

A CONCURRENT PASCAL SPOOLING PROGRAM

by

MICHAEL EUGENE PRESS

B. S., Kansas State University, 1976

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1979

Approved by:


Major Professor

Document
LD
2668
.R4
1979
P74
C.2

TABLE OF CONTENTS

CHAPTER ONE

1.1	Project Overview.....	3
1.2	Document Outline.....	7

CHAPTER TWO

2.1	Brief Module Description.....	9
2.2	Virtual Disk Space and File Structure.....	13
2.3	User-Supplied Information.....	14
2.4	Intermachine Communication Protocol.....	16
2.5	Disk Backup Procedure and Priorities.....	20
2.6	Memory Usage.....	27
2.7	Scenario of Operation.....	30
2.7.1	Opening a File.....	30
2.7.2	Adding Data to an Existing File.....	32
2.7.3	Closing an Open Input File.....	33
2.7.4	Scheduling a File for Print.....	33
2.7.5	Printer Process Action.....	35
2.7.6	Memory Reclamation.....	36

CHAPTER THREE

3.1	Project Summary.....	37
3.2	Proposed Enhancements.....	39

APPENDICES

A1	Bibliography.....	43
A2	Detailed Module Description.....	45
A3	Operator Initialization and Run-time Command Summary.....	92

1.1 PROJECT OVERVIEW

The advantage of input and output SPOOLing (Simultaneous Peripheral Operations On-Line) for increasing peripheral utilization and for virtualizing a limited number of physical devices for simultaneous sharing among several users has long been known [M1]. Typically these spoolers have shared a computer with other processing tasks and competed with these tasks for system resources through the services of the local operating system, and therefore have been limited in accessibility to users of the machine in which the spooler resides (see Figure 1). This can be adequate, but in a computing environment with multiple mincomputers it is not economically feasible to provide each computer with a full set of even basic peripheral devices. It is for this particular environment that this spooler has been developed.

This output spooler, written in the high level language Concurrent Pascal (CPascal), is intended to operate as a stand-alone program in a minicomputer dedicated to spooling operations only (see Figure 2). A high level language was chosen, in part, because of the readability of the finished code, and the ease of coding and maintenance when compared to a typical assembly language. More importantly, however, is the advantage of portability gained by programming in a non-machine dependent language [H1]. CPascal in particular

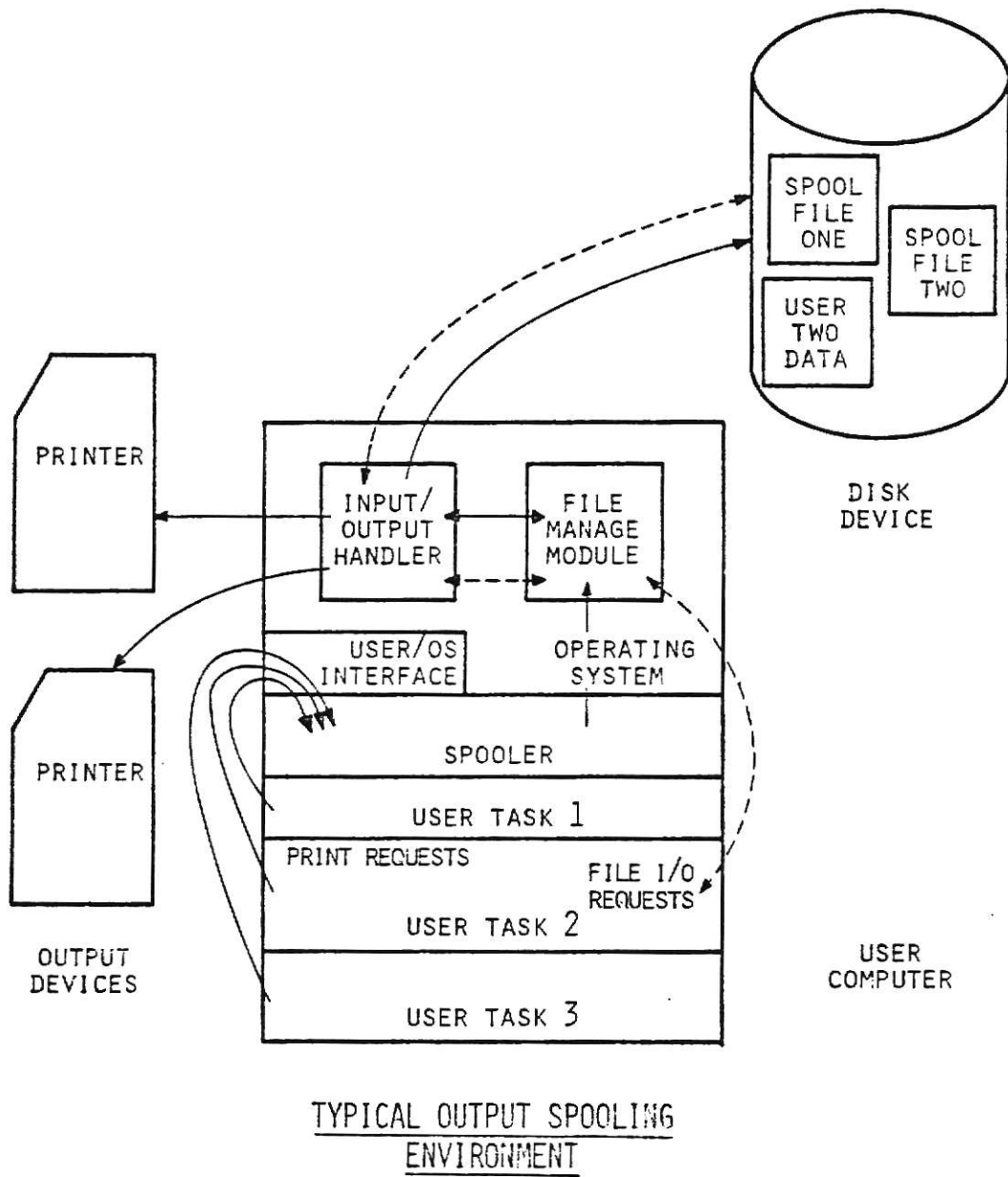


FIGURE 1.

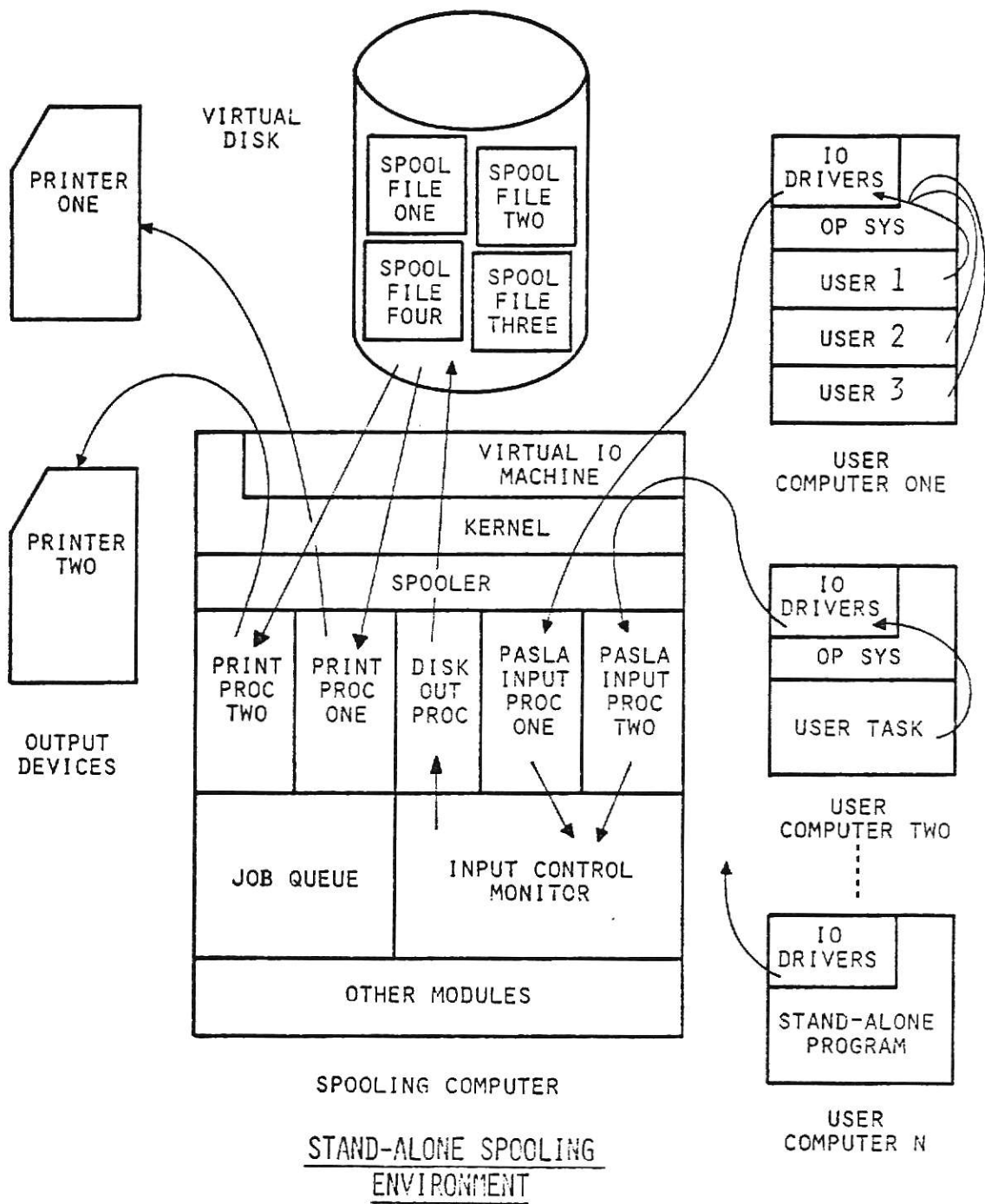


FIGURE 2.

lends itself to this application because it supports flexible data type definition, access right checking, concurrent processes, and data-sharing structures.

The general goal of the spooling program is to make any of a set of printers simultaneously accessible to a number of user computers. Other benefits especially important to smaller machines are realized as well. These include the off-loading of processing time, operating system overhead, disk spool space, and memory space used by each individual computer to support its own spooling systems.

There are other design objectives whose inclusion is necessary for reliable and convenient operation. First, the operator may optionally require the spooler to maintain an up-to-date copy of its control information on disk. This provision helps combat the effects of inevitable machine "crashes", particularly those in the spooling computer. If, in the user environment, print files are lengthy or are the product of hours of computing, it is desirable to expend the extra overhead to save the effort of regenerating the file, at the cost of additional time and money. With backup enabled, all files ready to print and currently printing are completely guaranteed against loss of data. If a malfunction does occur, or the system is shut down, upon initialization of the spooling program the operator may select whether to continue the spooler where it was, thereby keeping all spool files intact, or he

may completely reinitialize the spooler, effectively deleting all files and freeing all disk space. At the same time, he may select whether the newly reinitialized spooler is to now maintain disk backup of all control information.

Second, operator commands may be entered from the operator's console to pause or continue any selected printer, re-position a selected printer anywhere in its current spool file, delete any selected spool file, optionally discard spool files to a "null" printer, and to list the status of files in the system at the console.

Third, a user may, under the control of his program, select which printer(s) his file is to be printed upon, the number of copies to print, and whether the file is to be printed in IMAGE or FORMAT mode.

Finally, consideration is given to make the spooler configurable at compile time by altering program constants. These constants can vary the number of printer processes, the number of input processes, the amount of disk space to be used for spool files, and the size of control lists in the internal modules of the system.

1.2 DOCUMENT OUTLINE

The second chapter of this document is divided into seven sections. First, a brief description of the major system modules and their functions is presented. Second, a discussion of virtual disk requirements and the chosen

spool file structure is given. The third section contains a description of the information a user must supply to the spooler for proper operation. Section four contains the logic and structure of the Intermachine Communication Protocol used for transferring data between user and spooling computers. The next two sections describe, respectively, disk backup procedures and main memory usage. The last section of Chapter Two provides a scenario of typical spooler operation. It begins by opening a new file and tracing the file completely through the system.

Chapter Three of this document is a discussion of possible system enhancements and a summary of the project. Immediately following this summary are three appendices giving a detailed description of each program module, a bibliography, and a summary of initialization and run-time commands.

2.1 BRIEF MODULE DESCRIPTIONS

The spooler consists of two main passive modules which contain system data structures. These are the JOBQUEUE, and the INPUT_CONTROL_MONITOR (ICM). One other smaller passive module, the DISK_SCHEDULER, schedules all disk accesses according to the current priority of the requesting process. Active modules which operate upon these data structures are the Pasla Input Processes (PIP), the Printer Processes (PP), an Operator's Console Process (CP), a Memory Reclamation Process (MRP), a Message Posting Process (MPP), and a high priority Disk Output Process (DOP). Figure 3 is a complete access diagram showing both control and spool data flow information. General reference is made to this figure throughout the "Brief Module Description".

The JOBQUEUE is the heart of the system. It contains lists which completely describe the status of every file known to the spooler, whether it be open for input, awaiting print, currently printing, or under reclamation. All logic governing the movement of a file through the system resides in the JOBQUEUE. Each operator request is funneled through the JOBQUEUE and is either executed there or passed on to the appropriate process for execution.

The INPUT_CONTROL_MONITOR (ICM) assumes complete responsibility for the generation of spool files as they are received, buffer by buffer, from the user computers.

INITIALIZATION ACCESS — — — — —>
GENERAL ACCESS —————>
DATA FLOW —————>



Since the spooler is a stand-alone program, no file-handling capabilities are available from a local operating system and the ICM must supply its own. The structure of these files is discussed in section 2.2. The ICM receives data, sorts it by a four character identification code (ID) and blocks it up into 512-byte data pages while maintaining a separate pagemap for each opened file. If needed, it opens a new file with a unique filename or closes a file when an End-Of-Medium (ECM) is detected, and notifies the JOBQUEUE with the information it needs to maintain its control lists.

The Sector Map, a subsection of the ICM, controls the allocation and de-allocation of disk space used for spool files. When disk spool space is exhausted and an input process requests disk space, that process will be delayed in the Sector Map subsection until disk space becomes available.

The ICM also contains one other subsection which buffers any data to be written to the disk by the high priority Disk Output Process. This buffering is done in order to minimize the time a given process must tie up the ICM doing disk I/O. It also allows moving all disk I/O overhead outside the ICM. The net effect is to allow more processes to use the ICM in a given amount of time by decreasing the time they must spend there.

Active modules in the system perform various functions.

The Pasla Processes receive transmissions from external user computers and then send the received data and control information to the ICM for storage in the appropriate spool file. There is a separate process for each user computer serviced by the spooler. These processes are the only means of acquiring spooled data from an external source.

In a similar fashion there is one Printer Process for each printer device supported, and its sole function is to read through the spool file and print it.

The Memory Reclamation Process (MRP) releases all disk space used by a spool file when the spooler has successfully printed all requested copies.

The Console Process (CP) provides the means by which the operator may alter the normal operation of the spooler. All commands are parsed by the CP for correct syntax, and appropriate error messages are printed if an improper command is issued. When the system is initialized, this process acquires control of the console device, issues a read, and waits for an operator command. Therefore, in order to allow error messages to be posted by the Message Posting Process (MPP) at any time during operation without the pending read being satisfied, the CP releases control of the console device approximately once per second unless it is actually inputting a valid command line. This allows the MPP to acquire control of the console device long enough to write its message(s), and the processes posting

messages will not deadlock the system by being indefinitely delayed. The MPP has a line buffer through which it is made accessible to any process in the system wanting something written to the console.

The Disk Output Process (DOP) always uses the disk at the highest priority. Its sole function is to write data to disk as quickly as possible. The DOP is concerned with data written from the input section of the spooler and is accessible only through the special buffer handling subsection of the ICM.

2.2 VIRTUAL DISK SPACE and FILE STRUCTURE

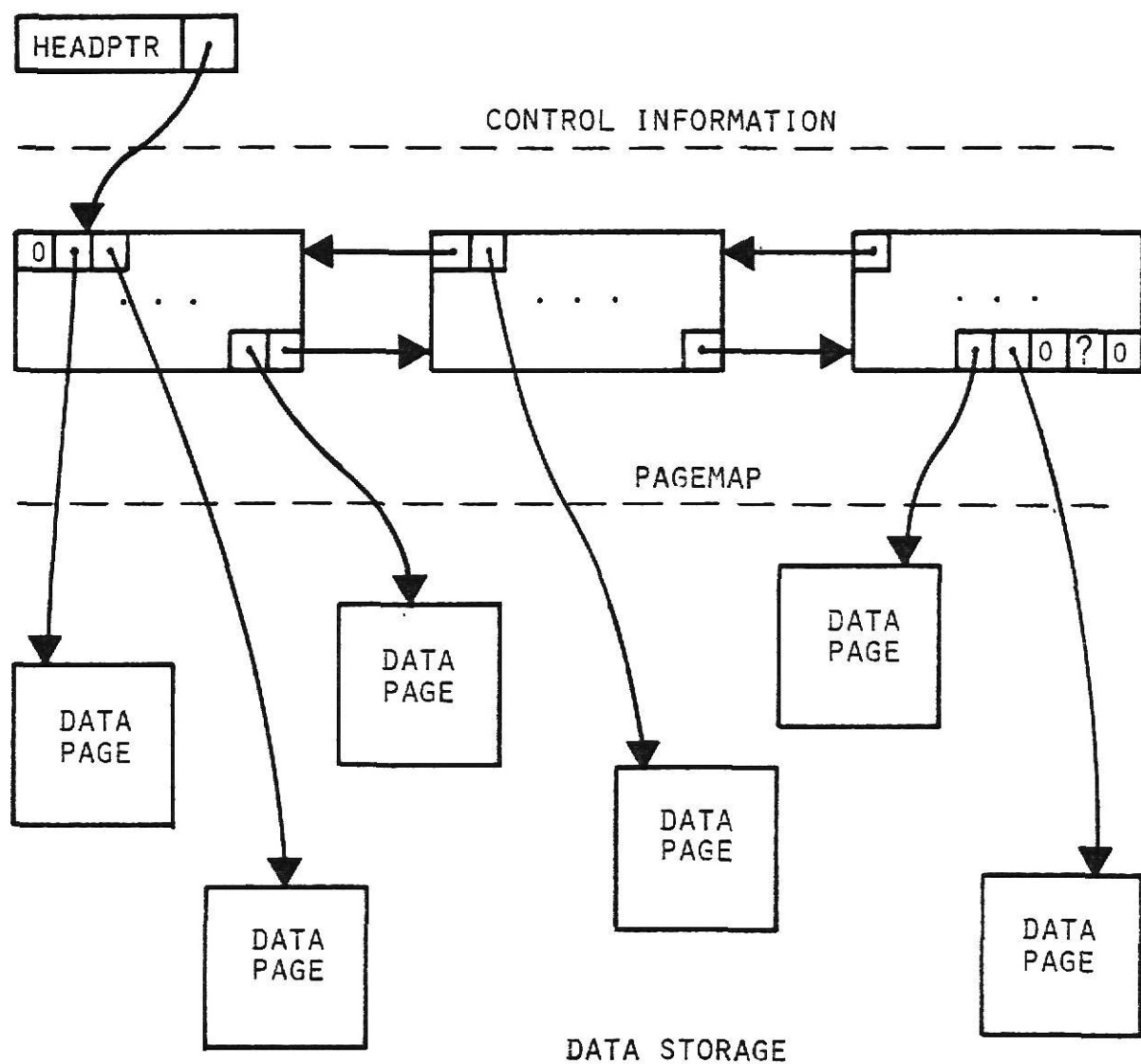
The disk space used by the spooler is a variable length virtual disk allocated under Interdata's OS/16 MT2 operating system as a contiguous disk file [I1,I2]. This is done to keep the disk compatible with other uses not necessarily spooler-related and to keep from tying up disk space unnecessarily. In this way, the smallest possible disk file may be allocated (with corresponding changes to constants within the spooler) using as its filename SPOOLER.VD. The spooler divides the available space into 512-byte pages of storage and reserves approximately seven of these pages for control information backup. The remainder will be used to build spool files.

In order to facilitate moving forward and backward through the spool file without an excessive amount of disk

activity, the ICM builds a randomly accessible file (see Figure 4). The pagemap of a spool file is a doubly linked list consisting of any number of pages. This allows a file to be created that uses all of the available disk spool space. Each page of the pagemap contains two pointer fields used for pagemap linkage and 254 fields used as pointers into the spooler's disk spool space. Actual data resides in the 512-byte disk pages pointed to by these fields. Incoming data buffers of varying length are blocked up in the ICM for storage in these disk pages so that all 512 bytes of each page are filled. With this file structure, only the address of the first page of the pagemap (HEADPTR) need be kept in any particular spooler control list. The rest of the file resides on disk and is accessed beginning with just this information.

2.3 USER-SUPPLIED INFORMATION

The only information the JOBQUEUE knows about a file upon opening it for input is the beginning address of its pagemap and its unique ID. As a file is built, the ICM accumulates part of the file's parameters (its length, for example) for use by the JOBQUEUE. The remainder, User-Supplied Information, is acquired in the following fashion. With each transmission of data from the user computer to the spooling computer, a header segment is added to the front of the data buffer segment (see Figure



FILE ORGANIZATION

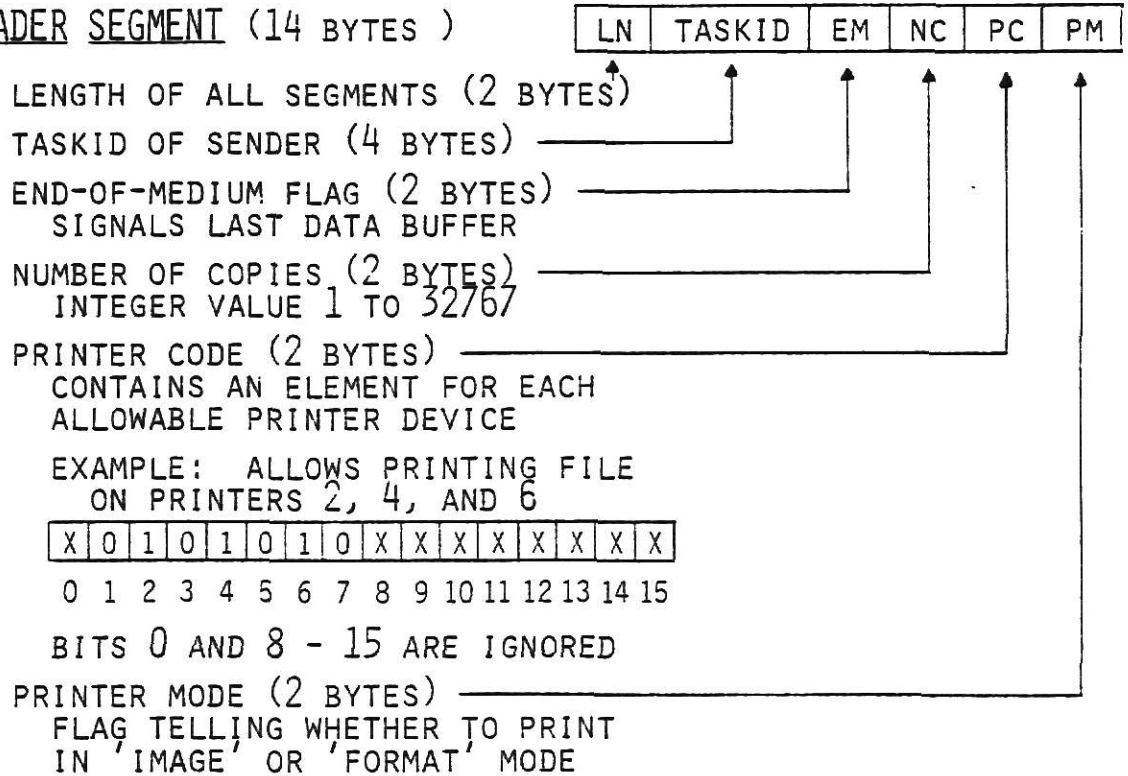
FIGURE 4.

5). This header contains control information which allows the Pasla driver routine to check for correct reception of transmitted data, as well as the four character ID which the ICM uses to direct input data to the correct spool file. In addition, an EOM flag, the number of copies to print, which printer(s) to use, and the print mode are all contained in the header segment. The last three of these fields need be valid only when the EOM flag is true. They are ignored by the spooler at all other times. When the EOM is signaled as TRUE, the file is closed out of the ICM. Then control information gathered in the ICM and User-Supplied Information obtained from the transmission header of the Pasla Input Process are forwarded to the JOBQUEUE.

2.4 INTERMACHINE COMMUNICATION PROTOCOL

The function of the Intermachine Communication Protocol (ICP) is to control the correct exchange of four segments of information: the header, the data buffer, the checksum, and a positive or negative acknowledgement. This protocol is implemented in the Pasla Driver Class (PASXMTRECCL) which controls a hardware device call a Programmable Asynchronous Single Line Adapter (PASLA) [13]. A Pasla attached to the sending computer is wired for full-duplex communication with another Pasla attached to the spooling computer.

HEADER SEGMENT (14 BYTES)



DATA BUFFER SEGMENT (1 TO 512 BYTES)

CHECKSUM SEGMENT (1 BYTE)

CHECKSUM MODULO 256 CALCULATED OVER EVERY BYTE
OF THE HEADER, DATA BUFFER, AND CHECKSUM SEGMENTS

USER-SUPPLIED INFORMATION

FIGURE 5.

The ICP is really a half-duplex protocol but it was thought better to use the Pasla in this mode than to add the additional complexity of line contention to the protocol. Since data will flow predominantly in one direction, there is no need to support simultaneous sending and receiving.

The Pasla is capable of a number of speeds which can be program selected. Currently the devices are set at a speed of 9600 baud, but that may be doubled to a maximum rate of 19,200 baud. This means that, at 9600 baud, it takes approximately 8.8 seconds to transmit a 10K-byte data file, using 512 bytes per data buffer segment, and including the extra transmission time for header, checksum, and positive acknowledgement segments.

The protocol exchanges four separate segments. Three of these, the header, the data buffer, and the checksum, are transmitted by the user computer to the spooling computer. The spooling computer then checks to see if the data received is correct based on the received checksum. If it is, a positive acknowledgement (ACK) is returned to the user computer and the exchange is complete. If it isn't, then a negative acknowledgement (NAK) is returned and the computers retry the transmission.

As it turns out, the Pasla is not a particularly dependable device. It appears to return a status code of zero (OK) randomly when in actuality an overrun has

occurred and a character is lost. This causes problems since return from an I/O command is based either upon reception or transmission of the specified number of bytes, or upon a return error status of non-zero from the Pasla. In our case, for example, when the receiver is receiving fourteen bytes, but there is an overrun and one of these is lost, leaving the receiver waiting for the fourteenth byte, the transmitter either goes ahead with the next segment or waits to receive the acknowledgement, depending upon which segment is being exchanged when the error occurs. This either deadlocks the Paslas (leaving them both waiting indefinitely to receive) or throws them out of synchronization with each other (for example, transmitting a checksum, but receiving it as part of the data buffer). Because of these unpredictable situations, a timeout feature was added to the Kernel's Virtual I/O Machine [C3] to allow the protocol to abort the transmission or reception of any segment if it does not complete within a specified period of time. When a timeout occurs, the protocol assumes something went wrong and the condition is not reflected properly by the Pasla Hardware.

Timeout values exist only for the data buffer and checksum segment and not for the header and ACK/NAK segments. The computer will wait indefinitely when receiving the header (i.e. the start of a new transmission) or the ACK/NAK segment. This is done for the following

reason. If any error is detected while receiving a segment, the whole transmission is a failure. The receiving computer then skips over any remaining segments and goes immediately to the ACK/NAK response. This approach forces synchronization at the beginning and ending of a transmission. It also wastes the least amount of processing overhead when bad segments are received and guarantees recovery from any error situation (short of catastrophic Pasma hardware failure). It has proven itself effective by allowing the operator to shut one machine off, repower it, initialize the system, and have the Pasmals continue their communication. While this is not always the case, the protocol does not fail when one or both machine(s) are halted in random order and then have their execution continued.

2.5 DISK BACKUP PROCEDURE AND PRIORITIES

The disk is the single most critical performance bottleneck within the actual spooler itself. Because of this, the backup of all control information on disk is an operator option selectable at initialization time. If this option has not been selected and the system fails for some reason, making reinitialization necessary, then the system must either be completely initialized leaving no files in existence and all disk space free, or partially initialized setting the spooler to the current state of the backup

control pages stored on disk (unknown at this point).

Using a 10K-byte data file as an example, without the backup option specified and requesting only one printed copy, the spooler will make forty-three disk accesses from the time the file is opened to the time the system has completely flushed the file from its lists. If disk backup has been selected, this number rises to 97. Table 1 shows how these accesses spread themselves during different phases of a spool file's existence and which spooler modules are responsible for them.

From Table 2 it can be seen that when the system is backing itself up on disk, the input section of the spooler (ICM) does approximately 72% of the disk I/O, the JOBQUEUE does approximately 8.5%, the Printer Process does approximately 17.9%, and the Memory Reclamation Process does the remaining .85%. Without the backup option, the percentages are less lopsided. Each of the ICM and Printer Process do about 48.8% and the MRP does the remaining 2.3%. The JOBQUEUE does no I/O. Note also that the disk accesses are not necessarily spread evenly across the spool file's time in existence (see Figures 6 & 7).

Therefore, in order to minimize delays of input processes due to disk I/O, and to relieve as much of the bottleneck on the input side where it is most critical, the following two measures have been taken.

First, the Disk Scheduler Monitor schedules all disk

	INPUT CONTROL MONITOR	JOB QUEUE	PRINTER PROCESS	MEMORY RECLAMATION PROCESS	SUB TOTAL
	REQ'D	BACKUP	REQ'D	BACKUP	REQ'D
OPENING A FILE	1	5	---	---	1
BUILDING	18	36	---	---	18
CLOSING & ENTERING	2	2	---	---	2
STARTING PRINT	---	---	1	---	1
PRINTING	---	---	20	---	20
RECLAMATION	---	1	---	1	1
TOTAL	21	44	21	---	43
					54

TABLE 1

	INPUT CONTROL MONITOR	JOB QUEUE	PRINTER PROCESS	MEMORY RECLAMATION PROCESS	TOTAL
WITHOUT BACKUP	21	---	21	1	43
WITH BACKUP	65	10	21	1	97

TABLE 2

DISK ACCESSSES vs. TIME (w/o DISK BUFFERING)

WITH DISK BACKUP

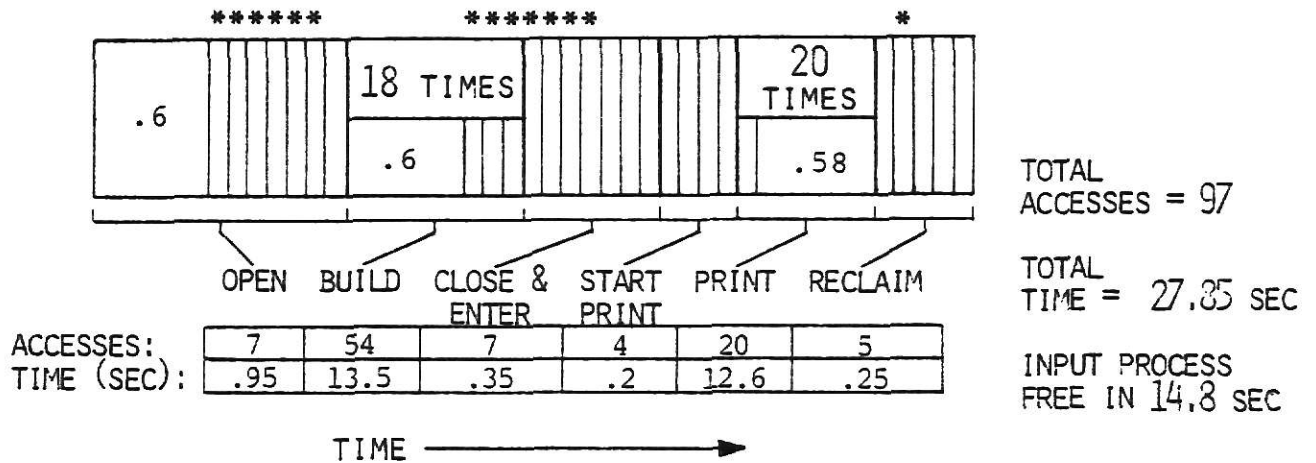


FIGURE 6.

* - ICM DISK ACCESSSES

WITHOUT DISK BACKUP

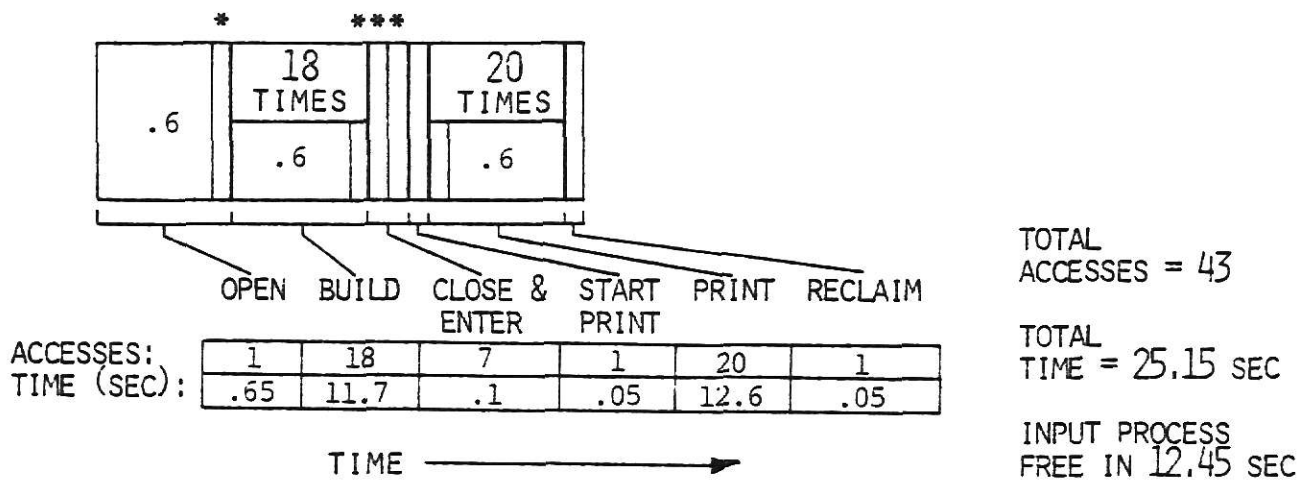


FIGURE 7.

requests using a simple priority scheme. Each priority has its own FIFO of queues (if more than one possible process may be waiting at once with that priority) or a single queue variable. When any process requests the disk, if the disk is not in use, control is immediately given to that process. If however, the disk is currently in use, the process is delayed on the appropriate priority queue. When the current disk user returns control of the disk to the scheduler, it is immediately rescheduled to the longest waiter on the highest priority queue. For example, the priorities ranging from highest to lowest are INPUT_PRI, JCBQ_PRI, PRINT_PRI, and GARB_PRI. Therefore, if the MRP requests the disk at GARB_PRI after two input processes have requested the disk as INPUT_PRI, the MRP will not receive control of the disk until both input processes have finished -- regardless of the time it must wait in its queue.

Table 3 gives a summary of possible processes requesting each priority and the functions they perform. Note that any given process may use the disk at different priorities depending upon which module of the spooler it requests the disk from.

The second method to minimize input process delays is the use of a simple buffering scheme contained within the ICM itself. Whenever a process wishes to write a page to disk, it fills the DOP's buffer in the ICM and requests the

DISK USER PRIORITIES AND REASONS

INPUT PRIORITY

DISK OUTPUT PROCESS - WRITING DATA TO DISK FROM ANY
PROCESS WHICH USES THE ICM

PASLA INPUT PROCESS - BACKING UP CHANGED JOBQUEUE
CONTROL LISTS

JOBQUEUE PRIORITY

CONSOLE PROCESS - BACKING UP CHANGED JOBQUEUE CONTROL
LISTS

PRINTER PROCESS - (SAME)

MEMORY RECLAMATION PROCESS - (SAME)

PRINTING PRIORITY

PRINTER PROCESS - READING PAGEMAPS AND DATA PAGES

MEMORY RECLAMATION PRIORITY

MEMORY RECLAMATION PROCESS - READING PAGEMAPS FOR
SUBMISSION TO THE ICM FOR RECLAMATION

TABLE 3

DOP to service it by calling the Disk Scheduler. The DOP services this request in one of two modes -- 'I/O and Proceed' where a buffer is deposited and the process immediately continues, or 'I/O and Wait' where the process deposits its buffer and is delayed until its I/O is complete. Under normal conditions, there is no need for the requesting process to be delayed for the length of a disk write as when it is responsible for its own disk I/O -- it is free to continue processing immediately after depositing its buffer and therefore requests the 'I/O and Proceed' option. However, the 'I/O and Wait' option is exercised under certain conditions such as when closing a file, for example, to ensure that the disk contains a valid pagemap and valid data pages before the file is scheduled for print and information is retrieved from the disk by a printer process.

Regardless of the option selected, the next process to use the DOP's buffer will be delayed if the buffer is full, or will deposit its page using one of the options described and then lodge its request with the Disk Scheduler. Once the DOP has been continued, it copies the buffer from the ICM to its own buffer space. Governed by the option selected, the DOP eventually marks the ICM's buffer as free and continues the delayed processes. By moving the disk I/O function to the DOP it is possible to reduce the 100 milliseconds needed for two consecutive disk writes to

essentially the time needed to copy two data pages (on the order of 200 microseconds). This much time is saved on behalf of the process using the ICM, speeding up the use of the ICM and making it free a greater percentage of the time. Compare figures 6, 7, 8, and 9.

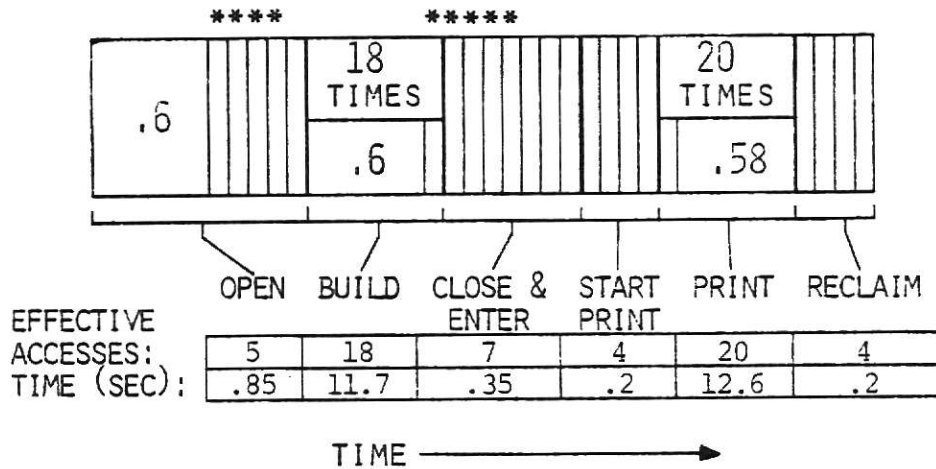
2.6 MEMORY USAGE

This section is included to show the distribution of main memory among the various program functions in the spooling computer. The memory map in Figure 10 shows approximate sizes of different modules as well as their order of occurrence in memory. Some of these values do not change and some vary with adjustment of program constants. For example, the number of buffers used for open files or the number of occurrences of a process' data and stack space will vary as the spooler is tailored to meet various operating environments. Entries in the figure that are marked with an asterisk are those which vary.

Memory is divided into three major sections. The first is a machine-dependent interface (the Kernel) which provides support for CPascal language primitives and other needed features on a particular machine. This support includes main memory allocation and de-allocation, system initialization and error handling, time slicing, interrupt handling, a real-time clock, system and language libraries, and a Virtual I/O Machine to execute device drivers written

DISK ACCESSES vs. TIME (WITH DISK BUFFERING)

WITH DISK BACKUP



TOTAL EFFECTIVE
ACCESSSES = 58

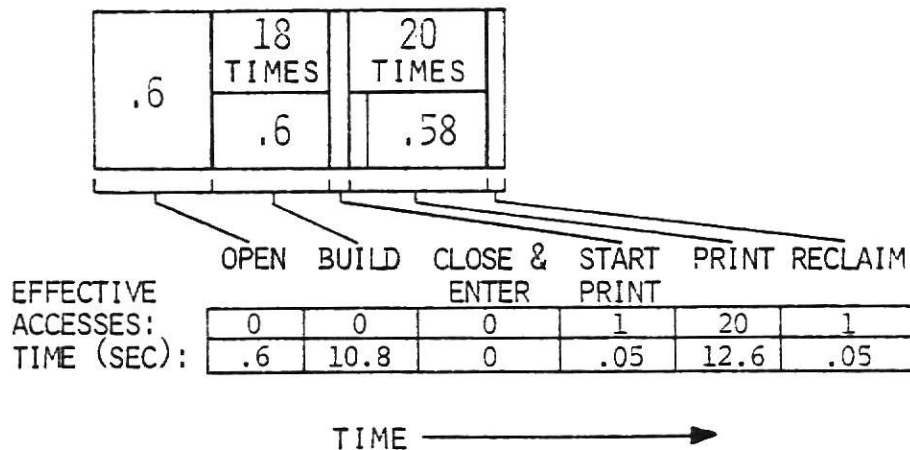
TOTAL
TIME = 25.9 SEC

INPUT PROCESS
FREE IN 12.9 SEC

FIGURE 8.

* - ICM DISK
ACCESSSES

WITHOUT DISK BACKUP



TOTAL EFFECTIVE
ACCESSSES = 22

TOTAL
TIME = 24.1 SEC

INPUT PROCESS
FREE IN 11.4 SEC

FIGURE 9.

KERNEL - 7.3k	LANGUAGE SUPPORT - 3.0k
	VIRTUAL IO MACHINE - 2.6k
	LIBRARY ROUTINES - 1.7k
	LINKAGE - .13k
SPOOLER CODE SPACE - 28.31k	CONSOLE, DISK, PASLA IO DRIVERS & ASSOCIATED CODE - 8.07k
	JOB QUEUE - 9.27k
	INPUT CONTROL MONITOR - 4.6k
	SYSTEM PROCESSES - 4.53k
	INIT CODE - .55k
	CONSTANTS - 1.11k
	MONITOR & CLASS DATA SPACE - 11.25k
STACK & DATA SPACE - 18.83k	DOP DATA SPACE - .62k
	MPP DATA SPACE - .69k
	INPUT PROC DATA SPACE - .98k
	PRINT PROC DATA SPACE - 1.79k
	PRINT PROC DATA SPACE - 1.79k
	MRP DATA SPACE - .71k
	CONSOLE PROC DATA SPACE - .94k

MEMORY USAGE

FIGURE 10.

in the CPascal program. The second section is the actual code space of the spooler. Code generated by the compilers is re-entrant and therefore its space requirements remain constant regardless of the number of times an instance of a given type is declared (a Printer Process, for example). What does change with the number of instances of a type, however, is the third section - stack and data space. Note that though there is only one instance of the re-entrant code for the Printer Process, since the spooler supports two printers (each with its own process), there are two instances of Printer Process stack and data space areas. Each instance of a process has its own stack and data space.

2.7 SCENARIO OF OPERATION

The scenario is a pictorial and verbal explanation of the operation of the spooler as it processes a buffer of data. Steps through the discrete phases of the spool file are presented, beginning with the opening of a new file, followed by scheduling and printing, and finally reclamation.

2.7.1 OPENING A NEW FILE (see Figure 11):

When a Pasla Input Process receives a block of data (A), it is submitted to the ICM for processing. Since there is not a file currently open for input using the ID

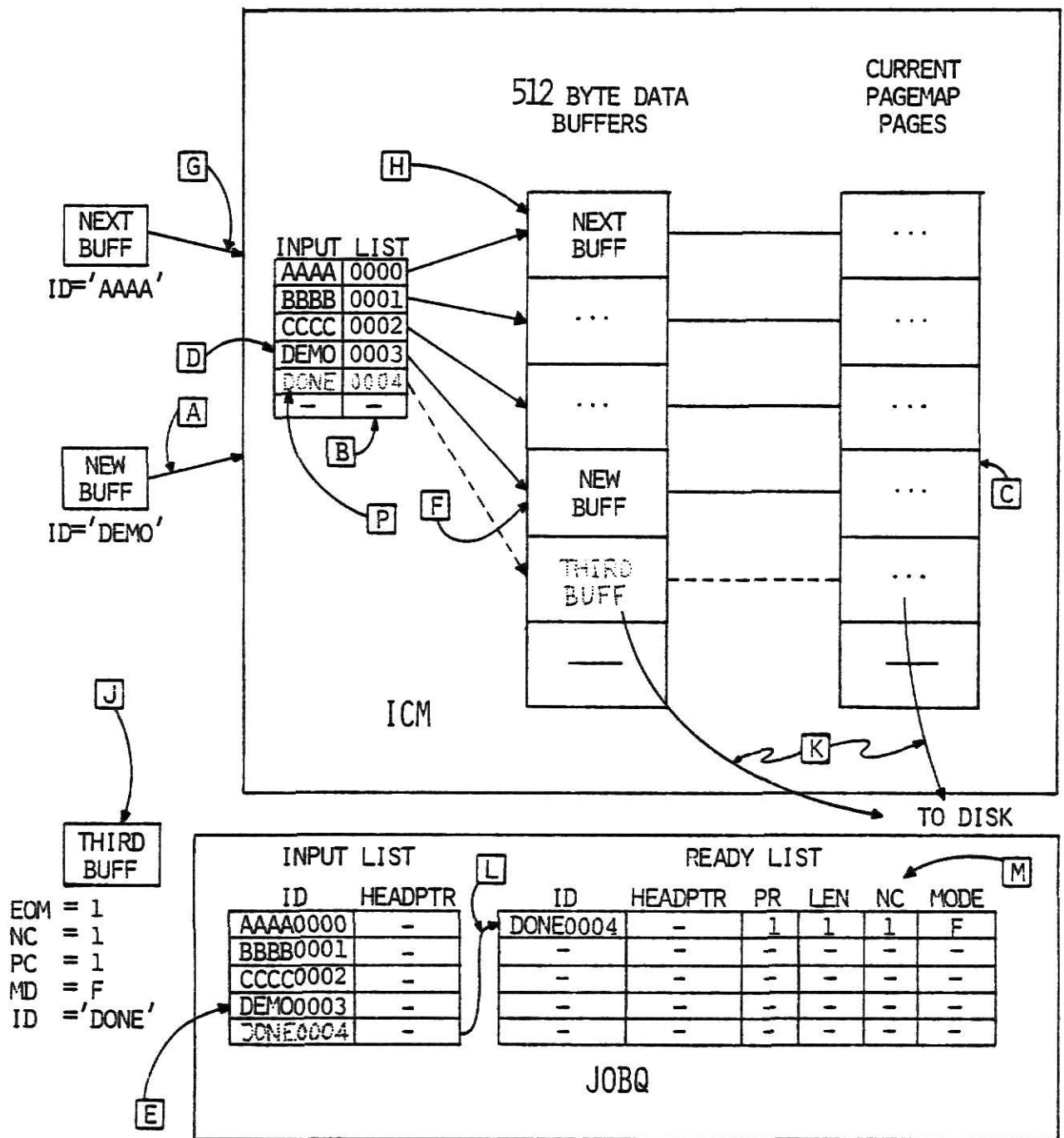


FIGURE 11.

of 'DEMO', the ICM checks and finds three spaces on its input list for new files to be opened (B). The Pasla Input Process is returned a boolean value TRUE stating that room was found and the data was accepted by the spooler. The process then takes whatever action is defined in the Intermachine Communication Protocol. Since the data was accepted, an empty pagemap is created (C), the ID of the sender is concatenated to a four-digit number to make a unique filename, and entries are made in both the ICM's and the JOBQUEUE's input lists (D,E), thus initializing the list values and marking the file as open for input. Finally the input data buffer is copied to the ICM's data buffer (F).

2.7.2 ADDING DATA TO AN EXISTING FILE (see Figure 11):

Every time a new buffer of data is received from a remote computer (G), the ICM checks to see if the sending ID has a spool file currently open. If it does, the buffer is blocked up with data previously received for that particular ID (H). If necessary, buffers and/or pagemaps are written to disk when completely filled and new ones are begun in the ICM with the data that would not previously fit. These actions take place whether or not the disk backup option is active.

2.7.3 CLOSING AN OPEN INPUT FILE (see Figure 11):

When an EOM is detected by the remote computer and it sets the header's EOM flag on the transmission of the last data buffer to the spooler (J), the data is handled as specified in the previous section with the following additional actions. All data in the updated pagemap is flushed from the ICM's buffers to the spool file on disk (K). The Pasla Process initiates the transfer of the file's entry from the JOBQUEUE'S input list to the JOBQUEUE'S ready list (L), and at the same time supplies the JOBQUEUE with additional information accumulated by the ICM and the User-Supplied Information taken from the last transmission header (M). Then all references to the file are closed out of both the JOBQUEUE'S input list and the ICM's input list (P). This empties all input list space, buffers, and pagemap pages in these modules for another file.

2.7.4 SCHEDULING A FILE FOR PRINT (see Figure 12):

Once a file is on the ready list it waits for printing until a user-specified printer is idle (in this example, printer P1). At that time, the printer has a copy of the file's pertinent information transferred from the ready list to its printing list entry (A). Once a file is scheduled for printing, if an additional copy is to be printed, the entry is left in the ready list (with its

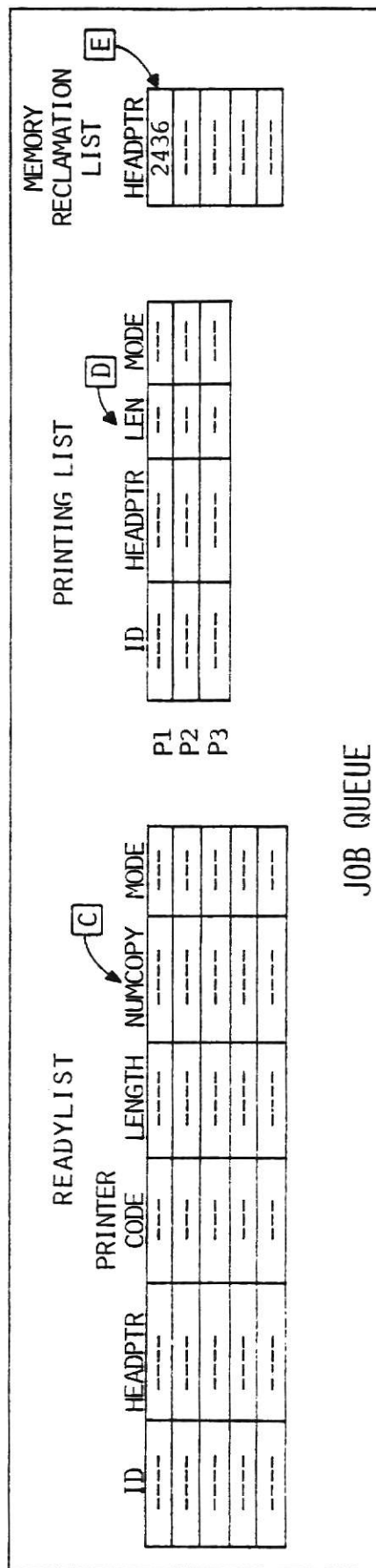
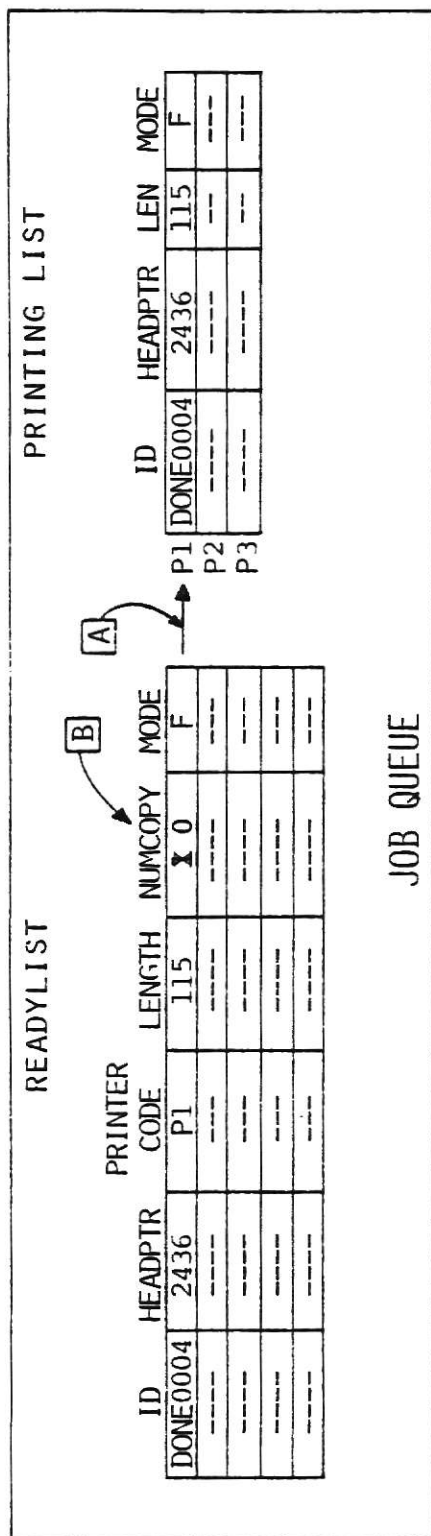


FIGURE 12.

'NUMCOPY' count decremented) so that the next available printer can work on this file as well, thus allowing multiple copies to print simultaneously. When the last printer is assigned the last copy of a file, 'NUMCOPY' goes to zero (B), and the entry is removed from the ready list, leaving it empty (C). As soon as that printer has finished it notifies the JOBQUEUE, its entry is removed from the printing list, leaving it empty (D), and the file is moved to the Memory Reclamation list for the collection and freeing of used disk pages (E). Print scheduling is currently based on a simple first-come first-served basis. However, as can be seen in the scenario figures, information is available in the JOBQUEUE's ready list to schedule printing according to the printer(s) chosen, length of the file, number of copies desired, ID of the user, or some combination of these.

2.7.5 PRINTER PROCESS ACTION (no Figure):

Upon operator initialization of the system, each printer process immediately requests a file to print. Once a copy of a file is assigned, the printer process reads and formats the file, printing it line by line. For each page of printed data, the printer process checks with the JOBQUEUE once to see if the page is to be printed or discarded as selected by an operator option. At the same time any operator command applicable to that printer

process is received from the JOBQUEUE. If there is no command, then printing continues normally. If a command to go forward or backward in the file or to rewind to the beginning of the file is given, the process repositions itself and continues printing. If the command to purge the current file is given, then an End-Of-File (EOF) is simulated and normal termination takes place. When done printing the current file, the process notifies the JOBQUEUE so JOBQUEUE lists can be updated, and then requests another file to print.

2.7.6 MEMORY RECLAMATION (see Figure 12):

The Memory Reclamation Process (MRP) is active as long as there are files to be collected (E). It reads a page of the pagemap, beginning with the 'HEADPTR', and submits that page to the ICM. The ICM then releases every data page listed in that particular page of the pagemap and releases that page as well. This continues until all data pages and the entire pagemap have been freed. When the MRP is finished, it notifies the JOBQUEUE of its completion and the pointer to the file's pagemap is deleted from the MRP's control list, leaving the list in the example empty.

3.1 PROJECT SUMMARY

This spooler has been under off-again/on-again development for about one year and has progressed through several distinct phases.

In the beginning, about a month was spent on the original design and coding. As originally implemented, the spooler executed on an Interdata 8/32, the Computer Science Department's main research computer, under a virtual Pascal machine designed to execute code generated by Hartmann's CPascal and SPascal compilers [B1,H2]. Since the 8/32 had printer devices and fast terminals attached to it, and was large enough to run the compilers, it was the natural starting place for initial development. Once a version was running correctly on the 8/32, the transition was made to a 16-bit machine - the target architecture.

With the transition from the 32-bit to the 16-bit architecture, a completely different I/O structure had to be overcome. Since all I/O used the Virtual I/O Machine (VIM) in the Kernel, drivers for the disk, printer, and console had to be added in the concurrent code, because these were not supplied in the 16-bit stand-alone environment. This forced the addition of approximately 1000 lines of code to the final version of the spooler.

Once these changes were accomplished, facilities for communication with external computers were added. Up to this point, the spooler had generated its own spool files

for testing purposes. The first step was to develop a suitable protocol using Paslas as the hardware devices for transmission. Without a doubt, this was the most frustrating portion of the entire project. At least two months were spent trying to determine from Kernel interrupt traces, core dumps and the like, why communication would arbitrarily fail. Once the few software logic errors were removed, it became apparent the Pasla device itself was unpredictable. Since there was no clean way to circumvent the problem of inaccurate error statuses, the Kernel's VIM was modified to allow setting timeout values, in tenth-second increments, on any chosen input or output device. This addition allowed us to obtain reliable service from the Pasla hardware.

Concurrently with the Pasla protocol implementation, the spooler was modified to return all received data to the sender for checking. We were then able to use the system to check the accuracy of the spooler and communication link operation. Using this ability, another software driver was added to allow bi-directional communication between the spooler and user computer over the University's KSUBUS (a direct memory access data mover developed at KSU) [C1,C2]. This allowed demonstrating the speed of the KSUBUS in a real life situation and allowed timing the relative speeds of the spooler using the Paslas for input in one version and the KSUBUS for input in the other.

When the demonstration was complete, the spooler was modified again to conform to the original design. All of the unnecessary functions were removed. Then, as it was more carefully scrutinized, final modifications were made to the structure of the spooler in order to increase its speed of operation and minimize its computer memory requirements. Speed increases were realized by adding the Disk Output Process and its buffers, prioritizing disk usage, and making disk backup optional. Memory requirements were reduced by adding the Message Posting Process and minimizing the occurrences of disk classes and TERMINAL classes, thus yielding the current version of the spooler as described in this document.

3.2 PROPOSED ENHANCEMENTS

There are three proposed enhancements that are worthy of mention as useful improvements.

The first is changing the mode of communication between user computer and the spooler. The current method using Paslas as the input medium is quite slow and the overhead of handling Pasla interrupts on a byte-by-byte basis taxes the VIM structure. As has been proven in operation, a Direct Memory Access (DMA) method of communication such as the KSUBUS is far superior. In practice, we found that to transmit a reasonably sized data file (58.5 K-bytes) over Paslas took approximately 2 1/2 minutes, while the same

file took less than 20 seconds using the KSUBUS. This saved about 90% of the transmission time. Since these files were effectively transmitted twice, once going to the spooler and once returning from the spooler to the sending computer, these times may be divided by two, yielding roughly a 10-second transmission time. Considering the final restructuring of the spooler (not implemented when the KSUBUS testing was done), the 10-second transmission time should drop even more because of the increased speed of the spooler's input section.

A second enhancement directly related to the first, and made more necessary by it, is the reduction of disk delays. The most reasonable proposition seems to be a number of smaller dedicated disks to replace the single disk currently in use. Though not actually measured, the specifications of the current disk device claim an average access time of 50 milliseconds (the time used in all calculations). With multiple disk devices, the average time to use the disk would be inversely proportional to the number of units the spooler has at its disposal. This would reduce the single greatest bottleneck within the spooler itself.

The third enhancement is proposed from the vantage point of experience. In the current implementation of the 16-bit Kernel, as has already been stated, all I/O is handled by a Virtual I/O Machine which executes virtual

instructions coded into the high level CPascal program. The major complaint waged against the VIM is that of wasting space for the gains derived. Lack of space became a problem because of the limited 64K-byte address space of our implementation (typical to many small computers). The VIM consumes about 2.6K (30%) of the Kernel's and library's 8.9K of code space. In the spooler's concurrent code, about 7.9K, out of the 28K used, is devoted to basic I/O support for the disk, terminal, and printer. This code could be efficiently moved to the Kernel itself, yielding a good space savings. Since drivers for a wide range of devices already exist for most machines in some operating system written for them, it becomes a reasonable matter to extract this code and implement the drivers directly in the Kernel and do away with the VIM. By using this scheme, the 10.5K used for I/O support in the current implementation, could be reduced to approximately 3K and still yield the same capabilities. These sized are calculated using actual drivers taken from Interdata's OS/16 MT2 device driver libraries [11,12].

Original advantages of abstracting the I/O driver from the real hardware just did not seem to surface. As much effort was required to make the abstract drivers work as if they had actually been coded in the machine's assembly language. In fact, at times, more work was involved in trying to make the VIM instructions do what could have been

accomplished more easily with assembly language instructions. Since an understanding of the assembly language level of operation was involved in the writing of our I/O drivers, with I/O being as intimately tied to the hardware level as it is, using the VIM only turned out to be an additional step and an unnecessary abstraction.

Other reasons made the VIM even more unattractive in our particular application which, in themselves, could not be held against the VIM structure. These mainly involved large compile times for the CPascal code vs. shorter assembly times for the Kernel code. Apart from this, however, when considering the relatively slow rate of change of the low level of software, the knowledge necessary to write effectively at this level (whether in a high level language or not), the hardware idiosyncrasies that work their way up even into the "portable" high level code, and the additional driver code space required in each program that uses this configuration, this enhancement seems a good one to implement.

APPENDIX 1

Bibliography

- [B1] Ball, M., PASCAL Compiler for the Univac 1110, Pascal Newsletter, No. 6, Implementation Notes, pg. 89, November 1976.
- [C1] Calhoun, The KSUBUS, A Functional Description, CS 78-01, January 1978.
- [C2] Calhoun, The KSUBUS, A Detailed Description, CS 78-02, January 1978.
- [C3] Cottel, D. and Zaun, J., Concurrent PASCAL Interdata 7/16 Computer, Preliminary Releases, Naval Undersea Center, December 1976.
- [H1] Brinch Hansen, Per, The Architecture of Concurrent Programs, New Jersey: Prentice-Hall, Inc., 1977.
- [H2] Hartmann, Alfred C., A Concurrent PASCAL Compiler for Minicomputers, PhD Thesis, Pasadena, California: California Institute of Technology, 1976.
- [I1] Interdata, Subsidiary of Perkin-Elmer, OS/16 MT2 Operator's Manual, Publication No. 29-430, Oceanport, New Jersey, September 1975.
- [I2] Interdata, Subsidiary of Perkin-Elmer, OS/16 MT2 Programmer's Reference Manual, Publication No. 29-429, Oceanport New Jersey, September 1975.
- [I3] Interdata, Subsidiary of Perkin-Elmer, Programmable Asynchronous Single Line Adapter (PASLA) Programming Manual, Publication No. 29-446, Oceanport New Jersey, June 1975.
- [M1] Madnick, S. and Donovan, J., Operating Systems, Computer Science Series, McGraw-Hill Book Company, Inc., 1974.

APPENDIX 2

Detailed Module Descriptions

2.1	Constants.....	46
2.2	Types.....	51
2.3	Classes.....	58
2.4	Monitors.....	68
2.5	Processes.....	85

APPENDIX 2.1:
CONSTANT DESCRIPTIONS

LINELENGTH = 132;

LINELENGTH is the number of characters (bytes) in a data line.

PAGELength = 512;

PAGELength is the number of characters (bytes) in a data page.

ENDMED = 1;

ENDMED is the index of the End-Of-Medium flag (EOM) in the array FUNC of the Intermachine Communication Protocol (ICP) header segment.

NUMCOPY = 2;

NUMCOPY is the index of the 'number of copies to print' in the array FUNC of the Intermachine Communication Protocol header segment array.

PRCODE = 3;

PRCODE is the index of the printer code in the array FUNC of the Intermachine Communication Protocol header segment.

PRMODE = 4;

PRMODE is the index of the mode of printing in the array FUNC of the Intermachine Communication Protocol header segment.

NUM_PRINTER_PROCESSES = n;

This entry is used to to define system variables throughout the spooler that depend upon the number of Printer Processes (i.e. arrays of queues). The value 'n' must equal the number of Printer Processes in the spooler.

INPUT_LIST_LEN = n;

Controls the maximum number of files to be open at any given instant. INPUT_LIST_LEN may never be larger than 32. For every possible open file, 1K of buffer space will be reserved by the INPUT_CONTROL_MONITOR for blocking and pagemap storage.

READY_LIST_LEN = n;

Controls the number of files that may await printing at any given instant. The maximum value of READY_LIST_LEN is 32. Changing its value does not effect the size of the spooler.

GARBAGE_LIST_LEN = n;

Controls the number of files that may await reclamation by the Memory Reclamation Process at any given instant. Its maximum value is 32 and it does not change the size of the spooler as it is varied.

PRINT_LIST_LEN = NUM_PRINTER_PROCESSES;

The length of the Printing list depends upon the number of Printer Processes in the system. The maximum value is 15.

IPAGELength = 512;

IPAGELength designates the number of characters to be kept in the INPUT_CONTROL_MONITOR's data buffer pages. This many characters will be read into the buffer before it becomes full and is written to disk. Maximum value is 512.

PGMAPLENGTH = 256;

PGMAPLENGTH controls the number of pointer entries in a page of the pagemap. The maximum is 256.

NUMPROCESSES = n;

This is the total number of Printer Processes, Pasla Input Processes or other types of input processes, plus the Memory Reclamation Process and the Console Process. It

does not include the Disk Output Process or the Message Posting Process.

NUM_PASLA_PROCESSES = n;

NUM_PASLA_PROCESSES is the total number of any type of input processes.

PAS_PLUS2 = "NUM_PASLA_PROCESSES + 2 =" n;

PAS_PLUS2 is used in variable declarations where the number of queues, for example, must vary according to the number of input processes and must allow for two other processes that always exist.

ALL_PROCESSES = n;

ALL_PROCESSES is the total number of processes in the entire system. This does not include the Kernel's 'Initial Process' which disappears once system initialization is complete.

IMAGE = 1;

IMAGE is one possible value supplied in the MODE field of the Header Segment FUNC array when FUNC[ENDMED] is true.

FORMAT = 0;

FORMAT is the other possible value supplied in the MODE field of the Header Segment FUNC array when FUNC[ENDMED] is true.

MINSEC = "SECMAP_BASE_DSKPG + NUM_DISK_SECTORMAP_PAGES =" n;

MINSEC, the smallest disk page address the spooler may use for building spool files, must be adjusted to allow for two variables. SECMAP_BASE_DSKPG allows for spooler control list backup pages. NUM_DISK_SECTORMAP_PAGES allows for sector map backup.

MAXSEC = "MINSEC + (NUM_K_STORE * 1024) - 1 =" n;

MAXSEC is the maximum allowable disk address to be used for data and pagemap storage.

NUM_K_STORE = n;

NUM_K_STORE determines the number of 512-byte data pages available to the spooler in the virtual disk. The integer 'n' is the number of pages expressed in multiples of 1K.

PAGEOFFSET = MINSEC;

PAGEOFFSET is used in disk page address calculations in the INPUT_CONTROL_MONITOR's Sector Map subsection. It is renamed here only for clarity's sake.

FILENAME = 'SPOOLER ';

FILENAME is the 8-character name of the virtual disk created under OS/16 MT2. The name must be padded with enough blanks to make it 8 characters in length.

EXTENSION = 'VD ';

EXTENSION is the 4-character extension of the virtual disk filename. The name must be padded with enough blanks to make it 4 characters in length.

PASLA_DSKPG = m;
JOBQ_IN_DSKPG = m+1;
JOBQ_R1_DSKPG = m+2;
JOBQ_R2_DSKPG = m+3;
JOBQ_PR_DSKPG = m+4;
JOBQ_GAR_DSKPG = m+5;

These entries are disk page addresses of control list backup pages the spooler is to use when the backup option is specified. The value 'm' is usually at the beginning of the virtual disk space (page 0). The addresses 'm' through 'm+5' must not overlap addresses between MINSEC and MAXSEC, inclusive.

SECMAP_BASE_DSKPG = m+6;

This is the beginning address of the INPUT_CONTROL_MONITOR Sector Map subsection's disk backup of the Sector Map.

NUM_DISK_SECTORMAP_PAGES = n;

Tells the number of reserved backup pages there are for Sector Map storage. Each page holds enough sector map to control the allocation and deallocation of 4K disk pages.

DISK_PAGE_K = "(NUM_DISK_SECTORMAP_PAGES * 4) - 1 = " i;

Gives the number of useable disk pages (expressed in 1K increments) indexed from zero.

HDRLLEN = 14;

The number of bytes in the Header Segment of the Intermachine Communication Protocol (ICP).

MAX_INSTRUCTION = 35;

Maximum size of the instruction array in an Virtual I/O Machine instruction packet. This is valid in all cases unless a constant, local to a system type, overrides this value.

APPENDIX 2.2:
TYPE DESCRIPTIONS

TYPE LINE = ARRAY [1..LINELENGTH] OF CHAR;

LINE is the basic input and output type to devices other than the disk.

TYPE PAGE = ARRAY [1..PAGELENGTH] OF CHAR;

PAGE is the basic input and output type used for the disk. All disk I/O must be in a type of this size.

TYPE INTPAGE = ARRAY [1..256] OF INTEGER;

INTPAGE is the same size as PAGE, only the type is broken down into integers instead of characters. This is done for type checking reasons.

TYPE PROCESS_PRIORITY =
(INPUT_PRI, JOBQ_PRI, PRINT_PRI, GARB_PRI);

These are the possible priorities at which a process may request use of the disk device.

TYPE MAP_PAGE = ARRAY [1..PGMAPLENGTH] OF INTEGER;

Page used to store pointers to data pages and to store pagemap linkage pointers.

TYPE SECTOR_SET = 0..MAXSEC;

Subrange of allowable disk page addresses. SECTOR_SET should really be MINSEC..MAXSEC, but type checking necessitates the alteration.

TYPE SECTORS = SET OF 0..127;

This set is used as the basic building block out of which the Sector Map is constructed. Each of the 128 elements corresponds to an individual data page.

```
TYPE PAS_ID = ARRAY [1..4] OF CHAR;
```

 Type for ID as it is received from the user computer in the Header Segment.

```
TYPE SYS_ID = ARRAY [1..8] OF CHAR;
```

 Type for user ID after the spooler has concatenated a unique integer to it.

```
TYPE PRINT_PROC_NUM = 1..NUM_PRINTER_PROCESSES;
```

 Subrange of allowable printer numbers.

```
TYPE PRINTER_COMMANDS =  
(FORWARD, BACKWARD, PURGE_CURRENT, RESTART, NOT_COMMAND);
```

 Enumeration of allowable printer commands.

```
TYPE PASLA_IO_RECORD = RECORD  
    ID1: PAS_ID;  
    ID2: PAS_ID;  
    PGMAPINDEX: INTEGER;  
    CURDATPTR: INTEGER;  
    CURPGMAPADDR: INTEGER;  
    LENGTH: INTEGER  
END;
```

 One of these records is kept by the INPUT_CONTROL_MONITOR for each open file it is building. Record elements are used as follows:

ID1 and ID2: These each store half of the ID of an open file. The ID is split to facilitate searching the INPUT_CONTROL_MONITOR's control lists.

PGMAPINDEX: A pointer to the current element in the current pagemap.

CURDATPTR: A pointer to the current location in the data buffer to be used for blocking up incoming data.

CURPGMAPADDR: Disk address of the current pagemap being built.

LENGTH: Accumulates the length of a file as it is built. It reflects the number of data pages that have been stored on disk.

```

TYPE INPAGE = RECORD
    IPAGE:  ARRAY[1..IPAGELNGTH] OF CHAR
END;

```

INPUT_CONTROL_MONITOR input page type. Allows for varying input page lengths.

```

TYPE LISTNAME = (INPUT,READY,PRINTING,GARBAGE);

```

Enumeration stating possible list names in the JOBQUEUE monitor.

```

TYPE INPUTREC = RECORD
    ID:  SYS_ID;
    HEADPTR:  INTEGER;
    LIST:  INTEGER;
    PURGE:  INTEGER;
    DUMMY:  INTEGER
END;

```

This record is used by the JOBQUEUE to build its Input List. DUMMY is included only to keep the record 16 bytes in length.

ID: System ID of the open file.
 HEADPTR: Address of the first page of the file's pagemap.
 LIST: A flag stating whether the file has been listed or not.
 PURGE: A flag stating that when the file is closed, it is to be deleted immediately, and not passed to the Readylist.

```

TYPE READYREC1 = RECORD
    ID:  SYS_ID;
    HEADPTR:  INTEGER;
    PR:  INTEGER;
    LEN:  INTEGER;
    NC:  INTEGER
END;

```

This is the first half of the JOBQUEUE's Readylist. It contains a partial description of all files waiting to be scheduled for print.

ID: System filename.
 HEADPTR: Address of first page of pagemap.

PR: Code determining which printers are allowed to print this particular print file. There is a fifteen printer maximum using this 2-byte integer. The most significant bit is ignored.

LEN: Length of the print file in data pages.

NC: The number of copies left to be assigned to a printer.

TYPE READYREC2 = RECORD

```

        DUMMY1: ARRAY [1..8] OF CHAR;
        MD: INTEGER;
        LIST: INTEGER;
        DUMMY2: INTEGER;
        DUMMY3: INTEGER
    END;
```

This is the other half of the Readylist record. DUMMY1 through DUMMY3 are padding used to keep the record 16 bytes in length.

MD: Code representing which mode the file is to be printed in (currently IMAGE or FORMAT).

LIST: Flag indicating whether the file has been listed or not.

TYPE PRINTINGREC = RECORD

```

        ID: SYS_ID;
        HEADPTR: INTEGER;
        LIST: INTEGER;
        FLENGTH: INTEGER;
        MD: INTEGER
    END;
```

This record keeps current information needed by the Printer Processes as they print files.

ID: System filename.

HEADPTR: Address of first page of file map.

LIST: Flag indicating whether the file has already been listed.

FLENGTH: Length of the current print file.

MD: Flag indicating IMAGE or FORMAT mode.

```
TYPE PRINT_COM_REC = RECORD
    FLAG:    BOOLEAN;
    COMMAND:  PRINTER_COMMANDS;
    COMM_VALUE:  INTEGER
END;
```

An array of these records (one element for each Printer Process) keeps track of the pending commands for each process.

FLAG: Flag indicating whether or not a command is waiting for its Printer Process when the process checks in with the JOBQUEUE as it prints each page.

COMMAND: Represents which command the Printer Process is to execute.

COMM_VALUE: If there is an argument associated with a given command, it is stored in this variable.

```
TYPE DUMMYLIST = ARRAY [1..32] OF INPUTREC;
```

The JOBQUEUE uses this type in a shared list manipulation routine. DUMMYLIST may hold any control list.

```
TYPE FRAMEHDR = RECORD
    LENGTH:  INTEGER;
    TASKID:  ARRAY [1..4] OF CHAR;
    FUNC:    ARRAY [1..4] OF INTEGER
END;
```

This record is the format of the Intermachine Communication Protocol header segment as used by the Pasla driver class (PASXMTRECCL).

LENGTH: This is the length of the entire transmission. It includes the length of the header, data buffer, and checksum segments.

TASKID: ID of the user task in the sending computer. It is used to direct incoming data to the proper spool file.

FUNC: This array contains the following four pieces of information in this order - the End-of-Medium flag, the number of copies to print, the printer selection code, and the mode flag. These are supplied by the user and must be valid on transmission of a file's last data buffer.

```
TYPE INT_REC = ARRAY [1..HDRLen] OF CHAR;
```

This array is used to bypass compiler type-checking by some routines in the Pasla Driver Class. It's only value is that it is 16 bytes long.

```
TYPE IO_PKT_TYPE = RECORD
    IC:    INTEGER;
    STATUS: INTEGER;
    COUNT: INTEGER;
    FILLER: ARRAY [0..12] OF INTEGER;
    INST:  ARRAY [0..MAX_INSTRUCTION] OF
        INTEGER;
END;
```

This is the I/O packet for the Kernel's Virtual I/O Machine. Different versions of the I/O Packet occur at various places in the spooler, but the only difference is in the length of the INST array. Other entries remain the same.

IC: Instruction Counter which points to the instruction in the INST array to be executed.
 STATUS: Result of the I/O command returned by the Kernel to the user.
 COUNT: Number of bytes processed by the command.
 FILLER: Locations used by the VIM.
 INST: Array of virtual instructions to be executed by the I/O Machine.

```
TYPE IORESULT =
    (COMPLETE, INTERVENTION, TRANSMISSION, ENDFILE,
    ENDMEDIUM, STARTMEDIUM);
```

These are the possible results upon return from using the Pasla Driver Class (PASXMTRECCL).

COMPLETE: I/O completed satisfactorily.
 TRANSMISSION: Failure of transmission.
 (Others are not used)

```
TYPE CONTYPES = (CONTTY, CONPASLA);
```

This enumeration designates possible console types. The console driver is written to service two types of terminals - a standard Teletype and a Pasla-driven

terminal. In order to specify the difference, one of these values is passed the driver during initialization.

CONTTY: The terminal is a Teletype.

CONPASLA: The terminal is a Pasla-driven device.

TYPE RETURN_CODE = (TIMED_OUT,GOOD_CHAR);

This enumeration designates the two possible values returned to the TTY_CLASS INPLN procedure as it waits for a read to be satisfied. As noted in the document, a read by the TTY_CLASS may be terminated with a good character or it may have timed out in order to free up the console.

TIMED_OUT: No character was received during the specified time period.

GOOD_CHAR: A valid character was received before timing out.

APPENDIX 2.3:
CLASS DESCRIPTIONS

@MN: PASXMTRECCL

@MT: CLASS

PURPOSE:

This class handles the transmission and reception of varying length data buffers over the Paslas. To send a data buffer, the class must be supplied the buffer (≤ 512 bytes in length), the length of the buffer (for a write), and some header information. If receiving, all this information is returned to the Input Process. Also a result giving the status of the transmission is returned to the caller. Provisions are made to count the number of errors made in transmission or reception, if desired. For a more detailed account of the protocol, see the Intermachine Communication Protocol (ICP) in section 2.4 of the document.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: INTEGER;

The device address is given to the class when its initialization takes place.

VARIABLE NAMES and their FUNCTIONS:

SENDPKT: I/O packet for the send side of the protocol.
RECVPKT: I/O packet for the receive side of the protocol.

ENTRY PROCEDURES:

```
@PE: SENDBUF(VAR BUF: UNIV PAGE;  
             LEN: INTEGER;  
             VAR HDRBUF: FRAMEHDR;  
             VAR RESULT: IORESULT;  
             VAR BLKCNT: INTEGER;  
             VAR ERRCNT: INTEGER);
```

BUF is the data buffer to be sent. HDRBUF is the header segment partially filled in. LEN is the length (in bytes) of valid data in the buffer. BLKCNT is a returned count of the number of blocks. ERRCNT is a returned count of error blocks. RESULT returns the status upon I/O completion.

```
@PE:  RECVBUF (VAR BUF:  UNIV PAGE;  
              VAR LEN:  INTEGER;  
              VAR HDRBUF:  FRAMEHDR;  
              VAR RESULT:  IORESULT;  
              VAR BLKCNT:  INTEGER;  
              VAR ERRCNT:  INTEGER);
```

All variables are the same as SENDBUF except for LEN, which is set to the number of bytes received.

@MN: SEND_CLASS

@MT: CLASS

PURPOSE:

This class accepts a line of data and puts it in the I/O packet for the Teletype or the Pasla-driven device, and supplies any additional control characters if needed (for example, null characters for slow devices).

ACCESS RIGHTS and LOCAL CLASSES:

@AR: CONTYPES;

@AR: INTEGER;

The device address and the device type (Teletype or Pasla) are given this class. There are no other access rights.

VARIABLE NAMES and their FUNCTIONS:

IO_RECORD: This is a version of the standard I/O packet as described in TYPES AND CONSTANTS.

LN: A local line is needed to hold the output line in order to be able to manipulate it. In the case of a string constant, without assigning it to a local variable (LN), SEND_CLASS could not alter it.

NULLS: Four null characters for the Teletype.

ENTRY PROCEDURES:

@PE: OUTPUT(VAR LINE: UNIV STRING76;
VAR LENGTH: INTETER);

LINE is the buffer to output. LENGTH is the number of characters to output from LINE.

@MN: RECEIVE_CLASS

@MT: CLASS

PURPOSE:

The RECEIVE_CLASS reads characters from a console device one at a time. This class sets the timeout for the console read, issues the read, and returns the result to its calling class. Currently the timeout value is set at 1 second. If a good character is received, then the return code is set to GOOD_CHAR and the parity bit is stripped off. Otherwise, a status of TIMED_OUT is returned.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: CONTYPES;

@AR: INTEGER;

No access is needed to any modules by this class. Only the console type and console device address are passed to the class upon initialization.

VARIABLE NAMES and their FUNCTIONS:

IO_RECORD: I/O packet containing virtual instructions for receiving one character or timing out.

CH: Variable to receive the byte that was input.

ENTRY PROCEDURES:

@PE: INPUT(VAR DATA: UNIV INTEGER;
VAR RCODE: RETURN_CODE);

DATA is the area the received byte is to be read into. RCODE tells the receiver whether the byte is valid or not.

@MN: TTY_CLASS

@MT: CLASS

PURPOSE:

TTY_CLASS combines the SEND_CLASS and RECEIVE_CLASS together, with additional logic, to provide complete two-way communication with a terminal device.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: CONTYPES;
 @AR: INTEGER;
 @AR: TYPRESOURCE;

As in previous classes, the console type and console device address are given to the class upon initialization. Access is also given to a TYPRESOURCE to guarantee mutually exclusive access to the console device.

@LC: SEND_CLASS;
 @LC: RECEIVE_CLASS;

An occurrence of the SEND_CLASS is used to output the prepared line and an occurrence of RECEIVE_CLASS is used to input characters to build the input line.

VARIABLE NAMES and their FUNCTIONS: NONE

ENTRY PROCEDURES:

@PE: OUTLN(LINE: STRING76);

OUTLN determines the number of valid characters in the LINE, adds any necessary control characters peculiar to the output device, and then outputs it via SEND_CLASS.

@PE: INPLN(VAR LINE: STRING76;
 VAR LENGTH: INTEGER);

INPLN contains the logic to build a received line, byte-by-byte. It also is responsible for the release of and re-request of access to the console device in the event of a timed-out character. NOTE - access to the device for a read is acquired in this class, but access to the device for a write is acquired in the TERMINAL class.

@MN: TERMINAL

@MT: CLASS

PURPOSE:

The TERMINAL class is the program-level interface the user sees in writing or reading a line. On a read, the line is essentially passed straight through to a TTY_CLASS. On a write, first, access is requested to the output device, second, if the current process is different from the last one to use the device, an identification line is printed, and third, the line is output and access to the device is released.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: TYPERESOURCE;
@AR: CONTYPES;
@AR: INTEGER;

TERMINAL is passed the console device address and its type. It also has access to a TYPERESOURCE monitor to guarantee mutually exclusive access of the output device.

@LC: TTY_CLASS;

The TERMINAL has one occurrence of a TTY_CLASS used for communication with the operator's console.

VARIABLE NAMES and their FUNCTIONS: NONE

ENTRY PROCEDURES:

@PE: READ(HEADER: LINE;
 VAR TEXT: LINE);

HEADER is a line identifying the user of the terminal. TEXT is the line to be read.

@PE: WRITE(HEADER: LINE;
 TEXT: LINE);

Same as above except TEXT is written out instead of read in.

@MN: TERMINALSTREAM

@MT: CLASS

PURPOSE:

In some cases it is desirable to read a line character-by-character from the device (in parsing a command, for instance). TERMINALSTREAM allows the programmer to do this. It will read an entire line in, supply the user from that line one character at a time, and read more lines as necessary.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: TERMINAL;

TERMINALSTREAM is given access to a TERMINAL declared in the user's program (it actually shares it with the user).

VARIABLE NAMES and their FUNCTIONS:

The spooler only reads character-by-character, so the TERMINALSTREAM has be modified to support that function alone. This leaves the following local variables. TEXT is a LINE of storage to supply the user's character reads from, and COUNT keeps track of how many character are left unread in TEXT. ENDINPUT is a boolean flag indicating the need to read another line from the input device (via TERMINAL) upon depletion of the TEXT line buffer.

ENTRY PROCEDURES:

@PE: READ(VAR C: CHAR);

C is the character to be returned to the user. Since C is two bytes long, the character is returned in the least significant byte.

@MN: IO_STATUS_CLASS

@MT: CLASS

PURPOSE:

IO_STATUS_CLASS is provided to make the manipulation of disk I/O results easier for the DISC_CLASS.

ACCESS RIGHTS and LOCAL CLASSES: None

VARIABLE NAMES and their FUNCTIONS: None

ENTRY PROCEDURES:

```
@PE: IO_MACHINE(    STATUS: INTEGER;  
                  VAR RESULT: IO_RESULT);
```

STATUS is the integer result of the I/O operation as returned by the Kernel. It is decoded and the logical result is returned in RESULT.

@MN: DISC_CLASS

@MT: CLASS

PURPOSE:

DISC_CLASS reads or writes 512-byte pages to the virtual disk at the address requested by the user.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: DEV_CHANS;
@AR: DEV_CHANS;
@AR: DEV_CHANS;
@AR: LINEBUFFER;

DISC_CLASS is passed the Disk Controller Direct Access channel, the Disc Controller channel, and the Removeable Disk address. Access is given to the Message Posting Process LINEBUFFER so error messages can be posted.

@LC: IO_STATUS_CLASS;

IO_STATUS_CLASS is used to help decode I/O statuses.

VARIABLE NAMES and their FUNCTIONS:

IOPKT is a large I/O Packet. Other local variables define and store allowable ranges for cylinder, sector, channel, and block values.

ENTRY PROCEDURES:

@PE: READ_PAGE(PAGEADDR: INTEGER;
VAR BLOCK: UNIV INTPAGE);

PAGEADDR is the address of the desired page (relative to the start of the virtual disk). BLOCK is the buffer where the 512 bytes is to be stored.

@PE: WRITE_PAGE(PAGEADDR: INTEGER;
VAR BUFR: UNIV INTPAGE);

(Similar function to READ_PAGE).

```
@PE:  INIT_VIRT(      VD_NAME:  DISK_NAME;  
                      VD_EXT:  DISK_EXTENSION;  
                      VAR BLOCK:  UNIV INTPAGE);
```

This entry procedure is called once for each occurrence of the DISC_CLASS. In the spooler, there is only one occurrence and it is initialized by the JOBQUEUE.

VD_NAME is an 8-character name matching the name of the virtual disk. VD_EXT is the extension of the virtual disk name. BLOCK is a buffer used by the class during initialization to hold temporary data.

APPENDIX 2.4:
MONITOR DESCRIPTIONS

@MN: RESOURCE

@MT: MONITOR

PURPOSE:

One occurrence of a RESOURCE exists for each device for which mutually exclusive access must be provided. All processes using the device must call it's RESOURCE and wait if necessary on a FIFO queue until access can be granted. When done with the device, the process must call the RESOURCE again to release the device for other users.

ACCESS RIGHTS and LOCAL CLASSES:

@LC: PROCESSQUEUE;

@LC: FIFO;

A RESOURCE keeps a queue (PROCESSQUEUE) for delaying processes on. Request are ordered in its FIFO.

VARIABLE NAMES and their FUNCTIONS:

The boolean value FREE tells whether or not the device is in use.

FUNCTION ENTRIES:

@FE: ARRIVAL: INTEGER;

ARRIVAL returns the index of the next empty slot in the FIFO.

@FE: DEPARTURE: INTEGER;

DEPARTURE returns the index of the head of the FIFO (i.e. the longest waiter).

@FE: EMPTY: BOOLEAN;

EMPTY returns a boolean value telling whether the FIFO is empty (EMPTY = TRUE) or not (EMPTY = FALSE).

@FE:· FULL: BOOLEAN;

FULL returns a boolean value telling whether the
FIFO is full (FULL = TRUE) or not (FULL = FALSE).

@MN: TYPERESOURCE

@MT: MONITOR

PURPOSE:

TYPERESOURCE is basically a RESOURCE with the addition of a mechanism to identify the user of the device, if the user is different from the previous one.

ACCESS RIGHTS and LOCAL CLASSES: None

ENTRY PROCEDURES:

@PE: REQUEST(TEXT: LINE;
 VAR CHANGED: BOOLEAN);

TEXT is the identification of the process wishing to use the device. If TEXT does not equal the previous value of TEXT when REQUEST was last called, then CHANGED is returned TRUE, else it is returned FALSE.

@PE: RELEASE;

RELEASE frees the requested device for someone else and starts the next waiting process if there is one.

@MN: LINEBUFFER

@MT: MONITOR

PURPOSE:

The LINEBUFFER monitor controls access to the Message Posting Process (MPP). All system processes wishing to post messages to the console may deposit a LINE in the LINEBUFFER and the MPP will write it to the console.

ACCESS RIGHTS and LOCAL CLASSES:

@LC: FIFO;

VARIABLE NAMES and their FUNCTIONS:

LINEBUFFER has a local line buffer (BUFF) which it uses to store a message in until it can print it. Other variables include a FIFO of queues to delay processes on until the buffer is empty.

ENTRY PROCEDURES and ENTRY FUNCTIONS:

@PE: POST(MESS: LINE);

Processes posting messages call this entry point and deposit MESS in the local buffer. If BUFF is full, they are delayed until it is their turn to deposit their buffer, then they are continued.

@PE: RETRIEVE(VAR MESS: LINE);

The Message Posting Process (MPP) calls this entry point to retrieve a message to print. If there is no message, it is delayed until one is posted.

@FE: MORE: BOOLEAN;

The Message Posting Process calls MORE to see if there are more messages to post before it releases the console device. This avoids unnecessary 1-second delays between each of a series of messages while waiting for the CP to release the console device.

@MN: DISK_DEVICE

@MT: MONITOR

PURPOSE:

The DISK_DEVICE (DD) Monitor exists only because a class must be owned by someone. The arguments given the DD are passed on unchanged to an occurrence of the DISC_CLASS. The DD duplicates the entry points of the DISC_CLASS.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: LINEBUFFER;

Access is granted to LINEBUFFER in the event messages need posting.

@LC: DISC_CLASS;

This occurrence of DISC_CLASS does all the system's disk I/O.

VARIABLE NAMES and their FUNCTIONS: None

ENTRY PROCEDURES:

@PE: WRITE_PAGE(ADDR: INTEGER;
VAR BUFF: UNIV INTPAGE);

The page ADDRESS to be written and the BUFFER to write from are passed unchanged to the DISC_CLASS.

@PE: READ_PAGE(ADDR: INTEGER;
VAR BUFF: UNIV INTPAGE);

The page ADDRESS to be read and the BUFFER to read into are passed directly to the DISC_CLASS.

@PE: INIT_VIRT(VAR BUFF: UNIV INTPAGE);

The initial process calls this entry point from the JOBQUEUE during its initialization. Virtual filename and extension are supplied by the DD, and along with the JOBQUEUE's BUFFER, are passed to the DISC_CLASS so it can initialize itself.

@MN: DISK_SCHEDULER

@MT: MONITOR

PURPOSE:

The DISK_SCHEDULER (DS) allows prioritizing disk usage according to a process' current priority. These priorities are, from highest to lowest, INPUT_PRI, JOBQ_PRI, PRINT_PRI, and GARB_PRI.

When a process requests the disk, and the disk is not in use, the process receives control immediately. If, however, the disk is in use, the process is delayed on it's current priority's queue. When the disk becomes free, the longest waiter on the highest priority queue is continued.

The DS also services INPUT_CONTROL_MONITOR requests directed to the Disk Output Process (DOP). The DOP waits in the DS until there is a page to write to disk. It then goes to the INPUT_CONTROL_MONITOR and picks it up, writes it to disk (at top priority), and returns to the DS to await another page.

ACCESS RIGHTS and LOCAL CLASSES:

@LC: FIFO;

VARIABLE NAMES and their FUNCTIONS:

Important variables in this monitor are queues (or FIFOs of queues) kept for delaying requesting processes. These are

INP_DELAY: Input priority,
JOBQ_DELAY: JOBQ priority,
PRINT_DELAY: Printing priority, and
GARB_DELAY: Reclamation priority.

FIFOs exist only for the input and print priorities. Other priorities have only one process at a time requesting the disk.

ENTRY PROCEDURES:

@PE: REQUEST_DISK(PRIORITY: PROCESS_PRIORITY);

Processes requesting the disk call this entry point, giving their current priority. They are scheduled accordingly.

@PE: RELEASE_DISK;

When the process has finished its disk I/O, it calls this entry point to free the disk for the next user.

@PE: REQUEST_PROCESS;

The process currently in the INPUT_CONTROL_MONITOR that wishes something written to disk, calls this entry point to start up the DOP.

@PE: WAIT_FOR_BUFF;

The DOP calls this entry point repeatedly as it waits for another buffer it can write to disk. If none exists, it is delayed.

@MN: JOBQUEUE

@MT: MONITOR

PURPOSE:

The JOBQUEUE is the heart of the spooler's logic. It contains control lists, print scheduling logic, console formatting and list manipulation routines, operator command processing routines, and spooler initialization code.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: TYPRESOURCE;
 @AR: DISK_SCHEDULER;
 @AR: CONTYPES
 @AR: INTEGER;
 @AR: LINEBUFFER;
 @AR: DISK_DEVICE;

The JOBQUEUE has access to a TYPRESOURCE to ensure mutually exclusive console access, to DISK_SCHEDULER and DISK_DEVICE for prioritized disk use, and to LINEBUFFER for posting error and trace messages. The address and type of the console device are given to the JOBQUEUE so it can initialize its own occurrence of an OPERATOR class.

@LC: FIFO;
 @LC: TERMINAL;

The TERMINAL is used by the JOBQUEUE to do console reads during its initialization.

VARIABLE NAMES and their FUNCTIONS:

The Input, Ready, Printing, and Garbage lists in the JOBQUEUE must be 512 bytes long. This is done so it is possible to write the lists directly to disk without any additional processing.

INPUTLIST: Input control list
 READYLIST1,
 READYLIST2: Combined to make the Ready control list
 PRINTINGLIST: Printer control list
 GARBAGELIST: Reclamation control list

Queues used to delay processes in the JOBQUEUE are defined as follows:

READYFULLQ: Ready list is full (FIFO)

PRINTER_DELAY: No files for printer
 GARBAGEFULLQ: Garbage list is full (FIFO)
 GARBAGEQ: No files to reclaim

Remaining variables are defined as follows:

DELAY_PENDING: An entry of boolean flags indicating whether or not the associated Printer Process is to be delayed on its next JOBQUEUE call.
 COMMAND_PENDING: An array of PRINT_COM_RECs describing the next command the Printer Process is to execute.
 PRINTER_WAITING: Array of flags indicating idle Printer Processes.
 TEXT: A LINE used during initialization.
 BACKUP: Boolean flag indicating whether the disk backup mode is in effect or not.
 DEBUG: Boolean flag indicating whether trace statements are to be posted on the console.
 PRINT_IT: Flag indicating whether to print or discard the prepared print line (selected by operator OPTION command). Default is PRINT_IT = TRUE.
 SEARCH_IN_PROGRESS: Flag used during listing of control lists.

ENTRY PROCEDURES and ENTRY FUNCTIONS:

@PE: GET_GARBAGE(VAR SECTOR: SECTOR_SET);

The Memory Reclamation Process (MRP) calls this entry point to ask for a spool file to reclaim. If none exists, it is delayed on GARBAGEQ until one is ready. SECTOR is the address of the first page of the pagemap that is returned to the MRP.

@PE: GARBAGE_COLLECTED;

When the MRP has completely collected the spool file, it calls this entry point to notify the JOBQUEUE it is done.

@PE: OPEN(NEWHEADPTR: SECTOR_SET;
NEWID: SYS_ID);

When the INPUT_CONTROL_MONITOR opens a new spool file for input, this entry point is called to notify the JOBQUEUE. NEWHEADPTR is the address of the first page of the pagemap of the new file. NEWID is the 8 character ID of the new file. OPEN is always guaranteed room on its lists by the

INPUT_CONTROL_MONITOR, so there is no possibility of being delayed as a result of this call.

```
@PE:  ENTER(NEWNC:  INTEGER;
          NEWPR:  INTEGER;
          SYSTEM_ID:  SYS_ID;
          NEWLEN:  INTEGER;
          NEWMODE:  INTEGER);
```

ENTER is called by an Input Process as soon as the spooler has received the last data buffer from the user (EOM is TRUE). When this happens one of two actions may take place. If the file was previously marked for delete by the operator command DELETE <filename>, then it is immediately moved to the MRP's control lists for reclamation and is purged from the system. Normally, when the ENTER procedure is called, the file is moved from the JOBQUEUE's Input list to its Ready list. If the Ready list is full, then the Input Process is delayed on a FIFO queue (READYFULLQ) until there is space on the Ready list and it can be continued. Once the file is on the Ready list, as the Input Process leaves the ENTER procedure, it tries to start a Printer Process who can work on the file. If one is idle that can print this file, it begins printing. Also, that Printer Process will try to continue another idle Printer Process so that as many copies of the file as possible may be printed at once. This is governed by the printer code (NEWPR).

```
@PE:  DONE_PRINTING(PNUM:  PRINT_PROC_NUM);
```

The Printer Processes call this entry point and identify themselves by their PNUM when they have finished printing their current copy of the file. If the finished copy is the last one to be printed, then the file is moved to the Garbage List for reclamation. The MRP is started if it was delayed.

```
@PE:  GET_JOB(VAR NEWID:  SYS_ID;
             TPNUM:  PRINT_PROC_NUM;
             VAR PAGENUM:  SECTOR_SET);
```

GET_JOB assigns the requesting Printer Process (TPNUM) a file to print. When the request is registered, the JOBQUEUE checks to see if there are any files for the printer. If not, then the process is delayed. If there are, then a copy is assigned to the

printer, and its process is continued if it is delayed. If there are no more copies to be printed after this one, then the file's entry is removed from the Ready list. As the Printer Process exits the GET_JOB entry point, it tries to continue any other Printer Processes that may have a file to print.

NEWID is the filename of the spooled print file. PAGENUM is the address of the first page of the pagemap for the file. These two values are returned to Printer Process TPNUM.

```
@PE: CHECK_STATUS(VAR PCOMM: PRINTER_COMMANDS;
                  VAR VALUE: INTEGER;
                  TPNUM: PRINT_PROC_NUM;
                  VAR WANTS_PRINT: BOOLEAN);
```

For every page of printed information, the Printer Process (TPNUM) checks with the JOBQUEUE to see if any operator commands have been logged in the JOBQUEUE which will affect the process' normal execution. If the DELAY_PENDING flag is set, then the process is delayed in the JOBQUEUE. Otherwise, any commands for the process are returned to it in PCOMM (the command) and VALUE (the command's argument). WANTS_PRINT tells the process if it is to print the next data page. It may be changed by the OPTION operator command.

```
@PE: CHANGE_PAGE(PNUM: INTEGER;
                  N: INTEGER;
                  NEWCOMMAND: PRINTER_COMMANDS);
```

The Console Process (CP) calls this entry point to initiate execution of the FORWARD and BACKWARD operator commands. PNUM is the printer the command is to be directed toward, N is the number of data pages to be moved in the file, and NEWCOMMAND is either FORWARD or BACKWARD, indicating the direction to move.

```
@PE: PAUSE(PNUM: INTEGER);
```

The Console Process (CP) calls this entry point as the result of a PAUSE operator command. PNUM is the printer to be paused at the end of its current page of print.

```
@PE: CONT(PNUM: INTEGER);
```

The Console Process (CP) calls this entry point as the result of a CONTINUE operator command. PNUM is the printer to be continued.

```
@PE: RESTART_PRINT(PNUM: INTEGER);
```

The Console Process (CP) calls this entry point as the result of a RESTART operator command. PNUM is the printer which is to be rewound to the beginning of its print file.

```
@PE: DELETE(      SID: SYS_ID;
               VAR CONTINUE_READYWTR: BOOLEAN);
```

The Console Process (CP) calls this entry point to have the JOBQUEUE delete a file. If the file is not found, then an error message is posted saying so. If it is found, then one of three actions takes place. If the file is on the Input list, it's PURGE flag is set and as soon as that file is ENTERed it is moved immediately to the Garbage List. If the file is on the Ready list, it is moved to the Garbage List, and all remaining copies to be printed are scrapped. If it is currently printing, then the first Printer Process found that is printing a copy of the file is given instructions to execute the PURGE_CURRENT command.

If the Ready list was full when the file was deleted from it, CONTINUE_READYWTR is returned TRUE to the CP so it can continue an Input Process trying to ENTER a file (if one was waiting) by calling the CONT_RDYWTR entry point in the JOBQUEUE.

NOTE - the lists are searched in this order - Input, Ready, and then Printing. The first occurrence of SID that is found gets deleted.

```
@PE: LIST(      NAME: LISTNAME;
               VAR NOMORE: BOOLEAN;
               VAR INFO: LINE);
```

LIST is called by the Console Process (CP) so it can display the contents of a given JOBQUEUE list on the console. The JOBQUEUE returns the lines entry-by-entry to the CP. The CP is responsible for printing the list, not the JOBQUEUE. In this way, the JOBQUEUE is kept as free as possible.

When a list is printed, the CP specifies the

listname (NAME) and gives the JOBQUEUE a line buffer to write the formatted information into (INFO). As long as the current list has not been completely printed, NOMORE is returned to the CP with a FALSE value. As the last entry of a list is being returned, NOMORE is set to TRUE.

@PE: CONT_RDYWTR;

The CP calls this entry point to continue the next process waiting to ENTER a file into the Ready list.

@FE: WANT_BACKUP: BOOLEAN;

WANT_BACKUP is called by modules during their initialization in order to set their local booleans to reflect whether or not the spooler is to do disk backup of its control lists. It is set by the JOBQUEUE during initialization dialogue with the operator.

@FE: WANT_DEBUG: BOOLEAN;

WANT_DEBUG is set in the same way as WANT_BACKUP only it signals the trace mode.

@PE: SET_PRINT(VALUE: BOOLEAN);

The CP calls SET_PRINT to control the printing of all files. If VALUE is TRUE then all files are printed. If VALUE is FALSE, the files are discarded page by page to a "null" printer device. The default value is TRUE.

@MN: INPUT CONTROL MONITOR

@MT: MONITOR

PURPOSE:

The INPUT_CONTROL_MONITOR builds the spool files on disk, buffer-by-buffer, as they are received from user computers. There are three main subsections in the INPUT_CONTROL_MONITOR. These are the Disk Output Process Buffer Management subsection which controls access to the DOP's buffer, the Sector Map subsection which controls allocation and deallocation of virtual disk pages, and the Input subsection which sorts incoming buffers, blocks them up in the correct file (by ID), and maintains each file's pagemap.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: DISK_DEVICE;
 @AR: DISK_SCHEDULER;
 @AR: LINEBUFFER;
 @AR: JOBQUEUE;

The INPUT_CONTROL_MONITOR has access to the DISK_DEVICE and DISK_SCHEDULER for disk I/O, to the LINEBUFFER for posting messages, and to the JOBQUEUE. The JOBQUEUE is only used during initialization. The INPUT_CONTROL_MONITOR obtains information from the JOBQUEUE telling if disk backup is to be maintained and if debugging messages are to be posted to the console. This temporary use of the JOBQUEUE allows questions to be asked only once for the entire system initialization, instead of once per module initialization.

@LC: FIFO;

VARIABLE NAMES and their FUNCTIONS:

These variables are the main ones in the INPUT_CONTROL_MONITOR:

DISK_PAGES: Array built out of SECTORS types which is the Sector Map's bit map (1 bit per disk page).
 BACKUP_FLAGS: Array of boolean flags indicating which parts of the Sector Map need backing up. When the backup option is in effect, only that part of the Sector Map which has actually been changed is backed up.
 BUFF: The DOP's information buffer.
 PGADDRBUFF: The DOP's page address buffer. It holds

the address that BUFF is to be written to on disk.

FULLQ: Array of queues to hold processes waiting to deposit information in the DOP's data BUFFER.

SEC_DELAY: Array of queues to hold Input Processes waiting for disk space to be freed.

IN_STORE: Array of INPAGES (one for each possible open file) to hold data for each file as it is blocked up.

IN_DATA: Array of PASLA_IO_RECORDs. These records make up the Input List in the INPUT_CONTROL_MONITOR. This list must be 512 bytes long.

PAGEMAP: Array of MAP_PAGES for storage of each open file's current pagemap.

BACKUP: Boolean flag stating whether or not to do disk backup.

DEBUG: Boolean flag stating whether or not to post debugging messages.

WAITQUEUE: Queue to delay an input process which has requested the 'I/O and Wait' mode of disk use.

ENTRY PROCEDURES:

```
@PE:  GET_BUFFER(VAR PGADDR:  SECTOR_SET;
                VAR DATA:  UNIV PAGE;
                VAR WAITER:  BOOLEAN);
```

GET_BUFFER is called only by the Disk Output Process (DOP) when it is requested to write a buffer to the disk. The requesting process deposits the buffer in BUFF and then calls the DISK_SCHEDULER's REQUEST_PROCESS entry point to start up the DOP. The DOP is given the page address (PGADDR) to write the received DATA buffer to. WAITER tells the DOP if the requesting process is waiting until the output is completed (WAITER = TRUE). If WAITER is TRUE, the DOP will call CONTINUE_WAITER and CONTINUE_DEPOSITOR entry points in the INPUT_CONTROL_MONITOR when it completes its I/O request.

```
@PE:  RELEASE_PAGEMAP(PAGEMAP:  INTPAGE;
                    PAGEPTR:  SECTOR_SET);
```

The Memory Reclamation Process (MRP) submits a page of the pagemap (PAGEMAP) to be released. Each entry in the PAGEMAP is released and then the pagemap page itself (PAGEPTR) is released. If any process is waiting for a disk page, then it is continued as the MRP leaves the INPUT_CONTROL_MONITOR. Then as that continued process leaves the INPUT_CONTROL_MONITOR, it,

in turn, tries to start any other process waiting for disk space (since at least two pages must have been released).

@PE: CONTINUE_WAITER;

The DOP calls this entry point to continue the process waiting for its I/O to complete.

@PE: CONTINUE_DEPOSITOR;

The DOP calls this entry point to continue the next process waiting to do disk I/O (if one is waiting).

```
@PE: WRITE_DATA(      DATA: PAGE;
                      BUFLen: INTEGER;
                      OLDID: PAS_ID;
                      ENDMED: BOOLEAN;
                      VAR ACCEPTED: BOOLEAN;
                      VAR NEWID: SYS_ID;
                      VAR FILE_LENGTH: INTEGER;
                      VAR NEED_OPEN: BOOLEAN;
                      VAR NEED_ENTER: BOOLEAN;
                      VAR NEW_PTR: SECTOR_SET);
```

WRITE_DATA is the entry point called by all Input Processes when they have a buffer of data to be spooled. If a file needs opening, closing, or DATA simply needs to be added to its spool file, it happens here.

An Input Process supplies WRITE_DATA with several pieces of information. These are

```
DATA:   The buffer of incoming information,
BUFLen: The number of valid data bytes in DATA,
OLDID:  The ID of the user task in the sending
        computer, and
ENDMED: The flag signaling the last data buffer.
```

As a result of the WRITE_DATA call, an Input Process may be directed to do different things according to these returned values.

```
ACCEPTED: If this is TRUE then the incoming data
           has been accepted. If it is FALSE, then the
           data was refused by the spooler (no room for a
           new file to be opened).
NEWID:    This is the 8-character ID of the file
```

after the INPUT_CONTROL_MONITOR has made it unique by concatenating the user ID with a 4-digit number. The NEWID is needed in certain cases by the Input Process.

FILE_LENGTH: Returns the length of the file for ENTERing in the JOBQUEUE.

NEED_OPEN: A flag indicating that the Input Process needs to do an OPEN in the JOBQUEUE. NEWID and NEW_PTR are needed by the Input Process to do the OPEN.

NEED_ENTER: A flag indicating that the Input Process needs to do an ENTER in the JOBQUEUE. NEWID and FILE_LENGTH are needed by the Input Process to do the ENTER.

NEW_PTR: The address of the first page of the file's pagemap returned to the Input Process.

```
@PE: KILL(      ID: SYS_ID;  
        VAR FOUND: BOOLEAN);
```

When the Console Process does a DELETE <filename>, the first list it checks is the input list. Since that list exists in two places (INPUT_CONTROL_MONITOR & JOBQUEUE), the CP must check the INPUT_CONTROL_MONITOR first. If the file is found in the Input list, it is closed and deleted from the INPUT_CONTROL_MONITOR's Input list, and FOUND is returned to the CP with a value of TRUE. If the file is not in the INPUT_CONTROL_MONITOR's Input list, then FOUND is set to FALSE.

APPENDIX 2.5:
PROCESS DESCRIPTIONS

@MN: DISK_PROCESS

@MT: PROCESS

PURPOSE:

The DISK_PROCESS reads data buffers from the INPUT_CONTROL_MONITOR. It then requests the disk at the highest priority, and writes the buffer to disk. If the disk I/O was 'I/O and Wait', it will continue the delayed process and next depositor in the INPUT_CONTROL_MONITOR.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: DISK_SCHEDULER;
@AR: INPUT_CONTROL_MONITOR;
@AR: DISK_DEVICE;

The DISK_PROCESS has access to the DISK_SCHEDULER and DISK_DEVICE, and to the INPUT_CONTROL_MONITOR.

VARIABLE NAMES and their FUNCTIONS:

There are two local variables for information storage after it is retrieved from the INPUT_CONTROL_MONITOR. These are the DATA buffer and the page address (PGADDR) buffer for the disk page address. A boolean WAITER signals the DOP whether or not it is to call the ICM CONTINUE_WAITER and CONTINUE_DEPOSITOR entry points.

PROCESS LOGIC:

1. Wait for a buffer in the Disk Scheduler.
2. When continued, go to the INPUT_CONTROL_MONITOR and retrieve the waiting data buffer. If WAITER is FALSE, continue another process waiting to deposit a buffer.
3. Request the disk device at the highest priority.
4. Do the disk I/O.
5. Release the disk and go to 1.
6. if WAITER = TRUE then call the INPUT_CONTROL_MONITOR to continue waiting processes.
7. Go to 1.

@MN: POSTING_PROCESS

@MT: PROCESS

PURPOSE:

The Message Posting Process (MPP) takes lines from its LINEBUFFER and writes them to the console. Once started printing, the MPP posts all outstanding messages before releasing the console device.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: LINEBUFFER;
@AR: TYPRESOURCE;
@AR: CONTYPES;
@AR: INTEGER;

The MPP has access to the LINEBUFFER to RETRIEVE lines, to TYPRESOURCE to request the console, and is passed the console type and console device address upon initialization so it can set up its TTY_CLASS.

@LC: TTY_CLASS;

The TTY_CLASS is used to write messages to the operator's console.

VARIABLE NAMES and their FUNCTIONS:

There is a LINE (MESS) to store a message for output.

PROCESS LOGIC:

1. Retrieve the message from LINEBUFFER.
2. Request the console device.
3. Output the line.
4. If there are more messages, retrieve them and go to 3.
5. Release the console device.
6. Go to 1.

@MN: SPOOL_IN_PROC

@MT: PROCESS

PURPOSE:

The SPOOL_IN_PROC (SIP) reads data from the remote computer and submits it to the spooler for processing. If for some reason the data is rejected, then the process posts an error message.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: INTEGER;
@AR: INPUT_CONTROL_MONITOR;
@AR: JOBQUEUE;
@AR: LINEBUFFER;

The SIP has access to the INPUT_CONTROL_MONITOR for submitting its data, the JOBQUEUE for doing ENTERs and OPENs, and the LINEBUFFER for posting messages. It is passed the address of the Pasla it is to service during initialization.

@LC: PASXMTRECCL; ,

PASXMTRECCL (the Pasla Driver Class) is used to transmit and receive data buffers over the Pasla.

VARIABLE NAMES and their FUNCTIONS:

The following variables are used in the SIP:

BUF: The data buffer written into by the Pasla driver.
HDRBUF: The header segment passed to the Pasla driver.

PROCESS LOGIC:

1. Receive the data buffer from the remote computer.
2. If bad transmission go to 7.
3. Submit the data buffer to the INPUT_CONTROL_MONITOR.
4. If not accepted, then post error message.
5. If an OPEN is needed, do it in the JOBQUEUE.
6. If a CLOSE is needed, do it in the JOBQUEUE.
7. Go to 1.
8. Post a message stating that the receive failed. Go to 1.

@MN: PRINTER_PROCESS

@MT: PROCESS

PURPOSE:

The PRINTER_PROCESS (PP) reads a spool file, formats it, and prints it line-by-line. It has the capabilities to move forward and backward in a spool file, reposition itself at the beginning, and simulate an EOF.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: JOBQUEUE;
 @AR: PRINT_PROC_NUM;
 @AR: DISK_SCHEDULER;
 @AR: DISK_DEVICE;
 @AR: CONTYPES;
 @AR: INTEGER;
 @AR: LINEBUFFER;

The PP has access to the JOBQUEUE for scheduling, the DISK_SCHEDULER and DISK_DEVICE for doing disk I/O, and the LINEBUFFER for posting messages. It is passed its unique Printer Process number, type of printer device, and device address during initialization.

@LC: TTY_CLASS;

TTY_CLASS is the printer driver for the Printer Process in the current version of the spooler.

VARIABLE NAMES and their FUNCTIONS:

The following variables are the more important local variables:

PAGEMAP: The current pagemap of the file being printed.
 ID: ID of the current print file.
 PCOMM: The printer command received from the JOBQUEUE.
 CURPAGE: The current data page of the file. The process prints from this page.
 TEXT: The formatted line ready for output.
 WANTS_PRINT: Boolean reflecting whether to print the page or not.

PROCESS LOGIC:

1. Request a job to print.
2. Initialize the pagemap.
3. Check to see if the JOBQUEUE has any outstanding commands.
4. Print a page.
5. If no command, go to 7.
6. Execute the command.
7. If not done with file, go to 3.
8. Notify the JOBQUEUE that DONE_PRINTING current file.
9. Go to 1.

@MN: GARBAGE_PROCESS

@MT: PROCESS

PURPOSE:

The Memory Reclamation Process (MRP) reads pagemap pages and submits them to the INPUT_CONTROL_MONITOR for freeing.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: DISK_SCHEDULER;
@AR: DISK_DEVICE;
@AR: JOBQUEUE;
@AR: INPUT_CONTROL_MONITOR;

The MRP has access to the JOBQUEUE for scheduling, the DISK_SCHEDULER and DISK_DEVICE for disk I/O, the INPUT_CONTROL_MONITOR for releasing the pagemap, and the LINEBUFFER for posting messages.

VARIABLE NAMES and their FUNCTIONS:

Only two local variables are necessary for the MRP. These are a page for the pagemap (PAGEMAP) and an integer for the page address (PAGEPTR).

PROCESS LOGIC:

1. Request a file to reclaim.
2. Read a page of the pagemap from disk.
3. Submit it to the INPUT_CONTROL_MONITOR.
4. If more file left to release, go to 2.
5. Notify the JOBQUEUE of GARBAGE_COLLECTED.
6. Go to 1.

@MN: CONSOLE_MONITOR

@MT: PROCESS

PURPOSE:

The Console Monitor Process (CP) reads console commands, parses them, and initiates their execution.

ACCESS RIGHTS and LOCAL CLASSES:

@AR: TYPRESOURCE;
@AR: JOBQUEUE;
@AR: CONTYPES;
@AR: INTEGER;
@AR: INPUT_CONTROL_MONITOR;

The CP has access to the JOBQUEUE for executing listing commands, manipulating control lists, and passing printer commands on to the proper Printer Processes, to the INPUT_CONTROL_MONITOR to delete a file, and to TYPRESOURCE to guarantee mutually exclusive access to the console device.

@LC: TERMINALSTREAM;
@LC: TERMINAL;

The TERMINALSTREAM allows character-by-character processing of operator commands by the Console Process. TERMINALSTREAM gets its lines from the TERMINAL class.

VARIABLE NAMES and their FUNCTIONS:

The CP uses the following variables:

OPTION_SET: A set of allowable one-character OPTION commands.
INT_SET: A set of integers (0 - 9).
ARG_SET: A set of LIST command arguments.

PROCESS LOGIC:

1. Read a command from the operator's console.
2. If it is not a valid command, go to 7.
3. Get the arguments (if any).
4. If the arguments are not valid, go to 7.
5. Initiate execution of the command.
6. Go to 1.
7. Print an error message. Go to 1.

APPENDIX 3

SYSTEM INITIALIZATION
and
RUN-TIME COMMAND SUMMARY

3.1	System Initialization.....	93
3.2	Operator Commands.....	93

3.1 SYSTEM INITIALIZATION

Upon initialization of the system, operator responses must be made to three questions in order to describe the mode in which the spooler is to operate. The first question is

```
'INITIALIZATION PHASE'
'SELECT I(NIT) OR C(ONTINUE)'.
```

The operator types in an 'I' if he wishes to completely free all disk space and clear all spool control lists. If he types in a 'C', the spooler resumes operation using the current state of the disk backup pages. The spooler will wait indefinitely for an 'I' or a 'C' to be typed.

The next question,

```
'BACKUP? TYPE Y OR N'
```

tells the spooler whether to maintain disk backup for recovery purposes in case of system failure. The proper response of either a 'Y' or an 'N' must be entered before the spooler will continue with initialization.

Last, the spooler asks,

```
'TRACE? TYPE Y OR N'
```

to see if trace statements showing changes in system status should be routed to the console. Once answered with a 'Y' or 'N', initialization continues and the spooler becomes fully active.

3.2 OPERATOR COMMANDS

Operator commands take the following general format:

```
COMMAND DL ARGUMENT1 DL ARGUMENT2
```

where

```
COMMAND      = a valid spooler command (see following
               summary),
ARGUMENTn    = a required or optional argument depending
               upon the selected command, and
DL            = a delimiter character - a single blank or
               comma.
```

Any time 'COMMAND' or 'ARGUMENT_n' are words, only the first letter of the word need be typed. The only exception to this rule is a 'filename', which must be entered in its

entirety. For example, the following commands are identical in function. They request the current status of all JOBQUEUE lists be printed on the operator's console.

```
LIST ALL
LIST A
LIST
L
L A
```

All commands are terminated by a carriage return.

COMMAND SUMMARY:

BACKWARD - Moves a specified printer process 'p' toward the beginning of its print file. The process repositions itself 'n' number of data pages from its current data page. If too large a number is specified, the process simply begins printing the file again.

```
BACKWARD p,n
```

CONTINUE - Continue a paused printer process 'p'.

```
CONTINUE p
```

DELETE - Deletes the first occurrence of 'filename' from the spooler.

```
DELETE filename
```

FORWARD - Similar to the BACKWARD command but moves the printer process toward the end of the file. If the number 'n' is larger than the number of data pages left to print, an EOF is simulated.

```
FORWARD p,n
```

LIST - Print the status of the requested JOBQUEUE list on the console.

```
LIST          - print all lists
LIST ALL      - print all lists
LIST INPUT    - print the Input list
LIST READY    - print the Ready list
LIST PRINTING - print the Printing list
```

OPTION - Specify an operator option.

OPTION DISCARD - route all printing files to a
null printer

OPTION PRINT - print the files (this is the
default mode)

PAUSE - Pause the specified printer 'p' at the end of
the page it is currently printing.

PAUSE p

A CONCURRENT PASCAL SPOOLING PROGRAM

by

MICHAEL EUGENE PRESS

B. S., Kansas State University, 1976

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1979

ABSTRACT:

This masters report is a thorough documentation of an output spooling program written by the author in the language Concurrent Pascal. Presented is a description of the spooler's intended use, all operator commands and their functions, and the information needed to install the system in a user environment. The report also presents concepts involved in designing and building the CPascal spooling system, how these concepts were achieved in the chosen implementation structure, design tradeoffs and their impact upon the project's usability, and possible extensions to the system.