WW

BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

HOSPITAL COMPUTER PROJECT

MEMORANDUM NUMBER SIX-B

EXECUTIVE SOFTWARE STATUS REPORT

1 MAY 1966

CAMBRIDGE NEW YORK CHICAGO LOS ANGELES

HOSPITAL COMPUTER PROJECT

A JOINT RESEARCH EFFORT OF THE MASSACHUSETTS GENERAL HOSPITAL AND BOLT BERANEK AND NEWMAN INC

MEMORANDUM NUMBER SIX-B

EXECUTIVE SOFTWARE STATUS REPORT

Submitted by Bolt Beranek and Newman Inc Cambridge, Massachusetts 1 May, 1966

HOSPITAL COMPUTER PROJECT*

M E M O R A N D U M S I X B

Status Report on Executive Software

Submitted by

BOLT BERANEK AND NEWMAN INC

Medical Information Technology Department

Jordan J. Baruch, Principal Investigator, 1962-1966
Paul A. Castleman, Principal Investigator, 1966-
Richard H. Bolt, Supervisor, 1966-

Nancy Adams
John Barnaby
Sheldon Boilen
Richard A. Bolt
Michael Cappelletti
Jonathan Cole
Bernard Cosell
Edward Gilbert
Edith Grossman
Nancy Haggerty
Alice Hartley
Gray Hodges
Toby Jensen
Charles Jones
Larry Kershaw
Philip LaFollette
Nancy Lambert
Maureen Letu
Sanford Libman
Barbara Lieb
William Lucci
Richard Mahoney
Ernest McKinley
Charles Morgan
Andrew Munster
Nicholas Pappas
Robert Payson
Lawrence Polimeno
Sylvia Sarafian
William Slack
Sandra Sommers
Lee Stein
Peter Storkerson
Sally Teitelbaum
Frederick Webb
Walter Weiner
Steven Weiss

* This joint research project is supported by Grant FR 00263 and Contract PH 43-62-850 from the National Institutes of Health and the American Hospital Association. Efforts contributed by the Laboratory of Computer Science, Massachusetts General Hospital, are directed by Principal Investigator G. Octo Barnett, M.D. The report, Memorandum Six B, gives a technical description of Executive software developed by BBN for the computer system being used on this project.

TABLE OF CONTENTS

TABLE OF CONTENTS cont.

FIGURES

TABLES

HOSPITAL COMPUTER PROJECT
STATUS REPORT MEMORANDUM SIX B
EXECUTIVE SOFTWARE

I. INTRODUCTION

In its several years of existence, the BBN-MGH Hospital Computer Project has gone through two major stages and is currently entering a third. In its first stage, affectionately called "Little Hospital," a prototypal system was built in order to test out the basic gross concepts of the project.

In this "Little Hospital" phase, we tested the basic techniques of: Time-Sharing, inter-communication among terminals under program control, file-dependent error checking routines and a very tentative exploration of the file structures necessary to support a hospital activity.

In the second phase of the project, called "Big Hospital," a larger computer, having independent memories and a large data storage system was used to explore further developments in Time-Sharing, file structures, routines suitable for in-hospital use, the interactions between hospital and engineering personnel, on-line computer usage within the hospital itself and some of the timing and reliability questions associated with hospital computer utilization.

The third phase of the Hospital Computer Project has just begun. In this phase further exploration of the areas of program applicability will be undertaken. A new Time-Sharing system

embodying many of the developments dictated by our experience in phase two is under construction and new files with more extensive protection and other reliability features are being implemented.

Because the phase-two system has become part of the Time-Sharing art and because widespread interest has been expressed on the part of both hospital and non-hospital personnel engaged in dedicated system design, we have decided to make Memorandum Six as complete a description of the overall hospital system as is practical. One is torn in writing such a description between making it a true scientific thesis and making it an instruction manual.

Were it a true thesis, we would delineate our hypotheses, document our agonies, validate our conclusions, make estimates of reliability and carefully describe those areas which we feel merit further exploration.

Were we simply to write an instruction manual, we would describe conditions as they are without regard to history but with great regard to service, functioning and repair.

Actually, however, scientific papers are best written at the conclusion of scientific endeavors and instruction manuals are best written at the conclusion of construction programs.

The present project is, of course, neither. Memorandum Six is also neither a thesis nor instruction manual. Phase two of our program has been a combination of exploration and engineering development. We have taken approaches that are not to be dignified with the term "working hypotheses"; we have made

intuitive guesses that we shall not dignify with the name "insights" and we have compromised, improvised and otherwise dealt with reality in the age-old fashion of engineers.

As a result, Memorandum 6 is at best an interim engineering report. It will deal in part with progress made and shortcomings overcome as well as with some dissatisfactions and shortcomings newly uncovered. By and large, however, Memorandum 6 is a guide to others who would experiment with real-time Time-Shared computer systems dedicated either to the health profession or to some other similar service.

Broadly speaking, Memorandum 6 has been divided into five separate logical areas. For convenience in management, each of these areas has been published as a separate volume. The first volume (Memorandum 6-A) contains a detailed description of the hardware on which the present system has been implemented and has been previously published. The description is detailed enough so that competent hardware people, skilled in the art, can form rational comparisons between pieces of hardware with which they may be confronted and the pieces used for the present implementation. Instruction sets, timing and hardware philosophy are all discussed in great detail. Memorandum 6-A forms a necessary background for anyone wishing to duplicate the present system or to design a substitute containing some significant percentage of its features.

This second volume (Memorandum 6-B) is a detailed description of the programs that are fairly machine dependent and that convert the hardware of Memorandum 6-A into a Time-Shared Computer System of reasonable generality but of specific applicability

to the needs of the medical profession. This volume reviews the concepts of Time-Sharing, describes the computer facility and the Executive routines in the present system. It serves as a detailed programmer's manual for those who would use these Executive routines as a background framework for programming.

These common routines lie intermediate between the machine-dependent Executive routines and the hospital-dependent user programs. It is these routines that make the information system independent of the machinery on which it was implemented and independent of the user for whom the system was designed. It seems clear to us now that future systems will concentrate more and more of their effort on the design of more powerful common routines in order to achieve greatest flexibility.

Memorandum 6-C together with Memorandum 9, put out by the Massachusetts General Hospital, concerns itself with the characteristics of the programs and the programming problems that are evident in the hospital. In this volume, details of file construction, details of overt program behavior, and any special characteristics of individual user programs are discussed. In addition, it deals with a set of programs that form the program library system. Memorandum 6-C should be read in conjunction with Memorandum 9 by anyone interested in adapting some of the particular user programs and user programming techniques to his own system.

The fourth volume in the series, Memorandum 6-D, is devoted to a description of the Research Cycle of programs. This cycle forms the prototype for a new system in which the user has active linguistic control over the characteristics of the system itself. It is now clear to us that the ongoing development

of the system should be easily controlled by medical personnel without the constant intervention of professional programmers. The research cycle acts as a useful first experimental model for the kind of system that will eventually become commonplace. Through its use, we have become more aware of both the benefits and limitations of a question-answer language and have thus turned our attention more and more to the development of a procedural language.

Memorandum 6E, the last in the series, contains a detailed description of the programming system used for writing the present operating programs. This programming system contains input and editing facilities, a powerful recursive macro assembler, on-line programming-file manipulation tools, suitable on-line debugging aids and an overall administrative structure. It should be noted that the reason for the inclusion of such a programming system in this report is twofold. First of all, its inclusion will serve as an aid to its duplication for those who are convinced, as we are, that this kind of programming is a useful adjunct to the development of a real-time on-line system. Secondly, the programming system description permits one to learn the language in which all of the source language programming has been done and hence to read these source language programs with some degree of facility.

For those who will have followed Memorandum 6 through this last volume, there is an appendix containing some 2,000 pages of source language programs. This appendix contains the annotated listing of the entire system. Because of the enormous cost of publishing such an appendix, this appendix is available only on microfilm (or in xeroxgraphic production) from University Microfilms, Ann Arbor, Michigan.

Perhaps one of the clearest conclusions evident in this set of memoranda is the fact that a complex system, such as a hospital computer system need not be viewed as a single very large program. It can be treated as many individual programs running under overall surveillance from a major Executive program. This view permits one to deal with these individual programs independently, debugging them and checking them out at a reasonable rate and with a reasonable application of resources. The characteristic all-or-none philosophy and static structuring of the very large system is thus avoided. A set of small independent programs can continue to develop both in application and form while the system is actually operating. This adaptability is the most important single characteristic of the present system.

II. AN INTRODUCTION TO THE SYSTEM

A. Goals

Since its inception, the Hospital Computer System has developed four goals that have proven useful as a frame of reference in discussing the functional parts of the system. In Memorandum 5(1) Pages 21-38, we discussed the concept of an interpretive communication system. Such a system is one which is conditioned by its users to process and route current messages (packages of information) in ways dependent on the information content of those messages and of previous messages. Realizing such a system has been our first goal. While each of the following goals could be subsumed under such a general definition, we will treat them separately.

A second goal is the provision of an orderly storage and retrieval system for hospital data. By "orderly" we mean not only well organized in the computer sense but capable of assisting the user of the system to bring order to his own record keeping, his own thinking processes and his own retrieval requests. Such a storage and retrieval system must operate both on records that are generated by normal hospital discourse at whatever stage of development the hospital computer system has reached at that time and upon records that are inserted de novo into the system by a researcher. This dual system would give us the

(1) All references are made to the list at the end of the volume.

capability to build a data base through normal hospital commerce with no pressure on the hospital to extend the scope of that data base before it is ready. The direct insertion of data by the researcher lets him build his own private data base independent of the information commerce activities of the hospital.

To use such an orderly storage and retrieval system, we must provide certain language capabilities. These capabilities are to permit ready specification of retrieval algorithms with which hospital people will be concerned. In the section on the research files we will discuss a questionnaire form of such a retrieval language. We will discuss the limitations of such a question-answer retrieval language and how some of those limitations may be met by the proper design of a procedural retrieval language.

Our third goal is to provide a system that can do useful abstracting of information from the patient record on demand. It has become clear that any large clinical research project forced to use both the entire patient's medical record and the entire patient population as its fundamental data base would soon find itself so severely limited that much of the advantage of a real-time system would be lost. Further, it is clear that such an extended data base, while needed for the ensemble of all research projects, is seldom useful for any individual project.

We would like to provide a facility that permits the researcher, administrator or practicing physician to describe an abstracted record by describing those fields

of information such as "patient name", "admission diagnosis", etc., which interest him. We would further like to enable him to describe the population which interests him in terms of mathematical and logical operations on the values of fields in the record. Thus he may be interested in all males, or all females over 35 years of age. The facility would then go through the total set of medical records and abstract the fields of interest only across the population of interest. It would thus generate a private field for the researcher. Such an operation would produce a collapsed file for the researcher's use as part of his real-time activity.

An extension of this type of extraction is its application to the records of the in-house patients. Here, instead of generating a collapsed file for subsequent reference, it is most likely that the user would be interested in printing out the abstracted information immediately. Suitable programs for performing such abstraction both from tape and drum have not yet been written for the system although their need has helped to shape the system design and the Research program experiments.

As a fourth goal, it has seemed most desirable to provide the researcher with a tool for doing remote mathematical manipulation in a most general form. After reviewing the field, we have selected the Rand Corporation's "JOSS" Program as typifying the kind of mathematical manipulation facility that would be useful to a hospital population. Our own TELCOMP Program is an adaptation of JOSS for use with Teletype terminals. In order to provide for the

ready flow of information between the retrieval operation and the mathematical manipulation operation, certain modifications to JOSS have been necessary and many others are envisioned. It is probable that the abstracting program described earlier will, in fact, be made a part of the TELCOMP interpreter.

These four goals are set forth here more as a clarification of the project's emphasis than as a planned extension of our previous activities. It is to be hoped that as we incorporate such other functions as the accounting and financial record keeping of the hospital into the overall system during the coming years, the same interpretive communication, storage and retrieval, abstracting and mathematical manipulation techniques will prove sufficient to meet those needs as well as the needs of the other members of the hospital community. It is our belief that by taking this general approach, one permits the hospital to dictate both the direction and rate of its acceptance and utilization of the hospital computer system.

B. System Approach

Figure II-B-1 is a gross block diagram of the overall system now in use on the Hospital Computer Project. We have combined the hardware with that part of the Executive Program associated with Time-Sharing under the single heading of a "Time-Shared Computer Facility." In addition to this facility, the system contains a set of routines that are used in common by all the programs, a start-up program which enables individual programs to be

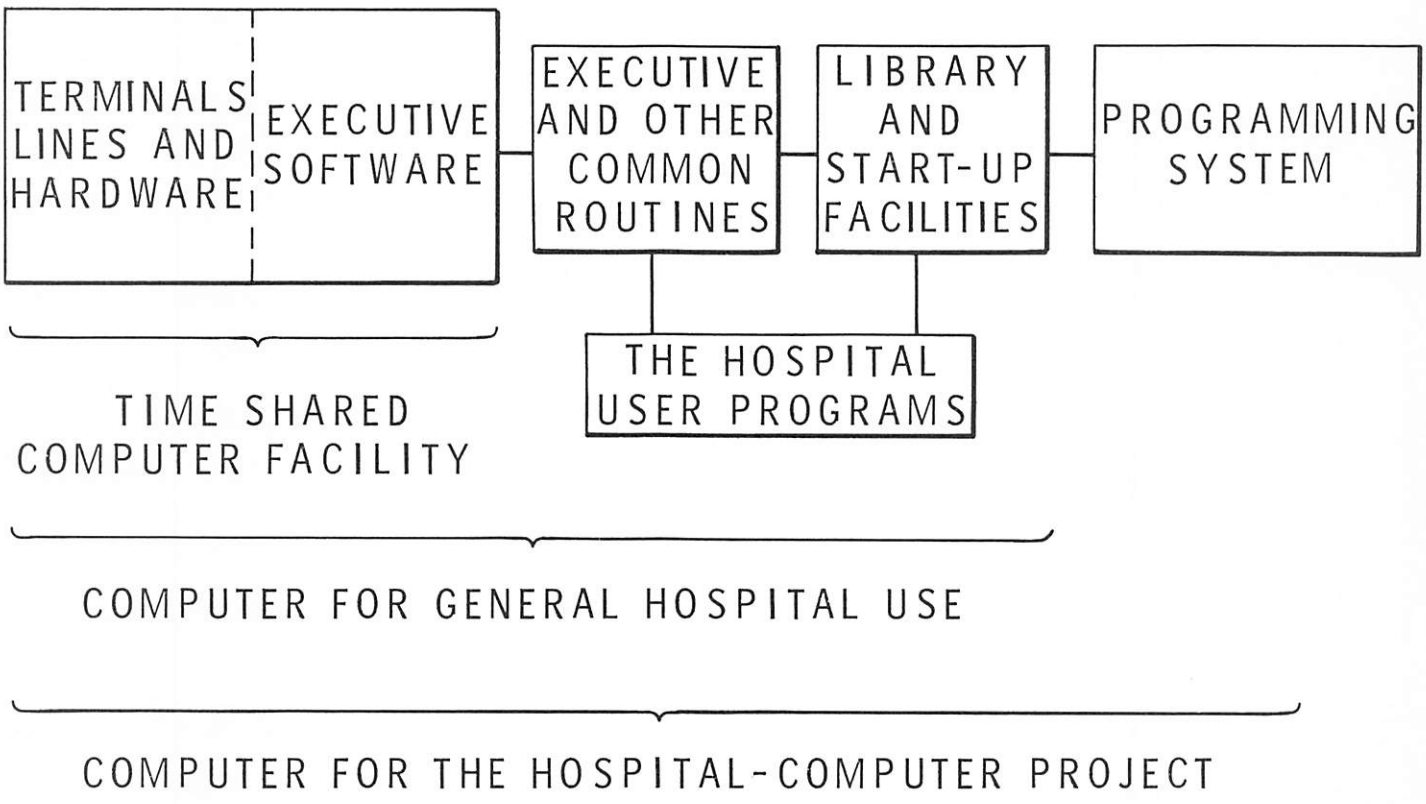


FIG. II-B-1 OVERALL GROSS SYSTEM LAYOUT

called at will, and the library of Hospital User Programs. Combined with the Time-Shared Computer Facility, these three additional blocks make up the system as needed for general hospital use. Since the Hospital Computer Project is a research and developmental project, it requires a block of programs to be used for composing, assembling and debugging all of the programs in the system. These three programs are lumped together under the heading of "Programming System" and, with the preceding parts, makes up the computer system as needed for the Hospital Computer Project.

C. An Introduction to Time-Sharing

While Time-Sharing has recently been much in the literature (2) (3)(4), it is probably wise, in a text like this, to review some of its basic principles and their logical basis.

The simplest, but most expensive, form of a Time-Sharing system can best be thought of as a high-speed central processor having an infinite core memory, some interrupt features and some memory protection features. Let us examine the functioning of such a hypothetical system providing service to a group of independent users. Such a system is illustrated in Figure II-C-1 where three users are shown sharing a common central processor. The memory associated with the processor has been divided into four parts, one being an Executive area and the other three areas being allocated to the three separate programs being used by the three users. To simplify the operation, each user is assumed to have associated with his terminal a

message buffer which is a discrete storage box capable of holding an individual message. Both Akron Children's Hospital and the Carnegie Tech. Time-Shared system use such buffers. To avoid oversimplification, however, we will assume that the message buffers are finite in size and actively signal the central processor by initiating an interrupt whenever a message buffer is either "almost full" or contains a finished message (as indicated by the presence of some terminating symbol from the user terminal).

Let us now follow the operation of such a simplified Time-Sharing system. We will assume that the Programs 1, 2 and 3 are in core and that Users 1, 2 and 3 are operating. User 2 now types an end-of-message delimiter (EOM) which initiates an interrupt at the central processor as soon as the processor has finished with its current instruction cycle. If we assume that Program 3 was running when the interrupt occurred, we can trace through a reasonable course of events. The occurrence of an interrupt causes the central processor to store the contents of all the live registers (such as the accumulator and program counter) in specific locations in the Executive memory area. These locations are the so-called sequence-break or program-interrupt storage locations. Transfer of control then takes place (a resetting of the program counter) to an area of Executive memory where a routine resides for determining which of the users initiated the interrupt and what to do about servicing it. In this case, since the interrupt was caused by User 2, the Executive Program restores the live registers for Program 2. In particular it resets the program counter to the value that it had when Program 2 was last

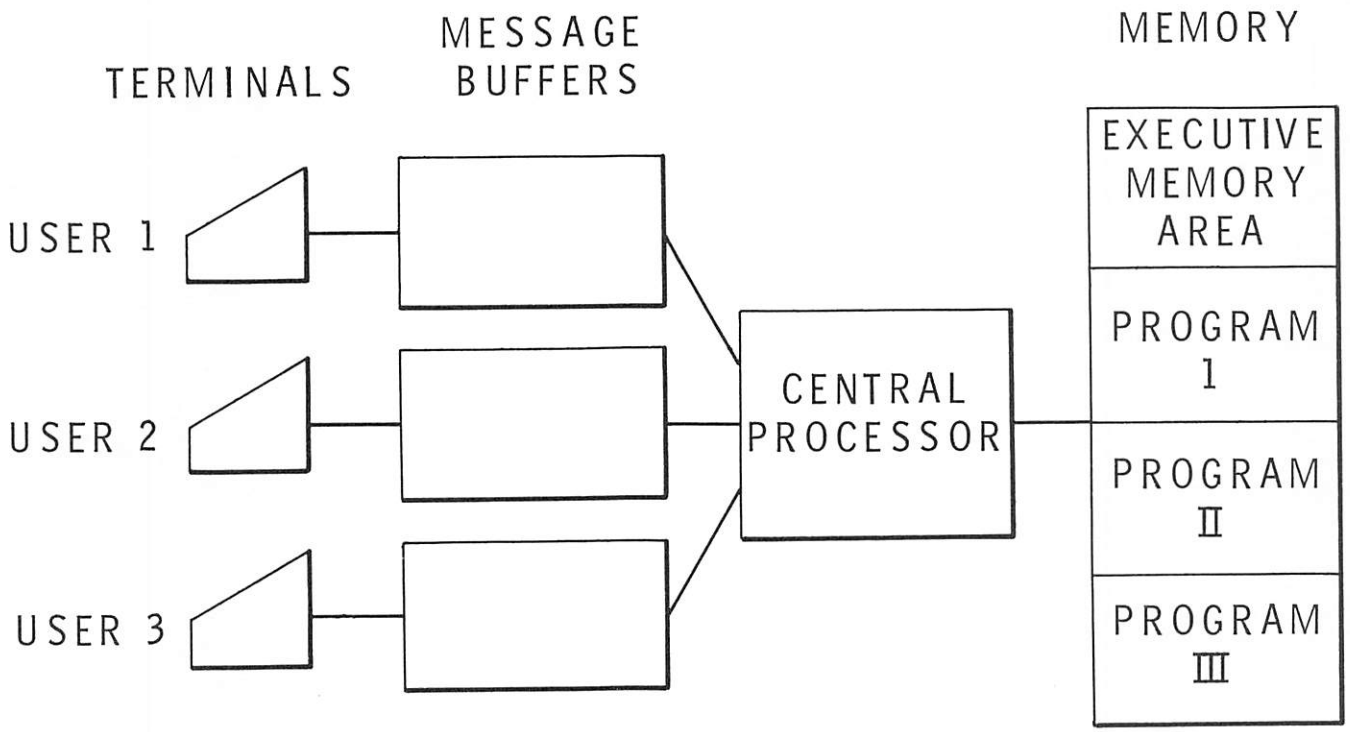


FIG. II-C-1 PROTOTYPICAL TIME-SHARING SYSTEM

interrupted, and transfers control. Program 2 now runs until it is interrupted by the real-time needs of one of the other users or until it attempts to execute an instruction of a kind that it is not allowed to do for itself.

Let us consider the second of these cases, the occurrence of a command of a nature that could ordinarily put a computer system into a suspended state. An obvious example is the command "HALT". Such a command characteristically suspends operation of the central processor. Clearly, Program 2 must not be permitted to execute such a command lest Users 1 and 3 be thereby deprived of the use of the processor.

Another form of instruction that must be prevented is any instruction from a program which references an area of memory not allocated to that program. Thus, if Program 1 were to try to write in the areas occupied by Programs 2, 3 or the Executive Program, that instruction would have to be either prevented or trapped. Were it not, logical errors in one program could destroy the entire system rather than just the program in which the logical error occurs. It is this localization protection that permits a Time-Shared system to be treated as a set of dissociated programs rather than one large program. We could, of course, prevent Programs 1, 2 and 3 from executing any such dangerous commands by having all of their instructions executed interpretively by the Executive Program. Such a procedure, while satisfactory for some usage is obviously inefficient for a more general system.

To avoid this inefficiency, let us build a piece of hardware that automatically stores those input-output transfer (IOT) commands, such as "HALT" or "TYPE OUT A CHARACTER AND WAIT FOR A COMPLETION PULSE". These commands will be stored in a register we call the "trap buffer" and will each initiate a sequence break or program interrupt at the central processor. The occurrence of such a sequence break again stores the contents of the live registers in the Executive memory area (albeit in different locations) and transfers control to that portion of the Executive memory which interprets the action of the central processor to be taken for trap-buffer interrupts. In other words, the Executive Program now can run as an interpreter for those commands which, for system reasons, the user programs must not be allowed to perform directly.

It should be noted that if Programs 1, 2 and 3 all have long computational procedures uninterrupted by IOT's, the last program having IOT activity will run to completion before the preceding one is restarted and that, in turn, will run to completion before the earliest one is restarted. Thus, if each program in our example has 5 minutes of running time, then Program 2 will run to completion before Program 3 is restarted by the occurrence of the HALT IOT in Program 2; Program 3 will then be restarted and run to completion, whereupon Program 1 will be restarted by the occurrence of the HALT IOT in Program 3. Program 2 will thus be completed in 5 minutes, Program 3 in 10 minutes, and Program 1 at the end of 15 minutes.

While this solution is extremely efficient and is similar to the handling of different programs in a "job shop" computer center, it can lead to trouble. There is, for example, no way of determining the time left to run on a computer program. As a result, a one-second program might have to await the completion of a one-hour program. To prevent this state of affairs we may let the programs run in turn, each for a short period of time. Usually those that have the shortest time to run will finish first. We make the assumption that the value of the system will be greatest if it "keeps everybody reasonably happy", rather than allowing one big user program to delay all the others.

To make it possible to cycle among the users, we must add to our simple Time-Sharing system an internal clock. We can then set this clock so that after some quantum of time, say one second, it will initiate another sequence break. The section of the Executive Program to which control is transferred after this break will start the next program in line waiting to run. We call such programs "runnable" to distinguish them from those that are "hung up" waiting for external activity. By following this procedure we allow those programs which have a short time to run to reach completion rapidly. Those programs which are simply computing a message to be typed out thus do not produce inordinate waits at the output terminals.

Clearly, the initiation of a sequence break after every quantum of time and its analysis by the Executive Program produces a certain amount of overhead inefficiency in the

system. The selection of the size of the quantum is thus a compromise between the immediacy of response required at the terminal and the amount of overhead cost one is willing to pay for such immediacy of response. Some rather sophisticated queueing algorithms have been worked out by Corbato and others (5) to take advantage of a human operator's expectations and tolerance. For example, Corbato makes the assumption that the demand for immediacy (i.e., the expectation of a prompt response) drops off exponentially with the time elapsed from the previous input-output activity. Thus if User 1 types in a message which should generate a negligible amount of computation before an answer is forthcoming he expects an answer immediately. On the other hand, if he expects a fairly lengthy computation, his tolerance as to the length of that computation goes up greatly. An algorithm which exhibits such behavior will be discussed in Chapter III.

We have now seen the development of three areas of the Executive Program: the Terminal Service Routine, the trap buffer manipulation routine also called the Dispatcher, and the clock routines called the Swapper. Our complement of equipment has further increased in size to include the trap buffer itself and the internal clock.

Clearly, if the separate programs deal with a common type of task as in a hospital, part of the Executive memory area can be dedicated to a set of routines used frequently by each of the running programs but not frequently enough to warrant hardware implementation. Routines to do floating-point arithmetic, routines to pack text into memory space

efficiently and routines to keep track of time in the real world through use of the internal clock fall into this class. These routines, since they are in the Executive Memory area, can be entered interpretively via the trap buffer. Further, since they may run for many instruction cycle times, they must be interruptable just as the User Programs were interruptable. Since they are shared, of course, the instructions within the routines cannot be modified during computation lest confusion be caused on an interrupt.

One way of solving this problem is to assign a small group of registers in both the Executive Memory area and the User Program area that can be modified and use these registers for communication between the User Programs and the common routines stored in the Executive Memory.

When an interrupt to a common routine in Executive Memory occurs, the contents of these few registers in Executive Memory are transferred to the corresponding registers in User Memory. As a result, each program contains a record of its own modified registers. The program that handles interrupts now takes the responsibility for preserving the temporary storage of the interrupted program. The common routines that exist within the Executive Memory area are programmed in the form of "READ ONLY" routines and could thus be contained in a read-only memory were such a part of the computer.

Just as a set of dedicated registers can be allocated in each of the program areas for the Common Routines, a set can also be dedicated for other communication needs of the

Executive routine. Since these registers are used by the Executive Program, they must only be changed by the User Program interpretively. They must thus be located in a region of memory that is inaccessible to User Program commands. Clearly, in an unlimited core memory, whether the dedicated registers are located in the actual program areas or in the Executive area is relatively unimportant. We point out that they can be located in the program area because of the requirements set by a system that has less than an unlimited core memory.

Let us now consider what would happen if the system of Figure II-C-1 had a memory only half the size shown, that is a memory capable of storing only the Executive Program and one User Program. Figure II-C-2 shows a diagram of such a limited-memory Time-Sharing system. We have added to the system a drum-storage memory which, in this example, contains three tracks, each containing an image of one of the programs. We shall further assume that the central processor has the capacity to read the contents of a register in core memory and write it out on one of these drum tracks and read the contents of another drum track and write it into that location in core memory all in one memory cycle.

Let us assume that the user program sections of memory (and the drum tracks) each contain 4,000 words of memory. If neither a memory cycle nor the passage time of one drum location exceeds 8 microseconds, we can read the current state of Program 1 from core memory onto the drum and the latest state of Program 3 from the drum into core memory

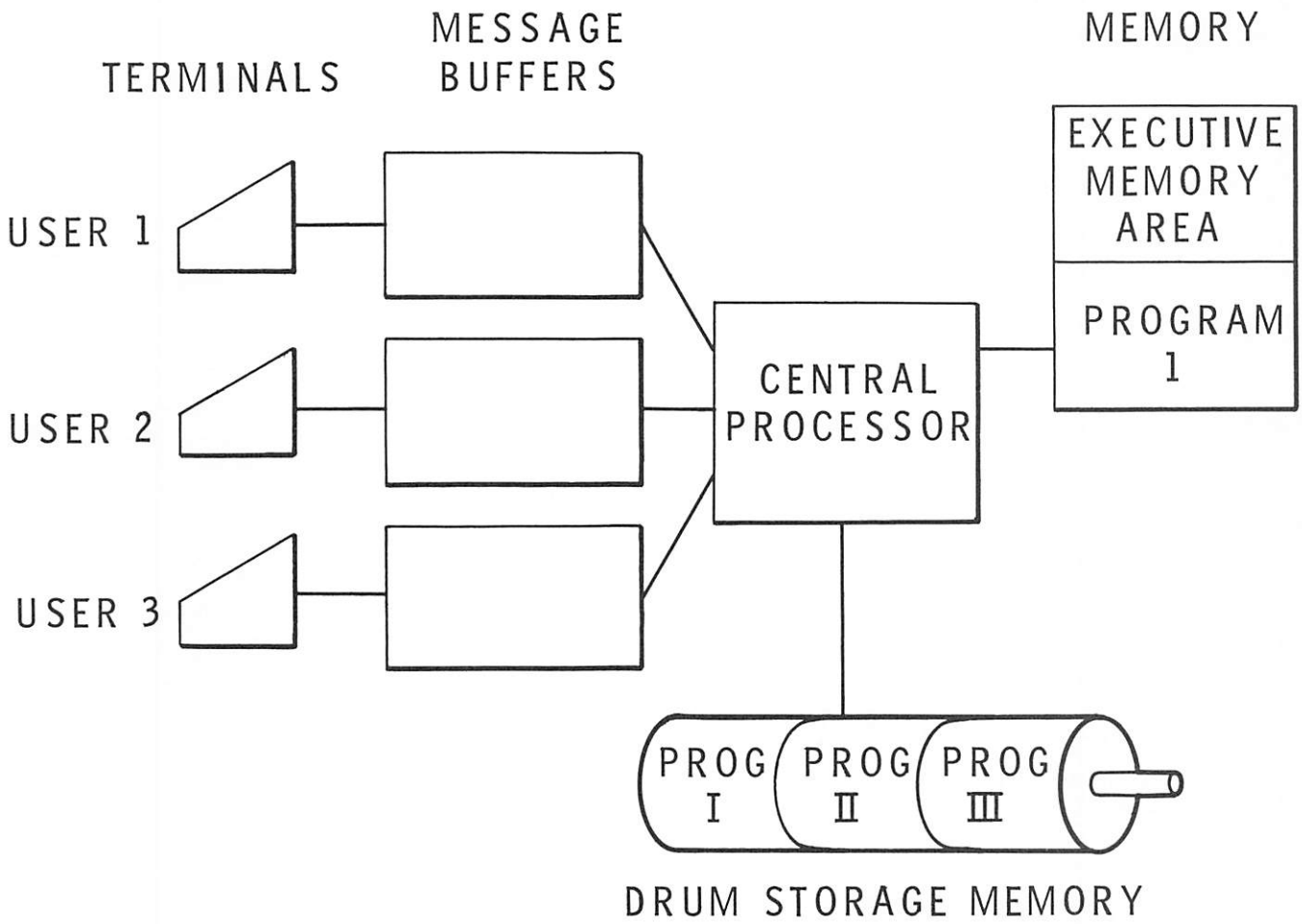


FIG. II-C-2 LIMITED-MEMORY TIME-SHARING SYSTEM

in a total elapsed time of 32 milliseconds. Clearly, since this kind of transfer must be done synchronously with the rotation of a mechanical drum, it cannot be interrupted. Hence the entire system ceases to function (as far as the outside world is concerned) for 32 milliseconds each time such a transfer takes place. This requirement for fixed quiescent periods conflicts with the requirement for not losing input messages. The added requirement that the system have the ability to handle peak traffic loads in turn demands a large-size message buffer.

Because interchanging two programs requires a total loss of 32 milliseconds, the sophistication of the Swapper algorithm becomes even more important to the efficient functioning of the system. A detailed analysis of the timing and generated weights of such a system is given on Pages 85-91 of Memorandum 5.

In order for a program to function properly if it is not continually connected to the outside world, the Executive routine must be able to communicate the state of the world to the running program. One technique for achieving this goal is to provide a set of registers in the user core whose contents can be modified by the Executive routine but not by the user. Another set, for less critical intercommunication between user and Exec can be set by convention rather than by hardware. In the present system, indeed, we reserve 40_8 registers at the bottom of core for the Executive's use only and an additional 40_8 just above that for use by both.

The system of Figure II-C-2 is similar to the Phase I system that existed on this project until the summer of 1964. The drum storage memory actually had 32 tracks, rather than 3. Five of these were used for programs and the remaining tracks were used for data.

The introduction into our developing system of the need for bulk data storage in other than high-speed memory introduces further constraints and requirements on the Executive Program. If, for example, we use a large drum or tapes or other such sequential storage system, the sychronization requirements leave us with the problem of either transferring data from bulk storage very ineffiently (because we are willing to interrupt data transfers) or losing immediacy of response (because we permit long data transfers to take place even while someone is waiting for an answer).

To avoid this Hobson's choice, we must make further design modifications to our hardware configuration. We could, for example, have the central processor of Figure II-C-2 merely initiate the transfer of data between the Program 1 area of memory and the drum but not take part in that actual transfer. To do so would require that the User Program area of memory have its own memory buffer and memory address buffer and that the drum have associated with it the necessary logic for keeping track of the start and finish of the transfer operation.

The central processor could than act as a switch to connect the User-Program memory to either the processor or to the drum as the exigencies of the situation required. Even further efficiency could be secured by having two such User-Program memories so that while one is transferring data to and from the drum the other one could be connected to the central processor. Figure II-C-3 shows such a configuration using a larger swapping drum. In this diagram the Executive Memory and User-Program Memory 1 are shown as connected to the central processor which is a condition analogous to that of Figures II-C-1 and II-C-2 while User-Program Memory 2 is shown connected to the drum logic and is in the process of transferring a previously run program onto the drum and accepting a program next to be run from the drum. While this 32 millisecond swap is taking place, the outside world is still being serviced both by the Executive Memory and by the User Program stored in User-Program Memory 1 both of which are connected to the central processor.

This concept of a multiple switch is analogous to the concept of renaming the memory. In Figure II-C-3 the User-Program Memory 1 is "named" the Central Processor Memory while User-Program Memory 2 is "named" the Drum Logic Memory. We can rename these either by making the connections shown as dotted circles or by setting a bit in the address part of the memory. Either accomplishes the same function. The concept expressed in Figure II-C-3 can be extended to other processor-like devices such as a high-speed data channel, a communication link, or other central processors. In each case the only requirement is that

each processor be capable of reading a set of instructions contained in core memory, following those instructions in accordance with some internal code and making its transfers of data to or from core memory directly. It is assumed in addition that each such processor can have its activity initiated by the central processor and will, in turn, signal the central processor upon completion of its activity by the generation of a sequence break.

Figure II-C-4 illustrates such a configuration having both a high-speed drum for program swapping, a lower-speed drum for bulk random access storage and a set of tape units as well as a central processor. Provision for more than three memories is also shown as is the fact that the Executive Memory need not be the same size as the User-Program memories. This configuration is the actual configuration of the present system and is discussed in detail in Chapter III.

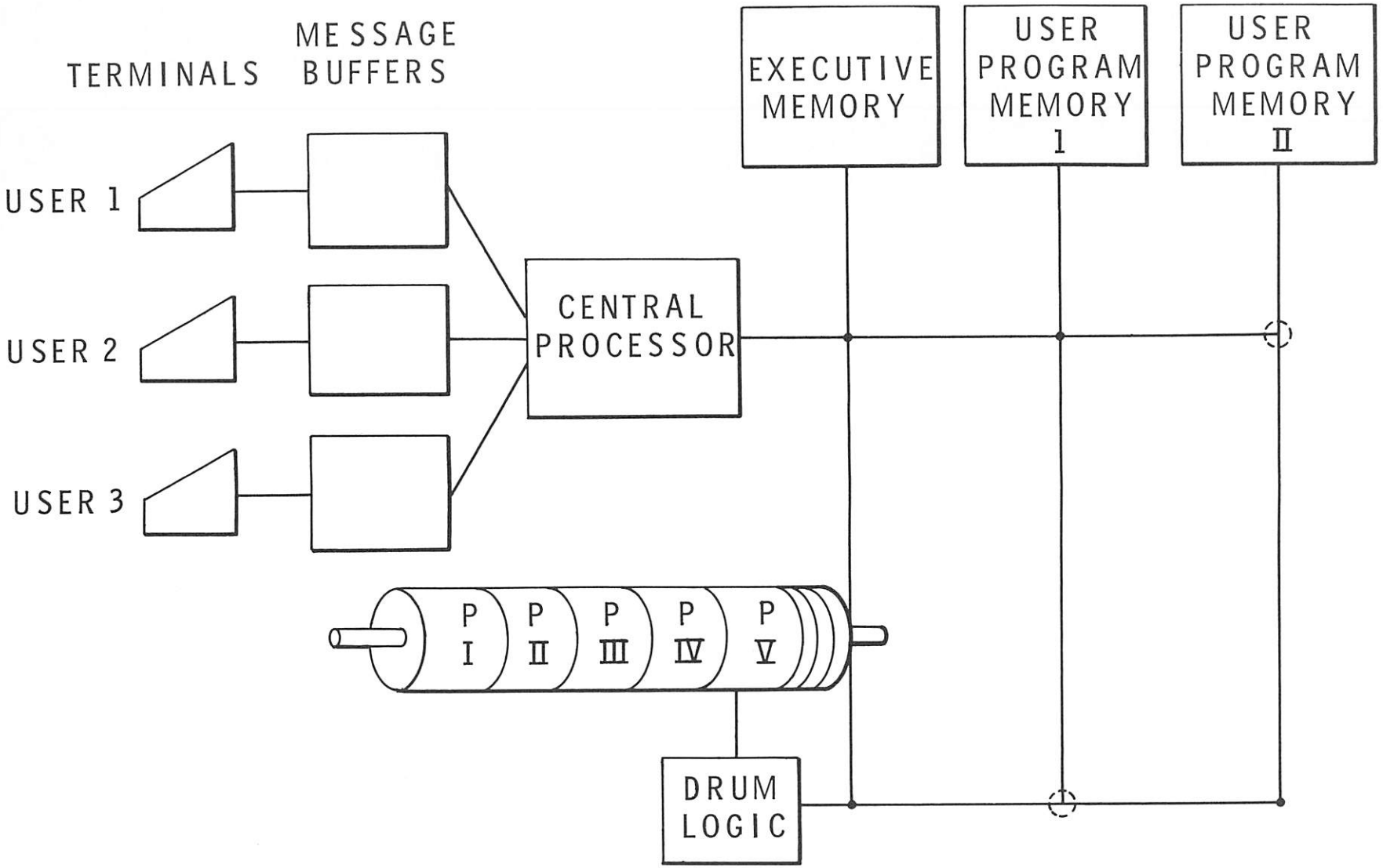


FIG. II-C-3 SYSTEM WITH SEPARATED MEMORIES AND PROCESSORS

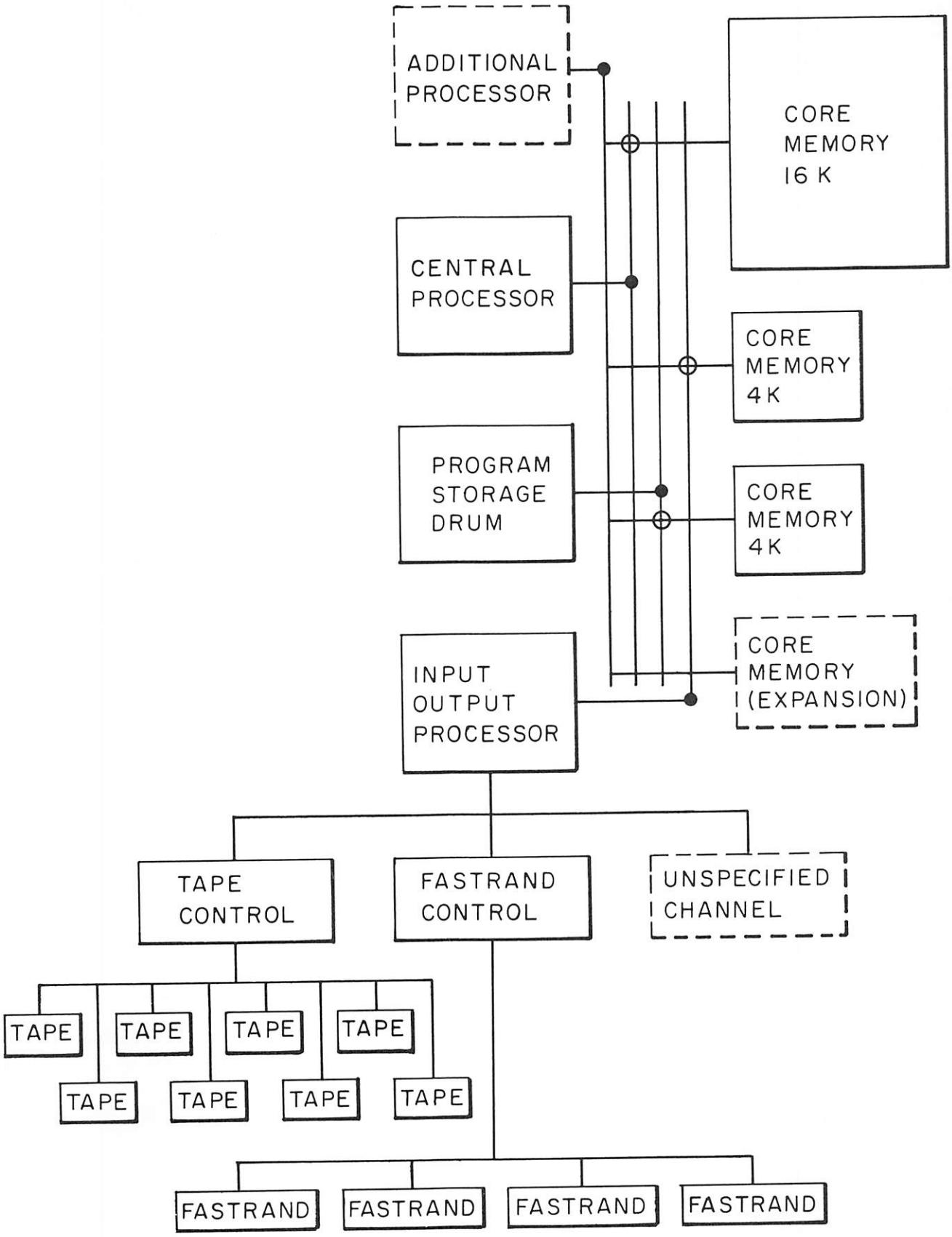


FIG. II-C-4 NEW EQUIPMENT DIAGRAM

III. THE TIME-SHARED COMPUTER FACILITY

As mentioned in Section II, the Time-Shared computer facility is composed of a basic hardware-software complex. Memorandum 6-A described in detail the hardware and the command structure that is hard-wired into the system. The following description of the software must, of course, be read in conjunction with Memorandum 6-A in order to secure a full understanding of the program operation.

A. The Interrupt System

The 16 priority-organized channels in the sequence break system each have assigned to them 4 memory locations in Executive Memory. Channel Ø has assigned to it memory locations Ø, 1, 2 and 3; Channel 1 uses locations 4, 5, 6 and 7, etc. If a break is initiated on a channel having priority, the contents of the accumulator (AC) are stored in the first location for that channel, the contents of the program counter (PC) in the second and the contents of the in-out register (IO) in the third. The PC is then reset to the address stored in the fourth location and the program begins operating in that new sequence. Thus, each of the activities which may occur in a computer system can be assigned a priority level and interrupts can be permitted to any activity by a demand having a higher priority. For example, we want to make sure that data transfers can take place from bulk storage and that they cannot be interrupted by other system activities which can wait. Note that the assignment of priority carries with it no connotation of "importance" but rather the

connotation of "immediacy" as determined by the characteristics of the device producing the break. Thus, if the data channel has been looking for a particular word on a rotating drum and suddenly finds it, its signal to the central processor for attention must be handled immediately lest a complete rotation of the drum be required before the information can actually be transferred. On the other hand, there is no immediacy in the start of the search for such a word and hence initiation of such a search can take place on a low-priority sequence break channel.

Note that the various sequence break channels may have breaks initiated on them by such mechanical activities as the positioning of the paper tape reader or a completion pulse from the paper tape punch or console typewriter; by electrical activities such as the ticks of the 32 millisecond clock and the 1 minute clock; by program outputs from other processor-like devices such as the data channel or the swapping drum; or by program statements from the central processor itself.

The channel assignments for the sequence break system are as follows:

$Channel_8$	Assignment
Ø	Unused (Highest Priority)
1	High-Speed Data Channel
2	Paper Tape Reader
3	Unused
4	Controller Commands

$Channel_8$	Assignment
5	Program Swapping Drum
6	Terminal Scanner
7	Unused
1Ø	One Minute Clock
11	Unused
12	Paper Tape Punch
13	Unused
14	Console Typewriter
15	I-O Processor Program
16	Restrict Mode Trap
17	32 Millisecond Clock

Note that the numbering in the table above is in octal to correspond with the internal number system used in the central processor. We shall frequently use octal numbering when it appears advantageous to facilitate the reader's understanding.

Clearly, such a priority-oriented sequence break system permits, at least conceptually, the kind of program interleaving that allows various activities to take place in accordance with the state of the system at the moment. Any break that occurs automatically causes transfer of control to that portion of the program whose address is stored in the fourth register of that sequence break. To understand the functioning of the Exec, therefore, let us examine the organization of the Executive Core Memory, the individual programs associated with each of the sequence break channels, and the registers which the programs use for mutual intercommunication.

B. Memory Organization

In order to understand the functioning of the Executive system, let us use the actual organization of the present 16,384 word Executive core area as a model. While much of what we say here, and all of the coding, is highly machine dependent, the principles governing the memory organization and Exec operation are applicable, with thought, to Time-Sharing systems in general. Executive Memory is divided into four "cores" each having 4,Ø96 18-bit words. For reasons having to do with the memory protection logic, these cores are numbered 14, 15, 16 and 17.

The gross organization of memory uses core 14 for the Terminal Service Routine, the Swapper and the basic Dispatcher as well as for certain minor I-O routines associated with the paper tape reader, punch and console typewriter. Core 15 is dedicated to the terminal buffer storage area and the on-line debugging program (DDT) which can manipulate the Executive Program directly while the system is running. Core 16 contains the basic Common Routines and the Executive communication registers for communicating with the delegated programs. (Event Detector and Type-Out Text.) Core 17 is the I-O processor and contains the routines for communicating with the high-speed data channel and the file structure in bulk storage. Figure III-B-1 is a more detailed diagram of the layout with the quantitative allocation of memory space indicated by the octal numbering of the registers along the left side of each core.

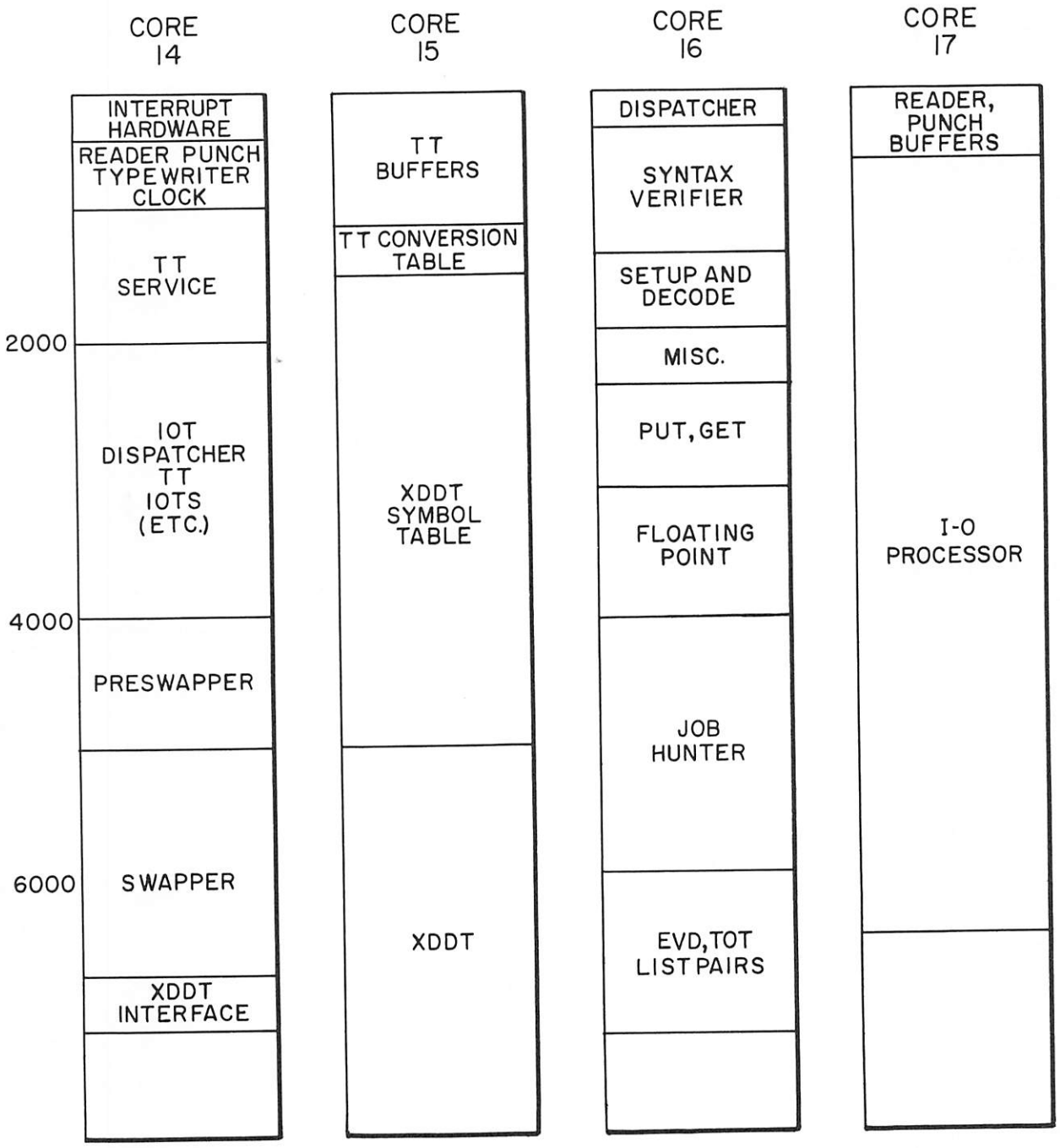


FIG. III-B-1 LAYOUT OF MEMORY

Let us go through the organization of memory as shown and discuss the individual areas in some detail.

In Core 14 the first 100_8 registers are reserved for the 20_8 sequence-break channels that require four registers each. The terminal pointers occupy the next 400_8 registers of memory. The system is currently designed to work with a 100_8 line scanner and requires four registers per line to record the state of activity on that line and the condition of the line. These registers and their codes are described further under the section on the Terminal Service Routine.

As part of the bootstrap operation of getting the system started, an initializer is included in Core 14 as are a set of basic routines to permit reading and punching paper tape and the printout of malfunction messages on the console typewriter. In this same area of the Executive routine is a pair of routines for general use by other parts of the Executive Program. The first of these keeps track of the one-minute sequence breaks that are generated on Channel 10 and suitably indexes a time register and date register. Because of the needs of Event Detector, it also keeps track of 5-minute intervals. The second routine is the routine to monitor the sequence breaks generated by the swapping drum on Channel 5. The routine located here checks for parity errors and timing errors and does the necessary recovery operations in case of a parity error. If no errors have taken place, this routine initiates a Channel 17 sequence break to start the preswapper (qv).

The Terminal Service Routine occupies the next portion of memory and is treated here as a separate entity.

C. The Terminal Service Routine

1. General Discussion

The function of the Terminal Service Routine (TSR) is is to handle the transmission of messages, on a character-by-character basis, from its buffers in the lower part of Core 15 to the terminals, and to receive characters sent in from the terminals. In addition, it must recognize when certain conditions pertinent to the needs of other parts of the Executive program arise. The TSR responds to these needs by setting alarm configurations in its communication registers and initiating sequence breaks as needed.

All characters sent by and to the terminals are treated as 8-bit synchronous serial codes. In order to permit as wide a range of terminals as possible, no checks are hard-wired into the system. All character manipulation and recognition is on a programmed basis. Furthermore, the scanner system can handle bit rates from 6Ø bits per second to a rate in excess of 4,ØØØ bits per second. The Model 33 KSR Teletype used as the basic terminal in the present system operates at 11Ø bits per second. For storage space-economy reasons, the TSR converts each of these characters into one or more six-bit characters.

The Executive Program maintains, in Core 15, one 8-word ring buffer (24 characters) for each communication line in the system. Each buffer serves one particular scanner line, and the same buffer is used for both input and output. These buffers are used by both the Terminal Service Routine and the Dispatcher, and may be addressed by either routine.

As an example of their use, let us assume that a user program wants to type out a long message. The Dispatcher, as we will see in the next section, takes characters from the user program and puts them, one by one, into the appropriate terminal buffer until it is full or until it encounters a message terminator from the User-Program. The Terminal Service Routine in turn, sends these characters to the terminal via the scanner at the proper rate for that scanner line until the buffer is almost empty. The buffering of input is similar; the TSR receives characters from the scanner and puts them in the buffer until the buffer is almost full, or an end-of-message character is received. The user program that "owns" that terminal is then brought into core by the Swapper and the Dispatcher takes the characters from the terminal buffer and transfers them to the user program.

2. Character Codes

The ASCII(*) code presently received by the scanner contains 7 information bits. From other terminals,

*American Standards Committee on Information Interchange.

8 information bits are received. Unfortunately, 7-bit and 8-bit characters do not fit conveniently into the PDP-1's 18-bit word without excessive waste of space. When dealing with 7-bit terminals, the Terminal Service Routine translates each of the 63 most commonly used 7-bit codes into a 6-bit representation called internal code. Three such characters in internal code fit nicely into an 18-bit word. The less frequently used 7-bit codes are represented internally by two 6-bit codes, the first of which is all 1's, and is called a "warning" character. On output, the TSR translates from the internal code back to 7-bit code.

Internal code maintains collational sequence. For 8-bit terminals, the 8 bits are used for input only and are converted to two 6-bit characters containing the high order 2 bits and the low order 6 bits respectively.

A table of internal codes for the various 7-bit characters is included in the section on MIDAS, the programming language. The actual table used for translation in the TSR is located in Core 15. The same table is used for both input and output, and in the case of input also contains 5 bits of control information. These control bits currently describe the incoming character as being one of the following types:

a) normal 6 bit character
b) normal 12 bit character
c) end of transmission

d) null
e) rubout
f) end of message
g) ignore except in control mode
h) control mode terminator

The TSR is started whenever a break occurs on Channel 6; that is, whenever one full character has been received by the scanner. The character received may have been typed in by someone at a terminal, or it may be the "echo" of a character sent out to a terminal by the TSR some milliseconds earlier. This echo is generated internally in the scanner by electrically coupling the transmit and receive relays. Thus on transmission of characters to the terminal from the TSR the break serves the role of a terminal completion pulse.

The 4 words stored in Core 14 for each communication line contain information needed and maintained by the TSR.

Word Ø - Program Flag word

Bit Ø = 1, then suppress echo check
1 = 1, programmer's terminal
2 not used
3, 4, 5, 6, 7, 8, 9 } counter of number of nulls received

Bit 1Ø link bit
11 Ring mode flip-flop (always 1)
12 Program Flag 1 - = 1, Type active (output mode)
13 Program Flag 2 - = 1, in 8-bit mode
14 Program Flag 3 - = 1, just typed out backslash (one echo check has occurred)
15 Program Flag 4 - = 1, terminal off
16 Program Flag 5 - = 1, control mode
17 Program Flag 6 - = 1, line open

Word 1 - Service Pointer, to next character to be removed from buffer

Word 2 - User Pointer, to next available character position in the buffer

Word 3 - Program word
Bits Ø - 5 drum field (Ø - 64) associated with this terminal
6 used by Dispatcher
7 14 echo checks in a row have been received
8 terminal has been interrupted by null
9 terminal did not start
1Ø EOT echo check
11-17 echo check character

When a Channel 6 break occurs, the Terminal Service Routine assumes control and stores away the current status of the hardware program flags. It then reads the scanner to get the number of the terminal causing the break and resets the 6 hardware flags according

to the status of that terminal, as represented in bits 12-17 of word Ø of the 4 words associated with that terminal. Then, if Flag 6 is on (the line is open), it simply ignores the character and debreaks. This immediate debreak prevents a lot of wasted time on an open line since an open line sets Flag 6. When a non-null character is received on a line flagged as open, that flag is cleared and the line is considered closed. If the line is not open, the routine then determines whether it is dealing with input or output by examining Flag 1.

3. Input

If Flag 1 is off, control transfers to the input portion of the routine. The addresses of the service and user pointers associated with this buffer are stored in the locations used in checking for a full buffer, the character itself is read, and the scanner is released. If Flag 2 is off, so that the system is not in 8-bit mode, the base of the translation table is added to the character, and the internal code equivalent is brought into the AC for processing. If bit Ø of the control bits is a 1, the character is a special message terminator, and if the terminal is in control mode as indicated by Flag 5 being on, an alarm must be given to the Swapper to indicate that a time for action has arrived.

If bit 1 of the control bits is a Ø then the incoming character is a 6-bit character. It is stored in the 8-word ring buffer for this terminal, the service pointer is indexed one character, and a check is made for a full buffer. If the buffer is full, an alarm is set. An alarm is also set if the buffer is within three characters of being full. If Flag 2 had been set, the incoming character would have been treated as a straight 8-bit code and would be stored as two 6-bit characters. Eight-bit code is used for external control devices such as the Datacoder.

Eight-bit mode is important to such a system since it permits any arbitrary string of bits, arriving at 11Ø bits per second, to be broken into 8-bit elements and stored internally. These bits may thus arise from analogue converters, timing samples, or other external signal generating devices.

If bit 1 of the control bits is a 1, it means we have an infrequent character which gets translated into a 12-bit character. The 12-bit, or non-simple, character routine looks at the 3 "type" bits in the translation table and dispatches on their octal value as follows:

Ø Normal 12-bit; A "77" and the character's translation are stored, and control is transferred back to where the checks are made for full buffer. If the buffer will hold the "77" but not the code, the code

is lost. Because of the "almost full" check, this case never arises except during a machine failure.

1 EOT (end of transmission). The terminal must be considered "turned off." Flag 4 is set and an alarm is set.

2 Null or break; 1 is added to the null count. If the null count is full (i.e., 128 nulls) the line is considered to be open, Flag 6 is set, and an alarm is given. The very first null will clear Flag 4, since a null turns the terminal on, and will also cause an alarm to be set.

3 Rubout

4 Normal message terminator (EOM). The null count is cleared, the character is stored, and an alarm is set.

5 The character is a control character and is ignored unless the terminal is in control mode. If it is in control mode, the character is stored as a 12-bit character and an alarm is given.

After the character has been stored and the full-or-nearly-full checks have been made, the routine prepares to debreak, but just before doing so, it

checks the status of the Channel 6 break so that if there is another character waiting (from another terminal), it can be serviced immediately without having to take the time to debreak and break again.

4. Output

If Flag 1 is on when a Channel 6 break occurs, control is transferred to the output portion of the TSR. The addresses of the service and user pointer for this buffer are stored away in the area used to check for an empty buffer, and the character in the scanner buffer is read. The scanner is not released at this time. This "echo" character, received back at the computer every time it transmits a character, serves two purposes. First, like a completion pulse, it tells the TSR that it is time to transmit the next character. Second, the character received is checked to see whether it is the same one that was sent. If not, then an "echo check" has occurred, and it is assumed that the terminal misprinted the character.

To indicate to the user at the terminal that the previous character was a misprint, a backslash is transmitted. If a further echo check occurs on the backslash, fourteen attempts are made to send a proper character. The null counter in word Ø of the 4 words is used to make the count. After that length of time, the TSR quits trying and an alarm is given. The characters in the ring buffer that were not sent remain in the buffer, so that the associated program

can read the remaining characters back into its core and determine exactly how much was transmitted and how much was not. This feature permits restarting a typeout of a listing after a line or terminal failure.

The echo check character received may be a null indicating that the operator wishes to interrupt or that the line has just been open circuited by an accident. In that case, an "interrupted by null" alarm is sent to the Swapper, and the terminal goes passive; that is, transmission of characters is stopped and Flag 1 is set to zero. This technique permits the user to interrupt transmission by depressing the break key. Since the terminal has gone passive, the TSR will now accept input characters from that terminal.

If the echo check character is an EOT, then the terminal is presumed to be turned off, an appropriate alarm is sent, and the terminal again goes passive.

There is one peculiar case of an echo check. If an echo check is received when the terminal is off (i.e., PF4=1), it is taken to mean that the terminal has been turned on. Consider the way a terminal is turned on from the computer. Three "nulls" that are supposed to turn it on are sent. Three "rubouts" are then sent to mark time for 3ØØ milliseconds of warm-up time for the terminal. After that time an interrogation command or WRU is sent. If the terminal has properly turned on, the interrogation command will cause it to

transmit one or more characters. If we now send more "rubouts", from the computer they will be garbled by the character or characters that the terminal is sending and produce an echo check. If the terminal did not turn on, the WRU will not be "answered" and no echo check will be received. Thus, the TSR sets PF4 to a zero if an echo check is received when the terminal was off.*

If, in transmission, no echo check has occurred, the buffer is checked to see if it is empty. If not, we have the normal case. The next character is gotten from the buffer, translated into 7-bit ASCII code and stored away for echo-checking in word 4 of the terminal's information words. The character is transferred to the scanner's transmit buffer and the scanner released. The ring buffer is also checked for being within 3 characters of empty. If so, an alarm is set. These 3 character times are a matter of design and, for 11Ø bit per second transmission, give the Swapper time to get the associated user program into core and replenish the ring buffer without a visible "stutter" or pause at the terminal.

*Due to mechanical difficulties in the terminals this technique was temporarily replaced with one which sent 2 null characters and some "rubouts" and simply assumed that the terminal was on.

In the case of an empty buffer, the line "goes passive." Flag 1, indicating type-active condition, is cleared.

A summary of the conditions recognized by the Terminal Service Routine and transmitted as alarms is listed below. A number designating the specific alarm and the associated terminal number are put in a ring buffer for the preswapper to examine and take action on.

Alarm Type	Meaning
Ø	EOM, buffer almost full, DDT terminator
1	14 echo check errors in a row received
2	Line open
3	Interrupted by null (ordinary terminal)
4	Interrupted by null (control terminal)
5	Not used
6	EOT received as echo check
7	Terminal buffer almost empty
1Ø	Not used
11	Program turned terminal off (EOT received)
12	Terminal turned on (control type)
13	Interrupted by null (Executive terminal)
14	Terminal turned on (ordinary terminal)

Clearly, many of the behavior characteristics of the Terminal Service Routine must be under control of a user program. Shifting to 8-bit mode, instructing the service routine to type out some text in user core

and other activities are controlled by a specific set of IOT's which affect the Terminal Service Routine. The reader is referred to Chapter IV which deals with the Executive Common Routines since these IOT's fit better under that heading.

D. The Dispatcher

As pointed out in our discussion of a prototypal Time-Sharing system in the preceding chapter, any program struction that would cause a holdup of the system is trapped by a trap buffer and initiates a sequence break to the portion of the Executive Program called the Dispatcher. Since the low order 15_8 bits of the instruction are stored in the trap buffer, the Dispatcher can use an address table in order to branch on the numerical value of the instruction code. The present system thus has the facility for recognizing 8,192 such trapped instructions. Of these, only a few hundred have as yet been implemented. Each such IOT is represented by the starting address of a routine in Exec and the Dispatcher executes an indirect jump through that starting address stored in the dispatch table.

It can be seen that the IOT's handled by the Dispatcher are, to the user programmer, essentially the same as new machine commands. In general these IOT's are more complex than machine commands, may require somewhat longer calling sequences and in general perform those operations normally associated with closed subroutines. Because of this similarity to machine commands, we will not go through the IOT's as we go through the memory organization but will treat them separately in the next chapter.

E. The Preswapper and Swapper

Core 14, from location 4000_8 to 6400_8, is occupied by the pair of routines which are, in essence, the heart of our Time-Sharing system. These routines permit swapping running programs from the small high-speed memory to the bulk storage element that backs it up (in this case a drum). In more general terms, the Preswapper and Swapper transfer control among the programs in accordance with the system's internal algorithm as to who shall run and who shall wait. As described in the previous chapter, this function would be required even if all of the programs were stored in high-speed memory.

The action of Exec in transferring control from one user program to another can be broken down into two parts: (1) analyzing the signals that have arrived from the outside world, from other portions of the computer and from other parts of the Executive Program in light of the status of each program in the system and revising those status descriptions accordingly; (2) actually moving programs from bulk storage into high-speed storage and transferring control of the computer in accordance with the status descriptions. This latter function must provide for the efficient utilization of memory space and the time available to the computer.

1. The Preswapper

All activity on the part of the Preswapper is initiated by a sequence break taking place on Channel 17. These breaks are initiated by ticks of the 32 millisecond clock and also, of course, by commands from other parts of the Executive routine. Thus we have a guarantee that the Preswapper will examine the outside world at least thirty times per second.

a) Program Status

The condition of a program is set by the Preswapper in the first six bits of a status word. This word contains, in its last twelve bits a description of the program's queue level. The various statuses are described below. The queue level will be described in Section 2.

The term "program status" is applied simultaneously to the program and to the section of memory in which it is located. The status conditions recorded by Exec and the octal equivalent of the high order six bits in the status word are as follows:

Runnable ØØ A program is runnable if transfer of control to the location indicated by its program counter would either permit that program to run or would execute an IOT for the first time.

Type-out Hung 71 A program is hung for type-out if it has issued a type-out command that demanded the transfer of more characters from the local program buffer into the Executive terminal buffer than the Executive terminal buffer could hold. When this condition occurs, the program is put into a suspense state until such time as the terminal buffer is sufficiently empty to warrant another transfer of characters from the program's local buffer to the terminal buffer. In the present system, such a transfer is considered warranted when the terminal buffer is within three characters of being empty.

Note that if the transfer of fewer than 24 characters was demanded by the program or if an EOM was encountered on any transfer, the program is not hung but continues its computation while the Executive Terminal Service routine is typing out the characters on the proper terminal.

Type-in Hung 7Ø A program is hung for type-in if it has requested that characters be transferred from the terminal buffer to its local buffer but no characters are available in the terminal buffer for that purpose. The program is put in a suspended state until such a transfer can economically be made. At present, we consider that a transfer can economically be made

when the terminal buffer is within three characters of being full or a message terminating character has been received.

The nature of a message terminating character depends on the classification of the terminal connected to the program. In general, however, the EOM is a message terminating character recognized by all programs. No transfer of control takes place to the user program until a message terminating character is received or the terminal buffer is nearly full.

Reader Hung 72 Since the computer center contains a perforated paper-tape reader, and since the reader is under character-by-character control of the Exec, data transfer from the perforated tape to user programs is accomplished through the perforated-tape reader IOT's located in Core 14. Since perforated tape does not, in general, contain end-of-message characters (message terminators), transfer of control to the user program must take place as the tape-reader buffer fills. A special 2ØØ-word reader buffer is located in Core 17 for this purpose. The tape buffer is larger than a terminal buffer because the reader runs much faster than a terminal.

Punch Hung 57 Hung waiting for the punch buffer to empty. The central installation contains a medium-speed tape perforator which is accessible from the user programs via a set of IOT's. Core 17 contains a 12Ø-character punch buffer which Exec handles like a terminal buffer. A user program that is punching tape is put in suspense until the punch buffer is within 2Ø characters of being empty at which time a suitable alarm is set for the Preswapper. Because of the higher speed of the tape reader and punch, each of these buffers operates with a 2Ø-character margin as opposed to the 3-character margin used by the terminal buffers.

Get-Terminal Hung 74 When a program wishes to initiate activity on some terminal, it may request Exec to assign that terminal to it. If the Terminal is already occupied, i.e., it is assigned to some other program, the requesting program is "Get Terminal Hung" until the Preswapper receives an alarm from the Terminal Service Program indicating that that terminal has just been released. Note that this particular type of "hang" may put a program into a suspended state for very long periods of time since its termination may depend on human operator action. For this reason, it is wisest to communicate with other terminals through Type-Out-Text (Volume VI-C) when possible.

Data-Channel Hung 73 When a user program requests the transfer of data from bulk storage to its core area, the data channel may already be occupied performing such a transfer for another program. In this case, the requesting program is data-channel hung until such time as the I-O processor section of Exec signals that the request can be serviced.

Data Channel Occupied 40 Since the transfer of information from bulk storage is a transfer from an inertial device, such transfers may not be interrupted. To prevent such interruption, the program participating in such a transfer is marked as data channel occupied. No alarms are serviced for that program until such time as the I-O processor section of Exec indicates that the data transfer has been completed.

Memory Area Free 76 If a memory area is tagged as being free, then it means that the program that had occupied that area (and may still be occupying that area) has completed its activity and come to a HALT instruction. Such an area is available for assignment to any new terminal requesting service.

Memory Area Unavailable 77 If a program attempts to execute an instruction which is undefined, or if parity errors occur repeatedly

on reading, the status of that program (and hence the memory area associated with it) is changed to "memory area unavailable." Since this alarm indicates a non-recoverable hardware or software malfunction, manual intervention by the system operator is required to modify this status.

Terminal Start 24 This status indicates that a terminal which had previously been off has received a "null" and is hence presumed to be turned on and interested in communicating with the computer.

Terminal Null 7Ø A program is marked as Terminal Null if the terminal assigned to that program was a control-type terminal and a null was received on that line.

Get List Pairs Hung 5Ø The special programs to which the Executive routine delegates its activities obtain their "orders" as "list-pairs" stored in a list-structured buffer in Core 16. These list-pairs are sent to Exec via an IOT from user programs or from other sections of Exec. When a delegated program requests a list pair on which to act and finds none, it is then assigned the status "Get List Pairs Hung" until the execution of a list-pair assignment IOT.

Event Detector Start-Up 26 Since a program can be started up by one of Exec's delegated programs called Event Detector, this status is assigned to a memory area to indicate such a program start-up.

There are two additional status notations, Core Hung and Wanted by Exec DDT. These very special statuses are used only by Executive DDT to get control over programs. They are not otherwise available.

In order to understand how the Preswapper and Swapper deal with alarms in accordance with a program's status, we must first understand the queuing algorithm for the present system.

b) The Queue

The 12 bits of the status word following the octal digit codes in the preceding section, are used to designate one of twelve queues in which that particular program is located. Every user program capable of running (i.e., whose status word is set to "runnable") is in one of the twelve queues. A program in the highest priority queue is allocated a quantum of time of 32 milliseconds to run when its turn comes up. A program in the n-th lower priority queue is allocated $32x2^n$ milliseconds to run when its turn comes up. Within a given queue, programs run in rotation.

By this algorithm, a program in the highest priority queue gets only 32 milliseconds of computer time when it is finally set running. A program on the lowest priority queue, however, gets slightly over 2 minutes of running time when it is finally set running.

Since the Preswapper re-evaluates the decision as to who should run at least every 32 milliseconds, this algorithm ensures us that programs communicating with the outside world (those that are on high queue) will be serviced rapidly while those that are doing long computations will, when they run, be spared the computational inefficiency of frequent swaps.

It is important to note that no program is swapped in to run if there is any runnable program in a higher priority queue. When a program runs to the end of its allocated quantum of time without getting hung or being interrupted by a higher priority program, it is put at the end of the next lower priority queue. Thus, programs which are predominantly occupied with computation gradually give way to those which are exchanging information with the outside world.

c) Alarm Handling

As described in Section III-C, the Terminal Service Routine, there are 16_8 types of alarms generated in recognition of remote terminal activity. They are:

Alarm Type	Meaning
Ø	EOM, buffer almost full, DDT terminator
1	14 echo check errors in a row received
2	Line open
3	Interrupted by null (ordinary terminal)
4	Interrupted by null (control terminal)
5	Terminal did not turn on
6	EOT received as echo check
7	Terminal buffer almost empty
1Ø	Plant a call to library DDT
11	Program turned terminal off (EOT received)
12	Terminal turned on (control type)
13	Interrupted by null (Executive terminal)
14	Terminal turned on (ordinary terminal)
15	Plant a call to start-up program

In addition to these alarms, there are two alarms generated by the paper-tape reader and paper-tape punch which are also processed by the Preswapper. Further, the data channel which controls the transfer of data from bulk storage to core sets a flag in the Preswapper upon making the transition from busy

to free as does the swapping drum. If we refer back to Figure III-B-1, we will recognize that these alarms from the input-output Processor and program storage drum permit the Preswapper to control the setting of the connections to those processor-like devices.

Each of the terminal alarms and the number of the terminal to which it is pertinent is placed in a ring buffer from which it is removed by the Preswapper. The user of a ring buffer insures that the alarms will be handled in sequence. This ring buffer is examined every time that a break occurs on Channel 17 regardless of whether that break is occasioned by the ticking of the 32 millisecond clock or by the program initiation of a sequence break. Only in the case of a 32 millisecond clock break, however, does actual swapping take place. In the other cases, the Preswapper merely uses the alarm buffer to update the statuses of the pertinent programs.

Let us now examine the specific servicing of the alarms. When an alarm is discovered in the buffer, the Preswapper immediately checks on whether there is a program associated with the terminal causing that alarm. Only in the case of alarm type 12 and alarm type 14 is it permissible to have an alarm without an associated program since both of these indicate that a terminal just turned on. If the alarm is neither 12 nor 14, the alarm is removed from the buffer as meaningless and ignored. If it is not meaningless, the Preswapper dispatches on the alarm type as follows:

Alarm Type Ø (EOM, buffer almost full, DDT terminator)

If the user program is not type-in hung (as indicated by his status word), then the information in the buffer is garbage and not pertinent to the program so the alarm is ignored. If the user program is type-in hung, it is waiting for this information. The program status is then set as runnable and the user is placed on the highest queue.

Alarm Type 1 (14 echo check errors in a row received)

This alarm indicates a malfunction of the communication line to the terminal and hence produces an error message type-out on the console typewriter at the operator's position in the central office. The user is returned to high queue so that the computer may try his line again. Should repeated echo check messages be printed out at the operator's position because of repeated 14-character failures, the operator may intervene manually and set that line to "bad condition."

Alarm Type 2 (Line Open)

This alarm is generated by the receipt of 128 nulls or by approximately 13 seconds of no current on the line. A suitable error message

is typed out at the central console typewriter but the failure is treated exactly as is Alarm Type 1.

Alarm Type 3 (Interrupted by Null on ordinary terminal)

This alarm is generated by the receipt of at least one null when the terminal is not running under DDT and is not in "restart mode" both of which are discussed later on. The receipt of a null causes the user's status word to be set to "INTERRUPTED BY NULL". If the user is type-in hung, the null is treated like an end-of-message but produces a first or "bad" return from the type-in IOT because of the condition of the terminal status word. If the user is not type-in hung at the moment, the execution of the next terminal IOT, whether for inputting or outputting, results in a first return. In either case, word 45 of the user's core has bit 8 set to a 1 to indicate that a null interrupt has occurred. The user program deals with that fact either according to some internal subroutine or by executing a "HLTBAD" IOT as discussed in the next chapter.

Alarm Type 4 (Interrupted by Null on control terminal)

This interruption indicates that the control terminal wishes to return control to the DDT program Volume VI-E. The Preswapper puts a

breakpoint to DDT in the user's status word and, if the user is not using the data channel, prepares to swap. If the user is using the data channel, then the flag indicating readiness to swap is not set to zero and more alarms are looked for. This procedure permits synchronous data transfers to proceed to completion.

Alarm Type 5 (Terminal did not turn on)

This alarm indicates that an attempt was made by a program to turn on a terminal but that the electro-mechanical turn-on was not accomplished. An appropriate error message is typed out on the central console typewriter and, if the user is type-in hung, his program counter is set back one instruction so that the system will try again to turn on the terminal. If repeated messages are typed out on a central typewriter, the terminal is assumed to be inoperative and the operator initiates a service call.

Alarm Type 6 (EOT echo check error)

This alarm indicates that a line malfunction caused a normal character sent out by the computer to be transmitted on the line as an EOT which would turn the terminal off. At the present moment, for simplicity, this error merely causes the transmission of a backslash

which is generally sufficient to restart the terminal. Should the line malfunction continue, it will be detected by the failure of the backslash to transmit properly and a normal echo check error situation will occur. For that reason, Alarm Type 6 is currently handled in exactly the same way as Alarm Type 1 (with a different message, of course). It is reserved as a separate alarm type, however, to provide for the use of terminals in the system which would not normally be turned on by the transmission of a backslash.

Alarm Type 7 (Terminal buffer almost empty)

The user program is examined to see if it is type-out hung. If not, the fact that the buffer is now almost empty is no longer of concern. (The program has presumably gone back to computing) and so the alarm is ignored. If the program is type-out hung, then it has more information in its local buffer for transmission to the terminal buffer. As a result the program status is set to highest queue and runnable, and the swapping flag is set to Ø preparatory to swapping.

Alarm Type 1Ø (Plant a call to Library DDT)

This alarm, used in debugging, permits the call of a special program with characteristics that will be discussed in Volume VI-E. It transfers

the user to highest queue, sets its status as runnable and prepares to enter the actual Swapper.

Alarm Type 11 (Program Turned terminal off)

This alarm indicates to the Exec that a terminal which has been busy with a program up to now has just terminated its connection to that terminal. As a result, the terminal is now available to receive any messages that may be waiting for it as a result of earlier communication. On receipt of this alarm, the Preswapper first examines the special program "Type-Out Text" to see if it is looking for any terminal at all (i.e., is it Get Terminal Hung). If so, it does not check on whether this is the terminal that is being sought but sets Type-Out Text's status to runnable and places it on the highest queue. Type-Out Text will then, of course, re-execute its Get Terminal instruction and if this is not the terminal being sought will automatically go to a "Get Terminal Hung" status.

After such an abortive trial or if Type-Out Text was not originally Get Terminal Hung, all of the user statuses are examined to see whether any user is Get Terminal Hung. If not, the alarm is of no importance and is cleared. If a user is found in Get Terminal Hung status, the number of the terminal for which he is looking is compared to the number of the terminal

just released and, if a match is found, that user is placed in Run On Highest Queue status and the Preswapper prepares to enter the Swapper.

Alarm Type 12 (Control terminal just turned on)

This alarm indicates that a control terminal which had hitherto been quiescent was just activated, presumably by the operator pressing the signal key and thus transmitting breaks down the line. The Preswapper searches the list of available drum fields to determine whether there is sufficient memory space left to initiate a new program. If no memory space is available, the Preswapper types back the message "Please Call Back Later" and turns the terminal off.

If there is free memory space available, that drum field is assigned to the calling terminal, the number of the terminal is stored in the Teletype buffer area of the Swapper and the status word is set to indicate a call to the Library Start-Up Program. The program then prepares to enter the Swapper.

Alarm Type 13 (Start Exec DDT)

This alarm is used by the special debugging program Executive DDT so that it may examine any of the user programs that are running under

the Executive at any time. It serves to restore the actual status value for programs "held" by Executive DDT, sets the Exec DDT status to run on highest queue and prepares to swap.

Alarm Type 14 (ordinary terminal just turned on)

Because the system does not at present distinguish between control and ordinary terminals before they are activated, this alarm is handled exactly as is Alarm Type 12.

Alarm Type 15 (Plant a call to the Start-Up Program)

This alarm is used by the Swapper in loading an available space on the drum with a library copy of the Start-Up Program, entering the terminal number in word 65 of that core space and starting the Start-Up Program.

2. The Swapper

Once the Preswapper has updated the status words of the various programs, it falls to the Swapper to perform the necessary evaluations, execute the necessary program exchanges and transfer control to the proper user program.

a) Swaper Status Tables

In order to perform its functions properly, the Swapper makes use of three tables. The first and most complex is the STAT table, consisting of the status-queue words discussed in the previous section. There is one such word for each program space in the system. With our present swapping drum, we allow for 32 simultaneous programs and hence have 32 entries in the STAT table.

The second table contains an entry for each user core bank to indicate the status of that core. The core status word may have one of three values. It may be zero, indicating that that core is currently engaged in a little-drum swap, or it may be negative indicating that it is connected to the data channel. A positive entry in the core-status word indicates that the core is occupied by a program. The positive number in that case is a pointer to the entry in the STAT table containing the status and queue level of that program.

With those two pieces of information available, the Swapper needs only to know the status of each of the processors in the system in order to have the system completely defined. The processor-status table has three entries, one for each of the processors:

1) The central processor

2) The data channel

3) The swapping drum

The high-order 6 bits of the processor status word contains the number of the physical core being accessed (4 or 1Ø in the present case). The lower-order 12 bits contains the address of the entry in the core-status (STAT) table for that core. If the processor is idle (i.e., not connected to any core) then that processor entry is zero.

b) The Swapper Decisions

Based on the entries in the status tables, the Swapper makes a simple set of decisions. It's primary responsibility is the interchange of programs between core and the swapping drum. Naturally, if the drum is engaged in a swap, no interchange is possible. If, on the other hand, the drum is free (a zero in word 3 of the processor table), then the Swapper goes through its routine.

By examination of the STAT table, the Swapper determines the highest-queue program on the drum (excluding those in core). It also determines the lowest-queue program in core. (An empty core is the very lowest queue!) If the highest

drum queue is higher than the lowest core queue, the Swapper makes the necessary table entries and initiates an interchange of those two programs. At the completion of the interchange, it starts its comparison over.

If the swapping drum is busy, or if the STAT entries show no interchange to be made, the Swapper transfers control. If one of the cores contains a runnable program, control is transferred to it through the post-swapper. If, on the other hand, neither core is "runnable" because it is hung, or transferring data etc., control is transferred to the Preswapper "waiting loop" which marks time until the status of one core changes.

F. File Organization

Before we can go on to discuss the Executive Common Routines, we will need to understand the file structure as it exists on the Fastrand drum which serves as the bulk storage medium for the present Hospital System.

As is common with file structures in general, we will occasionally talk about logical entities in the file and occasionally talk about physical entities. A logical entity is one whose structure is determined by and (therefore hopefully reflects) the logical structure of the information with which it deals while the physical structure in general reflects the characteristic of the particular machine-dependent medium with which it deals.

1. Internal Logical Structure

While much of the internal logic of our files is similar to that reported earlier in this project, there are enough salient differences between the present Phase 2 system and the Little Hospital System reported in 1964.

The logical structure of our files reflects in part the many tasks which must be done by a Hospital Computer System. Our files are organized into research files, the files containing the active patient records, the programmers files, the library files and the files of messages awaiting manipulation by other programs. The most complex of these and the one that is most closely related to the substantive task of manipulating hospital records is the active patient record file or the APR file.

The APR file is organized by patient (access to the file will be discussed later). The largest logical structure within the file is a record and is designed to contain all the information dealing with that patient. The record itself is subdivided by episode, an episode being logically defined as the period between an admission and a discharge inclusive. Within the episode, information is organized by item class. Thus, for example, medication order items and medication charting items have the same class since both deal with medication in general. Within the class we divide our information into a logical item

which is logically defined by the program generating it and usually refers to a body of information generated at one time either in connection with a single interview or in connection with a single occurrence.

Within the item itself we have fields and groups of fields. The field is the smallest piece of information which is a logical entity and is thus the smallest element normally manipulated by outside programs. Each such field has a name (such as "address") as well as a value such as "147 West Moreland Avenue."

Fields are frequently logically grouped to indicate a concatenation. For example, the field named "operation" might be grouped with the field named "surgeon" in order that we might be able to do a search for a patient who had an appendectomy performed by Dr. Jones. Without such grouping, if the appendectomy were performed by Dr. Smith and a tonsillectomy had been performed by Dr. Jones, the patient would show a positive truth value for the surgeon Jones and for the operation appendectomy. The concept of grouping is required in order to give logical meaning to some of the Boolean operations required for sensible retrieval.

Although it is usually only important to programmers, one has elements which are frequently called primitive fields or in our case "atoms." In our example, the first primitive field in the field named "surgeon"

that had the value "Jones" would be the character "J". Naturally, because of the nature of our internal storage, there is a smaller subdivision than the primitive field, namely, the binary integer or bit. Only in dealing with encoded fields, however, are we likely to be concerned with their binary representation and hence with the use of bits as primitive fields.

The logical structure of the Hospital System is such that no limit is imposed on the level of subscripting of episodes within a record, on items within an episode, on groups within an item or fields within a group. Further, the primitive fields, making up the information fields, are unrestricted in number and that number needs not be specified.

The representation of information within a field is either in the form of text or in the form of an encoding which involves some algorithmic transform. The text has the characteristic that it can be directly transferred to communication routines without a decoding operation. It further logically represents information that can only be expressed on a nominal scale. It is amenable only to symbol manipulation routines and not to the logical or mathematical routines that require ordinal or interval scale representation.

Encoded data may or may not be logically and internally manipulable by Boolean or mathematical routines. The algorithm used to convert it may have been one which recognized the quantitative value of the symbol string used to input the information (such as a string of symbols 846 which has a quantitative meaning in the decimal system). Alternatively, the encoding algorithm may have been a simple dictionary look-up or may represent some complex hierarchical encoding structure such as the international code for the classification of disease. In any case, the value of the field is the encoding and not the symbol string used to derive the coding nor the symbol string which will be used to represent the value of the field on presentation to an observer.

2. Internal Physical Structure

While the structure of the bulk storage hardware is described in detail in Section 2.4 of Memorandum 6-A, the Fastrand has the following essential characteristics:

There are 64 heads which can be moved as a group to one of 96 positions for a total of 6144 "tracks." Each track is composed of 64 sectors, each of which contains 50 information words and one tag word. The tag word is used internally for list structuring with the result that there are some 19,000.000 words available on the bulk device.

Program optimization dictates that as much be done without moving the boom as possible in order to minimize time loss and that a "write command" be executed on the nearest available block or sector. This optimization is done by the I-O Processor section of the Executive Program by the routines stored in Core 17.

While system usage has not yet required it, the present system is designed to handle four drums, each of twice the capacity just described.

In addition to the movable recording and reading heads, the drum contains a set of fixed heads, seven of which are used for storing information that requires rapid access and one of which is used for test and service purposes.

The 96 track positions are divided into thirds of 32 each, only one of which is used for current activity, the others being retained for experimentation and debugging. The 32 tracks actively being used are divided among the research files, the APR files the library files, the index area, the message area and the programmer's tracks.

The smallest physical block of information currently handled by the I-O Processor is the segment containing 5Ø 18-bit words. The smallest logical information subdivision handled by the I-O Processor is the item. An item, therefore, always consists of an integral number of 5Ø-word blocks. The blocks making up an

item may be either list structured (chained) or may be organized through the use of a Table of Contents or "toc". In the latter case, one block (the first) contains the addresses of all the other blocks in that item.

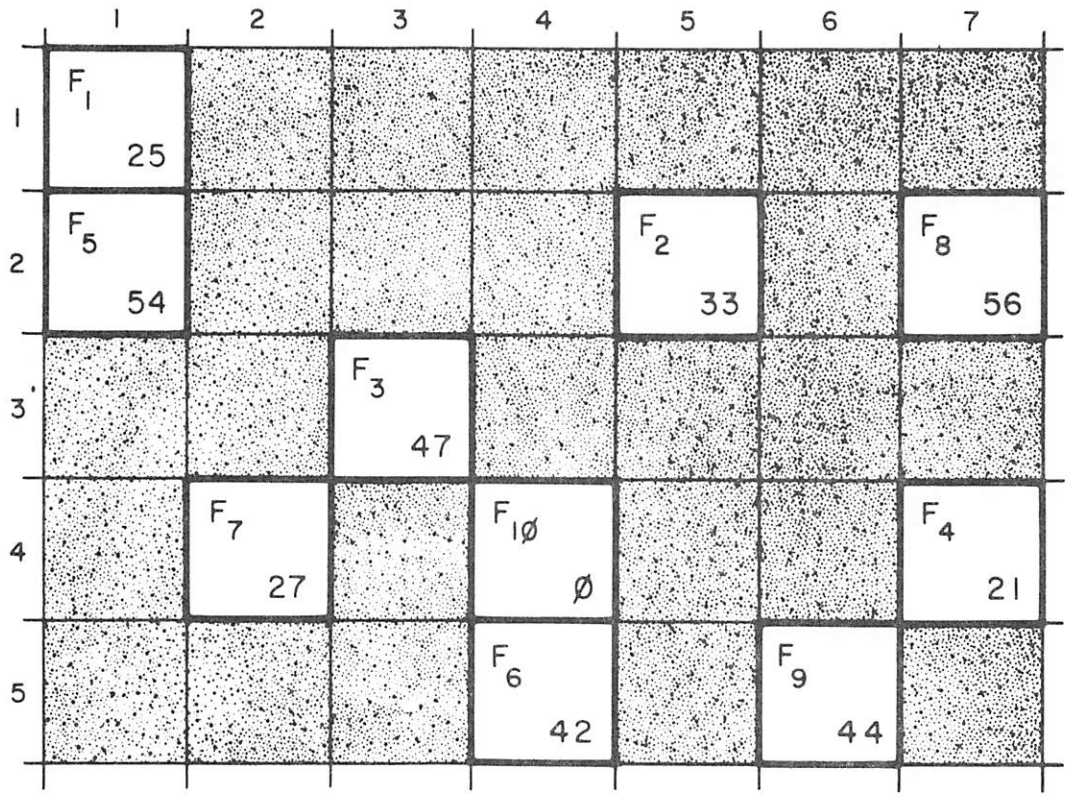
While chained files have been previously described, their organization is repeated here for convenience.

A block is the smallest piece of memory space that will be filled or emptied at any time. Figure III-F-1 is a picture of a small region of a memory map containing 35 such blocks arrayed in five rows and seven columns. The crosshatched blocks are assumed to contain information as a result of prior operations and the non-crosshatched blocks labeled F_1 through $F_{1\emptyset}$ are blocks that are free and ready to accept information storage. One of the characteristics of a chained file is that it is capable of completely filling a space such as that shown in Figure III-F-1. This figure represents one 4Ø96-block head position on the drum.

Each free block in Figure III-F-1 contains in its tag word the address of the next free block. For example, if we look at the tag word of block F_3 we see that it contains the number 47 which is the address of the next block F_4. Note that the word "next" refers only to this list (the so-called free list) and does not imply any physical proximity. We also keep a set of four control registers in our example: The first of

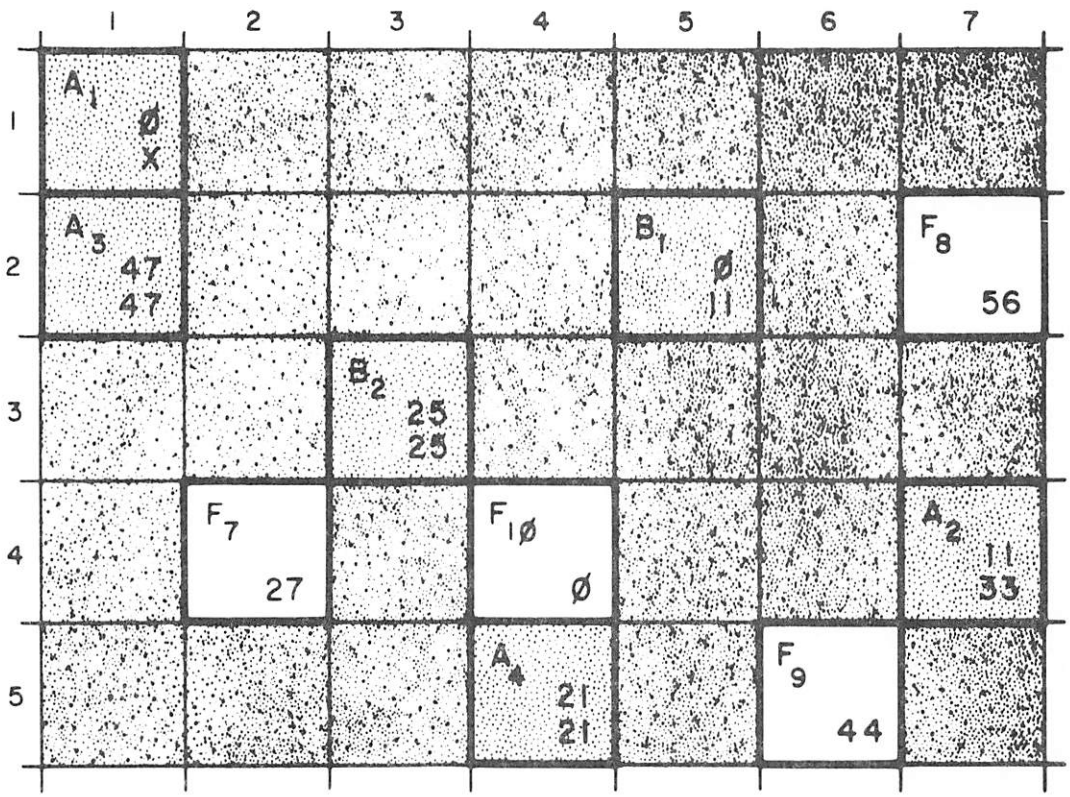
these, labeled "free", contains the address of the first free block available, in this case the block labeled F_1. The second, third, and fourth registers labeled "last a", "last b" and "last" contain respectively the locations of the last "a" type block filed, the last "b" type block filed and the last block of any type that has been filed in this file space. We are assuming that we have not yet filed an "a" type or "b" type block, so those two registers contain zeroes, whereas the last block of any type was filed at some arbitrary location which we have labeled x. These control registers are shown at the bottom of Figure III-F-1.

Let us now consider filing 6 blocks of information in our demonstration space. The blocks, and the order of their filing, will be a-1, b-1, b-2, a-2, a-3, a-4. The foregoing list is both a single list of 6 blocks and the merging of two lists, one being a-1, a-2, a-3 and a-4 and the second being b-1 and b-2. For convenience in searching, our filing system should be capable of tracking each of these lists independently. Referring to Figure III-F-2, we can see the state of the system after having filed the 6 blocks. At the bottom of the figure we have shown, reading from left to right, the state of the four control registers after filing each of the blocks.



FREE	11
LAST A	Ø
LAST B	Ø
LAST	X

FIG. III-F-1 THE FILE SPACE



	A_1	B_1	B_2	A_2	A_3	A_4
FREE	11	25	33	47	21	54	42
LAST A	Ø	11	11	11	47	21	54
LAST B	Ø	Ø	25	33	33	33	33
LAST	X	11	25	33	47	21	54

FIG. III-F-2 THE FILE SPACE AFTER FILING SIX BLOCKS

The basic filing algorithm is such that as each block is to be filed, it is put into the space occupied by the first free block (i.e., the block whose address is in the control register labeled "free"). The address found in that free block points to the next available free block and hence is substituted for the address in the "free" register. The address that had been in "free" is inserted in the control register labeled "last" since it now points to the last (most recent) item filed. It is also inserted in either "last a" or "last b" depending on whether the block just filed was an "a" or a "b". At any moment then, the control registers (4 in our example) indicate the locations of the first available free block, the last block filed, and the last block according to category.

Thus when we file block a-1 in position 11 we pick up the number 25 which was in that position (see Figure III-F-1) and use that as the address of the new first free block. Since we have just used up block 11 we list that as the last block filed and since we filed an "a type" block in that location we list 11 as the last "a" block. As a result, the second column of control registers now shows block 25 to be the first one free. It shows both the last "a" block and the "last" block in block 11 and the last "b" block labeled as ∅.

The next block we wish to file is a "b" block. By our rule it gets filed in block 25 where we find the new address 33. Thirty-three now becomes the new contents of "free" and block 25, the block we have just occupied becomes the new entry under both "last" and "last b".

The reader is urged to follow through the remaining steps until he arrives at the final state of the example wherein block 42 is the first free block, block 54 is both the last "a" filed and the "last" block filed and 33 is the last "b" type block filed. Note that we are now left with four unoccupied blocks, having filled six of the previous ten. One should also note that the address in Block F1Ø, the last block available in our file space, contains the address Ø. When Ø is uncovered in the free list, the memory space will have been completely filled. At this point, the I-O Processor will cause a boom movement.

In each block filed we carry, in one register, the location of the previous block filed. We also carry, in the next register, the location of the previous block of the same class. Thus the "a" type blocks contain a list of all the "a" type blocks.

The process of unfiling is completely analogous to the process of filing. For example, let us suppose that the "a" blocks all refer to one laboratory item that the "b" blocks all refer to another. Let us now suppose that we wish to expunge item b. Figure

III-F-3 illustrates the file space after unfiling the "b" blocks. The rule for unfiling is the inverse of the filing rule. It states that we take the "last b" entry and substitute it for the contents of "free" writing the "free" entry into the tag-word of the space vacated by the last "b" block. Thus the first available free space becomes the second in the free list.

The pointer to the next "b" block in line occupied the next-to-last register of the "b" block just unfiled and is substituted for the contents of the "last b" register. Note from Figure III-F-3 that we are now left with only "a" blocks filed and with a 6 block "free" list.

Just as this technique of filing blocks within a logical item has obvious advantages, the items within a record are handled the same way. Each item contains a header which points both to previous items of that type within the record and to the previous item in that record regardless of its type. The patient parameters contain the control registers for each patient.

The other form of item construction with which we are concerned is the TOC'd item. The first block of a TOC'd item contains a list of addresses of the blocks making up that item. The first word of the TOC block is the item length in words and following it is a list

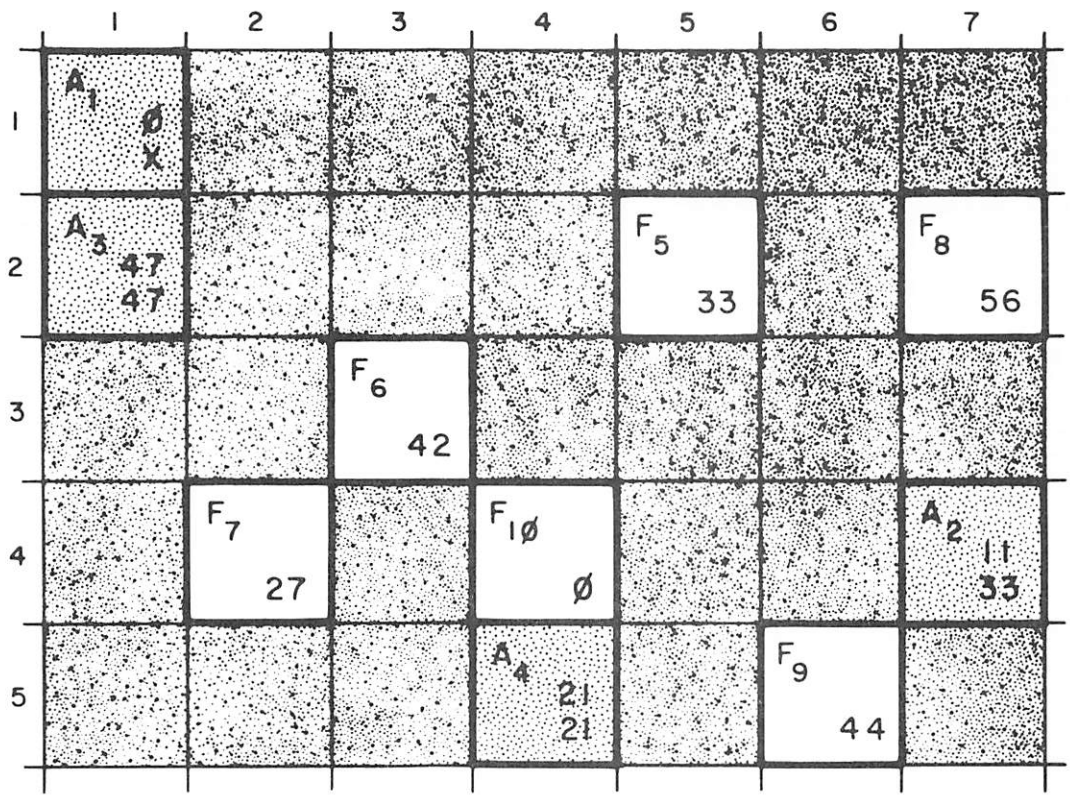
of block addresses. This form of item has some advantage over the chained item in terms of retrieval speed of very large items and is a convenient way of structuring the item when the information contained in it is to be frequently rewritten.

The detailed handling of the chained and TOC'd items is treated under the section describing the IOT's used in conjunction with those items.

3. File Access

Because of the type of structure represented by a hospital in a community, we have chosen to use a hierarchical access system organized on the basis of Organization (or Hospital), Patient Care Unit, and Bed. The fixed heads on the drum are treated like a linear file on which address arithmetic can be performed. Given the bedspace, room, Care Unit and hospital, address arithmetic permits the retrieval of a block of six registers two of which contain the patient unit number for that patient record and the others of which contain pointers to certain entry points into the record itself. The structure and use of the actual linear fixed head files is discussed under the patient parameter section of the Executive Common Routines.

In addition to the most frequently used method of access, a hierarchical indexing system is provided which permits a programmer to specify up to a 36-bit



	UNFILE B_2	UNFILE B_1
FREE	33	11
LAST A	54	54
LAST B	25	Ø
LAST	54	54

FIG. III-F-3 THE FILE SPACE AFTER UNFILING THE B BLOCKS

indexing code for any given record. This code may then be used in the future to provide a high-speed access to the record by means of those record attributes used in deriving the 36-bit encoding. The indexing, index retrieval and index file structures are described in Chapter V under the indexing function.

Much of the activity within this fairly complex file structure takes place out of the user programmer's sight, since the Executive routines in Core 17 (the I-O Processor) maintain the necessary surveillance over the available memory space and are concerned with the transition from logical entities to physical entities. In addition, those user programmers who wish to utilize some mode of optimization of their own have a wide freedom and can control both logical and physical entities at will with suitable safeguards imposed in keeping with the portion of the file structure being utilized. The scope of this freedom is more clearly evidenced in the next chapter.

IV. EXECUTIVE COMMON ROUTINES

The inter-communication between programs and the communication between programs and the outside world is accomplished through the user programmer's use of a set of trapped commands called Executive IOT's. In addition, the Executive Program provides the user programmer with many general facilities. These are needed sufficiently frequently by user programs to warrant their call by other IOT's. This entire group of routines, called the Executive Common Routines, forms a very important part of the overall system and provides it with much of its usefulness. In the following chapter, the IOT's calling these common routines are grouped according to their primary function. Many of these IOT's have more than one function or have functions which do not fit into any simple classification scheme. In such cases, their assignment to groups is arbitrary.

Returning to our analysis of function by core location in the Executive Program, most of these IOT's refer to routines stored in Core 16 from $\emptyset$ to $6{,}\emptyset\emptyset\emptyset_8$ and in all of Core 17. The upper part of Core 16 contains the conversion tables for the Terminal Service Routine, certain tables used by Event Detector which will be discussed further on, and the list structured buffer for the list pairs previously mentioned.

Core 15 is concerned almost entirely with the Executive DDT. DDT is discussed in the last volume under the programming system.

A. Communication Registers

In order for the Executive Program to maintain communication with each of the user programs, the lower 100_8 registers of each user core bank are reserved as communication registers. The first 40 of these (since they dictate system performance) can be modified only by the Executive Routine. They must, therefore, be written and read by the user program only through the execution of an IOT which in turn is handled by the Executive Program. Table IV-A-1 gives a list of the lower 100_8 registers with their functions. The registers from 40 to 77 can be changed either by the user or by the Executive Program, and proper usage of them is left to convention and diligence on the part of the programmer.

Communication with the Executive Program by a user program through the first 40_8 registers is accomplished by two IOT's. The mnemonic code for the IOT is given on the left, the next column is the actual octal code for the IOT and the remainder of the space is devoted to a brief programmer's description of the IOT's use.

RPP 723000 RPP (read a p-pointer) is an IOT whose execution permits the user programmer to read one of the first 40 registers in his core. The accumulator contains the register number of the p-pointer (1-37) and the IOT returns to the instruction immediately following the IOT with the register contents in the I-O.

WPP 72Ø32Ø Write a p-pointer (WPP) permits the user program to change the contents of one of the registers in the first 4Ø. The AC contains the number of the register and the I-O contains the new contents. The IOT has one return to the instruction immediately following the IOT. This IOT is intended for use only by the Start-Up Program for initializing the program number, name, etc., and should never be used by a user programmer. It is included here for reference purposes only.

In addition to modifying the p-pointers, the user program can communicate directly with the Executive Program by asking the Program to reserve for it a number from Ø through 7. This number is held by the Executive Routine until the user program either releases it or halts. This reservation scheme is useful in cases where several running copies of a single program may access the same pending file or common file area. By having the program reserve such a number before using the file and releasing it afterwards, each copy of the program is assured that no one will be changing the data in that file while he is using it.

HOLDPF 72252Ø The accumulator contains the number from Ø to 7 which is to be reserved. HOLDPF has two returns. The first, or bad

Word	Contains	Used by
Ø	lowest queue to which program can fall	swapper
1	accumulator	
2	program counter	
3	in-out register	
4	program flags, DDT indicators	Teletype service
5	running segment number	DDT
6	calling Teletype's scanner line	
7	number of milliseconds user has left this run	swapper after a user has been interrupted
10-13	DDT's pointers to its segments	
14-23	temporary storage	Core 16
24	drum pointer to programmer's index	programming system
25	program number	
26-30	program name	
31	typist's initials	
32	starting date	
33	starting time	
34-35	unused	
36	number of characters handled	
37	number of Teletypes held	
40-41	unit number of patient of interest	APR IOT's
42	bedspace code of patient of interest	APR IOT's
43	upper limit of item buffer	reading item IOT
44	upper limit of user's Teletype input buffer	Teletype input IOT's
45	type of error encountered when an IOT fails	
46	unused	
47-50	unit number of second patient of interest	patient-transfer IOT
51	bedspace of second patient of interest	patient-transfer IOT
52-57	unused	
60-62	temporary storage	DDT
63-64	unused	
65	scanner line number of Teletype currently being used	
66	name number of Teletype currently being used	
67-72	AC, PC, IO and the instruction itself when an illegal instruction is executed	
73-75	AC, PC, IO when core 16 is entered	
76	unused	
77	executed when "← -" is typed to JBH or HLTBAD is executed	

TABLE IV-A-1 LOWER CORE REGISTERS

return is to the location immediately following the IOT and indicates that the requested number is already being held by some other program. The second return, to the second location following the IOT indicates successful completion of the reservation.

RELSPF 722521 RELSPF is executed with the accumulator containing a number from Ø through 7 and releases that number. If the RELSPF IOT is not used, the release will nonetheless be accomplished when the program halts. RELSPF has two returns. The location immediately following the IOT means you tried to release a number you didn't own, and R2 means a successful release.

Two other system commands which represent communications directly with the Executive are the two halt commands used by user programs.

HLTGUD 72544Ø The good or normal halt command is used to indicate successful completion of the user program and is the normal terminator used for all user programs. Upon receipt of the IOT, the Executive Routine spaces the paper four lines, types "-THANK YOU-" spaces the paper up further and then proceeds to do the status restoration necessary to mark that program as halted.

HLTBAD 72546Ø A bad halt is used as a return from error conditions with which the user program cannot cope. For example, if information is lost on transfer to the data channel or there is a non-recoverable data channel error the bad return from that IOT might simply be "HLTBAD". This IOT spaces the page up, types -X- and executes a jump to location 77 of user core. This location may consist simply of a HLT command which is trapped by Executive and causes normal halting or may be a jump to some error recovery routine in the user program for such purposes as unfiling information already filed, typing out additional messages through the terminal, or initiating such other further action as may be necessitated by the nature of the program.

In addition to direct communication with the Executive Program, user programs are capable of communicating with Executive Surrogate Programs such as Type-Out Text and Event Detector which are described in the next chapter. Communication with these programs takes place by means of the list structured buffered area in Core 16. Two IOT's are provided, one for adding a pair of words to this list and a second for retrieving a pair from the list.

ADDLP	722Ø4Ø	Add List Pair is executed with the first word in the AC and the second word of the pair in the I-O. ADDLP has only one return.
GETLP	72226Ø	Get List Pairs is an IOT executed by the Surrogate Programs Type-Out Text and Event Detector only. It is used to withdraw and expunge instruction pairs from the list structured buffer area and is included here for reference purposes only.

Having considered communication between user program and the Executive Program, we may now consider communication between user program and an outside terminal via the Executive Program and the Terminal Service Routine already discussed.

B. Terminal Communication

Under ordinary operating conditions, the Terminal Service Routine assumes that the 8-bit ASCII codes that it receives represent ordinary text. In handling this text, for reasons of efficiency, it tries to assemble complete messages before calling the program requiring the information into active status. In this mode of operation, it recognizes only the EOM, the rubout character and the break as message terminators. It can ordinarily be counted on to take no action on any of the text strings that it receives until such a message terminator is received.

For certain programs, primarily DDT and TALK, the program cannot conveniently wait for an EOM to be typed but must respond much more immediately. This type of communication, where the terminal is being used in a very intimate control activity is called Control Mode and defines a large number of characters to be control characters. These include carriage return, line feed, space, most of the punctuation characters, EOM, and all of the characters typed with the "CONTROL" key depressed.

Whether in normal mode or in control mode, the Service Routine converts the 8-bit incoming ASCII code groups to 6-bit or 12-bit internal code characters via the character dispatch table. For many uses, such as analogue-digital conversion, coordinate recording and the use of some densely-coded input terminals, we would like to preserve the 8-bit character of the codes. If a terminal is operating in so-called 8-bit mode, the Executive Routine does not use the Terminal Dispatch Table but converts each 8-bit code into a pair of 6-bit codes. The high-order 4 bits of each represents the high-order and low-order 4-bit segments of the original 8-bit code. Various IOT's are provided to permit a user program to convert from one mode to another.

ECM 72Ø22Ø ECM is the IOT used by a user program when it desires to enter control mode. Program Flag 5 of the Terminal Routine is set and the Terminal Service Routine now uses the extended set of message terminators.

LCM	72Ø26Ø	Leave Control Mode zeroes Program Flag 5 in the Terminal Service Routine and restores the terminal to normal mode in which only EOM, rubout and break are recognized as terminators.
ICM	7ØØ2ØØ	ICM is an IOT used by a user program in interrogating the Executive Routine to determine whether the terminal to which it is connected is in control mode. The IOT has one return. The AC equals zero if the terminal is not in control mode and equals -Ø (all 1's) if the terminal is in control mode.
EEBM	72234Ø	This IOT, enter 8-bit mode, causes the Executive Program to store all of the incoming 8-bit codes as pairs of 6-bit characters. The first character represents the high-order 4 bits and the second character represents the low-order 4 bits (each in its high-order 4 bits). EEBM continues until a character consisting of all 1's is received at which time that character serves as a message terminator and causes Program Flag 2 of the Terminal Service Routine to be set back to Ø. Program Flag 2 is the 8-bit mode flag and the terminal is in 8-bit mode whenever Program Flag 2 is set equal to 1.

LEBM 7225ØØ This IOT, Leave 8-bit Mode, resets Program Flag 2 of the Terminal Service Routine and returns the terminal to normal mode.

All of the above IOT's have single returns.

In order for a user program to type out a message on a remote terminal, it may make use of several system IOT's. These IOT's permit it to control the terminal and to utilize the power of Exec for moderately complex message generation.

GTY 72232Ø This IOT is used by a user program to get a terminal. Register Number 65 contains the number of the terminal allotted. The Executive Program will check the status of the terminal whose number is in Register 65. If that terminal is off, GTY will send out a string of nulls to activate the motor turn-on relay and will return to the second instruction following the IOT.

If the terminal was already in use, GTY will cause the program to be set to inactive until such time as that terminal is free. Note that GTY needs to be used with some caution since it can put a program into a suspended state pending the release of the terminal to which it referred.

GTY has two returns, returning to the word following the IOT if the line to the terminal called is open.

RTY 722240 This IOT is used to release the terminal whose number appears in Register 65.

GTYS 722322 GTYS is a special version of GTY which is used to get a terminal for the Type-Out Text Program discussed in the next chapter. Because of the nature of the program, the program must not be rendered inactive if the terminal is occupied so GTYS has two returns, returning to the register after the IOT if the terminal is occupied and returning to the second register after the IOT if the terminal has been secured.

TTCKS 722140 This Check Status IOT is used by a program to determine whether any abnormalities have shown up in the functioning of the terminal. It returns to the location after the IOT with the inclusive or of the terminal error conditions contained in word 45 of core as follows:

Bit 0 Terminal not yours
1 TIS Max Exceeded
2 TOS ran over end of core
7 14 Echo Checks received

	bit 8	Interrupted by NULL (Signal key depressed)
	1Ø	EOT echo check received
	15	Terminal is off
	17	Line is open
TTH	722ØØØ	This Terminal Hang IOT is used for Type-Out Text only. Execution of the IOT sets the program to inactive status until one of the buffers of any active terminal which is owned by Type-Out Text is within three characters of being empty. This IOT is used for monitoring multiple buffers and is restricted to Type-Out Text since completion of the IOT by one of the buffers becoming nearly empty reactivates the program located on drum-field one which is exclusively reserved for Type-Out Text. The IOT is included here for reference only.

In the following IOT's we shall be using the term "Type-Out" quite frequently. While this term is proper when the terminal is a typing terminal such as a Teletype, it is not proper when the terminal is some other form of display. Nonetheless, the phrase serves as a simple shorthand for the actual process. In the actual process, a 6-bit or 12-bit character, depending on its nature, is converted to an 8-bit ASCII code and transmitted out along the line preceded by a suitable start (space) pulse and terminated by a suitable stop (mark) pulse. These 8-bit code bursts

are assumed to be sent asynchronously between bursts but synchronously within the burst itself. These restrictions impose no significant loss of generality on the method of display despite the use of the simplified term "Type-Out".

TYO 72ØØ6Ø This IOT is called with a 6-bit code in bits Ø through 5 of the AC. It types out the character represented by that internal code on the line whose number is contained in Register 65. The contents of the accumulator are unchanged. TYO has two returns, returning to the instruction immediately following the IOT if the attempt was unsuccessful and to the next location if the attempt was successful. As is the case with all type-out commands, the reason for the unsuccessful return can be found in the contents of the terminal error word (word 45 of core) whose contents are as described under TTCKS.

TOS 72216Ø This Type-Out a String command is the most common output command used in the system. The AC contains a 15-bit byte pointer to the beginning of the text string. TOS will continue typing out characters until it encounters an EOM terminator or until the pointer exceeds the top of core. In the first case it returns to the location second following

the IOT with the contents of the accumulator pointing to the terminator. If the pointer runs past the end of core, a first or error return is generated with the error cause being shown by word 45 of core.

TOSS 722Ø2Ø This IOT is the same as TOS except that it is a version used by the Type-Out Text Program. If the buffer associated with the terminal whose number is in Register 65 fills as a result of the TOSS command, a valid (second) return is made as usual but with the contents of the I-O set to -Ø. This IOT therefore permits the servicing of multiple terminal buffers without undue suspense of the program. Since it does not involve a reactivation of the program on drumfield 1, it is not reserved for use by Type-Out Text.

TAS 72254Ø The Type A String IOT functions like TOS but with an arbitrary 6-bit terminator which may be stored in the I-O in bits Ø through 5 at the time of the execution of the IOT. If a valid return is made to the second location following the IOT, the contents of the AC are a 15-bit pointer to the terminating character.

The third set of IOT's that deals with Terminal Communication, is the input editing and storage set. These commands utilize the routines of the Executive Program to interrogate an input Terminal, to provide certain system editing functions to that terminal, and to provide for the orderly storage of the information received. In order to understand the input routines more fully, we must be familiar with two special registers as follows:

TISMAX ØØØØ44 Since string-inputting commands are permitted, some upper bound must be specified for the buffer area which may be occupied by such a string inputting IOT. This upper bound protects the user program against accidental obliteration. It further generates an error return (return 1) if any input command attempts to exceed that location. In the event of such an error, word 45 returns with bit 1 set. The value of TISMAX is set by the user programmer by initializing the contents of Register 44.

CRF ØØØ363 CRF is a special register used by the Edit Routine as a flag. If CRF is equal to a Ø then a carriage return--line feed combination is changed to a space prior to storage. If the contents of CRF is other than Ø, carriage returns and line feeds are stored within the text unchanged. This command permits the storage of text in a format-free form.

TYI 72Ø1ØØ This IOT transfers the next 6-bit character from the Executive Terminal Service Routine buffer to the high-order 6-bits of the accumulator and clears the low-order 12 bits. The IOT has two returns with the second return being the successful one and the first return being accompanied by the error diagnostic in word 45.

TIS 72222Ø The Type In String IOT is analogous to the type-out string IOT previously discussed and is the form generally used for inputting text information into the system. The IOT is called with a 15-bit character pointer in the AC indicating the location where the first character is to be stored and with an address in TISMAX (word 44) indicating the end of the available input buffer. The characters are transferred until TISMAX is exceeded or until an EOM terminator is encountered. If TISMAX is exceeded, an error return with bit 1 set in word 45 is made to the location immediately following the IOT whereas a proper return to the second location following the IOT has a 15-bit pointer in the accumulator pointing to the EOM terminator.

EDIT 72416Ø The EDIT IOT is called with a 15-bit character pointer in the AC and converts a "raw" text string to a "finished" text string. The EDIT IOT is commonly called after a TIS and provides the system functions of backslash and rubout. For each backslash that EDIT encounters in the text string, it deletes the immediately preceding character and that backslash. Thus for seven backslashes it will delete the preceding seven characters and the seven backslashes, closing the gap in between.

When it encounters a rubout, EDIT deletes all of the characters in that string preceding that rubout and the rubout character, again closing the gap.

EDIT also replaces carriage returns and line feeds with spaces as described under CRF. There is no limit to the number of backslashes or rubouts that may be used within a single text string.

TYIHNG 7222ØØ This IOT is a "hang" command used when word 45 returns with bit 15 set. This bit indicates that the typist has, for some reason, turned off the terminal. If the programmer wishes the program to wait until the terminal is again turned

on, without wasting valuable Central Processor time by entering a delay loop, he may cause it to execute a TYIHNG command. The program is set to inactive status until a break is received, indicating that the signal key has been depressed to restart the terminal. This command permits the user, such as a nurse, to turn off the terminal during a long output program such as a listing without losing valuable data when the terminal is restarted.

The preceding IOT's represent communication with user terminals while the ones before that represented communication with the Executive Program itself. A third form of communication is provided in the system to permit the user program to communicate both with the operator at the remote computer and with the real world. Both of these types of communication are considered central, in the system sense, and are treated in the next section.

C. Central Communication

The three primary central communication facilities have to do with the ability of a user program to read a set of clocks. Two are real time clocks and the third is a manually established set of registers.

RCK 7223ØØ This IOT is used to read the internally contained millisecond clock. This clock is a 16-bit register indexed once each millisecond for a total count of 6Ø,ØØØ or one minute. Since a user program can be interrupted between an observation and the execution of this command, RCK is not useful for the control of or measurement of outside activities by a user program. It is included to facilitate the collection of statistical timing data and for debugging purposes.

GTD 722Ø6Ø The Get Time and Date IOT reads an internal clock and calendar and returns with the contents of the AC equal to the number of days since December 31, 1848 and with the contents of the I-O equal to the number of minutes since midnight at the time of the execution of the IOT. The IOT has only one return to the next instruction and must, as with all real-time IOT's be used cautiously because of swapping times and other unscheduable program manipulations.

GDTIME 72236Ø Execution of the GDTIME IOT returns with a date in the AC and a time in the I-O in the same form as GTD. The date and time indicated, however, are the next scheduled shutdown time for the remote

computer center. Execution of this IOT can be performed both as a protective measure and by a program specifically designed to warn users of impending interruption of a service.

In addition to communicating with the central clocks, the user programmer has the ability to communicate with the central paper tape punch facility of the remote computer center. While it might seem more reasonable to include paper tape IOT's along with the magnetic tape IOT's, the fundamental nature of the paper tape loading process in the present computer suggests its inclusion at this point rather than with the I-O Processor commands.

CLRDR 72Ø24Ø This IOT has one return and its execution serves to clear the reader buffer of any code that was previously entered. If a read command were given without being preceded by a CLRDR, the first character returned would be likely to be garbage.

RPA 72ØØ2Ø Read Paper Tape, Alphabetic, is an IOT with one return that interprets one line of punches from the paper tape as a single 8-bit character and deposits it in bits 1Ø through 17 of the I-O.

PPA 72Ø12Ø Punch Paper Tape, Alphabetic is an IOT which takes the contents of bits 1Ø through 17 of the I-O and punches them as a single 8-bit line on the 8 channel tape punch.

RPB 725520 Read Paper Tape, Binary is an IOT that reads three lines of punches on a paper tape (ignoring channel seven) and skipping any rows for which channel eight is not punched. It assembles the three sets of 6-bit codes thus read into an 18-bit word and deposits that word in the I-O.

ERIM 725540 Enter Read In Mode is an IOT which simulates the act of depressing the read-in switch on the computer console when the console is out of Time-Shared Mode. This IOT is for the specific purpose of loading paper tapes that have been pre-punched with a loader segment and is used only by programmers during the early stages of adapting a non-Time-Sharing Program to the Time-Sharing Mode.

The above IOT's essentially complete the list of IOT's used for communication purposes. We are now ready to examine how a user program utilizes the common routines of the Executive Program to secure information from the outside world, validate it, code it, store it and file it in the file structure of the system.

D. Job Hunter

One of the first tasks to be solved in a real-time communication system is the gathering of data by the computer from a human operator located at an input terminal. Job Hunter is a program, or more specifically, a combined programming language and program, that is designed to facilitate this automated information entry.

Anyone who has worked with printed forms, whether for use with a digital computer or with a manual filing system, has already participated in an effort to automate the data entry process. The printed form has the ability to define both the location and the format of the individual fields of information which go to make up the record. The printed form, while serving as a convenient starting point for a discussion of automated information entry has obvious problems when one wishes to deal with a record that is very much more complex than a simple admission form or an application for insurance. One general inadequacy of printed forms for more complex records may be inferred from their use as check lists. Such check lists frequently have the undesirable property of requiring one to read through many non-applicable questions before discovering the few that he should really answer.

A more subtle, but deadlier danger in the use of printed forms is the high degree of structure that they may impose on the information gathering process. The economic and social costs involved in altering the form or eliminating it tend toward preserving the status quo. One must also

recognize that, while the printed form may specify rules for location and format of entries, its passive nature precludes it from any form of effective intervention should such rules be violated.

At the other end of the spectrum of aids to information entry is the skilled interviewer who elicits information from us, jogs our memory, probes to uncover internal inconsistencies, observes the many forms of non-verbal communication which we use in describing a situation and records this information in the format of some complex model whose structure he varies from interviewee to interviewee.

The first, the printed form, yields a well-organized rigid store of information that can be routinely searched by file clerks but which is sorely circumscribed in the set of questions it can answer. The second data store, that generated by the skilled interviewer, is less well organized and can be searched only by one skilled in interpreting the models according to which the information was stored. Because of the richness of the interactions it describes, however, and the depth of the information it contains, the second data store can answer far deeper and more convoluted questions than the first. The printed form storage system will consequently allow highly precise retrieval while the second store will provide less precision but, in many cases, greater accuracy.

The interviewer's technique, in a reasonably simple case, involves probing questions to define areas of concern and penetration in depth within those areas. In addition, one area will frequently lead to another by implication.

This technique of initial probing and subsequent follow on in depth saves a considerable amount of time. Just as a human interviewer will not ask a recent high-school graduate detailed questions about his undergraduate and graduate university training nor a pediatric patient about his marital status, so must Job Hunter facilitate the entry of information by threading its way through the maze of all of the possible questions to be asked in a manner that depends heavily on the information gathered by all of the previous questions.

Clearly, any system requiring information from a human operator cannot at the same time require infallibility. Thus, Job Hunter must make it possible for the human respondent to make corrections to his responses at any time with a minimum duplication of effort.

When a response has been corrected, Job Hunter must, therefore, re-assess its path through the maze; asking those new questions which have become pertinent and refraining from reasking those which have already been answered. Figure IV-D-1 shows a diagram of a hypothetical question-answer complex and the conditional branching pattern implied by it. Figure IV-D-2 which is a hypothetical

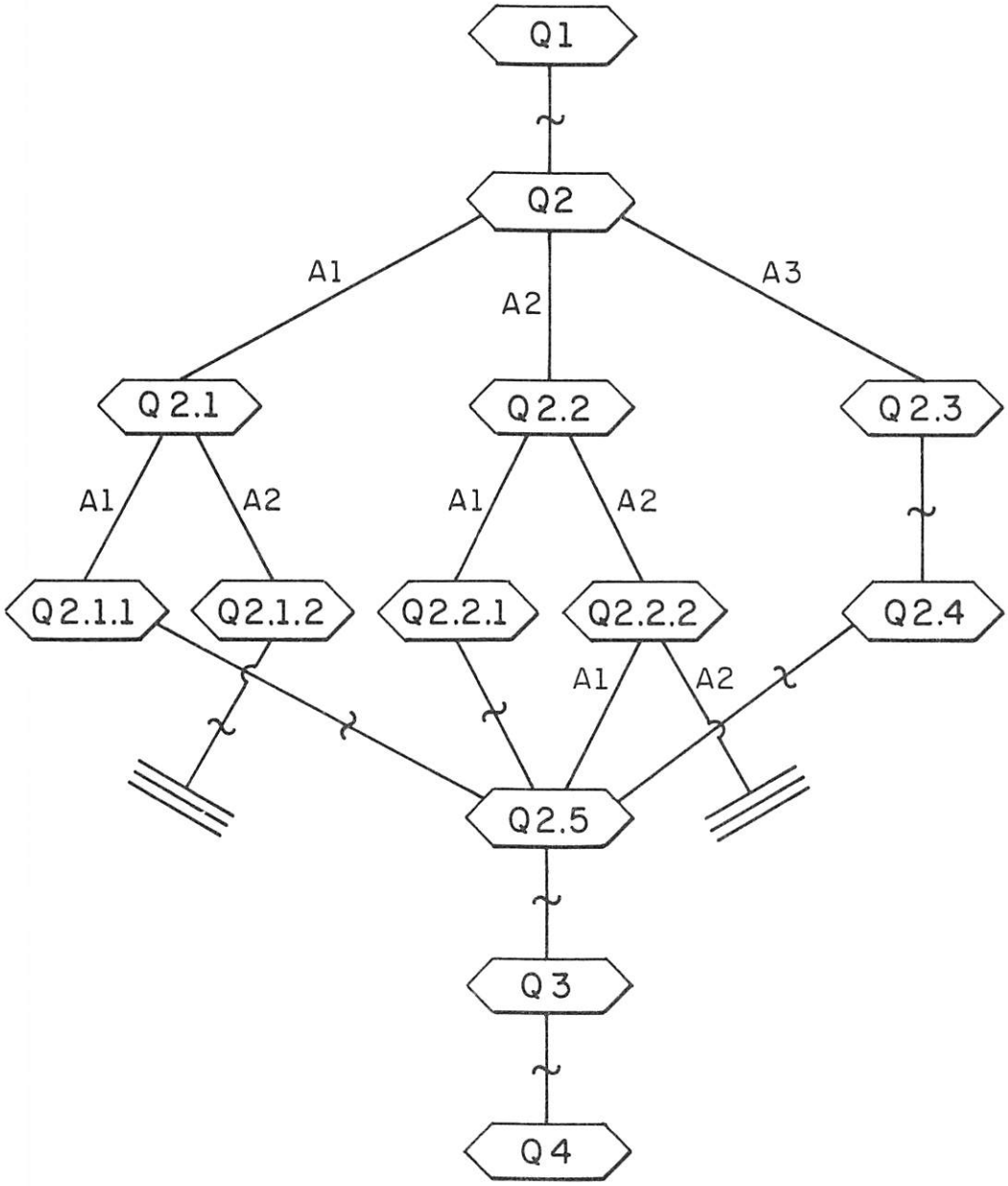


FIG. IV-D-1 EXAMPLE OF A BRANCHING Q-A PROGRAM

```
Q1...A
Q2...A3
    Q2.3...A
    Q2.4...A
    Q2.5...A
Q3...A
Q4...←2
Q2...A3     A1
    Q2.1...A1
            Q2.1.1...A
Q4...←LIST

Q1...A
Q2...A1
    Q2.1...A1
            Q2.1.1...A
    Q2.5...A
Q3...A
Q4...
```

FIG. IV-D-2 HYPOTHETICAL QUESTION - ANSWER SHEET

typescript corresponding to the question-answer tree indicates how Job Hunter operates on a map such as that shown in Figure IV-D-1 in order to produce a conditional questionnaire complete with question features.

It is important to recognize that there is no fundamental restriction on the complexity of the branching pattern shown in Figure IV-D-1. The present version of Job Hunter, however, does not permit internal looping within the question-answer framework.

In addition to facilitating the entry of information by the respondent, we felt it necessary for Job Hunter to participate in some small measure in the training of the respondent and to contribute to the respondent's feeling of confidence. Toward this end, we have provided the respondent with a rather simple meta-language so that he may deal with the form of the questionnaire as well as with its substance. Job Hunter uses a horizontal arrow typed by the respondent as a control symbol. It indicates to the program that we are departing from the language of the questionnaire and are using the meta-language to communicate with the machine about the questionnaire itself.

If we look at Figure IV-D-2 on the seventh line, we see that the respondent has typed a horizontal arrow followed by "2." Because the horizontal arrow was typed, the 2 was not taken as an answer to question 4 but rather as a request to repeat question 2 which the machine does on the following line.

With that much introduction, we can examine the external operating characteristics of the Job Hunter Program itself.

1. External Features of Job Hunter

Job Hunter is basically a program through which the user programmer dictates the form of an automated data entry session. Job Hunter handles the book-keeping, conditional branching, and some meta-linguistic communication.

Questions are typed out as follows:

a. First, a carriage return--line feed is typed so that each question starts on a new line.

b. Second, spaces are typed to indent the question according to the subscripting level (as is done in outline format).

c. Third, the question number, e.g., "1.3.3" is typed followed by two spaces.

d. Fourth, the text of the question is typed.

e. Fifth, two spaces are typed to indicate that the machine is awaiting a response from the respondent.

At this point, the respondent may either enter the answer to the question or may, by typing a horizontal arrow, communicate with the computer about the form itself. To accomplish this communication, the respondent uses a set of special characters, each of which must be followed by an EOM (depressing the enter button). These special characters and their meaning are as follows:

Typing a minus sign (-) in the meta-language indicates to Job Hunter that the user wishes to abort the program. Job Hunter executes a HLTBAD IOT as explained earlier.

The use of a slash (/) indicates that the respondent wishes to change the mode of the question text. Each question has two forms associated with it; generally a short form for the respondent who is very familiar with the program and a somewhat longer, more detailed form of question for the respondent who is just learning the program. At any time, the respondent may switch from one form to the other by typing a slash in the meta-language. (See JBF in the program explanation.)

HOW Typing HOW in the meta-language causes Job Hunter to type out the example or explanation that it has stored for that particular question. It then types two spaces and again waits for an answer.

n If the user types a question number, including any decimal points, in the meta-language Job Hunter will type back that question and the previously entered answer, followed by two spaces, at which time it will accept either a meta-language "OK" or a new answer.

OK This meta-language entry is only valid at the time a re-asking of a question has been ordered. It causes no modification of the previous answer.

Cn A "C" followed by a valid question number in the meta-language causes Job Hunter to copy the previous answer to question n as the answer to the present question. The answer is typed out so that it may be verified by the typist.

Ln An "L" followed by a question number causes Job Hunter to type out the question n and its previous answer and then revert back to the question it had been asking without waiting for modification. This feature is generally used only on very long forms or in correction routines where the user merely wishes to look at the answer to a previous question without changing it.

In all of the preceding constructions, if n is not a valid question or is one later in the sequence, Job Hunter will inform the respondent of that invalid request without causing further difficulty.

Another convenient meta-linguistic command is:

LIST The typing of LIST in the meta-language causes Job Hunter to space the page up and type out a clean copy of the entire questionnaire as though all corrections had been properly made on the initial pass through. If any questions had been made irrelevant by a change in answer, these irrelevant questions are not repeated nor are any of the meta-linguistic interchanges. The listing terminates with Job Hunter awaiting a new answer in the same place it was when the meta-linguistic transfer was made.

Having looked at the external behavior of Job Hunter we can now examine its internal workings.

2. Job Hunter, Internal

Job Hunter is an Executive Routine resident in Core 16 which is used by the user programmer through the use of a set of IOT's. Job Hunter not only uses Core 16 but uses a series of registers in user core extending from Register 364_8 through Register 417_8. These registers may be manipulated either by Job Hunter or by the user and indeed are used for the two-way communication.

In addition to the communication registers, Job Hunter builds two tables within user core. Indeed, as far as the user program is concerned, it is the construction of these tables that is Job Hunter's raison d'etre. Job Hunter builds a table of question-answers from location T in core upward until it runs out of space. It builds another table, primarily of pre-asked questions and other internal housekeeping details downward from location X in core. The locations T and X are specified through the use of one of the IOT's used to communicate with Job Hunter.

In general, answers coming into Job Hunter are checked out against a Syntax Verifier Program although that step is certainly not required. The interface between the two programs will become clear upon reading the section on Syntax Verifier. As each question is answered, the text answer is stored away in the buffer starting at T and may then be manipulated as desired by the user programmer. Job Hunter is a Core 16 routine which permits the calling of any other Core 16 routines without causing overlap in the temporary storage areas involved.

The general internal function of Job Hunter is such that one may pass through the Job Hunter coding in one of four modes: simulation mode, normal mode, error correcting mode or listing mode.

In the simulation mode, the program is initialized and run through with the exception that no type-out is performed on the terminal. This mode is used in order to follow through the question map after a correction has taken place to see if any changes must be made. A simulation always starts with the most recent "INIT" IOT in the user program. If, in a simulation, the program comes to a question which has not been previously asked, Flag 1 is set to a ∅ which permits the type-out of the question and the receipt of an answer. As soon as the simulation chain again leads to a previously answered question, Flag 1 is set to a 1 and type-out is again suppressed.

In normal mode, everything is typed.

In error correction mode, a simulation takes place from the INIT to the question to be corrected. At this question Flag 1 is turned off, the question and answer are typed out, "→OK" is set as a valid entry and after an answer the simulation proceeds. This technique, of course, permits the immediate detection of any unasked questions required by the change in the answer to the corrected question.

In listing mode, the system behaves as though it were simulating starting with the INIT but suppressing no type-outs, typing out both the question and the answers and returning to normal mode upon encountering an unasked question. This question should, of course,

be the question at which the →LIST command was issued. Mode control within Job Hunter is maintained by an internal register.

JLIST ØØØ4ØØ JLIST is a Job Hunter control register maintained in user core. If it contains a negative number Job Hunter is in simulation mode. If JLIST contains a Ø, Job Hunter is in normal mode; a 1 means error-correction mode; and a 2 means listing mode.

Several other useful registers which the user programmer should know about are as follows:

JBF ØØØ373 This is a register maintained by Job Hunter as a flag. When it contains a 1, the first mode of the question text will be typed out on asking a question, while when it contains a 2, the second mode will be typed out.

JANS ØØØ364 JANS is a register containing a character pointer to the beginning of the current answer in the user's buffer area.

JQN ØØØ376 JQN and the register immediately following it contain six 6-bit fields each interpreted in binary format. These fields give the current question

number. Subscripts that do not exist are carried as zeroes. The high-order 6 bits are the first number, the next 6 bits are the first subscript, etc. As a result, the highest question number that can be typed by the system is:

63.63.63.63.63.63

providing six levels of subscripting with up to 63_{10} entries in each.

JNTP ØØØ371 JNTP contains a pointer to the next available space in the user's text buffer. It is the location where the next answer will be written by Job Hunter in the normal course of events.

JVAN ØØØ415 JVAN contains a pointer to the start of the text string entered in response to the last validating question (see the IOT TOVQ). We shall discuss validating questions and their application separately however.

JBCQ ØØØ374 JBCQ and the register immediately following it contain six 6-bit fields in the same format used in JBQN. These 6 fields contain the number of the question to be corrected as Job Hunter is dropping through a simulation in correction mode.

With this brief introduction to the internal registers used for communication and housekeeping, we can examine the IOT's used by the user programmer in calling for the features of Job Hunter.

INIT 72466Ø INIT is an IOT called with two arguments in the form:

INIT T,X

T is the lower bound specified by the user programmer for Job Hunter's text buffer area and X is the upper bound. The user programmer may not use any of the registers between T and X since Job Hunter builds inward from both ends.

This IOT initializes the Job Hunter system and signifies to Job Hunter the start of the Job Hunter segment of the user program.

TYOC 72476Ø TYOC is an IOT used in typing out a comment during a Job Hunter Program. It does not wait for an answer and its type-out is suppressed if Flag 1 is set. The Job Hunter system is not affected by the type-out of a comment and the only reason for the existence of the IOT at all rather than the use of a TOS is to suppress the type-out during simulation. TYOC is called with

one argument as follows:

TYOC Y

Where Y is a character pointer to the start of a comment terminated by an EOM.

TYOQ 725000 TYOQ is called with one argument as:

TYOQ Y

Where Y is a character pointer to three consequtive text strings each terminated by an EOM. The first text string is the form of a question to be used on initially running through the program, the second text string is the string to be used after an odd number of slashes have been typed in the meta-language and the third text string is the string to be typed out in response to a meta-linguistic HOW.

Execution of the TYOQ generates a new question number by incrementing the presently pertinent subscript level. Thus if 3.2 were the last question asked executing a TYOQ would generate question number 3.3. If the question so generated has previously been asked, the program checks register JLIST. If, as a result, the question is to be simulated, Flag 1 is turned on and

all output is suspended. If the question is to be asked, Flag 1 is cleared and the question is asked as previously described.

For timing purposes, it should be noted that TYOQ does not wait for an answer. The answer must be provided by one of the type-in IOT's.

TYIN 725020 TYIN accepts a typed in answer and puts it in the user buffer starting at the location pointed to by JNTB. The IOT is terminated by receipt of an EOM symbol and leaves the pointer JANS pointing to the answer received. Type in is, of course, also suppressed under control of Flag 1.

TYIV 725040 TYIV is the input command used to serve as an interface between Job Hunter and Syntax Verifier. It is called with one argument as follows:

```
TYIV Y
```

Where Y is a pointer to the Syntax definition located at Y. The nature of the Syntax definition is discussed in the next section under the Syntax Verifier Program. The IOT is terminated by the receipt of a text

string terminated by an EOM that meets the requirements of the Syntax Verification definition located at Y.

If the typed-in string does not pass the verification, Job Hunter will type out "FIX", reask the previous question and again wait for an answer under the same TYIV.

In addition to the previous IOT's for asking for and receiving information, the structure of the path through the tree must be specified to Job Hunter. Two instructions are necessary for this purpose. One of these causes the system to drop down to the next lower level of subscripting (logically implying a further, in-depth, pursuit of the subject). The second involves a shift upward in subscript level to indicate logically a return to a plane of increased generality.

DNLV 724700 The downlevel command sets the level to be incremented by the next TYOQ one position to the right. Thus if question 3.3 were followed by a TYOQ the next question would be 3.4. If, however, a DNLV were inserted between question 3.3 and the next TYOQ, the next question would be 3.3.1. DNLV has only one return since it assumes

that the user programmer will not down level past the lowest value. If the programmer does go below 6 levels of subscripting, the user program will fail as a result of trying to execute an illegal IOT.

UPLV 7247ØØ Up Level causes the current subscript level to be cleared to Ø and the subscript pointer moved one position to the left. In the case given above, if an UPLV were interposed between question 3.3 and the next TYOQ, the next question asked would be numbered simply 4. Any effort to up level beyond the zero subscript level is again treated as an illegal IOT.

Note that since INIT sets the subscript value initially at zero, then logically every UPLV command in the system must be preceded by a DNLV command. In a simulation, if all of the questions between a DNLV and its following UPLV were simulated then it can be assumed that no new information entered the system so that the simulation can continue. As a result, Flag 1 remains set between the UPLV and the

TYOQ immediately following it. If, on the other hand, any instruction between the DNLV and the UPLV was actually asked, Flag 1 is turned off between the UPLV and the next TYOQ.

XDNLV 72474Ø This IOT is included to permit asking a subscripted question without asking the header question above it. It increments the question number at the previous level before dropping to the next level of subscripting. Thus in the example given before if an XDNLV were included between question 3.3 and the next TYOQ, the question typed out would be 3.4.1. Note that in this case question 3.4 is never in fact asked.

While Job Hunter is intended for use with Syntax Verifier, it can also (see Chapter VI on Formulary Validation) respond to semantic invalidities. To do so, Job Hunter contains two commands which permit it to duplicate the effect of a TYIV failure.

FAIL 725Ø6Ø FAIL is an IOT that causes the type out of "FIX" and a re-execution of the preceding TYOQ. The command actually causes a complete restart doing a complete simulation through until the preceding TYOQ. As a result, any core

changes introduced by the preceding syntactically correct but semantically incorrect answer are wiped out.

REASK 7251ØØ REASK is identical to FAIL with the exception that the type out of "FIX" is suppressed. It is frequently used when the user programmer wishes to type out a more descriptive diagnostic through the use of a TYOC.

Notice that a great deal of effort has been expended in insuring that core changes produced by erroneous answers are continually rectified by changes and corrections put in from the terminal. Job Hunter does not, however, have any facilities for undoing any changes that the program may have wrought in bulk storage or through other irreversible commands. All such commands should follow the last input command to Job Hunter (either a TYIN or TYIV) and should, in turn, be followed by a new INIT. If this precaution is not observed, then the irreversible changes will be repeated or the bulk memory filing will be repeated every time a restart, a simulation, a FAIL or a REASK is executed.

In many of the hospital activities, the form of the input desired for communication to the computer is highly stylized and well defined through tradition. The input represents the information for several

fields and hence the answer to a group of questions. A typical example might be:

ASPIRIN, 0.6G, PO, QID, 2 DAYS

By tradition, this form of a prescription means that the patient is to receive the drug aspirin at a dosage of 0.6 grams by mouth four times a day for a period of two days.

Note that five separate fields have been specified, delimited only by commas. The name of the drug, its dose, its route of administration, the frequency with which it is to be given, and the duration of the prescription.

As will be seen in the next section, Syntax Verifier is capable of separating out the five individual fields and leaving pertinent pointers to them despite the fact that they are initially all submitted as an answer to one question. The human engineering problems involved in the error correction task, however, comes up at this point. If, for example, the respondent types in 0.6H instead of 0.6G, Syntax Verifier can and will reject this response because it is an illegitimate dose statement. If we used the IOT TYIV for accepting this prescription, an error such as 0.6H would cause the type-out of the word FIX and would require that the entire prescription be re-entered.

While this technique is simple, it does not really meet the needs of the hospital and we would like some technique for validating a discrete portion of a complex response and then reassembling a proper response from the individual discrete validations. To this end, Job Hunter contains the facilities for asking and accepting such validating questions. When combined with the use of TYOC to type-out guiding diagnostic comments, they can form one of the most powerful aids and teaching facilities of Job Hunter. Examples of its use in the hospital will be found in Memorandum 9 under the Medication Order Program.

TOVQ 725260 TOVQ is an IOT called with one argument as follows:

TOVQ Y

and is used to type-out a question starting at Y which will not be numbered and will not appear during listing. An example of its use is as follows:

ASPIRIN, 0.6H, PO, QID, 2 DAYS

DOSE? 0.6G

A TOVQ was used to type out the question "DOSE?". Note that the response to the validating question, generally obtained through a TYIV must now be used with the previous answer to assemble a clean answer to the overall question.

As mentioned previously, the pointer to the validating answer is contained in Register JVAN. The user program must now copy the answer starting from the position pointed to by JANS into the free section of the buffer pointed to by JNTP until he gets to the previously invalid portion of the question. He will substitute for this the answer pointed to by JVAN and then continue transcribing the previous answer until he comes to the EOM. Note that he will now repeat his Syntax Verification, since the Syntax Verification presumably failed at the dose and hence the route, frequency and duration have still to be checked.

After the copying is completed, the programmer must see to it that JANS now points to the start of a fresh copy. In order to permit restarts and subsequent simulation without confusion, it is necessary that the question-answer table that Job Hunter has been building down from location X (the second argument in the INIT call) be modified to show the new reassembled question as being the answer to the proper question.

REPLAC 72522Ø REPLAC is called with two arguments as follows:

REPLAC A, B

Where A is the address of the previous TYOQ and B is a character pointer one position past the end of the newly copied text. This IOT sets the contents of JANS to the former contents of JNTP (which is where the copying was started) and sets the contents of JNTP to the value of B.

Although several replaces may be done consecutively, in order to facilitate repetitive validation of succeeding fields in a complex answer, and although each updates the question-answer table a programmed restart must be executed before executing the next TYOQ.

RESTAR 72524Ø RESTAR has the effect of passing through the program in simulation mode so that if the new answer just assembled implies the necessity of asking some new question, that eventuality will be taken care of in the normal course of events. If such is not the case, RESTAR merely drops through the program and produces no change. A RESTAR should always be executed between any replace and the next immediately following TYOQ.

KILL 725160 KILL is an IOT called with one argument as follows:

KILL Y

Where Y is the location of a TYOQ command in user core. The IOT marks that particular question as never having previously been asked. KILL must be followed by a RESTAR before the next TYOQ.

KLAST 725200 KLAST is an IOT used with one argument as:

KLAST Y

and is used to mark the question located at Y in the user core as undone on the most recent time it was asked. This type of KILL is used where a question is used repeatedly. It eliminates the most recent instance of that question. KLAST must, like KILL, be followed by a RESTAR before the next TYOQ.

The validating question responses do not recognize the meta-linguistic symbols. In order to permit the meta-linguistic hyphen to be used as a program terminator in a validating question, one additional IOT has been provided for Job Hunter.

JTIS 725560 JTIS is similar to an ordinary TIS except that it is used within Job Hunter and permits the meta-linguistic hyphen to execute a HLTBAD. JTIS is executed with a character pointer in the AC, it has only one return and the AC points to the EOM. All editing has been done. JTIS is useful instead of TYIV for the answers to validating questions since it permits all of the necessary TYIV features as well as program termination.

We have not included any detailed coding of a Job Hunter Program although it is fairly evident that Job Hunter is designed to work closely with Syntax Verifier, an interpretative program for producing syntactic verification of input strings. Syntax Verifier is described in the next section.

E. Syntax Verifier

Although an early version of the Syntax Verification Program was described in Memorandum 5, the introduction of an extensive library of IOT's, the reorganization of the Executive Program and certain important additional features warrants the inclusion of a detailed description of Syntax Verifier at this time.

Syntax Verifier is an interpretive program which operates on a set of pseudo-instructions occurring in user core. There are two basic IOT's used to call Syntax Verifier. One, TYIV, has been previously discussed under the Job Hunter Program. The other is STV.

STV 724020 The IOT STV is called with a character pointer in the AC pointing to the text to be verified and with a word pointer in the I-O pointing to the string of code that represents the pseudo-instructions forming the Syntax Verifier definition. STV has two returns, returning to the location immediately following the IOT if the text string failed the Syntax Verification and returning to the next location if the text string passed.

It is clear that the primary function of Syntax Verifier is to compare a specific string of text, terminated by an EOM symbol to a definition in order to determine whether the string meets a definition. Further features of Syntax Verifier, however, permit it to do simple encoding, string segmentation and other housekeeping operations attendant on input verification.

In order to understand the use of Syntactic Verification, we should understand a bit about Backus Normal Form representation. A basic feature of the Backus meta-language (called BNF for short) is the use of a small set of meta-linguistic symbols. These symbols are as follows:

< > ::= |

In order to be completely valid and to prevent internal paradoxes, the BNF symbols may not be used in the object language itself. The meaning of the symbols is simple. A string of characters enclosed in the brackets < > is called a meta-linguistic variable. A meta-linguistic variable defines a class of "things" in the object language.

The vertical bar serves as a conjunction in the meta-linguistic statements and means "or."

The mark ::= is the defining operator used in a statement. Let us, as an example, define a decimal number. First we define our alphabet by writing:

<DIG> ::=∅|1|2|3|4|5|6|7|8|9

This statement roughly translates as DIG (our meta-linguistic name or shorthand for a digit) is defined as a ∅ or a 1 or a 2 or a 3 or a 4 or a 5 or a 6 or a 7 or an 8 or a 9. Obviously <DIG> is the meta-linguistic variable or name we have given to the class of things that we ordinarily consider decimal digits.

BNF's particular strength lies in the ability to define variables recursively. Thus we may define a meta-linguistic variable <INT> which describes the class we normally consider to be decimal integers. The definition looks as follows:

<INT> ::= <DIG>|<INT> <DIG>

This definition states that an <INT> is defined as a <DIG> or an <INT> followed by a <DIG>. Similarly we may define our decimal number by the following:

<DECIMAL NUMBER> ::= <INT>|<INT>.|.<INT>|<INT>.<INT>

This last definition may be read as "A member of the class 'DECIMAL NUMBER' is defined as an <INT> or an <INT> followed by a period or a period followed by an <INT> or an <INT> followed by a period followed by an <INT>."

A BNF definition may be considered as an operational statement and hence becomes closely analogous to the test procedure used to validate an expression against such a definition. Note that a string needs only to satisfy the definition between any two "or" symbols or between the end of the expression and the last "or" symbol or between the first "or" symbol and the defining operator in order to pass the definition.

Just as BNF is a defining language which operates on characters in a text string, so our Syntax Verifier Executive Program operates on a character string. In order to group characters by functional class, however and to avoid the long involved insertional statements made in many character manipulation programs, Syntax Verifier classifies all of the system's available input characters in two main classes, puncuational and arithmetic and within those classes separates each class into 13 subgroups. All references to this classification system, or atom table as it is called, are made through a 15-bit numerical representation. The 1's present in the 15-bit representation represent the inclusive or of the classifications which they describe. Table IV-E-1 illustrates this atom table. The punctuational class is selected by setting bit 3 to a zero while if bit 3 is a 1 the entire 15-bit number represents the arithmetic

class. Note that one can form a 15-bit representation of the logical or of any groups within a class. Thus the number 4276ø refers to left parenthesis, right parenthesis, left bracket, right bracket and the two broken brackets which are the less than and greater than signs. We shall see shortly that this representation gives us a rapid type of shorthand for many of our definitions.

Having stated the basic purpose of the Syntax Verifier and its fundamental classification scheme, we can go on to consider the definitions themselves. It must be understood that while the definitions are not IOT's in the usual sense but are registers which are interpreted by a portion of the Executive core other than the basic dispatch table, we shall use the ordinary IOT notation for discussing them. Before reviewing the operation codes, however, let us, as in the case of Job Hunter, define some registers used internally for communication between Syntax Verifier and the user core.

FSA øøø12ø FSA is the register whose contents are a character pointer which points to the character in the text string currently under test.

OPB øøø323 OPB is the start of a 32-word buffer running from OPB to OPB+31. In this buffer, Syntax Verifier stores the various pointers and constants which are used in segmenting and encoding input strings.

ATOM TABLE CONFIGURATION
CLASS

Bit No.	Value	Punctuational	Arithmetic
17	1	consonant	asterisk
16	2	digit	slash
15	4	exclamation point	colon
14	1Ø	question mark	up-arrow
13	2Ø	space	left parenthesis
12	4Ø	hyphen	right parenthesis
11	1ØØ	plus sign	left bracket
1Ø	2ØØ	period	right bracket
9	4ØØ	vowel	less than
8	1ØØØ	apostrophe	equal sign
7	2ØØØ	comma	greater than
6	4ØØØ	semicolon	code "77"
5	1ØØØØ	quotation	special*
4	2ØØØØ	any arithmetic character	any punctuation character

*This bit is used for all remaining characters, in particular: #, $, %, &, ©

TABLE IV-E-1

SUG ØØØ424 SUG is a pointer which points to the next register in the 32-word OPB buffer available for entry.

SUS ØØØ411 SUS is a register which contains a 15-bit atom table definition. It is used in conjunction with certain overall string validation procedures to ignore those characters which are not important to the meaning being validated but are generally present for esthetic or format purposes in the input string.

These registers are essentially the only ones which the user programmer need refer to in the normal course of events although if he is interested in other registers such as those controlling the internal levels of the program and the push-down list, he is referred to the listing.

ILG ØXXXXX The Ø operation code is the first of the pseudo-instructions which are interpreted by Syntax Verifier. The five X's following it represent a 15-bit atom table definition as previously described. This operation code is analogous to an "or" string against which a single character is to be checked. This pseudo-instruction causes the next character of the input string to be looked up in the atom table. The bit found there is compared with the bit pattern in bits 4 through 17 of the ILG pseudo-instruction and the class bit

is checked against bit 3. If the class and the character bits match, the character is considered to have passed the test. If there is no match, the character has failed.

On passing, the test pointer FSA is incremented one character and the next pseudo-instruction in the program is interpreted. In case of failure, FSA is reset and Syntax Verifier enters the failure evaluation routine.

The ∅ Op. code instruction checks one and only one character in a string and is the most elemental form of instruction in Syntax Verifier.

SAD 5XXXXX The SAD Op. code pseudo-instruction is the equivalent of an indefinite iteration of the ∅ Op. code pseudo-instruction. It is an expression which can appear in Syntax Verifier and not in Backus Normal Form notation because Syntax Verifier deals with bounded strings ultimately delimited by an EOM character.

The SAD pseudo-instruction is the equivalent of repeating the ILG instruction until a failure is found. On failure, however, FSA is left pointing to the character that failed and the next pseudo-instruction in sequence

is interpreted. This instruction is particularly useful for checking the type of definition found in the following definitions:

<INT> ::= <DIG>|<INT><DIG> (2)

which could have been written as "A member of the meta-linguistic class <INT> may be defined as an arbitrary, non-zero length string of members of the meta-linguistic class <DIG>." Definition (1) which is written:

<DIG> ::= ∅|1|2|3|4|5|6|7|8|9 (1)

then defines <DIG>. The same definition using the SAD string-checking pseudo-instruction would be written as:

```
INT,        SAD 2
            LAC 1          (2a)
```

where SAD 2 assembles a bit in position 16 of the pseudo-instruction and checks the input string until it finds a character whose description in class ∅ of the atom table does not have a bit in position sixteen (i.e., a non-digit). The LAC command, as discussed later, is used by Syntax Verifier in its failure evaluation routine.

While the previous two IOT's have given us the facilities for checking a character or successive string of characters according to some definition of class membership, it is frequently the case that we wish to verify an input string against a very specific string of literal characters stored in core.

XCT 1XXXXX The five X's in this instruction can have two forms. If bit 5 is a ∅ then the last 12 bits are a pointer to a word in core containing, in its left most byte, the start of the text string against which validation is to be performed. This text string terminates by a normal EOM sign. If bit 5 is a 1, then the low-order 12 bits are a normal indirect pointer. If the register to which they point also has bit 5 set, the indirect chain will continue. The terminus of the chain is a register containing a normal byte pointer which points to the text string against which the input string is to be validated.

The XCT command utilizes two text pointers, FSA for scanning the input text and another pointer for scanning the validating text. As FSA is incremented to each character, that character is looked up in the atom table and its class and group bits are compared with the bit pattern in register SUS. If a match is found, no further validation of that character is performed and the pointer FSA is incremented one character. This step has the powerful feature of permitting the skipping of all those characters in an input string represented by the bit pattern of SUS.

Naturally, the characters to be skipped must not appear in the validating text string since such an appearance will guaratnee failure.

As an example, it should be noticed that by setting SUS equal to 22Ø and setting the text pointer to point to the text string QIDΦ, the execute command will pass as valid the text strings:

QID
Q I D
Q.I.D.
Q. I. D.

Thus the respondent has the freedom to use both the space and the period as format characters in his input string without generating a syntactic failure.

This identity checking routine can be used for checking a portion of the input string as well as the whole input string. The text string to which the pointer address of the pseudo-instruction points, must start at the high order bit of a word, must have at least one character and must be terminated with an EOM. If all of the characters of the text string (with the exception of EOM) are found in that sequence in the input string, the text is passed. At that time, interpretive control passes to the next pseudo-instruction with FSA set to the next character of the input string.

If the text fails (i.e., if any character in the text string does not appear in the input string), the input pointer is reset to where it was before the pseudo-instruction was interpreted and Syntax Verifier enters its failure evaluation routine.

IDX 3XXXXX IDX is a pseudo-command, the low-order 15 bits of which are interpreted in the same way as in the XCT command. The IDX pseudo-command is a Syntax verification command with many of the powers of a context free encoder. Specifically, it is used to compare a text string against the entries and subentries in a dictionary, returning with an indication of the definition.

The pointer in the pseudo-instruction points, directly or indirectly, to the start of the dictionary. The dictionary is terminated by three EOM terminators. Like the ordinary dictionary, the IDX dictionary consists of a series of delimited text strings which are grouped together by some attribute important to the field to be stored. The individual strings are delimited by single EOM terminators and the groups are delimited by double EOM terminators. The dictionary is thus a denumerable set of groups and the test provided by the IDX pseudo code results, not in a simple truth value, but

in the number of the group to which the input string belongs. If the input string does not match any of the groups, of course, then Syntax Verifier enters its failure evaluation routine.

In case membership is detected in one of the groups, a number is left in OPB equal to one less than the ordinal number of the group to which the input string belongs (i.e., the numbers in OPB run from Ø up).

It should be noted that even though the search through a dictionary can be a long process, Syntax Verifier can use a concatenation of instructions in order to do a preliminary screening test of the input string, select among a large set of subdictionaries and then perform the IDX within that dictionary.

Such an operation requires either a very large core to hold all the dictionaries or requires some means of bringing dictionaries in from bulk storage in accordance with the partial execution of a string of Syntax Verifier pseudo-instructions. Given this latter capability, however, it can be seen that Syntax Verifier contains within it the facilities for providing very detailed

dictionary information in order to make it possible for Job Hunter to do very complex branching decision problems.

The IDX command is interpreted under control of the mask in register SUS in exactly the same fashion and with exactly the same restrictions as the XCT command.

JMP 6XXXXX The JMP pseudo-instruction permits the transfer of control directly or indirectly to the address given in the address part of the command. The string of code to which the command points is not interpreted as Syntax Verifier pseudo code but as actual machine coding. This coding may contain Executive IOT's with the exception of those that use a common storage area with Syntax Verifier. The only important ones in that class are: FLOPE, FLOPF, GET, and PUT.

This ability to use Executive IOT's, including the file access IOT's, permits the JMP pseudo-command to be used in extending the power of Syntax Verifier from the detection of pure syntactic errors to the detection of file dependent and hence moderately simple semantic errors. For example, it is the JMP instruction which would be used after a pre-screening

operation in order to retrieve the pertinent dictionary discussed under IDX. Furthermore, it is the JMP instruction which is used for Formulary Verification as described in Memorandum 9. Clearly, since the new coding is strictly machine coding, some method must be provided to return control to the Syntax Verifier interpretive program.

STVW 724040 The STVW or Syntax Verifier "win" IOT is one of the two IOT's which may legally terminate a string of code entered from a JMP pseudo-command. Control is returned to the Syntax Verifier routine at the level of the JMP pseudo-command. The JMP is considered to be a Syntax Verifier verification code and the input text string is treated as having passed that test. None of the pointers within the Syntax Verifier program have been moved by this action unless the machine coding to which the JMP exited caused such pointer modifications.

STVL 724060 The STVL IOT or Syntax Verifier "lose" is the second instruction which may be used to terminate a JMP subroutine. In this case, control is returned to the Syntax Verifier interpretive program at the

level of the JMP pseudo-command and Syntax Verifier enters its failure evaluation routine just as though the input text string had failed to pass that particular test. Note that because of the restriction on common space, the JMP subroutine cannot include a call to Syntax Verifier and hence cannot be used to nest Syntax Verifier definitions.

DIP 3XXXXX The DIP pseudo-instruction is a call to a subroutine which will also be interpreted by Syntax Verifier. It permits other Syntax Verifier definitions to be incorporated into a Syntax Verifier definition program.

When a DIP pseudo-instruction is interpreted, the address portion points to the location in memory of the subdefinition to be checked. If the input string passes the subdefinition, interpretive control is returned to the location following the DIP pseudo-instruction. In case of failure, (i.e., the input string does not pass the subdefinition test) Syntax Verifier enters its failure evaluation routine at the point of the DIP pseudo-command. After a failure, the input pointer is reset to where it was before the DIP pseudo-instruction was attempted. If the input string passes, the input pointer is set as determined by the subdefinition.

As an example in:

<EXP> ::= (<INT>) (3)

we have defined an "expression" as a left parenthesis followed by an integer followed by a right parenthesis. The same definition written for the Syntax Verifier program utilizes the previous definition of <INT> (definition (2)) and is written:

```
EXP,    ØØØØ2Ø
        DIP  INT
        ØØØØ4Ø                    (4)
        LAC  1
```

Where the two octal numbers are ILG codes to look for a left and right parenthesis respectively. It should be noted that the DIP pseudo-instruction may carry as its address part the address of the main instruction in which it is imbedded. This capability permits iterative definitions to be written rather simply. For example, an alternate way of writing the definition of <INT> is:

```
INT,        2
            DIP   INT
            LAC   1          (2b)
```

This definition states that an <INT> is a digit (ØØØØØ2 or just 2) followed by anything which will pass the <INT> test. It is expressly equivalent to the BNF definition (2).

ADD 4XXXXX The ADD pseudo-command permits denumeration, an operation which is extremely awkward in normal BNF notation. The address part of the pseudo-instruction is taken to be a number and causes the pseudo-instruction immediately following it to be interpreted repeatedly that number of times. Functionally it is exactly the same as rewriting that instruction that number of times. It should be pointed out that because it is exactly the same as rewriting the instruction that number of times it is validly used only with those instructions which may logically succeed themselves such as ILG, SAD, JMP, etc.

An example of the use of the denumeration pseudo-instruction is given in the BNF definition:

<UNI>=<DIG><DIG><DIG><DIG><DIG><DIG><DIG> (5)

This statement defines a <UNI> as being a succession of seven digits. The same definition may be written for the Syntax Verifier program utilizing the denumeration pseudo-instruction as:

```
UNI,        ADD  7
            2                    (6)
            LAC  1
```

Since many of the tests for legality in the input format with which we deal are denumeration tests, this pseudo-instruction provides a major saving in programming time and space.

Having described the operational commands of Syntax Verifier, we must now discuss the conditions for termination of a Syntax Verifier definition that lead Syntax Verifier to return control either to the failure location immediately following the STV command or to the success location which is the second register following the STV command.

When an input text string has failed to meet the requirements of the definition, Syntax Verifier enters its failure evaluation routine. It moves its instruction pointer along the list of code reading (but not executing) each instruction until it comes to a terminator instruction. The terminator instructions all have a code starting with the octal number 2 so their detection is simple.

If, on the other hand, the text string has passed the definition, the instruction pointer is moved to the next pseudo-operation code which, if it is a terminator, causes a successful return either to the main program or to the main Syntax Verifier program in the case of a DIP instruction. It should be recognized, that in case of a failure, pointer FSA will be pointing to some arbitrary character within the text string. In the case of a successful passage of a verification, FSA will be pointing either to some character within the text string if less than the whole input

string has passed the test or will be pointing to the EOM terminator if the entire input string has passed the test.

In the case of a successful test, the next pseudo-instruction in line is interpreted while in case of a failure, the failure evaluation routine causes the instruction pointer to be advanced through the Syntax Verifier coding until it encounters a terminator having an operation code starting with the octal number 2.

LAC 2ØØØØØ This terminator literally means "or." If the Syntax Verifier program reaches a LAC with FSA pointing to an EOM symbol, it means that the entire input string has passed the previous tests. It is therefore a valid string under the BNF definition.

If the Syntax Verifier program reaches a LAC with FSA pointing within the message rather than to the EOM terminator, it means that only part of the message has passed the test and therefore the input text string as a whole has failed. Since the LAC command means that there are more definitions to follow, the pertinent pointers within Syntax Verifier are reinitialized, FSA is set to the beginning of the input text string and the Syntax Verifier interpretive program starts interpreting the string of pseudo-codes starting immediately after the LAC.

Notice that this terminator and indeed the meta-linguistic symbols in BNF notation do not permit passing an input string if only a portion of it passes the definition.

LAC 2 2ØØØØ2 The LAC 2 pseudo-instruction represents an actual addition to the BNF language. Specifically, it permits handling trailing garbage in a definition. When a LAC 2 terminator is encountered during the failure evaluation routine, it behaves exactly as a LAC terminator. When it is encountered in the course of a normal step through, however, it assures a successful return whether or not FSA is pointing to the EOM terminator. In fact, upon encountering a LAC 2 pseudo-instruction outside of the failure evaluation routine, FSA is automatically advanced to the next EOM terminator if it is not already on one.

This instruction is extremely valuable in permitting checks for such things as abbreviations. For example, we need not require the spelling out of the words "male" or "female" for we only have a one-bit choice to make. Actually a program might be written as:

```
SEX,        XCT   PTM
            LAC   2
            XCT   PTF          (7)
            LAC   2
            LAC   1
```

where PTM is the address of a register containing the internal code for the letter "M" followed by an EOM symbol and PTF is the same thing for the letter "F." With the definition of (7), the typist may enter "M" or "F" or "MALE" or "FEMALE" or indeed "MASCULINE" or "FEMININE." All that the definition of (7) requires is a string of characters starting either with the letter "M" or the letter "F." The remainder of the string may contain anything.

LAC 1 2ØØØØ1 LAC 1 is a terminator which signifies the end of a set of Syntax Verifier definitions. If it is encountered during failure evaluation, it immediately causes a return to the failure location following the STV call. If it is encountered in the normal stepping operation and the FSA pointer points to anything but an EOM terminator, it again produces a failure return while, if the pointer FSA does point to the EOM terminator it occasions a successful return to the second register following the STV call.

Actually, all of the terminators return to the main program if they are used in the main chain of a Syntax Verifier definition but return one level up from the subdefinition if they are used in a subdefinition.

It should be clear from the foregoing that many operations other than simple syntactic definition may be carried on within the confines of a Syntax Verifier call. The fact that FSA is a text pointer which is moved through the input text string during validation permits it to be used in segmenting or partitioning such input strings. Further, the fact that SUG (the pointer to the next available space in the output buffer table) is reset in the same fashion as FSA permits the value of FSA or a constant to be easily stored within the output buffer in accordance with the progress of the Syntax verification. This feature is frequently used to generate codes prior to storage or to set up pointers that separate out the various fields contained in a single input text string.

A Syntax Verifier call can thus both verify an input string and structure a table in accordance with that string.

F. Interpretation Routines

An often used set of routines that have been included in Core 16 of the Executive Program involves the interpretation of frequently used strings and internal encoding. It is in this area that the peculiar qualities of the Executive Common Routines show their greatest strength for here they are equivalent to routines that are frequently performed by hardware in other machines. Indeed, since the calling sequence to these routines is extremely simple and since the call is made generally through a single IOT, the Core 16 common routines are an excellent place to test

special instructions and to examine their economics while deciding whether to implement them in hardware or to leave them in the form of common routines. The conversion to hardware can be made backward compatible with the routines implemented by means of Executive software. The most obvious routines to consider in this class are the routines which permit the basic system hardware (twos-complement single-precision 18-bit binary hardware) to deal with 36-bit floating point numbers with reasonable programming efficiency and with precision.

All floating point numbers in the system are stored as 36-bit binary floating point numbers in two successive 18-bit registers. The low-order 28 bits represent a ones-complement binary fractional mantissa while the high-order 8 bits makes up a ones-complement binary characteristic. The characteristic, which ranges from minus 127 to plus 127, represents the power of two by which the 27-bit signed binary mantissa must be multiplied. The range of the absolute magnitude of the floating point number X is given by $2.9387358 \times 10^{-39} < X < 1.7014118 \times 10^{38}$. The distribution of bits between mantissa and characteristic was chosen to cover the expected magnitude needed in hospital work and still to provide the 8 decimal digit accuracy required by many statistical analysis programs.

In order to understand the use of the IOT's which deal with the floating point arithmetic, we need first to examine those registers that are used for communication between the Executive Routine and user core.

FAC ØØØØ47 FAC is the first of two registers used as the floating accumulator in user core. All of the floating point routines either refer to or modify FAC and FAC+1.

FOV ØØØ422 When the absolute magnitude of a characteristic exceeds 127, an exponent overflow or underflow is considered to have taken place. At that time each of the floating point routines will set the users overflow flip flop and execute a JMP to register FOV. FOV initially contains a Ø when the users program is started up by Exec and the user is expected to replace this Ø with a JMP instruction to the overflow-handling routine suitable for his particular purposes.

Note that when the decimal floating point routine sets the users overflow flip flop, it also inserts a JMP instruction into the register immediately following FOV. The address part of the JMP instruction is the address in the users coding to which control would have been transferred had no overflow occurred. It is clear then that if the programmer replaces the initial Ø in FOV with an NOP, control will flow through FOV to the register immediately following it and through it to the normal return. Similarly, should he wish to return control to

his routine after taking care of the overflow, he can do so by executing a JMP to the register following FOV.

Since all of the floating point routines are otherwise transparent to the overflow flag, the NOP in FOV may be used throughout a long sequence of floating operations and an overflow may then be tested for after all of the operations have been completed.

Division by Ø is not considered an overflow condition.

The floating point routines are transparent to all the registers in user core except those which they expressly modify in the following definitions. The floating output routines do, however, use the first eight registers of the Syntax Verifier Push Down List area and hence may not be called from a Syntax Verifier Routine.

Only the FIX routines ever modify the users AC and I-O. In the following definitions, all of the addresses may be either direct or indirect and indirect addresses will be followed to any level. The floating arithmetic commands which follow are all macro commands and the macro is given within the definition. The IOT code next to the command is the IOT code used in the macro and is not a direct substitution. It is included here for the sake of programming

assistance only. The first instructions to be discussed are analogous to the normal hardware machine language instructions in the system.

FLAC X 724440 This command assembles two registers of code, the IOT followed by the address part of the call X. This routine simply replaces the contents of the floating accumulator (FAC and FAC+1) with the contents of a two-word floating point register (X and X+1). No normalization takes place during the FLAC called. The contents of X and X+1 remain unchanged.

FDAC X 724460 This command assembles two words in user core, 724460 followed by the address part of the call X. This routine replaces the contents of X and X+1 with the contents of FAC and FAC+1. No normalization is done and the contents of FAC and FAC+1 remain unchanged.

In addition to the above two Executive routines for moving floating point information, we have four arithmetic routines also stored in Core 16. For brevity we shall treat FAC and X as 36-bit registers in the following.

FADD X 724340 FADD assembles two registers in user core, 724340 and the address part of the call X. This routine adds the floating point number

found in address X to the contents of the FAC and leaves the results in the FAC. The contents of X remain unchanged. The normal return is to the first line following the FADD.

FSUB X 72436Ø FSUB assembles two lines of code in user core, 72436Ø and the address part of the call X. This routine subtracts the floating point number found at address X from the contents of the FAC and leaves the results in the FAC. The contents of X are unchanged. The normal return is to the first line following the FSUB.

FMUL X 7244ØØ The FMUL instruction assembles two words in user core, 7244ØØ and the address part of the call X. The floating multiply routine multiplies the contents of the FAC by the contents of the register X leaving the results in the FAC. The contents of X are unchanged. The normal return is to the location following the FMUL.

FDIV X 72442Ø The FDIV routine assembles two words in user core, 72442Ø and the address part of the call X. Unlike the previous routines, the floating divide subroutine has two returns. When the divisor is Ø the divide does not take place and a return is made to the instruction immediately following

the FDIV macro in user core. If the divisor is non-zero, the return is to the next following location. The FDIV routine divides the contents of the FAC by the contents of X and leaves the results in the FAC. The contents of X are unchanged.

These four commands permit all of the necessary floating point arithmetic. In addition to the commands for performing the arithmetic functions, the system also requires a set of commands for converting input strings to internal floating point format and for converting internal format to usable output strings for printing purposes. Similar routines are provided in the system for the fixed point conversion of fixed point registers. Although they are not, strictly speaking, properly part of the floating point section, these routines will be discussed first because of the insight they give us into the use of the other routines. Let us consider the output routines first.

STS ØØØ121 STS is a text pointer used by the output routines to indicate where strings of text are to be stored prior to printing.

SNM 72422Ø SNM is an IOT called with a binary number in the AC and with a pointer to the text buffer in STS. The number in the AC is converted to a text string of decimal digits starting with a minus sign if the number was negative and suppressing all

leading zeroes. The internal codes for the decimal digits are stored in the buffer and STS is left pointing to the next free space in the buffer. SNM does not store an EOM terminator.

SNM2 724260 SNM2 is identical to SNM except that it deals with a 35-bit signed integer, the high-order end being in the AC and the low-order end being in the I-0. Both SNM and SNM2 are IOT's and not macros.

FLOPF ADR, ACC, DIG

FLOPF is a three-argument macro call which assembles three registers in user core, 724320, ADR, + (400000+1, 000(XAC)(XDIG)). As mentioned before, the floating output routines use the first 8 registers of the Syntax Verifier Push Down List and may therefore not be called during a Syntax verification.

ADR is the address of a register containing a character pointer to the buffer area where the text is to be stored. At the completion of storage, the pointer to the first unused character position is placed in register STS. Note that if ADR is set equal to STS, that aspect of the floating output routine is the same as the SNM routines. ACC is a number from one through

8 and is the number of significant figures to which the floating output routine will round the 8-significant figure result that it starts with. Seven-digit accuracy is suggested as the maximum to be asked for. Note that this is a rounding instruction not a truncating one.

The third argument of the macro "DIG" is a number from ∅ through 777_8 and is the number of digits to be stored to the right of the decimal point. This format control character is used primarily for columnation and is independent of the specification of accuracy. If the accuracy is greater than the number of digits specified, the routine rounds to the lower number of decimal places. The buffer is filled with trailing zeroes as required to secure the proper number of digits to the right of the decimal point regardless of the accuracy.

Text is stored as the internal code representation of a string of digits starting with a minus sign if the number was negative, eliminating all leading zeroes and inserting the proper code for a decimal point where necessary. An EOM is not stored by the routine. If the number to be stored is positive, a space is stored as the first character.

FLOPE ADR, ACC, DIG — The FLOPE command assembles exactly the same words in user core as the FLOPF command except that the third word of the macro is positive instead of negative. In the FLOPE command exactly the same operations as those previously described take place except that the text is stored in exponent or scientific notation format. The E format consists of a space or minus sign, one digit, a decimal point, DIG more digits, an E, a space or minus sign, and exactly two more digits. Its length is always thus exactly seven plus DIG characters long and is a convenient format for columnation. In both FLOPF and FLOPE, even if DIG is set equal to Ø, a decimal point is always stored.

In addition to the direct output routines, the system also has routines for converting from internal floating point notation to fixed point notation.

FIX N 72452Ø — The FIX IOT assembles two words in user core, 72452Ø and the argument of the call N. N is a number from Ø through 17_{10}. This routine converts the floating point number in the FAC to fixed point notation without rounding. The fixed point number is left in the actual AC. The contents of the FAC are unchanged. The location of

the binary point in the resulting 6 point number is dictated by the argument to the call N. If N is Ø, the binary point is to the right of bit 17 and the binary number is a pure binary fraction; if N is plus 17_{10}, the binary point is to the right of bit Ø and the binary number is a pure binary integer. If the fixed point number is too large for an 18-bit word, an exponent overflow trap occurs and the AC remains unchanged.

FIX2 N 72456Ø The FIX2 IOT assembles two words in user core, 72456Ø and the argument of the call, N. This routine converts the floating point number in the FAC to a 36-bit fixed point double precision number without rounding. The fixed point number is left in the combined AC and I-O. The binary point of the fixed point number is determined by N where N can lie from Ø through 35_{10}. If N is zero, the binary point is to the right of bit 35 and the binary number is a pure fraction while if N is plus 35, the binary point will be located to the right of bit Ø and the binary point will be a pure double precision binary integer. If the fixed point number is too large for a 36-bit word, an exponent overflow trap occurs and both the AC and I-O are left unchanged.

In addition to routines to convert internally-stored floating point numbers to both fixed point numbers and to text strings for representation to the outside world, we need a set of routines to convert internally-stored binary fixed point numbers and incoming sets of text string to floating point numbers. Similarly, we need routines for converting incoming text strings to standard internal binary numbers.

DNM 7242ØØ The IOT for decoding a number is called with a character pointer in the AC pointing to an internal code string. Starting at that point, the characters are examined one by one until a character other than the digits Ø through 9 is found.

FSA then contains a pointer to the first non-interpretable character in the string. The resulting positive binary number is left in the AC. Note that DNM does not recognize minus signs or other symbols which must be recognized by other routines. If the number encountered is greater than 131Ø71 or if no digit at all is encountered, an abnormal return is made to the location immediately following the DNM. If the decoding is successful, the return is made to the second location following the DNM.

DNM2 724240 DNM2 functions the same way as DNM except that the resulting double precision positive binary integer is left in the low-order 35 bits of the combined AC and I-O. DNM2 also has two returns. Note that DNM and DNM2 are direct IOT's and not macro calls.

FLOAT N 724500 This macro call assembles two words in user core, 724500 and the argument N. This routine treats the contents of the accumulator as a fixed point binary number, multiplies it by 2^N and converts it to a floating point number which is left in the FAC. The contents of the AC are unchanged. The float command expects the AC to represent a binary integer. Thus, if the binary point is located to the right of bit 17 conversion ordinarily takes place through FLOAT 0. If, in the system being used, the contents of the AC are not intended to represent a binary integer, multiplication by the proper power of two must take place to inform the float routine of this fact. Thus, if it is intended that the AC represent a pure binary fraction (i.e., the implied binary point is to the right of bit 0) conversion is made by FLOAT -17. Note that for extreme values of N exponent overflow trapping can occur.

FLOAT2 N 72454Ø This macro call generates two words in user core, 72454Ø and the argument of the call N. This routine behaves just like FLOAT N except that it assumes that the contents of the AC and I-O form a signed 35-bit fixed point double precision integer. Note that for extreme values of N exponent overflow trapping can occur.

FLIP X 7243ØØ FLIP is a macro call that generates a two-word entry in user core, 7243ØØ and the argument to the call, X. X is the address of the register containing a character pointer to a character in a string in user core.

Starting at the character thus pointed to, the characters are interpreted one by one and a floating point number is developed. Interpretation stops when a character is encountered which is incompatible with the foregoing characters or which is not a constituent of a legal floating input text string. FLIP, unlike DNM, will operate on plus, minus, decimal point, E and the decimal digits Ø through 9. The floating point number is put in the FAC and a pointer to the first non-interpretable character is left in register FSA. The routine is subject to exponent overflow.

FLIP has two returns, the second register following the FLIP command is the legal return to which control is returned if any legal string is found. If the combination of characters starting at X cannot be interpreted as a floating point number, control is returned to the register immediately following FLIP and the FAC is left unchanged.

The following examples are all legal strings. In each case the second column contains the floating point number that will be put in the floating accumulator. The arrow represents the contents of FSA.

1.ØABC	1.	(All letters except "E" are non-interpretable)
Ø.1.35-	Ø.1	(A second decimal point doesn't make sense)
+1.3E5 A	$1.3 \times 1Ø^{5}$	(Space is never interpretable)
-1.6-3E5BQ	-1.6	(The second minus sign has no meaning)
+1.6E+7-3.	$1.6 \times 1Ø^{7}$	(The minus sign has no meaning except immediately after the "E" or as first character)
1E-1.6	$1 \times 1Ø^{-1}$	(A decimal point has no meaning intthe exponent field)
1E1E1	$1 \times 1Ø^{1}$	(The second "E" has no meaning)
Ø,3	Ø	(Commas and all other special characters are non-interpretable)

The following examples are all illegal. The return will be to the first line after the flip with the floating accumulator unchanged. Here the arrow points to the character causing the illegality.

```
 ↓
ABCDEF

1.E↓AA

1E.↓AAA

 ↓
.E1 AA

 ↓
+-3AAA

 ↓
++3AAA

 ↓
..3AAA

 ↓
+E+1
```

The foregoing constitutes all of the numerical conversion routines that are included in the Executive system.

In addition to the purely numerical routines, however, certain frequently-used fields are kept in a densely, coded form. Coding and decoding routines for these fields are also kept in Core 16.

The most important of these routines have to do with times and dates. A date is stored internally in this system as an 18-bit binary integer representing the number of days from January 1, 1849. January 1, 1849 is date ∅ and all of the days from then on are numbered consecutively.

Time is also stored as a binary integer. Zero means midnight and each successive minute is numbered consecutively. Thus the time integer can range from Ø through 1339.

There is also a one-word format for both time and date in which the left 11 bits are the number of days since the January 1 following the last leap year. Currently, January 1, 1965 is defined as date Ø. The low-order 7 bits are the time in quarter hours since midnight, an integer from Ø through 95. As described earlier, GTD permits the user program to reference the current time and date in a two-word format.

TD2T01 72542Ø This IOT is used to convert from the two-word format for time and date to the one-word format. It is called with the date in the AC and the time in the I-O. It returns to the first register after the TD2T01 command with the date and time in the AC truncated to the lowest quarter hour. Note the routine does not round but truncates.

TD1T02 7254ØØ This IOT is called with the time and date in the user's AC in standard one-word format and returns to the first register after the IOT with the standard two-word format with the date in the user's AC and the time in the I-O.

In addition to internal routines for translating internal time and date representation, there is one major routine for typing out internal two-word representations.

SUTD 72414Ø SUTD is called with STS containing a pointer to the buffer in user core where a text string is to be stored. The AC contains the date and the I-O the time in internal representation. The routine returns to the register immediately following the IOT. The format is hour, colon, minute, space, am or pm, space, day, slash, month, slash, year, EOM. Although the time used internally by the two-word format is in "minutes since midnight" (and, consequently, based on the 24-hour clock), the 12-hour system is used for print-out. There is no hour zero (this is converted to twelve), and 12 AM is defined as midnight. Examples are in quotes:

"12:ØØ AM 7/1Ø/1941" or
"4:32 PM 1/13/187Ø".

SUTD also stores an EOM after the text string and leaves register STS pointing to the next free character space following the EOM.

TDNUM 72534Ø This routine stores the hour, minute, month, day and year as integers in five sequential registers in user core. Hour is a number

from zero through 23. The IOT is used with the register immediately following the IOT containing a direct or indirect pointer to the five-word register buffer in user core.

The IOT is called with the date in the user's AC and the time in the I-O and the return is made to the register following the pointer.

DCDTD 725360 This IOT is the exact inverse of TDNUM and is used in conjunction with Syntax Verifier for encoding dates and times into the form used by the system. The IOT is called with the AC containing a pointer, direct or indirect, to the block of five registers to be decoded. If the decoding can be validly accomplished, that is, if the integers in the five registers represent real dates and times, a return is made to the second register after the IOT with the standard two-word format in the AC and I-O. If any of the integers is too large for its class or if they form an invalid set (e.g., February 29, 1961) the return is made to the first register following the IOT. In this case the contents of the AC and I-O are undefined.

With the completion of the time and date routines, we have covered all of the basic encoding routines in the current Executive. Thus we now have described a system which allows us to take fairly complicated text strings, secure them from a respondent through a highly branched logical structure, validate them and encode them as necessary. In the next section we will examine those routines which permit us to structure these fields in the logical form which makes their storage and retrieval a relatively simple matter.

G. Logical Structuring

Those programs that use the active-patient record portion of bulk storage for their records have available an internal set of programs to facilitate the construction of items from sets of fields. These programs further facilitate the subsequent dissection of these items on retrieval.

An item is a set of fields stored together for convenience. Within an item storage may be fairly complex. The information may be stored in multiply subscripted form of essentially unlimited depth. A single Executive routine, PUT, stored in Core 16 is used to order both the fixed and variable length fields, to provide for their subscripting and to form them into a single dense body of data. A second routine, GET, also stored in Core 16 of Exec is used to retrieve individual fields from these items given certain information about the field name and its subscripts. PUT and GET have both been written as extremely general routines, and currently are only limited by the requirement that the entire item must fit in core at one time.

It is assumed that any item for which PUT is used as a logical structuring mechanism may have an overhead section which is not pertinent to the structuring of information within the item but is pertinent to that part of the Processor which deals with filing and unfiling of such items. Figure IV-G-1 is a diagram of the overhead contents of any item in the active patient record. It should be noted that each item has 11_8 words of overhead information consecutively numbered from Ø through 1Ø. Words Ø and 1 are not supplied by the user program, but are supplied by the processor section of Exec upon filing. All other words are supplied by the user program as part of the calling sequence to the filing IOT. Word 1Ø is a character pointer to the next free byte in the string area relative to the start of the string area itself. The first word of the string area is thus referred to as word Ø in that character pointer. Word 7 is a word pointer to the start of the string area relative to the pointer address itself. As a result, the word part of word 1Ø plus the value of word 7 plus $1Ø_8$ yields the length of the item in words. (In order to keep from writing out empty words, this number is reduced by one if the character pointer of word 1Ø is pointing to the first byte of a word.)

It will be noticed that the first two words of the item provide a pair of chains through the record. These chains link the items that make up a single record. Words 3 through 6 contain logical entries for searching and ordering. Word 2, bits 2 through 7, is the item subscript to make the set of items within a record effectively a two dimensional

array. Bits 13 through 17 are a set of protection bits used in rewriting and correcting. Their function will be found discussed under the IOT WAIA in the next section.

The body of the information, words 11 et.seq., is the part of the information which is logically structured by the PUT routine. In doing such structuring, PUT modifies the contents of word 7 and 1Ø and hence it considers that the item given in the example of Figure IV-G-1 has only seven actual words of overhead (i.e., word Ø through 6). We shall, therefore, in our discussion of PUT and GET consider words 7 and 1Ø of the example to be words Ø and 1 of the area with which PUT and GET are concerned. Before we can discuss the mechanism by which PUT and GET structures the information, we must examine the structure itself in order to satisfy ourselves that it is general, dense and (with effort) even understandable.

An item may contain fields in two formats: fixed length or variable length. Fixed length fields are generally used for codes or other kinds of information which conveniently fit into a restricted space. Most of the information systems that use punched cards are accustomed exclusively to the fixed length format. For convenience in our discussion we will refer to these fields as simply fixed fields.

The second format used is the variable-length field which has no specified length. The variable field is most often used for strings of text that may range from a single

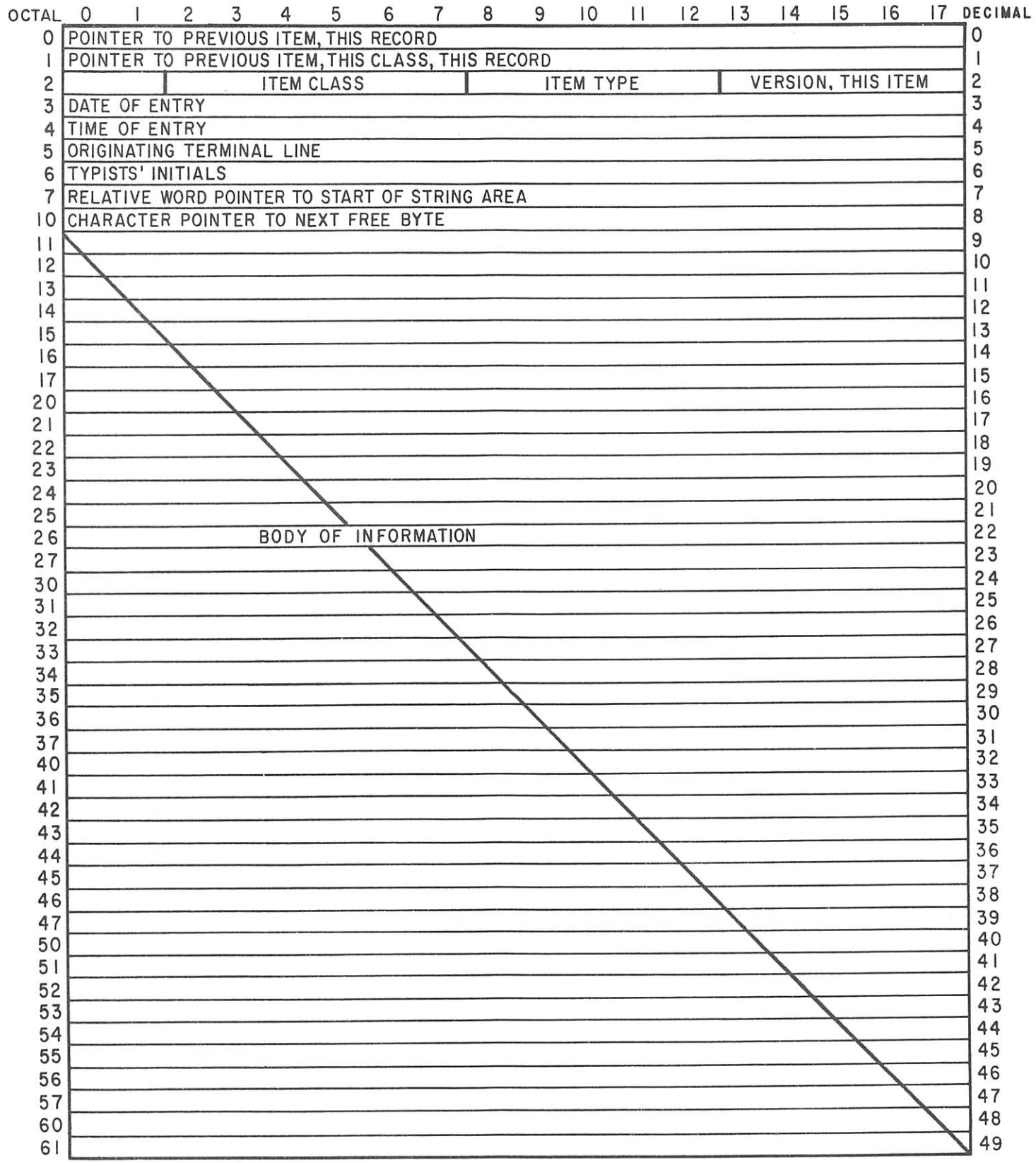


FIG. IV-G-1 OVERHEAD OF ITEM

character to pages of narrative. These variable fields are strung out one after another in the part of the item called the string area. Three characters are packed to an 18-bit word (except for occasional 12-bit characters) and each string is terminated by an EOM terminator. The order of the strings within the string area is completely unrelated to the logical structure of the information within the item. The string area is merely a convenient storage buffer which follows after all of the logical structure has been specified.

All variable fields are represented within the logically structured portion of the item (that portion that lies between the overhead and the string area) by character pointers. Each such pointer points to the pertinent text string.

If we think of each of the text pointers as a one-word fixed field, we should have little difficulty in following their handling by GET and PUT. All such pointers are character pointers relative to the start of the string area. All other pointers in the item are pointers relative to the location of the pointer itself. Notice that the use of relative pointers permits GET and PUT to operate with any buffer area in user core regardless of its actual core location. The stored items are fully relocatable.

The fields in an item are assembled into logical groups which are united by some implication of commonality. Such a grouping might link together the name of the surgeon and

the operation he performed or the patient's unit number and his date of birth. In either case, the smaller elements are fields but most of our logical searching, our subscripting and our Boolean tests must recognize the existence and the composition of the individual groups.

In addition to the fixed versus variable distinction among fields, we must also distinguish those fields or groups that are subscripted from those that are not. We will simplify our task by dealing only with groups. If there is a single field we consider it merely as the degenerate form of a group. Groups are classified as primary or secondary. The primary group exists once and only once within every record. The primary group, in fact, serves as the identifier of that record and contains information which is unique to the record. Since the implication of commonality or the linkage among the primary fields consists of their forming part of a single record, there must be one and only one primary group in every record regardless of its nature.

Secondary groups carry most of the information in a record. To refer to a secondary group, one must assign it a subscript in order to identify it within that record. Indeed, since pieces of information within the record may "belong" to many other pieces of information, it may take a large number of subscripts to identify any single piece of information.

As a somewhat frivolous example, each of the groups containing a surgeon's name and an operation might have a pointer associated with it pointing to the collection of nurses which participated in that operation. We could thus refer to the third nurse of the second operation, a reference which has two subscripts. Similarly, associated with each nurse's name might be a pointer pointing to the collection of her boyfriends. We could thus refer to the second boyfriend of the fifth nurse of the second operation, a reference containing three subscripts.

Conceptually, if not usefully, this concatenation and continuing subdivision can go on indefinitely. Thus, a single piece of information in the active patient record may require an indefinite number of subscripts for its unique specification. Furthermore, the same item which contained the information about operations, surgeons, nurses and boyfriends may also contain an equivalent batch of information about office procedures, doctors performing them, etc. Thus, in any one item of a record there is not only no restriction as to the number of subscripts that a group may have but there is no restriction as to the number of kinds of groups within an item.

While this highly general structure may seem to be complete on casual observation, it is actually suitable only for information which can be structured in the form of a tree. For the kind of information that must be structured by a map having closed loops, the present system, as is the case with any subscripting system, requires replication of some of the fields.

If we examine the information we have just discussed, we can detect an isomorphism among the various levels of subscripting. If we take a group at any subscripting level, that group contains specific information pertinent to the group itself as well as pointers to those other groups for which that information constitutes an implication of commonality. In other words, any group contains pointers to those groups that it subsumes. The information in the group serves as the value, on a nominal scale, of the lower-level groups' subscript.

Since we are dealing with a multi-dimensional space having nominal rather than ordinal scales, we may permit a single group to have several pointers. In our previous example, a group might consist of the name of the operation, the name of the surgeon who did it and, as described before, a pointer to the nurses who assisted in the operation. An additional pointer in the same group might be a pointer to the names of the equipment manufacturers whose equipment participated in the operation. Each such lower-level group might then have a pointer to the names of the pieces of equipment so involved. Clearly, the structure of the information we are describing looks very much like the structure of a tree. Like a tree, in which it is essentially impossible to distinguish between a section cut off at the trunk and one cut off at a limb or even at a smaller branch, the forms are truly isomorphic.

Note that with this description, it may seem somewhat unnecessary to distinguish between the primary group and secondary gourps. We might simply consider the primary group as being the level of zero subscript. If it were not for the fact that we insist on the existence of the information in the primary group, this distinction would in fact not be necessary. In the following discussion, we will ignore the fact that we insist on the information in the primary field and will instead treat it as information at level zero of the subscripting tree.

Clearly, the problem that we have is one of mapping this multi-dimensional tree space onto a one-dimensional array. This problem is commonly met with in most compilers and has been treated in the literature. Since our item organization permits the nominal structuring of the subscript scale, the problem is more complex as is our requirement that the storage array generated be both one-dimensional and everywhere dense.

We will, in order to make the information structure and the mapping algorithm clearer, go through an actual illustrated example. Since an isomorphism exists between subscript levels, it will be unnecessary to carry our example through the entire structure as long as it indicates the mapping to be followed.

Figure IV-G-2 illustrates the case of a company served by two trunk line telephone numbers, each tended by a secretary at a numbered desk location. We will further assume that the first of these trunks has two extensions; the

first of which serves two lawyers and the second of which serves only one. The second secretary's telephone has but a single extension serving a single lawyer. If we examine the groups at level 1 we see that there are three expressed fields (secretary name, phone number and desk location) and one implied field, an indicator to show which of the level 2 groups "belong" to that particular level 1 group. Similarly, a level 2 group consists of one expressed field (the extension number) and one implied field namely an indicator as to which level 3 groups belong to that particular level 2 group. At level 3 we have a set of degenerate groups each made up of one field.

We have shown the dashed line from extension number 3 on level 2 to lawyer number 3 to indicate the kind of information dependence that our particular subscripting structure cannot handle. Should we wish to indicate lawyer 3 "belonging" to extension 3 as well as to extension 2, we would have to have lawyer 3 duplicated in our tree structure.

Since we are interested in mapping the diagram of Figure IV-G-2 onto a one-dimensional array or line, we can visualize writing the groups of each level in consecutive order on that line. Further, we can specify that groups "belonging" to the same group will be treated as a list so that if the end of the list is somehow delimited, the group owning the list of subsidiary groups need point only to its beginning. Figure IV-G-3 is an example of this mapping technique. To understand the structure, we must realize that the boxes on

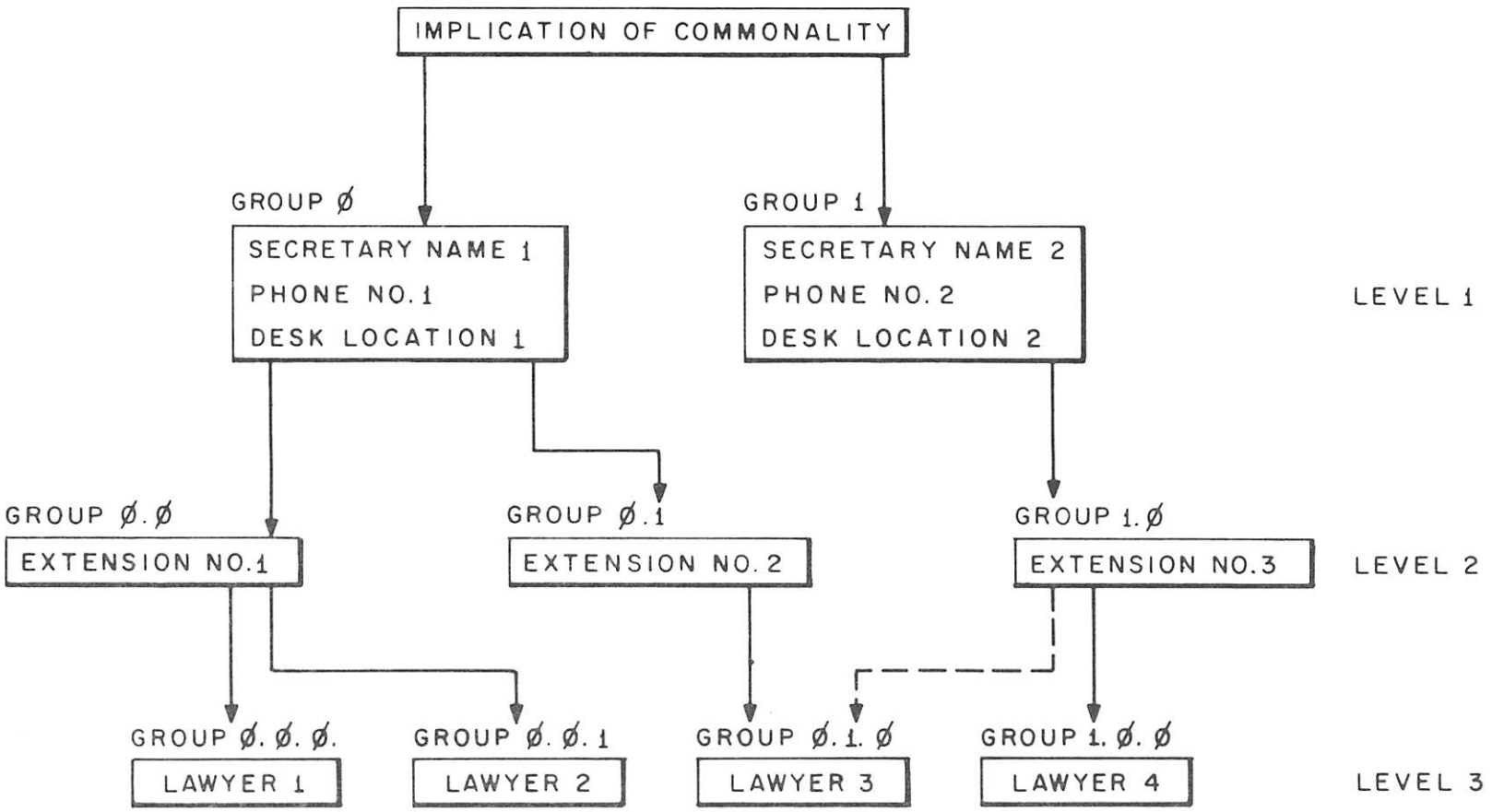


FIG. IV-G-2 DIAGRAM FOR TRIPLY-SUBSCRIPTED INFORMATION

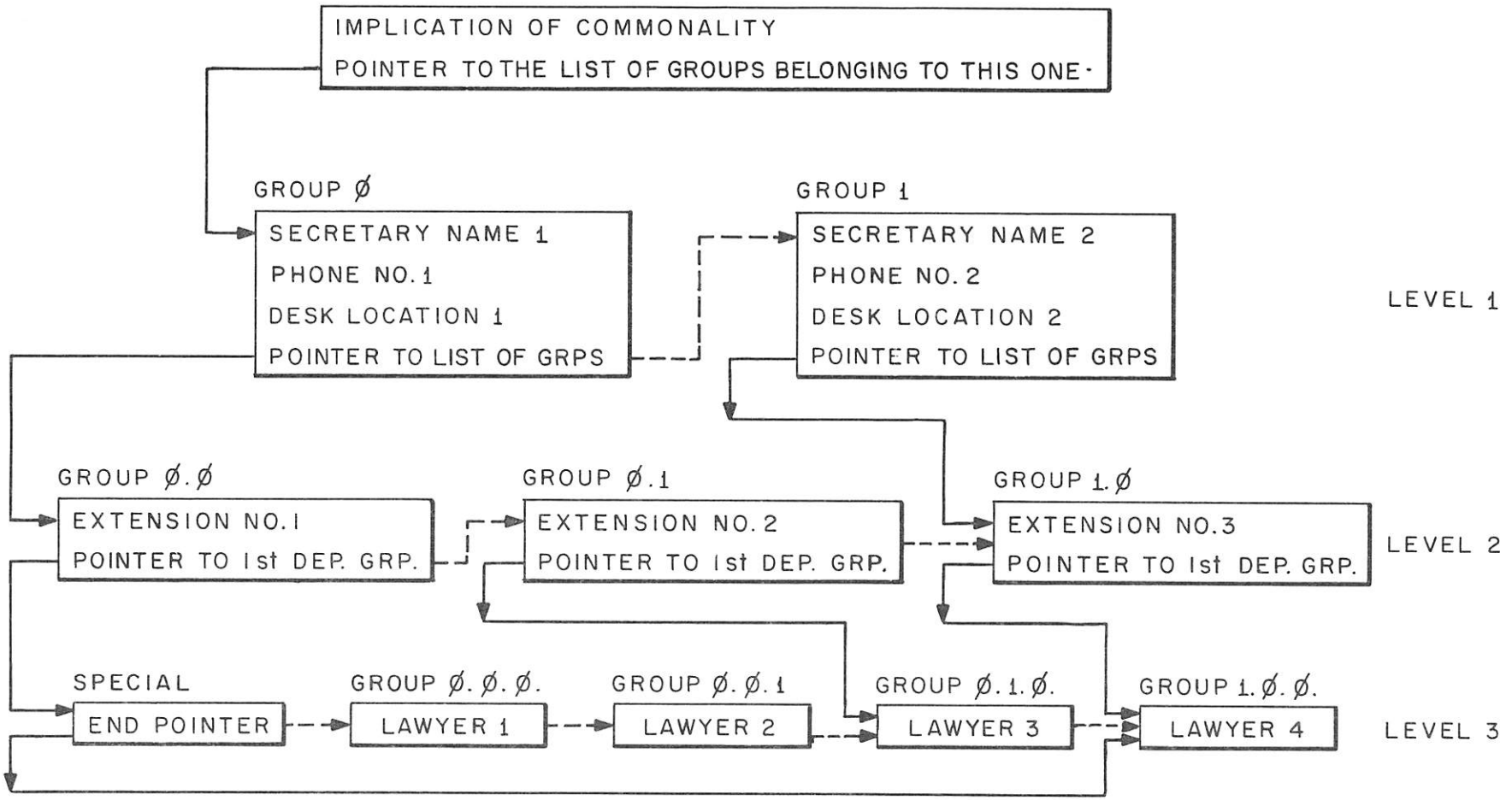


FIG. IV-G-3 TELEPHONE EXAMPLE WITH POINTERS

each level must be visualized as written out consecutively on a line reading from left to right on the level and then starting with the left end of the next level. If there were multiple trees, each such tree would be taken in turn. The dashed arrows within level 1, 2 and 3 indicate this consecutive writing.

The first group contains the implication of commonality which may simply be the primary information for the item and a pointer to the list of groups "belonging" to this one. The first such group contains within it the values of the actual nominal fields (secretary name, phone number and desk location) as well as a pointer to the list of level 2 groups belong to group $\emptyset$ on level 1. Now we must decide what the next location on that line (indicated by the dashed arrow) represents. It can either represent the start of another group in the same list as the preceding group, the start of another group on the same level but in a different list, or the start of information at a lower subscripting level. Here we can ascertain that it is the start of another group in the same list since we have not encountered any other pointer to that location starting from the top of the tree. We shall see that this additional pointer criterion is sufficient to separate out the three possibilities.

We note that group 1 also contains the values for the three fields and a pointer to the list of groups belonging to group 1. The next location in line after group 1 is actually the extension number 1 although we have no way of knowing this simply from its position on the line. Again

it could represent either the start of another group in the same list, the start of another list at the same level or information on a lower level. In this case we find that there is indeed a pointer pointing to the same address and the pointer originates in a level 1 group. Since all pointers point to groups at a level below themselves, the information at the location where we are now cannot be either a group in the same list as Ø and 1 or a member of another list at the same level. It must therefore be the start of a list at the next lower level.

We can now continue along our line through group Ø.1 until we come to the start of group 1.Ø. Here again we must decide whether the information at this point on the line belongs to the same list as group Ø.Ø and Ø.1, whether it is the start of a new list at the same level, or whether it is again the start of a new level. In this case we find a pointer from level 1 to this location so that the location must be a level 2 location. Since it has a pointer to it, it does not belong to the same list as the previous two groups but is the start of a new list.

We see, therefore, that the ability to trace back pointers is sufficient to determine whether a group is a continuation of a list, starts a new list at the same level, or starts a new level. At the lowest level of the tree, however, we must introduce a special pointer shown as the left most pointer in level 3 in Figure IV-G-3. This pointer does nothing but point to the end of the lowest level of the tree. Clearly, in order for this technique

to work, the lowest level of the tree must be marked as being the lowest level. One could avoid this requirement by providing a null group which would be the sole member of the lowest level. To do so, however, would restrict the possible contents of groups. It should also be noticed from Figure IV-G-3 that the pointer in the first group at any level points to the first group in the next level.

In Figure IV-G-4 we have shown this same information actually mapped linearly along a vertical line and have also shown how the pointers to other trees would be handled if they also existed. For programming convenience, we have made the pointer the first register in each group rather than the last register as shown in Figure IV-G-3. Clearly, this modification is one of convenience only. Note that the right most set of pointer arrows delimits the start of each level since they are pointers from the first group in the preceding level. Also note that since the pointers are now the first register in each group, pointers always point to pointers.

In our little example, we have treated all fields as though they were fixed length as indeed we may. Any fields that are not fixed lengths are represented in our diagram by fixed-length pointers to text as described earlier. Let us carry our diagram one step further now and show the information incorporated in an actual item. Figure IV-G-5 is such an item. We have arbitrarily shown the item as having a five-word overhead and have, on the right hand side indicated the subscript levels of the information. On

the left side of the diagram, we have shown the number of the field type which we will use later on in identifying these fields in terms of the item map.

Both GET and PUT are IOT's which are called in conjunction with a pointer to an item map. For generality, this map must in fact, provide a complete specification of the interconnection logic of the item. It cannot, of course, say anything about the actual number of groups existing at each level. It must clearly describe each group and each field within the group. It must describe which groups belong on which levels; which groups are primary and which are secondary; which are variable and which are fixed and must also indicate which groups represent the lowest level in their respective trees. In order to contain all this information, the item map is made up of individual field maps. Figure IV-G-6 shows the four possible types of field maps. Each map is three words long and contains all the information required about a field or the group to which that field belongs.

The fields in an item are of four forms. They are primary fixed length, primary variable length, secondary fixed length and secondary variable length. In addition, each of the fixed-length fields can either contain a pointer or information. The variable fields are, of course, all represented by pointers. As can be seen from Figure IV-G-6, there is a field map format for each of the above types of fields. The item map is made up of a collection of such field maps.

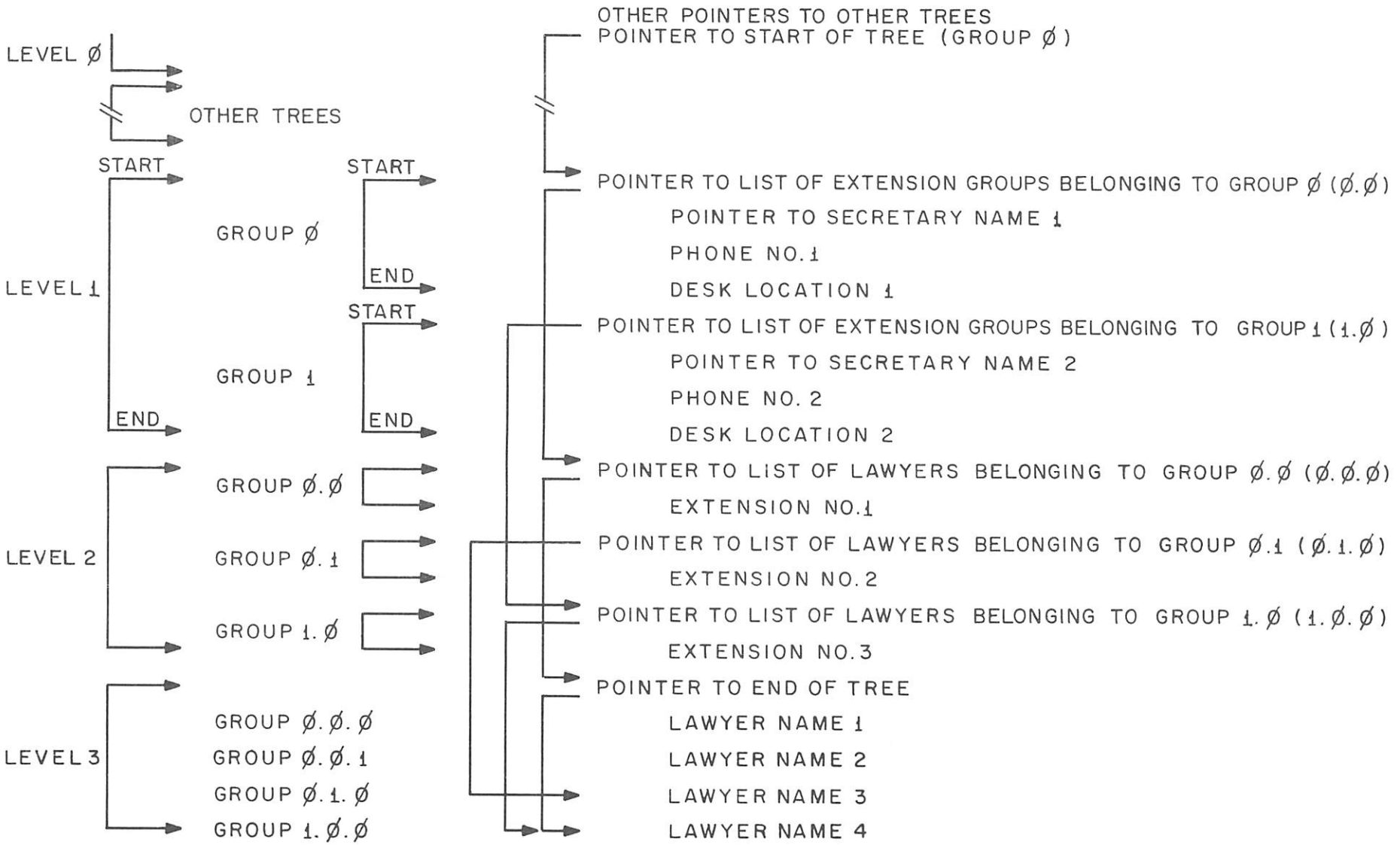
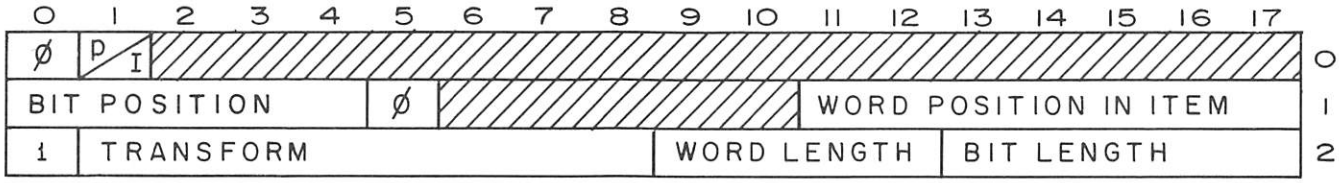


FIG. IV-G-4 LINEAR INFORMATION MAPPING

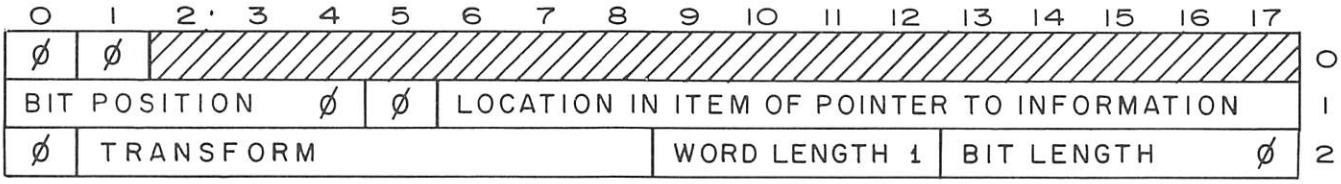
FIELD TYPE	WORD IN ITEM	OVERHEAD AS REQUIRED BY FILE / FIELD VALUE				WORD NUMBER	
						0	
						1	
						2	
						3	
						4	
0	0	WORD POINTER TO START OF STRING AREA (RE THIS ADDRESS)		3	4	5	
1	1	BYTE POINTER TO NEXT FREE BYTE (RE START OF STRING AREA)	2,	2	4	6	
2	2	WORD POINTER TO START OF TELEPHONE TREE (RE THIS ADDRESS)			5	7	
3	3	BYTE POINTER TO COMPANY NAME (RE START OF STRING AREA)	Ø,		Ø	10	LEVEL Ø
4	4	BYTE POINTER TO COMPANY ADDRESS (RE START OF STRING AREA)	Ø,		3	11	
5	5	COMPANY RATING (FIXED FIELD ENCODED)		2	7	12	
6	6	COMPANY TYPE (FIXED FIELD ENCODED)			3	13	
7	7	WORD POINTER TO EXTENSIONS ON PHONE 1 (GROUP Ø.Ø) RE THIS ADDRESS		1	2	14	
10	10	BYTE POINTER TO SECRETARY'S NAME 1 (RE START OF STRING AREA)	2,		6	15	
11	11	PHONE NO 1 (FIXED FIELD, 7 BYTES)	4	9	1	16	
11	12	BYTES 4, 5 AND 6	1	8	5	17	
11, 12	13	BYTE 7 AND DESK LOCATION 1 (FIXED FIELD ENCODED)	Ø	Ø	3	20	LEVEL 1
7	14	WORD POINTER TO EXTENSIONS ON PHONE 2 (GROUP 1.Ø) RE THIS ADDRESS		1	1	21	
10	15	BYTE POINTER TO SECRETARY'S NAME 2 (RE START OF STRING AREA)	1,	1	1	22	
11	16	PHONE NO 2 (FIXED FIELD, 7 BYTES)	B	I	4	23	
11	17	BYTES 4, 5 AND 6	9	3	2	24	
11, 12	20	BYTE 7 AND DESK LOCATION 2 (FIXED FIELD ENCODED)	2	Ø	4	25	
13	21	WORD POINTER TO LAWYERS ON EXTENSION 1 (GROUP Ø.Ø.Ø) RE THIS ADDRESS			6	26	
14	22	EXTENSION 1 NUMBER	2	7	2	27	
13	23	WORD POINTER TO LAWYERS ON EXTENSION 2 (GROUP Ø.1.Ø) RE THIS ADDRESS			6	30	LEVEL 2
14	24	EXTENSION 2 NUMBER	2	8	1	31	
13	25	WORD POINTER TO LAWYERS ON EXTENSION 3 (GROUP 1.Ø.Ø) RE THIS ADDRESS			5	32	
14	26	EXTENSION 3 NUMBER	2	4	4	33	
—	27	POINTER END OF LEVEL 3 (SPECIAL) RE THIS ADDRESS			4	34	
15	30	BYTE POINTER TO LAWYER 1 NAME (RE START OF STRING AREA)	Ø,	1	4	35	
15	31	BYTE POINTER TO LAWYER 2 NAME (RE START OF STRING AREA)	1,	1	6	36	LEVEL 3
15	32	BYTE POINTER TO LAWYER 3 NAME (RE START OF STRING AREA)	2,	2	Ø	37	
15	33	BYTE POINTER TO LAWYER 4 NAME (RE START OF STRING AREA)	Ø,	2	3	40	
	34	WORD Ø, STRING AREA	B	B	N	41	
	35	1	Δ	I	N	42	
	36	2	C	•	⊕	43	
	37	3	5	Ø	Δ	44	
	40	4	M	O	U	45	
	41	5	L	T	O	46	
	42	6	N	⊕	H	47	
	43	7	A	R	R	50	
	44	1Ø	I	E	T	51	
	45	11	⊕	P	A	52	
	46	12 STRING AREA	U	L	I	53	
	47	13	N	A	⊕	54	
	50	14	H	U	G	55	
	51	15	H	E	S	56	
	52	16	⊕	B	O	57	
	53	17	I	L	E	60	
	54	2Ø	N	⊕	B	61	
	55	21	E	L	L	62	
	56	22	E	R	⊕	63	
	57	23	M	A	N	64	
	60	24	N	⊕		65	

FIG. IV-G-5 EXAMPLE ITEM

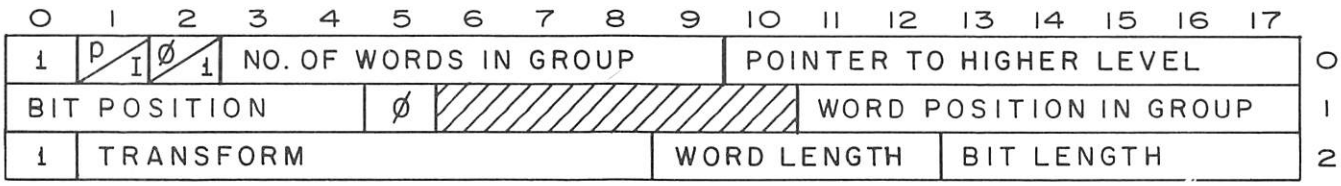
FIELD MAP LAYOUT FOR PRIMARY FIXED FIELDS



FIELD MAP LAYOUT FOR PRIMARY VARIABLE FIELDS



FIELD MAP LAYOUT FOR SECONDARY FIXED FIELDS



FIELD MAP LAYOUT FOR SECONDARY VARIABLE FIELDS

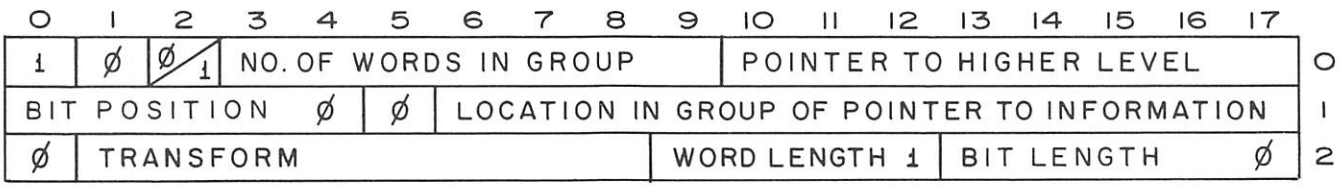


FIG. IV-G-6 FIELD MAPS USED IN THE ITEM MAP

Referring to Figure IV-G-6, the map for a primary fixed field contains, in word Ø, a Ø in bit Ø to indicate that it is a primary field and a Ø or a 1 in bit 1 of word Ø to show whether the field contains a pointer or information. In word 1, bits Ø through 4 contain the bit position of the first bit in the field value and bits 11 through 17 contain the item address in which the field value begins. Bit 5 always contains a Ø for programming purposes. Word 2 of the primary-fixed-field field map contains the 1 in bit Ø to indicate that it is a fixed field and the length of the field in words (bits 9 through 11) and in remainder bits (bits 12 through 17). Primary fixed fields may be packed as tightly as desired in order to conserve space. Good practice for saving time on retrieval, however, dictates that each field be encoded in whole words.

Next in Figure IV-G-6 is shown the field map for a primary-variable field. Again, bit Ø of word Ø is a Ø to show that the field is a primary field. Bit 1 is necessarily a 1 since all variable fields are information fields. Bits 6 through 17 of word 1 contain the item address, not of the information, but of the pointer to the primary variable information. Note that all variable-field maps refer to information whose length is unspecified, so they actually locate pointers within the item. Bits 9 through 12 of word 2 contain a 1 (the length of the pointer in words) while bits 13 through 17 contain a zero (the remainder bits). In all of the field maps, bits Ø through 8 of word 2 designate one of the 512 transforms that may be applied to the data as an encoding operation before storage. This transform data is not used directly by GET or PUT.

The third and fourth field maps shown in Figure IV-G-6 are for the secondary fields. These differ from the primary fields in that bit 2 of word Ø is a switch set to a 1 if the field is at the lowest level of its tree. In bits 3 through 9 in both secondary field maps we specify the number of item words in the group of which this field is a member. Bit Ø, word Ø of all secondary field maps is set to a 1. Bits Ø through 4 and 11 through 17 of word 1 again define the bit and word position of the fixed-field information as they did for the primary fixed field. In the case of secondary fields all pointers are relative to the beginning of the group in which the field is located since no absolute addressing can be done within a tree without an a priori knowledge of the number of members. Note that the relative addressing convention is consistent with considering the primary fields to make up the level Ø group.

In the secondary variable field, the only additional change occurs in word 1 where the address of the byte pointer is now given in group address form rather than in item address form.

In both secondary field maps, bits 1Ø through 17 of word Ø contain a pointer that is used in the item map rather than in the item itself. This pointer is an absolute word pointer to the next highest-level field map in the item map. Absolute word pointers can be used here since the item map is made up entirely of fixed-length entries. The highest level pointed to by this pointer in the item map will be a

pointer to the beginning of the tree. Note that this system of pointers lets us progress upward through the levels in the item map to find the start of the pertinent tree.

In Figure IV-G-7 we have assembled the item map for our example item. The first word of the item map contains, in its first seven bits the number of field maps contained in the item map. For our type of item this number is 16. In bits 7 through 13 we have the number of primary field maps contained in the item map, and bits 14 through 17 give the number of overhead words contained in the item itself. These last two figures are 7 and 5 respectively for our item type.

Following this first word we list the 16 field maps for the 16 fields contained in this item map. Field map Ø describes the primary-fixed field containing the byte pointer to the beginning of the information string area. If we examine this field map (contained in words 1, 2 and 3), we see from bit 1 of word 1 that it is a pointer field, we see from bits 6 through 17 that the pointer is located at word Ø in the item. From the last word of the field map we see that the field is one word long. The reader can follow down the remainder of the field maps. The fundamental rule is as follows: The second map describes the pointer to the next free byte of string storage. Following this field map are the maps describing the pointers to the beginning of each of the individual trees. The maps appear in the same order in the item map as the pointers do in the actual item. The trees in turn appear in the same relative position within the item as do the pointers.

Following the maps for the tree pointers are the maps which describe the primary variable field pointers in the order that these pointers appear in the item. Following these are the maps describing the primary-fixed length fields again in the order in which they appear. Following these, all of the secondary field maps in any order whatsoever. The order of the secondary field maps within the item map and the order of the secondary fields themselves within the item are completely independent. The order is implied by the back-pointers contained in the field maps.

A group of macros have been written for the Midas Assembly language to facilitate assembling item maps like the one shown in Figure IV-G-7. The macros are as follows:

OVERHEAD L — OVERHEAD is a macro of one argument and serves to set bits 14 through 17 in the item map. The argument, L is the length in words of the OVERHEAD in the item up to the pointer to the string area. This macro must be the first entry in a map.

TREE — The TREE macro has no arguments. It is used once per tree immediately following the OVERHEAD macro. As with any line of coding, it can be given a tag.

V — The V macro is used once per variable data field.

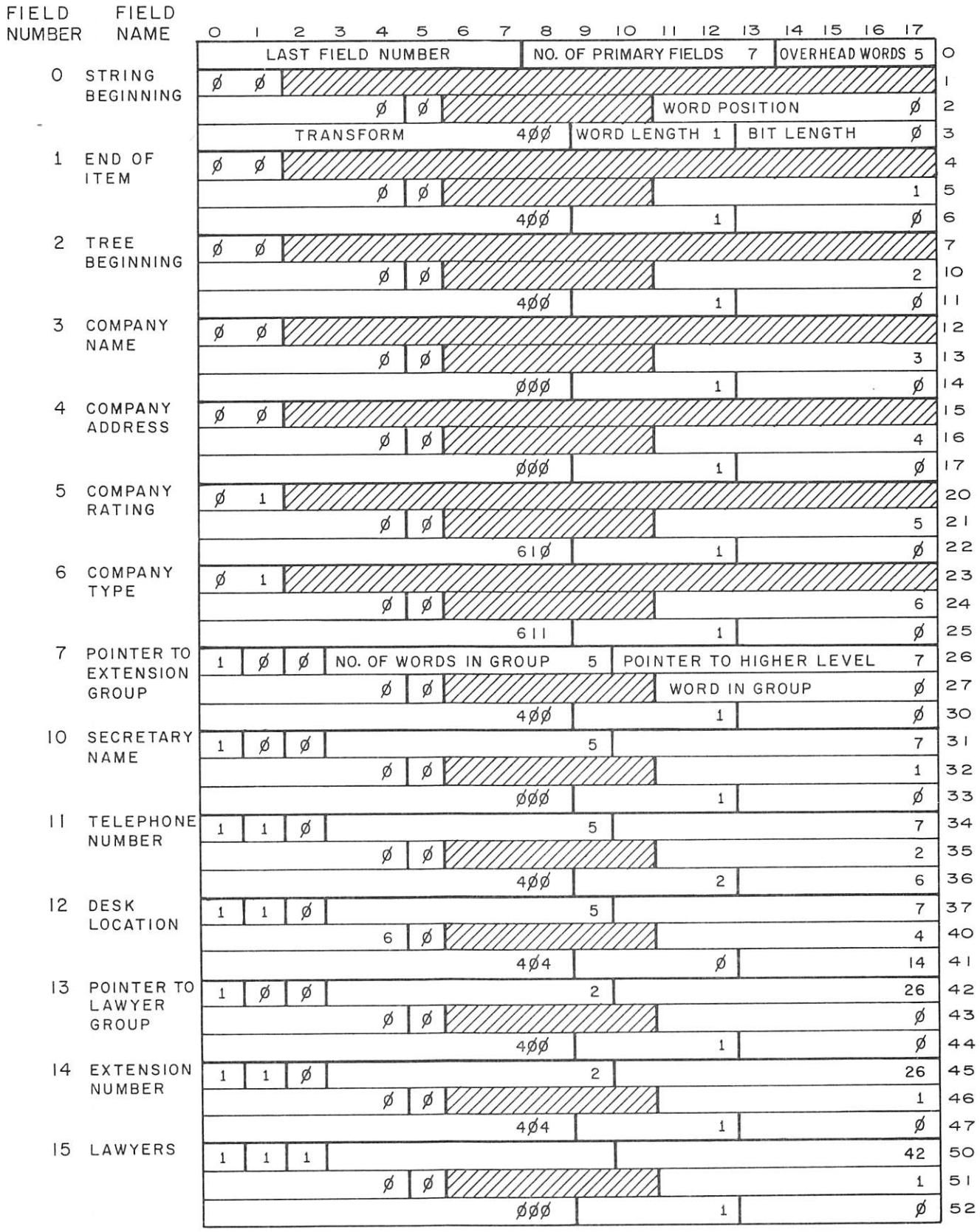


FIG. IV-G-7 ITEM MAP FOR "EXAMPLE ITEM"

F T, W, B — The F macro is used once for each fixed-length field and contains 3 arguments.

T is the transform code to be used with that field for translation purposes.

W is the field length in integer words.

B is the additional length in bits.

As mentioned previously, all new system maps should store fixed information in an integral number of words.

LEVEL — The LEVEL macro has no arguments and indicates a shift to the next lower level of a tree providing that level is not the last level of the tree.

LAST — The LAST macro has no arguments and indicates a shift to the lowest level of a given tree.

END — The END macro has no arguments and delineates the end of the item map.

The following is the MAP macro corresponding to the item as shown in Figure IV-G-7.

```
EXMAP,   OVERHEAD 5
         TREE          /there is one tree
CONAME,  V             /the company name
COADD,   V             /address
CORATE   F 21Ø, 1, Ø   /rating
COTYPE,  F 211, 1, Ø   /type
         LEVEL         /start of tree, level 1
SECNAM,  V
TELNO,   F Ø,2, 6
DESK,    F 4,Ø, 14
         LEVEL         /start of level 2
EXTNO,   F 4,1, Ø
         LAST          /start level 3, the lowest
LAWYER,  V
         END
```

Notice that the use of a MAP macro greatly simplifies the construction of the item map since the programmer needs know nothing now about the actual structure of the field maps themselves. Further, the use of such a macro permits the map itself to be given an address (in this case the address is EXMAP) and the fields themselves can be tagged for use in the GET and PUT IOT's.

GET 72462Ø The GET IOT, as might be expected from the complexity of the item is called with a fairly complex calling sequence. It is used to retrieve information from an item and provides valuable error diagnostics if improper requests are made for retrieval.

The GET IOT is executed with an address in the AC. This address is a word pointer to the start of a five-word table. The first word contains the starting address of a block of three error returns, the second word contains the relative address of the field map in the item map, the third word contains the address of the actual item in core, the fourth word contains the address of the item map in core and the fifth word contains the address of the subscript area where the block of subscripts to be used by GET in arriving at the proper field is located. There will be one such subscript entry for each level of dependence which must be searched to arrive at the field. Hence the block length depends on the level of the field being sought.

The three error returns are three consecutive addresses in core to which GET will return in case of a misrequest. If the first word of our five-word table points to the address "ERROR" then this is the address to which GET will return if the field number requested is too high for the item map. ERROR+1 is the location to which GET returns if a subscript higher than the highest present in that level in the actual item is requested. ERROR+2 is the location

to which GET returns if the request for information confused a pointer field with an information field. It is expected that the user program will contain JMPs at these locations to internal routines for handling such errors.

The second word of the five-word block contains a number which is 1,4,7, etc. This number is the relative address of the field map within the item map. It can be easily calculated since it is one plus three times the number of the field, where fields are numbered starting with zero.

The fourth word contains the address of the item map, in this case it would contain the address EXMAP. The fifth word contains a pointer to a consecutive set of registers containing the subscript numbers for each level which must be searched through to find the pertinent information. If we were looking for lawyer 2 in our example, the group with which he is associated is ØØ1 and hence the three registers would contain Ø,Ø, and 1 respectively. A typical string of coding to retrieve the name of lawyer 2 follows:

```
XX,     LAW A
        GET
          .
          .
A,      ERROR     /ERROR RETURN POINTER
        5Ø        /RELATIVE ADDRESS OF LAWYER FIELD
                   MAP (OCTAL)
        ITEM      /LOCATION OF ITEM IN CORE
        MAP       /MAP LOCATION
        SUBSCR    /SUBSCRIPT AREA
          .
          .
          .
          .
ERROR,  JMP ER1   /JUMPS TO ERROR ROUTINES
        JMP ER2
        JMP ER3
SUBSCR, Ø
        Ø
        1
        .
        .
        .
```

On a return from GET the AC contains a pointer to the requested information in core. It is a bit pointer if the information requested was fixed length information and it is a byte pointer if the information requested was variable length information. The I-O register contains

essentially the third word of the field map. Bit Ø is a Ø if the information requested is variable and 1 if the information is fixed, bits 1 through 8 specify the encoding of the field and bits 9 through 12 specify the length in whole words and bits 13 through 17 specify the remainder in bits. This information is sufficient for a complete decoding.

PUT 724640 Unlike GET, PUT has no error returns. All erroneous PUT commands are treated by the Executive as an illegal IOT. PUT is called with a pointer in the AC to the start of a five-word block. Further, since it is common to have the MAP macro assembled at the same time as the PUT routine, the five-word block can use symbolic information. The first word of the five-word block is a pointer to the location in core of the information to be put away. The second word is the symbolic tag for the field in the MAP macro. The third word is the location of the start of the item being assembled, and the fourth is the symbolic address of the start of the MAP macro. The fifth word is the location of the start of the subscripting area as in GET.

Any field to be put away by the PUT routine must, of course, be of proper length if it is a fixed length field and must be a string of characters terminated by an EOM symbol if it is a variable field.

BGI 7246ØØ BGI is the IOT used to initialize the item area. It provides the overhead and other necessary markers. BGI is called with the address of the start of the item to be generated in the I-O and with the address of the start of the map in the AC. BGI is called once and only once for each item to be made up and is generally the IOT immediately preceding the first PUT IOT in that routine.

In order to prevent erroneous information retrieval, BGI and PUT load the item area with the standard "information unknown" symbols for all items. For fixed fields, the field area is loaded with all ones and for variable fields, a simple EOM entry is made.

These IOT's and their calling sequence provide all of the necessary structuring to handle the filing logic of the system. They provide for retrieving information from items located in core and assembling information into items located in core. To complete the information storage and

retrieval system, it is necessary only to prepare an interface between those items and the bulk storage medium. The next section discusses that interface.

V. FILING ROUTINES

While the filing routines may be considered technically to be part of the common routines provided to the programmer by the Executive, we shall treat them as a separate group because of their central nature in any medical information system.

A. Drum Organization

The bulk memory, a Fastrand drum, has been organized into physically discreet sections both as a security measure and to permit different maintenance procedures to be followed with the various sections.

Consistent with this physical organization, the files themselves have been classified according to their use. These files are the Programmer files, the Research files, the Library files, the Message files, the Active Patient files, and the Index files. Each of these files is accessed by the movable heads on the Fastrand boom. In addition to these files, the Patient Parameter files are accessed by fixed heads on the Fastrand drum and will be considered as part of the Active Patient files. Each of these files has special IOT commands that reflect its particular special function. Overlaying the entire set of files, however, are a set of common filing instructions which may be used with many of the individual areas merely by changing the suffix in the IOT word. We will discuss these general commands first.

B. General Commands

In the following section, each of the IOT commands given is assumed to be applicable to the Programmer's, Research, Library, Message, and Active Patient Record tracks on the drum. In any case where the APR is not pertinent to such a track, the excluded section of the drum is designated by letter in parentheses following the IOT. In each case the IOT contains an X in the position that would ordinarily be occupied by a P, R, L, M, or A in order to designate the drum section on which the IOT is to operate.

When a filing IOT command is successfully carried out, the Executive returns control of the system to the user program at the second location after the IOT. However, any filing IOT may result in an abnormal condition. The Executive indicates unsuccessful execution of the command by returning control to the user program at the location after the IOT and by placing an error code in word 45 of the user's core. The following table lists the error code values and the causes of the abnormal conditions.

TABLE OF ABNORMAL RETURNS

ERROR CODE	CAUSE OF UNSUCCESSFUL EXECUTION
1	illegal drum address
2	no more blocks left in area
3	(rewrite) version number disagrees
4	iopmax < starting address on read
5	length is zero or greater than 1ØØØØ
6	unrecoverable hardware trouble
7	attempt to write on another programmer's quarter-track
1Ø	(rppu) no such unit number
11	no such nursing station or bed number
12	(write item) unit number disagrees with 4Ø and 41
13	(wita) illegal apr item class (like Ø)
14	ordinary data error--try again
15	(mag tape) read eof
16	(mag tape) read end-of-tape mark
17	unrecoverable tape controller error
2Ø	apr rewrite error (e.g., first or second words disagree)
21	attempting to expunge with wrong drum address

TABLE V-B-I

In all commands involving reads from the Fastrand into user core, the programmer may protect against excessively long reads by setting the value of IOP MAX to the highest address in the read buffer.

1. Physical Block Commands

The Fastrand drum is physically segmented into 51-word blocks. These blocks are further subdivided into a 1-word tag spatially separated from the 50-word body of information. This separation permits certain Executive commands to monitor the tag word before operating on the 50-word block of data without waiting for an entire drum revolution to take place. Except where otherwise noted, reference to a physical block in this report will mean only the 50-word data block and will specifically exclude the 1-word tag. The present Executive system (Exec II) does not distinguish absolutely between blocks that already contain useful data and those that do not. Such protection will be provided in the next version of the Executive system.

List of IOT Physical Block Commands

Mnemonic Symbol	Octal Value	Explanation
RABX	726XX1	Read Addressed Block. When RABX is executed, the contents of the AC are interpreted to be the core buffer address and the contents of the I-O to be the drum address.

The I-O Processor transfers the data from a single physical block to the buffer space designated in core. No check is made as to the contents of the block. This particular command is valid on all the tracks accessed by the movable heads.

WABX 726XXØ Write Addressed Block. WABX is probably the most dangerous command interpreted by the I-O Processor in that it transfers the 50 words from the buffer designated by the accumulator to the block whose address is contained in the I-O without regard to whether that block contains useful data at the time. No protection is provided. WABX is meant for use in system programs only and cannot be executed on the Library or APR tracks.

WNBX 726XX2 Write Non-Addressed Block. WABX transfers the 50 words of data starting at the location contained in the accumulator to the next physical block available on the drum free list in the section designated. It returns with the I-O containing the address where the block was written.

REBX	726XX3	Read and Expunge Block. REBX is similar to RABX. It simultaneously transfers the data from a 50-word data block and returns that data block to the free list. As pointed out earlier, it does so by modifying the tag word. As with all read commands, except where expressly noted, it is executed with the buffer core address in the AC and the specified drum address in the I-O.

The foregoing are all of the general commands that deal directly with the segments of the bulk storage into which the drum is physically divided. The next class of commands that we shall consider is the class which deals with a virtual file structure rather than a physical one. This is the class that deals with the concept of items as mentioned earlier.

2. Item Commands

While the item is a logical entity, it has been separated into three kinds of items in the interests of programming efficiency.

The first item we will consider is a standard chained item, which was discussed previously under Section III-F. A chained item consists of a physical set of blocks each containing a pointer to the previous block in the item. Reading such a chained item is relatively slow because the next block to be read by the Executive program is not uncovered until the previous block has been read. As a

result, reading such an item may require as many accesses as there are blocks.

The second type of item used is the TOC'd item in which the first block of the item contains the item length and a list, or Table of Contents, of all the physical addresses of the data blocks making up that item. For long items, the TOC'd format is preferable to the chained format in that the programmer for the Executive routine can read the TOC'd block and optimize the sequence of reading the remaining blocks in the item. With the further implementation of the scatter-gather capabilities of the system in future Executive systems, this kind of item will permit retrievals requiring only one full latency period for most items.

The third kind of item used in the system is, in reality, a subdivision of the TOC'd item. In order to provide protection for those programs that must rewrite items, a rewriteable item has been provided. The rewriteable item contains a version word, which is incremented by the Executive program as a protective device. To change a rewriteable item, the item is first read and then modified by the program and lastly rewritten. Because of the conflicts that arise in a time-sharing system, a second program may have accessed that same rewriteable item also for the purpose of rewriting it between the time that the first program read it and rewrote it. Were no protection offered, the first program would write its new version and the second program would then write its new version over that written by the first program. If the two programs had changed different portions of the item, the change introduced by the first program would be obliterated by the second program's rewriting.

The present Executive system provides protection as follows: When the first rewrite was done by the first program, the version word of the item would have been incremented by the Executive program. When the second rewrite was attempted by the second program, the version word contained in its core image would disagree with the version word contained on the drum and would result in an abnormal return with an error code 3 in word 45. At this point, an error routine in the second program should take over and update the modification in accordance with the information now contained in that item. Naturally, this modification is sometimes the result of user interaction, and such an update will then necessitate the reinitiation of the user program. However, this type of conflict is extremely infrequent, and programs are written to handle the case. All items filed on the library are filed in rewriteable item form although they can be written or rewritten only with operator intervention at the console.

The following IOT's are valid and defined for the Programmer, the Message, and the Research tracks only, with the exception of the last three, which deal with rewriteable format items that are defined and valid for the Library tracks also.

List of Item IOT Commands

Mnemonic Symbol	Octal Value	Explanation
WITX	726XX4	Write and Item. WITX is executed with the core address of the

information in the AC and returns control to the user after placing the physical drum address of the item into the user's I-O. The first word of the item must be the length of the item in words. Executive transfers that number of words from core to the drum in chained format in accordance with the first blocks available on the free list.

EXIX	726XX5	Expunge an Item. EXIX is essentially the inverse of WITX. EXIX returns the blocks of that item to the free list. EXIX is called with the drum address in the I-O.
RITX	726XX6	Read an Item. RITX reads an item from the drum into core starting at the address contained in the AC and terminating at the end of the item or at IOP MAX.
WTIX	726XX7	Write a TOC'd Item. The item to be written is located at the core address contained in the accumulator and the IOT returns with the drum address of the TOC in the I-O.
EXTX	726YYØ	Expunge a TOC'd Item. EXTX behaves exactly as EXIX.

RTIX	726YY1	Read a TOC'd Item. RTIX has the same restrictions as RITX. Note that in the interest of optimization a user program will frequently not use RTIX but will read only the TOC block of the item and will then read the pertinent block after that. Since the blocks are all equal known length, address arithemetic may be used and TOC'd items become useful for table storage.
WNRX	726YY2	Write Non-Addressed Rewriteable Item. Blocks needed are taken from Exec's free list and the return is made with the I-O containing the drum address of the TOC to the non-addressed rewriteable item.
RARX	726YY3	Read Addressed Rewriteable Item. RARX is analogous to RTIX except that it deals with rewriteable items. The first word of the item so read is the version number used for the rewrite protection previously discussed.
RWARX	726YY4	Rewrite Addressed Rewriteable Item. The first word of the item is used as the version number for comparison purposes. If the version number

in core and the version number on the drum match, the item is re-written. If the length of the new item is longer than the original version, additional blocks as necessary are secured from the free list, whereas if the item is shorter than the original version, excess blocks are returned to the free list.

The above commands are the general drum filing commands and are the only ones used on the Programmer's tracks and on the Message tracks. Additional commands as necessary for specific areas of the drum are discussed in the next section.

C. Special Commands

The following special-purpose commands are defined only for specific areas of the drum in order to implement features specific to the tasks associated with those areas.

1. Research Tracks

The research tracks are used for a generalized information storage and retrieval system that maintains its own storage organization protocol. Because of its internal requirements for indexing and its table minipulation, it has two special commands.

List of Special-Purpose Research IOT Commands

Mnemonic Symbol	Octal Value	Explanation
WATDR	726355	WATDR writes 3264 words on all 64 sectors under one head position on the Research tracks. Both the tag and the data words are written. The command is executed with a core address in the accumulator and with a 12.-bit drum address in the I-O specifying the "track" to be written on. No protection is provided by Exec during execution of this command.
RATDR	726356	RATDR is the inverse of WATDR and reads an entire ring of 3264 words into core at the address specified in the accumulator from the drum address specified by the first 12. bits of the I-O. This command and the one preceding it are particularly useful for dumping and restoring entire sections of the Research tracks onto tape for mass storage. They are also useful in building up internal indexes to the stored information and dealing with them quickly.
WFLR	726360	This command is not strictly a drum writing or reading command but

rather an organizational one. The Research programs maintain their own free list and manipulate their own blocks in accordance with that free list. In order to provide for permanence, the free list entry point is maintained in the Executive program and WFLR is the command used to set the free list to the address in the I-O.

RFLR 726361 RFLR is the command to read a free list pointer from one of the free lists maintained by Exec for the Research programs. It is called with the hi-order 6 bits of the I-O containing the hi-order 6 bits of a drum address and returns with the I-O containing the next available drum address on that ring. Both RFLR and WFLR have one return only.

2. Index Tracks

The Index tracks, which are used by the Index file and retrieve subroutines assembled together with user programs, carry a special internal substructure to permit the manipulation of 12-word blocks and the performance of address arithmetic. Index file and retrieve deal only with physical blocks on the Index tracks and maintain all protection and assignment control internally. The commands used are:

List of Index IOT Commands

Mnemonic Symbol	Octal Values	Explanation
WABI	726260	Write an addressed block.
RABI	726261	Read an addressed block.

The above commands are completely analogous to the addressed write and addressed read commands explained under V-B-1.

D. Active Patient Records

This section describes the IOT commands used in storing an retrieving information contained in the active patient records of the hospital. In the system described here, such information is stored in a particular section of the Fastrand drum, designated the Active Patient Record area.

To provide the user program with a convenient means of accessing a patient's record, the Executive program keeps an information block for every bed in the hospital. The blocks are stored on the Fastrand fixed heads and are organized as a linear file on which address arithmetic can be performed. The blocks are identified by a two-part number, called Bedspace, which contains the number of the Care Unit in the hospital and the number of the bed within the Care Unit. A user subroutine, ANYHOW, facilitates the conversion between the Bedspace number and the hospital's alphanumeric designation of a particular bed.

The information block (sometimes loosely called the "Parameters") contains the patient's unit number, the drum address of the patient identifier item, the drum address of the most recent item, and the drum addresses of the most recent items of all the possible 32 classes of patient information. Every item in a patient's record contains the drum address of the previous item and of the previous item of the same class. (See Figure V-D-1.) Therefore a user program may retrieve, in reverse chronological order, either all the items referring to a particular patient or just the items referring to one class of information about that patient.

To increase the safety and ease of access to the Active Patient Record area of the Fastrand drum, the Executive program provides the user program with automatic chaining of the data stored in a patient's record. When an item is written, the Executive adds the drum addresses of the previous items to the contents of the item being written and updates the "parameters" to refer to this item as the most recent one entered.

List of Active Patient Record IOT Commands

Mnemonic Symbol	Octal Value	Explanation
RPPB	726207	Read the Patient Parameters, given the octal Bedspace number. RPPB is called with a core address in the AC for a buffer area and the octal bedspace in word 42 of user core. It inserts the unit number into words 40 and 41 and reads 34 parameters into the buffer.

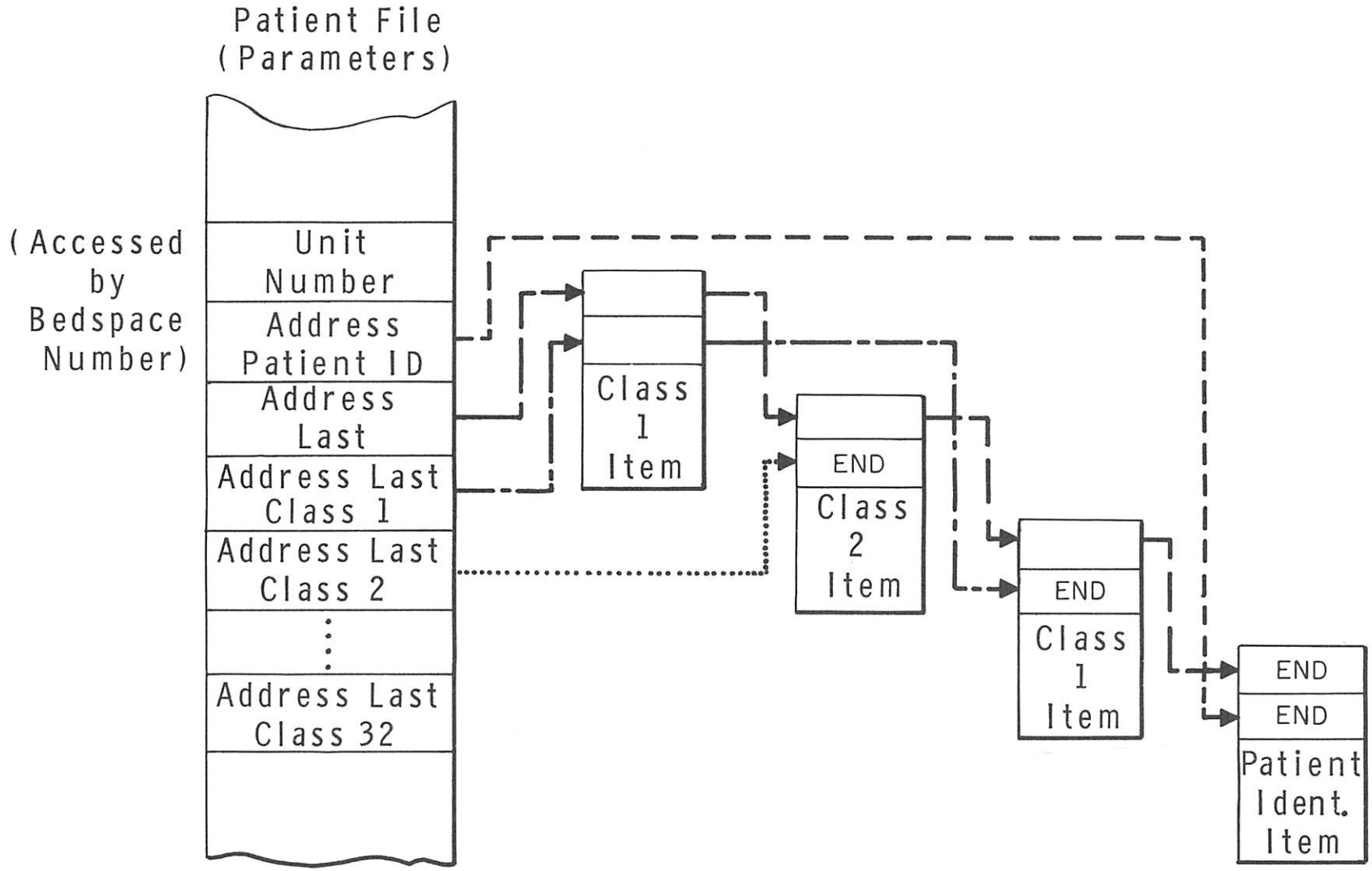


FIG. V-D-1 ACTIVE PATIENT RECORD

The 34 parameters represent the drum addresses of the Patient Identifier item, the last item written, and the last item of classes 1 through 32, inclusive.

RPPU 726210 Read the Patient Parameters, given the Unit number of the patient in the hospital.
The buffer address is located in the AC; the unit number is located in words 40 and 41. A valid read returns control to the second location after the IOT together with the octal value of the bedspace in word 42 and with the 34 parameters in the buffer. If no such unit number is in the hospital, the Executive returns control to the location after the IOT with a code 10 in the abnormal word (word 45).

EXPP 726215 Expunge Patient Parameters.
When a patient's record has been removed from the bulk storage, EXPP is executed with the bedspace in word 42 and the unit number in words 40 and 41. It returns control to the location after the IOT with an error code of 12 if the unit number differs from the unit number recorded for that bedspace.

BEDTR	726216	Bedspace Transfer. BEDTR is executed with the unit number and bedspace for the first patient in words 40, 41, and 42 and with the unit number and bedspace for the second patient in words 47, 50, and 51. If either unit number differs from that recorded for the bedspace, the command returns control to the location after the IOT with an error code of 12. If all values check, the unit numbers and address pointers are interchanged in the patient parameter file.
RABA	726201	Read Addressed block. This command is analogous to RABX.
WNBA	726202	Write nonaddressed block. This command is analogous to WNBX.
REBA	726203	Read and expunge block. This command is analogous to REBX (See Section V-B-1).
WITA	726204	Write an Item in the Active Patient Record files. WITA is the general item-writing command for those files. It is as close to a virtual file image system as we come in this version of the Executive program, since no

knowledge is required of the physical locations on the drum. The Executive program uses the WITA command to concatenate items for a patient.

The parameters of the bedspace in word 42 are updated to contain the drum address of the item as the most recent item of this class. The first two words of the item contain the drum addresses of the previous item and the previous item of this class. WITA is executed with the core address of the item in the AC and the returns with the drum address of the item in the I-O. The third word of the item contains the item class, type, and version. The 8th and 9th words are used by the Executive program to calculate the length of the item.

RITA 726200 Read an Item.
RITA is executed with the core address for the buffer area in the AC and the drum address of the item in the I-O. The drum address is secured, in general, by executing either an RPPB or an RPPU command followed by the necessary arithmetic.

Note that the presence of the RPPB and RPPU commands make the retention of absolute drum addresses in a user program unnecessary.

WPID	726211	Write Patient Identifier. WPID is the IOT command used to assign a patient in a hospital bed. It checks the bedspace to see if it is empty, writes the patient identifier item, and sets up the parameters.
WNRA	726212	Write Non-addressed Rewriteable item in the Active Patient Record area. WNRA returns with the address of the TOC in the I-O. It is called with the core address in the AC and the item length in the I-O.
RARA	726213	Read Addressed Rewriteable Item. It is called with the core address in the AC and the drum address of the TOC in the I-O.
RWARA	726214	Rewrite Addressed Rewriteable Item. Provides on the APR tracks the rewrite capability and protection described previously.

The foregoing three commands are used extensively on the APR tracks for the maintenance of small long-term dictionaries for such programs as the Laboratory and Formulary programs. Entry

to the dictionaries is made through a special bedspace (bedspace ∅) reserved for that purpose. The first item in bedspace ∅ contains a set of pointers pointing to various rewriteable items used by the dictionary program.

The preceding IOT's relate to the storage of information on the surface of the bulk random access storage drum. The information storage and retrieval programs provide some facility for recording on magnetic tape and hence for transferring data from the drum to the tape. While the detailed discipline of center organization and tape transfer has not been fully implemented, these tape IOT's are useable for the programmer in certain special applications.

E. Magnetic Tape Commands

The magnetic tape commands are designed to match the hardware of the Univac III-C tape drives and have error control codes suitable for those drives. The error codes are listed in Table V-B-1.

Since much of our tape use consists of absorbing tapes prepared on other machines in order to add information to our data base, IOT commands are provided for handling standard IBM format binary-coded decimal (BCD) records, containing 84 binary-coded decimal characters. The Executive program also provides IOT commands for handling 200_{10} word binary records.

List of Magnetic Tape IOT Commands

Mnemonic Symbol	Octal Value	Explanation
MRBCD	726000	Read BCD record. MRBCD is called with a core address in the AC and with the number of the tape unit in the I-O. It reads the first BCD block transferring it to core packed three characters per word.
MWBCD	726020	Write BCD record. MWBCD transfers 84 characters from core at the address contained in the AC to the tape on the unit whose number is contained in the I-O.
MBBCD	726140	Backspace one BCD record. MBBCD causes the tape on the unit whose number is contained in the I-O to backspace one block.
MWBIN	726040	Write Binary record. MWBIN writes a 200 word block from the address contained in the AC onto the tape mounted on the unit whose address is in the I-O.

MRBIN	726Ø6Ø	Read Binary record. MRBIN reads a 2ØØ word block into core at the address given in the AC from the unit whose address is in the I-O.
MBBIN	72612Ø	Backspace one Binary record. MBBIN causes the tape unit whose address is contained in the I-O to backspace one block.
MREW	72616Ø	Rewind. MREW rewinds the tape mounted on the unit whose number is contained in the I-O back to the load point thereby preventing its accidental restarting.
MWEOF	7261ØØ	Write End-of-File Mark. MWEOF may be used to separate files on tape and to mark the end of information on the tape.

The magnetic tape commands are used primarily by systems programs rather than by individual user programs. When used responsively via the programming language (See Volume Six E) they are used interpretively via the program Handle.