monitor ; since at the time the File Request Block is copied the OP.REQ byte in the FRB has been set to zero by the .DR routine (see [1]) , an additional byte (OPREQ) is needed to hold the value of the OP.REQ byte before execution of the disk handler.

(5) INTADD : This address is a special entry in the processor program to which the REAL-TIME CLOCK process may jump if one of two conditions hold :

. A console interrupt has been recognized ;

. A processor time-out condition occurred.

This address is initialized by the MONITOR to a value specified in the PROCESSOR INFORMATION AREA of the processor called ; the processor may update it during its execution. The address is typically the one of a closing sequence of the processor (see fig. 11).

(6) PROC : This byte is initialized to zero by the monitor before control is given to the processor. A normal execution of a processor is characterized by a zero value of this byte upon return.

Each bit contains a one value if a specific error occurs, as detailed below :

- bit 0 (low order bit) : set to 1 if an abnormal return from .DR occured and the File Request Block corresponding to the erroneous disk operation has been placed in ERRFRB and following.
- bit 1 : set to 1 if an abnormal return from .DR occured and the status of all files not in read-only is not guaranteed; this means typically that a disk error occured in the midst of the closing sequence of the processor (see fig 11).
- bit 2 : set to 1 if an error occured in the manipulation of a SYSTEM FILE ; this bit is set to 1 automatically if the operation in error occured via a request to the SYSTEM FILE HANDLER process.
- bit 3 : set to 1 by the REAL-TIME CLOCK process if a processor time-out condition was detected.
- bit 4 : set to 1 by the REAL-TIME CLOCK process if a console interrupt was recognized during the execution of the processor.
- bit 5 : set to 1 if a STACK OVERFLOW interrupt was processed during the execution of the processor.
- bit 6 : set to 1 if a POWER FAIL interrupt was processed during the execution of the processor.
- bit 7 (high-order bit) : set to 1 by the processor to signal the monitor that the processor execution was not successful. If this bit is the only non-zero bit in the PROC byte, the error will be interpreted by the monitor as a user error.

To each bit set to one upon return corresponds a special action from the monitor ; these actions are described in the ERROR section of the monitor description chapter.

Jumps to the interrupt entry of the processor (INTADD) can be prevented by the processor during execution of critical sections (i.e sections of program that must be executed "at once"). A special byte in the .FC sub-area of the communication area can be set to zero (interrupts disabled condition). If this byte is zero, the processor time-out and the console interrupt conditions will not cause any action to be taken in addition to the setting of the PROC byte third or fourth bit to one ; if the byte is equal to one, a jump to the interrupt address of the processor will occurr ; the value of the byte is checked periodically by the REAL-TIME CLOCK process.

In addition to the bytes described above, the PROC sub-area contains a small routine that takes the following actions successively :

(1) set the PROC byte to the inclusive OR of the old value of the Proc byte with the contents of the (A) register.

(2) jump to the processor interrupt address (INTADD).

This routine is used by the REAL-TIME CLOCK process ; however, it may be used by the processor itself, and the calling sequence is :

```
LDA=  OR mask for the PROC byte
LDX   .CA ( top of the PROC subarea address )
JMP+  7
```

The PROC sub-area physical structure is shown in fig 12.

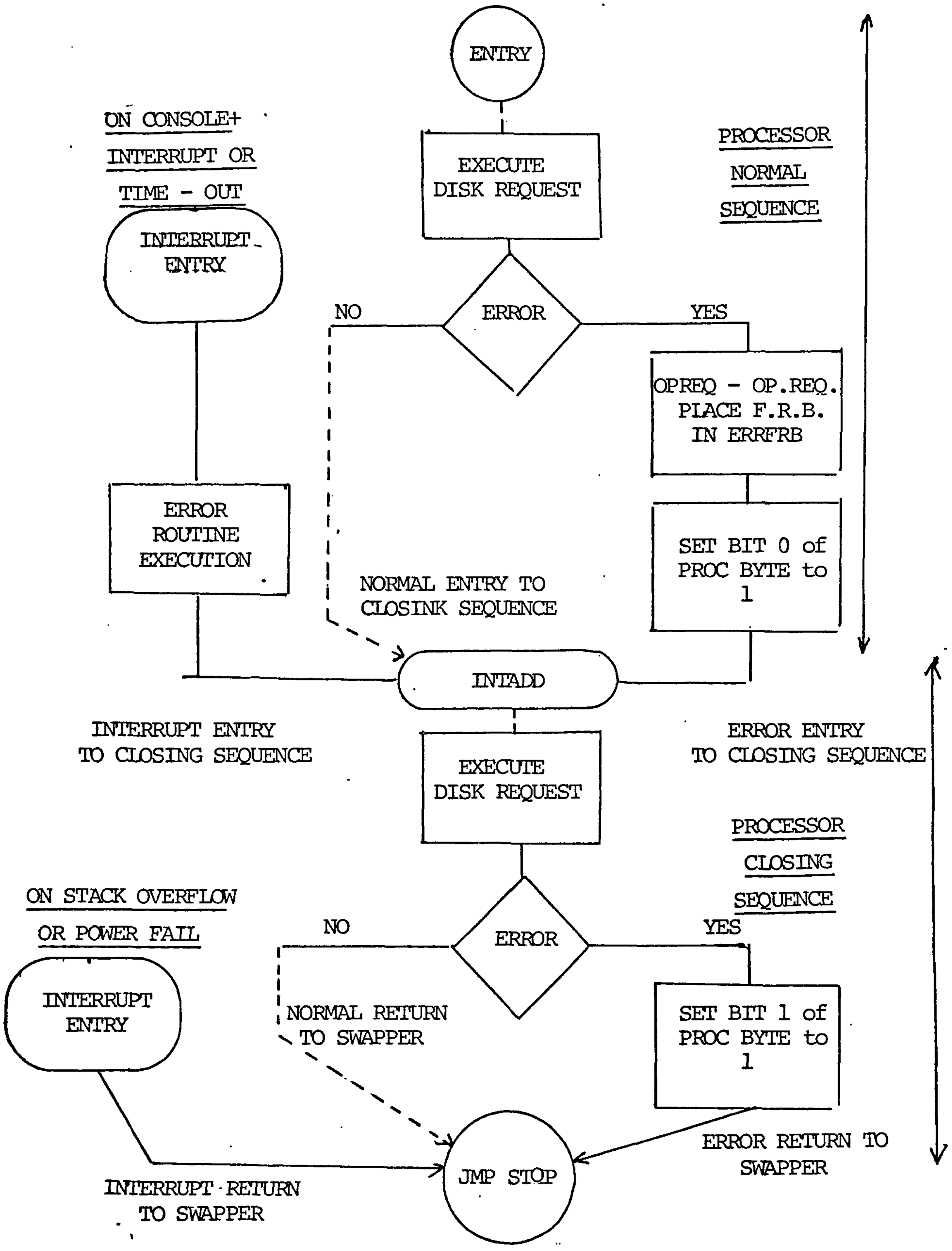

ENTRY
EXECUTE DISK REQUEST
PROCESSOR NORMAL SEQUENCE
ON CONSOLE+ INTERRUPT OR TIME - OUT
INTERRUPT ENTRY
ERROR
NO
YES
OPREQ - OP.REQ. PLACE F.R.B. IN ERRFRB
ERROR ROUTINE EXECUTION
SET BIT 0 of PROC BYTE to 1
NORMAL ENTRY TO CLOSINK SEQUENCE
INTADD
INTERRUPT ENTRY TO CLOSING SEQUENCE
ERROR ENTRY TO CLOSING SEQUENCE
EXECUTE DISK REQUEST
PROCESSOR CLOSING SEQUENCE
ON STACK OVERFLOW OR POWER FAIL
INTERRUPT ENTRY
ERROR
NO
YES
NORMAL RETURN TO SWAPPER
SET BIT 1 of PROC BYTE to 1
ERROR RETURN TO SWAPPER
JMP STOP
INTERRUPT RETURN TO SWAPPER

Fig 11 : GENERAL SCHEME FOR PROCESSOR EXECUTION

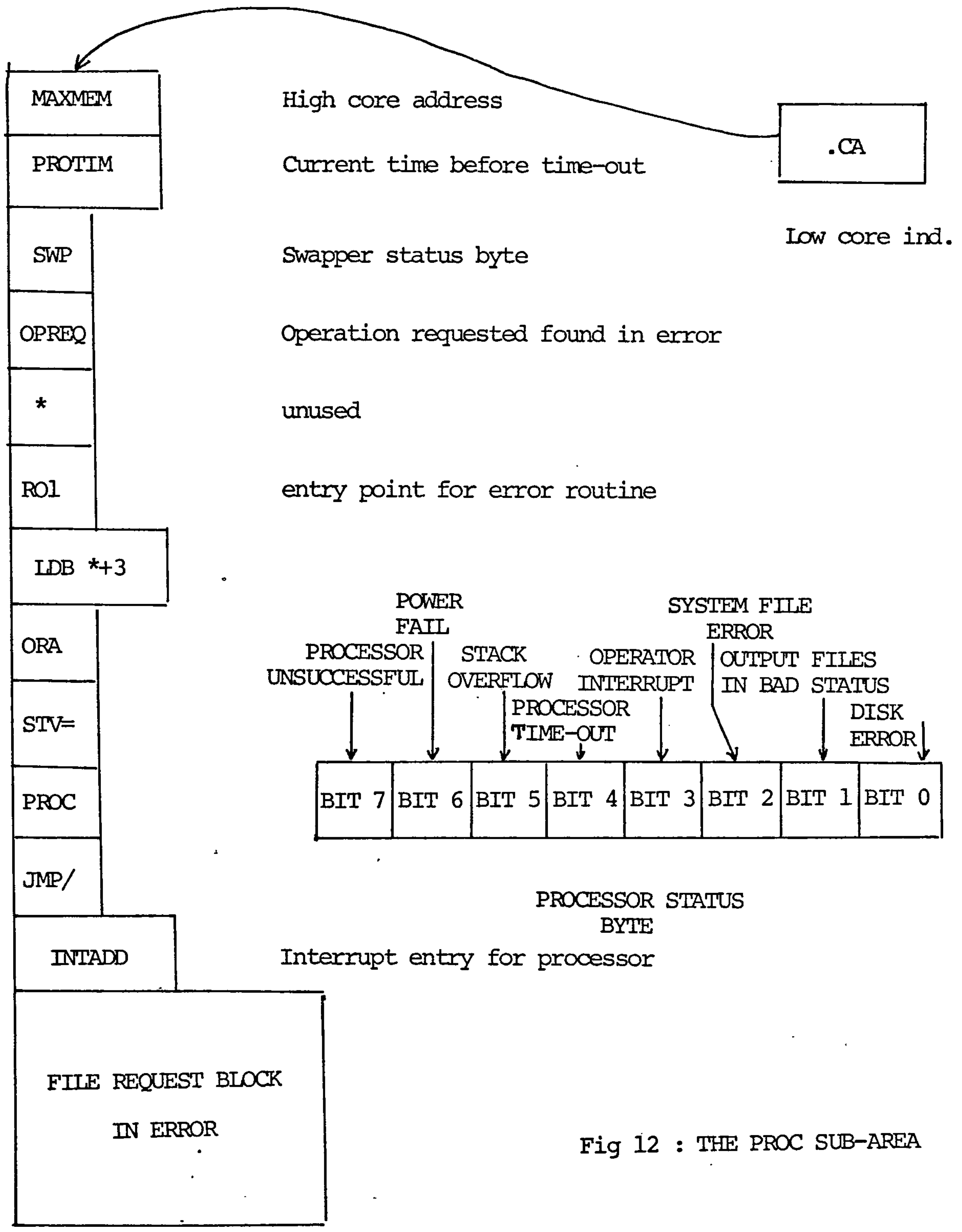


Fig 12 : THE PROC SUB-AREA

3. EFFECTS THROUGH REQUESTS TO OTHER PROCESSES

A. REQUESTS TO THE SYSTEM FILE HANDLER ROUTINE

Processors desiring to execute any I/O upon system files normally do it via a special routine called the SYSTEM FILE HANDLER PROCESS. This routine possess its own data chain, and thus constitutes a particular process in the system ; in this data chain are defined all the commonly used system files, including :

(1) the RUN$ file

(2) the SCR$ file (system scratch file)

(3) the PRO$ file (particular processor file containing the monitor)

(4) the disk directories DIR$(1-2-3-4).

The routine is called via the following sequence :

```
LDX=  File Request Block address
LDA=  6
ADA   .HPROG
STA   *+3
RTJ=  **
```

The INTERNAL NAME of a system file is equal to its external name (filename) .

The output is in the register (A) : A non-zero value of (A) indicates to the processor that the disk request was in error. The REQUEST BLOCK and FILE CONTROL BLOCK in error have been copied on the system teletype by the routine, and the bit # 2 of the PROC byte has been set to 1.

B. REQUESTS TO THE INPUT AND OUTPUT PROCESSES

Processors needing some input from the current system input medium or producing some listing on the current system output medium normally do it by placing requests on the READ (SYST. INPUT case) or PRINT (SYST. OUTPUT case) processes.

A print request is obtained by the following sequence :

```
LDX=  address of message to print
RTJ*  PRINT
```

The message is terminated by a zero byte.

As in the case of the SYSTEM FILE HANDLER process, a non-zero value of the (A) register on output indicates that an error occurred during execution of the process. The RUN byte (see MONITOR) bit 2 is set to one and a message is printed-out on the system teletype ; however, no action is taken upon the PROC byte by the PRINT process.

A read request is obtained by a :

```
RTJ*  READ
```

A positive value of (A) upon return indicates that the card read was a control-card ; a negative value is associated with error cases similar to the PRINT case above.

C. DATA FLOW INPUT AND DATA FLOW OUTPUT FILES

However, we mentioned in the informal presentation in Chapter 3 that the READ, PRINT processes (as well as the system file handler) are part of the "extended resident". Some processors are either too big or require large tables (such as the ASSEMBLER, see [1]. to be kept permanently in memory together with the extended resident ; an additional buffering is done by the monitor in the particular case of these processors.

Two additional files, of names CR$ and PR$ are always temporarily assigned to the current RUN ; these files are dedicated to hold :

(1) data flow input as read from the current input medium (CR$) ;

(2) data flow output as to be printed later on the current output medium (PR$) .

The monitor copies into CR$ all "data" cards (i.e cards following the processor control cards, up to the next control card excluded) by placing requests on the READ process. The processor may then use the standard disk driver (.DR) or the variable length record driver routines to perform the following :

(1) read input from the CR$ file ;

(2) produce listing onto PR$ file ;

Upon return from the processor, the monitor copies the contents of the PR$ file onto the current output medium of the system, using the variable length record driver and placing requests on the PRINT process.

Though clearly inefficient, this scheme can be justified by the following considerations :

(1) memory storage, as described above ;

(2) flexibility : the "data flow input" medium is now defined in the processor data chain ; thus a simple change in the binding of the corresponding item redefines the data flow input for a particular execution of a processor ; for instance, by simple modification of the bin-

ding, the ASSEMBLER will execute on a program stored in any file in the disk file system ; this binding can be done by the monitor upon user request on the processor call control card.

D. RETURN TO THE SWAPPER

The normal termination sequence for a processor is :

```
JMP    STOP
```

STOP is an address in page zero of memory executing a jump to the beginning of the part of the swapper that reloads the monitor. Before reloading the monitor, an additional checking is done that the following conditions are satisfied :

(1) all disk I/Os are completed

(2) the LINK/UNLINK stack is in a correct status.

Recursive calls on the swapper are not possible in the current version of the system. Thus, a processor must manage its own overlays if necessary (see EXECUTE PROCESSOR example).

CHAPTER IX :

THE REAL-TIME CLOCK PROCESS AND THE ERROR INTERRUPT ROUTINES

An " interrupt entry " was defined for each processor in the preceding chapter. A jump to this entry point is executed an the case an error condition is detected. The MONITOR program posseses such an entry point as well. A process activated by the REAL-TIME CLOCK provided by the microprograms is in charge of the treatment of these error conditions. The microprograms are able to detect the following interrupts (in addition to the normal device interrupts) :

(1) console interrupt

(2) power fail and power restart interrupts ;

(3) stack overflow interrupt ;

(4) real-time clock interrupt ;

1. STACK OVERFLOW, POWER FAIL, POWER RESTART

It was decided that the stack overflow and power fail interrupts were to be considered as fatal errors in the system. A jump to the error analysis programs of the monitor is executed, and the operator is able to specify which action to be taken, among the following :

(1) continue execution ; only the last processor is in error ;

(2) abort the current RUN ;

(3) reload the system.

In the case of a power fail or stack overflow, the third of these actions is recommended, since the stack overflow and power fail programs do not guarantee that the statuses of the files is correct.

The power fail program executes the following :

(1) disable the real-time clock ;

(2) reset the power fail bit in the system stack ;

(3) halt ;

The power restart program sets the sixth bit of the PROC byte to one (PROC = X'40') and executes a jump to the STOP routine to reload the monitor if necessary (JMP STOP).

The stack overflow interrupt program sets the fifth bit of the PROC byte to one (PROC = X'20') and executes a jump to the STOP routine as above.

2. CONSOLE INTERRUPT

The only action taken by the console interrupt program is to set the fourth bit of the " interrupt requested " byte to one for ulterior treatment by the real-time clock process (see below) .

3. REAL-TIME CLOCK

The Operating system makes use of the real-time clock provided by the microprograms to execute the following :

(1) update the time couters : two time counters are existing in the system, specifically :

. a count of the RUN time ; the RUN time is decremented by one every second from a value set by the RUN processor. A value of zero of the RUN time causes the RUN time to be reset to a large value and the bit 1 of the RUN byte to be set to one ; after completion of the current processor, the RUN will be aborted by the monitor.

. a count of time for the current processor, as defined in the preceding chapter ; this time is used to detect a loop in execution of a processor ; the initial value set by the monitor does not correspond to the average time of execution of the processor, but rather to a maximum time, after which one can be sure that the processor was not executing normally. Upon a zero value of this time, the time is reset to a large value, and the third bit of the " interrupt requested " byte is set to one (X ' 08').

The above times include CPU time, disk I/O times and input/ output time. They do -nt constitue any representation of the CPU time.

(2) restart the concurrent spooler routines if they need to be restarted (see [1] for specific description of the cases).

(3) check whether a special condition (CONSOLE INTERRUPT, MAX TIME for current processor) occured since the last interrupt by checking the value of the " interrupt requested " byte defined in the .FC sub-area of the communication area. If a non-zero value of the byte is detected, a check upon the " interrupt enabled " byte (in the same sub-area) is done ; if the byte has a zero value, it means it has been set by a processor, currently executing a critical section.

No action is then taken immediately. If the interrupts are enabled, the following steps are executed :

. update each corresponding bit of the PROC byte to the value of the " interrupt requested " byte ;
. zero the " interrupt requested " byte ;
. wait for the queue of disk operations to be empty ;
. jump to the interrupt address of the currently executing program, as defined in the PROC sub-area of the communication area.

This sequence gives a chance to the currently executing process to take appropriate action of recovery.

CHAPTER X :

THE MONITOR PROCESS AND THE CONTROL LANGUAGE

1. FUNCTIONAL DESCRIPTION

Given the general scheme of execution of a processor, the functions of the monitor process can be readily described by the following :

(1) Determine the processor to be called.

(2) Create the processor data chain ; bind swapper chain, processor chain and subchains.

(3) If PR$ is in the processor data chain or any subchain, initialize PR$ to empty ; if CR$ is in the processor data chain or any subchain, initialize CR$ to contain the set of cards separating the current control card from the next, excluded.

(4) Initialize the communication area for processor execution.

(5) Link the swapper chain.

(6) Wait for swapper completion.

(7) Unlink the swapper chain.

(8) Analyse "results" of processor execution through communication area and chains.

(9) If PR$ is in the processor data chain or any subchain, copy the contents of PR$ onto the current system output medium.

(10) Go to step (1).

One execution of the monitor will be defined as a single execution of steps (1) through (10). It involves the execution of a single processor.

The MONITOR program will be examined successively with respects to the following :

(1) Determination of the processor information area.

(2) Creation and binding of the swapper, processor data chains and subchains.

(3) Initialization of the communication area, transmission of arguments to the swapper.

(4) Normal return from processor sequence.

(5) Error conditions and routines.

(6) CR$-PR$ files management.

A sample of the control language will be studied ; the syntactical definitions and error messages are also presented in this chapter.

2. DETERMINATION OF THE PROCESSOR INFORMATION AREA

As mentioned in the general processor description chapter, the information necessary to build the processor chain(s) is held within an area of the RUN$ file called PROCESSOR INFORMATION AREA. The first step for the monitor is thus to establish the correspondence between the PROCESSOR NAME specified on the control card and the sector address of the particular processor information area needed within the RUN$ file.

For each RUN is specified a list of processor names available to the user. This list is kept in a particular area of the RUN$ file called CURRENT PROCESSOR TABLE. A user can dynamically add, modify or delete processors from the current processor table by use of the PROC processor (see PROC processor description chapter).

The current processor table establishes the correspondence between processor identification and sector address of processor information area. It is defined within 3 consecutive sectors of the RUN$ file, and the address of the first sector from the beginning of the RUN$ file is found by the monitor in the RUN sub-area of the communication area. The format of the current processor table is given in fig`13.

A processor identification, as defined in the current processor table, is composed of the following elements :

(1) a NAME field (up to 8 letters or digits, eventually completed to 8 by trailing blanks) ;

(2) an OPTION field (one single letter or blank).

The processor identification characterizes the processor within the current processor table.

A "control card" in the system consists in any number of physical records on the current system input medium. The first character of the first of these records need to be a " character, in order for the monitor to be able to separate control cards from data cards ; as a consequence, no data card may begin by a " character ; the same consideration holds for the second, third,... records of a control card, if several physical records are needed.

The programmer must specify both the NAME field and the OPTION field for a correct processor call to be executed. The processor NAME must immediately follow the " character on the control card ; it may be abbreviated to the smallest subset of characters that determine the pro-

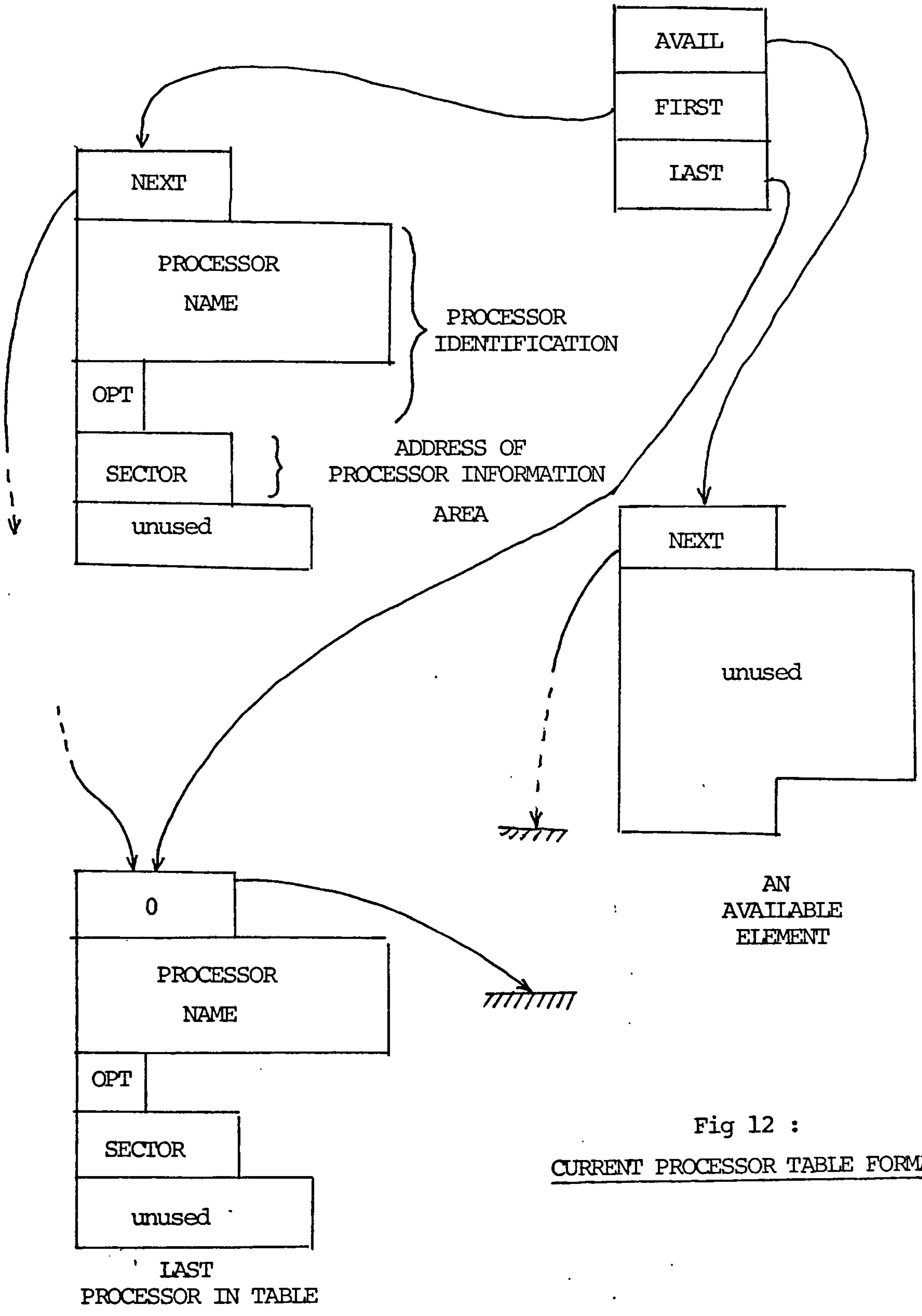


Fig 12 :

CURRENT PROCESSOR TABLE FORMAT

cessor name completely within the names in the current processor table. In the case of a non-blank option, a comma must immediately follow the NAME field, itself immediately followed by the letter constituting the OPTION field ; in the case of a blank option, the comma may be omitted. In all cases, AT LEAST ONE BLANK must follow the processor identification sequence.

An exception occurs in the case of the processor "FIN terminating a RUN. The processor "FIN is not defined within the current processor table. The sector number of the processor information area of the "FIN processor is a constant in the system, and is defined in page zero of memory. A checking for the sequence of character "FIN is done by the monitor before any search in the current processor table. Thus, any control card beginning by the character sequence "FIN will be interpreted as a call to the "FIN processor.

3. CREATION AND BINDING OF SWAPPER AND PROCESSOR DATA CHAINS

A. PROCESSOR INFORMATION AREA

The processor identification field on the control card had the only function of enabling the monitor to load into memory the processor information area defining a particular processor in the system.

A processor information area is created on the disk via use of the PROC processor (see corresponding chapter). "Permanent" processor information areas correspond to standard use of the processors in the system and are constantly defined within the RUN$ file ; some "temporary" processor information areas may be tailored by a user .

The processor information area is described in fig. 14. It contains the following basic elements :

(1) The definition of the processor program absolute image. This includes a file-name and a sector number ; file name and sector number define an address in the disk file system where the dictionary of the processor program's absolute image can be found ; the file-name must either be PRO$ (particular system file containing the monitor) or be defined in the CURRENT ASSIGNED FILE LIST as a user file.

(2) The definition of the processor default data chain. For each item in the processor default data chain are defined the following :

. INTERNAL NAME of the item ;

. TYPE of the item ;

. ACTION BITS of the item ;

. # DEFAULT OBJECTS associated with the item ;

. MINIMUM expected number of objects to be bound to the item ;

. MAXIMUM expected number of objects to be bound to the item ;

. DEFAULT BINDING of the item.

The default binding may be empty (in this case # DEFAULT OBJECTS = 0) ; it may be outside of the MIN/MAX range. The format of an item in the default processor data chain is compatible with the general format of an item, as defined in the Data Chain Implementation chapter ; MIN and MAX (Minimum and Maximum expected number of objects to be bound to the item) are appended at the end of the item definition.

Three additional bytes defining the Default Processor Data Chain (D.P.D.C.) as an object (of the swapper chain via an item of type item) are to be found within the Processor Information Area (P.I.A.). They are the minimum, maximum and access bytes relative to the D.P.D.C. as an object of the swapper chain.

An additional difficulty comes from the fact that pointers in data chains are absolute ; it was thus decided that all data chains held on the disk were to be stored as if the area were to be loaded into page zero of core (Note : the same convention was adopted for all structures involving absolute pointers if they were to be stored on the disk ; this includes in particular DIRECTORIES, the CURRENT ASSIGNED FILE LIST and the CURRENT PROCESSOR TABLE) . Consequently, a traversing of data chains is needed (to update pointers) each time a disk operation occurs. In addition, the FILE-ID zero on the disk is interpreted by the monitor as the file-id of the current RUN, as defined in the RUN sub-area of the communication area.

(3) The definition of the MEMORY REQUIREMENTS of the processor.

A minimum memory requirement (interval MAXMIN-MINMAX) and a maximum memory requirement (interval MINMIN-MAXMAX) are defined for each processor in its Processor Information Area. The monitor makes sure that the MINIMUM requirements are satisfied, and will allocate the maximum memory available within the MAXIMUM required. The upper memory available to approcessor may change from processor call to processor call, and is placed by the monitor in the PROC sub-area of the communication area for the processor information.

(4) A definition of a special entry (INTERRUPT ENTRY) for the processor. This entry is used by the REAL-TIME CLOCK process in the case where an abnormal interrupt is detected (see REAL-TIME CLOCK PROCESS chapter).

(5) A PROMPT message to be printed by the monitor upon the processor call ; this message consists of ASCII characters, and is terminated by a zero byte.

(6) Some additional "PROCESSOR CONSTANTS" to be used by the monitor.

- . A PROCESSOR NUMBER ; as shall be seen in the description of the PROC processor, a user may change the name of a processor in the system ; the processor number is thus the only way for the monitor to tell the console operator that a given processor was found in error.
- . A "CONCURRENT I/O ACCEPTANCE INDICATOR". This byte is SET if the SPOOLING process is allowed to run during the processor execution, reset otherwise.
- . A PROCESSOR LEVEL. To each processor is associated a level, which is attempting to estimate the degree of importance that a failure of this processor has with respects to the current ! user of the system ; the current values of the LEVEL are :

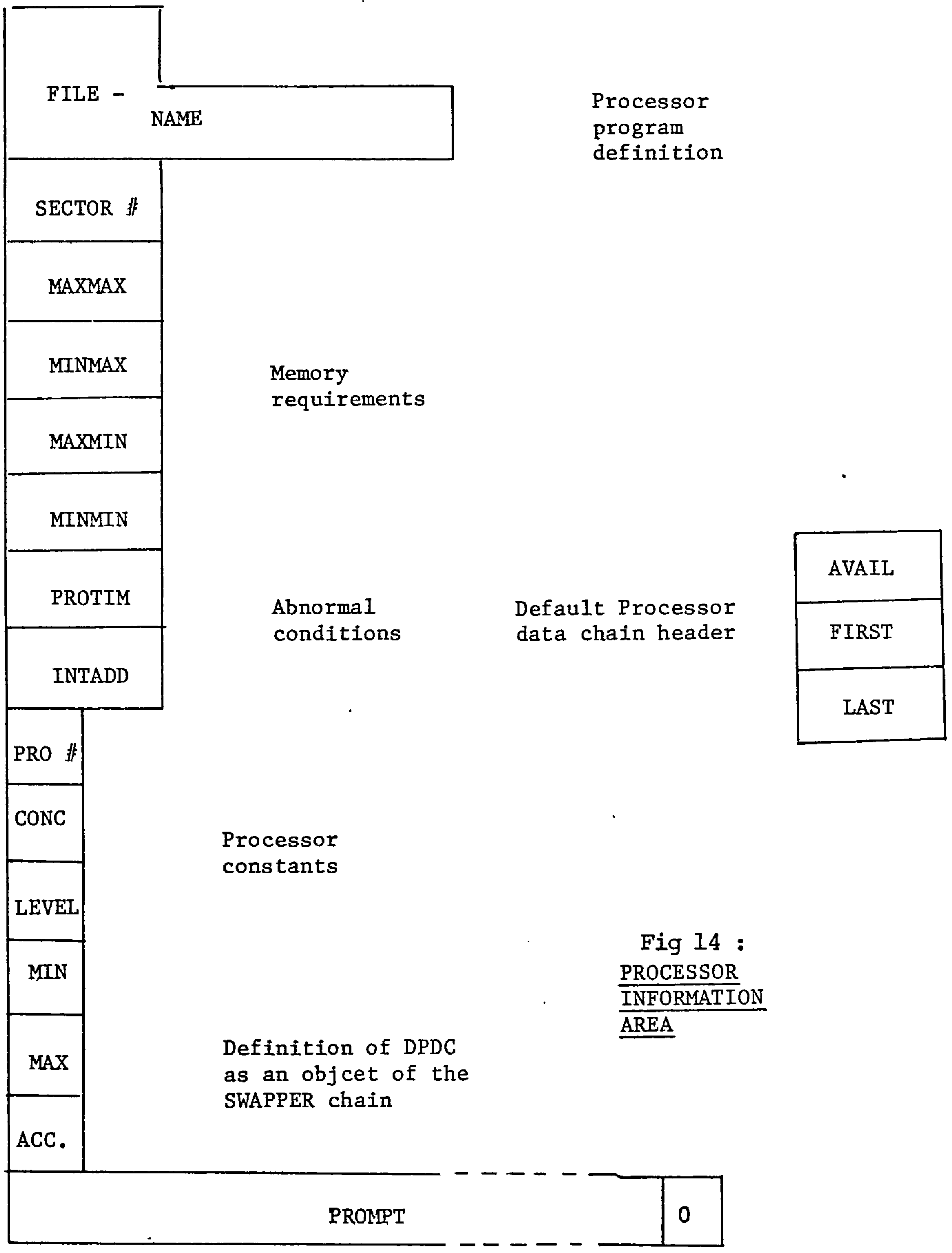
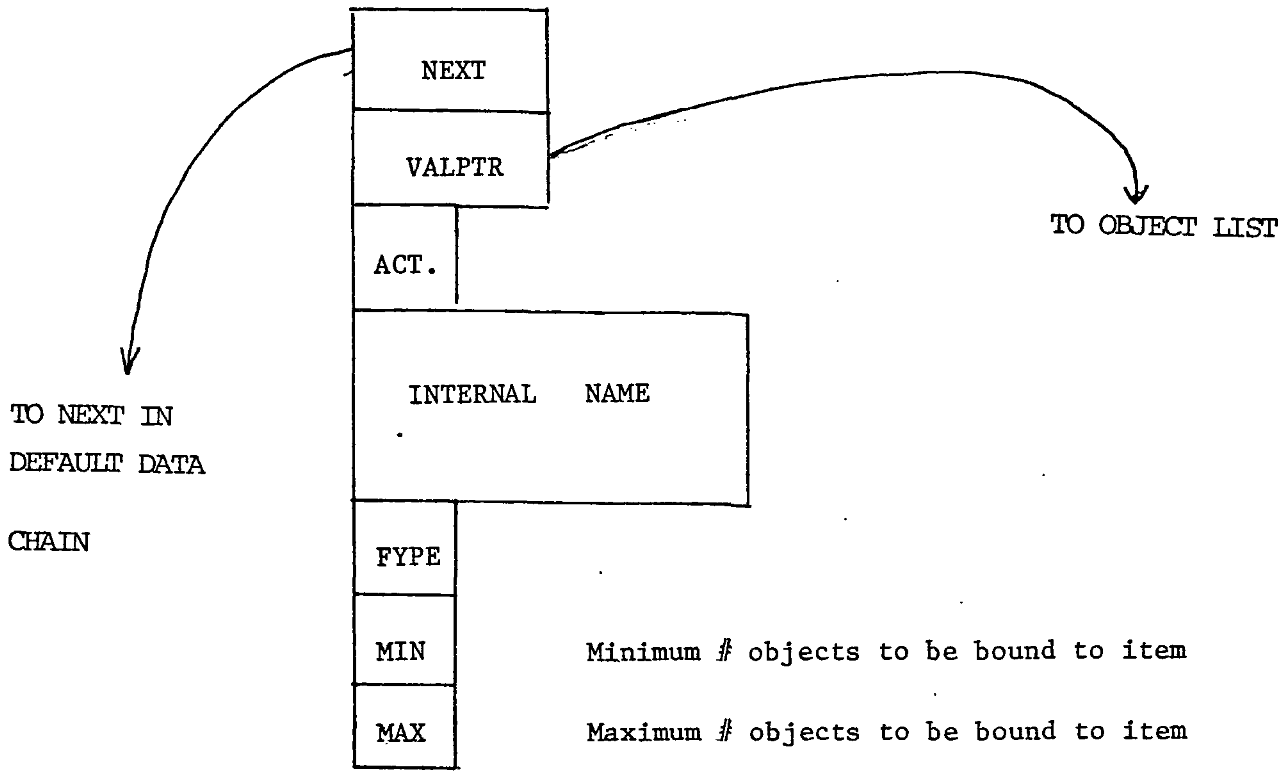


Fig 14 :
PROCESSOR INFORMATION AREA


NEXT
VALPTR
TO OBJECT LIST
ACT.
INTERNAL NAME
TO NEXT IN DEFAULT DATA CHAIN
FYPE
MIN
Minimum # objects to be bound to item
MAX
Maximum # objects to be bound to item

<u>AN ITEM IN THE DEFAULT PROCESSOR DATA CHAIN</u>

LEVEL = -1 : a processor error does not abort the RUN

LEVEL = +1 : a processor error always aborts the RUN

LEVEL = 0 : a processor error aborts the RUN if the system is in BATCH configuration, does not abort the RUN otherwise.

Examples of processor default data chains can be found in the description of each processor, both in this thesis and in [1].

B. SPECIFICATION FIELD ON THE CONTROL CARD

The control card specification field enables the user to specify the Processor Data Chain for the processor to be called. Reference to a particular item in the processor data chain can be done either by explicit mention of the internal name of the item (EXPLICIT field) or by position in the control card (IMPLICIT field). Recursion for the treatment of subchains of the processor data chain is obtained by enclosing the subchain within parenthesis. Some processors may call for the creation of user-defined items which are not in the Default processor data chain.

a) The control card specification field consists in a number of fields (from zero to 255) separated by commas, terminated by a closing delimiter (" , ; or)). A control card specification field may spread over any number of physical records on the input medium ; it must be separated from the processor identification field by at least one blank. The closing delimiter terminating the specification field also terminates the control card ; thus, no closing delimiter is needed if there is no data card following the control card (the " character at the beginning of the next control card will terminate the current one). However, since it is needed if any data card(s) is present, systematic usage of the termination character can be considered good practice.

b) Each field corresponds EXPLICITLY or IMPLICITLY to a single item in the processor data chain. The correspondence is EXPLICIT if mention is done of the internal name on the control card. If no internal name is specified in the Nth field of the control card, the programmer is assumed to refer to the Nth parameter in the Default Processor Sata Chain, and the correspondence is said to be IMPLICIT. If the internal name specified (EXPLICITLY) is not found in the default processor data chain, the user is assumed to declare a new item (USER-DEFINED ITEM) in the chain. Whether the parameter be defined EXPLICITLY, IMPLICITLY or be a USER-DEFINED ITEM, the programmer may (or may not) specify a binding for the item by means of an object list.

```
field  :=  implicit field     explicit field
explicit field  :=  item definition  =  object list
implicit field  :=  object list
```

Note : the monitor, in order to be able to make the distinction between explicit or implicit field, "looks ahead" for a '=' character to the next ',",'(' or closing delimiter WITHIN THE LIMITS OF THE CURRENT PHYSICAL RECORD ; this causes a restriction to be done to the general rule of free formatting of the control-card, since the item definition and the '=' character must be on the same physical record, inorder for the field to be interpreted correctly ; however, this seems to be no real restriction in practice, since programmers tend to group the internal name and '=' character on the same physical record anyway.

c) ITEM DEFINITION

When an implicit reference is made of an item (by position on the control card), the TYPE, ACCESS, MIN and MAX of the item are defined to be the default TYPE,ACCESS,MIN and MAX bytes of the relevant item in the DPDC. When the reference is explicit, the same rule normally applies ; for a user-defined item (i.e. an item that is not in the DPDC), the defaults for the TYPE, ACCESS, MIN and MAX are defined to be the following :

(1) blank type (any file in the disk file system) ;

(2) X'11' access (open to READ and BIND only) ;

(3) MIN = 0 (minimum number of objects expected = 0);

(4) MAX = X'FF' (maximum number of objects expected = 255) .

However, for explicit reference to an item, and in the case of user-defined items, the user may override the defaults by explicit mention of all of the TYPE, ACCESS, MIN and MAX, or by mentioning only some of them.

A user-defined type may be mentioned explicitly by appending a '-' character to the internal name of the item ; the type (one of the characters 'D',' ','S','R','A','I','H','C','B' or 'F'): Must immediately follow the '-' character without intervening blanks. However, if the item referenced is defined in the default processor data chain, the user-defined type must match the type of the corresponding item in the DPDC, otherwise an error results and the processor will not be executed.

The MAX,MIN and ACCESS bytes may be redefined by the user according to the following rules :

```
item definition  :=  internal name   item specificetion
item specification  := [ type ] [ nummaxacc ]
type  := -  S R A B C I F blank
nummaxacc  := ( [ max ][,[ min ][,[ access ]]] )
```

The min , max and access are considered by the monitor as objects of type 'H' and must be explicited accordingly on the control card (see objects of type 'H' section later in this chapter).

If any of these is explicitly mentioned, the following rules hold :

(1) the MIN of the resulting item will be in all cases taken to be equal to the maximum of the user-defined MIN and of the default MIN ;

(2) the MAX of the resulting item will be in all cases taken to be equal to the minimum of the user-defined MAX and of the default MAX ;

(3) if the item is a user-defined item, the resulting access will be obtained by applying the following rules :

. a temporary access is created by taking the user-defined item ;

. a logical AND is taken between this temporary access and the ACCESS of the data chain in which the item is defined, considered as an object of a parent chain ;

(4) if the item exists in the default processor data chain, the resulting item will be obtained by application of the following :

. the four lower bits are temporarily taken to be equal to their DEFAULT definition (i.e. their definition in the chain) ;

. the fifth bit (BIND access, allowing the user to redefine the default binding of the item) is temporarily taken to be equal to the logical AND of the fifth but in the data chain (default value) and of the user definition for the fifth bit of the access byte ;

. finally, the resulting item is obtained by applying the same rule as above in case (3), specifically a logical AND is taken between the temporary access and the access byte of the data chain in which the item is defined.

d) OBJECT LIST

The object list enables the user to redefine the binding of the current item. The list of objects is specified as a succession of subfields separated by commas, enclosed within parentheses ; if there is only one object in the list, the parentheses may be left out, except in the case of an item of type item ('D' type).

If no object is specified, the binding is assumed to be equal to the default binding, i.e. :

(1) the binding defined in the Default Processor Data Chain for the items already defined in the DPDC ;

(2) the empty binding for user-defined items.

<object list> := <empty list> | <single object list> |
<multiple object list>

<empty list> := <nil>

<single object list> := <object>

<multiple object list> := (<dobject> $\left\{ , \text{<dobject>} \right\}_{0}^{\infty}$)

When one of the subfields is left out, the corresponding default object from the DPDC is assumed, if any ; if none, an error results and the processor is not executed.

<dobject> := <nil> | <object>

If the number of subfields explicitly or implicitly specified within an object list is greater than the MAX of the corresponding item, an error results and the processor is not executed.

If the number of subfields specified within an object list is smaller than the MIN of the corresponding item, the monitor will attempt to complete with default objects, until the number of objects is equal to MIN ; if the number of default objects is strictly less than MIN, an error will result and the processor will not be executed.

For each subfield which is explicitly specified, one of three conditions must hold :

(1) the ACCESS fifth bit (bit # 4) of the relevant item is set ;

(2) no default binding is specified for the subfield in the DPDC ;

(3) the default object for the subfield is equal to the object specified by the user.

e) COMPLETION CONDITIONS

Not all items in a default processor data chain need to be explicitly or implicitly mentioned by the user on its control card. If an item of the DPDC is left unspecified, it will be appended to the (memory) processor data chain at the end of the processing, and its binding will be assumed to be equal to the default binding, as specified in the DPDC. This remark holds for subchains of the PDC as well.

This can be considered as an exception to the completion rule for objects as explained in the preceding paragraph ; in the case where the item . is of type item, all itemsobjects in the DPDC will be present in the final chain ; in the case of items of other types, the completion rulw will hold only until the final number of objects reaches the MIN value.

f) OBJECTS

The objects specified in the value list must have the type expected from the item definition. The different object types are examined below successively.

g) FILE OBJECTS (TYPES 'S','R','A' or ' ')

A file object is defined on the control card by a FILE-NAME and a FILE-TYPE.

The file-name is constituted of two parts : the FILE-ID part and the USERNAME part. The file-id part is characteristic of a user or a group of users in the system ; it is constituted of from 1 to 4 hexadecimal characters ; if the file-id part is left out of the control card, it is assumed to be equal to the CUURENT USER FILE-ID, as defined within the RUN sub-area of the communication area. The file-id part, if explicited on the control-card, is separated from the username part by the special character : '*' ; since the monitor "looks forward" for a '*' in order to know if the file-id is explicited, the file-id and username parts must belong to the same physical record ; however, any number of blanks may be left before and after the '*' character. The username part is constituted of up to 8 letters or '$', eventually completed on the right by trailing blanks.

The FILE-TYPE may be explicitly mentioned on the control card, and is this case must immediately follow the character '-', itself appended to the end of the file-name. If the type is explicit, it must be equal to the type of the corresponding item if the type is one of the three types 'S','R','A' ; if the type is not explicited, it is assumed to be equal to the type of the corresponding item if the type of the item is 'S', 'R' or 'A' ; an error results in all the other cases.

The complete identification of a file in the disk file system consists in the file-id, username and file-type. Two files with the same file-id and username but different types (such as CR$-S and CR$-A) may coexist in the system. A demonstration of the actual way a file-name is stored internally by the system is shown in fig. 15.

The monitor is in charge of constructing the file control block for the file required. The file must be defined in the current assigned file list, containing the list of the files currently assigned to the RUN.

The current assigned file list format is described in fig.116. the informations relevant to the construction of the FILE CONTROL BLOCK include the following items :

(1) file-name ;

(2) current END-OF-FILE ;

(3) disk/platter byte ;

(4) offset to first allocation of file within platter ;

These elements are to be placed in the file control block by the monitor for later use by the disk driver routine (.DR).

In addition to these,elements, the "STATUS 1" byte contains 8 bits defining the type.of access allowed to a user upon the file concerned ; the four lower bits define the possible ways in which a user with the same file-id as the file-id of the file concerned may access the file (READ, WRITE, PROTECT, UNPROTECT accesses in this order from bit 0 [lower bit]); the four high-order bits apply to all the other users in the system.

If the file-id of the current user is the same as the particular file-id, the four lower bits areaconsidered ; otherwise, the four higher bits are considered. If, for each of the four lower bits in the access byte of the current item which is set to one, the corresponding bit among the four chosen is also set to one, the required access upon the file is acknowledged ; otherwise, an error results and the processor call is ignored.

The "STATUS 2" byte of the current assigned file list contains in particular the following bits :

(1) a "in use" bit (bit 1), set to 1 when the file is currently in use

(2) a 'file possibly in bad status" bit (bit 6), set to one when the contents of the file are doubtful ;

(3) a "file in hardware error" bit, set to one when a disk error occured when reading, writing, protecting or unprotecting the file ;

When the file control block is created by the monitor, the "in use" bit of the status 2 is set. When the file control block is destroyed (upon return from the processor) , the monitor resets it. Upon error returns from the processor, the monitor may set the "file possibly in bad status" and "file in hardware error" bits of the status 2.

Normally, when a file is requested by a processor, the "in use" bit must be reset ; however, a file may be requested by different subchains of the processor data chain. In this case, the file control block is not created, but the VAL pointer of the new item is set to point onto the file control block already created. A specific check is made that the file is not requested from two different items belonging to the same data chain (see fig.3, case d).

Finally, the binding of a file object by the monitor can be explained by the following steps :

(1) check that the file requested is in the current assigned file list ; return in error otherwise.

(2) check that the access required (in the item definition) is acknowledged by the definition of the file in the current assigned file list ; return in error otherwise ;

(3) look-up in all chains and subchains created to this point whether a file control block was already created with the same file-name.

- If found, check whether a reference to the FCB is done from an item in the same data chain as the current item ; return in error if so ; otherwise simply set the VAL pointer of the current item to point onto the FCB found ;
- If not found, check that the file is not in use (i.e. that the "in use" bit in the assigned file list definition for the file is not set) ; return in error if the file is found already in use ; otherwise set the file to "in use" and create the File Control Block in memory from the definition of the file in the Current Assigned File List.

h) ITEM OBJECTS (TYPE 'D')

Chain objects enable a processor to define subprocesses. On the control card, it is simply a recursion over the syntactical definition of the " specification field " within parentheses ; the maximum recursion depth has been arbitrarily set to 5, which is more than sufficient for all practical purposes.

As noted before, here the parentheses cannot be left-out, for reasons of ambiguity of the control language.

The access of an item of type item is used as a mask for all items in the dependant chain ; this is used in the system to prevent any subprocess from having more power than its parent.

Another exception has already been noted for the case of items of type item : in the case where some items of the dependant chain are left unspecified, all items in the chain of the DPDC corresponding to the dependant chain are appended (as explained under e) of this paragraph), instead of only those necessary to complete the number of objects to MIN as in the case of items of other types (see b) of the present paragraph).

i) MEMORY-BOUND OBJECTS (TYPES 'I','B','H','C','F')

Objects of type 'I' correspond to full-word integer values, and are thus held in two bytes of core ; on the control-card, they can be written in one of two forms :

(1) decimal form (normal signed or unsigned integer value) ;

(2) hexadecimal form, format compatible with the ASSEMBLER hexadecimal form .

Objects of type 'H' correspond to half-word integer values, and are thus held in one single byte of core ; on the control-card, they can be written in one of three forms :

(1) decimal form, as above;

(2) hexadecimal form, as above;

(3) character form, format compatible with the ASSEMBLER character format; the single quote character has been added, and may be written as two consecutive quotes.

Objects of type 'C' correspond only to characters ; on the control card, they must be written in the character form specified above ; the character 'C' preceding the quote may be left out.

Objects of type 'B' are meant to hold boolean values ; they are, however, held in a whole byte of core as a binary 0 or 1 ; on the control card, they can be written either as a 0 (resp : 1) or as one of the words NO or YES.

Objects of type 'F' are file-names considered by the system as memory-bound objects. No file control block is created, no search in the current assigned file list is done by the monitor. They cannot be used to actually execute disk I/Os on the file, but rather seek to transmit a file-name as a simple parameter (see example of use in the "FILE processor, in [1]) ; they are stored in the packed 8-bytes version compatible with the format of file-names in the file control block ; on the control-card, they are written exactly as an object for an item of file type ; if no explicit type is mentioned on the control card for the file, the type is implicitly assumed to be 'S' ; if a type is explicited, it must be equal to one of the file-types 'S', 'R' or 'A'.

j) MULTIPLE CONSTRUCTS

A writing convention was designed to simplify the writing of multiple object lists in the following cases :

(1) objects of type 'C' or objects of type 'H' written in character format ;

(2) objects of type 'H' or 'I' written in hexadecimal form.

The "multiple construct" :

'MESSAGE : '

will be interpreted by the monitor as the more complex :

```
( 'M','E','S','S','A','G','E',' ',':',' ' )
```

If the item is of type 'H', the following expression :

```
X'01011'
```

will be interpreted as if the following had been written :

```
( x'01',X'01',X'1' ) or ( 1,1,1 )
```

If the item had been of type 'I', the previous expression would have been interpreted by tee monitor as the following :

```
( X'0101',X'1' ) or ( 257, 256 )
```

Note that the result in both cases would have been radically different if one had written X'1011' instead of the above.

These multiple expressions cannot spread over several physical records ; however, this is not a limitation since such constructs can be found within multiple object lists ; to split the above character formation, one could have written for instance :

```
( 'MESSAGE ' ,
  ': ' ).
```

4. ALLOCATION OF PROCESSOR DATA CHAINS

Processor data chains are allocated by the monitor in high core DOWNWARDS. When a new item is allocated, it is allocated the first empty storage below the items already allocated ; a consequence ot this allocation scheme is that successive objects in an object list are to be found by negative offsets from the first one (the actual value of the offset depends on the item TYPE according to the memory requirements for each object).

The higher address for storage of the processor data chains depends on whether or not the processor allows the spooling process to be running during its execution, as defined by the "concurrent I/O indicator" byte defined in the processor information area. If the byte is reset, the spooling process is put in a wait condition by the monitor before the beginning of the chain allocation phase ; the image of the spooling process area is temporarily stored in an area of the SCR$ file. The processor data chains may then be allocated from the top of memory downwards. Otherwise, they are allocated from the bottom of the spooling process area.

In allocating the data chains, the monitor makes sure that the MINIMUM requirements of the processor are satisfied (as defined by the variable MINMAX of the processor information area) ; otherwise, an error will result, and the processor will not be executed.

Adiitional memory constraints for the allocation process are the minimum requirements of the monitor itself, as described in section 7.

5 INITIALIZATION OF THE COMMUNICATION AREA AND ARGUMENTS FOR SWAPPER

Additional information for transmission of information to the processor and to the swapper include the following :

(1) printing of the prompt message included in the processor information area (a bit enables a user to prevent the printing of this prompt, as well as the printing of control cards ; it can be reset and set via calls to the "IO processor, see [1]).

(2) initialization of various parts of the communication area, including the following :

  . reset PROC and SWP bytes (in the PROC sub-area) ;

  . initialize PROTIM and INTADD (in the PROC subarea) to the corresponding values defined within the processor information area ;

  . set the MAXMEM location in the PROC sub-area to the maximum address available to the processor, as resulting from the data chain allocation process explained in the preceding paragraph.

  . set the "processor number" byte in the .FC sub-section of the communication area to be equal to the corresponding byte defined within the processor information area.

(3) transmission or the arguments necessary to correct swapper execution . The File Control Block of the file where the processor program absolute image is to be found is created in low core within the swapper data chain ; the name of the file is found in the processor information area ; the name may be FFFF*PRO$, in which

case the file control block is found within the chain of the MASTER process (see general processor description chapter, section 3.A) ; otherwise, the file must be defined within the current assigned file list, and the file control block is created by following the steps specified in section 3.B) of this chapter.

The sector number within the file is to be found in the processor information area, as well as the low core address for the swap-out process (= MINMIN).

The high core address for the swap-out is computed from both :

. the memory requirements of the processor and
. the first address not used for storage of the processor data chains, as resulting from the allocation process.

6 NORMAL RETURN FROM PROCESSOR SEQUENCE

The steps followed by the monitor after a normal execution of a processor include the following :

(1) reset the communication area relevant objects, test for errors ;
(2) copy the data flow output from PR$ onto the current system output medium, if PR$ is present in any of the processor chains ;
(3) update the current assigned file list ;
(4) if necessary, reload the spooling process area from the scratch file ; if necessary, enable the spooling process.

Step (1) : the communication area (PROC subsection) is interrogated with respects to processor errors and swapper errors ; the PROC and SWP bytes are reset ; the INTADD is set to point on the monitor interrupt address ; the PROTIM location is reset to the monitor time-out condition ; the interrupts are enabled ; the processor number byte in the .FC sub-area is reset to zero (MONITOR IN).

STEP 3 : the current assigned file list, both in memory and on the disk, is updated as follows : each file in the current assigned file list is tested for the "in use" condition ; if the file is found to be in use, a search for the file control block corresponding to the file definition is done through the swapper and processor chains ; if the file is found, the "in use" bit is reset and the END-OF-FILE is updated to the current value of the END-OF-FILE in the File Control Block.

7 BUFFERING AND REQUESTS UPON OTHER PROCESSES

The monitor executes requests upon the following processes :

(1) the SYSTEM FILE HANDLER ROUTINE (MASTER process) in order to execute the following disk operations :

- load the current processor table in memory ;
- load the processor information area in memory ;
- load the current assigned file list in memory ;
- store the current assigned file list on the disk.

The last of these operations is normally done twice per execution of the monitor, once before the processor execution, and once after return from the processor.

(2) the READ process, in order to perform the following operations :

- read the successive physical records constituting the control card ;
- copy the data flow output, if any ;
- skip to the next control card, in error cases ;

(3) the PRINT process, in order to perform the following operations :

- print the control cards images (unless specific user request, see "IO processor, in [1]) ;
- print the procesoor prompt (same remark as above) ;
- copy the data flow output from PR$, if any ;
- print error messages, if any ;

The formatting of the two dedicated files PR$ and CR$ is done by records ; the monitor is thus able to use the variable length record driver routines in order to execute the necessary COPY operations.

After each cycle of execution of the monitor, both of these files are reset by the following operations :

(1) write an end-of-file mark onto the first two bytes of the first sector of the file (sector zero) ;

(2) reset the END-OF-FILE location to zero in the current assigned file list definition of the file ;

(3) return to the available list all allocations of the file except for the first allocation.

The monitor makes use of a single buffer for all disk operations. The buffer is successively overlayed by the following structures :

(1) the current processor table ;

(2) the processor information area ;

(3) the buffer necessary to the eventual COPY operation of the input data flow ;

(4) the buffer necessary to the eventual COPY operation of the output data flow ;

In addition, the buffer holds the error messages loaded from the PRO$ file, if loading them ever becomes necessary.

The buffer is located in high core, above all the monitor programs. The buffer size is necessarily 3 pages during the (1) and (2) stages ; however, the variable size structure of the processor information area implies that not all the buffer may actually be used for the storage of pertinent information ; the first unused location is defined by the first two bytes of the processor information area (see fig. 14) . Thus, the remaining of the buffer may be allocated by the monitor to the storage of the processor data chains ; however, at least one page must be kept to meet the minimum buffer size requirements from the copy operations. the actual buffer size during the stage 4 depends on whether the system is in BATCH configuration ; the buffer size during the stage (3) depends upon the size of the processor data chain allocated and upon whether the system is in batch configuration or not.

As a consequence of both the section 4 considerations and the above discussion, the monitor will allocate the processor data chain (from the top of memory downwards) until one of the three following conditions does not hold :

(1) the minimum processor memory requirements are not met ;

(2) the monitor buffer size becomes less than one single page of core ;

(3) the processor data chains begin to overwrite the processor information area.

If one of these conditions does not hold, an error results and the processor call is ignored.

8.ERROR CASES AND ERROR ROUTINES. ABORT CONDITIONS

The following will be examined successively :

(1) Errors during control card processing ;

(2) Errors occurring during disk transfers ;

(3) Swapper errors ;

(4) Processor errors ;

(5) Run abort conditions and routines ;

A. ERRORS DURING CONTROL CARD PROCESSING

A variety of user errors may occur during control-card processing. A list of the specific errors can be found in a later section of the present chapter.

The error messages are located within a particular area of the PRO$ file ; the sector address is defined in low core. All errors result in a non-execution of the current processor call ; the actual penalty may include abortion of the current RUN if the user is in BATCH configuration. Otherwise, the cards are skipped until the next control card is met, and normal execution continues from this point.

An indication is given to the user of where on the control-card the monitor was able to detect the error ; however, this is simply an indication that may well not reveal where the true mistake was.

B. ERRORS DURING DISK TRANSFERS

Any error occurring during a disk transfer will cause a jump to the RUN byte analysis routine, and the RUN will normally be aborted.

C. SWAPPER ERRORS

Any swapper error will cause the printing of the message :

'SWAPPER ERROR :'

on the system teletype ; following this message, the SWP byte will be dumped ; the file request block and the file control block relative to the disk operation found in error will be dumped on the system teletype. The PROC byte high order bit (BIT 7) will be set to 1, as well as the RUN byte third bit (BIT 2) ; thus, any swapper error causes abortion of the current RUN.

C. PROCESSOR ERROR ANALYSIS

In the case of a non-zero PROC byte upon return from processor execution, the following actions are taken by the monitor in this order :

(1) if the seven low-order bits of the PROC byte are zero, GO TO step (11) ;

(2) print upon the system teletype the following message :

'ABNORMAL RETURN FROM PROCESSOR : '

followed by the contents of the PRONUM bytes of the .FC sub-area of the communication area (processor number) ; print :

'PROCESSOR STATUS BYTE : '

upon the system teletype, followed bythe contents of the PROC byte ; print upon the current system output medium (i.e. for user information) the following message :

'SYSTEM OR DISK ERROR'

(3) if the BIT 6 of the PROC byte is set (POWER FAIL), print :

'POWER FAIL'

on the system teletype ; set the bit 3 of the RUN byte to one ; set the bit 1 of the PROC byte to 1.

(4) if the BIT 5 of the PROC byte is set (STACK OVERFLOW), print :

'STACK OVERFLOW'

on the system teletype ; set the bit 3 of the RUN byte to one ; set the bit 1 of the PROC byte to one.

(5) if the bit 3 of the PROC byte is set, print :

'TIME-OUT'

on the system teletype ; set the bit 3 of the RUN byte to one.

(6) if the bit 4 of the PROC byte is set (CONSOLE INTERRUPT), print :

'OP. INT.'

on the system teletype ; if the LEVEL of the processor is positive, or if the LEVEL is equal to zero and the system is in batch configuration, set the bit 4 of the RUN byte to one.

(7) if the bit 2 of the PROC byte is set, set the bit 2 and the bit 0 of the RUN byte to one.

(8) if the bit 0 of the PROC byte is set, print the file request block and the operation requested byte of the disk operation found in error on the system teletype ; these are found respectivelt in the ERRFRB and OPREQ areas of the PROC sub-area of the C.A. ; traverse all processor data chains in search for the internal name in ERRFRB ; if a file control block is found, dump it on the system teletype ; if it is not found, print :

'NOT FOUND'

on the system teletype ; set the "file in hardware error" bit in the status 2 of the current assigned file list definition for the file.

(9) if the bit 1 of the PROC byte is set, take the following action with respects to all files in the current assigned file list :

. if a file is not "in use", ignore it ;

. if a file is not defined in any of the swapper, processor data chains, ignore it ;

- if a file is found within at least one of the chains, but the corresponding items pointing to the file have READ-ONLY access bytes, simply reset the "in use" bit in the status 2 of the assigned file list definition for the file ;
- if a file is found and at least one item points to it in either WRITE, PROTECT or UNPROTECT access, reset the "in use" bit and set the "file possibly in bad status" bit (BIT 6) in the status 2 of the assigned file list definition for the file.

(10) print on the current system output medium :

'PROCESSOR IN ERROR'

(11) print on the current system output medium :

'PROCESSOR NOT SUCCESSFUL'

if the LEVEL of the processor is positive OR if the LEVEL is zero and the system is in BATCH configuration, set the high-order bit of the RUN byte.

E. ABORT CONDITIONS. RESTART ROUTINES

An abort condition is characterized by a non-zero value of the RUN byte. The bits constituting the RUN byte have the following meaning :

(1) bit 0 (low-order bit) : this bit is set to one if a disk error occured during the current RUN ;

(2) bit 1 : set to one if a RUN time-out condition was detected by the REAL-TIME CLOCK process.

(3) bit 2 : set to one if a disk error occurred during the current RUN upon one of the system files.

(4) bit 3 : set to one if a system fatal error or an abnormal interrupt occurred during execution of the current RUN.

(5) bit 4 : set to one if a console interrupt was recognized and the system is in BATCH configuration.

(6) bits 5 and 6 : unused.

(7) bit 7 : set to one if the RUN is to be aborted because of a user error other than the occurence of the max RUN time.

The monitor checks the value of the RUN byte after execution of each processor. A non-zero value of the RUN byte causes the following sequence to be executed :

(1) if the bit seven is the onlt one to be set, go ABORT the RUN (call to the FIN processor);

(2) if the bit one is set, go ABORT the RUN via a call to the FIN processor ;

(3) if the bit zero is set, and the bit four is set, print on the system teletype the following message :

'FATAL SYSTEM FILE ERROR'

(4) if the bit three is set, print on the system teletype the following message :

'FATAL SYSTEM ERROR'

(5) disable the spooling process, and wait that the queue of operations on the disk be empty.

(6) type :

'PLEASE ENTER SENSE SWITCHES'

on the system teletype.

(7) halt ; wait for operator to press the RUN switch on the console.

(8) enter sense switches ;

(9) if the switches are all reset, go back to step (6) ;

(10) if the leftmost switch is up, jump to the beginning of the TOS program ; upon return from TOS, jump back to step (6) ;

(11) if the next switch is up, go abort the current RUN by executing a call to the FIN processor ;

(12) otherwise, if the next switch is not up, go back to step (6) ;

(13) if the next switch is up, set the RUN byte back to zero, and continue the execution normally.

9. SYNTACTICAL DEFINITIONS OF THE CONTROL LANGUAGE

The syntactic equations defining the control language are grouped together below. An "extended" notation is used to express the sintactic rules, where in particular :

$\left\{ \langle G \rangle \right\}_a^b$ stands for repetion from a to b times of G ;

$\left\{ \langle G \rangle \right\}$ stands for $\left\{ \langle G \rangle \right\}_1^1$

$\left\{ \quad \right\}_a^\infty$ stands for "indefinite repetition, at least a times";

[ ] stands for $\left\{ \langle G \rangle \right\}_0^1$

<nil> stands for the null element ;

(1) <control card> := <identification> <blank> <chain definition>

(2) <identification> := " <processor name> [, <processor option>]

(3) <processor name> := $\left\{ \text{letter} \right\}_1^8$

(4) <processor option> := <letter>

(5) <chain definition> := [<item sequence>] $\left\{ , \langle \text{item sequence} \rangle \right\}_0^\infty$ <closing delimiter>

(6) <closing delimiter> := ; | " |)

(7) <item sequence> := <explicit field> | <implicit field>

(8) <explicit field> := <item definition> = <object list>

(9) <implicit field> := <object list>

(10) <object list> := <default list> | <single object list> | <multiple object list>

(11) <default list> := <nil>

```
(12)   <single object list> := <object>    ( <data chain object>
                                             <closing delimiter>

(13)   <multiple object list> := ( <dobject> {, dobject}_0^∞
                                 <closing delimiter>

(14)   <dobject> := <default object> | <object>

(15)   <default object> := <nil>

(16)   <object> := <memory-bound object> | <file object>


(17)   <item definition> := <internal name> <item specification>

(18)   <itemspecification> := [ <type> ] [ <nummaxacc> ]

(19)   <type> := - { S|R|A|B|blank|C|I|H|F|D }

(20)   <nummaxacc> := ( <max> [ , <min> [ ,<access> ] ]<closing delimiter>

(21)   <max> := <nil>|<hexal>

       <min> := <nil> |<hexal>

(23)   <access> := <nil> |<hexal>

(24)   <internal name> := <letter> {<letter> |<digit> }_0^5


(25)   <data chain object> := <chain definition>


(26)   < file object>:=file identification> <file type>

(27)   <file identification> := [ <fileid>  * ] <user-defined name>

(28)   <fileid> := <hexa2>

(29)   <user-defined name> :=   <letter> | $

(30)   <file type> := [ - {S|R|A|blank}]
```

(31) <memory-bound object> := <Fobject> | <Iobject> | <Hobject> | <Cobject > | <Bobject>

(32) <Fobject> := <file identification>

(33) <Iobject> := <decimal> | <multvalhex2>

(34) <Hobject> := <decimal> | <multvalhex1> | € <multvalchar>

(35) <cobject> := [C] <multvalchar>

(36) <Bobject> := <true boolean> | <false boolean>

(37) <multvalhex2> := X ' $\left\{ \text{<hexa2>} \right\}_0^\infty$ [<hexa1>] '

(38) <multvalhex1> := X ' $\left\{ \text{<hexa1>} \right\}_0^\infty$ '

(39) <multvalchar> := ' $\left\{ \text{''} \mid \text{<non-quote ASCII character>} \right\}_0^\infty$

(40) <decimal> := <positive> | <negative>

(41) <positive> := [+] <integer>

(42) <negative> := - <integer>

(43) <true boolean> := 1 $\left\{ \text{<digit>} \right\}_0^\infty$ | Y $\left\{ \text{<letter>} \right\}_0^\infty$

(44) <false boolean> := 0 $\left\{ \text{<digit>} \right\}_0^\infty$ | N $\left\{ \text{<letter>} \right\}_0^\infty$

(45) <hexa2> := $\left\{ \text{<hexa>} \right\}_3^4$

(46) <hexa1> := $\left\{ \text{hexa} \right\}_1^2$

(47) <hexa> := A | B | C | D | E | F | <digit>

10. ERROR MESSAGES

The error messages occurring during control card processing are listed below as they can be found in the FFFF*PRO$ file.

(1) UNKNOWN PROCESSOR IDENTIFICATION : the processor name and option fields as specified on the control card do not match any of the processors in the current processor table.

(2) ILL-DEFINED PROCESSOR IDENTIFICATION : the processor name and option fields do not determine the processor uniquely within the current processor table. Hint : do not abbreviate the processor name too much.

(3) CHAIN RECURSION LEVEL TOO DEEP : the level of embraced parentheses is too deep (greater than 5).

(4) BINDING CANNOT BE OBTAINED BY DEFAULT : no default value can be bound to the item in this position. Hint : probably a comma too many.

(5) DUPLICATED INTERNAL NAME : The same internal name was used twice in the same data chain. Hint : check items generated by default in processor data chain and their associate positions on the control card.

(6) TOO MANY ITEMS FOR THIS DATA CHAIN : the number of items in the current data chain is greater than the MAX of the chain as an object. Hint : one of the internal names specified in the chain was mispelled.

(7) TOO FEW OBJECTS FOR THE CURRENT ITEM : The number of objects bound to the item is less than the MIN specified for the item. Hint : one of the objects was forgotten.

(8) TOO MANY OBJECTS FOR CURRENT ITEM : the number of objects bound to the item is greater than the MAX specified. Hint : probably a comma too many, or check definition for item.

(9) ATTEMPT TO CHANGE THE BINDING OF A PROTECTED ITEM : the item is 'frozen' to its default value, and an attempt was made to bypass the default.

(10) UNCORRECT FILE-TYPE : either an explicit file type was incorrect, or the file type was not explicited and the item is of type ' '.

(11) ILLEGAL FILE-ID : The file-id field in the file-name is incorrect.

(12) UNVALID BOOLEAN : uncorrect spelling of a boolean object.

(13) QUOTE ERROR : a quote was forgotten or the expression within quotes is incorrect.

(14) OVERFLOW : too large a value for an integer or half - word object expressed in decimal form.

(15) '=' SIGN FOUND MISSING : self explanatory. Hint : the internal name may be incorrect, or an object list was mispelled.

(16) UNCORRECT TYPE : The type explicited for the itsm is incorrect

(17) UNABLE TO ALLOCATE DATA CHAIN : self-explanatory.

(18) ACCESS ERROR : the file required is not available for the access required.

(19) FILE STATUS ERROR : the file requested is already in use. Hint : a file requested for processor execution (eventually by default) is used as input or output medium for the system.

(20) DUPLICATED FILE CONTROL BLOCK WITHIN CHAIN : an attempt was made to request the same file from two differnnt items of the same data chain. Hint : check default assignments for items of file type.

(21) FILE NOT FOUND WITHIN RUN$: the file requested is not currently assigned to the current RUN.

(22) END OF FIELD NOT MET : a comma or termination character was expected by the monitor program at this point of the processing. Hint : this error message can be printed-out in reason of many different errors ; however, the error can be located precisely from the monitor indication.

A general remark need to be done concerning the error diagnostics generated by the monitor : a '*' character is printed to indicate to the user the point in the processing where the monitor was able to recognize the error ; however, it should be understood that this indication may not always be meaningful ; for instance, the character will be in many cases located in front of the semicolon terminating the control card, since most of the default processing is done at that time ; the error (6), for instance, will in most cases be printed at the end of the processing of the control card, though it was caused by an error occurring generally much before this point.

Improvements are needed in the precision of the above messages ; a scheme has to be found to be able to give more precise indication of which particular field of the control-card was erroneous.

11. PROPERTIES OF THE CONTROL LANGUAGE

A. EQUIVALENCE BETWEEN CONTROL CARD AND INTERNAL REPRESENTATION

This approach taken in the beginning of this chapter for the description of the control-card processing mechanism was an internal approach. An identity such as :

item specification := object list

, on the control card, was described in terms of its internal equivalent, i.e in terms of such concepts in the system as items, objects and data chains.

Conversely, each item in any data chain for any process may be described in terms of its expression on a control card ; each default processor data chain is equivalent to a "default control card" ; a call to the "PROC processor (see this chapter) enables a user to list the "default control card" of a processor in his processor table.

This equivalence can be demonstrated by examining the case of the ASSEMBLER processor in the system ; the case is not chosen because of its generality but rather because it is a simple and typical example in the system.

The Assembler processor data chain contains the following four items (MIN = MAX = 4 , ACCESS = X'11' for the processor data chain as an object of the swapper chain) :

(1) item # 1 : internal name : IN; type : 'S' (expects a file of type source); MIN = MAX = 1; access required : X'11' (read only) ; default binding : 0000*CR$ (standard data flow input file for the current user). This item is used by the program to read the input source assembly language program.

(2) item # 2 : internal name : OUT ; type : 'R' ; MIN = MAX = 1 ; access required : X'12' (write only) ; default binding : 0000*REL$-R (standard relocatable file of the current user).
This item defines the file where the relocatable output of the Assembler is to be put.

(3) item # 3 : internal name : FLAG ; type : 'B' ; MIN = MAX = 3 ; access required : X'11' (read-only) ; default binding : YES,YES,YES ;
This item let a user specify if listing of the assembled program is wanted (first boolean) , if a listing of the symbol table is wanted (second boolean), and if a relocatable output is to be effectively written onto the output file (third boolean) .

(4) last item : internal name : PRINT ; type : 'S' ; MIN = MAX = 1 ; access : X'12" ; default binding : 0*PR$ (standard output data flow file). This item defines the file where the listing is to be produced ; this file is by default the standard output file, to be copied by the monitor onto the current system output medium.

The "default control card" corresponding to this default processor data chain will be the following :

```
"ASSEMBLE IN-S(1,1,X'11') = CR$,
          OUT-R(1,1,X'12') = REL$,
          FLAG-B(3,3,X'11') = ( YES, YES, YES ),
          PRINT-S(1,1,X'12') = PR$ ;
```

The above control card constitutes a valid call to the ASSEMBLER. However, it is not always the case that a default control card is a valid call to the processor ; for instance, we can define from the above processor another processor called "ASSEMBLE,Z having the same data chain, except for the second item (OUT-R(1,1,X'12')) , which, on the disk, does not have any object bound to it (empty binding). Then, the equivalent control card :

```
"ASSEMBLE,Z IN-S(1,1,X'11') = CR$,
            OUT-R(1,1,X'12') = ,
            FLAG-B(3,3,X'11') = ( YES, YES, YES ),
            PRINT-S(1,1,X'12') = PR$ ;
```

will no longer constitute a valid call to the assembler, since the second item is associated with an object list containing a number of objects (0) outside the MIN - MAX range (1 - 1). In the above case, the monitor program will, immediately after processing the second line, produce the following :

```
                    OUT-R(1,1,X'12') = ,
                                       *

ERROR : TOO FEW OBJECTS FOR CURRENT ITEM - PROCESSOR IGNORED
CARDS SKIPPED
```

B. SIMPLICITY OF USAGE

The expressions given as an example in the previous paragraph should not cause one to believe that the control language is not easy to use. they were shown in this form for the sake of example only.

Actually, the two following cards would produce exactly the same calls as in the preceding paragraph :

```
(1)                     "ASSEMBLE ;
(2)                     "ASSEMBLE,Z ;
```

By the very definition of the default card, the same effect can be achieved by specifying the processor name only, since the resulting processor data chain is to be identical to the default processor data chain.

In fact, in the case of the second call, the error indications given by the monitor become somewhat less clear than they were in the previous paragraph :

```
                        "ASSEMBLE,Z ;
                                    *
ERROR : TOO FEW OBJECTS FOR CURRENT ITEM - PROCESSOR IGNORED
CARDS SKIPPED
```

This example corroborates the judgments given at the end of the section 10 of the current chapter.

The programmer is allowed to be even less verbose :

```
                        "ASSE;
```

More abbreviation would be erroneous :

```
                                "ASS;
                                   *
ERROR : ILL-DEFINED PROCESSOR NAME - PROCESSOR IGNORED
CARDS SKIPPED
```

It was kept as a general rule to the definition of processor data chains that standard use of the system should be as simple as possible ; the following sequence will assemble, load and execute a single program using the standard input/output files :

```
"ASSEMBLE ;
"LOAD
"EXECUTE ;
```

If the assembly of a program stored permanently on the disk file system is needed, the programmer need to specify the file-name in order to overrule the default given :

```
"ASSEMBLE IN-S(1,1,X'11') = ( MYFILE-S ) ;
```

the specifications for the item may be left out, since they are not different from the specifications in the D.P.D.C. :

```
"ASSEMBLE IN = ( MYFILE-S ) ;
```

since a single object is needed, and since the file is expected to be of type 'S' from the item type, both parentheses and file-type can be left out :

```
"ASSEMBLE IN = MYFILE ;
```

Finally, the internal name itself can be taken out, since it corresponds to the first item of the DPDC :

```
"ASSEMBLE MYFILE ;
```

If the item is not the first in the DPDC, the programmer may opt for any of the following :

```
(1)                     "ASSEMBLE ,,(,NO) ;
(2)                     "ASSEMBLE FLAG = (,NO) ;
```

which in either case stands for :
" Assemble the following program, list the assembled program but do not list the symbol table " ; note that the first and third object of the FLAG item need not be explicited ; for an assembly with no listing at all, the following is valid :

```
"ASSEMBLE  FLAG = NO ;
```

Note that the following expression is erroneous :

```
"ASSEMBLE ,,,FLAG = NO ;
```

since it leads the monitor to try to define the item FLAG twice in the same data chain.

It is understood that the specification of a parameter located far in the list can be tedious for the programmer. Thus, the following actions are recommended when defining processor cards :

(1) use of standard "natural" internal names ; in the present state of the system, the following are commonly used :

. IN to define the natural input of the processor ;

. OUT to define the natural output of the processor ;

. LIST to define the various listing options ;

. PRINT to define the printing medium ;

. SPEC to define the command medium (specification file).

(2) block the parameters that programmers are more likely to modify at the beginning of the control card (IN, OUT, LIST) ; an option in the "PROC processor enables a user to "reorder" a default processor data chain (see corresponding chapter).

(3) provide the user with different default control cards for each given program in the system ; for instance, create a processor called "ASSEMBLE,N, so that the following call :

"ASSEMBLE,N ;

be interpreted as the previous call :

"ASSEMBLE LIST = NO ;

The facility of tailoring a control card already existing to a particular user's needs is given by the "PROC processor.

A standard use of options for these alternate default control cards should be recommended when declaring the control cards.

In conclusion to this paragraph, it appears that standard use of the system may involve very simple control card usage, if care is taken in the setting of the default options. No knowledge at all should be necessary, to the average user of the system, of the internal representation of control cards ; it is felt that the object types have been kept sufficiently basic and simple, so that no difficulty should be met in the setting of more complex control cards.

The item specification field (MAX, MIN, ACCESS, TYPE) is expected to be little used, and its existence should be ignored by most programmers. However, the creation of processors, the declaration of control cards, or extensive use of the control card as a device (see later in this paragraph) may require the understanding of this feature, and thus a minimum understanding of the underlying structures behind the control language.

C. GENERALITY OF THE CONTROL LANGUAGE

The monitor program processes the control cards in a strict move-forward fashion. The two following exceptions were made to this rule :

(1) a "look ahead" was done to determine whether a particular field is explicit or implicit ;

(2) a look ahead was done to determine if a file-id is explicited or not.

In these forward moves, the monitor does not cross record boundaries (i.e it will not pass from one card to another) ; two restrictions result from this fact, but should be of minor concern to a programmer.

The choice of the basic objects was mainly governed by a desire for simplicity ; in a few cases, the need was felt for a WORD object type, so that names be considered as objects in themselves, rather than as combinations of character objects.

On the whole, the control language was felt to be sufficiently general for the needs of the system ; the opposite effect was actually feared, specifically that the full power of the language be never put into use ; only one processor in the actual configuration, the EXECUTE processor (and, in a certain measure, the PROC processor), actually justify the existence of items of type item (i.e, the recursive definition of chain definitions). However, the full generality of the control language is hoped to find its justification in later system developments.

D. THE USE OF THE CONTROL CARD AS AN INDEPENDANT DEVICE

Emphasis is put here on the use of the control card as an independant device in the system. As shall be seen in the description of the EXECUTE processor, the user data chain is declared by the programmer on his control card, as a subchain of the EXECUTE processor data chain ; this particular chain (or eventually subchains) is meant to hold two classes of objects :

(1) file objects, in order to enable the user to use the disk driver to execute I/Os on files in the disk file system. The two stan-

dard input and output files (CR$ and PR$) are bound by default to two items defined for the user (IN-S and OUT-S) ; the files are given in read-only and write-only respectively. Thus, without any specification of the data chain, the user program may read data input from the current input medium, and produce output on the current output medium, by setting file request blocks with names IN-S and OUT-S and calling one of the resident or library driver routines. If any different file is needed, the user may override the default association for the items IN and OUT, or create any item of his choice. However, no request for PROTECT or UNPROTECT accesses will be satisfied.

(2) memory-bound objects ; any user program has access to the resident routine FETCH, and is thus enabled to read memory-bound objects within his data chain. The user may define items on his control card, and place requests to the routine FETCH to read one of the associated objects into the area of his program. A user program is entitled to the same facilities than a processor for communication with the control card. This may be used for one of the following reasons :

- it permits the isolation from the standard input stream of certain control parameters for clarity purposes ;
- it gives readability to the data by associating a name and type to each variable, array or string input ;

- it transfer the task of conversion to binary format from the user program to the monitor ; this should be particularly useful in the debugging of assembly language programs.
- it saves disk transfers.

However, the following inconvenients should be understood :

- it is not possible to declare arrays or strings whose dimensions are bigger than 255 ;
- the physical storage of data chains diminishes the total area available to the user program.

In conclusion to this paragraph, it appears that the control card is an elegant and convenient way for a user program to acquire a limited amount of data ; the tool should be particularly valuable in the debugging of assembly language programs. Standard read/write operations on system files are made possible by the declaration of items of file type on the EXECUTE processor control card.

12. EXAMPLE OF USE OF THE CONTROL LANGUAGE

No user manual was included in this thesis as such, for the reasons specified in the introduction. A short program is given in the overview chapter ; simple examples for the ASSEMBLER card were analysed in the preceding section. The default control card of each processor is given at the beginning of the description of the processor, both in the present thesis and in [1].

The following examples are given below in order to clarify the most complex default rules given in the section 3 of the present chapter. They are in no measure typical of average use of the control language.

An item in a (fictive) default processor data chain is defined by the following :

(1) internal name : ANITEM ;

(2) type : ' ' (file-type, particular type of file not specified) ;

(3) MIN = 2 ; MAX = 4 ; ACCESS = X'1F' ;

(4) number of default objects : 3 ; default objects as found on the disk : 0*CR$-S,0*PR$-S,0*ABS$-A ;

In addition, the item ANITEM is found in a data chain, the access byte of which is equal to X'13' ; it is the third item of the chain. The current file-id, as defined within the RUN sub-area of the communication area, is equal to X'1000'.

(1) Equivalent control card : if no explicit or implicit mention of the item ANITEM is done on the control card, the item generated will behave as if the following explicit mention had been done :

```
ANITEM(4,2,X'13') =
( 1000*PR$-S, 1000*CR$-S, 1000*ABS$-A ) ,
```

Note that the protect, unprotect accesses get masked by the access of the chain as an object, and that the zero file-ids gett set to the current file-id.

The default generation will also be caused by any of the following implicit declarations :

```
ANITEM = ,
ANITEM = (PR$,CR$,ABS$),
ANITEM = (,,),
(,,),                  ( on the third field ).
```

However, by application of default rules, the following calls :

```
ANITEM = ( , ) ,
ANITEM = ( ) ,
```

will behave as the more explicit following call :

```
ANITEM = ( PR$-S, CR$-S ),
```

The third default object is not bound to the final chain, since only MIN defaults are generated ; this rule would not hold if the item was of type 'D'. Note that the two calls :

```
ANITEM = ,
```

and :

```
ANITEM = (),
```

are not, in this particular case, equivalent.

(3) Modifications of the default bindings may be obtained by the following calls :

```
ANITEM =(PR$-S, ABS$-A  ) ,
ANITEM = ( PR$-S, 1000*ABS$-A)
```

Combinations of explicit and default values can be obtained :

```
ANITEM = (,ABS$-A) ,
```

(4) Modification of the item characteristics are demonstrated below :

```
ANITEM ( X'FF',0,X'13') = ,
```

will behave as the equivalent control card given by default.

```
ANITEM(2,0,X'13') =,
```

will behave as the more explicit :

```
ANITEM(2,2,X'13') = ( PR$-S, CR$-S ),
```

since the MAX has been changed.

```
ANITEM(,3) = (),
```

will generate the complete default binding as in :

```
ANITEM = ,
```

and by opposition to :

```
ANITEM = (),
```

, since the MIN is now equal to 3.
The following call :

```
ANITEM ( ,,3) = MYFILE-S,
```

is in error, since the default binding can no more be redefined from the change in the access byte fifth bit.

(5) the following calls are in error and are shown together with the error messages produced by the monitor :

```
                    ANITEM = ( , , ABS$-A, ) ,
                                             *
    ERROR : BINDING CANNOT BE OBTAINED BY DEFAULT - PROCESSOR IGNORED

                    ANITEM = ( , , , PR$-S ) ,
                                           *
    ERROR : DUPLICATED FILE CONTROL BLOCK WITHIN CHAIN

                    ANITEM = ( , , , TEMP$-S,TEMP$-A ) ,
                                              *
    ERROR : TOO MANY OBJECTS FOR THE CURRENT ITEM

                    ANITEM(3) = ( , , , TEMP$-S ) ,
    ERROR :                           *
    ERROR : TOO MANY OBJECTS FOR THE CURRENT ITEM

                    ANITEM-S = ,
                               *
    ERROR : UNCORRECT TYPE

                    ANITEM(,4) = ( , , TEMP$-S ) ,
                                                 *
    ERROR : TOO FEW OBJECTS FOR THE CURRENT ITEM
```

```
                    ANITEM =  PR$ ,
                                  *
ERROR : ILLEGAL FILE-TYPE
```

```
                    ANITM = PR$-S ,
```

may not be an error since it may be the creation of a user-defined item, if the MIN and MAX of the chain are not equal. It will be considered so at the time the above identity is processed, and thus the error will not be immediately recognized ; however, at the end of the processing of the chain, ANITEM will be created, together with its default binding ; if for the chain MIN = MAX, the total number of items in the chain will eventually become greater than MAX, and the monitor will print :

ERROR : TOO MANY ITEMS FOR THIS CHAIN

, with the '*' character in front of the closing delimiter terminating the chain (and eventually the control card).

```
                    "WHATEVER ANITEM=,,;
                                       *
ERROR : DUPLICATED INTERNAL NAME- PROCESSOR IGNORED
```

```
                    ANITEM(,,3) = ABS$-A,
                                        *
ERROR : ATTEMPT TO CHANGE THE BINDING OF A PROTECTED ITEM
```

CHAPTER XI :

THE LINKAGE TO A USER PROGRAM : THE

THE EXECUTE PROCESSOR

1. GENERAL DESCRIPTION

In many systems, a user program can be considered as a particular processor in the system. However, this scheme was not possible in the present Operating System, since no protection mechanism is offered by the microprograms ; no hardware or firmware protection is present to prevent a program from accessing any part of the memory, of any of the devices. A special processor, the EXECUTE processor, had thus to be implemented to limit, as far as possible, the destructive action that a user program may have on the system ; an additional memory overlay (see SWAPPER chapter) is thus necessary to bring the user program in core.

Protection is particularly needed for the following objects :

(1) system files ; files that are not in the current assigned file list are protected (see [1]) ; however, such system files as SCR$ and RUN$ are not protected when control is given to the EXECUTE processor ; the EXECUTE processor thus needs to protect such files to prevent accidental writing by a user program. The user does not have access to the PROTECT and UNPROTECT actions, since the corresponding bits are reset in the access byte of the user data chain (see next section). However, accidental execution of disk commands may not be prevented.

(2) memory ; the effects of a user program on the memory can only be one of the following :

. extensions of files in the track maps ;

. changes in the definition of end-of-files in the data chain.

The entire memory is thus swapped out before the user is brought in memory ; the user is given only a copy of the resident part of the system that he is entitled to use, specifically :

. disk driver and track maps, device tables ;

. swapper and its chain ;

. chain manipulation routines ;

. sequential access disk driver routines ;

. data chains.

At the end of the user program execution, the user context (the whole memory) will be swapped-out (if possible) ; the old context (of the EXECUTE processor) will be swapped in. then the modifyable part of the old context will be updated according to the saved user context, if the changes are legitimate. However, destruction of the necessary swapper by the user program cannot be prevented. Thus, the EXECUTE processor sets a bit in the RUN$ file before control is given to the user program ; a message is also printed out on the system teletype.

In the case when a user program is in a dead loop, or halts, the operator can take one of the following actions ;

. press on the interrupt key ; if the swapper is not destryed, control will be given to the entry point of the swap-out program, and normal execution of the EXECUTE processor will continue.

. load a special processor in memory (if the above method fails) ; this processor looks up the RUN$ file for the bit set by the EXECUTE processor ; if this bit is set, the processor is able to reload the context of the EXECUTE processor, and execution can continue normally from this point.

2. CONTROL CARD AND OPTIONS

The default control card for the EXECUTE processor can be written as follows :

```
"EXECUTE
ABS-A(1,1,X'13') = ABS-A,
SECTOR-I(1,1,X'11') = 0,
CHAIN-D(X'FF',2,X'13') = (
  IN-S(X'FF',1,X'11') = CR$,
  OUT-S(X'FF',1,X'12') = PR$   )   ,
DUMP-A(1,0,X'13') = ,
TIME-I(1,1,X'11') = 60,
MODE-B(1,1,X'11') = NO,
TOS-B(1,1,X'11') = NO,
LIST-S (1,1,X'01' ) = PR$-S ;
```

The successive items and their meaning are examined successively below.

(1) Internal name : ABS ; type : A ; MIN=MAX=1 ; ACCESS = X'11' ;
default binding : ABS$-A.
This parameter defines the file where the absolute program oan be found by the swapper.

(2) Internal name : SECTOR ; type : I ; MIN=MAX=1 ; ACCESS = X'11' ;
default binding : 0.
This item defines the sector number within the file ABS where the program dictionary can be found.

(3) Internal name : CHAIN ; type : D ; MIN = 2 ; MAX = @%% ;
ACCESS = X'13' ; default binding : the items :

. IN-S (MIN = 1, MAX = X'FF', ACCESS = X'11', default binding : CR$ (standard input data flow)) ;

. OUT-S (miN = 1, MAX = X'FF', ACCESS = X'12', default binding : PR$ (standard output data flow)).

This item defines the user data chain. Any number of items can be user-defined (255) ; the accesses given include READ and WRITE accesses, but not the PROTECT and UNPROTECT accesses (ACCESS = X'13') ; the user may thus define any item of his own, including items of type item and memory-bound items (for use of the control card as a device, see preceding chapter, section 11.D). However, the two following items are given by default ;

. IN-S is normally associated with the standard input data file (CR$) ; however, the user may redefine this item to any read-only file among the files in the current assigned file list ; any number of files needed in READ-ONLY by his program can actually be bound to this particular item (MAX = X'FF') ;

. OUT-S is normally associated with standard output data file (PR$) ; however, the user may redefine this item to be bound to up to 255 files to be accessed in WRITE-ONLY by his program.

(4) DUMP ; type : 'A' ; MIN = 0 ; MAX = 1 ; ACCESS = X'12'; default binding : empty binding.

If any file is specified by the user, the swapper will swap-out the memory image of the user on this file, instead of the standard SCR$ file ; this particular file will then be available to the user for :

. taking a post-mortem dump ;

. executing the output (CHECKPOINT facility).

(5) MODE ; type : 'B' ; MIN = 1 ; MAX = 1 ; ACCESS = X'11' ; default binding : NO ;

This parameter is transparent to slave users (i.e, the access byte is set to X'11' for this category of users). If the value of this item is YES, (MASTER mode), the system files will not be protected, and more entry points in memory will be available to the user program.

(6) TOS ; type ; 'B' ; MIN = MAX = 1 ; ACCESS = X'11' ; default binding : NO .

This parameter is "frozen" (by setting the access byte to X'01') for the users of the system in spool configuration. If this parameter is equal to YES, the EXECUTE processor will make the Teletype Operating System available to the program by :

. replacing the concurrent I/O area by TOS before giving control to the user program ;

. setting the trap location so that a TRP instruction or a console interrupt execute a jump to the beginning of the TOS programs.

A user in stand-alone configuration is thus able to use TOS for the debugging of his program. However, the EXECUTE processor then needs the concurrent I/O area for the storage of TOS (in the 16K configuration), and this area is thus necessarily not available to the user program.

(7) LIST ; type : 'S' ; MIN = MAX = 1 ; ACCESS = X '01'; default binding : PR$.

This parameter is completely "frozen" for all categories of users. It is used by the EXECUTE processor to be able to list the eventual user program output on the current system output medium after execution of the user program ; the EXECUTE processor is actually given the file in write-only access, since PR$ is normally a write-only file ; it actually updates the access to be able to read the file.

(8) TIME ; type : 'I' ; MIN = MAX = 1 ; ACCESS = X'11' ; default bin-

ding ; 60.

This parameter defines a max-time for the user program ; this time is initialized by the EXECUTE, and decremented every second by the real-time clock process during execution of the user program ; a minute is standard. If a max time occurs, a jump to the beginning of the swapper is executed.

3. THE SWAPPING PROCESS

As mentioned in chapter VI, the EXECUTE processor doos not use the system swapper, but rather a specialized swapper carrying out the following operations :

(1) swap the user in ;

(2) swap out all the user context ;

(3) swap the old context back in memory.

A. SWAPPING IN OF THE USER

At the beginning of this phase, the dictionary of the user program is already in core, and the verifications concerning the validity of the loading addresses are already done (see following section).

The control is given to the swapper to swap the user in, and the swapper gives control to the user with the values of the registers defined in the dictionary (A,B,X,W/O,P).

B. SWAP OUT OF THE USER CONTEXT

The conditions for completion of the user program are the following :

(1) execution of a :

```
JMP    STOP
```

is the normal completion sequence.

(2) power fail, power restart and stack overflow will cause also execution of the above sequence.

(3) max time (see previous section) ;

(4) console interrupt ; however, if TOS was required, a jump to TOS is executed instead of the previous sequence.

Upon completion of the user program, the control is given to the swapper, which performs the following operations :

(1) stop and disconnect I/Os upon the devices (card reader, line printer, disk) ;

(2) disable the interrupts and the real-time clock ;

(3) swap out the whole memory on the system scratch file ; this image will be eventually copied by the EXECUTE processor on a user file if a DUMP was required (see previous section).

C. SWAP IN THE OLD CONTEXT

The swapper then reloads the context of the EXECUTE processor (the whole memory) from the SCR$ file in a single call to the .DR routine. This operation requires the following conditions to be met :

(1) the disk interrupt has to be the only interrupt that can occur during the operation ;

(2) the queue of disk operations must contain only one operation (the swap operation) ;

(3) the current context and the SCR$ context must have in common the following piece of program (waiting loop for the disk interrupt and return from .DR routine) :

```
CAL   .DR
LDV+  8
NAZ   *-2
```

so that, when the piece of program is loaded from the disk, the execution flow stay undisturbed ;

(4) the byte defining the operation requested in the file control block in the SCR$ context, needs to be non-zero ;

(5) upon reception of the end-of-transfer interrupt, the SCR$ context (now fully loaded into core) must be compatible (stack, device queue, FCB) with the configuration of a program awaiting the end-of-transfer interrupt from the disk ;

(6) the disk image in the SCR$ file must be within one single allocation, in order for the .DR routine to be able to execute the transfer in one single step.

4. VERIFICATIONS AND PROTECTION BEFORE USER EXECUTION

In order to avoid as much as possible the number of system crashes and their severity, the EXECUTE processor does as much verification of the user program as possible, and also tries to protect the system against unintentional mistakes in the user program.

A. VERIFICATION PROCESS

Verification is done that the file declared by the ABS item of the control-card is formatted correctly ; more precisely, the sector defined by the item SECTOR of the control card needs to be formatted as an absolute program dictionary. Reference is done to MANGIN, [1] for a description of the absolute program dictionary. The specific verifications are :

(1) number of segments   59

(2) for each segment,

  low program limit   segment start address   segment end address
  high program limit

(3) low program limit   execution address   high program limit

(4) first location available to user   low program limit

(5) high program limit   top memory available to user program

(6) stack location = system stack location or

  low program limit   stack location   high program limit

If all of the above conditions hold, the file is considered as correct ; otherwise, en error message is printed and the user program is not executed.

B. PROTECTION PROCESS

The protection process seeks to protect the system against mistakes in the user program. The entire memory is swapped out on the system scratch file (SCR$) and the disk image is updated for the swap back by the swapper. As said in the last section, the waiting loop for the two swappers is the same :

```
SDR     DC      **
        CAL     .DR
        LDV+    8
        NAZ     *-2
        JMP*    SDR
```

The swap out process is executed by a return jump to SDR.

This provides the SCR$ context with the property of compatibility for the swapping back process (configuration of program awaiting an end-of-transfer interrupt from the disk).

In order to be able to distinguish between the return from the SDR program at swap out time and at swap (back) in time, the contents of the locations SDR and SDR+1 of the DISK image are updated, so that a jump to the proper part of the EXECUTE processor be executed at the end of the execution of the swapping process.

Furthermore, a byte in the current user area of the RUN file is set to one to indicate that, in case of a complete crash of the user program, a memory image on the disk is ready to be restarted.

The other protection steps are relative to the protection of system files.

C. OTHER ACTIONS

The other actions performed by the EXECUTE processor before giving control thethe swapper include the following :

(1) write on the system teletype the expected time of execution of the user program (as specified on the control card) ;

(2) load the Teletype Operating System in the concurrent I/O area if the TOS option on the control card is set ;

(3) set the PROTIM location of the PROC subarea of the communication area to be equal to the user maxtime ;

(4) set the interrupt address defined in the PROC area to the start of the swapper program ;

(5) set the console interrupt address to the TOS execution address if the TOS option on the control card is set.

5. UNPROTECTION AND VERIFICATIONS AFTER USER EXECUTION

After the swap back of the EXECUTE processor context, the system files are unprotected. The message :

USER OUT

is printed on the system teletype. The byte set to one in the RUN$ file is reset.

The EXECUTE processor needs to record the legitimate changes to the memory done by the user.

Files extensions involve modification of the EOF byte in the FCB definition, and modification of track maps. A traversing of all the File Control Block is done to check that the corresponding first allocation offset has not been modified ; the eventually added allocations are compared with the allocations bound to the file in the previous context. The following properties must be true :

(1) the chain (starting from the first allocation) of the allocations already bound before must still exist after execution of the program.

(2) The extensions must belong in the AVAIL list of the original context.

(3) No allocation may be shared betwwen several different files.

(4) The End-Of-File defined in the File Control Block must be smaller or equal to 192 x N, where N is the total number of allocations bound to the file.

If all the conditions are true, the file extensions are reported on the track maps and file control block of the EXECUTE processor. In the opposite case, the original definition of the file is kept and a warning message is printed out on the current system output medium.

If the list file was not found in error by the previous algorithm, it is copied by the EXECUTE processor onto the current system output medium (via the PRINT process). All characters are checked to be ASCII, and, if not, replaced by the character @. The printing is aborted when too many non-ASCII characters have been encountered.

CHAPTER XII :

PROCESSOR EXTENSION AND TAILORING :

THE PROC PROCESSOR

1. PROCESSOR DEFINITIONS

In the description of the MONITOR, two aspects have been overlooked, specifically how current processor table and processor information area are created.

Normally, the current processor table is initialized at RUN time to one of several DEFAULT PROCESSOR TABLES on the disk. The pariicular default processor table may be selected by the user among a list of default processor tables available to his password.

The set of processor information areas that can be reached through at least one default processor table constitute the set of "permanent" processor information areas. They are defined permanently in the RUN$ file, and define standard options for the processors in the processor files. There may be several processor information areas associated with a single program, corresponding to different default control cards or memory requirements. Moreover, certain items may be "blocked" to their default bindings. For instance, we may decide that the TOS option in the EXECUTE processor be given only to a certain class of users ; for the other users, the access byte fifth bit of the TOS item is set to zero, so that the users cannot change its value ; only the new EXECUTE processor (which may have another name, such as EXECUTE,T) would be placed in one (or several) of the default processor tables available to the class of users. Another example would be that we decide that a "big" Assembler is needed for the assembling of large programs. A larger symbol table would be made available by setting the "concurrent I/O acceptance indicator" byte of the processor information area to zero, and modify accordingly the maximum memory requirements (MAXMAX) of the new processor.

Thus, many processor information areas may be associated with a given processor. Only one copy of the program need to be kept on the disk, and, in a multiprogrammed environment, the absolute programs can be written as a pure code, each user accessing the unique copy mapped in memory via a separate data chain. Very general processors can be created, and adapted to particular needs by creating different processor information areas relative to the program.

Since the current processor table is initialized at each RUN, a user may tailor a processor defined in the system to his own needs. For instance, a user working regularly on a given program may create a version of the EXECUTE processor with all the arguments needed to the execution of his program, such as absolute file name, sector address, standard parameters in the data chain of the program, a.s.o. The modified default control card would be kept in an vailable area of the RUN$ file (a "user" processor information area is thus created). At Run completion, the used areas are returned to a free list, so that the changes pertain to a user only.

2. THE PROC PROCESSOR

The PROC processor enables a user to create new processors in the system, tailor exeisting processors to his own needs, and make libraries of absolute programs available to his RUN.

The creation of new processors is a MASTER option, since it enables a user to execute one of his own programs without the system files and the resident memories being protected as in the case of a program executed via the EXECUTE processor. The successive phases needed to introduce a new processor in the system are desribed in section 4.

On the other hand, the option of tailoring a processor to a user's needs is available to all users ; the process is described in section 3. of this chapter.

The PROC processor data chain is described in section 5. Since it is a complex chain, several processor information areas should be created ; an ACTION byte decides which option is demanded ; the ACTION byte is "frozen" for slave users.

3. TAILORING A PROCESSOR

Tailoring a processor is defined as a user-available option. A new processor information area is created from modifications of a processor information area already existing and accessible through the current processor table. Controls are provided that the new processor is more "restricted than the ol processor.

Mention of the processor to be modified is done by the processor name. A user is entitled to the following changes ;

(1) the processor name itself. If no processor name is explicitly mentioned, the processor name is assumed to be unchanged.

(2) the prompt message printed out by the monitor ;

(3) if the "concurrent I/O indicator" is et, it can be reset by the user ;

(4) the TOP memory requirements can be increased (but not diminished) ; the LOW memory requirements cannot be changed ;

(5) the default processor data chain can be modified with respects to the following :

. a MIN can be specified ; the resulting MIN will be taken to be the maximum of the user MIN and the previous MIN ;

. a MAX can be specified ; the resulting MAX will be taken to be the minimum of the user MAX and the previous MAX ;

. an ACCESS may be specified ; the fifth bit of the resulting access will be taken to be the logical AND of the two corresponding bits ; the other bits cannot be changed, and the user specifications will be ignored ;

These considerations are valid for the definition (MIN, MAX and ACCESS) of the default processor data chain as an object. If the resulting access bit is set, the default binding of the resulting item is taken to be equal to the binding specified for the item on the control card. However, no mixing of default and user-defined objects is possible on the control-card.

New items can be created by a user ; all the user created item will be appended at the end of the default processor data chain, in the order in which they appear on the control card ; the user is responsible for the modifying of the MIN and MAX of the data chain in which the item is inserted, so that the resulting number of items be still correct ; the access byte of the newly created item is masked by the access of the data chain in which it is inserted.

The order of the items in default processor data chains can also be modified. This constitutes a separate option (ACTION = 12) ; the user need not specify any of the item types, characteristics, or default bindings of the items referred, since they will not be modified. The order of items in subchains can also be changed by use of this option.

The name and prompt message associated with the resulting new processor can be redefined by the user as in the previous case (ACTION = 9).

The two above actions can be combined (ACTION = 13). In this case, as might be expected, the modifications are carried out before the reordering is done.

The additional option is given to a user to delete a particular processor in the current processor table (ACTION = 2).

Both in the cases of modification and reordering, the name of the new processor can be any name of a processor which is not already in the current processor table ; in the case when the user does not want to redefine the processor name, the original processor need to be deleted ; the deletion operation can be combined with any of the above (ACTION = 11, 14 and 15).

4. ADDITION OF NEW PROCESSORS IN THE SYSTEM

A new processor can be introduced in the system by carrying out the following operations ;

A. INITIAL DEBUGGING

An initial debugging of the processor is necessary to make sure that the processor does not destroy any vital information in the system files and the communication area. This part can be carried out by use of the EXECUTE processor ; if any effect upon system files is to be analysed, or if any routine of the extended resident is needed, the MASTER option of the EXECUTE processor needs to be set ; the default processor data chain needs to be specified on each call to the EXECUTE processor.

B. DEBUGGING, PHASE II

Additional debugging is generally necessary to make sure that the processor carries its purpose in the system. In particular, the effect upon the communication area could not be conveniently analysed by use of the EXECUTE processor, since the communication area image is reloaded together with the rest of the memory at the end of the processor execution.

A special option of the PROC processor (ACTION = 1) enables the user in master mode to declare a new processor information area for the processor ; the user may specify the default control-card ; control card that do not represent avvalid call for the processor (for instance if an item must have MIN = MAX = 1 but no default binding) cannot be directly declared, but need a second call to the PROC processor to modify the default data chain. For instance, an item of a processor default data chain such as ;

ANITEM(1,1) = ,

would be initially declared as :

ANITEM(1,1) = CR$-S, / R$-S is a dummy/

then modified by a second call to the PROC processor (ACTION = 11)to be :

ANITEM(1,0) = ,

so that, from the rules given in the previous section, the resulting item be correct.

A user can then run easily a variety of test cases.

C. MAKE THE PROCESSOR PERMANENT IN THE SYSTEM

The processor created above is not declared outside of the RUN for two major reasons :

(1) the current processor table is destroyed at the end of the current RUN, as well as the "user-defined" processor information areas.

(2) the processor declared above is in one of the user's personal files, rather than in one of the processor files.

A "provisional" permanent processor in the system can be created by :

(1) making the processor information area "permanent" by deleting it from the AVAIL list of the RUN$ file and appending it to the permanent processor information area list (see [2]) ;

(2) insert the processor name in one of the default processor tables (or several of them). This can be done by executing the following steps :

. use the special options (ACTION = 5 and 6) of the PROC processor to create in the current processor table the exact copy of the desired contents of the desired default processor table ;

. copy the current processor table into the relevant default processor table.

(3) place the program in one of the system libraries.

The actions (1) and (3) above, as well as the second step of the action (2), must be done by hand in the current version of the system. A processor could be easily created to carry them out.

It is recommended that a special default processor table, available under a single password, be created to contain the most general default processor data chain for each of the processors in the system. The user of the password may then decide to put "restricted" DPDCs in other default processor tables.

D. THE FINAL STEP

The above steps are not sufficient to make a processor permanent, for the RUN$ file is normally copied from a read-only back-up at each system generation. When the steps above have been successfully carried out, a copy of the updated RUN$ file must be done on the back-up file. It is advisable that this copy be done by hand.

5. THE PROC PROCESSOR DEFAULT DATA CHAIN

The default control card is the following :

```
"PROC
ACTION-H(1,1,X'11') = 0,
NAME-C(10,0,X'11') =,
PARAM-D(X'FF',0,X'11') = (),
LIST-B(3,3,X'11') = ( YES, NO, NO),
PROMPT-C(X'20',0,X'11') = ,
FILE-A(1,0,X'11') = ,
SECTOR-I(1,0,X'11') = ,
MEMORY-I(4,0,X'11') = ,
CONC-B(1,0,X'11') =,
LEVEL-H(1,0,X'11') =,
TIME  -I(1,0,X'11') = ,
INTADD-I(1,0,X'11') = ,
NEWNAM-C(10,0,X'11') = ;
```

A. THE LIST BOOLEANS

The item LIST expects three different objects :

(1) if the first boolean is YES, the PROC processor will acknowledge successful execution of each action ;

(2) a true value of the second boolean causes the current processor tablr ;

(3) a true value of the third boolean causes the processor given by the item NAME to be printed out.

B. POSSIBLE ACTIONS

The possible actions include the following ;

(1) ACTION = 0 ; this a "no action required" option ; the PROC can still be used for listing purposes, via the LIST options.

(2) ACTION = 1 ; the processor whose name is specified in the item NAME is deleted from the current processor table.

(3) ACTION = 2 ; option of creation of a new processor ; all the parameters in the processor information area must be defined.

(4) ACTION = 6 ; this option causes all processor names in one of the default processor tables given to a user, as specified in the sector SECOOR, to be appended to the end of the current processor table, as long as their names are not already defined within the current processor table ; if a processor name is specified in NAME, only the processor name specified is searched for in the default processor table specified.
This option can be used to call libraries of absolute programs.

(5) ACTION = 5 ; this option causes all processor names in the default processor table specified in the sector SECTOR to be deleted from the current processor table, if their names are defined in both tables ; if a processor name is specified in NAME, only this processor will be deleted, if it is found in the processor table defined by SECTOR.
This option can be used in conjunction with (4) above to create default processor tables, as described in the previous section.

(6) ACTION = 10 ; this option enables a user to tailor a processor to his own needs, as described in section 3 of this chapter. No reordering of the parameters takes place, and the new items are appended to the end of the data chain in which they are declared. No deletion operation takes place, and hte new processor name must be defined under the name NEWNAM (old processor : NAME), and the name must not be already in the current processor table.

(7) ACTION = 11 ; same effect as above, except the processor name NEWNAM is deleted prior to the execution of the above. If no processor name is declared under NEWNAM, the new processor name is assumed to be NAME, and the processor defined by NAME is deleted from the current processor table.

(8) ACTION = 12 ; the processor NAME is renamed in NEWNAM and the prompt is changed ; if any item is specified in the chain defined by PARAM, the items in the corresponding chain in the processor information area are reordered accordingly ; not all items need be mentioned, and the item specifications and/or the bindings eventually specified on the control card are ignored. The prompt is redefined to be the PROMPT message specified by the user, in all cases, even if the PROMPT is left empty.

(9) ACTION = 13 ; same effect as above, except that a deletion occurs, as explained in (7) above.

(10) ACTION = 14 ; this option is a combination of the options (6) (ACTION = 10) and (8) A ACTION = 12) above, in this order, ie the items modified are appended to the beginning of the new processor data chain.

(11) ACTION = 15 ; this option is a combination of the options (7) (ACTION = 11) and (9) (ACTION = 13) above, in this order.

C. PROCESSOR NAMES

The items NAME and NEWNAM contain processor names. The formatting of the processor names must be as on the control card. However, names cannot be abbreviated and no trailing blank may be appended to the processor name. Example :

NAME = 'EXECUTE,X',

The item NAME contains the processor name to list is LIST(3) is set, the processor to delete if ACTION = 1, the processor to create if ACTION = 2 ; it reference a processor in a default processor table if ACTION = 5 or 6 ; it references a processor already defined if ACTION = 10 or above.

The item NEWNAM gives a name to "tailored" processors in the cases where ACTION = 10 or above. If no name is declared in NEWNAM, the new processor name is assumed to be equal to the previous one.

D. PARAM

The item PARAM is used to contain the processor data chain of the created processor if ACTION = 2 ; the MIN, MAX and ACCESS are the MIN, MAX, and ACCESS of the resulting default processor data chain.
The item PARAM also contains the user modifications that a user wishes to perform on a default processor data chain already existing ; the rules for the modifications are explained in section 3. this use concerns the values 10, 11, 14 and 15 of the ACTION item. The item PARAM are also a way to redefine the ordering of a default processor data chain (or of subchains of this chain) in the case of values 12 or above of the ACTION item ; as specified above, in the case of the actions 12 and 13, nothing needs be specified except the names   the item

E. FILE and SECTOR

These parameters are used to reference the location of the disk file system where the absolute program is to be found in the case(ACTION = 2) of declaration of a new processor. The SECTOR is used to refer a default processor table in the case ACTION = 5 and 6 (the FILE is then not used). The parameters are dummy in all other cases.

F. PROMPT

The parameter PROMPT contains up to 32 ASCII characters to be printed at each execution of the processor defined or modified. The PROMPT can be empty (no prompt will be printed). This applies to the values 2, 10 and above of the ACTION item ; it is a dummy in the other cases.

G. MEMORY

This item defines the memory requirements of a processor created, as defined in the processor information area, in the order MAXMAX, MINMAX, MAXMIN, MINMIN.

In the case of tailoring of an already existing processor, only the two first objects (MAXMAX and MINMAX) have any meaning, and the modifications can be in the sense of increases in the memory requirements.

H. CONC, LEVEL, TIME, INTADD

These parameters correspond to the "concurrent I/O indicator", "processor level", "time-out condition" and "interrupt address" defined in the processor information area.

In the case of the creation of a processor, all items must be specified. In the case of modification of a processor (ACTION = 10, 11, 14 and 15), they may be omitted.; if they are specified, the following hold :

(1) the resulting "concurrent I/O indicator" is equal to the logical AND of the CONC specified and the old value ;

(2) the processor level may only be increased ;

(3) the time-out condition may only be diminished ;

(4) the interrupt address cannot be mofified.

The parameters are dummy in all other cases.

CHAPTER XIII :

CONCLUSIONS

Conclusions are drawn below with respects to the following :

(1) satisfaction of the requirements ;

(2) extensions.

1. SATISFACTION OF THE REQUIREMENTS

An attempt is made below to estimate some characteristics of the O-perating system, with respects to the requirements given in chapter II.

A. RELIABILITY

The reliability of the system as a whole cannot be honestly estimated in the present, since, at the time this thesis is submitted, the system has been used by the authors only. The basic routines (Disk driver, chain manipulation routines, I/O handlers, ...) have been fully tested through use of the system in its intermediate form by students in a Computer Science course.

It is felt that the reliability of the error-handling routines is doubtful. As was explained in the present thesis, no error routine was incorporated to the resident, but rather the monitor was given all responsibilities concerning error analysis and recovery ; the effectiveness of this scheme still has to be confirmed. It is the author's opinion that at least a system teletype handler and a simple error recovery routine should be added to the minimum resident, and that theses routines can be kept sufficiently simple, so that the resulting system does not exceed the 16K limit.

B. PROTECTION

(1) Memory protection : the memory protection implemented in the EXECUTE processor is felt to be a reasonable software solution to the lack of memory protection by the microprograms. However, a true memory protection scheme should be implemented by a modification of the existing microprograms. The checking of a slave/master bit should be done in order to control the following :

. execution of certain instructions (disk I/Os, ...) ;

. WRITE access to certain parts of the memory.

(2) Disk protection : Protection of a file against unintentional destruction is insured by the software at different levels, specifically :

. when a file is recalled (i.e. placed in the current assigned file list), a specific access is specified ; the FILE processor checks that the access is compatible with the definition of the file in the proper directory ; if the access is in read-only, the file is not unprotected ;

. when a processor or user program requests a file, a specific access is specified, and this required access is checked by the monitor against the access in the current assigned file list ;

. when a disk operation is requested (via the .DR routine), the disk driver makes sure that the file is available for the corresponding access.

The resulting protection scheme is felt to be both strong and flexible. Additional protection of system files is done by the EXECUTE processor before the user is given control.

(3) System protection : a recovery processor is strongly needed to restart the system, in case of a system crash.

C. DUAL CONFIGURATION

The basic goal of a dual configuration (stand-alone, batch) has been achieved. However, the following remarks can be done :

(1) the spooling system lacks in generality ; too much attention was paid to giving the card reader a fast servicing rate ; the interrupt driven scheme would be advantageously replaced by a method in which a scheduler would drive the I/Os during the idle timed of the C.P.U.

(2) the handling of end-of-device conditions couls be implemented via recursive calls to the FILE processor, rather than performed by routines of the extended resident ; this method would have the following advantages :

- core storage : the extended resident routines would be simplified.
- generality : explicit stacking as well as explicit unstacking of input/output drivers could be dynamically treated via a recursion over the monitor.

(3) OPEN and CLOSE functions for a device should be implemented, in order to be able to incorporate the tape as an alternative input/output medium of the system.

D. EXTENSIBILITY

Additions of new processors to the system can be carried out simply by use of the PROC processor. Addition of new input/output devices (paper tape,...) require minimum modification of the programs.

In general, modularity in the system is obtained via systematic usage of the data chain technique largely described in this thesis. The set of objects in the system that a particular process accesses and/or modifies can be readily recognized. However, a reorganization of the communication area appears as being a necessity.

Minor modifications in the residents does not require any recompilation of processors, as long as the overall boundaries are respected (X' 1000' for the minimum resident and variable length record driver, X'2000' for the extended resident).

2. EXTENSIONS

Some propositions for improvements and/or extensions of the existing system are given below.

A. MINOR EXTENSIONS

A set of routines are needed in order for the system to be usable, including the following :

(1) a RUN and a FIN processors ;

(2) accounting routines, additions of passwords, disk dump routines ;

(3) a recovery processor ;

(4) error analysis and recovery routines..

B. MAJOR IMPROVEMENTS

Improvements requiring a significant modification of the existing system may be conducted in the following directions :

(1) Microprogramming, including :

. memory protection and instruction control ;

. microcoding of some of the resident routines ;

. implementation of a stack underflow interrupt.

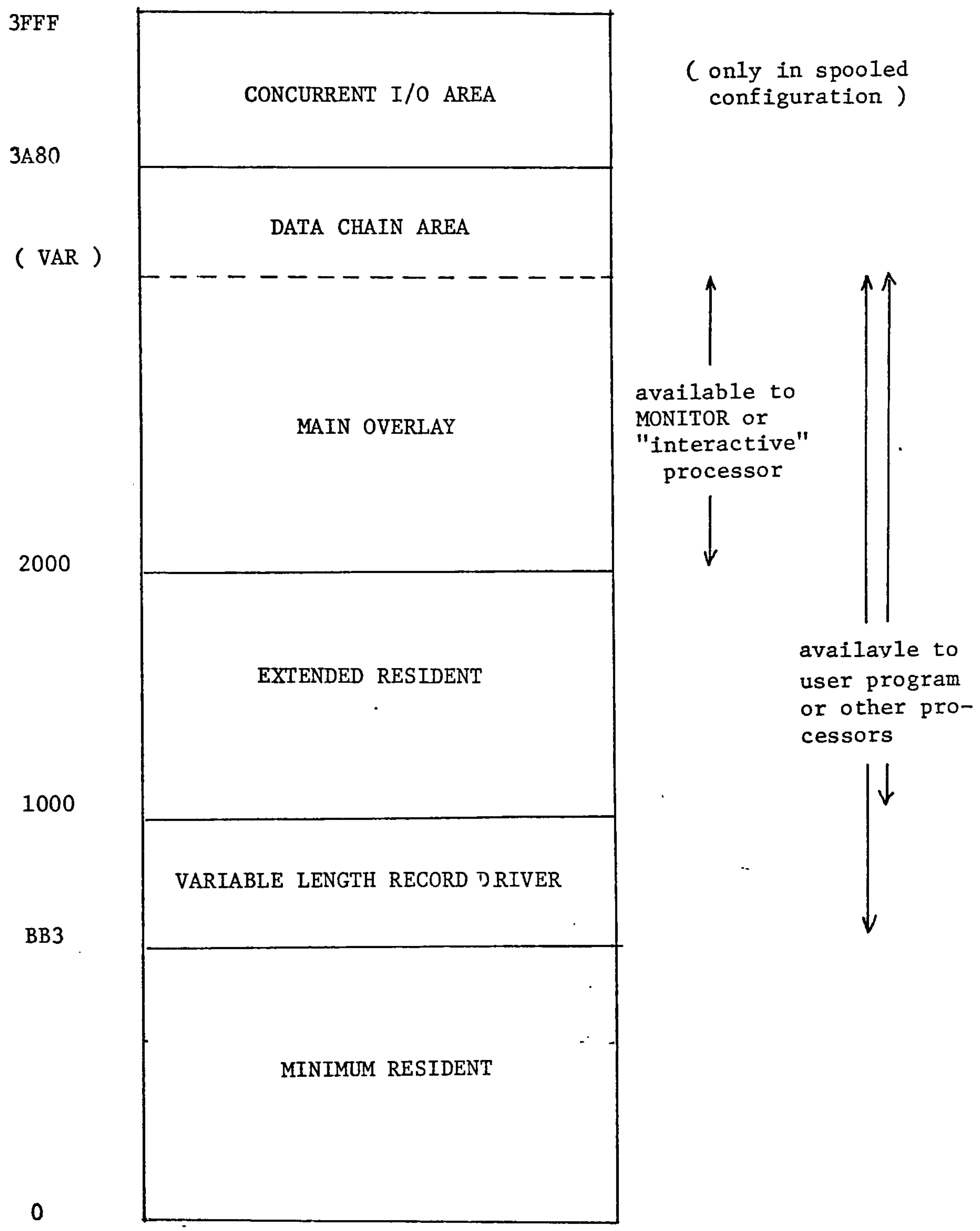
(2) Data chains ; it is felt that data chains could constitute the core of a serie of possible extensions of the system. For instance, an additional "buffer" memory-bound object type could be implemented (a page in memory, associated with a semaphore), so that buffer allocation be done by the monitor rather than by the processes themselves. Primitives to protect, unprotect and write upon memory bound objects could be designed and added to the minimum resident.

(3) Reentrant coding ; these extensions in data chains should enable one to rewrite both MONITOR and SWAPPER as pieces of pure code ; the necessary buffers and temporaries would be accessed through use of data chains. It is the author's opinion that these modifications can be carried out within the 16K limits ; the additional memory storage needed should be obtained from the fact that the extended resident routines would be able to treat end-of-device conditions via recursive calls upon the swapper and monitor.

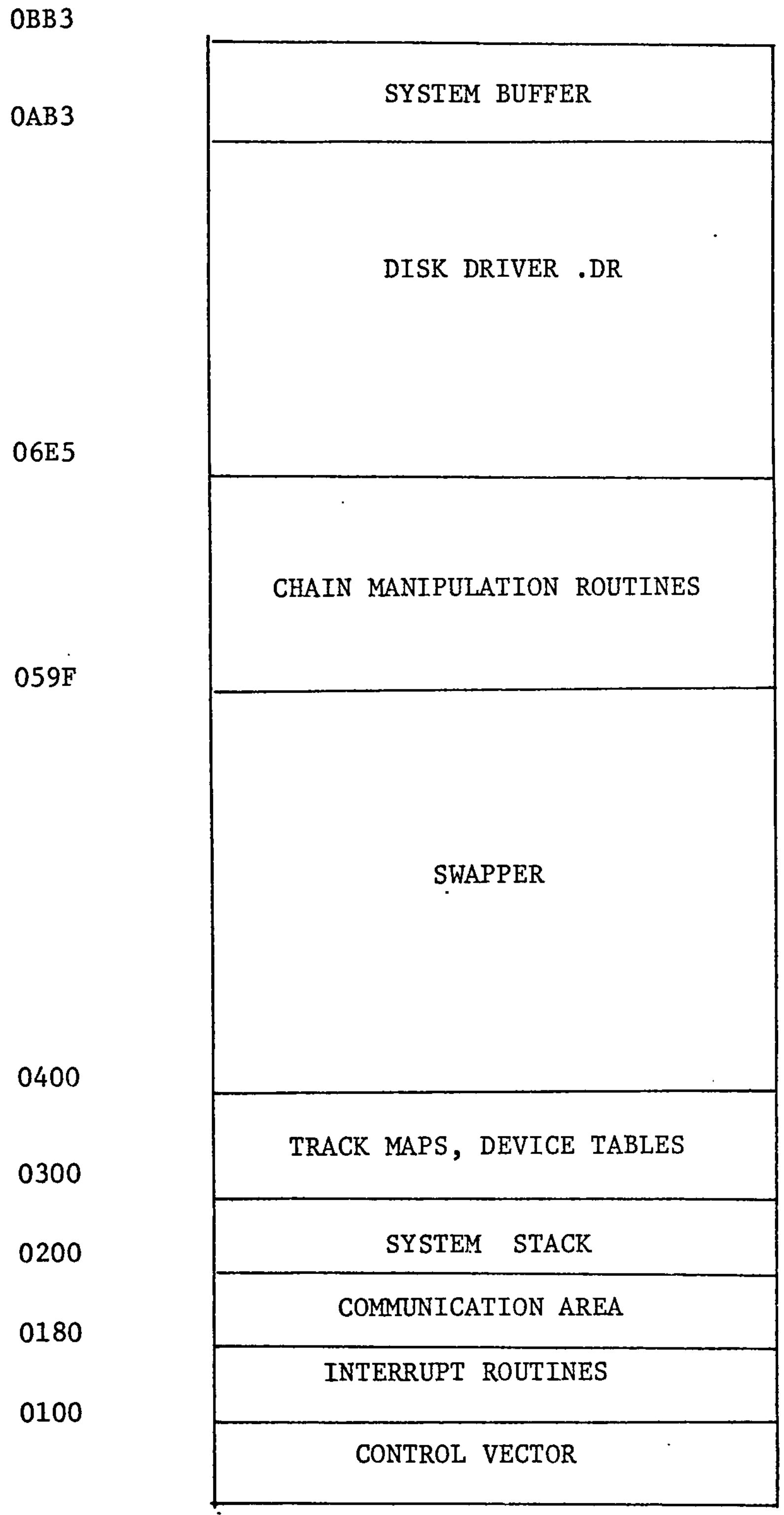
(4) File system ; some propositions are made in [1] ; however, any major extension of the file system (such as the design of program files) would make 32K of core necessary.

APPENDIX

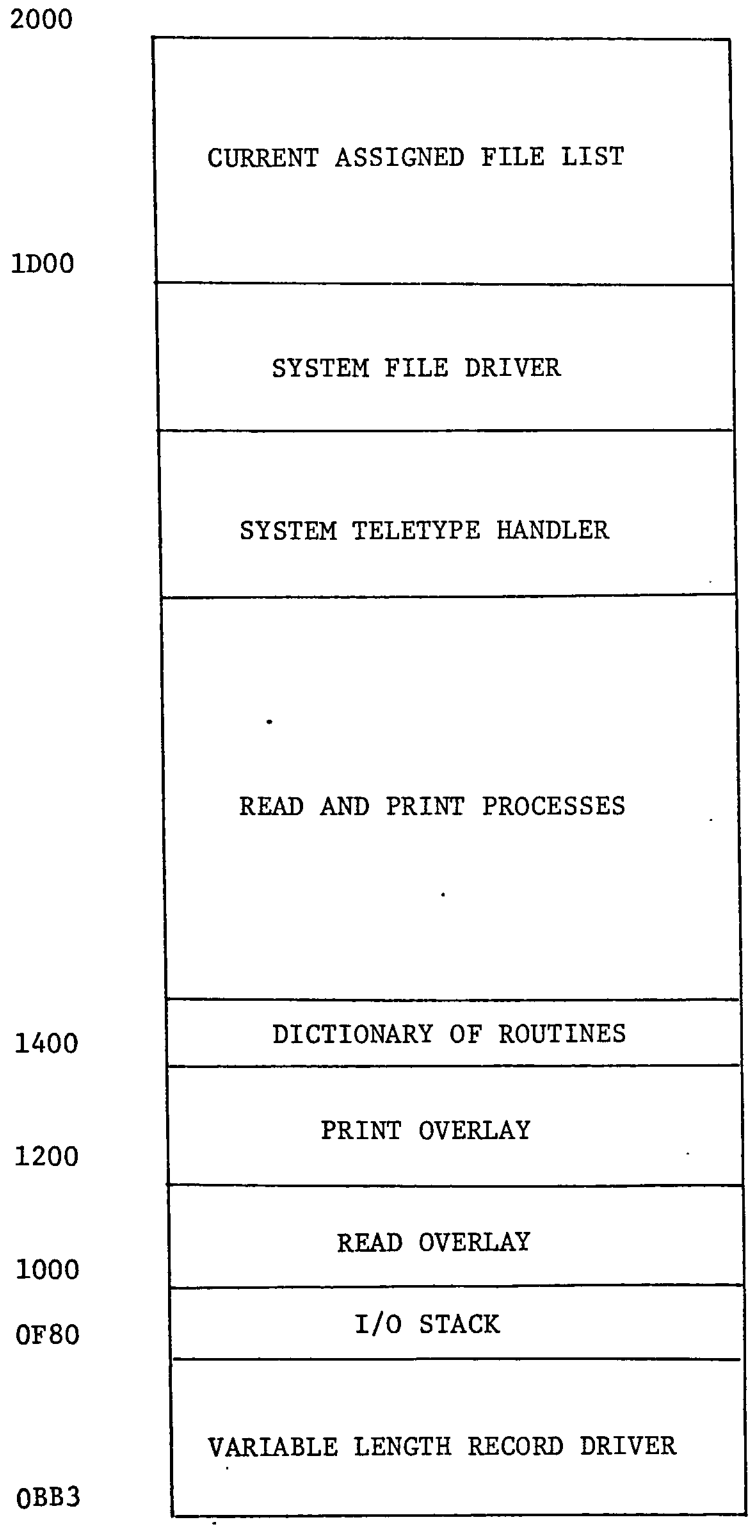
MEMORY MAPS


3FFF
CONCURRENT I/O AREA
(only in spooled configuration)
3A80
DATA CHAIN AREA
(VAR)
MAIN OVERLAY
available to MONITOR or "interactive" processor
2000
EXTENDED RESIDENT
availavle to user program or other pro-cessors
1000
VARIABLE LENGTH RECORD DRIVER
BB3
MINIMUM RESIDENT
0

Memory map



<u>MINIMUM RESIDENT</u>



EXTENDED RESIDENT

BIBLIOGRAPHY

[1] Xavier Mangin, Master Thesis, June 1975 (University of Houston).

[2] Microdata documentation , Assembly Language, Teletype Operating System

[3] Prime Disk Operating System , Prime 200 DOS Reference Guide (may 1973)

[4] E. W. Dijkstra , The Structure of the "THE" Multiprogramming System

[5] Computer Scince Department, University of Houston, Internal System Documentation