171

OPERATING SYSTEMS

George H. Mealy

May 1962 — 30

103 p $4.00 hc
$0.75 mf

P-2584

CONTENTS

INTRODUCTION 1

INPUT-OUTPUT SYSTEMS 5

Terminology 5

Symbolic Input-Output 15

Input-Output Executor 22

Channel Communication Cells 23
The ASSIGN Routine 28
Attention Requests 29
Other Assumptions 34

Buffering Systems 36

Alternating Buffering 36
Pool Buffering 42
Logical Flags 48
Multiple Use of Buffers 53
Select and Unit Interpretative Routines 54
Dispatching 57
Opening and Closing Files 60
Trap Protection 61

SUPERVISORY SYSTEMS 63

Introduction 63

The Supervisor 65

Definition of "Job" 65
Control Card Interpretation 66
Sequencing within Jobs 69
Processor Supervision 71
Unit Assignment 74

The Vigilance Committee 78

Hardware Monitoring 78
System Transfer Points 79
Execution Monitoring 81

Linkage Routines 83

Processor Linkage 83
Object Code Linkage 84
Supervisor Linkage 84

CONCLUDING REMARKS 86

Appendix I
I. INDEX OF TERMS 87

REFERENCES 92

OPERATING SYSTEMS

George H. Mealy*

The RAND Corporation, Santa Monica, California

INTRODUCTION

Over the past five or ten years, programmers have gradually been weaned away from the practice of approaching a bare machine with card decks and sharpened red pencils in their hands, fighting with the console for more or less extended periods of time, and leaving triumphantly with a sheaf of results or in defeat with a ream of post-mortem dump. Over the same period of time, operating systems have gradually evolved from attempts to bridge the gap between the programmer and the machine. This has not been an unalloyed blessing so much as a practical necessity. Machines have become faster and more complex, a large variety of programming and debugging aids have become available, and the problems themselves have become larger and greater in variety. It is no longer possible for an individual programmer to be

---

*Any views expressed in this paper are those of the author. They should not be interpreted as reflecting the views of The RAND Corporation or the official opinion or policy of any of its governmental or private research sponsors. Papers are reproduced by The RAND Corporation as a courtesy to members of its staff.

This paper was prepared for presentation at the University of Michigan Engineering Summer Conferences to be held in Ann Arbor, Michigan, June 18-29, 1962.

an expert in every phase of programming and machine usage; he now finds himself at the tender mercy of the operating staff and the system programmer responsible for providing him with some of the necessities of life.

As each new programming language has come along, there has been a great tendency to embed it into just enough of an operating system to allow it to be used at all. The result, especially on large and somewhat popular machines like the IBM 7090, has been a number of system tapes in an installation, all with different operating characteristics, and specialization of programmers according to programming systems (one, in most cases) with which they are familiar. There has been comparatively little effort devoted to consolidating our gains in one area and making them available to programmers working in another area.

To a great extent, the current emphasis on programming languages, as opposed to other aspects of the programmer's approach to the machine, is misplaced. The actual amount of time the programmer spends coding is small as compared to the time he spends in problem analysis, checkout, and setting up runs. An advance in the art such as the introduction of recursive procedures must be reckoned as small when compared with advances in symbolic modification and checkout methods or in our understanding of input-output systems. It is entirely possible that, in our current enthusiasm for so-called common languages, we may be

settling for a common level of mediocrity rather than a genuine advance in our operating methods.

The object of having a machine is to run jobs, not programming systems. To call the systems that stand between the programmer and the machine "programming systems" is to place undue emphasis on mechanical coding aids and not enough emphasis on the other aspects of operation. By "operating systems," we shall mean the whole complex of programming, debugging, and operational aids with which the programmer deals. For the purposes of discussion, we will divide the components of an operating system into three categories:

1. Input-output systems: Codes which, in conjunction with the hardware, get data in and out of the machine.
2. Processors: Codes which transform data.
3. Supervisory systems: Codes which are responsible for job or task sequencing and communication between the programmer and components of the operating system.

The existence of tens of thousands of words of code which perform the above functions does not guarantee that we have a system -- in many cases, it is just code. The word "system" implies organization and coherence, not size and complexity. Also, to be systematic is not necessarily to be inflexible. In these lectures, our main emphasis

will be on the provision of a coherent set of communication conventions (we hesitate to use the term "standards") which will promote rather than hinder flexibility of construction and use.

Terms used below will, for the most part, be introduced in context rather than by explicit definition. Each term introduced is underlined the first time it appears. Abbreviations for terms appear in parentheses following the first occurrence of the term. A list of terms used, together with the number of the page on which each is introduced, appears at the end of these notes.

INPUT-OUTPUT SYSTEMS

TERMINOLOGY

In order to discuss input-output systems in some detail without reference to a particular machine family, it is necessary to establish a terminology that is non-committal but at the same time can be interpreted reasonably precisely in the case of any given machine. This will be done on the basis of elementary hardware functions, irrespective of how these functions may be mechanized.

The essential principles involved in the construction of input-output systems are hardware-independent to a much greater extent than is commonly recognized. It is both possible and reasonable to design systems in such a way that the programmer can well afford to be unaware of what particular type of input-output device any information file may be associated with from one run of the program to the next -- except, of course, for matters of grand strategy in design of his program.

A basic distinction that we must maintain is that between storage that is used serially and that which is used randomly, both on the programming and on the hardware levels. However, an information file that is processed sequentially may reside on a random-access device and *vice versa*. From the programming point of view, we will for the most part be discussing serial information file processing irrespective of device type, although much of what

we say will be equally applicable to randomly-accessed information files.

On the device level, the essential difference between serial and random seems to be a question of one versus two or more physical degrees of freedom. More precisely, we should speak of a tape-like device as opposed to a drum-like device [1]. In the one case, the information is stored in a linear array -- in order to read a given block we may pass over unwanted information on the way. In the other case, we may start reading at any point, irrespective of the location of the block last read or written. On this basis we might assign the usual input-output devices as follows:

Tape-like	Drum-like
Tape	Core
Typewriter	Drum
Card Reader	Disc
Card Punch	CRT Display
Line Printer	Plotter
Transmission Line	Photographic Store
Keyboard	Switch Panel
	Lights

Note that whether transmission between the machine and the device is serial or parallel is inconsequential -- the above classification is based on the type of access to the medium involved. With a tape, access is produced by linear motion in one of two directions and maybe by also skipping information; with a drum, some kind of addressing is involved, whether or not physical motion (such as motion of

a disc access mechanism) is required as a result of addressing.

It is often the case that information is written as a sequence of words or characters separated by gaps. These continuous sequences will be called records. The word block will be reserved to denote a record of maximum size (when such a maximum exists). Block size will be potential length of a typed line, the number of columns on a card, etc. On reading, the end of a record may be recognized and will commonly terminate transmission, requiring another read instruction in order that transmission of the next record commence. For instance, cards constitute records, as may a printed line or a typed line followed by a carriage return or other special indication of end of record. Even on a drum-like device, information may be transmitted on a record basis, in which case records rather than words will be addressed. On some tape devices, indeed, a record and a block are identical (e.g., the Philco S-2000 or Minneapolis-Honeywell H-800 tape systems).

A higher grouping of records, called a file, is often recognized. End of file, then, is that condition that is recognized while reading at the end of the group and that is written in order to finish off an output group. Since the word "file" is also used in a logical sense, we need two terms. That is, there is a physical grouping as indicated above as well as a logical grouping which may or may not coincide with the physical one. We have already used

the word "file" in both senses. In cases where ambiguity may otherwise result, we will speak of P-files and L-files. The phrase "information file" will always imply that the file is an L-file.

In order to introduce the remaining hardware terminology, we must first resort to a picture (Fig. 1). This is, of course, not to be taken literally except on a functional basis, although it is fairly accurate for many of our present larger machines.

The function of a channel is to transmit information between a controller and the machine. This information may be both control information and data. Some rearrangement of information may be necessary (e.g., the storage is parallel, but controllers require serial information transfer). The channel must inform the processor of error conditions or termination of an operation. In more complex machines, the channel may also have processing capability and operate under control of a program independently of what the main processor may currently be doing. In general, the channels and the main processor will compete for main storage access cycles.

The function of a controller is to select a satellite unit, relay control orders to it (e.g., rewind, eject sheet, read forward, position access mechanism to a given address, etc.), and transmit data between the selected unit and the

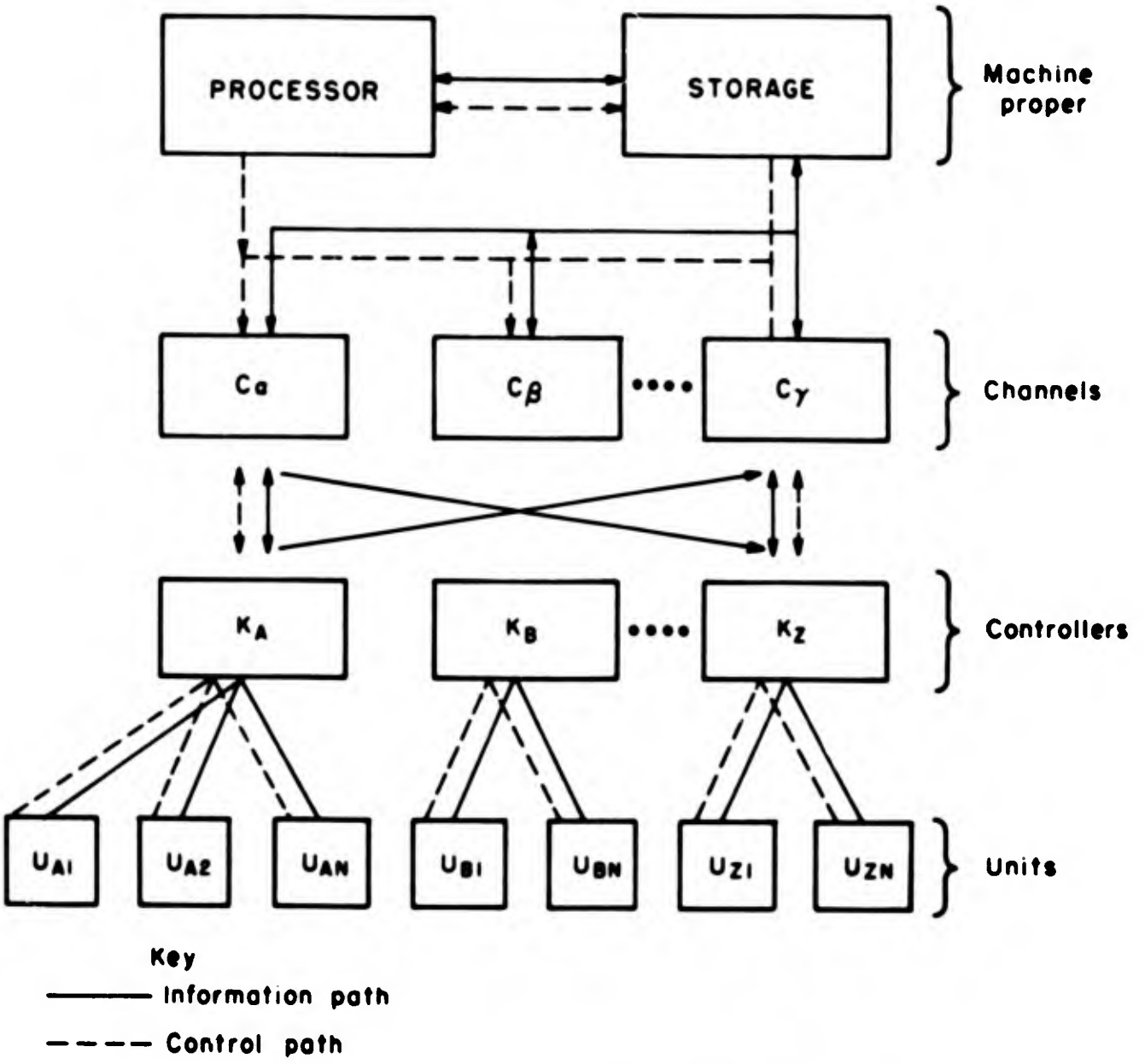

Fig. 1 — Input-output hardware

channel. It must also relay exceptional or normal conditions (e.g., parity error, end of record, unit busy, etc.) back to the machine via the channel.

It may be the case, as with the IBM 7090, that each controller is associated with only one channel or, as with the Philco S-2000, that any channel can be used with any controller -- the two possibilities should be kept in mind when we come to discuss input-output executors below. In a given hardware situation, of course, not all of these (channels, controllers, units) will be physically recognizable. Functionally speaking, however, these distinctions are necessary.

We can now define terms based on the type of information transmitted back and forth among the processor, channels, controllers, and units and the type of action to be performed by each functional link. We do not feel it necessary to distinguish between the ways in which control information may be transmitted. That is, control information may be transmitted over the same lines over which data is transmitted or may be transmitted over lines that are separate and, indeed, that may be separate lines for each type of control function. We shall speak as if the former were the case, but shall have to recognize the other possibilities in the discussion of <u>select routines</u>. (For the sake of definiteness, we will take as our running example the hardware organization of the IBM 7090, the 7909 data

channel, the 7640 hypertape controller, and the 7631 disc file controller.)

Let us first take the case of control information. We may suppose that this originates in words transmitted from the main storage to a channel. Commands are those words that initiate and control the action of the channel itself. Orders are those words that initiate and control actions of a controller. (Additional control lines will be required between processor and channel and between channel and controller, but we speak as if the bulk of control information were in the form of control data and originates in main storage.) The term "instruction" will be reserved for imperative words interpreted and executed by the main processor. It may be the case that commands or orders are executed sequentially qua program by the channels and controllers. In this situation, we will speak of channel programs and controller programs as opposed to the main processor program.

A channel program may, in the general case, consist of at least the following types of commands:

1. Control: Transmit an order or a sequence of orders to the controller.
2. Read: Set the channel, controller, and unit for reading data.
3. Write: Set up everything for writing.

4. Sense: Request the controller to send back status data (i.e., error conditions, unit ready, end of file, etc.).
5. Copy: Given a main storage origin, maybe a word count, and whether read, write or sense is to occur, start transmitting data.
6. Set Mode: Set the channel for any peculiar information translation to be accomplished during data transmission (e.g., binary or decimal mode).
7. Select: Connect up to a specified controller.
8. End: Signal the main processor that the channel program has ended (usually, via a trap).
9. Branch: Branch in the channel program, possibly conditionally.

An order, or a controller program, will exercise the various features of the input-output units attached to the controller. The only common order is probably:

Select: The following orders apply to a given unit attached to the controller.

In the case of tape-like devices, we might have: rewind, unload, forward space record or P-file, backspace record or P-file, carriage return, change ribbon color, ring gong, sound Klaxon, etc. In the case of drum-like devices, we might have: locate initial address for transmission, advance film frame, select plotting character, turn on a light, select switch or switch bank, ring gong, etc.

We must distinguish between those input-output operations that result in data transmission and those that result only in control operations (control operations will usually be necessary in either case). We will call these *data* versus *non-data* operations. The word "select" will be used in the wide sense to denote an entire input-output operation from its initiation by the main processor to its final conclusion, signalled by release of the channel. Thus we will speak of *data selects* and *non-data selects* to denote the two major types of input-output operations in their entirety.

Of great importance is the manner in which the main processor reacts (or does not react) to the completion of a select. It is generally true in our current larger machines that select completion results in a *trap*. That is, the location counter is stored in a fixed location and the instruction at another fixed location is executed. The details of this action differ considerably among different machines. For instance, non-data selects may never result in a trap at completion, traps may not be enabled at all, or a trap may occur when the channel senses an unusual condition -- such as a parity check -- even though the channel program has not reached completion. (We will call the latter type *unusual end*, as opposed to *normal end*.) A trap may occur even though the channel is not engaged in a select (for instance, a typewriter demands attention, a tape has just been loaded, a rewind has been completed, a

disc access mechanism has finished positioning itself, etc.). This class of conditions will be called attention requests.

We should note that the above terminology has, for the most part, been drawn from that associated with the IBM 7000-series machines, not because it is universally accepted, but because hardware counterparts exist for many of these terms and, hence, the terminology has at least some claim to priority.

SYMBOLIC INPUT-OUTPUT

Up to about five years ago, it was considered that the best practice in handling choice of input-output units by the object program was to include unit assignment as an assembly parameter or to read in unit assignments as data and initialize the program appropriately. This practice worked pretty well when it was followed, which was seldom. With the advent of near-universal use of supervisory systems, including a few that cooperate with the operating personnel in making unit assignments, it has been necessary to find a more foolproof and flexible manner of operating. The solution employed in the SHARE Operating System (SOS) [2-8], and with variations in many later systems, has been to reserve part of high-speed storage as a communication region in which unit information is kept. This solution will be covered more fully in the material on supervisors; suffice it to say here that unit assignment and status information is considered as part of the communication region, accessible both to processors operating under the supervisor and to object programs.

In SOS, input-output units are divided into three classes on the basis of usage: those units used by the system (library tape, peripheral input and output tapes, and scratch tapes for use by the various processors), tapes reserved for the use of a particular programmer or job (i.e., assigned to the job and kept in the tape vault), and tapes

which were on the machine and could be assigned as utility tapes for any job. For each of these functional units, a word was reserved in the communication region called an input-output unit control word (UCW) and a symbol of the form "SYSxyz" was placed into the system symbol table to allow reference to the control word.

The job of the unit assignment routines, then, was to fill the input-output unit control word with the proper channel and unit address when one of the units had an absolute equivalence assigned to it. At the same time, the unit status list was updated to give the unit status now current (detached from the machine, attached and available, assigned as system tape, assigned as utility tape, or assigned as reserved tape).

The Input-Output Control System for the 709 (IOCS) has followed a slightly different approach [9]. In this case, a unit control block (UCB) in the communication region is permanently associated with a physical unit. The unit control block contains the channel and unit address as well as unit status and type information and unit position information (i.e., for tapes, the current physical file and record numbers are kept up-to-date). There are, in addition, file blocks associated with the buffering system and object code. Each information file has its own file block and the origin of this is addressed by the object code. The file block, in turn, contains the address of the

UCB for the unit to be used for the file. Additionally, part of the UCB may be used by the buffering system, as is the case with the UCW used by SOS.

What is grandly termed "symbolic input-output" has at least two characteristics: (1) object programs refer to storage cells rather than to absolute unit addresses, and (2) unit assignments are made by the system and need not be known by the programmer in advance. This already represents a considerable advance over earlier treatment of input-output. For instance, it makes it possible to run programs which use differing normal assignments of peripheral tapes without making changes in the program or reassembling it. It also allows, say, a program written for a four-channel machine to run on a two-channel machine, albeit at reduced efficiency. To be able to do the latter, additional conventions besides the above two must be followed by the program; the program input-output logic must be so arranged that a select is not started on any given channel before the results of the prior select on the channel are determined, stored, and acted on. This matter will be explored more fully below.

Another way in which input-output may be "symbolic" was referred to in the section on terminology. Not only may the unit assignment affect addressing, it may also affect the choice of type of unit. For instance, the information file associated with a file block may be stored

on either tape or disc without the programmer knowing which will be the case. Again, a card deck may be read on-line or from a peripheral input tape -- it is possible to so arrange things that the program need not know which is the case, although the programmers may wish to take advantage of knowledge of the normal input-output arrangements.

Input-output on a machine equipped with input-output traps is a concrete example of multiprogramming. On the one hand, we have the trap supervisor which takes control during trap (i.e., at the conclusion of a select) and finally surrenders control back to the program that is using the input-output system; on the other hand, we have the subprograms which the trap supervisor calls to check the previous select and to start the next one -- these are part of the using program although only the trap supervisor ever transfers control to them. "Using program" in the above context might equally well read "buffering system," where routines in the buffering system are used as subroutines of the programmer's object code.

A more detailed picture of the input-output control situation is given in Fig. 2; it is the object of the following section to explain the picture.

We shall consider that an input-output system is composed of the following:

1. One or more buffering systems. They maintain the file blocks, communicate with the object

BLANK PAGE

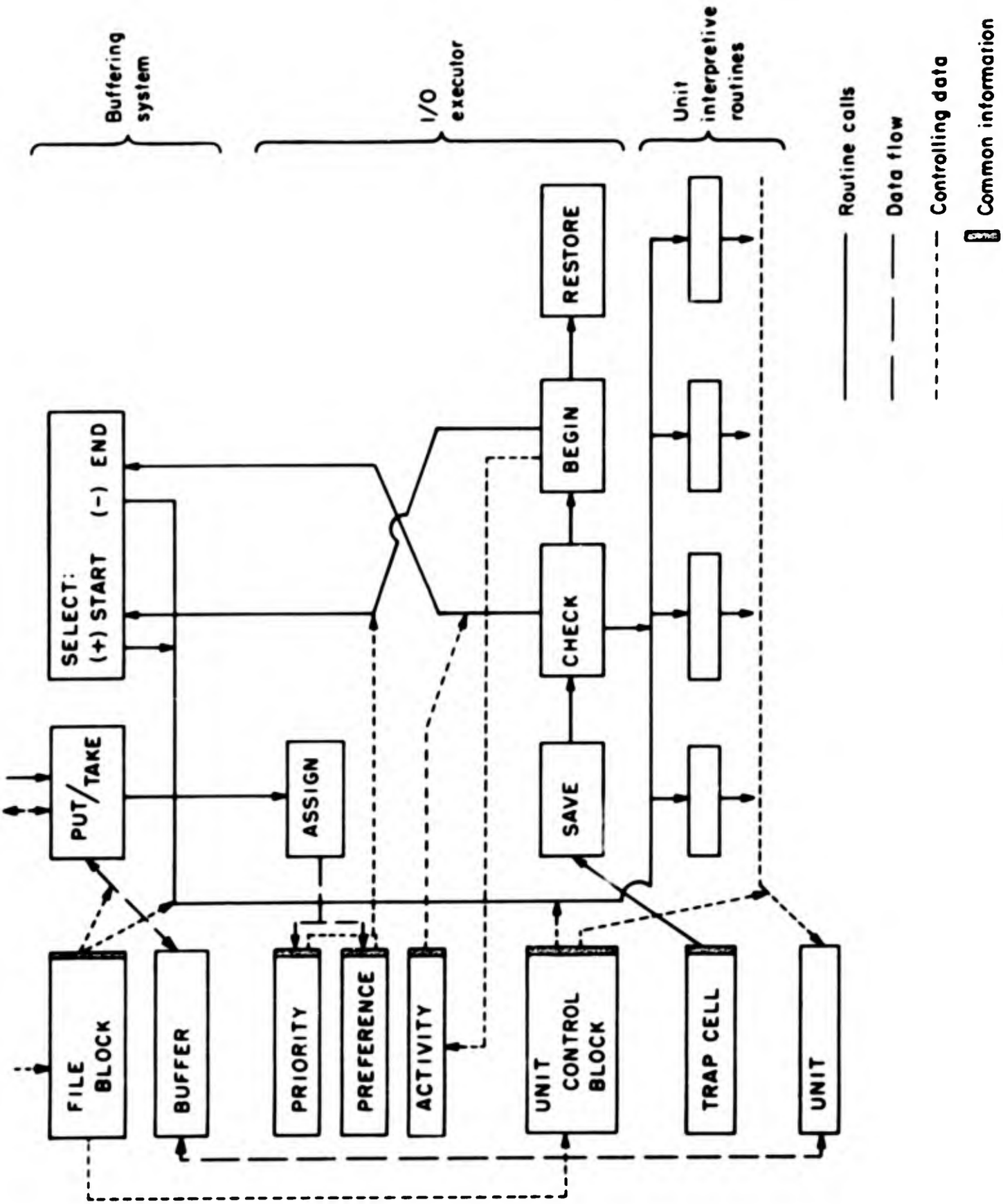

Fig.2—Input-output communication

code, put information to be written into buffers from main storage, take information read out of buffers, communicate with the input-output executor, and provide the select routines to be used at trap time.

2. The input-output executor (IOX). This is the trap supervisor mentioned above.
3. Unit interpretative routines (UIR). Each of these provides an elementary function used by the select routines, such as compute channel number, find absolute unit address, set up to write, set up for a non-data select, check reading.
4. Communication cells. File blocks and unit control blocks are examples.

More than one buffering system may exist in storage at the same time. It may be, for instance, that there is not room in storage for a massive general-purpose buffering system such as IOCS or the SOS buffering system while there is room for two or more special-purpose buffering systems. It might be the case that the general buffering system in use for most purposes will not accommodate blocking and unblocking peripheral tapes and, hence, a special buffering system is required for this purpose. It might also turn out that a part of the object code is using IOX directly to accomplish non-buffered input-output. This does not

exhaust the list of possibilities, but is indicative of the flexibility that is often required in an input-output system.

The unit interpretative routines (UIR) are singled out because they have a well-defined function which is not necessarily related to either the buffering system or IOX. It is their job to accomplish a given input-output function for a given type of channel and/or unit. This job can be done by bit-picking inside the select routines, but we have chosen to separate it out since it can be done better this way in practice.

The buffering system knows the intimate details of how the buffers are used, what is in the file blocks, etc. It need not, however, know the details of input-output coding for dealing with the particular units being used -- only how to communicate with the unit interpretative routines and IOX. Similarly, IOX need know next to nothing about the buffering system or the unit in use -- its job is trap supervision and the manipulation and use of the communication cells.

INPUT-OUTPUT EXECUTOR

IOX has the overall function of maintaining channel activity. That is, it is entered as the result of completion of a select and must see that the result is checked and that a new select is started if this is possible. The basic problem in doing this is in designing a good and sufficient set of communication conventions, especially under the policy that IOX should not know the details of the buffering system(s) in use or of input-output coding for controlling units being used. The kind of solution we shall indicate is a slight generalization of the methods used in SOS and IOCS. We will assume the situation in which each controller is permanently associated with a particular channel, as on the IBM 7090, waiting until later to detail how changes in assumptions affect IOX. We will assume that some channels trap only at the completion of a data select but not on completion of a non-data select or receipt of attention signals. It should be understood that once a trap has occurred, all other traps will be inhibited until exit from IOX, whereupon a delayed trap may take place.

It should also be kept in mind that the details that follow continue to use the IBM 7090 as an example. In particular, a storage cell in that machine can hold two machine addresses. This is not essential to the argument, of course, but should be kept in mind to avoid confusion. Many of our communication conventions will be so labeled.

Therefore:

C1. The first word of each file block will hold the location of its associated unit control block. IOX will be aware of no other aspect of file block format.

C2. The first word of each unit control block will hold:

1. The absolute unit address.
2. The absolute channel address.
3. The absolute controller address.
4. The location of the unit interpretation routine list for that unit type.

(The format for 1-3 may be various, but known by the UIR list.)

Channel Communication Cells

For each channel, IOX must know what select routine is to be used to check the current select on its completion and what select routine to use to initiate the next select on the channel. The select routine must know whether it is being entered at the start or end of a select and also what file block is involved (and, hence, what unit, what core area, what kind of a select, etc.). Words containing this information (i.e., location of select routine and location of file block) will be called select words. SELECT(+) will be the generic name for the

routine that starts a select and SELECT(-) for the routine that does the appropriate end action.

C3. IOX will enter select routines with the select word in the accumulator and an indication (multiplier-quotient register sign, for instance) as to whether start or end action is desired.

Select words will be stored in the channel communication cells. There are three possibilities:

C4. A channel dispatcher preference cell, if non-zero, holds the select word for the next select the dispatcher wishes to take place on that channel.

C5. A channel system priority cell, if non-zero, holds the select word for the next select on the channel. If present, this will override dispatcher preference.

C6. A channel activity cell, if non-zero, holds the select word for the action to take place at the conclusion of the current select. The activity cell will be non-zero if and only if the channel is active and the current select will end by causing a trap.

The essential difference between a dispatcher preference and a system priority is that in the latter case the object program is being held up until the requested select is completed. This may happen, for instance, if the program

attempts to read (i.e., take words out of a buffer) before a buffer has been assigned and loaded from tape. If a priority action is to take place, we do not necessarily wish to cancel the previous preference.

The dispatcher referred to in (C5) is a routine that fills the dispatcher preference cells. It is, in effect, the part of the buffering system which determines input-output strategy as far as channel usage is concerned. It will be considered in more detail in the next section. From our present point of view (IOX, that is), we can state two requirements:

C7. IOX must be able to call the dispatcher for each buffering system in main storage when a channel is free and no dispatcher preference is set.

C8. A dispatcher preference, as opposed to a system priority, is revocable. That is, a dispatcher may decide that on the basis of current conditions it prefers doing other than what its previous preference was. This implies that the buffering system may not assume that setting a preference will necessarily result in the requested select being accomplished.

One other point: the proper place for a dispatcher is in the buffering system rather than in IOX. To dispatch, one must be able to consult the file blocks. IOX, on the other

hand, is oblivious of their content, being a mere slave of the traps and the channel communication cells.

From our present point of view, then, the action of IOX can be represented by the flow chart in Fig. 3. A few things have been left out:

1. After determining the channel, IOX should collect the sense data (i.e., existence of parity checks, etc.), since this can be done by using the proper UIR and must, in general, be done in all of the select routines if IOX does not do it.
2. On exit from SELECT(-), corrective action may have made it necessary to issue a new select. In this case, IOX should skip down to restore the console and exit. This would happen, for instance, on a reading error where a reread has been started.
3. IOX should clear any channel communication cell whenever it is about to use the select word stored there.
4. If SELECT(+) starts a select that will not result in a trap, it should return zero and otherwise return the select word to be stored in the activity cell. In the former case, there is a legitimate question as to whether IOX can go back to look for a new priority or preference

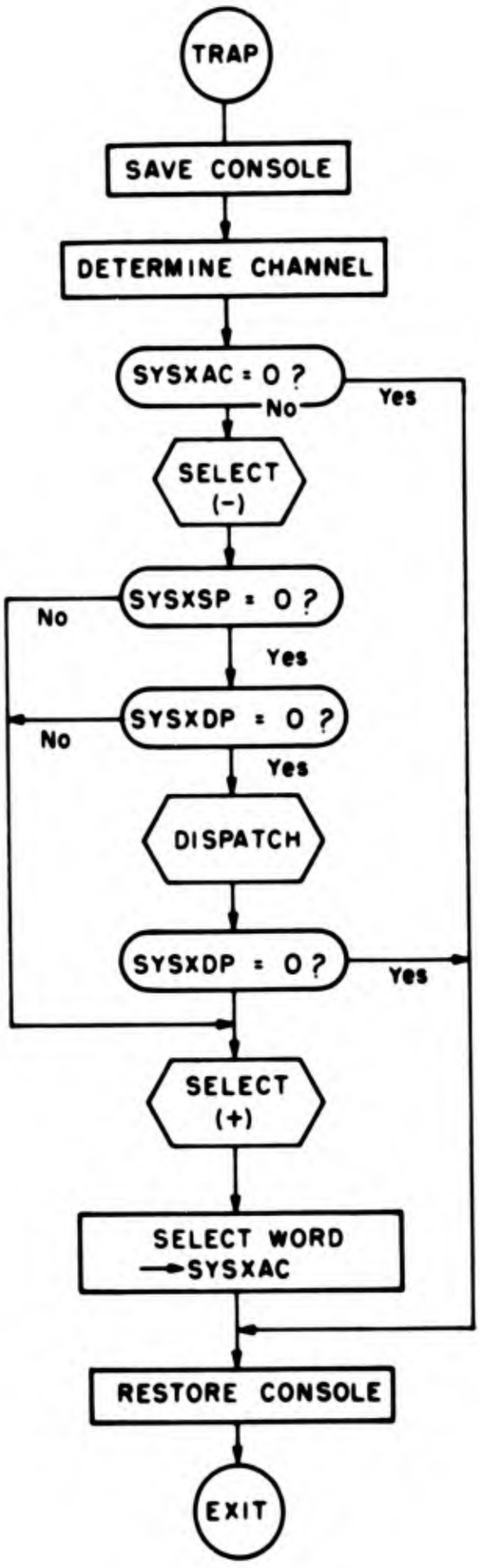

Fig. 3 — IOX action

to execute before its exit -- this is a matter of whether the main frame could be held up if it attempted to reuse the channel or controller. The SELECT(+) routine should know this, and could exit accordingly.

The ASSIGN Routine

Something must be provided to simulate the action of IOX in starting a channel once it has been allowed to fall idle, since IOX is entered only via a trap. It would also seem to make sense to use the same routine, given a select word and a particular sense of urgency, to fill either the priority or preference cell or to wait until the next trap on the channel and exit only when the select is started. The possible degrees of urgency are as follows:

1. System priority: If the channel is idle, enter SELECT(+) and then fill the activity cell. If the channel is busy, fill the priority cell. In either case, do not exit until the corresponding SELECT(+) has been executed.
2. Strong preference: If the channel is idle, enter SELECT(+) and post the activity cell. Otherwise, fill the preference cell.
3. Weak preference: Same as (2), but fill the preference cell only if it is cleared.

A point worth making is that ASSIGN may be entered either at trap time from a dispatcher via a SELECT or at non-trap time from the buffering system. ASSIGN may, in fact, be in simultaneous use in both ways. The IOX save and restore will take care of collision of console contents. Linkage information must, however, be stored in different places, depending on when ASSIGN was entered. A similar remark is true of the dispatcher(s) and other routines in the buffering system. This is the subject of trap protection, which will be taken up in the next section.

Attention Requests

On a machine whose channels may accept attention requests from the units (e.g., the IBM 7090), the IOX picture is more complex than was indicated above. This is especially the case if an attention request may occur on a channel that was otherwise idle or if the channel program can be interrupted by the attention request before it has come to normal or unusual end.* The possible cases to be considered, then, are:

*On the 7090 equipped with the 7909 data channel, unusual end or attention request will result in an interrupt (i.e., a trap in the channel program). The interrupt program may, if it wishes, then trap the main processor program. If interrupt occurred before completion of the channel program, the main processor program may restart the channel from the point at which it was interrupted. On other machines, a trap may result directly (e.g., the IBM 7040/44).

1. Channel was not active (i.e., nothing was posted in its activity cell).
2. Channel was active and attention request came from the selected unit.
3. Channel was active and attention request came from a unit other than that stated.

In the last two cases, subcases arise depending on whether or not the attention request came before normal or unusual end. (It is, of course, possible to have more than one unit requesting attention at the same time.)

It is clear that our present communication conventions are not sufficient to handle the situation in any way. The object program or buffering system may want to get control when particular units request attention -- if this is not the case, then IOX must have means for dealing appropriately with a request from that particular type of unit.

In the second case, it seems highly unlikely that an attention request could come before normal or unusual end. A more likely case would be, say, a request following termination of disc access mechanism motion or a tape rewind. The question here is whether the activity cells should have been posted in the first place. A reasonable supposition is that the SELECT(+) routine might know whether anything is to be done after the attention request comes along. For instance, if a disc read is to take place and SELECT had to accomplish a seek first, then SELECT must regain control

at completion of the seek. If this was as the result of a system priority, then SELECT(+) might keep the activity cell full to deny use of the channel to anyone else -- we will discover a better alternative below. Otherwise, however, it should be possible for someone else to use the channel during the seek (or rewind) so long as they are prevented from selecting the unit that is in seek or rewind status. This would result in the third case when the request finally came up.

In the final case, it is pretty clear that anything done by IOX to satisfy the attention request must not immediately result in starting a select. This is true even if the request comes at the same time that the channel ends activity, for an error condition may make it necessary for SELECT(-) to restart the channel. The most, then, that IOX can do is permit a system priority or dispatcher preference to be set as a result of servicing the attention request. Now, servicing the attention request must have fairly high priority, particularly in the case that a select routine was entered on system priority and must be re-entered in order to do the rest of its job. On the other hand, an existing system priority cannot be killed as the result of servicing an attention request.

A possible way of handling these situations is the following: Once IOX has collected the channel sense data and has determined that one or more attention requests are

up, it will execute an attention recognition routine for each unit requesting attention. This will be one of the UIR's and will, in general, set a flag in the UCB for later use. If this was also normal or unusual end and the activity cell was filled, IOX will now do the SELECT(-). If SELECT(-) exits saying the channel is again busy, IOX will exit, but otherwise the channel is free and further action can take place for the attention requests. If this was not an end, of course, IOX must restart the channel and exit.

Further action is necessary if the buffering system requested IOX to report back on an attention signal from a particular unit; this request can be made by storing a select word in the UCB (in its second word, say) on exit from the original SELECT(+). In this case, the UIR would have set the attention flag once the attention signal was recognized. IOX, therefore, should examine all UCB's on the channel and treat any attention select words exactly as if they had come from an activity cell. This should be done before setting up for a SELECT(+), since an input-output action that has already been started should have top priority for completion. This leave us with a more complete flow chart for IOX, shown in Fig. 4.

IOX action is now broken down into six phases:

1. Collect sense data and recognize attention signals.
2. Check previous channel activity, if any.

BLANK PAGE

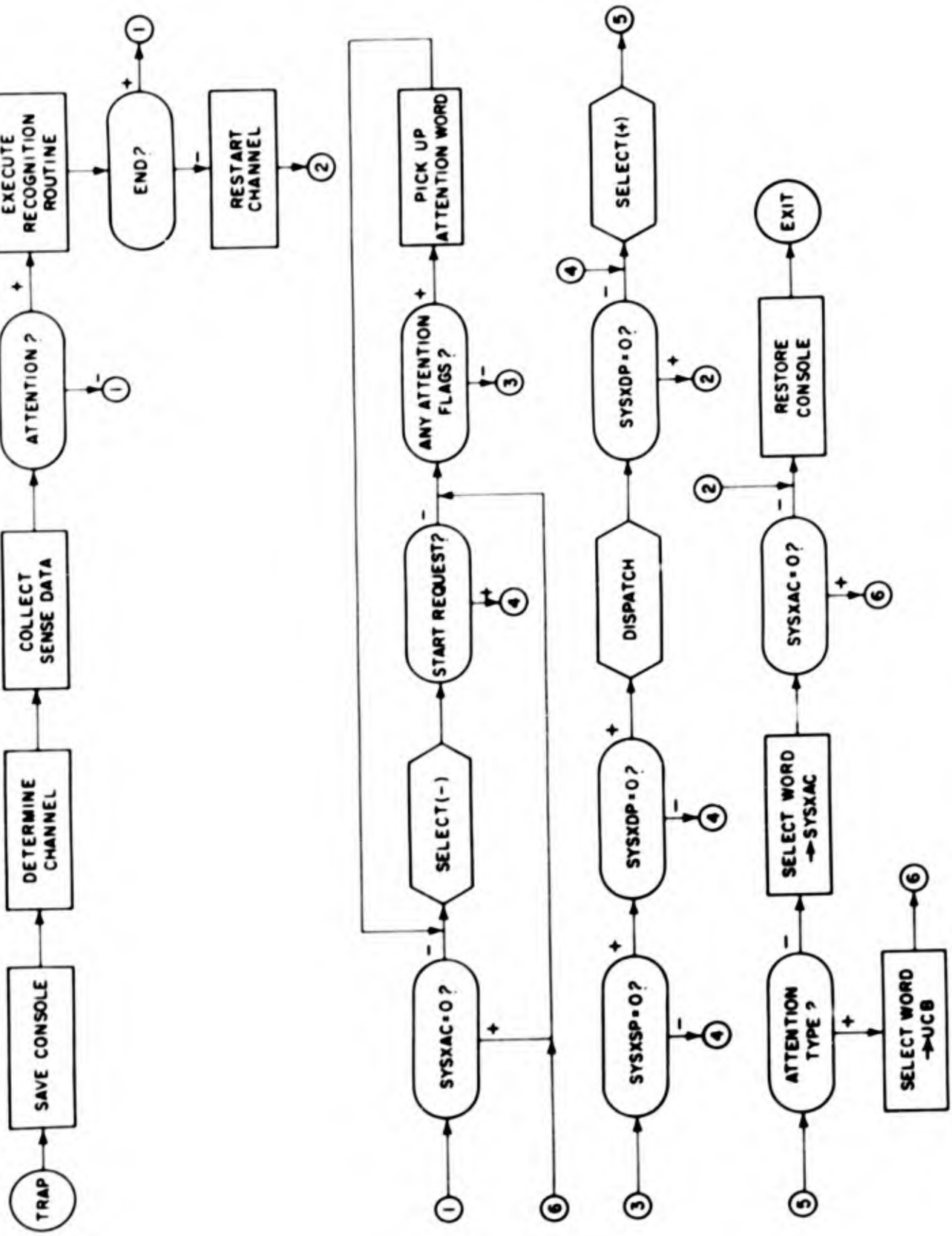

Fig. 4 — IOX Revised flow chart

3. Service attention flags by treating them as a cessation of activity. Note that SELECT(-) must exit with a select word if it wishes to start the channel -- IOX will then treat this as a SELECT(+).
4. Execute SELECT(+), given a select word from SELECT(-), the system priority cell, or a dispatcher preference.
5. If SELECT(+) indicates an attention type of activity, store the select word returned in the UCB. Otherwise, store it in the activity cell. If there is now no activity (SYSXAC = 0), go back to (3) to service other possible requests.
6. Restore the console and exit.

Other Assumptions

The two principal assumptions made in the foregoing which are not true on many machines concern the existence of traps and whether or not controllers are permanently assigned to channels. The case with no traps is easily disposed of: we keep our existing communications conventions and supply a routine which takes over the function of the trap supervisor with respect to handing the channel communication cells and entries to the select routines. This test routine will test each channel, and, for the channels not busy, will go through the action of Fig. 3.

It will be up to the buffering system and object program to enter TEST often enough to maintain channel activity.

In the case of a machine like the Philco S-2000, it might be more appropriate to keep only one priority and preference cell and to use part of each UCB as the activity cell for that unit. If it were possible, however, to discover what channel is assigned to a given activity and which channel just completed its activity, then activity cells could be kept on a channel basis. In other words, the detailed organization of IOX on such a machine might be quite dependent on the hardware, even though we might expect the general organization to look very much like that outlined above.

BUFFERING SYSTEMS

The basic idea behind input-output buffering systems is quite simple; rather than use only one area in main storage for data to be read or written, we use several in such a way that we can be loading one area while we are processing the last data that came in or emptying one area while we are producing new output data in another. In this way, it is usually possible to avoid input-output delays due, say, to the fact that we insisted in reading into only one area and, hence, could not start the next read select until we had completely finished processing the last set of data we read.

There are a number of buffering systems in existence today; in fact, there are some machines for which a number of buffering systems have been written. We will be more interested in the following in describing some fairly general features of these systems rather than in describing any particular system in detail.

Alternating Buffering

This is the simplest and most generally used method of buffering. With each read or write routine (or with each file block, which is preferable) are associated two buffer units, each one at least as long as the physical record that must be read or written. That is, the length of the buffer unit is equal to the block size. At any given time, one of the buffers is participating in a data

select and the other is associated with the program. Once an output buffer is full (i.e., ready for a write select) and the previous write select is ended or an input buffer has been used and the other buffer has been read into, the roles of the two buffers are switched.

The picture that illustrates our terminology is Fig. 5. "A" and "B" are two buffers. Input consists of loading a buffer (i.e., a read select into the buffer) followed by taking the information out of it; output is the act of putting data into a buffer and later emptying it out (i.e., a write select).

Note that buffer A is always associated with a load or empty and B with a put or take. This implies that the buffers themselves must, in effect, be treated symbolically. That is, "A" and "B" really stand for the buffer of the pair that currently assumes the role indicated.

To get down to cases, consider a read or input routine programmed without using IOX, as in the flow chart of Fig. 6. At entry to READ, the presumption is that we are through processing the last buffer load (B) and that the next record (A) has been loaded or is still on the way in.* Step (1) ensures that the load is completed. Step (2) does the work of SELECT (-), causing a few reload tries in case of parity error or a special exit from READ in case of end of file. Step (3) alternates the role of the two buffers.

*On initial entry to READ, we have to do step (4) first and then come back to (1). This action will be called priming.

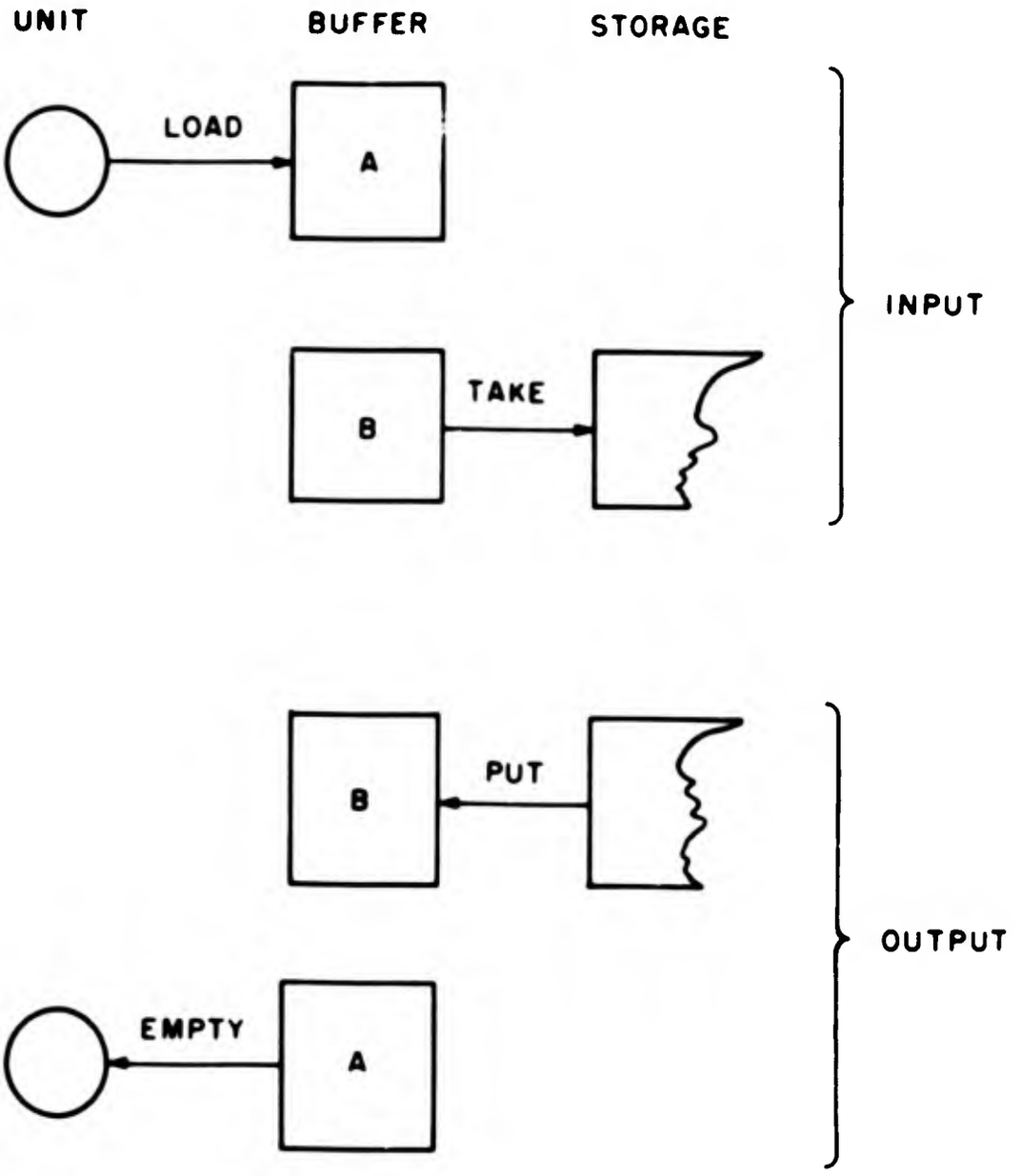

Fig. 5 — Alternating buffering

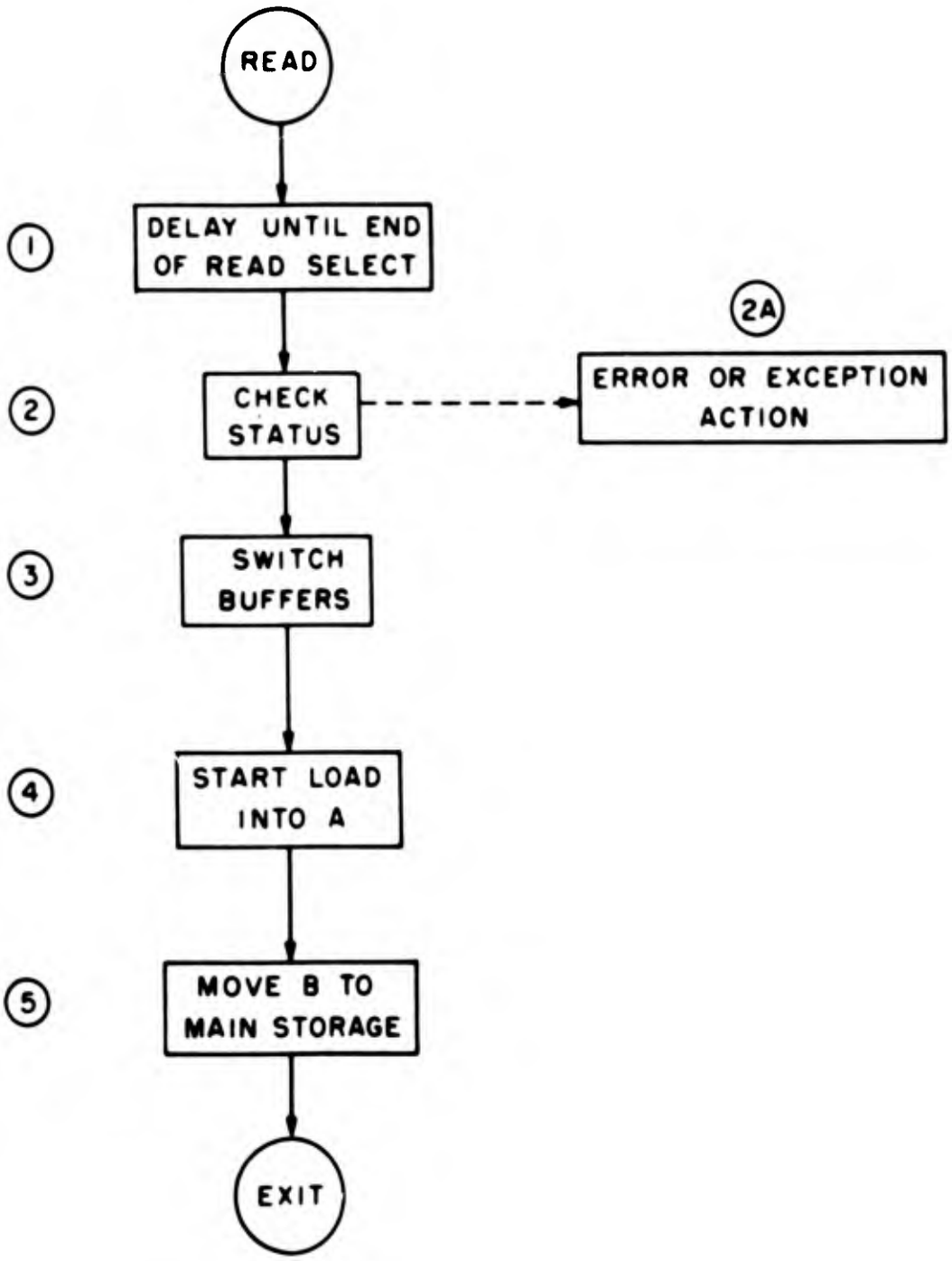

Fig. 6 — Alternating buffer read (no IOX)

This could be done by so dull a means as moving the contents of B to A. Another way is to exchange the contents of two addresses in the file block that point to A and B. Step (4) is the SELECT(+) action for the new buffer B. Step (5) either moves the contents of the new A into a working area in main storage (transmit) or makes the location of A available to the program calling READ (locate).

Alternating buffering is normally used in the locate mode; if the transmit mode were being used, there would be need for only one buffer, since we can do the transmission before starting the next load or empty. In a case where the routine calling sequence can request either locate or transmit, however, two buffers must be used as described above.

It has very often been the practice to associate the two buffers with a routine, rather than with a file block. This is expedient, especially in the absence of input-output communication conventions, but frequently rather wasteful of space or time. Suppose, for instance, we have one routine for each action and only four buffers, but four tapes to deal with. On reading, our only possible strategy is to assume that the tape last read will be read next. If, on entry to READ, a new input tape is specified, we must backspace the old one, since we were premature, and then prime the routine by reading and delaying. (Unfortunately, a card reader cannot be backspaced!) In the case of input

to a processor, this strategy is fairly reasonable, but it is costly if we switch input tapes very often. The situation is not so bad for WRITE, so long as we are preparing output only for one unit at a time.

It is preferable, however, to associate buffers with the file block. In this way, we may be able to get away with one READ and one WRITE routine and yet keep several input and/or output files open at the same time. This takes more buffer space, but this effect can be largely alleviated by use of buffer pools (discussed further below).

Let us now consider how to link a read/write routine of this type with IOX. This will require a select routine and some way of checking that the previous select has been completed and then starting the next one. If we have a dispatcher, we must have a way of informing it that this file needs action -- this could be done by storing the select word in a specified place in the file block, given a dispatcher that checks all file blocks for requests each time it is entered and sets preferences for each channel on this basis. (Two other methods of dispatching will be mentioned later.) With this method, then, we post the select word in the file block and call the dispatcher. If, however, no dispatcher is available, we must enter ASSIGN to get the select word posted in the preference cell. We should not be so impolite as to use the system

priority entry, because we may be disrupting activity for a file block that has a much higher level of activity than the one we are now working with. It is true that this preference may be clobbered before the select gets started -- this is why we should have a dispatcher around.

The job of the select routine, once entered from IOX, will be to start the channel using the UIR's and information from the file block such as mode, buffer origin, and buffer length. SELECT(-), later on, will have to check the transmission, store information such as end of file or unrecoverable transmission error back in the file block for later reference, and store an indication in the file block that transmission has been completed.

At the next entry to the read/write routine from the object program, we must check the file block to see if the transmission is completed. If it is not, we must enter ASSIGN with a system priority and then wait until the transmission complete indication comes up in the file block. A more detailed flow chart for the read/write routine, assuming a dispatcher and the locate mode, appears as Fig. 7.

Pool Buffering

As stated above, it is good practice to associate buffer units with file blocks rather than with routines. When more than a very few file blocks are in use, however, the total space requirement for buffer units can be

BLANK PAGE

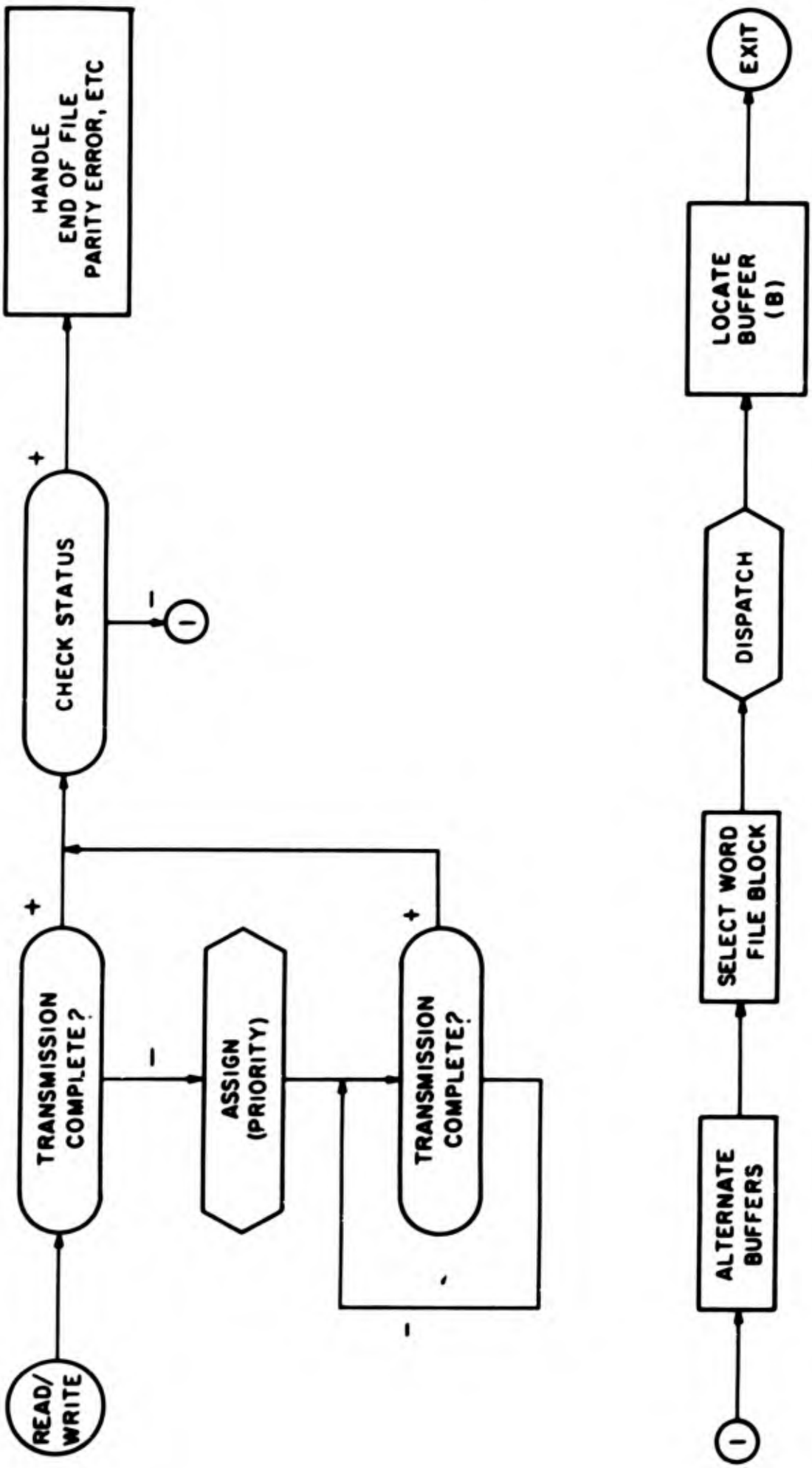

Fig. 7—Alternating buffering with dispatching

considerably larger than the amount of space that is actually needed at any time during object program execution. The natural solution to this problem is to establish one or more pools of buffer units and attach a buffer unit to a file block only when it is actually required. More than one pool may be desired if, for instance, two file blocks are associated with units whose block size is 28 words and others are associated with units whose block size is 256. In such a case, the name of the pool to be used must be stored in the file block.

Pools are generally organized in the fashion indicated in Fig. 8. Namely, buffers are chained together like NSS lists, each buffer containing the name of the next buffer in the chain. The pool control word(s) contain the location of the first buffer in the chain assigned to the pool (i.e., not otherwise in use) and the file blocks contain the location of the first buffer in the chain currently assigned to the files. The pool control word and/or the buffer control word(s) -- the first word(s) in each buffer unit -- also contain the block size for that pool. Fig. 8 shows buffers B3 and B1 assigned to the file and B2 and B4 assigned to the pool, or available for use.

The buffer control words may also contain a buffer pointer, or the address of the next word in the buffer into which data may be put or from which it may be taken, and a word count which shows how many data words are

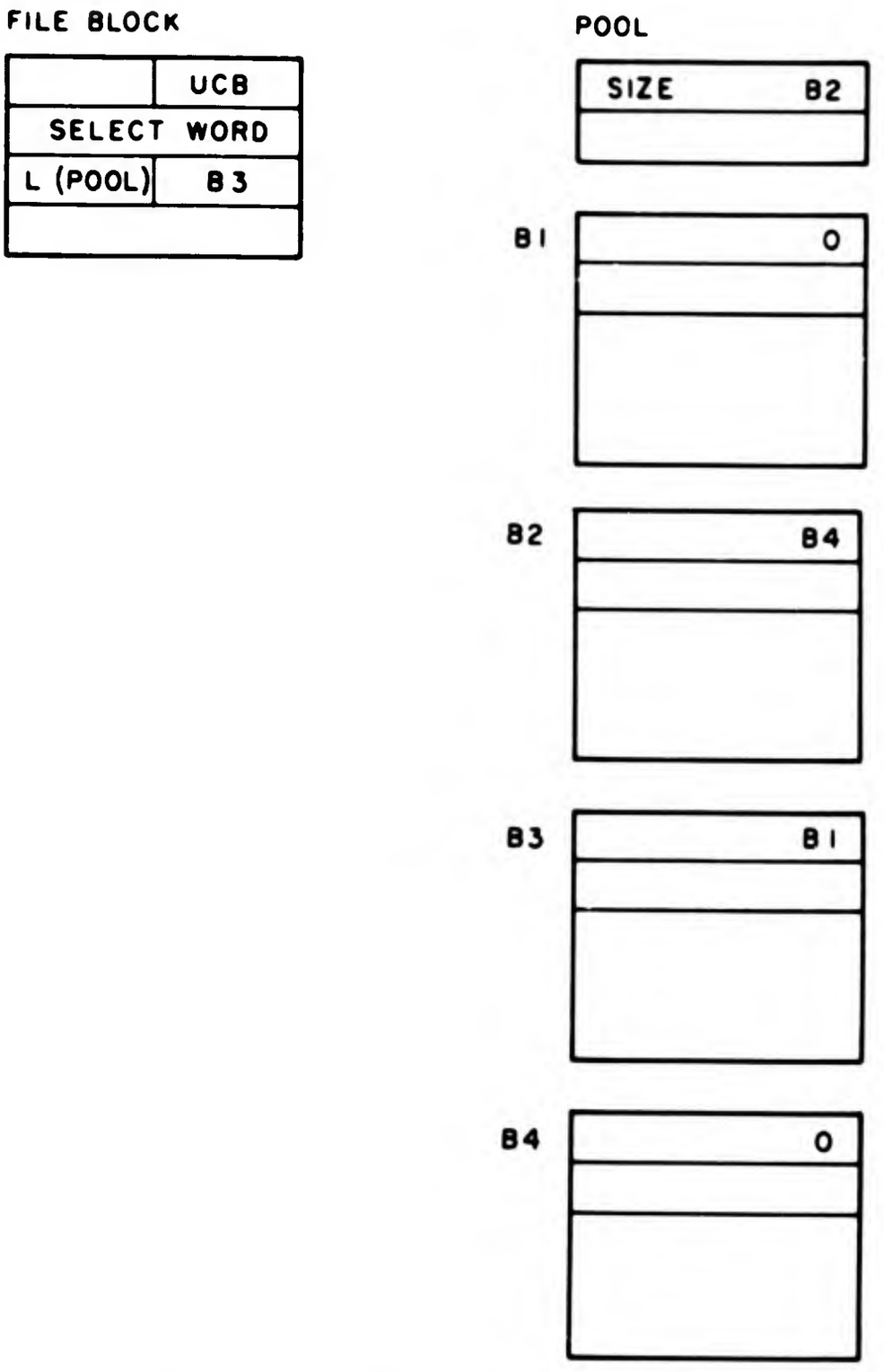

Fig. 8 — Buffer pool organization

actually in the buffer. (Recall that physical records may not be as long as blocks.)

Since we now have the possibility of attaching more than two buffers to a file block, it becomes possible for one read select to load more than one buffer (bad practice) or to read several records ahead with several read selects. Conversely, we can put data into several buffers without having to empty each out as soon as it is full. These possibilities were not available using alternating buffering. Furthermore, one usage of a read or write routine may involve putting or taking over buffer boundaries without necessarily encountering input-output delays. As soon as we do this, however, new problems arise.

When discussing alternating buffering, we indicated that it was reasonable to store status information in the file blocks. Now that a buffer chain of indefinite length may be attached to the file block, it becomes more reasonable to store status information in the buffer control words. Some of the terms used to denote buffer status are the following:

Moving: This buffer is currently involved in a read or write select.

Active: This buffer is currently involved in a put or take.

Quiet: This buffer is full of data.

Held: This buffer has been used by a locate and hence may not be returned to the pool just yet.

The first three terms are mutually exclusive. At any time, only one buffer can be moving and only one can be active; the others must be quiet. A held buffer may be active or quiet. SELECT(-) routines will change the buffer involved from moving to quiet -- this is the signal that transmission is ended as far as the read/write routines are concerned.

Another problem that pool buffering introduces is the possibility that when someone wants a buffer assigned to a file block, the buffer pool is empty. When this happens, there are only two possible ways of getting a buffer. First, one or more quiet output buffers may exist which can be emptied out. Second, one or more quiet input buffers may exist which can be used at the cost of backing up the unit from which they were read and rereading later on. The first process is called draining and the second flushing. Clearly, draining is preferable, if it is possible. If neither is possible, the program has collapsed of its own weight.

Another problem encountered is also related to the distinction between the logical position of the unit, as indicated by the buffer pointer in the active buffer, and the physical position of the unit, which is generally different. This problem arises when we switch from reading to writing or when we wish to accomplish a logical backspace following reading or writing. On a switch from

read to write, all quiet buffers must be flushed, and the unit must be backed to the beginning of the active record which will have to be rewritten. To backspace after a read, we must also flush and then behave appropriately according to the definition of logical backspace, which may be various. To backspace after write, we must truncate the current active buffer (i.e., ready it for a write select and make the buffer quiet) and then drain all buffers. The effect in all of these cases is to make the logical and physical position of the unit equal, treating the current active buffer appropriately.

Logical Flags

As we have seen, in the case of alternating buffering, a read/write routine usually handles a single physical record at each entry and locates it for use by the calling program. With pool buffering, the situation can become much more complex. IOCS, for instance, makes it possible for the calling program to do a combination of transmits and locates at each entry, but the format of the physical records must be completely known by the calling program. In the MockDonald buffering system of SOS [6], however, it is possible for the read/write routines to work with logical records, labels, etc. This is done, roughly speaking, by interspersing flags, or control words, among the data. Flags occur in almost bewildering variety, although there are only two basic types of flags, called block and

BLANK PAGE

whyte (sic).* Block flags precede blocks of information or data and contain the count of the length of the block. (This usage of the word "block" is not to be confused with our previous usage.) Whyte flags indicate logical divisions of data or data type. The roster of flags used in SOS include the following:

Block flags:

IOCP ORIGIN,,COUNT	Ordinary block flag
IOSPN ORIGIN,,COUNT	Symbol flag (for labels)
IOSPN SEQUENCE,,0	Sequence flag (for serializing)

Whyte flags:

Logical end flags:

TCH SYSPER,,COUNT	End of physical record
TCH SYSLER	End of logical record
TCH SYSPEF,,0	End of logical group
TCH SYSPEF,,1	End of logical file
TCH SYSPEF,,2	End of logical tape
TCH SYSPEF,,8	End of physical tape
IOCD TYPE,,SUBTYPE	Type flag
IOCT START,,0	Transfer flag

*This word is the result of an historical accident. In transcribing the proceedings of the SHARE 709 System Committee, a secretary insisted on reading "block" and "non-block" as "black" and "non-black." Thus, terminology evolves!

Flags are actually 709 channel commands, for a reason which we will not discuss further than state that this makes it possible to load the contents of tapes into core without use of the buffering system.

One of the main jobs of the SOS buffering system is flag interpretation. The buffer pointers always point to the next flag in the buffer. When reading logical records, only block flags and logical end flags are recognized -- everything else is skipped. When using a read routine that handles everything on a word basis, different exits are provided for block and whyte flags; it is up to the using program to do further interpretation.

This buffering system is probably more elaborate than was really justified for its intended use -- few programmers have exercised all of its capabilities, although the supervisor in SOS does use nearly everything. Due to the absence of a locate mode, some programmers have refused to use it at all on efficiency grounds. Nevertheless, this kind of a system certainly has its place and for many purposes is superior to a system like IOCS. A simpler type of buffering system with flags is very often used in connection with handling peripheral tapes in order to achieve a reasonable information density on tape and to allow several types of peripheral information to be placed on a single tape that will be handled by a satellite computer.

The idea of using logical flags has an interesting application in using random-access media as serial files. The way we can do this is to divide an area of the medium (for instance, disc or main storage) into a buffer pool. The buffer control words for each buffer will contain the location of the previous buffer in the chain as well as that of the next one. We must also use logical end flags. The select routines (actually, the UIR's) will now have the additional job of posting the buffer control words in the UCB's and using them to direct the controllers for addressing purposes. The flags are used to simulate end conditions that would have been present in the corresponding P-files using serial media. In this way, as we stated much earlier, the buffering system can be made unaware of what type of unit is in actual use. In conjunction with this mode of operation, the UIR's must have routines available to transfer buffer contents between the unit buffer pool and P-files. In the case that main storage is being used in this way, no selects actually occur, and with a small amount of trickery, it should be possible to use the same buffer units in both buffering systems and thus avoid transmission of buffer contents within the input-output system.

To return to our previous discussion, the main part of the subroutine hierarchy within a buffering system might look similar to that outlined below:

READ: Uses TAKE to locate buffer and flags. Transmits information blocks to main storage or locates them for use by the main program.

WRITE: Uses PUT to store flags and locate buffers. Transmits data into information blocks or locates them for the main program. Actual transmission for READ and WRITE may, of course, be done by PUT and TAKE.

TAKE: Locates the active buffer, takes flags and information from it, returns buffer to pool using BTP. TAKE will also have to use ASSIGN if it is at the end of the buffer chain and needs a new input buffer.

PUT: Locates active buffer, uses BFP and maybe DRAIN to get new buffers, transmits information and flags into the buffers. Uses ASSIGN to assign a dispatcher preference when a select is needed or, alternatively, posts requirements with the dispatcher.

SELECT: In addition to actually starting the channel, uses BFP and maybe DRAIN to get a buffer for a read select and BTP at the end of a write select to restore the buffer to the pool.

TRUNC: Truncates the current active output buffer, readying it for a write select.

BTP and BFP: Transfer buffers between file block chains and the buffer pool. Additionally, call the dispatcher to reset dispatcher preferences.

DRAIN: Performs a drain or flush action as requested, using ASSIGN on a system priority basis.

Multiple Use of Buffers

One disadvantage of the kind of buffering systems we have discussed above is that a buffer can be attached to only one file at a time. In some kinds of work, one may wish to read in a buffer, write it out on one or more files, and maybe also hold it for use by the main program. The MICA buffering system allows this possibility by methods roughly similar to those to be outlined below.

Let us assume that read/write routines handle a complete buffer at each entry and for each file there is some maximum number, M, of buffers that will ever be attached to the block. If the buffering system employs flags, then the read/write routine may operate on a logical record basis, but no logical record shall extend over more than one buffer. We will then add M words, called clues, to the file block, and each one will hold the location of the buffer assigned (if any). Buffer control words will hold the location of the pool, the buffer pointer, the buffer word count, and the location of the next buffer in the pool. Finally, if the buffer is attached to at least one file block, the number of file blocks to which it is attached will replace the location of the next buffer in the pool. This is called the use count. Note that since we store pointers to the attached buffers in the file block as clues, it is not necessary to chain the buffers together when they are in use.

We now proceed very much as before. Buffers get attached to a file block in one of two ways. BFP will get the buffer from the pool and set the use count equal to one, exiting with the clue. When a buffer is to be copied, its clue is picked up out of the source file block and put in place in the target file block and, at the same time, the use count is increased by one. When a buffer is no longer needed by a file block, we go to BTP with the location of the clue (BTP will reduce the count by one and restore the buffer to the pool only if the resulting count is zero) and then kill the clue. Note that only complete buffers can be copied, but that this method involves no transmission between buffers.

Select and Unit Interpretative Routines

We have seen that select routines must be rather intimately tied to the particular buffering system with which they are used, and we have given one example of a method by which the select routine can be made unaware of the type of unit with which it is working by installing a rather simple buffering system at the unit interpretative routine level. On machines such as the 7090 which require special conversion of information to be used with on-line equipment, as opposed to tape, this conversion could well be done at the unit interpretative level. We have also seen no case in which the select routine or the rest of the buffering system must be aware of the content of the UCB, so long as

unit interpretative routines are used. These considerations can be very important in the case of changing hardware (for instance, the introduction of the 7909 data channel, discs and hypertapes on the 7090, which effectively negated a large investment in programming systems, both on the part of the manufacturer and the customers). It should also be pointed out that, since select routines are properly part of the buffering system, use of a single general-purpose select routine sharply restricts freedom of design of the buffering systems to be used; this is another excellent reason for using select routines tailored to the buffering system and unit interpretative routines tailored to the hardware types in use on a given machine rather than to try to design one select routine that can deal with all possible hardware directly.

We spoke earlier of elementary functions executed by the UIR's. A representative list of these might run as follows:

1. Compute the channel number, given the UCB (used by ASSIGN and the dispatcher to get access to the proper channel communication cells).
2. Put the unit address into the control orders.
3. Set the channel program to copy the block contained in the buffer (i.e., origin and word count).
4. Set the channel program to control, read, write, or sense.

5. Set the channel mode. (This routine may be entered with mode information from the file block -- in general, file block information is carried into the UIR's and they may use UCB information.)
6. Set the channel program to select the proper controller.
7. Check conditions at end of a read, write, or non-data select. (Exits with status information to be ignored or stored in the file block or buffer or, in case of error recovery, so as to make SELECT(-) exit for a retry.)
8. Update unit position information for current type of select and control orders. (May be done as part of another action.)
9. Start the channel.
10. Select the proper exit from SELECT.

This list, together with amendations to actually make the process work for a given machine, may seem overly complex. In the case of many machines, however, the whole process may result in setting up parts of a single input-output instruction and then executing it. In such a case, it might be feasible to combine some of the above functions into one routine. Note, however, that the way in which some of these functions are coded may depend on the type of unit involved. Thus, on the 7090, we would end up with at least five lists of UIR's:

1. On-line card equipment, using 7607 channel.
2. 729 tapes, using 7607 channel.
3. 7631 disc file controller on 7909 channel.
4. 7640 hypertape controller on 7909 channel.
5. 1414 transmission line (etc.) controller on 7909 channel.

Some lists would correspond for most functions and others would hardly correspond at all (e.g., (2) and (4)). Where a particular type of hardware is not present, of course, the related UIR's need not be assembled into the input-output system.

Dispatching

As we have seen, the function of a dispatcher is to fill the dispatcher preference cells according to some fixed strategy. The dispatcher will normally be entered when a buffer is moved to or from an inactive buffer pool, when PUT or TAKE reach the end of a buffer, or when IOX has a free channel on its hands and no priority or preference exists. Several different dispatching strategies have been used in the past:

In the SOS buffering system, the dispatcher is given a list of units to be dispatched. Each entry in the list contains the location of the UCW (which doubles as a file block), whether input or output is being done, and how many buffers are to be kept loaded in the case of input.

Each time the dispatcher gets control it courses through the list once for each idle channel, checking if the requirement for each item is satisfied. If a non-satisfied requirement is found, a dispatcher preference is posted and the dispatcher advances to the next channel. This, then, is a simple priority scheme with priority ordering set by the programmer. The system also allows the programmer to replace this dispatcher by one he has coded employing a different strategy, but this facility seems never to have been used in practice. This would seem to be an indication that even a very simple-minded dispatching strategy is pretty good.

In the IOCS buffering system, the dispatching strategy is permanently welded into the input-output executor and the buffering system proper. Request chains are hung onto the UCB's, and the trap supervisor services the UCB's in strict rotation. If a given unit has just been serviced, it will be serviced again as long as a request chain exists (priorities are served immediately), and then the next UCB on that channel is examined. The buffering system assumes that if N buffers were required for the last READ then a request chain of N buffers must be loaded for the next READ. Request chains for output buffers are set up as buffers become quiet. This strategy is also simple and similar in effect to that used in SOS but, in practice, seems to require more complex bookkeeping and gives the programmer absolutely no control over dispatching.

The MICA buffering system operates on a first-come-first-served strategy, together with a determined attempt to keep a fixed number of buffers attached to each input file, regardless of file activity. The buffering system requests a load or empty by, in effect, attaching the clue to the end of the dispatcher request chain. The dispatcher will then work down from the head of the request chain in order to set preferences. This is another simple strategy which appears as if it might work quite well in practice.

Although little, if any, experimentation with different dispatching schemes has been done, it is still not obvious that the simplest strategy is the best -- it just works a lot better than no dispatching at all. We have observed speed improvements of three or four to one in programs using SOS buffering by making proper use of the dispatcher provided and by causing PUT to automatically set an output preference when a buffer is truncated and no preference exists. Presumably there exist situations in which additional speed improvements can be achieved by employing more complex dispatching strategies.

It is clear that a dispatcher will normally have to know something about the file block format and usage and, hence, that the dispatcher is properly a part of each buffering system rather than part of, say, IOX. In a case when more than one buffering system is in operation, we have the problem of joining the dispatchers in some way.

This can be done by providing IOX with the location of a list of dispatching routines which are to be entered in order. Since a dispatcher will normally pay no attention to existing preferences, the dispatcher last on the list will have priority.

Opening and Closing Files

To open a file is to set its file block for subsequent activity, initiate dispatching, and start activity on an input file; to close a file is to suppress dispatching for that file and wind up activity by restoring input buffers to the pool and draining output buffers. Details of opening and closing files may differ considerably between buffering systems. For instance, IOCS will not allow a given file to be used for both input and output, although certain types of files may be treated this way by using a different file block for input and output and closing one before opening the other.

The main process in file closing is called disconnecting the buffering system from the input-output unit; this process may be required for other purposes, such as in a buffered rewind routine. In most cases, a buffering system is used in conjunction with an operating system, and the supervisor will regain control at the end of the job. In the case where the job ran to normal completion, it may be safe to assume that all files got closed out; if the job folded prematurely, however, this is not a safe

assumption. In order to load the supervisor, it is necessary to make sure that all buffering systems are dead, since the main storage area used by the supervisor may be in use by one or more buffers being loaded. It is safe, but impolite, to kill all channel activity before loading the supervisor, but it is better if IOX has a way of disconnecting all files.

This can be done similarly to the way IOX handles dispatching. Namely, we provide IOX with a list of disconnect routines and file blocks to be disconnected. In this way, the supervisor is protected without losing any output that is ready to be written but otherwise would not have made it all the way out. This is particularly important in the case of debugging information.

Trap Protection

On most machines with traps, it is possible to disable traps under program control, quite apart from the automatic disable that occurs at the time a trap is executed. In the case of most buffering systems, particularly with pool buffering, it is necessary to use this feature to protect subroutine linkages or data that can be affected if a trap occurs. For instance, the buffer manipulation routines BFP and FTP may be entered either from SELECT or from other routines in the buffering system. The dispatcher, likewise, can be entered in trap from IOX or from BTP and BFP. Certain data in the file block or in

buffers may be picked up by one of the routines out of trap and then be changed by the occurrence of a trap before the data can be acted upon.

In order to protect the buffering system adequately, it is necessary to identify sequences of instructions that can be disrupted and make sure that traps are disabled while the sequence is being executed. In the case of subroutines, it is sufficient to identify the highest subroutines in the hierarchy that can be entered both in and out of trap and protect all calls for these by routines that are never executed in trap.

SUPERVISORY SYSTEMS

INTRODUCTION

The supervisory system, or monitor, is the part of an operating system that ties the rest of the system together. We have already seen in various places during our discussion of input-output systems that a coherent set of conventions for designing and using a system is of the utmost importance. This, of course, is also true in designing and using other subsystems, such as the processors that translate source languages into machine code or edit input and output data. Above all, however, this is true of supervisory systems, since they must work with all of the subsystems of an operating system. In fact, it may be said that an operating system is much more a set of conventions than it is a piece of code. The success of the design of an operating system can be measured almost entirely in terms of how good a set of communications conventions exist and how consistently they are adhered to in the design of the various subsystems.

The supervisory system will normally consist of a number of pieces of code, most of which will be in main storage only between jobs or between segments of a job. We can make a rough division of the system on a functional basis:

1. The supervisor. This is responsible for sequencing, both within and between jobs, control card interpretation, unit assignment, job accounting, and

set-up for use of processors or execution of object code.

2. The vigilance committee. This is a set of routines that monitor hardware status (especially abnormal conditions) and apply execution time and peripheral output limits.
3. Linkage routines. These provide linkage between the supervisor and processors, call for system routines that are not in main storage, load and transfer to object code, recover the supervisor, and possibly do dynamic storage allocation.
4. Utility routines. These may provide an on-line message writing routine, a library access routine, a pause routine, minimal unit assignment facilities for use during program and processor execution, emergency dump facilities, a clock reading routine, etc.
5. Communication region. In addition to the items mentioned under the heading of input-output systems for use by IOX, we need to store job data, date and time, output limits and current output values, processor options and variable parameters, and various other data for use by the supervisor, linkage routines, and the vigilance committee.
6. IOX. This will normally be assembled with the rest of the supervisor, as is the system symbol table.

THE SUPERVISOR

Definition of "Job"

As far as the supervisor is concerned, a job is a sequence of tasks that forms a unit for sequencing and accounting purposes. The job is a work unit which is embraced by some overall error criteria. Thus, we may delete an entire job because of the severity of the error encountered, but the processing specified within every job will be attempted. We also guarantee that the machine and the system are reset to a standard condition before processing for a job is begun.

The input for each job and each task will be preceded by a system control card which is read and interpreted by the supervisor. In particular, jobs are separated by job cards, which are system control cards that contain job identification and other information pertinent to the job irrespective of the tasks to be performed within the job. Such information might be various of the following:

1. Job number. The job identification for accounting purposes -- i.e., the account to which use of the machine will be charged.
2. Run identification. Identification of the deck, subjob, or machine run for the job. This is used only by the programmer.
3. Programmer identification. Whose job is this?

4. Expected total time required.
5. Expected amount of peripheral output.
6. Time limit for execution.
7. Output limit for execution.
8. Type code (such as production, assembly, code check, etc.).

This information will normally go into the communication region and part of it will be reproduced onto any accounting output the system produces.

Control Card Interpretation

System control cards should be uniquely identifiable as such by the supervisor. They have traditionally been punched in the same format as that used by the assembler(s) in the system -- not that they are assembled, but that symbolic instruction format has usually seemed the most reasonable one to use for this purpose. Since Hollerith cards are read by most system processors as well as by object codes, some type of protection against system control cards being swallowed (particularly cards for succeeding jobs) due to errors or improper deck make-up is required. A rather painful alternative is to follow each task by an end of file. A more foolproof alternative is available on machines which allow an intermixed binary and Hollerith input stream. As mechanized in SOS, this consists of punching the control cards in the normal way and inserting a control punch (7-8-9 in column 1 in this case) to cause the card to go onto tape

in the binary mode and in a unique binary format. This somewhat complicates the control card reading process, but meets the above two requirements, especially if processors back off after recognizing a system control card.

With the exception of the JOB card and cards which introduce data into the system, there is a control card for each task. A task requires a processor to work over the data between its control card and the following one. The information on the control card may be of three types:

1. Task name, such as LOAD, FORTRAN, DATA, COMPILE, etc.
2. Options for the processor, such as whether to punch a binary deck.
3. Variable parameters for the processor, such as name of input unit.

As an example, consider the SOS control card:

```
#    DATA   [EDIT  ]  [ GO   ]  [ SYSMOT  ]
            [NOEDIT]  [,GOIF ]  [,SYSxUn  ]
                                [ SYSxRn  ]
```

Each item within brackets is a possible field on the control card, and the contents of the bracket are the possible field variants, the underlined one being used if no one of the variants appears on the control card. EDIT means use the input editor to convert the following data, while NOEDIT means just transcribe the following data onto the

output tape. GO means continue the job in any case, whereas GOIP means continue the job only if the input editor encounters no bad data. The third field indicates the output tape to be used. Thus, the first and second fields are processing options and the third, a processor parameter. Note that with this kind of control card analysis, a large number of options can be indicated; but the programmer needs to indicate only those options that represent a departure from the normal (underlined) case. If the card can be designed in such a way that the same name is not used to indicate a variant for more than one option or parameter, then the control card scan routine can be written in such a way that the options mentioned need not be given in any particular order. It is possible to use the same name for variants of several fields if the field itself is given a name which appears on the card with the variant wanted, e.g.,

```
#     COMPILE  INPUT=SYSAR1,PUNCH=SYSBR2
```

Note that a variable field format for the control cards will normally be more convenient to use by the programmers and keypunchers than a fixed field format, since the options and parameters mentioned may run together without spacing over unused fields on the card.

During the course of the variable field scan of the control card, one can either set an option flag somewhere

or execute a set-up routine. At the end of the scan, the supervisor can then execute a routine specific to that control card -- this may wind up the action if a processor is not required or may call a processor into core for use.

Sequencing within Jobs

Normally, the action of the supervisor during a job is to read, interpret, and execute control cards in sequence. At certain times within the job, the supervisor may lose control to a task processor or to object code. It is the function of the linkage routines to get the supervisor back into control following task completion, at which time the supervisor must review any error indications, print any error messages, and then decide whether to continue the job or to skip to the next job. There are several types of error indications to be considered:

1. Normal return from object code -- the job may be continued.
2. Error return from object code, including the case in which error flags have been set by a member or members of the vigilance committee (such as "unreadable record on SYSAR2"). Error flags should now result in diagnostic messages being printed off-line, and the job should be terminated.
3. Exit from a processor with an error severity code -- continue to process the tasks within the job

except those for which the error severity limit has been reached (e.g., object code execution).

4. Catastrophic error exit from a processor -- this is so bad that the job should be terminated.

Note our insistence that there are two major modes of exit to the supervisor: continue the job and terminate the job. At least one operating system around today is incapable of doing anything else but attempting to continue the job, regardless of what happens!

Error severity codes reflect various types of errors that have occurred in the course of processing and a judgment by a processor as to whether they are probable, trivial, definite, etc., according to some error classification. Thus, a missing address on a symbolic instruction may be only a probable error; a missing optional field on an instruction may be trivial; a multiple definition is a definite error. A catastrophic error in processing should be reserved for a condition that causes the processor to go completely to pieces.

The GO-GOIF option on the DATA control card mentioned above is an example of specification by the programmer of what severity of error should be grounds for discarding the job -- this may vary according to the programmer's problem or work habits, but in any case should not be arbitrarily fixed by the system. If a non-zero error severity code is presented by a processor, the supervisor should use it to

control task execution when compared against the programmer's severity limit. In this way, for instance, a number of assemblies can be processed even though it is known that the resulting code will not in fact be executed.

At termination of the job, the supervisor may have additional duties to perform before starting the next job. Typically, these are the following:

1. Disassign any input-output units in use by the job, as opposed to in use by the system.
2. If the job has produced output for editing or debugging output to be converted, call the appropriate processor(s).
3. Clock the job out and prepare the accounting summary for the job.
4. Reset the system and machine to its base or reference state.
5. Search for the next JOB card, if it has not already been read.

Processor Supervision

We have pointed out two types of information from the control card that must be conveyed to a processor -- options and variable parameters. In addition, a processor may need (1) variable parameters that are generated or transmitted by the supervisor itself (for instance, when output from one processor becomes input to another or the size of available core) and (2) access to the system symbol table in

order to locate cells in the communication region or entry points to various routines belonging to the vigilance committee, linkage, or utility package. The best current practice is to provide, as one of the linkage routines, a processor load and entry routine, which contains a calling sequence for the processor itself, once loaded. The calling sequence will contain the variable parameters (about eight or ten will suffice for most purposes) for the processor, and the machine registers can contain the option indications on entry and the location and length of the system symbol table on entry and can return the severity code on exit. Given this information, the processor can initialize itself appropriately. If too many variable parameters are required, they can be put in a list elsewhere in main storage with a pointer to them in the calling sequence.

There is room for disagreement as to whether all options should be provided by the supervisor, whether all options should be conveyed by means of processor control cards that are read with the input data, or whether something in between is desirable. A good general rule to follow is that options which may change from run to run should be introduced by means of system, rather than processor, control cards.

Processors which are fairly complex, such as a compiler, may contain their own supervisory system, including

a communication region and various linkage and utility routines for use during processing. Above all, and this has rarely been done in practice, complex processors should include in their supervisory system routines to assist in debugging the processor itself. Such routines include facilities for printing out the processor's communication region, internal tables, contents of scratch tapes, and the processor code, either with special conversion facilities or by using debugging facilities belonging to the operating system. Another useful facility is a patch interpreter. This is a routine that gets control between each segment of the processor for the purpose of loading the next segment either from the system library or another source (in a case where a segment has just been reassembled) and applying patches to the segment. The patch interpreter might also include the ability to stop between segments to allow dumps to be made before proceeding or returning to the system. In some types of processors a test problem may be included in the debugging facilities.

In the case of many processors, a method of linkage with the system such as we have described also allows us to debug it as an object code by merely faking the system's method of entering the processor. If the system contains symbolic modification facilities, this is a much easier way of debugging than by trying to operate the processor in its normal environment, modifying the code by patching,

and using less than the full debugging capabilities of the system.

Insertion of debugging facilities within a task processor for use in debugging the processor itself represents a concession to the practical reality that processors are never completely debugged, and that they continue to evolve with the inclusion of new features and abilities.

Unit Assignment

For the purpose of discussion, we will make the same type of distinction between units as those made in SOS. Namely:

1. A system unit is a unit that is reserved for use by the system, such as peripheral input and output tapes, a system library tape, or disc or drum files used by the system.
2. A utility unit is a unit that is available for use for any purpose by the system or by object codes, such as scratch tapes.
3. A reserved unit is one holding a file or files belonging to a programmer. This will normally be assigned at the beginning of a job or object code execution and will be removed at the end of execution.

Somewhat different problems arise in handling each type of unit. In the case of system units, some units will be

semipermanently assigned, while others will not be. A peripheral input tape, for instance, will be mounted on a given unit, and the unit assignment routine must be informed where it is. At the end of the stack of jobs, the tape can be returned to the pool of available units. A peripheral output tape can be assigned from the available pool originally. When it is full, it is removed and a new one must be assigned. The old unit is now unavailable until the operator has somehow informed the assignment routine either that a utility tape has been mounted or a new input or reserved tape has been placed on the unit. Reserved units may be treated much like peripheral input units on assignment and peripheral output units on disassignment.

The status of each unit should be kept up-to-date either in the UCB for that unit or in unit status lists. The possible states, given the above classification, would be:

1. Detached: Not attached to the machine.
2. Attached: On the machine, but not available for use.
3. Available: Available for assignment (especially as a utility or peripheral output unit).
4. Utility: Assigned and in use as a utility unit.
5. Reserved: Assigned and in use as a reserved unit.

6. System: Assigned and in use as a system unit.

Note that an attached tape unit may either have no tape on it or may have a reserved or peripheral input tape on it awaiting assignment. Assignment of a unit means updating its status and putting the location of its UCB in some file block.

The unit assignment routines get information from the operating staff via system control cards; from the machine operator by reading cards, looking at a keyboard, or by examining keys on the console; or from calling routines. In SOS, reserved and utility assignment information is normally acquired by putting ASSIGN cards into the deck before writing the peripheral tape. The operator, however, gets a chance to review and correct or supplement the assignments at the time they are actually made. He may either punch and read cards or use the entry keys -- in practice, only the keys are used for this purpose.

The actual code for the unit assignment function may actually be distributed through the system. The function of saying to the operator, "I need a unit of such and such a type assigned for file block XXXXXX -- which one shall I use?" can be one of the utility routines. Routines that find an available unit and assign it as system or utility, restore a system or utility unit to the available pools, or put a unit in attached status can go almost anywhere

(including object code). The supervisor itself, or a processor that loads code and sets up for execution, may use a fairly general and massive unit assignment and checking routine. In any case, the machine operator should be treated as an active part of the assignment procedure rather than as someone who is to take instructions from the system. Units can go out of service in the middle of a batch of jobs, and the operator must be able to inform the system of this, among other things.

THE VIGILANCE COMMITTEE

Hardware Monitoring

Most large machines have a number of conditions that will cause traps or turn on indicators other than those with which IOX or an object code may be prepared to cope. Instances might be: transfer traps, floating point traps, storage protect traps, various indications of hardware malfunction, unexpected beginning of tape indication, etc. The fact that one of these conditions has occurred is not necessarily cause for alarm:

1. The action resulting in the condition was deliberate, and the condition should be ignored.
2. The fact that the condition has occurred is of no consequence.
3. The condition did not really occur -- the vigilance committee thinks it occurred due to a wild transfer in the object code.
4. The running code has tested for the conditions it expects if all goes well, leaving it up to the vigilance committee to haul up a flag if anything unexpected happens.
5. The condition can occur, but corrective action is possible.

This implies, then, that some flexibility of approach is required, at the option of the running code. It would be

as bad to make no provision for detection and disposal of these conditions as it would be to turn the vigilance committee into a lynch mob.

We do not wish to imply that the code representing the vigilance committee should necessarily be present in main storage at all times or that it should belong exclusively to the supervisory system, so long as it communicates with the supervisor by conventional methods. In many systems, this function is partially or totally carried out by code belonging to the processors and/or object programs; in other systems, such as SOS, one or more vigilance functions normally exercised by the supervisory system may be assumed by a processor or the object program during part or all of the course of the job. The technique of using system transfer points, or an equivalent technique, may be used for achieving the type of flexibility we have described.

System Transfer Points

SOS uses the convention that all system symbols are six characters long and begin with the characters "SYS"; these are the symbols in the system symbol table, representing locations of system parameters, data cells in the communication region, or entries to system routines which may be legitimately entered from a processor or object code. Certain of the system routines -- the members of the vigilance committee and the two linkage routines, SYSTEM and SYSERR, for normal and error return to the supervisor --

are entered via system transfer points of the form:

```
SYSxyz  TXH ENTRY,,EXIT
        TXL xyz,,**
```

In the normal case, which is restored each time the supervisor is loaded, ENTRY and EXIT are both zero, and a transfer to SYSxyz results in transfer of control to the routine "xyz." If EXIT is set to the location of another routine, "xyz" will gain control via the standard subroutine linkage but, instead of taking its normal exit, will transfer to EXIT when its action is completed. If the first cell in the transfer point is set to

```
TXL ENTRY,,EXIT
```

then a transfer to SYSxyz will result immediately in a transfer to ENTRY. The routine at ENTRY may, if it wishes, then transfer to SYSxyz+1, in which case "xyz" will execute and then exit as above.

Typically, members of the vigilance committee will exit to SYSERR unless their exit switch is set and will also hoist a flag causing the supervisor to print an appropriate error message when it gains control. Similarly, SYSERR will normally exit to SYSTEM. Thus, by proper presetting of the system transfer points, a program or processor may regain control for individual conditions either before or after an error flag has been hoisted, or may gain control at SYSERR for the remaining conditions for which presetting was not done, and, in addition, may regain

control prior to the termination of execution in order to execute clean-up or dump actions.

Execution Monitoring

The job of the execution monitoring routines, when present in a system, is to ensure that things don't go too far. Particularly in the case of checkout, a program can lose its head by looping or printing reams of garbage off-line. The machine operator may not be aware of either condition, unless he is familiar with the program being run or tested. Hence, a facility by which the programmer can post a watch for too much execution time or too much peripheral output can be quite valuable. In particular, it might often be convenient to post time and output limits for the job as a whole and, under program control, to post limits for a phase of the execution and regain control via the transfer points if they are exceeded. Timing is best done by a millisecond clock equipped with a trap for interval timing; a clock that requires deliberate action to read and check against the limits can be ignored by the program if the right type of loop occurs.

Another type of execution monitoring is the collection of statistics on the behavior of the program. A programmer might, for instance, wish to know where he is in his subroutine hierarchy at the time of transfer to SYSERR. This is fairly simple information to acquire, for if standardized subroutine linkages are used, it is a

simple matter to arrange to trace the subroutine tree and print out its structure. If subroutine entries are always defined by standard macros in assembly languages or are always compiled in a standardized fashion, it is also possible to collect other information such as how many times the subroutine was entered and the total (or average) time spent within the subroutine. Other possibilities might also suggest themselves. One would not want all of this happening in a production code, but it could be eliminated at the cost of setting a debugging mode switch which is tested at each entry, leaving the special code in place even in production. (Many a program suddenly reverts from production to checkout status!) Much would ordinarily be eliminated by the normal return to SYSTEM with abnormal returns to SYSERR enabling the debugging output.

LINKAGE ROUTINES

Processor Linkage

We have already discussed processor linkage from the point of view of processor supervision and communication of parameters and options between the processor and the supervisor. It is worthwhile pointing out, in addition, that certain processors may exist in main storage along with an object code for use by it. In such a case, the supervisor or the object code linkage and initialization routine may wish to enter the processor in the normal manner for initialization purposes, whereas quite another type of linkage may be required for actual use of the processor. Examples are buffering systems, debugging supervisors, and complex editing and conversion processors.

The mechanics of loading a processor vary considerably among different systems. Some older or more primitive systems rewind the system tape and then skip up a known number of files to reach the processor desired. More recent systems keep a table of file names in main storage to give position information. SOS even allows multiple appearances of a system component on its library tape and keeps track of the current position of the tape in order to use the closest occurrence of the desired file. With a medium such as disc storage, there is not any real problem about access time, but a table of file locations is still required, unless we are to allow each component to know the absolute address of any file it may wish to call!

Object Code Linkage

The job of getting object code into main storage and setting the proper initial conditions may range from a very simple to a very complex affair. As was noted above, the processor linkage routine may be required to load processors into the machine for use by the object code. Library subroutines may be found and linked with the code. It may be necessary to read control cards which control object unit assignment by setting up the file blocks that will be used by the code. A large part of the initialization work may be done either by the supervisor or by a separate processor, depending on the complexity of the task. If very much code is required to do the necessary work, it will usually be the case that code is written onto a scratch unit as it is processed for loading purposes and converted into its final absolute form. At this point, control will revert to the linkage routines in order to preset the remainder of main storage to some fixed bit pattern (normally zeroes), load and initialize any processors required, and finally load and transfer to the code. If more than one segment of code has been prepared, the object code linkage routine will get control back for the purpose of loading each segment as it is required.

Supervisor Linkage

The job of the supervisor linkage routine is to recover the supervisor after object code or a processor has

been in control of the machine. Several things must be done:

1. Disconnect the buffering system(s) in use.
2. Load the supervisor, if it is not in main storage.
3. Make any accounting entries that are required.
4. Preset the communication region to its normal within-job condition (e.g., reset the transfer points).
5. Transfer to a standard entry point in the supervisor if a special one was not specified when the supervisor last relinquished control.

CONCLUDING REMARKS

There are, of course, many other aspects of operating systems than those which have been discussed above. For instance, we have talked about the problem of communication between the processors and the rest of the system but not about the question of how many or what processors should exist in an operating system. Again, we have not brought up the subject of system maintenance, which is anything but a merely clerical task and for which few tools are available in most systems. Assembly systems, too, should be discussed in detail, since the facilities available in the assembly system will almost certainly have a fundamental influence on design, coding, checkout, and maintenance of the various parts of the operating system. An adequate description of the current state of the art in these areas, both with regard to currently available tools and with regard to those that could be provided, would be at least as lengthy as the material presented in this paper.

Appendix I

INDEX OF TERMS*

		Page No.
	Active	46
	Activity Cell	24
	Assign Routine	28
	Attention Recognition Routine	32
	Attention Request	14
	Attached	75
	Available	75
	Block	7
	Block Flag	48
	Block Size	7
	Buffering System	18
	Buffer Control Word	44
BFP	Buffer from Pool Routine	52
	Buffer Pointer	44
BTP	Buffer to Pool Routine	52
	Buffer Unit	36
	Catastrophic Error	70
	Channel	8
	Channel Program	11
	Chain	44
	Close	60

*Abbreviations for terms appear in the left column.

	Clue	53
	Command	11
	Communication Reg'on	15
	Controller	8
	Controller Program	11
	Data Operation	13
	Data Select	13
	Detached	75
	Dispatcher	25
	Dispatcher Preference Cell	24
	Disconnect	60
	Drain	47
	Drum-like	6
	Empty	37
	End	23
	End of File	7
	Error Return	69
	Error Severity Code	69
	Field Variant	67
	File	7
	File Block	16
	Flag	48
	Flush	47
	Held	46
	Input	37
IOCS	Input-Output Control System	16

Input-Output Executor 20
Input-Output System 3
Input-Output Unit Control Word 16
Instruction 11
Interrupt 29
Job 65
Job Card 65
Linkage Routi es 64
Load 37
Locate 40
Logical Position 47
Moving 46
Non-Data Operation 13
Non-Data Select 13
Normal End 13
Normal Return 69
Operating System 3
Open 60
Option 67
Order 11
Output 37
Patch Interpreter 73
Physical Position 47
Pool Control Word 44
Processor 3
Put 20

	PUT Routine	52
	Quiet	46
	READ	52
	Record	7
	Reserved Unit	15
	Select	8
	SELECT Routine	20
	Select Word	23
	SELECT (+)	23
	SELECT (-)	24
	Sense Data	26
SOS	SHARE Operating System	15
	Start	23
	Supervisor	63
	Supervisory System	3
	SYSERR	79
	SYSTEM	79
	System Control Card	65
	System Priority Cell	24
	System Transfer Point	79
	System Unit	15
	Take	20
	TAKE Routine	52
	Tape-like	6
	Task Name	67
	TEST Routine	34

	Trap	13
	Trap Supervisor	18
	Transmit	40
	TRUNC Routine	52
	Truncate	48
	Unit Assignment Routine	16
UCB	Unit Control Block	16
UIR	Unit Interpretative Routines	20
	Unit Status List	16
	Unusual End	13
	Use Count	53
	Utility Routines	64
	Utility Unit	16
	Variable Parameter	67
	Vigilance Committee	64
	Whyte Flag	49
	WRITE	52

REFERENCES

1. Leiner, Alan L., "System Specifications for the DYSEAC," Journal of the ACM, Vol. 1, No. 2, April, 1954, pp. 57-81.

2. Shell, Donald L., "The SHARE 709 System: A Cooperative Effort," Journal of the ACM, Vol. 6, No. 2, April, 1959, pp. 123-127.

3. Greenwald, Irwin D., and Maureen Kane, "The SHARE 709 System: Programming and Modification," Journal of the ACM, Vol. 6, No. 2, April, 1959, pp. 128-133.

4. Boehm, E. M., and T. B. Steel, Jr., "The SHARE 709 System: Machine Implementation of Symbolic Programming," Journal of the ACM, Vol. 6, No. 2, April, 1959, pp. 134-140.

5. DiGri, Vicent J., and Jane E. King, "The SHARE 709 System: Input-Output Translation," Journal of the ACM, Vol. 6, No. 2, April, 1959, pp. 141-144.

6. Mock, Owen, and Charles J. Swift, "The SHARE 709 System: Programmed Input-Output Buffering," Journal of the ACM, Vol. 6, No. 2, April, 1959, pp. 145-151.

7. Bratman, Harvey, and Ira V. Boldt, Jr., "The SHARE 709 System: Supervisory Control," Journal of the ACM, Vol. 6, No. 2, April, 1959, pp. 152-155.

8. IBM Corporation, SHARE SOS Reference Manual: SHARE Operating System for the IBM 709, Form Nos. 328-1219, 328-1262, 328-1377, 328-1395, 328-1406, X28-1213, IBM Applied Programming Publications, New York, 1960-1961.

9. IBM Corporation, Reference Manual: IBM 709/7090 Input/Output Control System, Form No. C28-6100-2, IBM Applied Programming Publications, New York, revised January, 1962.

10. Ferguson, David E., "Input-Output Buffering and FORTRAN," Journal of the ACM, Vol. 7, No. 1, January, 1960, pp. 1-9.