Introduction to

DECSYSTEM–20 Assembly Language Programming

Ralph E. Gorin

XKL, LLC

January 27, 2008

The following are trademarks of Digital Equipment Corporation: DDT, DEC, DECsystem–10, DECSYSTEM–20, DIGITAL, Massbus, PDP, and Unibus.

The drawings by Edward Koren appeared originally in *The New Yorker* ©1976, 1977, 1978, 1979, and 1980 by The New Yorker Magazine, Inc. It is unfortunate that these amusing drawings are not incorporated as part of the online work.

This manuscript was prepared using editing and text formatting facilities on DECsystem–10, DECSYSTEM–20, Xerox Alto, and Digital Equipment Corporation VAX computer systems. The final version was prepared using the LaTeX text formatting program and the Postscript document description language.

This manuscript is a revision of the work of the same title ©1981 by Digital Equipment Corporation. Previously published versions were produced using the SCRIBE text formatting program from Unilogic, Inc. on the computer facilities of Stanford University.

About the Author

Ralph E. Gorin is a computer scientist at XKL, LLC in Redmond, Washington, where he works on TOPS–20 development, systems architecture, and so forth.

He received his B.S. and M.E. degrees from Rensselaer Polytechnic Institute in 1970. He has been programming the DECsystem–10 in assembly language since 1969 and has been involved with the DECSYSTEM–20 since its debut in 1976.

During the period 1970–1992, Mr. Gorin held various positions at Stanford University. He has taught courses in computer science and electrical engineering. He was a developer of the WAITS timesharing system for the DECsystem–10 at the Stanford Artificial Intelligence Laboratory where he designed and coded the operating system support for several of the unusual peripheral devices attached to that system. In particular, he wrote the operating system support for the Xerox Graphics Printer and portions of the support for the Ethernet communications network. He invented the first computer spelling check and correction program for general text. He was Director of the Low Overhead Timesharing (LOTS), Stanford's academic computing center, and Director of the Computer Science Department's Computer Facilities. Later, he was Director of Academic Information Resources.

Mr. Gorin is one of the pioneers in the development and implementation of the local area computer network that linked together the many disparate computer systems at Stanford University. He promoted the development of a networked system of multiple microprocessors for instructional computing.

Since 1992, Mr. Gorin has been on the technical staff of XKL, LLC, where he commits computer science, computer systems architecture, programming, technical writing, and so forth.

Preface to the Third Edition

This as–yet unpublished edition integrates the study of extended addressing from the outset. This integration presently carries through chapter 26. The effort to integrate extended addressing has not yet touched chapters 27 and beyond.

The material in chapter 30 is probably redundant, as the intention was to make that material appear throughout the manuscript.

The material in chapters 31 and 32 is quite dated. Modern disks (chapter 31) are addressed by logical block numbers, not cylinder, head, sector. But the structure of the file system is not so different than what is described there. In the TOAD systems, IO is quite different from what is described in chapter 32.

R. E. G.
Redmond, Washington
December, 2004

Preface to the Second Edition

Since the publication of the first edition (1981), the situations in which assembly language programming is appropriate have been markedly reduced by improvements in compiler technologies, and the expressiveness of languages such as C and its object–oriented variants.

Nevertheless, there are still reasons to study assembly language, even if there are fewer appropriate applications for that knowledge. Computer systems architects should have a good understanding of the instruction set design: the tradeoffs between expressiveness, static space requirements, and dynamic reference patterns of programs. Such issues can best be studied from the basis of assembly language representation of programs. Compiler writers must also understand the machine operations of their intended target machine. Applications programmers may wish to take advantage of data structures and information representations that are not supported by compilers. Finally, in operating systems, some portions of the kernal, usually those involved with pager, cache, memory and device management, may be coded in assembly language, as conventional applications languages seldom have the mechanisms needed to cope with the particular details of computer system hardware that are prevalent in these corners of an operating system.

The second edition corrects errors in first edition. Some examples have been changed to avoid practices that are obsolete in the extended address space. This edition documents the "Giant" floating–point numbers that were added to later models of the DECsystem–2060. Also, some of the instructions added under the `EXTEND` operation are described.

Had time permitted, this edition would have introduced extended addressing at a much earlier point than Chapter 30. Extended addressing is commonplace in new programs and in the adaptations of old ones. Also, the methods described in Chapter 30 are considerably more awkward than those currently practiced. A reader who contemplates the construction of a new program should become familiar with the `.PSECT` operator in MACRO and the corresponding commands to LINK. A reader may tackle chapter 30 at any point after Chapter 17.

The second edition omits mention of the "String Instructions" that were introduced in the KL10. These are documented in the later editions of the hardware reference manual.

Finally, a note about the philosophy of the examples given in this book: most examples given are complete, working programs, as such they tend to be long. Further, the new components of each example are explained thoroughly, sometimes from multiple points of view. Some readers may find this tedious, others may be grateful for the alternative explanations. I hope you find it helpful.

The author is grateful to the staff and the managing member of XKL, LLC for their support of the second edition.

R. E. G.
Redmond, Washington
February, 2003

Preface to the First Edition

Assembly language programs can obtain access to the full power of the particular computer system for which they are written. Problem–oriented, high–level languages do not generally provide for this flexibility. Assembly language programs are used where high efficiencies are required, especially in areas of programming that do not yet have a problem–oriented language. So long as there are students of programming who are enthusiastic and inquiring, there will be a desire to know assembly languages.

Although it might be possible to teach assembly language programming for an abstract machine, we believe that a student's first exposure to assembly language should be coupled to some specific computer. Here we relate the study of assembly language to the Digital Equipment Corporation DECSYSTEM–20. The DECSYSTEM–20 has been selected for several reasons. It is gaining widespread acceptance in academic and commercial environments. For the student of assembly language programming, the DECSYSTEM–20 provides a timesharing environment in which extensive editing and debugging facilities are present; assembly language can be taught without the need for stand–alone systems. Finally, the instruction set characteristics of the PDP–10 central processor make the DECSYSTEM–20 an outstanding pedagogical vehicle.

The aim of this text is to provide a thorough treatment of assembly language programming for the DECSYSTEM–20, emphasizing the analysis of programs and various methods of program synthesis. This text presents the detailed structure of the DECSYSTEM–20 instruction set, explains assembly language programming, and demonstrates useful application techniques. The diligent reader will be able to use assembly language to write new programs and modify existing ones. The reader will also develop an understanding of how programs, the operating system, and the computer hardware interact.

The material here is an extension of a 30–hour lecture course at Stanford. At Stanford, we recommend that a student take courses in introductory and intermediate programming before studying an assembly language; such a background provides a framework for disciplined programming. The understanding gained by learning to program in one of the high–level languages is essential. The student using this text should be able to analyze problems and develop algorithms (i.e., procedures) to solve them. He or she should be familiar with the control structures and data structures available in a language such as Pascal. Also, he or she should be able to use the TOPS–20 system at least to the extent of editing files and using the `EXECUTE` command.

Assembly languages are not easy to teach because of the plethora of detail involved. We have chosen examples that gradually reveal the structure of the machine, the assembler, and the operating system. We will describe a number of common programming tasks and introduce various practical solutions. Thus, in addition to learning assembly language, the student will learn useful algorithms and techniques. Among these are sorting, hash–code lookup, lists, command processing, and some lexical analysis. We have included explanations to allow the student to make full use of

the input/output facilities provided by the TOPS–20 operating system.

This text deals primarily with those machine instructions that are useful when writing application programs. Many advanced operating system and assembler features are not mentioned, but the material presented here should be an adequate foundation for further individual study. This text is not intended to replace the reference manuals for the central processor [SYSREF], the JSYS calls [MCRM] and [MCUG], and the MACRO assembler [MACRO]. [1] Rather, those references should be called upon as a supplement when necessary.

This text is divided into three major areas:

- Chapters 1 through 18 are primarily a presentation of the machine instruction set and the assembler. Some operating system calls that deal with terminal input and output are presented also, but the primary focus is on the instruction set. The processor and memory, various representations of data, the purpose and function of the assembler, and the effective address calculation procedure are included in the topics.
- Chapters 19 through 29 present several very useful aspects of the operating system. In this portion we concentrate on interesting applications and programming techniques. Arrays, sorting algorithms, list structures, file input and output techniques, command parsing, processes and interrupts are among the topics covered.
- Chapters 30 through 32 concentrate on advanced topics: extended addressing, file system organization, and programming actual input and output devices.

As a whole, this text presents assembly language programming from the viewpoint of a systems programmer, i.e., a person interested in implementing utility programs, high–level languages, and particularly efficient applications.

Chapter 1 presents an overview of the three essential topics of interest: the central processor, the assembler, and the operating system. The interactive debugging facilities available in the DECSYSTEM–20 are mentioned.

A view of the computer system as a central processor and memory is introduced in Chapter 2. Memory is viewed as an array of words, each with a unique address. The concept of data and instructions as objects that are held in memory is presented. The execution of simple instruction sequences is explained.

A complete example of a small assembly language program appears in Chapter 3. The example is thoroughly explained, as it forms the foundation of all future examples. Three common system calls, including one for string output to the terminal, are introduced. The minimal set of assembler functions that are necessary for any program are explained; some of the most frequently used pseudo–operators are discussed.

Chapter 4 discusses the representation of data inside the computer. Binary and octal notation are explained. Fixed–point binary, two's complement arithmetic, and conversion between radices are demonstrated. The ASCII character code and its application in the PDP–10 are explained.

The format of instructions in memory, the meaning of the various instruction fields, and the nature of the translation effected by the assembler, are discussed in Chapter 5. Every instruction computes an effective address; the effective address computation and several examples are presented.

The most basic and most useful PDP–10 instructions are introduced in Chapter 6. Among these instructions are fullword manipulations, jumps, conditional jumps, and conditional skips. A set of examples illustrates the use of these instructions.

[1] See Appendix F, page 663 for references.

We introduce a system call that performs input from the terminal in Chapter 7. Further assembler features, additional pseudo–operators, and the literal facility are discussed.

The output of the assembler, the binary and listing files, are explained in Chapter 8. A brief discussion of the effects of the linking loader is included here.

Chapter 9 demonstrates the symbolic debugger, DDT, showing how it can be used to locate problems inside a program.

Pushdown stacks are presented in Chapter 10. The relevant instructions and pseudo–ops are shown. A simple application of a stack for reversing an input stream is demonstrated; the system calls that perform single–character input and output using the terminal are shown.

Chapter 11 introduces the byte instructions and the `POINT` pseudo–op. The byte instructions in the PDP–10 are very useful for string processing and for copying the contents of selected fields from memory to the accumulators. This chapter displays several examples of their use.

The halfword instructions are explained in Chapter 12. In the PDP–10, data is often found packed with one item in each halfword. The items may be addresses that describe data structures, or other 18–bit items. The halfword instructions are useful for manipulating such data items.

The details of the program counter, its associated flags, and the various subroutine calling instructions are introduced in Chapter 13. Examples to demonstrate the application of the different subroutine calls are given. The `PUSHJ` and `POPJ` instructions are discussed in detail; an example is shown in which a program is structured into manageable subroutines.

The Test instructions and the Boolean instructions are presented in Chapter 14. The same process performed by the example in Chapter 13 is re–programmed to take advantage of these instructions.

The Block Transfer instruction and the shift instructions are explained, with short sample applications, in Chapter 15.

Chapter 16 explains the integer and floating–point arithmetic instructions. The representation of floating–point numbers and the accuracy of floating–point arithmetic are discussed.

The concepts of macros and conditional assembly are introduced in Chapter 17. An example is presented that demonstrates these topics and integer arithmetic operations.

Local UUOs are presented in Chapter 18. An example of LUUOs and of floating–point arithmetic is given. This chapter 18 generally concludes the presentation of the instruction set, excepting only extended addressing; beyond this point, the text deals mostly with applications and additional system features.

An overview of the operating system functions that are so important in real programs appears in Chapter 19. These topics include input and output processing; process creation, control, and communication; interrupts; and command scanning. The chapters that follow expand on these topics and apply the instruction set to more complex data structures and examples.

A very simple first example in Chapter 20 demonstrates the system calls that effect file output. A second example presents a technique for managing an output buffer; the example also shows how to use the arithmetic instructions to perform extended precision calculations.

Chapter 21 describes arrays. One–dimensional arrays are introduced; several examples are given. The use of an index register to access array elements is demonstrated. An example is given in which an array is used to hold the digits involved in a calculation of large factorials. The discussion continues to two–dimensional arrays; two accessing techniques, address polynomials, and the use of indirect addressing via side–tables are demonstrated. An application of two–dimensional arrays to plotting is shown. The discussion of arrays ends with an extension of address polynomials to higher

dimensions.

File input operations are presented in Chapter 22. For file input the programmer must also cope with the end of file condition. A moderately useful file search program is developed as an example.

File directories and the `GNJFN` JSYS are discussed in Chapter 23. The example presents a simple technique for dynamic space allocation and demonstrates a Bubble sort. Dynamic and flexible allocation of memory space is one of the particular attractions of assembly language programming; many high–level languages do not provide adequate tools for storage management. Bubble sort is presented as the simplest of sorting algorithms. However, Bubble sort is inefficient, so the Heapsort algorithm is explained and a subroutine to implement Heapsort is developed to replace the Bubble sort subroutine.

Chapter 24 introduces records as a data structure and linked lists as a technique for organizing records. An interesting program presents an example of list processing, hash–code search techniques, the use of buffers to reduce the overhead of input and output operations, and an efficient list–oriented merge sort.

The most efficient method for doing disk input and output in TOPS–20, page mapping, is explained in Chapter 25. Also, random access I/O is introduced.

Command scanning is one of the areas handled least well by high–level languages, yet good human interfaces are necessary in any interactive program. Remarkably good command processors can be fashioned in assembly language programs. Chapter 26 contains an extensive discussion of the command scanning techniques appropriate in TOPS–20.

Chapter 27 discusses the process structure of jobs within TOPS–20. An example is given to show how new processes can be created and controlled.

Communication between separate processes is the subject of Chapter 28. The TOPS–20 Interprocess Communication Facility is demonstrated. A second technique, the use of shared, mapped file pages is also discussed.

Two more program control mechanisms, the trap and software interrupt facilities, are explored in Chapter 29. In the first example we demonstrate how to trap and process arithmetic exceptions; we also show the utility of co–routines as a control technique. The interrupt facility and an interesting application of TOPS–20 process structure are displayed in the second example. That example also discusses the format of the symbol table that is created by the assembler and the loader and that is used by DDT.

Chapter 30 presents the extended addressing features of the DECSYSTEM–2060. By means of extended addressing, the size of a single program's virtual address space can be extended to 8 million words in the 2060, and to one billion words in future machines. In extended addressing, the method of effective address calculation is more complex than the rules explained in Chapter 5. The new rules are presented. A short example subroutine is presented; this routine moves code and data from section zero to section one so that the remainder of the program can use extended addressing.

How TOPS–20 imposes a file structure on the disk is the subject of Chapter 31. The representation of files, directory information, and the maintenance of an accurate free space list are discussed.

No text on assembly language programming would be complete without some discussion of the machine instructions that affect input and output devices. The machine–level input and output facilities, the priority interrupt system, and data channel operation are discussed in Chapter 32.

The author wishes to thank the staff and management of the Stanford Artificial Intelligence Laboratory, on whose word processing facilities the early drafts of this manuscript were prepared. J. Q. Johnson patiently read several drafts and refused to allow me to leave poor enough alone. My thanks

also to the students who tolerated the earlier versions of this manuscript and who made corrections and many useful suggestions; Joe Weening found many of the errors in the penultimate draft. Mark Crispin wrote the first version of the Small Executive program, the example of the `COMND` JSYS, and made many helpful suggestions. Frank da Cruz of Columbia University organized an experiment in teaching from this manuscript; several of his suggestions resulting from that experience have been incorporated here. Peter Hurley of Digital Equipment Corporation contributed several ideas about writing more efficient programs in the TOPS–20 environment; these have been incorporated at those places in the text where they fit most appropriately.

R. E. G.
Stanford, California
June, 1981

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Assembly language programming is the way to get close to a computer, to know the precise details of its functioning. The computer is an obedient servant; assembly language provides precise and explicit control over the implementation and execution of programs. Unlike high-level languages, assembly languages allow access to a broad range of control techniques and data structures.

Programming requires clear thinking and attention to detail. Assembly language programming calls for practicing these skills to a particularly high degree. Assembly languages exact payment for the exercise of greater control; three, eight, or dozens of instructions may be needed to implement each high-level construct. In assembly language programming there is an inescapable tendency towards long programs. Composing and debugging a long program need not be difficult if approached properly. We will discuss ways to manage such tasks.

Among the benefits of assembly language programming is the ability to use all of the hardware and operating system features provided by the computer system. Assembly language programmers are not restricted to the features, control structures, data structures, and input/output facilities provided by any particular high-level language.

Assembly language is presently most suitable where the manpower expended in producing a program is expected to be small compared to the expense of running the program. In some cases, no other language will execute the program in a short enough time, or with so little expenditure of machine resources. The current activity in microprocessor-based systems has spurred a new interest in assembly language programming; assembly languages can get the job done in the minimum amount of hardware, a very important consideration in any situation where systems are being mass produced. However, as our understanding of optimizing compilers increases, as hardware becomes faster and cheaper, there will be fewer situations that require new programs in assembly language.

A vast number of programs have been written in assembly language. Often, these need to be modified; usually a patch is a more effective short-term solution than an elegant rewrite.

Assembly language skills are needed by the people who implement compilers, data base systems, and other application packages.

Knowledge of a variety of computer systems at the assembly language level is useful for understanding the issues of machine architecture and implementations. The most useful computers are the ones that have been designed with a conscious understanding of the problems of programming.

For these reasons, and perhaps for other aesthetic ones, people continue to study assembly language. Some find it fascinating.

Every computer implements a collection of primitive functions called *instructions*. *Programs* consist of sequences of these instructions. Historically, the first computers were programmed in *machine language*, in which programs were constructed by hand, literally bit by bit. When a particular primitive function, e.g., addition, is desired, the binary pattern corresponding to the `ADD` instruction is found in the instruction manual and copied by hand into the computer's memory. This description suggests that programming in machine language is exceedingly tedious. It is. Also, it is quite susceptible to errors.

Assembly language represents a significant advance over the tedium and uncertain results of machine language. Essentially the process of *looking up* the binary pattern for each instruction has been automated. The combination of the *assembler* and *loader* programs handle some of the details of allocating space in memory for the program and variables; together, these programs perform useful bookkeeping chores. Note, however, that we continue to deal with the primitive functions, instructions, that the computer itself implements.

We will study four principal aspects of assembly language programming:

- writing correct, understandable algorithms,
- proper use of the hardware instruction set,
- proper use of the software instruction set, and
- use of software aids such as the assembler, debugger, and loader.

To program in assembly language, it is necessary to learn something about each of these topics. So, we begin, a little at a time, to show the "tip of the iceberg" for each of these subjects.

1.1 Algorithms

The techniques of programming in high-level languages are relatively easily carried over into assembly language. We shall have occasion to demonstrate *algorithms*—computational processes—in both a high-level language and in assembly language.

Programming requires that problems be divided into subproblems. Divide and conquer is the most consistently successful strategy for program development.

In assembly language, the primitive operations are small. It takes several machine instructions to implement each high-level construction. Because programs in assembly language are usually longer than those in a high-level language, it is especially important to develop good habits regarding the structure and documentation of the programs we write.

These characteristics of structure and documentation are referred to as *programming style*. Style is important; a correctly functioning program is a necessary but not sufficient achievement. Beyond correct performance, the programs we write must be understandable by others. Proper style and documentation enhance a reader's understanding of the program. More than any other comparable human endeavor, programs exist to be changed. Well-commented, well-structured programs are easier to maintain and modify.

1.2 Machine Instructions

The *machine instructions* (or, *hardware instructions*) are the primitive operations with which we write programs. Learning the *instruction set* means learning what operations are performed by each

of the commonly used instructions. Programming is the art or science of combining these operations to accomplish some particular task. We'll give examples of what we hope are correct programs and useful techniques.

Learning the instruction set does require some rote memorization. As we discuss the instruction set, we will try to establish patterns to help you organize your thinking about the instructions.

1.3 Operating System - The Software Instruction Set

In most computer systems a special-purpose program called an *operating system* is used to manage the computer and to help programs perform input and output operations. Operating systems may also provide useful extensions to the instruction set. For example, if a machine doesn't implement multiply and divide instructions, the operating system might provide routines to simulate these. The operating system may redefine or extend the instruction set that the hardware implements. The operating system in the DECSYSTEM–20 is called TOPS–20.

When a computer such as the DECSYSTEM–20 is shared among many simultaneous users, the operating system separates users to prevent one user's mistakes from affecting any other users. For its own protection and the protection of other users the TOPS–20 operating system places various restrictions on the programs that it runs. These restrictions are implemented by running all programs (except for TOPS–20 itself) in *user mode*. In user mode, programs are restricted to memory assigned to them by TOPS–20; they may not perform any machine input/output instructions, nor can they perform certain other restricted operations (e.g., the `HALT` instruction). Editors, assemblers, compilers, utilities, and programs that you write yourself are all user mode programs.

To perform input and output operations, a program must request these functions from the operating system. Even a high-level language such as Fortran or Cobol must request these functions, although a user of such a language is usually not aware of the details of these operations. TOPS–20 provides various subroutines (accessed via the JSYS operation) by which a user program can communicate its wishes to the system. We shall have more to say about operating systems in Section 19, page 277.

1.4 The Assembler

Theoretically, understanding of the machine instructions alone is sufficient in order to program the computer. However, since these instructions are binary quantities and because programs are complex, a translation program, called an *assembler*, is available. The assembler program translates mnemonic instruction names and symbolic addresses into the binary quantities that the computer acts on. The assembler is a simple translation and bookkeeping device that relieves the programmer of a number of non-productive chores. Use of the assembler makes programming the computer easier and more convenient than if the only interface to the computer were binary. The translation of a program is written into an *object file*, a collection of binary data for the loader. In the DECSYSTEM–20 the assembler is called MACRO; the object file is sometimes called a "relocatable" or "`REL`" file by the type name that it bears. indexREL File

1.5 The Loader

The loader is responsible for creating a memory image (sometimes called a "core image") of the program from the object file created by the assembler. To create the memory image, the loader

chooses a virtual address (or it is told an address) at which to put the instructions found in the object file. Additionally, when a memory image is composed from several object files, as would be the case in a complex program, the loader connects the modules together. In the DECSYSTEM–20, the loader is called LINK.

1.6 Debugging Aids

The DECSYSTEM–20 has a very powerful debugging aid called DDT. The name "DDT" stands for "Dynamic Debugging Technique" and refers to a program used to get rid of a class of program errors, called *bugs*, that are impervious to dichloro-diphenyl-trichloroethane. By using DDT, a programmer can examine and change the contents of memory, either data or instructions. The programmer can use DDT to place breakpoints, single-step, and otherwise control the execution of the program that is being debugged. This form of debugging is unique to interactive computer systems. DDT is discussed further in Section 9, page 101.

In other computer systems, instead of an interactive debugger, a programmer may be limited to *core dumps* as the sole debugging tool. The core dump is a lengthy listing of the contents of main memory at a specified snapshot point or at an abnormal termination of the program. These listings are difficult to work with. Minicomputer systems may have some machine language debugging techniques that are nearly equivalent to hardware console switches and lights.

Chapter 2

Hardware Overview

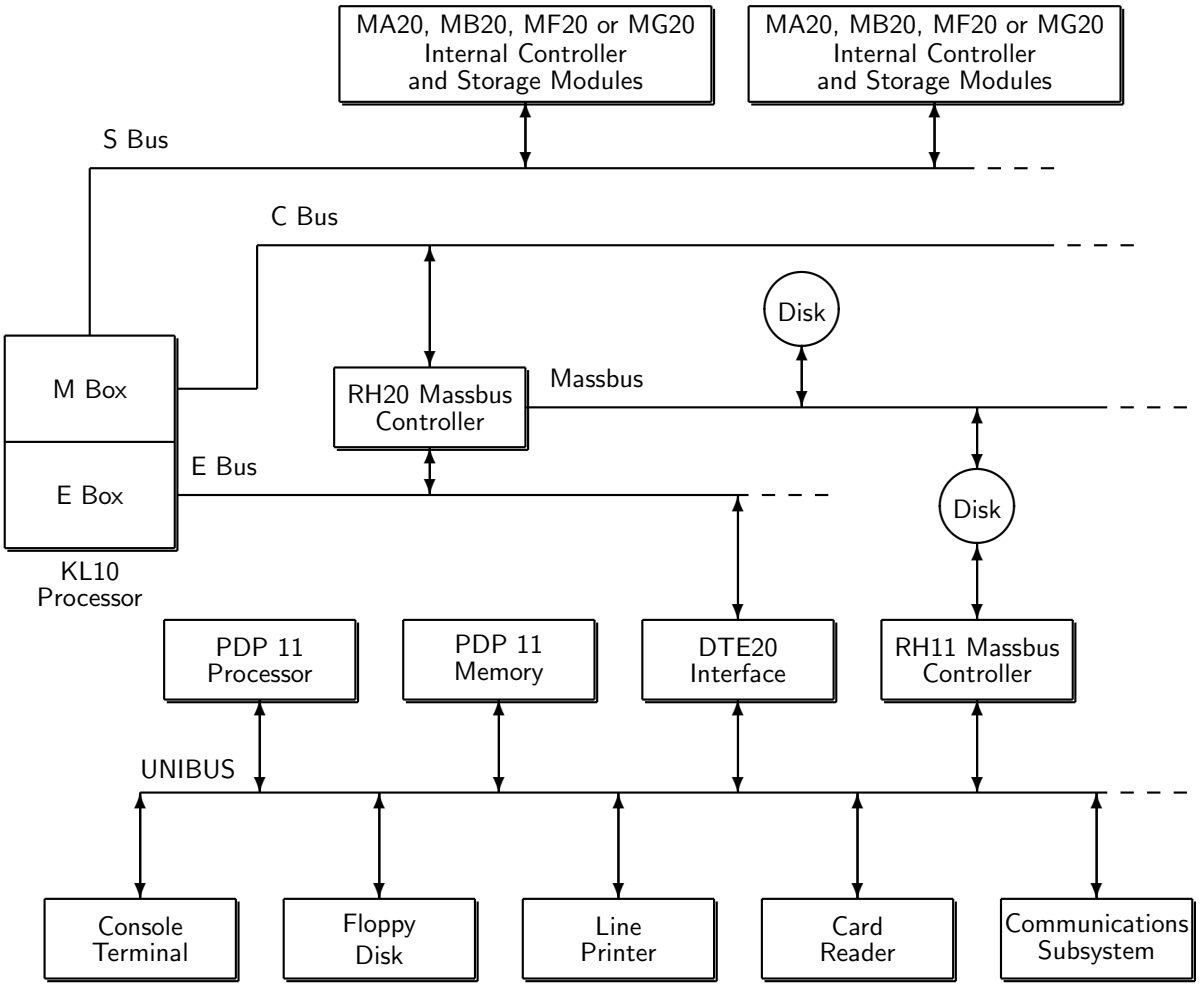


Figure 2.1: KL10-based DECSYSTEM–20 Configuration

A computer system is a whole composed of many parts. The DECSYSTEM–20 includes a central processor, a memory, secondary storage on disks, terminals, printers, and tape drives. A sample configuration is depicted in Figure 2.1. However, since an actual configuration is too complex to be our starting point, we shall begin by modeling the computer system as just a memory and a central processing unit as shown in Figure 2.2.

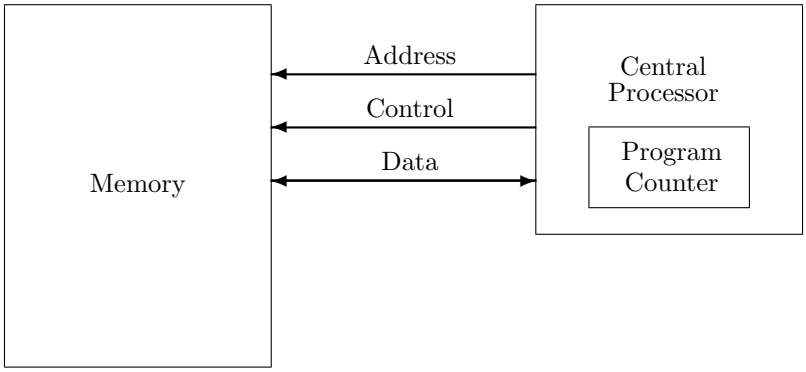


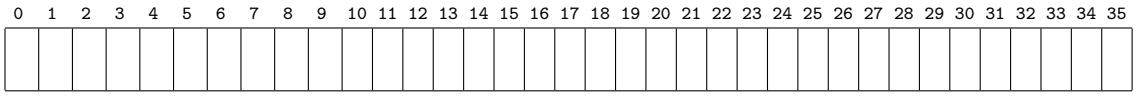
Figure 2.2: Processor and Memory Configuration

The memory stores and retrieves data under the control of the central processor. The central processor provides the arithmetic and logic functions in the computer system. The central processor includes a *program counter* that proceeds sequentially through the running program.

2.1 The Memory

The *memory* stores information for later retrieval. Memory is organized as an array of items called *words*. Every word contains `36` binary digits (called *bits*) that store information; each bit stores either a `0` or a `1`. Computer memories are organized as collections of objects that can take on either of just two states because of the fundamental engineering princple that distinguishing between two states of a component (e.g., on–off, charged–discharged, etc.) is easier (faster, more reliable) than distinguishing between three states, ten states, or any other number of states.

In the DECsystem–20, the bits in each word are numbered from left to right from `0` to `35`:



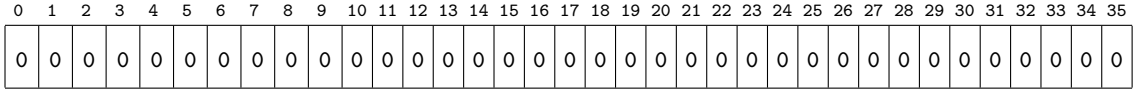
Fundamentally, the central processor can store (write) data into specific memory words and subsequently retrieve (read) that data. Any information that is contained within the computer is represented by patterns of ones and zeros that are stored in these words.

2.1.1 Data in Memory

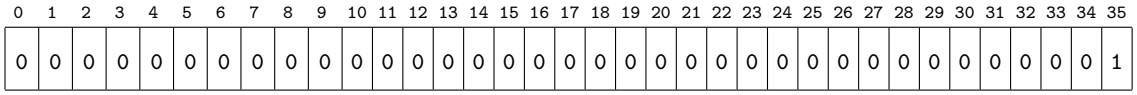
Though many different kinds of information are stored inside the computer, we often think of the computer as especially well suited for arithmetical computations. Naturally, in such cases the data

stored in the computer include numbers. Some example data words appear below. If you are unfamiliar with the binary (base two) notation, these patterns may appear to be quite meaningless; please bear with us until we arrive at Section 4.2, page 30.

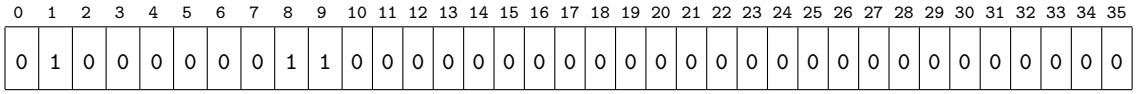
The following pattern represents a quite common number, `0`:



Another familiar number is the integer `1`:

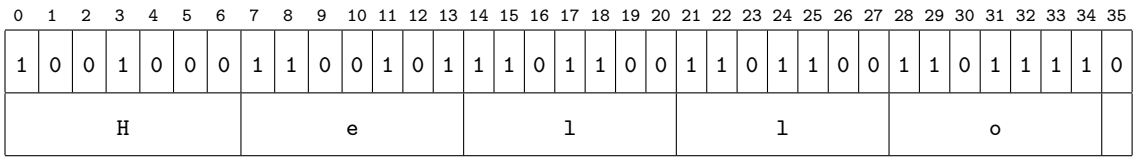


High-level computer languages such as Fortran or Pascal distinguish between integers and real numbers. In many computers the same distinction is drawn in the representation of such numbers. Although the integer `1` is represented with a simple pattern, the real (or *floating-point*) number `1.0` has a more complex pattern:



Even if you understand binary notation, this pattern isn't expected to make much sense until we discuss the details of floating-point representations in Section 16.5, page 222.

Among the other pieces of information you might find inside a computer is text. Programs that you write are stored and edited as characters before they are translated into a runnable form. Other text files appear in computers: your mail file, help files, and the manuscript of this book are all examples of text files. The following depicts a word containing the five letters "Hello":



In Section 4.8.1, page 39, we will discuss just how this representation for text comes about.

As we have mentioned, the programs that are run by the central processor consist of primitive operations called instructions. Each instruction is an item of data in memory. Although we are not quite ready to explain what the different instructions mean, it is instructive to look at one as it would appear in memory. As an example instruction, we have selected `MOVEI 10,0`. This instruction tells the computer to copy the constant `0` to location `10`. The instruction would appear in a computer word that looks like this:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Please scrutinize this pattern carefully. Note that it is precisely the same as the pattern that we previously identified as the real number `1.0`. These two examples were selected to drive home this important point: the memory stores only binary patterns; people, and the programs that people write, supply the interpretation of each pattern.

If this pattern were executed by the computer as an instruction, it would be the `MOVEI 10,0` instruction as we have described. If this pattern were used as an operand in a floating-point instruction, it would represent the number `1.0`. Indeed this pattern also has an interpretation as an integer (it happens to be `17381195776`) and an interpretation as text (the two characters blank and zero, followed by three null characters).

Again, the idea is this: the memory does not distinguish among the varieties of data stored within it. The program supplies the interpretation.

2.1.2 Addresses in Memory

Every word in memory has a unique *address*. The address of a word names the location or place where we can find that word. As with most other things in the computer, an address is a number. A program that runs in the traditional DECSYSTEM–20 sees a memory space that contains addresses in the range from `0` to `262,143`. We will often use *octal* (i.e., radix `8`) notation (to be explained in Section 4.2, page 30) when we talk about addresses and the contents of memory words. In octal, the range of traditional addresses is from `0` to `777777`, as shown in Figure 2.3. This figure also introduces a convention that will be followed throughout this book: smaller address numbers appear near the top of the page and larger addresses are near the bottom.

The word "traditional" appeared twice in the previous paragraph. The original PDP–10 had an address space of about a quarter million (2^{18}) words. By the late 1970s, people understood that a computer must handle large programs and larger volumes of data. Engineers at Digital Equipment invented a scheme to extend the address space to about a billion (2^{30}) words, thought of as 4096 *sections*, each section being 2^{18} words. In the DECSYSTEMs models 2060 and 2065 they implemented an address space of about 8 million (2^{23}) words, i.e., 32 sections. The billion–word address space has been implemented in the XKL TOAD–1 systems. The Extended Addressing model of the address space is depicted in Figure 2.4; it shows the address space divided into sections.

In the DECSYSTEM–20 the memory space seen by a program is spoken of as a *virtual memory*, as distinct from the actual *physical memory*, because the operating system creates the appearance of this memory space from fragments of real memory.

In order to provide both compatibility with old programs and allow for new programs to have large address spaces, the extended addressing computer may be thought of as two computers in the same box: the traditional computer knows only about section 0; the extended computer knows only about the sections above section 0, called non–zero sections.[1] There are ways for the programmer to force the computer to transition from one regime to the other.

The computer behaves differently in section 0 than in non–zero sections. For a while, programmers were unsure how to use this combination. Now, most programmers have settled on a simple rule.

[1] This model is overly simplistic. The awful truth is revealed in Section 5.3, page 45.

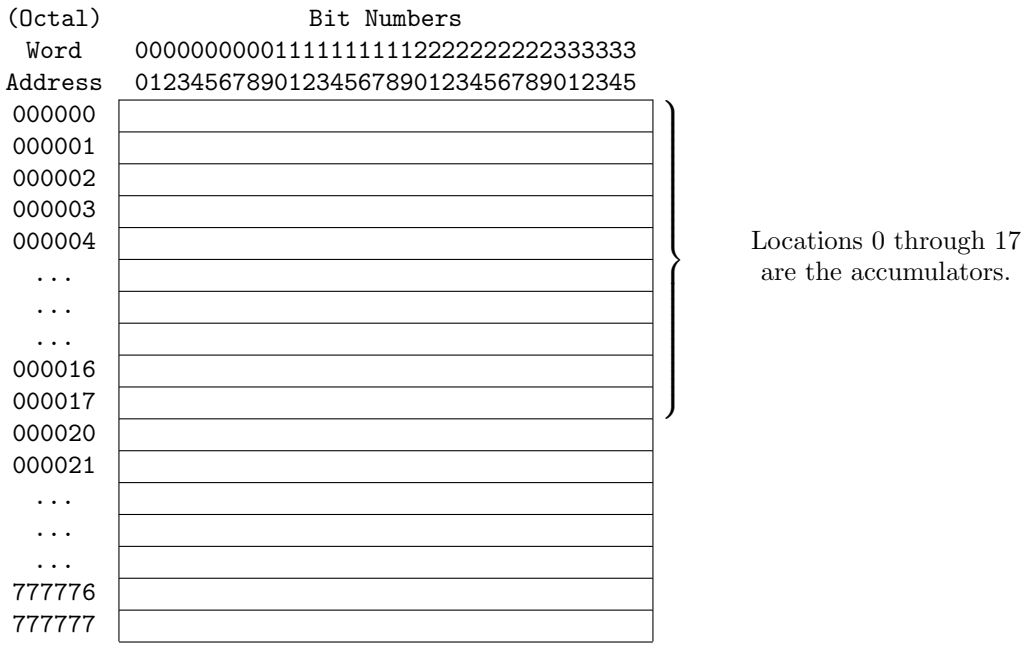


Figure 2.3: DECSYSTEM–20 Virtual Memory

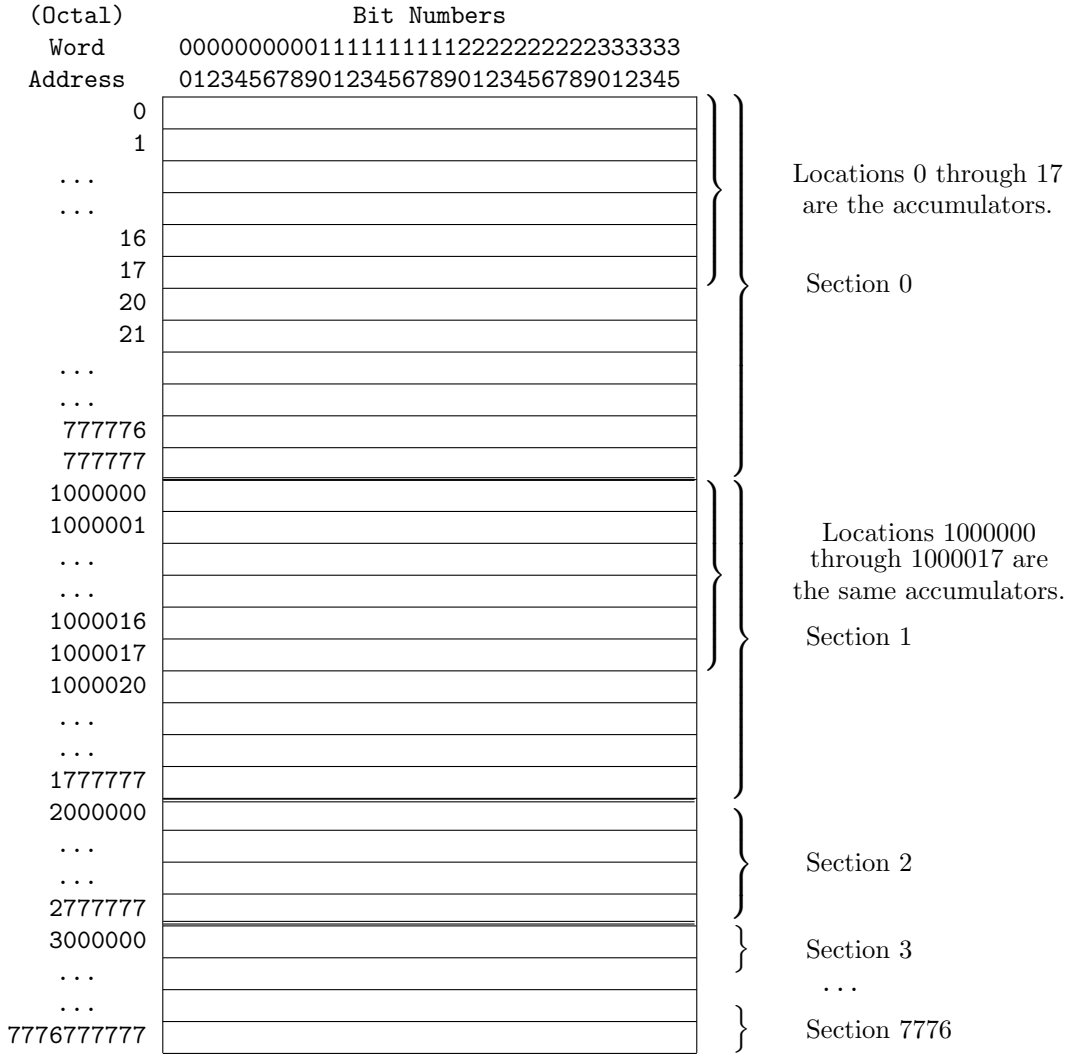
Old programs know only section 0. New programs shun section 0. Thus, in a single program, a programmer follows only one set of rules, either the traditional rules or the extended rules.

In the traditional computer, addresses range from `0` to (octal) `777777`. However, sixteen locations, addresses `0` through (octal) `17` are distinguished in several ways. These locations are called the *accumulators*. In many instructions, an accumulator will be selected as one of the operands. Also, any of the accumulators in locations `1` through `17` can be used as *index registers* to modify the selection of operands. As we shall see, the accumulators are very important in assembly language programming.

The extended addressing computer allows addresses in the range from `0` to `7776777777`. As suggested above, to keep our lives as programmers simpler, we ignore the addresses in section 0. Thus, the addresses of interest range from `1000000` to `7776777777`. Each section contains `1000000` addresses. Sections are important in the extended machine because it is easier to access data in the same section as the program (a *local address*) than it is to access data in other sections (a *global address*). A local address is specified by supplying 18 bits, i.e., a number in the range `0` to `777777`; the computer augments a local address with a 12–bit section number that is implied by context. A global address is 30 bits including an explicit section number.

As in the traditional computer, the extended addressing computer has sixteen accumulators. But each accumulator is known by two addresses. Local addresses `0` to `17` specify the accumulators as do global addresses `1000000` to `1000017`.

The word "*section*" is freighted with alternate meanings. When there may be ambiguity, this author shall try to use the words *address section* to mean one of these divisions of the machine's virtual address space.



The DECSYSTEMs 2060 and 2065 implement only sections 0 – 37.

Section 7777 is illegal.

Figure 2.4: DECSYSTEM–20 Extended Virtual Memory

2.2 The Central Processing Unit

The Central Processing Unit, or *CPU*, contains the arithmetic and control functions that follow the directions specified by a program. The CPU in a DECSYSTEM–20 is identical to those found in recent DECsystem–10 computer systems. We call the CPU a *PDP–10*, which is the name by which these central processors were once known. In this book we speak of the PDP–10 to mean precisely and only the central processor.

The PDP–10 central processing unit implements a set of elementary functions called *instructions*. Each instruction occupies precisely one word in the computer memory.[2]

The CPU contains logic and storage sufficient to *execute* (i.e., perform) one instruction at a time. Except for the program counter (explained below), the execution of an instruction does not change the CPU itself.[3] Instructions have the following kinds of effects:

- Instructions can change the contents of memory, including the contents of the accumulators.
- Instructions always change the program counter.
- Some privileged instructions effect changes in an input or output device. By such changes the computer transmits information to the outside world or receives information as input.

The CPU contains a *register* (that is, an array of one-bit storage elements) called the *program counter* or *PC*. The PC contains the memory address of the next instruction to execute. Each time an instruction is performed, the constant `1` is added to the PC. Adding one to a register is counting, hence the "counter" portion of the name "program counter." Because the PC increases by one each time an instruction is executed, consecutive memory addresses usually contain consecutive instructions.

A program consists of a series of instructions. To perform an instruction, the CPU first must *fetch* (i.e., read) the instruction that the PC addresses. After the instruction is fetched, the CPU *increments* (adds one to) the PC and then performs the function that was specified by the instruction. After executing the instruction, the CPU fetches the next instruction. Thus, after the instruction in location `2001234` is executed, the CPU normally executes the instruction at `2001235`, etc.

Special instructions called *jumps* and *skips* can change the sequence of execution by changing the program counter. A jump instruction supplies a completely new value for the program counter; a skip instruction increments the program counter an extra time, thus skipping over one instruction without executing it. By means of such instructions, program loops, control structures, and subroutine calls can be implemented.

Each word in memory contains some binary pattern; this pattern presumably is meaningful to the programmer. Some words that the computer reads from memory contain instructions to execute. Other words contain data. There are many different formats for data, and some data patterns cannot be distinguished from instructions. A word is executed as an instruction when the program counter addresses it. Storing instructions in the same memory as data allows great flexibility in what a computer can do; at the same time, it presents boundless opportunities for confusion.

[2]The extended instructions might be said to occupy more than one word, but for the moment we ignore such distinctions.

[3]The CPU also contains a small number of flags (single bit storage elements) that may be changed by the computation.

2.2.1 Computer Instructions

In assembly language programming, every instruction that we write contains a description of what operation to perform and which memory locations to affect.

2.2.1.1 Operation Code

Every instruction that is executed specifies a particular function to perform. Sometimes it is relatively simple, such as copying a word from one address to another. Sometimes the function is more complex, such as adding two words together. Each instruction has an *operation code* that specifies what function to perform.

Operation codes in the computer are really numbers. However, each of these numbers has been given a name, or *mnemonic* by which we expect to remember it. Among the functions of the assembler is the translation of the name we know into the numeric operation code that the computer acts on. An example of a mnemonic operation name is `MOVEI`, which we have already mentioned. You can probably guess the meanings of the operations called `ADD` and `SUB`. When writing an instruction in assembly language, the name of the operation appears first.

2.2.1.2 Operand Addressing

Most PDP–10 instructions allow two different memory addresses to be specified, but one of these addresses is restricted as explained below. These addresses define where the data comes from and where to place the result of the operation.

The PDP–10 CPU architecture employs what is sometimes called *one and a half address* instructions. The "one" address refers to the ability of every instruction to address any word in the memory space allocated to the program. The "half" address means that a second address, restricted to a small number of words, is also permitted. Since the second address is restricted, i.e., it cannot address all locations, it doesn't count for as much as the first one, hence, the expression "one and a half." It might be noted that one-, two- and three-address machines also exist. Further, some computers provide a combination of instructions, some that allow one address, others that allow two.

Although the "half" address is restricted, it is very important. It names one of the sixteen memory locations. These sixteen locations (addresses `0` to `17` octal) are often referred to as *registers*, or as *accumulators* (ACs). The accumulators can be used as normal memory locations whenever it is convenient to do so. Also, the accumulators are distinguished from other memory locations in three ways:

- Accumulators can be referenced as one of the operands in all data moving and test instructions. As such, accumulators are very useful for temporary storage.
- The accumulators numbered `1` to `17` can be used as *index registers* to modify any instruction's effective address calculation (see Section 5.3, page 45).
- Accumulators are implemented in high-speed solid-state memory rather than in the slower core or MOS memories. The accumulators are a fast and convenient place to hold frequently referenced data items.
- Accumulators can also be referenced as the general memory operand in an instruction. When an instruction refers to a memory location in the range `0` to `17`, or in the range `1000000` to `1000017`, it refers to an accumulator.

When writing instructions in assembly language, if an accumulator must be specified, write the accumulator number and a comma after the operation code. Then write the general memory operand. For example, in the instruction

```
MOVE    1,1000
```

the operation code is `MOVE`, accumulator number `1` is specified, and `1000` is the memory operand. This `MOVE` instruction copies the contents of the word at location `1000` into accumulator `1`.[4]

2.2.1.3 Instruction Sequences

Most instructions specify one arbitrary memory address and one accumulator address. Since it is not possible to reference two arbitrary addresses in one instruction, any operation in the computer that involves two arbitrary addresses must take at least two instructions, and must include storage of temporary results in an accumulator.

To give a specific example, suppose we want to copy the word at location `1000` to location `1437`. Since we can't reference both locations in one instruction, we are required to write a sequence in which we load the contents of location `1000` into an accumulator (choose any one) and then store the accumulator into location `1437`.

As mentioned above, the load operation is called `MOVE`, i.e., *MOVE* data from an arbitrary memory location to an accumulator. The store operation is called `MOVEM`; this means *MOVE to Memory*, that is, copy data from an accumulator to memory. For this example, we must write the instruction sequence:

```
MOVE    1,1000
MOVEM   1,1437
```

Here we have arbitrarily chosen the accumulator in location `1` as the place to hold the temporary result. The `MOVE` instruction destroys the previous contents of accumulator `1`; the `MOVEM` instruction leaves register `1` unchanged but overwrites the previous contents of location `1437`. (Please note that due to an occasional lapse of terminological exactitude, and to add variety to an otherwise dry and interminable narrative, we often use the words "register" and "accumulator" as synonyms.)

The accumulators are also involved in arithmetic operations. Suppose we must add the data in location `1000` to that in location `1234` and then store the result in `1437`. A sequence to accomplish this task is given below:

```
MOVE    1,1000
ADD     1,1234
MOVEM   1,1437
```

This sequence loads register `1` with the data from location `1000`. The `ADD` instruction adds the data found in location `1234` to the contents of register `1`, placing the sum in register `1`.[5] Finally, the sum is stored in location `1437`. Note that the result in register `1` was constructed in several steps; register `1` has been used here to accumulate a sum, leading to the name "accumulator" to describe these registers.

With these examples, we hope you now have some idea of the kind of demands that are placed upon

[4] To be perfectly precise, the instruction copies the contents of the word at the in–section (or local) address `1000`. This pedantic note applies to all the examples that follow in this chapter.

[5] This `ADD` instruction works for numbers stored in the PDP–10's fixed-point format.

the programmer. The two examples approximately correspond to the Pascal language statements "`C:=A`" and "`C:=A+B`". You can see that simple arithmetic operations can be translated in a relatively straightforward manner to machine instructions.

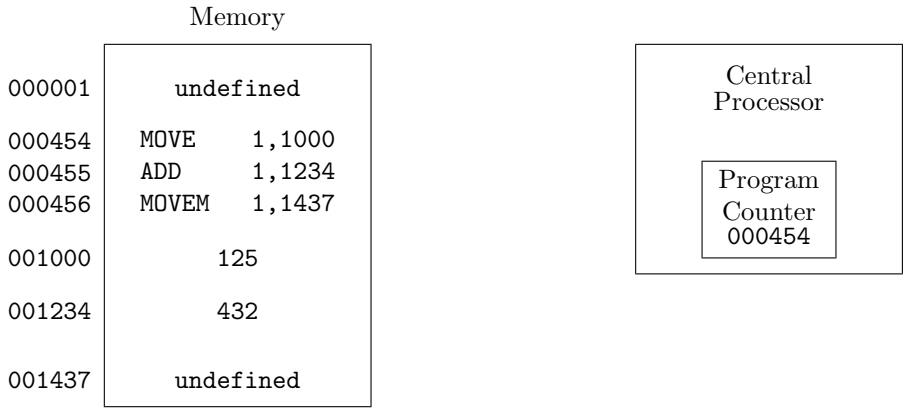
When it is necessary to copy data from one accumulator to another, usually one instruction is sufficient. The accumulator that is being copied from can appear as the memory operand in a `MOVE` instruction. For example, to copy the data from register `1` to register `16` we could write:

```
MOVE    16,1
```

In this instruction, the arbitrary memory address happens to be one of the accumulators. Data is copied from the memory operand (register `1`) to the accumulator specified (register `16`).

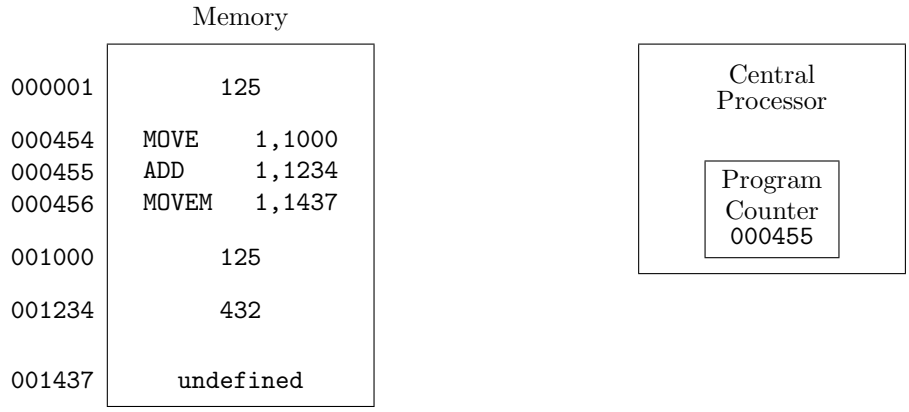
2.2.1.4 Instructions in Memory

We now expand on the ideas that programs reside in memory, and the program counter steps through the sequence of instructions. Repeating the example above in which we wrote a sequence to add two numbers together, let us display this program as it might appear in memory:



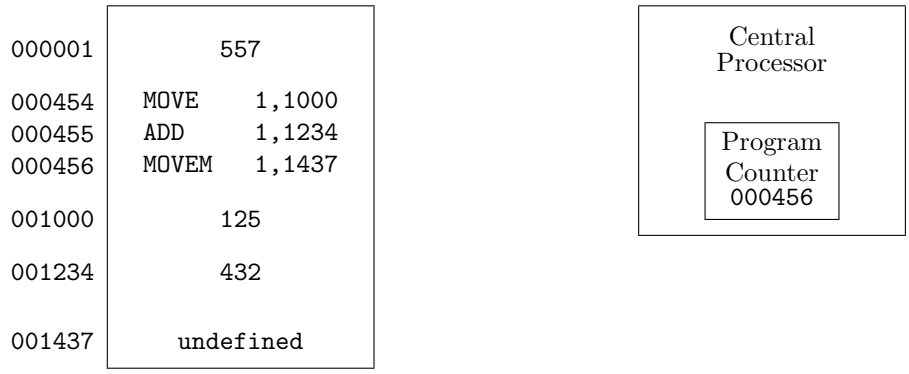
We have arbitrarily selected locations `454`, `455`, and `456` as the three consecutive locations to hold this program fragment. Generally speaking, this program fragment could be anywhere in memory. The only restriction is that, as we shall see, some memory locations are changed by the program; it would be a bad idea to place this fragment where it might change itself.

Initially, we shall set the program counter in the central processor to the value `000454`. This *points to*, or *addresses* the first instruction in the sequence. If we now tell the computer to start running, it will fetch the instruction that the PC addresses. This instruction, in location `454`, is `MOVE 1,1000`. The effect of this instruction is to copy the data contained in location `1000` to the accumulator at location `1`. As the `MOVE` instruction is being executed, the program counter is incremented to contain `455`. The state of memory and the program counter after the execution of this first instruction is now

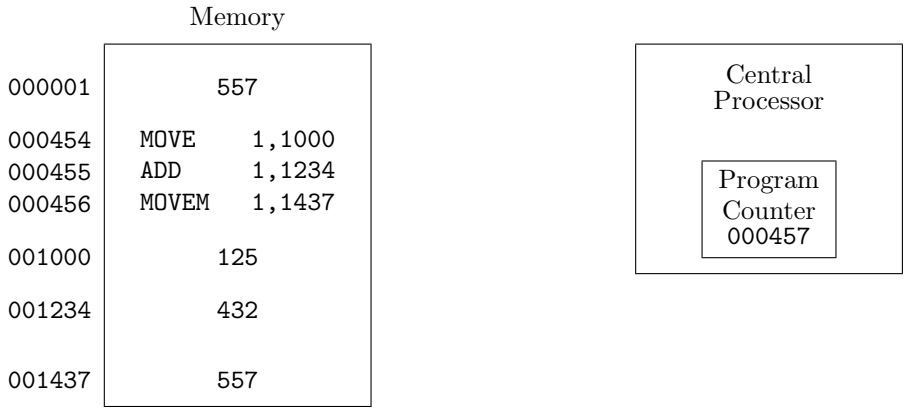


Note that the contents of location `1` have been changed. Also the program counter now points to `455`.

The computer fetches the next instruction, the `ADD 1,1234`, from location `455`. The computer performs the `ADD` operation by reading the data in location `1234` and adding it to the data found in location `1`. The result, `557`, is stored in location `1`; the program counter is incremented to `456`:



Next, the central processor fetches the instruction at `456`. The instruction `MOVEM 1,1437` directs the CPU to store the contents of location `1` into location `1437`; the PC advances to `457`. After the execution of this instruction, the CPU and memory look like this:



The program fragment that we have been examining has now been executed. The computer will go on to fetch whatever instruction is contained in location `457`, because `457` is addressed by the program counter. In the normal scheme of things, location `457` would contain another instruction.[6]

2.2.2 Historical Notes

Because the processors in the DECsystem–10 and DECSYSTEM–20 are identical, programs written for one system might be expected to run on the other. This is partially true: many DECsystem–10 programs will run on the DECSYSTEM–20, because the DECsystem–10 has had significant influence on the development of the DECSYSTEM–20, and because TOPS–20 includes a TOPS–10 emulation program called `PA1050`. Occasionally we shall note that some program feature or technique is compatible with the ways things are done in the DECsystem–10's TOPS–10 operating system. The DECSYSTEM–20 represents various advances beyond the techniques practiced in the DECsystem–10; consequently, DECSYSTEM–20 programs will not run on the DECsystem–10.

Historically, there have been various versions of the CPU. The original appeared in 1964 as the PDP–6. The KA10, which was the first of the PDP–10 processors, was first built in 1968. The KI10 followed the KA10. Two newer CPUs, the KL10 and the KS10 are being used in current systems. The KS10 appears in the 2020 model. The KL10 is present in the larger 2040, 2050, 2060 and 2065 configurations. We shall generally discuss the processors in which extended addressing has been implemented, e.g., the KL10 in the 2060 and 2065 and the XKL TOAD systems. The evolution of the DECsystem–10 and DECSYSTEM–20 is discussed in [BELL].

Several other companies have designed and built computer systems with processors that execute the PDP–10 instruction set. In the early 1970s the staff at Xerox Corporation's Palo Alto Research Center built "MAXC". Also in the early 1970s, DARPA funded the design of "Super Foonly" at Stanford University; several key ideas in that design were adopted in Digital Equipment's KL10. After the university ceased to work on that design, the Foonly company was formed; it built several machines. In the mid 1980s (after Digital Equipment dropped their plans to build a follow–on to the KL10), Systems Concepts, Inc. built the SC–30, the SC–40, and others; they licensed designs to Compuserve, who built additional machines. Tymeshare also designed a system. In the 1990s XKL Systems Corporation (later XKL, LLC) built the TOAD–1 systems.

[6] We shall discuss how to stop the computer at the end of a program in Section 3, page 17.

Chapter 3

First Example

We now show a sample program that actually runs and performs an output operation. These examples are a very important part of our method of instruction. The first several examples show primarily the manipulation of characters and the use of the timesharing terminal for input and output. After we present the entire machine instruction set, we shall go on to consider examples of disk input/output and other types of calculations.

This example opens small windows through which we can begin to view the three subject areas, the machine, the assembler, and the operating system. In later examples, we will strive to enlarge these windows, so that our understanding extends through a larger portion of each of these three subjects.

The program that we are going to construct approximately corresponds to the following Pascal program:

```
PROGRAM hi(output);
BEGIN
WRITELN('Hi')
END.
```

This program merely types "`Hi`", a new line, and stops.

In some sense, the "program" here consists of only the portion that says `WRITELN('Hi')`. But Pascal has rules that require the presence of the line that says `PROGRAM` and that require the `BEGIN` and `END`. In the same way, our assembly language program has a very small portion that does the work, and a large amount of other material that is necessary but not directly connected with our purpose.

In the DECSYSTEM–20 the "instruction" that types a string on the terminal is `PSOUT`, meaning *Primary String OUTput*. Although `PSOUT` appears in the program as if it were an ordinary instruction, it is actually a subroutine call to TOPS–20. Each such subroutine call to TOPS–20 is called a *JSYS*, meaning *Jump to SYStem*.

Since we know that our program must include a `PSOUT`, we start by creating a program fragment:

```
.
.
PSOUT
.
.
```

PSOUT may be thought of as analogous to the WRITELN statement in Pascal.[1] The WRITELN statement requires an *argument* that describes what string to output. Naturally, the PSOUT subroutine also requires an argument that describes the string to output.

Before we go on to further describe the argument that PSOUT requires, we must examine the nature of the string. The assembler program (together with the loader) is responsible for building a program in memory. The program consists of machine instructions, JSYS operations, and data. As we have said, the assembler knows how to translate mnemonic operation codes into the numeric codes that the CPU executes. The assembler also includes a variety of functions that assist in making data appear as the appropriate numbers in memory. These functions are accessed via *pseudo–operators* that are special commands to the assembler.

The ASCIZ pseudo–operator can be used to build a text string in memory. To build a string using ASCIZ, write the word ASCIZ and then some non–blank character that does not appear in the desired string. Follow that *delimiter character* with the text of the string, and another occurrence of the delimiter character. All the characters that appear between the two delimiters will be made into a string. For example to build a string containing the four characters "H", "i", carriage return, and line feed, we write the following fragment of assembly language:

```
        .
        .
        ASCIZ   /Hi
/
        .
        .
```

Here we have used the slash character (/) as the delimiter character. Note that the two letters "H" and "i" follow the first slash. The carriage return and line feed follow the letter "i" and come before the closing delimiter that appears on the next line. We'll add this fragment to the part we had before:

```
        .
        .
        PSOUT
        .
        .
        ASCIZ /Hi
/
        .
        .
```

At this point, the positioning of PSOUT before the string is not important; we could reorder these.

Well, now we have the string. How about the argument to PSOUT? In PSOUT, the argument is expected to be in accumulator number 1. Therefore, prior to executing PSOUT, we must be certain that 1 contains the right thing. Since strings are generally too long to fit into one word, we load a *pointer* that describes the string into register 1.

The pointer must include the address in memory where the assembler put the string. But how do we find out where the string is? The answer is relatively easy. The assembler allows us to define *labels*. A label is a name for a memory location. Once a label is defined, that memory location can be *referred to* by mentioning the name of the label.

So, the next thing to do is to plant a label on the computer word that contains the start of the

[1] Actually, it is more similar to WRITE.

string.[2] A label can be up to six letters long; we select the name `MESAGE` (an obvious corruption of "message") to label the string.[3] We change the line on which the string is defined to include the name `MESAGE` and a colon (`:`) at the left margin. The colon tells the MACRO assembler to define a label called `MESAGE`.

```
        .
        .
        PSOUT
        .
        .
MESAGE: ASCIZ   /Hi
/
        .
        .
```

The next thing to do is to get a pointer to this string into register 1. This is accomplished by the instruction `HRROI 1,MESAGE`. This instruction mentions register `1` as the accumulator; the symbol `MESAGE` appears as the memory operand in the `HRROI` instruction. The execution of the `HRROI` instructions builds one of the most common forms of a string pointer in register `1`. The mnemonic `HRROI` means *Halfword from the Right to the Right, Ones to the left half, Immediate.* This instruction loads the selected register, register `1`, with a pointer that is composed of ones in the left halfword and the address of `MESAGE` in the right halfword. `HRROI` is one of the halfword instructions that we will discuss in Section 12, page 155. (An aside: ones in the left halfword is not an obvious choice. Why did the designers of TOPS–20 and its predecessor, Tenex, choose this format? They use this format because the `HRROI` instruction makes it easy to produc such a pointer.)

The `HRROI` instruction must occur before the `PSOUT` so that register `1` can be properly set up prior to the `PSOUT`. We now add this instruction to the program fragment that we are building:

```
        .
        .
        HRROI   1,MESAGE
        PSOUT
        .
        .
MESAGE: ASCIZ   /Hi
/
        .
        .
```

Instructions are executed by the computer one after another. After the `PSOUT` the computer will want something to do next. Since the `PSOUT` concludes the work we intended, we tell the computer to stop running our program by including the `HALTF` operation. `HALTF` is another subroutine call to the system; it tells the system to stop executing this program. If we don't include a `HALTF`, the computer will fetch the next word following the `PSOUT` and execute it as an instruction. Since that word may not be an instruction at all, it would be a bad idea to allow the computer to attempt to execute it. So, under the `PSOUT` we add a `HALTF`.

---

[2] Actually, there is only one computer word in this string. In a longer string, this label would refer to the first computer word of the string.

[3] A version of MACRO that distinguishes longer identifiers has been developed and we are beginning to use it. However, in this book we shall continue to cajole you to use six–character names.

```
        .
        .
        HRROI   1,MESAGE
        PSOUT
        HALTF
        .
        .
MESAGE: ASCIZ   /Hi
/
        .
        .
```

Finally, we must tell the computer where to start this program. We do this by defining a label that we shall name `START`. Then, we must tell the computer that `START` is the name of the starting address. The assembler pseudo–operator `END` informs the assembler that the text of the program is complete; the argument that follows the `END` pseudo–operator names the starting address:

```
        .
START:  .
        HRROI   1,MESAGE
        PSOUT
        HALTF
        .
MESAGE: ASCIZ   /Hi
/
        .
        END START
```

The `END` pseudo–operator comes after all the text of the program.

We have built nearly an entire program. There are, however, some necessities that have been omitted as yet. The complete text of the first example appears below. We have undertaken this preliminary explanation so that by the time you see the full program most parts of it will be familiar; the rest will be explained following the text of this example program. The program below begins on the line containing `TITLE` and ends after the line containing `END`. We have added comments to this program to remind us of the meaning of each of the parts. The most frequently used form of a comment begins with a semicolon character "`;`" and includes the remainder of the line.

```
        TITLE   HI - Program to type "Hi".  Example 1

Comment $ Example 1, Program to type "Hi"

The following program types "Hi" and carriage return line feed (CRLF)
on the terminal.

$

        SEARCH  MONSYM          ;Add TOPS-20 symbols to MACRO
        .PSECT  CODE,1001000    ;put the program in section 1

START:  RESET                   ;RESET the state of I/O devices
        HRROI   1,MESAGE        ;Copy address of message to register 1
        PSOUT                   ;Output the message
        HALTF                   ;Stop execution here.

MESAGE: ASCIZ   /Hi
/
        END     START           ;End of program, start at START
```

You can run this program yourself by creating a file that contains everything from the word `TITLE` through the end of the line that says `END`. Be sure to finish the last line with a carriage return character. It doesn't matter what file name you choose, but the file type should be `MAC`. For instance, the file name `EX1.MAC` would do fine. After creating this file, the command `EXECUTE EX1` will run it.

Although this is a very modest example of assembly language programming, it is worth our attention because it contains elements that we shall find in all other programs.

3.1 Review of Example 1

Programs written in assembly language consist of instructions for the computer to execute when running the program, descriptions of initial data for the program, and instructions to the assembler. Every component of this program has a name and a specific function.

- The program contains six *pseudo–operators*: `TITLE`, `COMMENT`, `SEARCH`, `.PSECT`, `ASCIZ` and `END`.
- The program contains two user–defined *labels*: `START` and `MESAGE`.
- The program contains three *JSYS operations*, which are requests made to the operating system: `RESET`, `PSOUT`, and `HALTF`.
- The program contains one *machine instruction*, `HRROI`, that tells the CPU to perform one of its primitive operations.
- The remainder of the program is mostly comments, except for *arguments* to the pseudo–operators `TITLE`, `SEARCH`, `.PSECT`, `ASCIZ`, and `END`.

3.1.1 Pseudo–Operators

A pseudo–operator, or pseudo–op, is an instruction to the assembler. A pseudo–operator is called an *operator* because it appears in the text of the program in the same place that other operators

(i.e., machine instructions) appear. A pseudo–op is not really a machine instruction, even though it looks like one, hence the prefix *pseudo*. Pseudo–ops have effect at *assembly time*: distinction is drawn here between things that the assembler does while translating a program, in contrast to the computer instructions that are performed when the program is being run. Things done by the program, rather than by the assembler, are said to be done at *run–time*. The five pseudo–ops in example 1 are `TITLE`, `COMMENT`, `SEARCH`, `.PSECT`, `ASCIZ`, and `END`. Each pseudo–op performs some particular function. The pseudo–ops in this example are described below.

3.1.1.1 TITLE

The pseudo–op `TITLE` usually appears on the very first line of an assembly language program. `TITLE` serves two purposes. First, when the assembler makes a listing of your program, the data supplied in the remainder of the `TITLE` statement will appear at the top of each page of the listing.

The second function of `TITLE` is analogous to the function of the `PROGRAM` statement in Pascal. `TITLE` serves to give a name to program. As MACRO assembles a program, it builds a *symbol table* that contains the name and value of each symbol that was defined by the programmer. The program name is also the symbol table name. The `TITLE` pseudo–op takes the first six letters of the word following `TITLE` as the program name and the symbol table name. In this program, the symbol table name is `HI`. The symbol table name is used to select particular symbols when debugging a program with DDT, as we shall demonstrate in Section 9, page 101.

3.1.1.2 COMMENT

The word `COMMENT` begins a multi–line comment. The first non–blank character after the word `COMMENT` is taken as a *delimiter*; in this example the delimiter character is a dollar sign. All text up to and including the next occurrence of that delimiter character is the body of the comment. The body of the comment is ignored by the assembler.

A second way to make a comment in the text of the program is by means of a semicolon character (`;`). All text that follows a semicolon on a line is ignored by the assembler.

Comments, especially those which begin with a semicolon, should be used liberally throughout a program. There is no need to use comments to belabor the obvious: the instructions, after all, say what they are doing. Comments should be used to reveal the author's intentions and expectations about the state of the program and what is thought to be happening.

Comments are extremely important. We can all agree that a program is a means of communication from the programmer to the computer. However, a common and important use of programs is as a means of communication (or object of discussion) between two people. In fact, the two people, the author and the reader, may be separated in space or in time. Because the program itself is not sufficient to readily convey its meaning to the reader, we augment the program with comments to explain what we are doing. To the extent that the author and the reader are the same person, comments are merely helpful. When the author and reader are not the same person, comments are necessary to convey the entire meaning effectively.

Although in most circumstances the entire text of a comment is ignored by the assembler, the programmer should be aware that sometimes the assembler will treat the characters "{" and "}" as significant even if they appear in comments.[4]

[4]See the discussion of macros and conditional assembly, Section 17, page 235.

3.1.1.3 SEARCH

The assembler program performs a translation in which names that are meaningful to us are transformed into numbers that are meaningful to the computer. To effect this translation, the assembler uses a dictionary that we call the *symbol table*. The symbol table relates the name of each symbol to its definition. An extensive list of names and definitions is built into the assembler so that when it starts, many definitions are present.

Despite its extensive initial dictionary, further definitions must be added to tailor the assembler to TOPS–20 (the assembler was originally built for TOPS–10). These additional definitions are added by the `SEARCH` pseudo–op.

The pseudo–op `SEARCH MONSYM` augments the initial symbol table by adding the contents of the file `SYS:MONSYM.UNV` to the normal symbol table. The file `MONSYM.UNV` contains definitions of the TOPS–20 system calls (i.e., the JSYS names) and special definitions that are used in communicating with TOPS–20. A specific example of a symbol that is added is the definition of the `PSOUT` JSYS.

3.1.1.4 .PSECT

The `.PSECT` pseudo–op is a directive that tells the assembler that we are building a program that is partitioned into *program sections*. Further, MACRO will convey information to LINK, the loader, to direct where in memory the program should be loaded. A program section is also called a *psect*, following the name of the pseudo–op by which we create them. (This is another way in which we overload the word "section". We shall try to be unambiguous where the reader might otherwise become confused. This may become a wee bit tedious, for which apologies are offered.)

The first parameter to the `.PSECT` pseudo–op is the name of the program section into which to put this part of the program. The second parameter, which is valid only the first time this program section name is seen, directs LINK to put this program section into consectutive virtual memory locations starting at the given address.[5]

Sectioning programs with `.PSECT` is quite a bit more advanced than our present level of understanding. Sectioning is used when different parts of a program need to be treated in different ways. (For example, a program may be divided into instructions, constants, static–allocated data, and dynamic data, with the instructions and constants being protected from writing.)

The purpose of `.PSECT` in this program is to tell LINK to put the program into memory starting at address 1001000. This is the address of the first location on page 1 in address section 1. So that we shall be able to use the extended addressing capabilities of the DECSYSTEM–20, we place the program in a section other than section 0. We don't start at 1000000 because the accumulators are aliased to addresses 1000000–1000017. We could start the program section at 1000020, but this author prefers to start at a page boundary, a multiple of (octal) 1000.

3.1.1.5 ASCIZ

The `ASCIZ` pseudo–op tells the assembler to take the text that follows the word `ASCIZ` and assemble that text into computer words. The `ASCIZ` pseudo–op produces a text format that is very common within the DECSYSTEM–20. In the example, the character "`/`" following the word `ASCIZ` is a delimiter; it defines the extent of the text that is to be assembled by the `ASCIZ` pseudo–op. `ASCIZ`

[5]The location at which to load the psect is also restricted to the first mention of the psect name to LINK. When LINK is loading only one file, this restriction is easy to satisfy. For more complex program linking situations, other arrangements are recommended.

will assemble the text that it finds between the first "/" and the next occurrence of a "/" character. Note that even though the second "/" occurs on some other line, ASCIZ continues off the end of the first line until it finds the second "/".

ASCIZ does not require "/" to be the delimiter. The ASCIZ pseudo–op accepts the first non–blank character following the word ASCIZ as the delimiter; for the purpose of locating the delimiter, the tab character is considered as equivalent to a blank. ASCIZ processes all text up to the next occurrence of that delimiter character.

To *assemble* text means to take binary numbers corresponding to each character and place these numbers into computer words. Text data is assembled character by character and stored in computer words. When a word fills up with text, the assembler starts filling another word. The assembler program takes its name from the function of building entire words by gathering information from several constituent *fields*. We will provide a further explanation of the ASCIZ pseudo–op in Section 4.8.1, page 39.

3.1.1.6 END

The END pseudo–op signifies the end of the text that describes the program and it specifies the starting address. In this case the label START is designated as the starting address of the program.

In assembly language the starting address of the program need not be the first location loaded. In fact, seldom is the first thing written actually the starting sequence. This is similar to Pascal and other structured languages where declarations (of variables and procedures) come before the main program.

In a similar contrary fashion, the END statement does not signify the end of the execution of the program. END simply flags the end of the text. Execution terminates when the HALTF JSYS is performed.

Some text editors allow the last line of a file to end without having the carriage return and line feed characters. MACRO insists that every line end with a carriage return and line feed; if these are omitted from the END statement, MACRO will fail to see the END.

3.1.2 JSYS Calls

As we have mentioned, our programs can request specific functions from the TOPS–20 operating system by means of a special instruction called JSYS, meaning Jump to SYStem. This program contains three JSYS operations, RESET, PSOUT, and HALTF. Each of these can be thought of as a subroutine call to the TOPS–20 operating system. The particular JSYS name specifies which one of many functions to perform.

3.1.2.1 RESET

The RESET JSYS is an appropriate initialization instruction. It terminates any input/output activity that might have been pending and performs other useful cleanups. RESET should be at or near the start of every program.

3.1.2.2 HALTF

The HALTF JSYS tells the operating system to stop executing the program. This is the normal way to signify that a program has reached its end. When the TOPS–20 Command Language Processor (EXEC program) starts running a program, the terminal is made available to the running program for input and output activity. When the program executes the HALTF function the operating system stops running it. When the EXEC notices that the program has stopped, it uses the terminal to prompt for further commands. Technically, a program is run inside of an environment called a *process* or *fork*. HALTF is mnemonic for *HALT Fork*.

One feature of HALTF that is often overlooked is that the EXEC CONTINUE command will cause a program to continue executing at the next instruction that follows the HALTF. In this example, for simplicity, we have not provided for the possibility that a CONTINUE command might be given; if this program were to be continued, it would eventually execute an illegal instruction.

3.1.2.3 PSOUT

The PSOUT JSYS is used to send a string of characters to the terminal. As we have mentioned, PSOUT means *Primary String OUTput*, where "primary" usually means the terminal that controls the program.

When PSOUT is executed, TOPS–20 looks in accumulator number 1 for a pointer that describes the string to output. It is necessary to execute an instruction prior to the PSOUT to set up accumulator 1 with the correct pointer.

JSYS instructions that require information from the program will expect to find that information in low–numbered accumulators. Accumulators 1 through 4 are used to pass information between the user program and the TOPS–20 system. An item of information passed from the program to TOPS–20 is called an *argument*. If TOPS–20 returns information from a JSYS, that is called a *result*.

The number of accumulators needed for arguments and results varies depending on which JSYS function is performed. RESET and HALTF, for example, have neither arguments nor do they return results. PSOUT has one argument and one result: an updated string pointer, pointing to the end of the string that was typed by PSOUT, is returned in accumulator 1.

If a JSYS requires more arguments than would fit in four accumulators, then one accumulator will be used as a pointer to an *argument block*. An argument block is an array of contiguous memory locations that would be set up to contain appropriate information for a JSYS call.

3.1.3 HRROI Instruction

HRROI is the first machine instruction that we have encountered. The instruction HRROI 1,MESAGE loads a suitable pointer for the call to PSOUT into accumulator number 1. PSOUT accepts several different formats of pointer; without going into details, the HRROI instruction produces the most common pointer format. Note that accumulator 1 is mentioned as the accumulator field of the instruction (the part where it says "1,") and the symbolic name of the message string (MESAGE) appears in the address field of the instruction.

3.1.4 Symbols, Labels, and Values

In this program, the address field of the HRROI instruction refers to the *symbol* named MESAGE. The symbol MESAGE is *defined* by writing "MESAGE:" as the first thing on the line that we want to call by the name MESAGE. A symbol that is defined by appearing at the beginning of a line with a colon is called a *label*. A label is a handle, i.e., a name, by which we can reference the data or the instruction at the line where the label appears.

A *symbol* is an entity within the assembler that has a name and a value. One of the important functions of the assembler is to maintain the symbol table. As we have mentioned, the symbol table is a dictionary of symbols and their values. When the assembler sees something like "MESAGE:" it enters the name MESAGE in the symbol table. The value of a symbol such as MESAGE is essentially the address of the first word stored in memory immediately following the occurrence of the label. Simply stated, the definition of the symbol MESAGE is the address of the text on the line where MESAGE appears.

The purpose of keeping the symbol table is to permit the assembler to substitute the correct value for the symbol whenever the name of the symbol is referenced. Thus, when the assembler is confronted by a line containing the text HRROI 1,MESAGE, it looks up MESAGE in the dictionary and substitutes the value of symbol. In this example, the symbol MESAGE has the value 1001004.[6] When the assembler sees the text HRROI 1,MESAGE, it substitutes (the in–section portion of) the value 1001004 for MESAGE resulting in something equivalent to HRROI 1,1004. Of course, instruction names such as HRROI also have numeric values. The entire line HRROI 1,MESAGE is translated to the octal number 561040001004.

Even though a reference to the label MESAGE appears before the definition of the label, MACRO can make the proper substitution (of the in–section portion of 1001004 for MESAGE) because MACRO reads the program twice. On the first reading, it assigns values to symbols. On the second reading, MACRO makes the appropriate substitutions of values for symbols. This kind of operation, naturally enough, is called *two–pass* assembly.

The power of the assembler is that it frees us from having to perform rote translations (e.g., of HRROI to the number 561000000000). Another benefit is that when a label moves, that is, when the program changes so that the symbol MESAGE takes on a new value, say 1001011, the assembler will change every reference we make to MESAGE to reflect the new value. Thus, we are freed from a number of dull and tiresome bookkeeping chores.

In the MACRO assembler, symbols are limited to six characters: only the first six characters are significant. The characters may be letters or digits, but the first character must be a letter. Included among the letters are the three characters ".", "%", and "$". Lower–case and upper–case letters are equivalent. Some examples of legal and illegal symbol names follow:

Legal Symbols		Illegal Symbols	
MESAGE		1MORE	starts with a digit
.EXAM1		MARK!TWAIN	has an illegal character
mesages	this is equivalent to MESAGE		

[6] The value 1001004 results from the action of LINK in conjunction with the action of the assembler. In the assembler, MESAGE has the value 4 relative to the start of the "CODE" program section.

3.2 Programs and Memory

Your program will occupy some portion of the virtual address space of the computer. Generally speaking, all locations are equivalent, except for a small number of places that are reserved for special purposes. Locations `0` to `17` are the accumulators, and may be used as such, or as memory locations, interchangeably, at your convenience. Section 0 locations `40` and `41` are used by the hardware as the traditional `LUUO` trap locations (discussed in Section 18, page 257). In TOPS–10, the locations in the range `20` to `137` are called the *Job Data Area*; these locations are used by the loader program and by TOPS–10 itself to communicate some things to the program.[7] Except to accomplish specific functions, it's generally a good idea to leave these locations alone. By default, the loader will start loading your program at location `140` in section 0.

However, by means of the `.PSECT` pseudo–op, we have directed LINK to put the program in consecutive locations starting at `1001000`.

The assembler together with the loader produces a series of computer words loaded into memory. These computer words are the machine representation of the program. The computer words corresponding to this example are displayed in Figure 3.1. (You can see these values for yourself via the EXAMINE command in the EXEC. Type `examine 1001000` and the exec will display the contents of the selected location. Type just `examine` to see the next location.)

```
Address   Contents       Meaning
1001000   104000000147   RESET
1001001   561040001004   HRROI   1,MESAGE ;(MESAGE is 1001004)
1001002   104000000076   PSOUT
1001003   104000000170   HALTF
1001004   443221505000   ‘‘H’’, ‘‘i’’, carriage return, line-feed, and null.
```

Figure 3.1: Machine Representation of the Example 1 Program

Programs are primarily an instrument of communication. By programs we communicate our instructions to the computer and our intentions to other people. Although the computer can understand the octal listed above better than it can understand the MACRO program, we use MACRO because it is a convenient bookkeeping tool. MACRO increases our productivity, and it provides a sensible means for communicating programs among people who understand it.

3.3 Exercise — Self–Identification

Write a program similar to the one shown in Figure 3.2. Put the text of the program into a file called `HW1.MAC`. Execute the program by means of the TOPS–20 command `EXECUTE HW1.MAC` This command will run MACRO to translate your program into a *binary relocatable* file named `HW1.REL`. The `REL` file will be loaded into memory by LINK, and the program will be started.

When you are satisfied with the results, use the following two TOPS–20 commands to produce a cross–reference listing:

```
@COMPILE/COMPILE/CREF HW1.MAC
@CREF
```

[7] Although TOPS–20 itself doesn't use any of the locations from `20` to `137`, the loader will initialize (some of) them as it would for a TOPS–10 program.

```
        TITLE   HW1 Self-Identification

        SEARCH  MONSYM
        .PSECT  CODE,1001000

START:  RESET
        HRROI   1,MESAGE
        PSOUT
        HALTF

MESAGE: ASCIZ   /
My name is Ralph Gorin
I have taught this course too many times.
If I were a student in this course, I would mention my class
year and major, and say why I was studying assembly language.
/
        END     START
```

Figure 3.2: Sample Homework 1

The `/CREF` switch in the `COMPILE` command causes MACRO to write the file `HW1.CRF`; the `CRF` file contains a listing of your program that is augmented with special control characters for the CREF program.

The `CREF` command causes the CREF program to translate the `CRF` file into a readable cross–reference listing. The resulting file is sent to the printer by CREF; the `CRF` file is deleted. Note that no changes to the source file have been made.

Examine the cross–reference listing carefully. Turn it in. Although the cross–reference listing will not be discussed until Section 8.2, page 91, it will still be useful for you to study it.

Hints:

- After you get this to run, you must use the `COMPILE/COMPILE/CREF` command to force reassembly with cross–reference output. If you don't include the `/COMPILE` switch, the existence of an up–to–date `REL` file will inhibit the reassembly.
- If you forget the `CREF` command, the listing output will contain a large assortment of weird characters instead of a cross–reference. The `/CREF` switch causes MACRO to make a listing including these strange characters; the extra characters are used by the CREF program.
- If you leave off the closing slash (`/`) in the argument to the `ASCIZ` statement, the assembler will "eat" the rest of the program, including the `END` statement, and then complain that the `END` is missing.
- If you omit the carriage return and line feed that terminate the line where `END` appears, MACRO will complain that the `END` is missing.
- For more advice about what to avoid see Appendix E, page 661.

Chapter 4

Representation of Data

The computer's memory contains both data to be manipulated and programs stored as binary patterns. These binary patterns are sequences of ones and zeros, grouped into units called words. These patterns are intended to represent data that is of interest to us. Even when information is not numeric (this text for example) it is stored in the computer as binary patterns.

The computer cannot distinguish between a number representing π, 3.14159265..., an instruction to itself, or part of a Shakespearian sonnet; all would be stored as binary patterns. Any distinction that is drawn between these items is based on the interpretation of these patterns by a program.

4.1 Representations

A *representation* is a convention that relates marks on paper (or marks inside the computer) to numbers or other objects. A number such as one, two, or sixty–seven, has an existence that is independent of the characters that we use when we write the number. The characters that we use to write a number, e.g., 1, 2, or 67, are simply a conventional representation of the number. Most people are aware of at least one other convention for representing numbers. In Roman numerals we would represent these same numbers as I, II, and LXVII.

A representation is useful when it has a behavior that is analogous to the object or concept that it represents. That is, we are comfortable writing $1 + 1 = 2$; therefore, in any representation of numbers that we choose, we should find equal comfort in $\mathcal{R}(1) + \mathcal{R}(1) = \mathcal{R}(2)$, where $\mathcal{R}(1)$ signifies the representation of 1. Numbers are infinite, but computers — especially their primitive operations — are finite. Therefore, there will be a limit to the domain of numbers for which the primitive operations are appropriate.

Because of the fundamental property that the memory stores only ones and zeros, the computer represents numbers using the binary (base two) system. Binary numbers are at the heart of the representation of data in the computer; when the computer does arithmetic, it manipulates numbers in accordance with the rules of binary arithmetic. What this means to us is that we must become familiar with a new representation for numbers; we must develop an understanding of what binary numbers are and how to manipulate them. We start with binary integers.

4.2 Binary Integers

In the familiar decimal number system, a number is written as a pattern of digits. By convention the column containing the rightmost digit has weight `1`; the next column to the left has weight `10`, the next has weight `100`, etc. Each column has a weight that is a power of ten. (In the rightmost column, the weight `1` represents 10^0.) Each column holds one digit that can take any one of ten values (`0` to `9`).

The binary number system has a similar structure, except column weights are all powers of two, instead of powers of ten. Each column holds one binary digit (bit); a bit can range over only two values, `0` and `1`. The column containing the rightmost bit has weight `1`. Moving to the left we find columns with weights `2`, `4`, `8`, `16`, etc.

In the decimal number system, a number written as `123` is interpreted as one hundred plus two tens plus three ones. That is, the column weight is multiplied by the digit that appears in the column; the sum of these products is the intended number.

In the binary number system, the same interpretation applies. However, each column weight is a power of two, instead of a power of ten. The number that is written as `101` in the binary system is interpreted as being one four plus zero twos plus one one; the binary pattern `101` corresponds to the number five.

In a computer, binary numbers are used to represent the contents of a computer word. In a computer with a five–bit word length, binary numbers would be written as a pattern of ones and zeros in the grid that is shown:

16	8	4	2	1	Column Weight
2^4	2^3	2^2	2^1	2^0	Power of 2

By carefully selecting which bits are ones and which are zeroes, any number in the range from `0` to `31` can be formed. Some examples are shown below:

16	8	4	2	1	Column Weight		
2^4	2^3	2^2	2^1	2^0	Power of 2		
0	0	0	0	0	= 0		
0	0	0	0	1	= 1		
0	0	0	1	0	= 2		
0	0	0	1	1	= 3	=	2 + 1
0	0	1	0	1	= 5	=	4 + 1
0	1	1	0	1	= 13	=	8 + 4 + 1
1	1	1	0	0	= 28	=	16 + 8 + 4
1	1	1	1	1	= 31	=	16 + 8 + 4 + 2 + 1

In a machine with a word length greater than five bits, these numbers would be represented with the same patterns, but extra zeros would be added to the left of these binary digits.

4.3 Arithmetic in the Binary System

Some authors have called the binary system "lazy man's arithmetic", because the rules for doing arithmetic are so very simple:

0+0	=	0		0+1	=	1
1+0	=	1		1+1	=	10

Such authors notwithstanding, using the binary system is not really any shortcut. Although each binary digit is easy to deal with, binary numbers are longer than their decimal equivalents.

Some examples of binary addition follow. You should make certain that you understand these and understand how the results follow from the four rules stated above.

```
  10    11    101    1110    1101    10000
+ 1   + 1   + 11   + 101   +1011   +10000
  11   100   1000   10011   11000   100000
```

If our machine were limited to five bits, we could not compute the sum `10000+10000` because the correct result, `100000`, does not fit into five bits.

4.4 Representing Negative Numbers

Before we attempt to discuss the representation of negative numbers in the binary system, we shall investigate some representations in the more familiar decimal number system. The first representation that comes to mind is called *sign–magnitude*, in which a negative number has the same representation as a positive one except for some mark that signifies the negative sign. Our usual way of writing negative numbers, e.g., "`-6`" for negative six, is an example of sign–magnitude notation. Some computers use sign–magnitude notation for numbers; in binary the sign is usually represented as one bit. The PDP–10 uses a representation called *two's complement* for negative numbers. Before we venture to explain two's complement notation, we will examine an analogous representation, *ten's complement*, in the decimal number system.

4.4.1 Odometer Arithmetic and Ten's Complement Notation

When desk calculators contained gears instead of integrated circuits, some calculators had the interesting property that if you were to subtract one from zero, the result would be a string of nines. Another device that exhibits similar behavior is the odometer — the mileage indicator — found in automobiles. If it is run backwards, the number indicated after zero is a series of nines[1]. Another way to describe this is that when the odometer indicates | 9 | 9 | 9 | and we add one to it (by driving one more mile) the result is | 0 | 0 | 0 |.

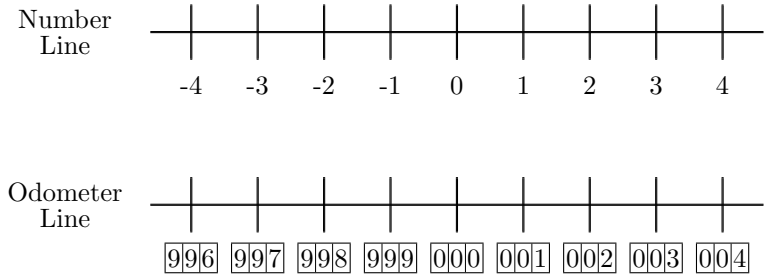
More compactly we could write | 9 | 9 | 9 | + | 0 | 0 | 1 | = | 0 | 0 | 0 |.

Some people find this result disturbing because it appears to violate the commonly held practices of arithmetic. Perhaps we can remedy this discomfort by declaring that in our system of "odometer

[1] Federal law forbids actual experimentation.

arithmetic" the pattern of characters | 9 | 9 | 9 | represents negative one.

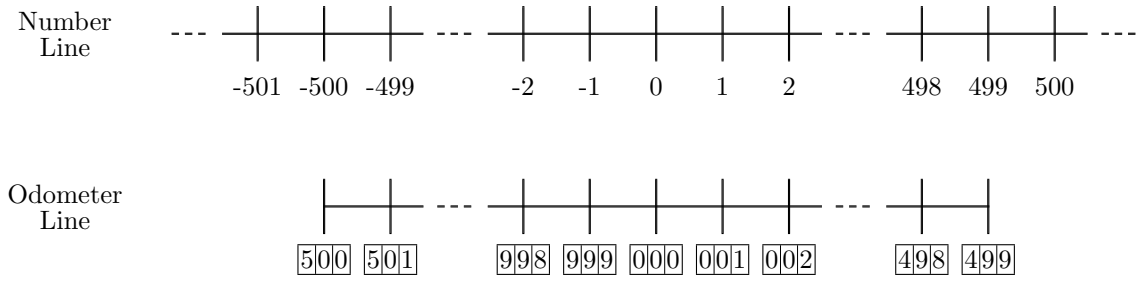
Another way to think of this is to picture the number line on which the numbers are written and compare it to the "odometer line":



This may seem like a strange way to write negative numbers, but it makes some sense. For example, add six to the representation of negative four: Negative four is represented by `996`. Six is represented by `006`. The sum is `002`, which represents two. Let us try another example: add negative three to two. Negative three is represented by `997`. Adding `002` produces `999`, which represents negative one. These examples demonstrate that this representation has an interpretation that makes sense in familiar terms.

There are one thousand different numbers represented on this odometer as it moves through all the states between `000` and `999`. We can partition these one thousand different numbers into two groups. The non–negative numbers are represented by the figures `000` through `499`. The negative numbers are represented by the figures `500` through `999`. There are five hundred distinct numbers in each group.

A more complete comparison of the number line and the "odometer line" appears below:



The number line is infinite in extent, containing all numbers. The "odometer line" that we have defined is finite, including representations for one thousand numbers. These representations have been mapped onto the number line. Note that there is no "odometer line" representation for any number larger than four hundred ninety–nine, nor is there any representation for numbers smaller than negative five hundred.

The figures `999` represent the *ten's complement* of `001`. To compute the ten's complement of any of the figures in this odometer arithmetic system, perform the following steps:

- First, form the nine's complement by subtracting the original number from `999`.

- Then, to the nine's complement, add `1`.

For example, the ten's complement of `123` is formed by subtracting `123` from `999` (the result is `876`) and then adding `1`. The final result is `877`. The ten's complement of `877` is computed by subtracting `877` from `999` (`122`) and then adding `1`. The result is `123`. It is reassuring to note that taking the ten's complement twice restores the original value, preserving the identity `-(-k) = k`.

Now, you might ask, "Why not describe this process as simply a subtraction from `1000`?" Well, that is also a correct way of looking at ten's complement arithmetic. The reason that we choose to describe this as a subtraction from `999` and the addition of `1` is to make the process more closely analogous to the two's complement arithmetic that we are about to describe. From a machine arithmetic view, the subtraction from `999` is easy because it can be accomplished without borrowing; `999` is a more tractable minuend than `1000`.

The figures `500` represent negative five hundred, a number for which there is no positive counterpart. It is inevitable that one representable negative number exists for which there is no positive representation: we have divided the representations equally among the non–negative and negative numbers. Since the non–negative numbers include zero, there must be one fewer positive numbers than there are negative numbers. Thus, there is a negative number that is one larger in magnitude than the largest positive number.

We hope that you are now somewhat fortified for the discussion of two's complement arithmetic that follows. There is a strong similarity between the working of two's complement arithmetic and this "odometer arithmetic" that we have used to demonstrate ten's complement.

4.4.2 Two's Complement Arithmetic

Using some specific number of bits, say, "`n`" bits, it is possible to represent 2^n different numbers. Instead of using these 2^n different patterns to represent non–negative numbers in the range from `0` to $2^n - 1$, we can allocate the 2^n patterns among both the positive and negative numbers. The number of bits, `n`, used in the representation of numbers is often called the length, or the *word length* of the representation.

Let us define `-1` to be the number which when added to `1` results in `0`. When we are using six–bit binary arithmetic, adding the number `111111` to `1` (i.e., `000001`) results in `0`. Thus, we have some reason to adopt `111111` as the representation of `-1` in this six–bit system. By extending this idea to represent the negatives of other numbers, we can define the two's complement system of representation.

In two's complement representation, it is relatively easy to convert from a positive number to a negative one and vice versa. To find the two's complement negation of a number, first subtract the original number from a number composed of all ones.[2] The result of this subtraction is called the *one's complement* of the original number. To form the two's complement, add `1` to the one's complement of the original number. In the computer, the operation of subtracting the original number from all ones (i.e., from `111111`) is accomplished by changing all the zeros in the original number to ones, and all the ones to zeros.

The examples below illustrate the negation of six–bit numbers by conversion to their two's complement form; the results are negated again to demonstrate the identity `-(-k) = k`:

[2]This operation is the analog of subtracting a decimal number from `999`.

Original Number		One's Complement		Two's Complement		One's Complement		Two's Complement
000001	⇒	111110	⇒	111111	⇒	000000	⇒	000001
011111	⇒	100000	⇒	100001	⇒	011110	⇒	011111
100011	⇒	011100	⇒	011101	⇒	100010	⇒	100011
000000	⇒	111111	⇒	000000	⇒	111111	⇒	000000
100000	⇒	011111	⇒	100000	⇒	011111	⇒	100000

In these examples, adding `1` to `111111` produces `000000` instead of `1000000`, because the length of the representations is limited to six bits. The carry out of the leftmost bit is discarded because there is no place to put it. In this case, despite discarding the carry, the result is correct (in terms of two's complement arithmetic).

Another way of thinking of two's complement arithmetic is that the weight of the leftmost bit has been negated. The leftmost bit is sometimes called the *sign bit*. In all non–negative numbers the sign bit is zero. In all negative numbers it is one. An equal number of non–negative and negative numbers are representable. Since `0` is included among the non–negative numbers, the most negative number is one larger in magnitude than the largest positive number.

On a computer that uses the two's complement system in a six–bit word, we could write binary patterns in the grid depicted below:

-32	16	8	4	2	1	Column Weight			
0	0	0	0	0	0	=	0		
0	0	1	0	0	1	=	9	=	8 + 1
1	0	0	0	0	0	=	-32		
1	0	0	0	0	1	=	-31	=	-32 + 1
1	1	1	1	1	1	=	-1	=	-32 + 16 + 8 + 4 + 2 + 1

There are several other systems for representing negative numbers in binary, such as *sign–magnitude* and *one's complement*. The two's complement system is advantageous because the normal rules of unsigned binary arithmetic apply without change to the addition of numbers in the two's complement system. The operation of subtraction, `A-B` is simply computed as the addition of `A+(-B)`[3].

In a positive number, any number of zeros can be added at the left end of the number without changing the value of the number. In a negative number it is the ones at the left end of the word that are non–significant.

4.4.3 Overflow in Two's Complement

In some cases the carry out of the leftmost bit is significant. In such cases, the computer thinks that you have made an error and it will so inform you. (Such errors are actually rather common, and in many cases you might choose to ignore them.) One instance shown in the examples above where the carry signifies an error is the case of taking the two's complement of `100000`. Taking the one's complement of this number produces `011111`. Adding `1` to form the two's complement yields `100000`, which is not the correct result. In fact, there cannot be any correct result since the

[3]Technically, this is computed by `A` + (the one's complement of `B`) + 1, without forming `-B` explicitly.

original number, 100000, represents -32, and +32 is not representable in the word length we have chosen. The way that the computer detects an error of this kind is by comparing the carry out of the leftmost bit to the carry into the leftmost bit. If these carries are the same (both 0 or both 1) then no error has occurred. If the carries differ, the result is wrong. In the example of taking the two's complement of 100000, the carry out of the leftmost bit was 0 but the carry in was 1.

In a computer implementation of the two's complement system, you must remember that the resulting number is limited to a fixed number of bits. Errors, i.e., the calculation of numbers that are not representable, are signified by the carry into the leftmost bit being different from the carry out of that bit. This kind of error is called an *overflow*; an overflow generally means that the result of some computation is not a representable number.

Some example additions (which are also additional examples) may help to clarify this:

```
      000001    ( 1)    000001    ( 1)    110011   (-13)    010000   ( 16)    110011   (-13)
     +111111   ( -1)   +110001   (-15)   +110000   (-16)   +010011   ( 19)   +101100   (-20)
      000000       0    110010   (-14)    100011   (-29)    100011   (-29)    011111    (31)
Carry in:  1                0                 1                1                0
Carry out: 1                0                 1                0                1
                                                           (error)           (error)
```

In the PDP–10 the word length for arithmetic operations is (usually) thirty–six bits. The leftmost bit is called bit number 0. The next bit to the right is bit number 1. The carry out of bit number 0 is called Carry0; the carry out of bit 1 (into bit 0) is called Carry1. The state of Carry0 and Carry1 determine whether the CPU thinks that there has been an overflow. The state of these carries is saved, and can be examined by some of the PDP–10 instructions.

4.5 Octal Notation

Because binary numbers are so unwieldy — they are more than three times longer than the decimal numbers that we are used to — people adopt a more compact notation for dealing with these quantities. Two such notations are popular, octal and hexadecimal. Octal is based on grouping three bits together into an octal digit. Hexadecimal takes four bits at a time. Traditionally, people who program the PDP–10 have used octal notation; hexadecimal is popular on some other computer systems. These notations allow us to write numbers more compactly without concealing the underlying structure of the binary numbers. The selection of either base eight or base sixteen is made because of the ease of converting between these numbers and binary and vice versa.

The three bits that form each octal digit can represent any one of eight states. These eight states are represented by the digits 0, 1, 2, 3, 4, 5, 6, and 7. The rightmost column of digits has weight 1. Moving left, the column weights are multiplied by eight at each column. Thus, the second column has weight 8, the third has weight 64, etc.

Some examples of decimal, octal and binary numbers are presented in Table 4.1.

4.6 Converting Between Number Systems

There is a definite algorithm that we can use to convert from one number system to another. In the number system of the original number, we divide the given number by the base of the target number system. Divide the resulting quotient, until the quotient becomes zero. The remainders that are

Decimal	Octal	Binary
0	0	0
1	1	1
2	2	10
5	5	101
7	7	111
8	10	1000
10	12	1010
13	15	1101
25	31	11001
31	37	11111
64	100	1000000
100	144	1100100
128	200	10000000
512	1000	1000000000
1000	1750	1111101000
1024	2000	10000000000
-1	...7777	...111111111111
-10	...7766	...111111110110
-32	...7740	...111111100000

Table 4.1: Decimal, Octal and Binary Equivalents

calculated during this process become the digits of the result.

For example, we shall convert the decimal number 123 to octal. We begin by dividing 123 by 8. The quotient is 15 and the remainder is 3. Next we divide the quotient, 15, by 8 again. The new quotient is 1; the remainder is 7. Finally, we divide 1 by 8. The quotient is 0 and the remainder is 1. Since the quotient is now 0 we can stop dividing. The remainders that were calculated, 3, 7, and 1, are the digits of the octal result, in reverse order. Thus, we have computed that the digits 173 are the octal representation of the decimal number 123.

```
  15                          1                          0   ←  third quotient
8)123  ←  original        8)15  ←  first           8)1   ←  second quotient
 -8       number           -8       quotient        -0
  43                         7  ←  second            1  ←  third remainder
 -40                            remainder
   3  ←  first
         remainder
```

We should verify this result. In octal, 173 means 1*64 + 7*8 + 3*1, which is 64+56+3 or 123 decimal.

We can of course convert octal 173 back to decimal in the same way as before. The catch is that since 173 is already in octal, we must do all our arithmetic in octal. The desired new base, decimal 10 is octal 12. Follow these steps closely:

```
   14                      1                      0   ←  third quotient
12)173  ←  original    12)14   ←  first       12)1   ←  second quotient
  -12      number        -12       quotient      -0
   53                      2   ←  second          1   ←  third remainder
  -50                         ←  remainder
    3  ←  first
          remainder
```

The remainders, in reverse order, represent the digits of the corresponding decimal number, 123.

There are two things about this example that might be disturbing. The first is the multiplication of 4*12 = 50. In octal 4*10 is 40, and 4*2 is written as 10. The sum, of course, must be 50 in octal. The second difficulty is that a remainder of octal 10 or 11 may result during this conversion process. These numbers represent the familiar digits 8 and 9 of the decimal number system. So, if a remainder appears as 10 or 11, write the corresponding decimal result as the digit 8 or 9, respectively.

The reason that this process works is fairly simple. Consider again the problem of converting decimal 123 to octal. In octal the representation of the number is some series of digits, say, X, Y, Z. The octal number XYZ can be decomposed into XY0+Z. The octal number XY0 represents a multiple of 8, and Z must be some number between 0 and 7. Since XY*8+Z must equal decimal 123, the digits XY must represent the quotient of 123 divided by 8 and Z must be the remainder. So, when we divide 123 by 8 the remainder is the least significant digit of the octal result; the other digits of the result can be formed from the quotient.

4.7 Octal Numbers in the PDP–10

Since octal notation groups three binary digits into one octal digit, the thirty–six bits of a PDP–10 word may be written as twelve octal digits. For example, the octal value 254000000145 might represent the contents of one word.

Often, to make reading the number easier, we write two commas to separate the number into a left half (bits 0:17) and a right half (bits 18:35). The example value in the previous paragraph might be written as 254000,,145; note that the leading zeros in the right half have been omitted in this representation.

4.8 The ASCII Code

We have remarked at length that everything inside the computer is a number. In order to store text or characters within the computer it is necessary to translate each letter or symbol into a number. In principle, any translation would do, but by convention, the PDP–10 has adopted one standard representation for text.

The DECSYSTEM–20 uses a representation called ASCII (the American Standard Code for Information Interchange) to communicate between the computer and its peripheral devices such as terminals and printers. This same code is also used for intermediate storage of data files on the disk.

The version of the ASCII code that we use stores each character in seven bits. Seven bits allow for 128 possible characters. Ninety–four of these codes correspond to graphical symbols: letters, digits, punctuation, etc. One code "prints" as a blank space. The remainder are *control characters*, some of which have special functions when sent to terminals, printers, or other devices.

In Table 4.2 we present the 7–bit ASCII code used in the DECSYSTEM–20. Interpret this table

of ASCII characters by adding the row label and the column heading corresponding to a given character. For example, the character `D` appears in column `4` at row `100`, thus `104` is the code for `D`.

The assembler knows quite a lot about the ASCII code, so it usually isn't necessary to memorize the ASCII character set. Nevertheless, it is a good idea to remember some of the special characters, such as carriage return, line feed, horizontal tab, and space. Note also that the codes for the digits are a compact set. That is, the code for the character `9` is nine (octal `11`) greater than the code for the character `0`. We shall have occasion to make use of this fact. The upper–case alphabet is also compact, and the lower–case letters are related to the upper–case letters in a straightforward mapping: we add octal 40 to the code for an upper–case letter to obtain the corresponding lower–case character.

	0	1	2	3	4	5	6	7
000	NUL							BEL
010	BS	HT	LF	VT	FF	CR		
020								
030				ESC				
040	SP	!	"	#	$	%	&	’
050	(	)	*	+	,	-	.	/
060	0	1	2	3	4	5	6	7
070	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	[	\	]	^	_
140	‘	a	b	c	d	e	f	g
150	h	i	j	k	l	m	n	o
160	p	q	r	s	t	u	v	w
170	x	y	z	{	|	}	~	DEL

Special Characters

NUL	Null Character
BEL	Bell
BS	Backspace
HT	Horizontal Tab
LF	Line Feed
VT	Vertical Tab
FF	Form Feed
CR	Carriage Return
ESC	Escape
SP	Space
DEL	Delete

Table 4.2: The ASCII Character Set

The characters with values smaller than 40 do not correspond to graphic symbols. Generally these characters are called *control characters*. Terminals, printers, and programs may respond in various ways to control characters. For example, carriage return, octal `15`, usually moves a print head or terminal cursor to the left margin; line feed, octal `12`, advances the paper or the video screen to the next line. The null character, octal `0`, is often used by software to signal the end of a string of text.

Programs may use control characters for special purposes. As terminal input, `CTRL/C`, octal `3`, summons the EXEC program; typing `CTRL/T` causes the EXEC to print a line of program and system status.

4.8.1 The ASCII and ASCIZ Pseudo–Operators

Now that we know something of the ASCII code and octal numbers, we can discuss the function of the `ASCII` and `ASCIZ` pseudo operators.

Consider the string of text: *This is ASCII.* Normally, when the assembler sees text such as *This* in a program, it attempts to look up the definition of the symbol `THIS` in its symbol table. To signify that the text *This is ASCII* is meant as a string, we surround it with delimiters and use some pseudo–op, such as `ASCII`, that specifies how to translate the text into the binary representation for the computer.

When the assembler sees the `ASCII` pseudo–op, in a context such as `ASCII/This is ASCII/`, it accepts the text within the delimiters as characters to translate into an ASCII text string. The character "T" becomes `124`, the character "h" becomes `150`, etc. As it translates characters according to the ASCII code, the assembler stuffs the resulting numbers into sequential fields and sequential words. Each of these numbers is seven bits wide; the assembler places the numbers corresponding to five characters into each word:

Text	Octal and Binary Character Data						36–Bit Octal
	0 6	7 13	14 20	21 27	28 34	35	
This␣	124	150	151	163	040		
	1010100	1101000	1101001	1110011	0100000	0	523215171500
is␣AS	151	163	040	101	123		
	1101001	1110011	0100000	1000001	1010011	0	647464040646
CII	103	111	111	000	000		
	1000011	1001001	1001001	0000000	0000000	0	416231100000

When the assembler runs out of text in the `ASCII` pseudo–op, it fills the remainder of the final word with zero bytes (called nulls). Since five 7–bit characters don't fill the entire word, bit 35 is always left as zero.

If the text given to the `ASCII` pseudo–op includes precisely some multiple of five characters, no null characters will be added to the final word. If a null character is desired, the `ASCIZ` pseudo–op guarantees at least one null following the text of the string. The text format generated by the `ASCIZ` pseudo–op is frequently used to communicate with the operating system and peripheral devices; the null byte (guaranteed by the `ASCIZ` pseudo–op) signifies the end of the string.

When a system call such as `PSOUT` is executed, the computer must have a means to determine the limit of the text string. In some other computer systems, in addition to requiring the address of a string, a system call such as `PSOUT` might require the length of the string as an explicit argument. We find that it is easy to use `PSOUT` since it figures out the length by itself. But there is one disadvantage: a routine such as `PSOUT` cannot be used to send the null character to a terminal. Usually this isn't really a problem; most terminals ignore any nulls that are sent to them.

The byte instructions that we discuss in Section 11, page 131 facilitate the manipulation of text that has been packed into words in this way.

4.9 Exercises

4.9.1 Decimal to Binary Conversion

Convert the following decimal numbers to the binary number system:

```
  27      39      144     255     768
 999     1020    1599    2060    4201
```

4.9.2 Decimal to Two's Complement Conversion

Convert the following decimal numbers into the appropriate two's complement form. Assume the word length is twelve bits. Are any of these numbers unrepresentable in twelve bits?

```
  -28     -43     -159    -206    -876
-1014   -1025   -1604   -2067   -4201
```

4.9.3 Binary to Octal Conversion

Express the results from the two previous exercises as octal numbers.

4.9.4 ASCII Text Assembly

By hand, assemble the following examples into 36–bit PDP–10 words. Show the results in binary and in octal.

```
ASCII /What?/
ASCIZ /Which/
ASCIZ /A stitch in time saves nine/
ASCIZ /A wise man doesn't play leapfrog with a unicorn/
```

Chapter 5

PDP–10 Instructions

In this section we discuss several fundamentals of PDP–10 assembly language programming. The three important ideas in this section are:

- what instructions look like in memory,
- what to write to make the assembler do what you want, and
- the meaning of the various parts of an instruction.

These ideas apply to all computer instructions that you write; it is extremely important that you understand these three ideas and the relationship between them.

5.1 Instruction Format in Memory

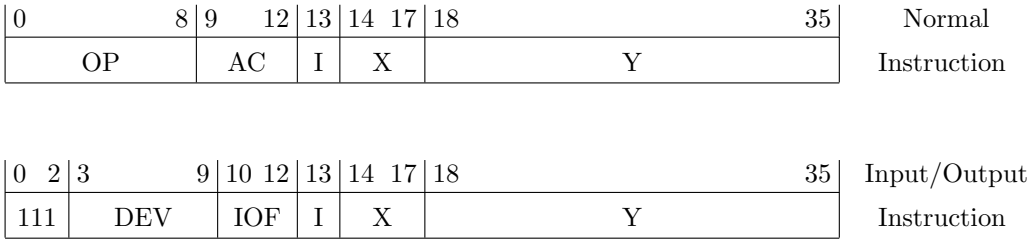
Every machine instruction occupies precisely one word of memory. There are two formats for instructions; however, one of these formats is illegal in user mode, so you aren't expected to have much occasion to use it. These two formats are shown in Figure 5.1.

Recall that we number the bits of a word from left to right, from `0` to decimal `35`. Instructions are stored as words in the PDP–10. Each instruction word is logically subdivided into areas, called *fields*, that have names. Except in privileged programs, Input/Output instructions are illegal. Thus, you are not expected to have occasion to write any Input/Output instructions soon. For completeness, we mention that the Input/Output instructions differ from the normal instruction format.[1]

5.2 How the Assembler Translates Instructions

In assembly language, we write one instruction on a line. Each instruction line in assembly language is translated to one machine instruction. The assembler follows very simple rules for performing this translation. Essentially, a one–to–one correspondence exists between things written in assembly language and instruction fields that are assembled.

[1] In the KS10 (2020) and XKL–1 (TOAD) no Input/Output instructions exist as such.



OP	Operation Code	Bits 0:8
AC	Accumulator Field	Bits 9:12
I	Indirect Bit	Bit 13
X	Index Field	Bits 14:17
Y	Address Field	Bits 18:35
DEV	Input/Output device number	Bits 3:9 of I/O instructions
IOF	Input/Output function code	Bits 10:12 of I/O instructions

The Input/Output instructions are absent from the KS10 and TOAD

Figure 5.1: PDP–10 Instruction Formats

For the moment, we are concentrating on how the assembler translates what we write into fields of machine instructions as they are stored in memory. The meaning of these fields, particularly the significance of the index register and indirect bit in the calculation of an effective address will be discussed in Section 5.3, page 45.

A PDP–10 instruction (other than an Input/Output instruction) includes the following fields:

- the operation code,
- an accumulator field,
- an indirect bit,
- an index register field, and
- an address field.

In assembly language we write each instruction in the following sequence. If any field is omitted, that field will be assembled as zero. However, when we make use of an operand whose value is `0`, we write the `0` explicitly.

- First, we write the mnemonic operation name. For example, we have talked of the `MOVE` operation. The assembler translates the operation name to a number and stores that number in the `OP` field (bits `0:8`) of the instruction word that is being assembled. Usually the mnemonic operation name is indented by one tab; this leaves room at the left margin for labels. Follow the operation name with a space or tab character.
- Following the operation name, if an accumulator is needed, write the accumulator number (or a symbolic name for the accumulator). Write a comma after the accumulator specification.

This number (or the number corresponding to the symbolic name) is placed in the `AC` field (bits `9:12`) of the word being assembled. If no accumulator specification is needed in an instruction, don't write anything for the accumulator field; zero will be assembled.

- The next fields specify the address. The address portion defines the *effective address* (see Section 5.3, page 45) of the instruction. The `I`, `X`, and `Y` fields that are present in every instruction contribute to the effective address. Although the assembler is quite flexible and permits a departure from the format described below, the `I`, `X`, and `Y` fields are conventionally written in the following sequence:
  - When an at–sign character (`@`) is present in the address portion of the instruction, the `I` bit will be set to `1`, signifying indirect addressing. Otherwise, the `I` bit will be `0`.
  - The `Y` field is set from the number, symbolic name, or expression representing the address portion.
  - If a non–zero `X` field (index register) is wanted, the desired index register name or number is placed in parentheses following the address field.
- Finally, a comment can be written on any instruction line. Comments form a very important part of every assembly language program. In MACRO, a semicolon (in most circumstances) makes the remainder of the line a comment.

Some of the general forms in which we write instructions are shown below:

`OP`		All unspecified fields are zero
`OP`	`Y`	AC, I, and X are zero
`OP`	`AC,`	I, X, and Y are zero
`OP`	`AC,Y`	The most usual. I and X are zero
`OP`	`AC,Y(X)`	I is zero
`OP`	`AC,@Y`	Including "`@`" sets the I bit; X is zero
`OP`	`AC,@Y(X)`	This is the most complex form

Some specific examples appear below. The assembler generally is free–format. Spacing and capitalization are not important. Only one instruction is written per line. By convention, instructions are usually indented by one tab to leave room at the left margin for labels. This improves readability. Some people leave a tab after the operation code; others leave a space. In the examples, unless otherwise stated or implied from context, all numbers (except bit numbers) are written in octal notation.

Our first example is quite simple:

```
JFCL
```

Here, `JFCL` is the operation code; `JFCL` is translated to octal `255`. All other fields are zero. The assembler will build the following binary pattern:

0 8	9 12	13	14 17	18 35	Octal	Source Text
010101101	0000	0	0000	000000000000000000	255000 000000	`JFCL`

Consider another example, one that we have seen before:

```
MOVE    1,1000
```

In this case, `MOVE` is the operation code; the assembler translates `MOVE` to `200`. The accumulator field (`AC` portion) is `1`; the address field (`Y` part) is `1000`. There is no indirect addressing and no indexing. This instruction assembles to the following binary pattern:

0 8	9 12	13	14 17	18 35	Octal	Source Text
010000000	0001	0	0000	000000001000000000	200040 001000	`MOVE 1,1000`

A more complicated example shows how we set the `X` field of an instruction:

```
HRRZ    17,1(3)
```

Here, `HRRZ` is the opcode with value `550`; the accumulator field is `17`. The `Y` portion of the address is `1`. The `3` in parentheses signifies the value of the `X` field. This assembles the following binary word:

0 8	9 12	13	14 17	18 35	Octal	Source Text
101101000	1111	0	0011	000000000000000001	550743 000001	`HRRZ 17,1(3)`

Our next example requests indirect addressing:

```
SOS     12,@17240
```

In this instruction line, `SOS` is the opcode with value `370`; the accumulator field is `12`. There is no `X` field, but the "`@`" character specifies that indirect addressing is to be used; the assembler sets bit `13`, the `I` bit, to a `1` because the "`@`" is present. The address (`Y` field) is `17240`. This assembles a binary pattern that looks like this:

0 8	9 12	13	14 17	18 35	Octal	Source Text
001111000	1010	1	0000	000001111010100000	370520 017240	`SOS 12,@17240`

We have mentioned that the accumulators can be used as normal memory locations whenever it is convenient to do so. If an address in the range from `0` to `17` appears in the `Y` field, an accumulator is being referred to as memory. There is often confusion about whether to reference an accumulator as memory or as an accumulator. The following two examples contrast some of the differences. First consider the instruction:

```
AOSGE   5
```

In this example, accumulator number `5` is being referenced as a memory location. The accumulator field has been omitted and is assembled as zero. The binary pattern assembled for this instruction is

0 8	9 12	13	14 17	18 35	Octal	Source Text
011101101	0000	0	0000	000000000000000101	355000 000005	`AOSGE 5`

Compare the instruction above to this one:

```
AOSGE   5,
```

In this instruction, accumulator number `5` is being referenced as an accumulator. The address field has been omitted and is assembled as zero. The binary pattern is quite different from the previous example. You would be right to expect that these two instructions do different things. The comma makes a big difference!

0 8	9 12	13	14 17	18 35	Octal	Source Text
011101101	0101	0	0000	000000000000000000	355240 000000	`AOSGE 5,`

5.3 Effective Address Computation

Without exception, when the computer executes an instruction it first calculates an *effective address*. The effective address is a 30–bit quantity plus a flag to distinguish between addresses that are local or global. (In the traditional machine, all effective addresses are 18 bits and local.)

In the execution of an instruction, the effective address may be used as data itself, or it may be used to address the operand or result word. The effective address is computed before the operation specified by the instruction takes place. It is not possible for any instruction to affect its own effective address computation in any way, because this computation is finished before the instruction operation is performed.

We have mentioned previously that the extended addressing computer has a kind of split personality: one that knows only the traditional way to compute addresses and another that knows the new way. Of course, there is really only one computer and its complete behavior is described in one figure. That figure is complicated, with a complicated explanation. We begin instead with a simplification: a program and figure that shows only the traditional effective address computation.

5.3.1 Traditional Effective Address Computation

Notation that is used in the effective address flow diagrams is shown in Figure 5.2. This notation is related to, but somewhat different than, the notation that we will use to explain individual instructions.

We define an *Address Word* as any word that is used in the calculation of an effective address. When an effective address does not involve indirect addressing, the sole address word is the instruction word itself. In the traditional effective address computation all address words have the same format, identical to instructions: the `I`, `X`, and `Y` fields are at bits `13`, `14:17`, and `18:35`, respectively.

The program fragment in Figure 5.3 depicts the entire instruction execution cycle, including the traditional effective address computation. This program resembles the Pascal language, but differs slightly in that the notation `<m:n>` denotes that the specific bits `m` through `n` are selected from the memory word. (Remember, we number the bits in decimal.) This program introduces some further CPU internals.[2] Among these are

- MB is the CPU's Memory Buffer, in which the CPU holds a word read from (or to be written to) memory.
- The RUN flag. RUN is true while the computer is running. A privileged instruction can stop the entire computer by setting this flag to false. In the programs that we write there is an analogy to the RUN flag: our program is started by the EXEC's `START` or `EXECUTE` command, and halted by an error or when it executes the `HALTF` command.
- The Instruction Register (IR) holds an image of the instruction portion (bits `0:12`) of the current instruction word. This normally holds the operation code (bits `0:8`) and the accumulator field (bits `9:12`). (In an Input/Output instruction the IR is interpreted as containing the device number (bits `3:9`) and the Input/Output function code (bits `10:12`).)

Figure 5.4 repeats the traditional effective address calculation as a flow diagram.

We will summarize the meaning of the traditional effective address calculation in the following paragraphs. We will then present some examples.

[2] An actual CPU may not contain these registers, but it will contain something to perform the same function.

`E`	The effective address resulting from the interpretation of the `I`, `X`, and `Y` parts of the instruction word and any subsequent address words referenced as a result of indirect addressing. In the traditional machine or in section 0, an effective address is an 18 bit quantity. In non–zero sections, `E` is 30 bits.
`G`	The global address flag, a further result of the effective address computation. If `G` is 1, the address is *global*; otherwise it is *local*. (All section zero addresses are local.)
`<n>` or `<n:m>`	As a subscript, $_{\texttt{<n>}}$ denotes bit number `n` and $_{\texttt{<n:m>}}$ denotes the field composed of bits `n` through `m`, inclusive. Bit positions are expressed as decimal numbers.
`C(`α`)`	The contents of the memory word addressed by α. If α is the address of an accumulator, this would be the contents of that accumulator. (An accumulator address is a local address in the range 0 to 17_8 inclusive, or a global address in section 1 to the same range of in–section addresses.)
`I`	The value of the `I` field from the current address word.
`X`	The value of the `X` field from the current address word.
`Y`	The value of the `Y` field from the current address word.
$\Leftarrow$	The assignment operator. The object on the left takes on the value of the expression on the right.
`PC`	The 30–bit contents of the program counter. Program counter bits are numbered `6:35`.
`IR`	The instruction register. It holds the `OP` and `AC` fields of the instruction word. `IR` bits are numbered `0:12`.
`MB`	The memory buffer. This 36–bit register holds the latest word read from memory.
$\wedge$	Logical AND, i.e., true when the left operand and the right operand are both true.
$=$	Numeric equality
$\neq$	Numeric inequality.

Figure 5.2: Notation Used in Effective Address Flow Diagrams

```
RUN := TRUE;                        (* The computer runs while RUN is true.         *)
PC := STARTPC;                      (* The program counter is initialized.          *)
WHILE RUN DO BEGIN                  (* The instruction fetch and execution loop:    *)
  MB := MEM[PC];                    (* Get the instruction word addressed by the PC *)
                                    (* This is the first address word.              *)
  IR := MB<0:12>                    (* Instruction Register is set from bits 0:12   *)
                                    (*    of the instruction word.                  *)
  REPEAT                            (* Compute the effective address:               *)
    Y := MB<18:35>;                 (* Initialize I, X, and Y fields from the       *)
    X := MB<14:17>;                 (*    address word contained in MB.  Initially, *)
    I := MB<13:13>;                 (*    this is the instruction word.             *)
    IF X = 0 THEN E := Y            (* In Direct Addressing, the effective address, *)
                                    (* in E, is the Y field of the instruction.     *)
             ELSE E := Y +          (* In Indexed Addressing, the effective address *)
                     MEM[X]<18:35>; (* is the sum of the contents of index register *)
                                    (* X plus the Y field of the instruction word.  *)
    IF I = 1 THEN MB := MEM[E];     (* If I isn't zero, read the next address word  *)
                                    (* from location E. Repeat with new data in MB. *)
  UNTIL I = 0;                      (* The calculation is finished when an address  *)
                                    (* word is found in which the Indirect field is *)
                                    (* zero.  While I is one, the effective address *)
                                    (* computation will loop.                       *)
  PC := PC + 1;                     (* Advance the program counter.                 *)
  ExecuteInstruction;               (* Execute the instruction found in IR[0:12]    *)
  END;                              (*     and loop to the next instruction.        *)
```

Figure 5.3: Instruction Loop Including the Traditional Effective Address Calculation

In the most usual case, where the `I` and `X` fields of an instruction are both zero, the effective address is just the `Y` field (bits 18:35) of the instruction word.

If the `X` field is non–zero, then `X` specifies the particular accumulator (one of the registers `1` through `17`) that is to be used as an *index register*. The value that is contained in the specified index register is added to the `Y` field to produce the effective address. The sum is truncated to `18` bits. If the `I` field is zero, that sum is the effective address.

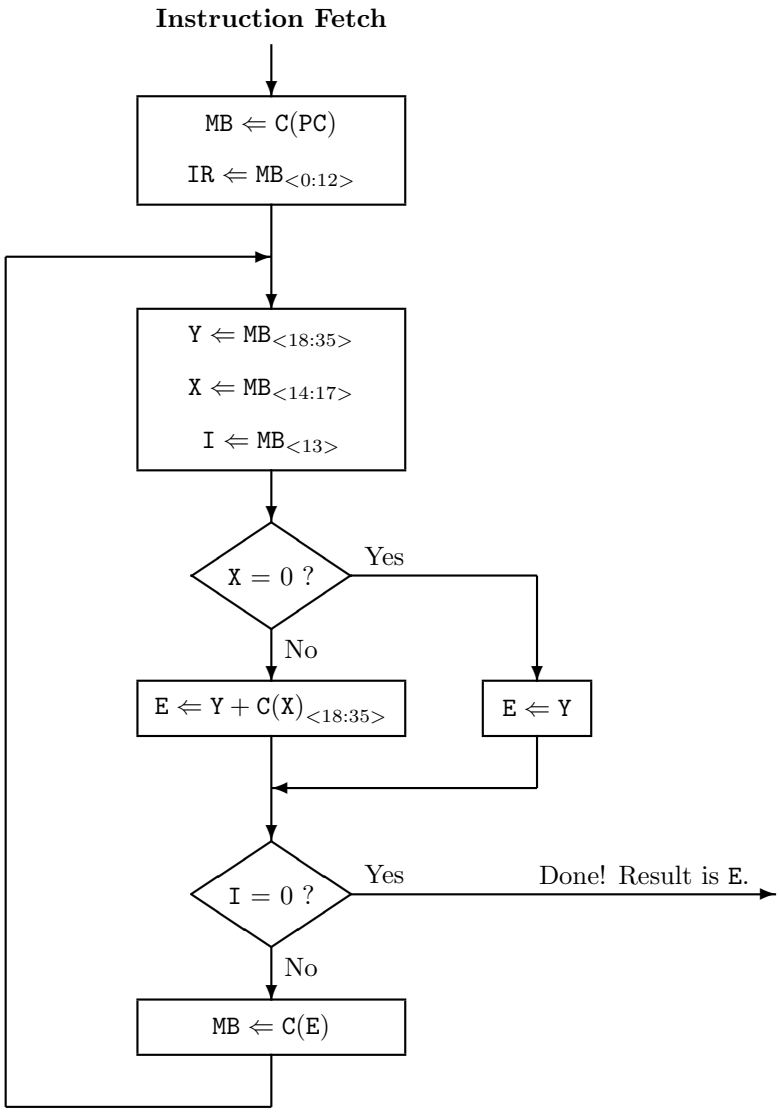
When the `I` field is `1`, *indirect addressing* is called for. To compute an indirect address, the CPU fetches the memory word specified by the effective address computed thus far (by considering the `X` and `Y` fields). That word is assumed to contain `I`, `X`, and `Y` fields in the same format as in an instruction word. The effective address computation continues, with the new values of `I`, `X`, and `Y` as specified in the word that was just fetched. Indirect addressing continues until a word is found in which the `I` bit is zero.

5.3.1.1 Examples of Traditional Effective Address Calculation

The effective address calculation process is very important to our further progress with assembly language. Every instruction calculates an effective address. In the traditional machine, all instructions calculate their effective addresses in precisely this way. (The process described here is a subset of the process found in the extended machines.) The computation of the effective address is the first thing that the CPU does when executing an instruction; the action of the instruction itself does not take place until the effective address has been calculated.

Since it is vitally important that you understand the effective address calculation, we offer several

Figure 5.4: Section Zero Effective–Address Computation



examples that you may find helpful.

Direct Addressing In the most usual case of effective address calculation, the `I` and `X` fields are zero. This is called direct addressing because the `Y` field in the instruction directly specifies the address. For example, consider the familiar instruction

```
MOVE    1,1000
```

In this case, the effective address calculation proceeds as follows:

In the instruction loop program, the instruction word that is addressed by the program counter is read into MB@. MB initializes the instruction register. The instruction register holds bits `0:12` of the instruction; these bits are the operation code and accumulator fields.

Under REPEAT, the value in MB is used to initialize the `I`, `X`, and `Y` variables.

The `X` field of this instruction word contains zero, so E is set from the `Y` field, bits `18:35` of the instruction word. In this example, the value is `1000`.

Since the indirect bit is zero in this instruction, no additional word is read to replace MB and the `UNTIL` clause is satisfied. No repetition of the addressing cycle takes place. The effective address computation is complete; the result, `1000`, is in E.

Again, in direct addressing, the `Y` field of the instruction supplies the entire effective address.

Indexed Addressing Index registers can be used to modify the address of an instruction. One of the common reasons for wanting to modify the address of an instruction is for accessing the data elements in an array or other data structure. Any of the accumulators `1` through `17` (but not `0`!) can be used as an index register to affect the effective address calculation of any instruction.

For example, suppose the symbolic name `TABLE` refers to an array containing `100` (octal) words. The words of this array would have addresses `TABLE+0`, `TABLE+1`, `TABLE+2`, etc., through `TABLE+77`. It is important to remember that when we write an expression such as `TABLE+32` we mean the value of the symbol called `TABLE` (which is an address) plus (octal) `32`. This is unlike most high–level languages in which such an expression would mean the contents of the variable called `TABLE` plus decimal `32`.

When we want to refer to some specific word, we could write a direct address such as `TABLE+43` in an instruction. However, if the address that we want is not explicitly known when we are writing the program, e.g., the address is based on the result of some computation, then we can use indexing to help form the effective address.

Consider the program fragment:

```
          MOVE     3,IDXVAL
          MOVE     1,TABLE(3)

          .  .  .

IDXVAL:   2

          .  .  .


TABLE:    1000
          1234
          2456
          7651
          .  .  .
```

The instruction `MOVE 3,IDXVAL` copies the data in the memory word `IDXVAL` (which contains `2` in this example) to register `3`. As a result of this instruction, register `3` now contains the value `2`.

The instruction `MOVE 1,TABLE(3)` specifies register `3` as an index register. From our previous discussion of how the assembler translates what we write, we know that the `X` field of this instruction word is set to `3`. Suppose that the assembler has placed the array `TABLE` in memory locations starting at location `752`; then the `Y` field of this instruction would contain the value `752`.

From the instruction loop program, we can see that since `X` is non–zero, E is set from the sum of the `Y` field plus the contents of register `3`. `Y` is `752`; register `3` contains `2`. The sum, placed in E, is `754` or, symbolically, `TABLE+2`. The indirect bit is zero in this instruction, so the effective address computation terminates. The result is `754`.

To summarize, when register `3` contains the value `2`, the address that was written as `TABLE(3)` is effectively `TABLE+2`. The contents of the specified index register have, in effect, been added to the `Y` portion of the instruction. Naturally, any change to the contents of the specified index register would change the result of an effective address calculation involving that index register. If the contents of the location called `IDXVAL` were changed to `15`, execution of this fragment would result in an effective address of `TABLE+15` being computed. Index registers are useful when accessing array elements, lists, and record structures.

As a further example of indexed addressing, suppose accumulator `17` contains the value `555`. Then the instruction

```
        HRRZ    3,-1(17)
```

would have an effective address of `554`, as follows.

The assembler can't fit a 36–bit `-1` into the `Y` field of a word, so it truncates the `-1` to an 18–bit quantity, octal `777777`. The `X` field of this instruction is `17`.

Referring to the program describing the effective address, E is set to the sum of the `Y` field plus the contents of the specified index register. The `Y` field is `777777`; index register `17` contains `555`. These are added; the result of this addition is `1000554`. However, the result is truncated to `18` bits in E. After this truncation, the result in E is `554`. No indirection is called for, so the effective address computation is complete.

There are basically two ways to think of index registers. In our first example of indexed addressing, the index is thought of as an offset to a fixed address that is supplied in the `Y` field of an instruction. For example, when we wrote the address expression `TABLE(3)`, register `3` is used to modify the

address `TABLE`; this is the usual viewpoint when `TABLE` is an *array* (see Section 21, page 297).

The second way of thinking of an index register is to use it as the address of (or pointer to) a *record* in memory.[3] Then the `Y` portion of the instruction represents a field name within the record. In this view, the index register contains the address of the record; the `Y` field is thought of as a modification to the record address. Of course, the arithmetic done by the computer's effective address calculation is the same in either case; it is just our view of which part of the address expression is the *base* of the structure and which part of the address forms the *offset* that differs. We illustrate these two views in Figure 5.5.

Array Access

```
MOVEI   3,5         ;index value
...       TABLE(3)  ;array access
```

Record Access

```
MOVEI   5,RO    ;record origin
...       4(5)  ;offset to the
                ;fourth word in
                ;the record
```

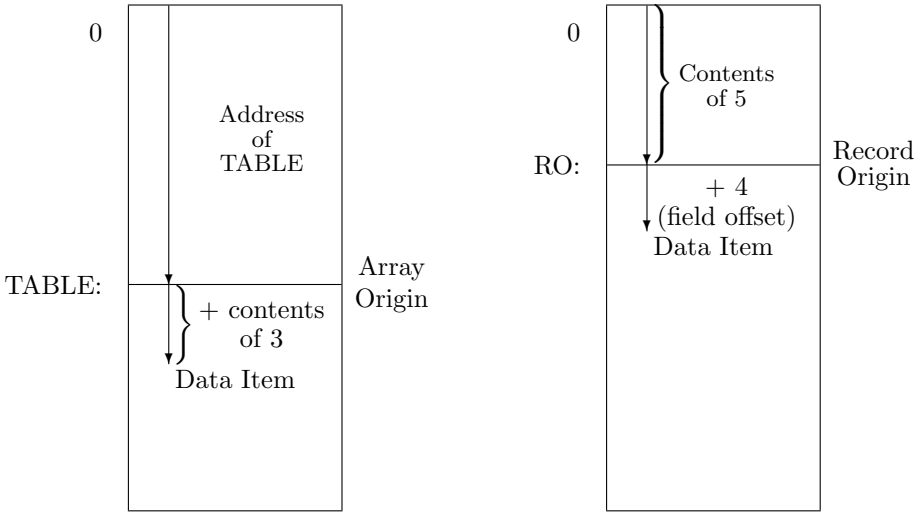


Figure 5.5: Comparison of Array Access and Record Access

To summarize, the use of an index register in an instruction allows the address to be modified under the control of the program.

Before index registers were available in computers, the indexing arithmetic was performed in an accumulator, and the result was stored into the instruction itself. This requires that the program change itself; programs that change themselves are inherently difficult to debug. For this reason, and to increase system efficiency, programs should avoid changing themselves.

Indirect Addressing Indirect addressing is another technique by which the effective address of an instruction can be changed. On the whole, indirect addressing is less frequently used than indexed addressing. However, there are some special situations where indirection is just the right thing; we shall have examples later on where indirect addressing is very helpful.

[3]Records will be discussed in Section 24, page 401.

(It is not absolutely necessary for the reader to understand indirect addressing on first reading. Indirect addressing is used in fewer than 1% of all executed instructions; its frequency in all written instructions is even lower.)

In indirect addressing, the effective address calculated from the combination of the X and Y fields specifies a word that contains a further set of I, X, and Y fields that are used to continue the address calculation.

For example, suppose location 17240 contains the value 167. Then the instruction

```
SOS     12,@17240
```

would have an effective address of 167, as follows:

The Y field is 17240; the X field is zero. Because the assembler sees an at–sign character (@) in the instruction, it sets the I field (bit 13) of the instruction word to 1.

In the instruction loop program, E is set to 17240, as X is zero. However, since I is one, MB is loaded from location 17240 and the UNTIL clause is not satisfied. Hence, a portion of the address calculation must be repeated. The program returns to the statement following the word REPEAT, in which new I, X, and Y values are set from the word in MB. Note that MB now contains a copy of the word at 17240. As postulated above, location 17240 contains 167; this means that the new Y field is 167 and I and X are both zero. E is set to 167. Since the new I bit is zero, the UNTIL clause is satisfied, and the effective address computation terminates with the value 167 in E.

For our final example of tradtional effective address calculation, recall that in a previous example we stipulated that register 17 contains the value 555. Suppose also that location 1767 contains the octal quantity 000017000002. Then, the instruction

```
MOVEM   12,@1767
```

would calculate an effective address of 557, by the following means:

The E is set from the Y field of the instruction, 1767. Since indirect addressing is specified, the contents of 1767 are copied into MB; UNTIL clause is unsatisfied so the REPEAT loop is executed again. The word contained in MB, a copy of location 1767, supplies new values for the I, X, and Y fields; these values are 0, 17, and 2, respectively.

Now, E is set from the value of the Y field plus the contents of register 17. Y is 2 and 17 contains 555. The sum, 557, is placed in E. Since I is now zero, the effective address computation terminates. The effective address in this case is 557.

Because indirect addressing will continue to fetch address words until one is found in which bit 13 is zero, there is a possibility that a mistaken use of indirect addressing can cause a loop that will not terminate. Such a loop is no more harmful (and no more beneficial) than any other kind of non–terminating loop.

5.3.2 Extended Effective Address Computation

Now that we have have mastered the traditional scheme for computing an 18–bit effective address, we turn to the computation of 30–bit addresses. To do so, we introduce new formats of Address Words:

- A *Local Format Address Word* is either the instruction word, or it is an *Instruction Format Indirect Word* (IFIW). A word is an instruction format indirect word when it is read as an address word, i.e., while following an indirect address, and either it is read from section 0 or it contains the binary pattern 10 in bits 0:1.

- A *Global Format Address Word*, also called a Global Indirect Word (GIW), is a word that contains a 0 in bit `0` that is read from a non–zero section while following an indirect address.

Figure 5.6 is a flow chart that depicts the entire effective address computation, both extended and traditional. The result of this calculation is a 30–bit effective address and the flag "G". When "G" is 1, it signifies that the resulting address is global; otherwise, the address is local.

Careful inspection of this diagram reveals that when the latest address word (including an instruction word) has been read from section 0, the computation reduces to precisely the traditional form. Otherwise, the different rules of extended addressing apply.

The complexity of this chart can be traced to three things:

- It encompasses both the traditional effective address calculation and the extended version: some tests in the flow chart depend on whether or not the current address word was read from section 0. This feature is essential: it shows how a user of extended addressing can get to section 0, whether that is intended or not.
- Additional complexity stems from indexed addressing: in extended addressing, the meaning of indexing depends on the data found within the index register.
- The meaning of an address word found via indirect addressing depends both on where the address word came from and what data it contains.

We will summarize the meaning of the extended effective address calculation in the following paragraphs. We will then present some examples.

With reference to Figure 5.6, the box below "Instruction Fetch" makes three important points. First, if PC bits `18:31` are zero, that is, if the in–section value of PC is in the range `0` through `17`, the instruction fetch is from the accumulators: instructions are fetched as if by local addresses. Second, E bits `6:17` are set from the corresponding PC bits: the default section for a local address is the section from which the address word was read. Third, the instruction register is set from bits `0:12` of the instruction word; these do not change during the effective address calculation.

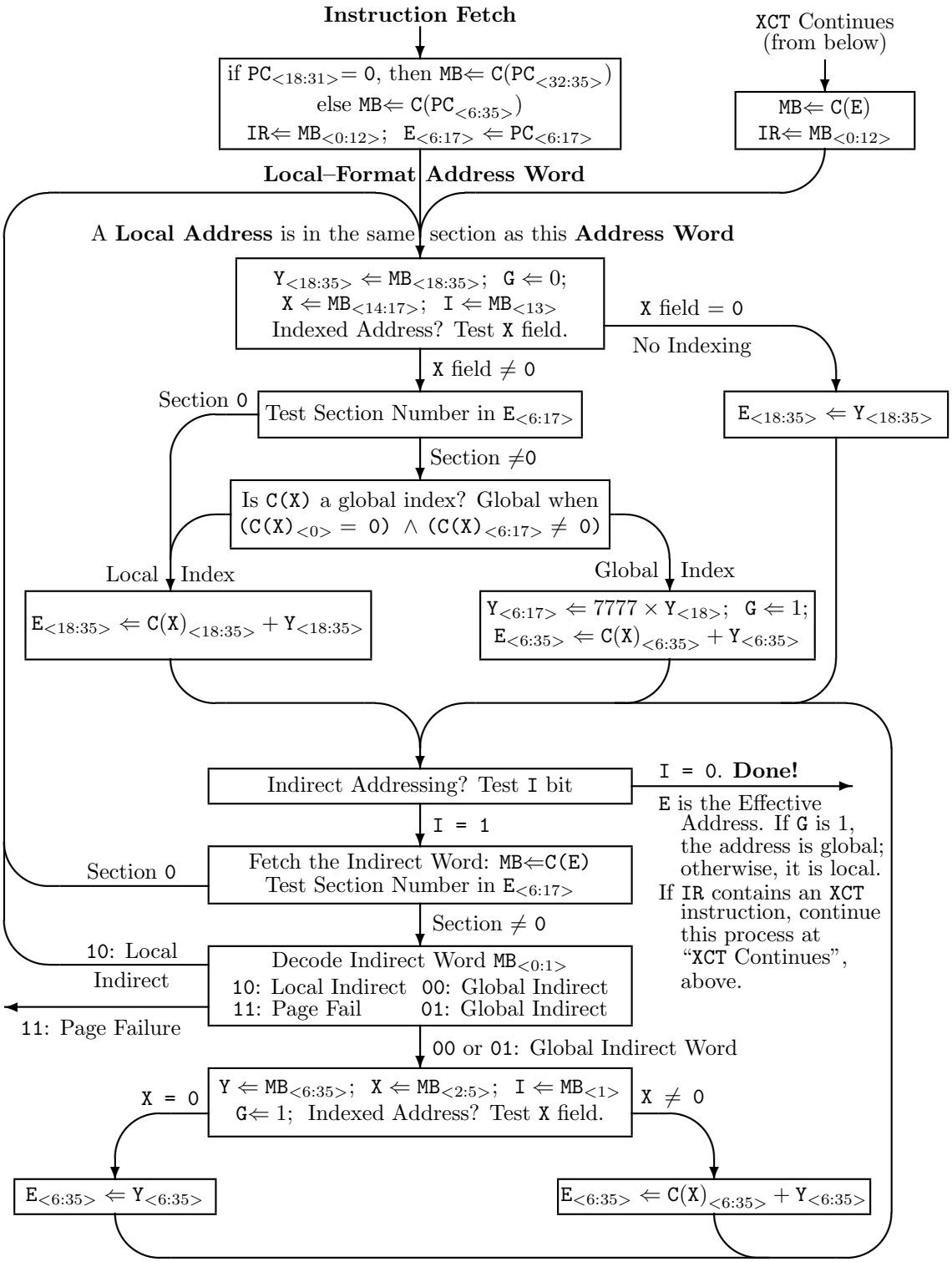
Flow continues to the box below "Local–Format Address Word". The process starts with the instruction word as the first address word. `I`, `X` and `Y` are initialized from the current address word's bits `13`, `14:17`, and `18:35`, respectively. The Global flag is cleared. If a local effective address is computed from this address word, it is local to the section from which the word was read.

The usual case is where the `I` and `X` fields of an instruction are both zero, the result is an effective address composed of the `Y` field (bits `18:35`) of the instruction word and the current section number (from the PC). This is a local address. If the `I` field is zero, this is the effective address.

If the `X` field is non–zero, then `X` specifies the particular accumulator (one of the registers `1` through `17`) that is to be used as an *index register*. If the current address word was read from section 0, traditional rules apply: follow the branch to "Local Index" (explained within the first bullet below). Otherwise, the contents of the specified index register determines which of local or global indexing will be applied:

- If the section field (bits `6:17`) contained in `X` is zero or if bit `0` in `X` is one, local indexing is requested. At "Local Index" (whether by this path or by reading an address word from section zero) the effective address is the 18–bit sum of bits `18:35` contained in `X` plus `Y`. There is no carry beyond 18 bits. This result is a local address.

Figure 5.6: Extended Effective–Address Computation



- If the section field contained in X is non–zero and bit 0 in X is zero then global indexing is requested. At "Global Index", the effective address is the 30–bit sum of bits 6:35 contained in X plus the value Y sign–extended to 30 bits (by duplicating Y bit 18 to bits 6:17). This result is a global address.

In either direct or indexed addressing, when the I field is 0 the value now found in E is the effective address and the calculation is complete. If G is 1 the result is a global address; otherwise, it is a local address.

When the I field is 1, indirect addressing is requested. The CPU reads another address word from memory at the location specified by E as computed thus far.

If the new address word is read from section zero, or if the address word contains the binary pattern 10 in bits 0:1, the word is a local format indirect word. Return to the box beneath "Local–Format Address Word" and continue this process with data from the newly–read address word.

If the address word is read from a non–zero section and it contains 0 in bit 0, it is a global address word. The new I, X, and Y fields are bits 1, 2:5, and 6:35 of this word, respectively. If X is zero, the effective address is the 30–bit value of Y. If X is not zero, the effective address is the 30–bit sum of Y and bits 6:35 of the contents of X. Continue in the flow diagram to the test of I. If I is zero, the process is complete; the effective address is the 30–bit value of E and it is global. If I is 1, the effective address computed thus far locates the next address word to read.

(If the address word is read from a non–zero section and it contains the binary pattern 11 in bits 0:1, the machine traps with a page failure.)

Lastly, there is a baroque appendage labeled "XCT Continues" at the top. We'll discuss this when we get to the XCT instruction.

In summary, the result of the extended effective address computation is a 30–bit address and an indicator to say whether the address is global or local. All section zero addresses are local. (The distinction between a global address and one that is local affects the behavior of a small number of instructions, as will be detailed in the discussion of the affected instructions.)

5.3.2.1 Examples of Extended Effective Address Calculation

In the extended machine, all instructions calculate their effective addresses in precisely this way. The computation of the effective address is the first thing that the CPU does when executing an instruction; the action of the instruction itself takes place after the effective address has been calculated.

Direct Addressing In the most usual case of effective address calculation, the I and X fields are zero. The Y field in the instruction directly specifies the in–section component of the address. For example, consider the familiar instruction

```
MOVE    1,1000
```

In this case, the effective address calculation proceeds as follows:

With reference to the flow chart, Figure 5.6, in the top box, the instruction word addressed by the program counter is read into MB. The contents of MB initialize IR, the instruction register, which holds bits 0:12 of the instruction; these bits are the operation code and accumulator fields. The section component of E is initialized from bits 6:17 of PC; this is the section from which the instruction was read and it will be section in which the resulting local address is located.

In the next box (beneath "Local–Format Address Word"), I, X, and Y are initialized from bits 13, 14:17, and 18--35 of MB, respectively. The X field of this instruction word contains zero, so E bits 18:35 are set from the Y field, bits 18:35 of the instruction word. In this example, the value is 1000. This is combined with the section component (E bits 6:17, set from the PC as described above); the result is a local address. Because the I bit is zero, the effective address calculation concludes.

Again, in direct addressing, the Y field of the instruction supplies the in–section component of the effective address; PC supplies the section numnber. The result is a local address.

Indexed Addressing In the extended effective address calculation, indexed addressing can act as it does in the traditional machine. However, the effect of indexed addressing depends on the contents of the left–half of the selected index register.

Local indexing is very like the traditional scheme. Local indexing applies when either the latest address word was read from section 0, or bit 0 of the index register contents is 1, or bits 6:17 of the index register contain zero. In local indexing the right half of the index register is added to Y and the 18–bit sum is the in–section portion of the local effective address. Local indexing is confined to the section from which the address word (which may be the instruction word) was read. Because local indexing is so similar to the traditional scheme, we shall not give redundant examples.

Global indexing is very different from the traditional scheme. Global indexing applies when the latest address word was read from a non–zero section and bit 0 in the index register is 0 and bits 6:17 of the index register are non–zero. In global indexing a 30–bit global address is formed from the sum of bits 6:35 within X plus the 30–bit quantity created from Y by sign–extension. In this case, sign extension means that bits 6:17 of the extended Y are either all zero or all one, to match bit 18 of Y. This may seem to be a very strange way to form the sum, but there is a purpose to it. When X addresses a record structure, it is sometimes useful to reach addresses that are just a little lower than the beginning of the record. That is, we want an address expression such as -1(5) to address the word prior to the word addressed by 5. In the traditional machine we accomplished this by truncating the sum C(X)+Y to 18 bits. But in the extended machine, X may address a structure in another section so truncations would not always work. But sign–extension does the trick.

We shall offer just two examples of global indexing. Suppose accumulator 7 contains the value 10123400, the address of a record struction in section 10. Then the instruction

```
HRRZ    3,21(7)
```

when performed in a non–zero section would have a global effective address of 10123421, as follows.

The assembler builds the instruction with X containing 7 and Y containing 21. When this instruction is executed, the MB is loaded with the instruction word; IR is loaded from bits 0:12 of MB; bits 6:17 of E are initialized to the (non–zero) section from which the instruction was read.

I, X, and Y are initalized from MB to 0, 7, and 21, respectively. As X is not zero, the indexing branch is followed. There, because the section number in bits 6:17 of E are non–zero, the contents of X are tested to see whether global indexing shall be performed. In this case, bit 0 of X contains zero and bits 6:17 contain 10, so global indexing is performed. Y is sign–extended, with the result being 0000000021. This is added to the contents of X. The sum, 0010123421 is the global effective address.

Second example. Again 7 contains the value 10123400. This time the instruction is

```
MOVE    1,-1(7)
```

When performed in a non–zero section, it calculates the global effective address 10123377, as follows.

As we've seen before, the assembler truncates the -1 to an 18–bit quantity, octal 777777 in Y. The X field of this instruction is 7. When this instruction is executed, the MB is loaded with the instruction word; IR is loaded from bits 0:12 of MB; bits 6:17 of E are initialized to the (non–zero) section from which the instruction was read.

I, X, and Y are initalized from MB to 0, 7, and 777777, respectively. As X is not zero, the indexing branch is followed. There, because the section number in bits 6:17 of E are non–zero, the contents of X are tested to see whether global indexing shall be performed. In this case, bit 0 of X contains zero and bits 6:17 contain 10, so global indexing is performed. Y is sign–extended, with the result being 7777777777. This is added to the contents of X. The sum, 0010123377 is the global effective address.

These two examples show access to records that are possibly remote from the section in which the program resides.

In using global indexing, Y can be thought of as the offset by which to reach a particular field in a structure. The offset is limited to the range -400000 to 377777.

Indirect Addressing Consider an array, BIG, that is allocated in a memory section other than the program's section (the PC section). We can not write MOVE 1,BIG(3) because the address BIG is not local to the program's section. Instead, we can construct a global indirect word that contains the 30–bit address of BIG and a place in which we specify 3 as the index register. Suppose BIG contains 400000 words and is allocated at locations 12770000 through 13367777. Construct a global address word in memory that contains BIG's first address and index register 3. Such a word would contain 030012770000. Let's imagine that we put that word at location 1765 in the program's (non–zero) section. Suppose 3 contains 157. Then the instruction

```
SETOM    @1765
```

when executed in a non–zero section would calculate the global effective address 12770157, as follows.

MB initially contains a copy of the instruction image. E bits 6:17 are initialized from the PC section. IR is initialized from bits 0:12 of MB.

I, X, and Y fields are initialized from MB as 1, 0, and 1765, respectively. The local effective address of in–section 1765 is generated. Because I is not zero, the computer copies the value located at in–section 1765 into MB. Because this word is fetched from a non–zero section, its contents are inspected. Because bit 0 contains 0, the word in MB is recognized as a global indirect word. Now, I, X, and Y are set from bits 1, 2:5, and 6:35, to 0, 3 and 12770000, respectively. Because X is not zero, a 30–bit effective address is calculated from Y plus bits 6:35 of register 3. The sum, 0012770157, is the global effective address.

Thus, we have a technique by which to access an array that is allocated in a memory section other than the program's section. In global indirect addressing note that the 30–bit contents of X is added to the 30–bit Y value supplied in the global indirect word.

5.3.3 Summary

Many people find themselves somewhat confused by their first exposure to this calculation. We do not expect that you fully understand the effects or applications of indexed or indirect addressing at this point. When we come to require these ideas, we will review them. As you gain some experience with this, we hope that you will come to regard the effective address calculation as a straight–forward process.

5.4 Exercises

5.4.1 Instruction Components and Addressing

Part A. In the following lines of code, identify the text that contributes to the `OP` field, and determine the octal value of the fields `AC`, `I`, `X` and `Y`. Values are stated in octal.

```
MOVE    5,1
AOS     674
SOS     5
SKIPE   5,
HRROI   1,5004
XCT     2025(6)
JRST    @672
SETOM   -5(7)
DMOVEM  3,-104230(6)
FMPRI   3,204500
MOVE    2,@561(1)
```

Part B. Assume the instructions are in section 0.

Compute the effective address for each of the instructions listed above. The values of the accumulators and memory locations are as indicated below. (Treat memory locations as in–section addresses where it is plausible to do so.) Calculate each effective address using the values listed below. Ignore any alterations in these values that might result from the execution of this instruction sequence.

```
accumulator 1:              14
accumulator 6:       7,,010427
accumulator 7:  770123,,003612
accumulator 11:      12,,123000
location 575:         1,,000673
location 672:   310000,,001772
location 10,,575:     1,,000673
location 10,,672:   310000,,001772
location 12,,124772:     17,,654321
```

Part C. Now assume the instructions are in section 10. Repeat the effective address calculation for each of the instructions above. Indicate whether the resulting effective address is local or global.

Chapter 6

Data Movement and Loops

Although the PDP–10 has more than 350 different instructions, learning the instruction set is somewhat simplified by the fact that large numbers of instructions fall into general classes whose broad characteristics are easily understood.

Instruction classes are formed by a mnemonic class name and one or more modifier letters. The modifiers usually signify some transformation on the data, or the direction of data movement, or the skip or jump condition. Some functional duplications and some no–ops (i.e., instructions that don't do anything) result from this scheme. However, despite these drawbacks, this notion of instruction classes and modifiers makes the instruction set easy to learn. For example, we shall see there are sixteen full–word `MOVE` instructions, which are four basic types each with four address modifiers. Since the modifiers for all types are the same, we really need to jam only eight facts (the four types plus the four modifiers) into our heads, rather than the sixteen facts (four types times four modifiers).

The power of this scheme is more evident in the half–word class where there are sixty–four instructions, composed of two sources (times) two destinations (times) four other–half specifiers (times) four address modifiers. Thus, rather than remember sixty–four unique instructions (including some that are quite useless), we need remember only 2+2+4+4 ideas. Also, there's some overlap, in that the address modifiers in the half–word class the same as those in the `MOVE` class.

Eighty–four different instructions are presented in this chapter. As you will discover, some of these are among the most frequently used instructions in PDP–10 assembly language programming. On the other hand, some of these are really quite useless. We will comment on the utility of particular instructions; some applications are demonstrated in Section 6.3, page 72.

6.1 Full–Word Data Movement

These instructions include some of the most frequently used instructions in the PDP–10. The general purpose is to move data between memory and the accumulators or vice–versa, occasionally with some transformation of the data.

Recall that the accumulators are the same as memory locations `0` to `17` (octal). The accumulators are special though; an accumulator address appears in all of these data movement instructions. An accumulator holds one of the operands in any arithmetic operation. Therefore, accumulators are an important resource. You will find that any program you write will contain numerous instructions

`E`	The effective address resulting from the `I`, `X`, and `Y` parts of the instruction.
`ER`	an 18–bit quantity composed of bits `18:35` of E.
`EL`	an 18–bit quantity composed of six zeros at the left and bits `6:17` of E.
`C(E)`	The contents of the word addressed by `E`.
`AC`	The value of the accumulator field of the instruction.
`C(AC)`	The contents of the accumulator selected by `AC`.
`CR(E)`	The contents of the right half of the word addressed by `E`.
`CL(E)`	The contents of the left half of the word addressed by `E`.
`L,,R`	The fullword composed of `L` in the left half and `R` in the right half.
`CS(E)`	The fullword composed of the swapped contents of `E: CR(E),,CL(E)`
`C(AC AC+1)`	A double–word accumulator in which `C(AC)` is most significant.
`PC`	The 30–bit contents of the program counter.
`<...>`	The pointed brackets group values into a fullword.
$\wedge$	Boolean (bitwise) AND
$\vee$	Boolean (bitwise) Inclusive OR
$\neg$	Boolean (bitwise) Negation (i.e., One's Complement)
$\oplus$	Boolean (bitwise) Exclusive OR
$\equiv$	Boolean (bitwise) Equivalence

Table 6.1: Notation for Instruction Descriptions

involved with bringing data into the accumulators, modifying the accumulators, and storing results in memory. The `MOVE` class that is described below is most frequently used for the purposes of loading and storing accumulators.

6.1.1 MOVE Class

The `MOVE` class of instructions perform full word data transmission between an accumulator and a memory location. In some cases, minor arithmetic operations are performed, such as taking the magnitude or negative of a word.

There are sixteen instructions in the `MOVE` class. All mnemonics begin with `MOV`. The first modifier specifies a data transformation operation; the second modifier specifies the source of data and the destination of the result. We summarize the sixteen `MOVE` instructions as follows:

`MOV` { `E` no modification; `N` negate source; `M` magnitude of source; `S` swap source halfwords } { ␣ from memory to AC; `I` Immediate : source is 0, , E to AC; `M` to Memory from AC; `S` to Self. From memory to memory. If AC $>$ 0, copy to AC also }

In the "algebraic" representations of these instructions that follow, a number of notational conventions apply. These conventions are explicated in Table 6.1. After you examine this attempt at terminological exactitude, you should be able to understand the details of the `MOVE` class presented below.

MOVE	C(AC)	:= C(E)
MOVEI	C(AC)	:= 0,,ER
MOVEM	C(E)	:= C(AC)
MOVES	C(E)	:= C(E); if AC > 0 then C(AC) := C(E)
MOVN	C(AC)	:= –C(E)
MOVNI	C(AC)	:= –¡0,,ER¿
MOVNM	C(E)	:= –C(AC)
MOVNS	Temp	:= –C(E); C(E) := Temp; if AC >0 then C(AC) := Temp
MOVM	C(AC)	:= |C(E)| i.e., absolute value
MOVMI	C(AC)	:= 0,,ER
MOVMM	C(E)	:= |C(AC)|
MOVMS	C(E)	:= |C(E)|; if AC > 0 then C(AC) := |C(E)|
MOVS	C(AC)	:= CS(E)
MOVSI	C(AC)	:= ER,,0
MOVSM	C(E)	:= CS(AC)
MOVSS	Temp	:= CS(E); C(E) := Temp; if AC > 0 then C(AC) := Temp

The MOVE instruction is the second most frequently executed PDP–10 instruction. It is used to read data from a memory location into an accumulator. MOVE may also be used to copy from one accumulator to another (the effective address names the source accumulator, the accumulator field names the destination). For example,

```
MOVE     7,1000          ;copy the data in location 1000 to
                         ;    location 7

MOVE     16,1            ;copy the data in location 1 (an
                         ;    accumulator) to location 16
```

The MOVEI (*MOVE Immediate*) instruction is also quite popular; it is useful for loading small positive constants into an accumulator. The mode *immediate*, signaled by the letter I in the instruction mnemonic, means that the effective address is the data itself. This contrasts to the usual case where the effective address locates the memory word that contains the data. For example, MOVEI 16,7 loads accumulator 16 with the constant 7. Note the contrast to MOVE 16,7 which loads register 16 with the contents of location 7. MOVEI can be used for numbers in the range from 0 to 777777 (0 to decimal 262,143).

In all but two immediate mode instructions, the operand is 0,,ER, i.e., a 36 bit quantity composed of zero in the left half and bits 18:35 of E in the right half. That is, regardless that E is 30 bits, only 18 bits are used in the immediate operand.[1]

Note that you cannot use MOVEI to load an accumulator with a negative constant. If you wrote MOVEI 3,-6 the assembler would translate the instruction to MOVEI 3,777772. The result in register 3 would be the value 000000777772; but -6 is really 777777777772. Use the MOVNI instruction, e.g., MOVNI 3,6, to load small negative numbers into an accumulator.

[1]The exceptions to this rule are XMOVEI and XHLLI; both of these consider the entire 30–bit effective address to be data.

MOVEM copies the contents of an accumulator to a general memory location. MOVEM is the usual way to store calculated results in a more permanent place. It is the nature of this machine to require the use of the accumulators for intermediate calculations. Since the accumulators are a scarce resource, we often copy the results to memory, thus allowing the accumulator to be used for other calculations.

```
MOVEM   7,1005    ;copy data from location 7 to location 1005
```

MOVEM is *not* the instruction of choice for storing data into an accumulator. Although MOVEM 7,1 works properly to copy data from 7 to 1, the computer works faster when you ask it to execute MOVE 1,7.

The direction of information movement is defined by the particular instruction we use, not by the arrangement of the operands. The order that we write the operands is always the same: first the opcode, then the accumulator field and a comma, finally, the effective address. You must select the opcode appropriate to the desired direction of data movement.[2] Compare the two instructions below. Note that in each case the form or *syntax* is the same. The direction of data movement is defined by the difference in the opcode:

MOVE	AC,MEM	Accumulator ⟵ Memory
MOVEM	AC,MEM	Accumulator ⟶ Memory

The MOVN class computes the two's–complement of the source word. This is the proper way to negate an integer or single–precision floating point number.

The MOVSI instruction is useful for loading constants that have only zero bits in the right half. This is sometimes used for floating–point numbers that represent small whole numbers. MOVSI is also useful for initializing an accumulator with a left–half control count, such as is used in the AOBJN instruction.

The MOVM class computes the absolute value of the source operand. If the source is positive, MOVMx is equivalent to the corresponding MOVEx instruction; otherwise, MOVMx acts like MOVNx. This is the correct way to compute the absolute value of an integer or single–precision floating point number. Note that MOVMI is equivalent to MOVEI since the immediate operand, 0,,E, is always a positive number.

6.1.2 EXCH Instruction

The EXCH instruction exchanges the contents of the selected accumulator with the contents of the effective address.

EXCH	Temp := C(AC); C(AC):=C(E); C(E):=Temp;

6.1.3 XMOVEI Instruction

The XMOVEI instruction is provided for manipulating extended addresses. It is one of only two immediate mode instructions that deal with all 30–bits of the effective address. The extended effective address is copied to the AC.

[2] In contrast, some other computers, e.g,. the PDP–11, define the direction of data movement by the order of the operands.

`XMOVEI`	C(AC) := <EL,,ER>;

In section 0, `XMOVEI` is like `MOVEI`: AC will be set to `0,,E`.

In a non–zero section, `XMOVEI` behaves as follows:

- if `E` is local and `E` specifies an accumulator address in the range from `0` to `17`, AC will be set to the global address of the accumulator: `1,,E`.
- AC will be set to the 30–bit value of `E`. If `E` is local, the left half of AC is set to the section from which the latest address word was read. (If there is no indexing or indirection, this will be the PC section.)

6.2 Jump and Skip Instructions

One of the most powerful tools available to the programmer is the computer's ability to decide whether to repeat groups of instructions. This ability allows the computer to deal with special conditions in a flexible way based on the state of the calculations thus far.

6.2.1 JRST

The most frequently executed instruction in the PDP–10 is `JRST`. The `JRST` instruction actually has several different functions; the particular function is selected by the value in the accumulator field.

When the accumulator field is zero, a `JRST` instruction is just an unconditional jump. When a `JRST` instruction is executed, the program counter is changed to the value given in the effective address of the instruction. That is, the execution of `JRST 12345` will cause the next instruction to be taken from (in–section) `12345`.

If the effective address is in a different section, `JRST` (or any other jump) will transfer control to the other section.[3]

`JRST 0,`	PC := E; Unconditional jump. The AC field must be zero. Note that E is a 30–bit value.

The other functions of `JRST`, selected by other values in the accumulator field, will be discussed in Section 13.2.4, page 168.

6.2.2 Conditional Jumps and Skips

The next sixty–four instructions are eight types with eight modifiers. The purpose of these instructions is to modify the flow of control in the program, the modification being based on the result of

[3]When the program has read an instruction word or an address word from section 0, the effective address will be in section 0. Thus, ordinary jump instruction can not escape from section 0.

an arithmetic comparison.

There are two kinds of modifications of control: jumps and skips. A jump, if the specified condition obtains, will cause the computer to alter its normal sequence of instructions and resume the program at the address specified by the effective address of the jump instruction. A skip, if satisfied, will skip over the instruction that immediately follows the skip. Skips often are placed immediately before unconditional jumps, and have the effect of making such an instruction conditional.

Six of the eight modifiers are arithmetic conditions, such as equal, greater, less or equal, etc. The other two modifiers are "A" meaning *always* jump (or skip), and blank meaning *never* jump (or skip). The eight condition modifiers and the eight jump and skip instructions are displayed below:

SKIP	test memory and skip	␣	Never
AOS	add one to memory and skip	L	if Less Than
SOS	subtract one from memory and skip	LE	if Less Than or Equal
JUMP	test AC and jump		if Equal
AOJ	add one to AC and jump		if Not Equal
SOJ	subtract one from AC and jump	GE	if Greater Than or Equal
CAM	compare AC to memory and skip	G	if Greater Than
CAI	compare AC immediate and skip	A	Always

6.2.2.1 JUMP Class

A JUMP class instruction compares the contents of the selected accumulator to the constant zero. The instruction will jump (i.e., change the PC to be a copy of the effective address of this instruction) if the specified relation is true.

JUMP	No Operation. Do not Jump.
JUMPL	If C(AC) $<$ 0 then PC := E;
JUMPLE	If C(AC) $\leq$ 0 then PC := E;
JUMPE	If C(AC) $=$ 0 then PC := E;
JUMPN	If C(AC) $\neq$ 0 then PC := E;
JUMPGE	If C(AC) $\geq$ 0 then PC := E;
JUMPG	If C(AC) $>$ 0 then PC := E;
JUMPA	PC := E;

It should be noted that the PDP–10 is unique among computers: it possesses an instruction named JUMP that never jumps.

The instruction JUMPA is an unconditional jump. However, the JRST instruction is preferred because JRST is faster than JUMPA on all CPU models.[4]

[4] JRST was discovered to be faster than JUMPA on the first CPU, the PDP–6. As a result JRST was adopted by programmers as the best unconditional jump, making JRST the most frequently executed instruction. Because they knew that JRST was executed with great frequency, the designers of all subsequent processors made a special effort to make JRST the fastest instruction.

We have mentioned that the ASCIZ string format — a string that ends with a zero character — is very popular in the PDP–10; the JUMP class instructions, particularly JUMPE and JUMPN, make the detection of the zero character very easy.

The following loop performs some processing on every character in a string, terminating after processing the zero character that terminates the string:

```
        . . .           ;initialize
LOOP:   . . .           ;get a character into accumulator 1
        . . .           ;process the character
        JUMPN   1,LOOP  ;continue processing until a null has been done
        . . .           ;here after the null character has been processed
```

This loop format is very similar to the Pascal REPEAT ... UNTIL... statement.

It is probably more common to omit the processing of the zero character at the end of the string. To accomplish this, we must move the test for zero into the middle of the loop. This more complex structure often appears as

```
        . . .           ;initialize
LOOP:   . . .           ;do whatever is necessary to get the next character
        . . .           ;into some specific accumulator, say number 1.
        JUMPE   1,LOOPX ;test for the end of string, jump to loop exit if
                        ;a null is seen
        . . .           ;process this character
        JRST    LOOP    ;jump back to get another character.

LOOPX:  . . .           ;here when the string to process has been finished.
```

This example introduces a typographical nuance of no consequence to MACRO: we like to leave a blank line beneath an unconditional break in the flow of the program. The JRST LOOP is not conditional. The program does not come to LOOPX from the instruction lines that immediately precede it, hence we add whitespace to give a hint to the reader to look for another way to get to LOOPX.

In this loop, whatever processing is applied to characters is omitted for the zero character. This approximates the Pascal WHILE ... DO ... statement, but it is somewhat more flexible. Occasionally in a structured language such as Pascal we are forced into an awkward construction because of limits of the control structure. A very common thing to see in Pascal is a fragment such as the following:

```
READ(...);              (* read the first record *)
WHILE NOT eof DO BEGIN
        ...             (* process one record *)
        READ(...)       (* read the next record *)
        END;
```

The occurrence of READ twice is awkward; it is possible to avoid this awkwardness in Pascal by introducing a Boolean function to read a record, but that partially conceals the purpose of the loop. It seems much more natural to write in assembly language something like

```
        . . .                   ;initialize
LOOP:   . . .                   ;read a record
        MOVE    1,EOF           ;get the End of File flag
        JUMPN   1,ENDLOO        ;Jump if EOF is true (non-zero)
        . . .                   ;process the record
        JRST    LOOP            ;go do another record

ENDLOO:                         ;here at end of file.
```

6.2.2.2 SKIP Class

A `SKIP` class instruction compares the contents of the effective address to the constant zero and skips past the next instruction if the specified relation is true. If a non–zero accumulator field appears, the selected AC is loaded from memory.

To say that an instruction *skips* means that it causes the CPU to avoid executing the instruction immediately following the skip. The skip is accomplished by incrementing the program counter one extra time. (Instructions that skip are relatively scarce in other processor architectures: a "skip" works only in the circumstance that every instruction is the same length.)

`SKIP`	If AC > 0 then C(AC) := C(E); do not skip;
`SKIPL`	If AC > 0 then C(AC) := C(E); If C(E) < 0 then skip;
`SKIPLE`	If AC > 0 then C(AC) := C(E); If C(E) ≤ 0 then skip;
`SKIPE`	If AC > 0 then C(AC) := C(E); If C(E) $= 0$ then skip;
`SKIPN`	If AC > 0 then C(AC) := C(E); If C(E) $\neq 0$ then skip;
`SKIPGE`	If AC > 0 then C(AC) := C(E); If C(E) ≥ 0 then skip;
`SKIPG`	If AC > 0 then C(AC) := C(E); If C(E) > 0 then skip;
`SKIPA`	If AC > 0 then C(AC) := C(E); always skip;

As the `JUMP` instruction never jumps, so too the `SKIP` instruction never skips.

Among the uses of the `SKIP` class is the testing of Boolean flags. A flag variable is one that takes on only two values, usually True and False. Although many representations are possible, when full words are used for flags the typical representation of False is `0`; True is represented as `-1`. This choice is made because the Boolean instructions (that we shall discuss in Section 14.2, page 192) allow logical operations (e.g., AND, OR, etc.) to be performed on these quantities. The following four instructions are commonly used for dealing with full–word flags:

```
        SETZM   FLAG            ;set FLAG to zero (false)

        SETOM   FLAG            ;set FLAG to all ones, -1 (true)

        SKIPE   FLAG            ;skip if FLAG is false

        SKIPN   FLAG            ;skip if FLAG is true
```

Suppose BV is the name of a Boolean variable. The Pascal WHILE bv DO ... loop can be approximated by the following assembly language structure:

```
WLOOP:  SKIPN   BV              ;skip if BV is true
         JRST   XLOOP           ;BV is false, exit from While loop
        . . .                   ;execute the body of the loop
        JRST    WLOOP           ;back to the top of the loop.

XLOOP:                          ;here when BV is false.
```

This example introduces another typographical nuance: we try to indent an instruction that is made conditional because it follows one that skips conditionally. This convention has no meaning to MACRO; it serves only to remind the reader that the instruction is somehow conditional.

It should be noted that it is somewhat wasteful to use an entire 36–bit word to store a two–valued flag. The Test class instructions allow a convenient way to manipulate single–bit flags; see Section 14.1, page 189.

6.2.2.3 AOS Class

An AOS class (Add One to memory and Skip) instruction increments (adds 1 to) the contents of a memory location and places the result back in memory. If the accumulator field is non–zero, the incremented result will be copied to the specified AC. Finally, the incremented result is compared to the constant zero. If the specified condition is true, an AOS class instruction will then skip.

AOS	Temp := C(E)+1; C(E) := Temp; If AC $>$ 0 then C(AC) := Temp; Do not skip.
AOSL	Temp := C(E)+1; C(E) := Temp; If AC $>$ 0 then C(AC) := Temp; If Temp $<$ 0 then skip.
AOSLE	Temp := C(E)+1; C(E) := Temp; If AC $>$ 0 then C(AC) := Temp; If Temp $\leq$ 0 then skip.
AOSE	Temp := C(E)+1; C(E) := Temp; If AC $>$ 0 then C(AC) := Temp; If Temp $=$ 0 then skip.
AOSN	Temp := C(E)+1; C(E) := Temp; If AC $>$ 0 then C(AC) := Temp; If Temp $\neq$ 0 then skip.
AOSGE	Temp := C(E)+1; C(E) := Temp; If AC $>$ 0 then C(AC) := Temp; If Temp $\geq$ 0 then skip.
AOSG	Temp := C(E)+1; C(E) := Temp; If AC $>$ 0 then C(AC) := Temp; If Temp $>$ 0 then skip.
AOSA	Temp := C(E)+1; C(E) := Temp; If AC $>$ 0 then C(AC) := Temp; Always skip.

The AOS instruction can be used to increment any memory address including an accumulator, e.g., AOS 5. Note once again that the instruction we write as

```
AOS     5       is a shorthand for      AOS     0,5
```

This instruction adds one to memory location `5` (which is also accumulator `5`). Since the accumulator field is `0`, no accumulator receives a copy of the result.

This is quite different from

```
AOS     5,      or                      AOS     5,0
```

either of which would add one to the contents of location `0` (an accumulator, addressed here as memory) and copy the result to accumulator `5`.

Unless a skip is needed, or unless a second accumulator must be loaded, avoid using `AOS` to increment an accumulator; `ADDI 5,1` is faster. Often, when it is necessary to increment an accumulator, a jump is nearby; see the discussion of the `AOJ` class to combine incrementing an accumulator with a conditional jump.

6.2.2.4 SOS Class

Each of the `SOS` (Subtract One from memory and Skip) instructions decrements (subtracts `1` from) the contents of the memory location specified by the effective address and stores the result back in the same location. The result is compared to zero to determine whether or not to skip. If a non–zero accumulator field appears in any of these instructions then the decremented result will be copied to the selected accumulator.

`SOS`	Temp := C(E)–1; C(E) := Temp; If AC > 0 then C(AC) := Temp; Do not skip.
`SOSL`	Temp := C(E)–1; C(E) := Temp; If AC > 0 then C(AC) := Temp; If Temp < 0 then skip.
`SOSLE`	Temp := C(E)–1; C(E) := Temp; If AC > 0 then C(AC) := Temp; If Temp ≤ 0 then skip.
`SOSE`	Temp := C(E)–1; C(E) := Temp; If AC > 0 then C(AC) := Temp; If Temp = 0 then skip.
`SOSN`	Temp := C(E)–1; C(E) := Temp; If AC > 0 then C(AC) := Temp; If Temp ≠ 0 then skip.
`SOSGE`	Temp := C(E)–1; C(E) := Temp; If AC > 0 then C(AC) := Temp; If Temp ≥ 0 then skip.
`SOSG`	Temp := C(E)–1; C(E) := Temp; If AC > 0 then C(AC) := Temp; If Temp > 0 then skip.
`SOSA`	Temp := C(E)–1; C(E) := Temp; if AC > 0 then C(AC) := Temp; Always skip.

A `SOS` class instruction can be used to increment or decrement any memory address, including an

accumulator. The discussion following `AOS` applies to `SOS` as well. See also the `SOJ` class.

6.2.2.5 AOJ Class

An `AOJ` (Add One to AC and Jump) class instruction increments the contents of the selected accumulator. If the result bears the indicated relation to the constant zero then the instruction will jump to the effective address, otherwise the next instruction in the normal sequence will be executed.

`AOJ`	C(AC) := C(AC)+1;
`AOJL`	C(AC) := C(AC)+1; If C(AC) $<$ 0 then PC := E;
`AOJLE`	C(AC) := C(AC)+1; If C(AC) $\leq$ 0 then PC := E;
`AOJE`	C(AC) := C(AC)+1; If C(AC) $=$ 0 then PC := E;
`AOJN`	C(AC) := C(AC)+1; If C(AC) $\neq$ 0 then PC := E;
`AOJGE`	C(AC) := C(AC)+1; If C(AC) $\geq$ 0 then PC := E;
`AOJG`	C(AC) := C(AC)+1; If C(AC) $>$ 0 then PC := E;
`AOJA`	C(AC) := C(AC)+1; PC := E;

The `AOJ` instruction will increment the selected accumulator without jumping; `ADDI AC,1` is more commonly used for that purpose.

6.2.2.6 SOJ Class

A `SOJ` (Subtract One from AC and Jump) class instruction decrements the contents of the selected accumulator. If the result bears the indicated relation to zero then the instruction will jump to the effective address.

`SOJ`	C(AC) := C(AC)–1;
`SOJL`	C(AC) := C(AC)–1; If C(AC) $<$ 0 then PC := E;
`SOJLE`	C(AC) := C(AC)–1; If C(AC) $\leq$ 0 then PC := E;
`SOJE`	C(AC) := C(AC)–1; If C(AC) $=$ 0 then PC := E;
`SOJN`	C(AC) := C(AC)–1; If C(AC) $\neq$ 0 then PC := E;
`SOJGE`	C(AC) := C(AC)–1; If C(AC) $\geq$ 0 then PC := E;
`SOJG`	C(AC) := C(AC)–1; If C(AC) $>$ 0 then PC := E;
`SOJA`	C(AC) := C(AC)–1; PC := E;

`SOJ` will decrement the accumulator without jumping, but `SUBI AC,1` is preferred for clarity.

The `AOJ` and `SOJ` class instructions are often used for loop control. For example, the following instruction sequence (or *code*) will repeat a loop five times:

```
        MOVEI   15,5            ;loop control count
LOOP:   ...                     ;execute for 15 containing 5,4,3,2,1
        ...                     ;any code that doesn't change 15
        ...
        SOJG    15,LOOP         ;decrement control count and loop
```

6.2.2.7 CAM Class

The `CAM` and `CAI` instructions compare two quantities, one from the accumulator and the other as specified by the memory operand. This is in contrast to the six jump and skip base forms that we have just seen in which data is compared to the constant zero.

Each of the `CAM` (Compare Accumulator to Memory) class instructions compares the contents of the selected accumulator to the contents of the effective address. If the indicated condition is true, the instruction will skip. The `CAM` class instructions are suitable for arithmetic comparison of either fixed–point quantities or normalized single–precision floating–point quantities. For the comparison to be meaningful both C(AC) and C(E) should be in the same format (i.e., either both fixed or both floating).

`CAM`	no op (reads memory)
`CAML`	If C(AC) $<$ C(E) then skip;
`CAMLE`	If C(AC) $\leq$ C(E) then skip;
`CAME`	If C(AC) $=$ C(E) then skip;
`CAMN`	If C(AC) $\neq$ C(E) then skip;
`CAMGE`	If C(AC) $\geq$ C(E) then skip;
`CAMG`	If C(AC) $>$ C(E) then skip;
`CAMA`	(reference memory) skip;

The `CAM` class, and the `CAI` class described below, are the only instructions by which two arbitrary numbers can be compared. Other control instructions implicitly use the constant zero as one of the operands.

6.2.2.8 CAI Class

The `CAI` (Compare Accumulator Immediate) instructions each compare the contents of the selected accumulator to the 36–bit quantity composed of zeros in the left half and ER (bits 18:35 of the effective address) in the right half. If the indicated condition is true, the instruction will skip. The immediate operand is always considered to be a positive number.

`CAI`	no op
`CAIL`	If C(AC) $<$ 0,,ER then skip;
`CAILE`	If C(AC) $\leq$ 0,,ER then skip;
`CAIE`	If C(AC) $=$ 0,,ER then skip;
`CAIN`	If C(AC) $\neq$ 0,,ER then skip;
`CAIGE`	If C(AC) $\geq$ 0,,ER then skip;
`CAIG`	If C(AC) $>$ 0,,ER then skip;
`CAIA`	skip;

The immediate compare instructions are useful in loop control. Another application of these instructions is in character processing. The ASCII characters are all small numbers (from `0` to octal `177`), and so may appear as the immediate operand in a comparison. For example, the following

fragment inspects the character in accumulator 1 to determine if it is a carriage return (octal 15) or a line feed (octal 12):

```
. . .           ;load character into accumulator 1
CAIN    1,15    ;test for carriage return
 JRST   DOCR    ;go process a carriage return
CAIN    1,12    ;test for a line feed
 JRST   DOLF    ;process a line feed.
. . .           ;character is neither carriage return nor
                ;   line feed.
```

We can do other forms of processing as well. The following is a way to test to see if a character (in AC 1) is a lower–case letter:

```
CAIGE   1,141   ;skip if it is >= to lower-case A
 JRST   NOTLOW  ;not a lower-case letter
CAILE   1,172   ;skip if it is <= to a lower-case Z
 JRST   NOTLOW  ;not a lower-case letter
. . .           ;the character is a lower-case letter.
```

This fragment can be improved in two ways. First, rather than be bothered by looking up the ASCII codes for lower–case A and Z, we can let the assembler do some of the work. When we write "a", i.e., the letter a enclosed in double quotes, the assembler will translate the letter to what we call right–justified ASCII. In this case the assembler produces the number 141 as the translation. When ASCII characters are stored in strings, they are left–justified within the computer word. However, due to the nature of the byte instructions (see Section 11, page 131), when single characters appear in an accumulator, they are right–justified. Anyway, to continue, the second way that this fragment can be improved is by means of what we call *nested skips*. We observe that in two cases this program fragment executes the instruction JRST NOTLOW. By means of making one skip instruction skip over another skip instruction, we save writing one of these JRST instructions:

```
CAIL    1,"a"   ;skip if smaller than lower-case A
 CAILE  1,"z"   ;skip if smaller than or equal to lower-case Z
  JRST  NOTLOW  ;either smaller than "a" or larger than "z"
. . .           ;this character is a lower-case letter.
```

(The double indentation of the JRST is meant to attract the reader's attention to the nested skips: this JRST is conditional on two instructions that precede it.)

Here is another example of combining instructions that skip in order to effect the AND of logical expressions. In this case imagine that the variable called I is being used as a subscript of an array that is defined to have legal subscripts in the range from 1 to decimal 100. We want to make sure that the following relation is true: (1 LTE I) AND (I LTE 100). This is easily accomplished by the following sequence:

```
MOVE    1,I     ;copy the value I to an accumulator
CAIL    1,1     ;skip if I is too small
 CAILE  1,144   ;skip if I is LTE decimal 100
  JRST  ARYERR  ;jump to Array Subscript Error routine
```

By the way, this approximates a high–level language statement such as:

```
IF (I < 1) OR (I > 100) THEN GO TO ARYERR.
```

6.2.3 AOBJP and AOBJN

The AOBJ (Add One to Both halves of the accumulator and Jump) instructions allow forward indexing through an array while maintaining a control count in the left half of an accumulator. Use of AOBJN and AOBJP can reduce loop control to one instruction.

AOBJN	C(AC) := C(AC)+<1,,1>; If C(AC) $<$ 0 then PC := E;
AOBJP	C(AC) := C(AC)+<1,,1>; If C(AC) $\geq$ 0 then PC := E;

In the typical use of the AOBJN instruction, the left half of an accumulator is set to the negative of the desired (maximum) number of iterations. The right half of the accumulator is usually initialized either to zero (the MOVSI instruction is good for this) or else it is set to the first address of an array. An application example to demonstrate the AOBJN instruction will appear in the discussion of loops that follows. Further demonstrations will be given in the larger examples.

6.3 Constructing Program Loops

These instructions, the jumps, the skips, and the compares, are quite useful in the construction of program loops. Some simple examples will be shown, as well as the improvements that are possible.

6.3.1 Forward Loops

One of the most frequent loop constructions starts the loop variable at some small constant and counts it by one up to some maximum. This is the usual case in Fortran DO loops and Pascal FOR statements. There are a variety of ways to implement this function. In many cases, it is a good idea to keep the loop variable in an accumulator for easy access to it. The following is an example of one way to do this:

```
        MOVEI   12,5            ;Initial count value is 5
LOOP:                           ;Jump back to here to perform
                                ; the function that is being repeated
        . . .
        ADDI    12,1            ;add the constant 1 to the accumulator
        CAIG    12,7            ;End test.  Skip if the contents of 12
                                ;                   are greater than 7
         JRST   LOOP            ;C(12) are LTE 7, repeat LOOP function
        . . .                   ;leave loop
```

In this example, the loop variable, kept in accumulator 12, takes on the values 5, 6, and 7. As soon as the contents of the accumulator exceed 7, the CAIG instruction will skip, and the program will leave this loop.

This loop can be generalized in several ways. If a variable contains the initial lower bound, then the MOVEI that precedes the label LOOP can be changed to a MOVE that initializes the accumulator with a copy of that variable. Similarly, if the upper bound were in a variable, the CAIG could be changed to CAMG. The following is an example to display this modification:

```
         MOVE    12,LO       ;Initialize count value to lower bound
 LOOP:                       ;Jump back to here to perform
                             ; the function that is being repeated
         . . .
         ADDI    12,1        ;add the constant 1 to the accumulator
         CAMG    12,HI       ;Skip if the AC is greater than high bound
          JRST   LOOP        ;the AC is smaller than high bound, repeat
```

It should be noted that this loop will execute at least once, even though the high bound might be smaller than the low bound. This is typical of the way that the Fortran–IV language implements DO loops.[5]

As long as the step size is one, we can save one instruction by making this code (i.e., instruction sequence) more compact:

```
        MOVEI   12,5      ;Initial count value is 5
LOOP:                     ;Jump back to here to perform
                          ;  the function that is being repeated
        . . .
        CAIGE   12,7      ;End test.  Skip if the AC is greater than 6
         AOJA   12,LOOP   ;Was LT 7.  Increment & repeat LOOP function
```

By rewriting this loop's end test, we have saved an instruction. Moreover, the instruction that we saved was one that normally would have been executed every time through the loop.

The first example showed that incrementing the loop index, testing for loop termination, and jumping back to the top of the loop could be thought of as three separate functions. This improved way of doing things bundles the increment and jump into one instruction. With respect to the operation of this loop, there is one further difference. In the first example, the loop was executed for accumulator `12` containing the values `5`, `6`, and `7`. The same is true in the example that uses `AOJA`. However, at loop exit, in the first case, the accumulator has been incremented to `10` (octal); in the second case, our loop that uses `AOJA` avoids incrementing the accumulator at the end of the loop, so register `12` is left at `7` when the loop exits. Depending on the instructions that follow this loop, that difference may or may not be significant.

There are other ways to implement loops. By placing the test at the end of the loop (called a *bottom test*) we force the program to perform the repeated function at least once. If this is objectionable, the test can be moved to the beginning (or *top*) of the loop. The top test loop is characteristic of Pascal and the Algol–style languages. Here is an example of one implementation of the top test loop:

```
         MOVE    12,LO          ;initial lower bound
 LOOP:   CAMLE   12,HI          ;compare to upper bound
          JRST   LOOPX          ;exit from the loop
         . . .                  ;the instructions to repeat
         AOJA    12,LOOP        ;increment count, jump to the loop top

 LOOPX:                         ;here when done.
```

Although three instructions are used inside the loop to effect control, it should be noted that only two of them are executed as part of the loop. The instruction `JRST LOOPX` is executed only once to

[5] A new standard for the Fortran language, Fortran–77, specifies that DO loops may avoid executing entirely. At the time this book is being written, the old Fortran–IV standard continues in widespread use.

escape from the loop.

6.3.2 Applying AOBJN

Sometimes the best way to accomplish a forward loop is to use the AOBJN instruction. AOBJN is especially useful in those circumstances where indexing is required also. For example, to increment the 12 (octal) words starting at TABLE, you could write the following loop:[6]

```
        MOVSI   1,-12           ;Initialize register 1 to -12,,0
LOOP:   AOS     TABLE(1)        ;increment one array element.
        AOBJN   1,LOOP          ;increment both the index and the
                                ;control.  Loop until the AOS has
                                ;been done 12 (octal) times.
```

In this loop, the left half of register 1 counts up from -12 to 0. The loop is executed while the left half is negative (i.e., for the left half values -12 through -1, a total of 12 times). Meanwhile, the right half of the accumulator is counting up, from 0 to 12; the values 0 through 11 appear in the right half of register 1 during the execution of this loop.

In the traditional machine, effective address calculation considers only the right half of the index register. In the extended machine, when the index register contains a negative left half, only the right half of the index register contributes to the effective address. Thus, this loop accomplishes references to TABLE+0 through TABLE+11. The loop that uses AOBJN contains fewer instructions and is usually superior to the loop that uses CAIGE and AOJA:

```
        MOVEI   1,0
LOOP:   AOS     TABLE(1)
        CAIGE   1,11
         AOJA   1,LOOP
```

One additional technique should be mentioned. Suppose that the function "increment every element of an array" is needed at several places in the program and that it is needed for several different arrays. Then we could write a subroutine that performs this function, in which the size of array and the name of the array are carried in register 1 as an argument. (The details of calling subroutines and returning from them will be discussed in Section 13.2, page 163.)

```
CX:     -12,,TABLE      ;the negative size, and address of the array.

        ...

        MOVE    1,CX    ;initialize register 1
        CALL    LOOP    ;Call LOOP as a subroutine
        ...

LOOP:   AOS     0(1)    ;increment one array element.
        AOBJN   1,LOOP  ;increment the index and the control count.
        RET             ;return from this subroutine.
```

[6] In this example and several that follow, we presume that TABLE is in the same address section as the code.

The key difference between this example and the previous example of AOBJN is that the reference to TABLE has been removed from the interior of the loop. This is important because some other piece of the program could initialize register 1 with some other count and array address. Then, by calling this subroutine this same function could be applied to a different array.[7]

6.3.3 Backwards Loops

It is easy to run the loop index backwards by means of the SOJ class instructions. Sometimes it is possible to make either zero or one the last value of index variable. In such cases the SOJG or SOJGE instructions are quite useful. Some examples will appear in the discussion of nested loops.

6.3.4 Example 2–A — Nested Loops

Nested loops are quite simple to do. For example, let us write a program to produce the triangular pattern depicted below. (We presume that we're restricted to a rather simple device in which we must produce the top line before the one beneath it, etc.)

```
*****
****
***
**
*
```

There are several ways to approach this problem. Perhaps the simplest is to number the lines, from top to bottom, 5 through 1. Then, for each line number, we must output exactly that number of asterisks. We will attempt to write the assembly language program corresponding to the Pascal program that writes this triangle:

```
PROGRAM TRIANGLE;
VAR i, j : INTEGER;
BEGIN
FOR i := 5 DOWNTO 1 DO
  BEGIN
  FOR j := i DOWNTO 1 DO WRITE('*');
  WRITELN
  END
END.
```

First, we need an outer loop that counts the lines from 5 down to 1. This is an easy loop to implement. We will use the SOJG instruction:

```
        MOVEI   10,5      ;let register 10 contain the line number.
                          ;set it to 5.
LINE:   . . .             ;print one line
        SOJG    10,LINE   ;decrement the line number held in 10.
                          ; If the result is positive, do another line
```

Next, we have to install the inner loop. This loop is responsible for printing one line. The environment to which the inner loop must be accommodated is that register 10 contains the line number (which is also the number of asterisks to print).

[7] The target arrays must all be in the same address section as the code.

Again, we use SOJG as the appropriate instruction. Since register 10 is busy counting the line number, we must write the inner loop to avoid modifying register 10. Not only is register 10 important to the outer loop, it is important to the inner loop: it specifies the number of asterisks to type. So, our first necessary action is to copy the data in register 10 to some other place; the inner loop will use and modify the copy, leaving register 10 unchanged.

```
LINE:   MOVE    11,10           ;copy the line number to register 11
STARS:  . . .                   ;Print one asterisk
        SOJG    11,STARS        ;decrement star count, loop if more
        . . .                   ;print carriage return and line feed
                                ;  to prepare for the next line.
```

These two fragments can be put together:

```
        MOVEI   10,5        ;let register 10 be the line number.
                            ;set it to 5.
;print one line
LINE:   MOVE    11,10       ;copy the line number to register 11
STARS:  . . .               ;Print one asterisk
        SOJG    11,STARS    ;decrement star count, loop if more
        . . .               ;print carriage return and line feed
                            ;  to prepare for the next line.
        SOJG    10,LINE     ;decrement the line number held in 10.
                            ; If the result is positive, do another line
```

The remainder of the program can be added. This program can be typed in and run.

```
        TITLE   TRIANGLE  Example 2-A
        SEARCH  MONSYM
        .PSOUT  CODE,1001000

Comment $ Program to print a Triangle $

START:  RESET
        MOVEI   10,5            ;let register 10 be the line number.
                                ;set it to 5.
;print each line
LINE:   MOVE    11,10           ;copy the line number to register 11
;print the stars on each line
STARS:  HRROI   1,ASTER         ;Print one asterisk
        PSOUT
        SOJG    11,STARS        ;decrement star count, loop if more
        HRROI   1,NEWLIN        ;print carriage return and line feed
        PSOUT                   ;  to prepare for the next line.
        SOJG    10,LINE         ;decrement the line number held in 10.
                                ; If result positive, do another line
        HALTF                   ;stop here

ASTER:  ASCIZ /*/
NEWLIN: ASCIZ /
/
        END     START
```

Perhaps the preceding example is sufficient to demonstrate the reasoning process necessary for

constructing loops in assembly language. The problem of writing the triangle is decomposed into a repetition of the problem of writing one line. The problem of writing one line is decomposed into the problem of writing the correct number of stars and then writing the end of line characters.

The decomposition of this problem in assembly language follows the same outlines as problem solving in Pascal or Fortran. The major difference in assembly language is that the level of detail is much greater. Careful attention must be paid to the interface between the instruction segments that solve each subproblem.

6.3.5 Example 2–B — Nested Loops

At the risk of over–doing examples, let us try one more problem. Again the pattern is a triangle, but it's quite a change from the previous one.

```
*******
 ******
  *****
   ****
    ***
     **
      *
```

This pattern contains seven lines. Each line has seven characters in it. If these lines were numbered from the top to the bottom, `0` to `6`, then the line number would also be the same as the number of spaces to write at the front of the line. The number of stars is whatever is necessary to fill out the seven characters on each line.

The inner loop will write seven characters (columns `0` to `6`) on each line. We will install a test in the inner loop so that when the column number is less than the line number, blanks are written. Stars are written when the column number is larger than (or equal to) the line number. The Pascal program that performs this function should make clear what we are doing:

```
PROGRAM TRI;
VAR i, j : INTEGER;
BEGIN
FOR i := 0 TO 6 DO BEGIN
  FOR j := 0 TO 6 DO IF j < i THEN WRITE(' ') ELSE WRITE('*');
  WRITELN
  END
END.
```

The outer loop will run a variable up from 0 to 6:

```
        MOVEI   12,0          ;AC 12 contains the line number
LINE:   . . .                 ;Print one line
        CAIGE   12,6          ;have we done enough lines?
         AOJA   12,LINE       ;no, increment the line number & loop
```

The inner loop is somewhat more complex. We must print seven characters on each line. These can be numbered left to right `0` to `6`. A character loop is needed to step through each character (or column) number.

```
 LINE:   MOVEI   13,0          ;character counter
 CHAR:   . . .                 ;print one character
         CAIGE   13,6
          AOJA   13,CHAR
         . . .                 ;print carriage return and line feed.
```

To make the decision about which character, space or asterisk, to print in each position, we observe that a space should be printed if the character count (register 13) has a smaller value than the line count (register 12). This is easily coded (i.e., written as an instruction sequence):

```
         CAML    13,12                  ;skip if character count is
                                        ;less than the line count
          JRST   PSTAR                  ;go print a star
         . . .                          ;print a blank
         JRST    ELIN                   ;test for end of line
 PSTAR:  . . .                          ;Print a star
 ELIN:                                  ;perform end-of-line test
```

These fragments can be combined (and augmented) as follows:

```
        TITLE   TRI AGAIN - Example 2-B
        SEARCH  MONSYM
        .PSECT  CODE,1001000

Comment $  A different triangle $

START:  RESET                   ;begin execution here
        MOVEI   12,0            ;initial line count
;here to print each line
LINE:   MOVEI   13,0            ;initial character count
;print one character on a line
CHAR:   CAML    13,12           ;skip if printing spaces
         JRST   PSTAR           ;go print a star
        HRROI   1,BLANK         ;print a blank
        PSOUT
        JRST    ELIN            ;test for end of line


PSTAR:  HRROI   1,ASTER         ;print a star
        PSOUT
ELIN:   CAIGE   13,6            ;have we reached the end of line?
         AOJA   13,CHAR         ;print the next character
        HRROI   1,NEWLIN        ;print the carriage return & line feed
        PSOUT
        CAIGE   12,6            ;finished all lines yet?
         AOJA   12,LINE         ;no.  do more.
        HALTF                   ;all done


BLANK:  ASCIZ / /
ASTER:  ASCIZ /*/
NEWLIN: ASCIZ /
/
        END     START
```

An alternative approach to controlling the execution of this program is presented below. In this approach, we assume that we will have to print a blank character. At the label `CHAR` register `1` is loaded with the string pointer to the blank character. Next, the program determines whether this assumption was correct. If the current character position is less than the current line number, the `CAML` instruction will skip. However, if the program has already put enough blanks on this line, the `CAML` does not skip and register `1` is loaded with a string pointer to the asterisk.

```
CHAR:   HRROI   1,BLANK          ;Assume we must send a blank
        CAML    13,12            ;Skip if near front of line
         HRROI  1,ASTER          ;Past the blanks, print a star
        PSOUT                    ;Send either a blank or a star
        CAIGE   13,6             ;Have we done enough here yet?
         AOJA   13,CHAR          ;Not yet.  Bump count and loop
        ...                      ;write end of line, do next line
```

6.4 Exercise

Write a program that uses the conditional jumps, skips, compares, etc, to type the following pattern on your terminal:

```
      *
     ***
    *****
   *******
  *********
 ***********
*************
 ***********
  *********
   *******
    *****
     ***
      *
```

Some hints:

- You haven't been told how to do arithmetic. If you had a desire to write, for example, a FOR statement such as `FOR j:=0 TO 19-i DO`, you could not form the difference `19-i`. Instead, you can transform it to be `FOR j:=i TO 19 DO`.
- You haven't been told how to multiply by two. One way to do it is to execute some piece of program twice. Another approach might be to output two asterisks at a time instead of one.
- Assume there are no blanks to the left of the longest line of asterisks.

Chapter 7

Terminal Input

We turn our attention next to the problem of obtaining character input from the terminal. The program that we shall construct demonstrates how to obtain input lines from the terminal. The program will prompt for a line, read the line, and then type the line back to the user's terminal. The program will loop until the user types a line in which the first five characters are "`LEAVE`", whereupon the program will stop running.

7.1 Example 3 — Terminal Input (RDTTY JSYS)

In order to obtain a line of characters from the terminal we will use the `RDTTY` JSYS. `RDTTY` accepts one line from the terminal and places it in the memory space belonging to the program. In order to do input, we must inform TOPS–20 of at least two things. First, we must tell TOPS–20 the location in our memory space where we want it to place the input line. Second, we must tell TOPS–20 how much space is available at that location. We supply these items of information as arguments to the JSYS.

7.1.1 BLOCK to Reserve Space

We must reserve space in memory for an *input buffer*. A buffer is just a place where we store input or output data. The MACRO assembler provides the `BLOCK` pseudo–operator to reserve space in memory. The `BLOCK` pseudo–op takes one argument, a number or symbolic expression that tells how many words to reserve. Each word that we reserve can hold five characters. If we decide to allow input lines of length `140` (decimal) then we'll need to reserve `28` words. Let's splurge and reserve decimal `32` (octal `40`) words for our input line buffer area. Because we'll need to tell `RDTTY` the location of this area, we must put a label on it. We have the following program fragment thus far:

```
IBUFR:  BLOCK   40              ;reserve space for the input line

        . . .

        RDTTY                   ;read one line from the terminal
```

The symbol `IBUFR` labels the first location reserved by the `BLOCK` pseudo–op. Symbolically, the locations in this buffer may be referred to as `IBUFR+0`, `IBUFR+1`, `IBUFR+2`, etc., through `IBUFR+37`.

```
IBUFR+0
IBUFR+1
IBUFR+2

 . . .

IBUFR+36
IBUFR+37
```

It is important to note that when we write something like `IBUFR+25` we mean the *value of the symbol* `IBUFR` plus (octal) `25`; in the case of a label such as `IBUFR`, the value of the symbol is the location or address of `IBUFR` at runtime. Contrast this to most high–level languages in which such an expression would mean the *contents of the location called* `IBUFR` plus decimal `25`. Apart from the difference between octal and decimal numbers, the interpretation of symbolic names is entirely different. In the assembler, symbols usually refer to addresses of things. In most high–level languages, symbols in expressions usually refer to the contents of addresses.

Another way of looking at the difference is that the assembler can not compute anything that has to do with the contents of memory locations at runtime: that is what your program is for. If you want to compute the *contents* of location `IBUFR` plus octal `25`, you must write something like:

```
MOVE    5,IBUFR
ADDI    5,25            ;(contents of IBUFR) + 25 (octal)
```

Again, the assembler does arithmetic based on the values of symbols. These values are often addresses. This arithmetic has nothing at all to do with what your program can compute at runtime.

7.1.2 Arguments to RDTTY

We must tell `RDTTY` that the symbolic name `IBUFR` is the beginning of our buffer space. The `RDTTY` JSYS accepts in register `1` a *destination designator* that points to the input area. Now, a destination designator can have any of several formats; in this case it is convenient to use a `HRROI` instruction to build it in register `1`. (This is the same pointer format as we saw `PSOUT` accept.) Our fragment will be augmented by the inclusion of a `HRROI` instruction preceding the `RDTTY`.

Next, we must set up the buffer length argument that describes how much space is available in `IBUFR`. If we didn't tell TOPS–20 how much space we have, we would have to accept the possibility that a long line could overflow the allocated buffer space and ruin some other memory locations. (A lot of programs suffer from this problem. The internet worm in November 1988 was a hacker's exploitation of an unguarded buffer's overflow.)

We tell TOPS–20 the character count — how many characters will fit into the buffer — in the right half of register `2`. Now, we know that there are `40` (octal) words available in the buffer. This area could hold up to `40*5` characters. For safety and ease of programming, we'll reduce this count by one. The `MOVEI` instruction is useful for loading small constants such as this one into a register:

```
IBUFR:  BLOCK   40       ;reserve space for the input line

        . . .

        HRROI   1,IBUFR  ;destination designator for the input buffer
        MOVEI   2,5*40-1 ;count of available space in the input buffer
        . . .
        RDTTY            ;read one line from the terminal
```

Note that it is completely permissible to use expressions such as `5*40-1` in the assembler. Again, it is important to remember that the arithmetic done by the assembler involves constants and the addresses of data items. The assembler cannot compute values based on the runtime contents of a memory location.

As an aside, we should mention that the left half of register `2` contains flags for `RDTTY`. The `MOVEI` instruction sets those flags to zero; this signifies that the carriage return and line feed character sequence will be accepted to mark the end of the input line. We should also say that TOPS–20 is normally set up so that when the user types a carriage return character, the program will see both the carriage return and a line feed. The characters carriage return and line feed are talked about so often that we refer to them as `CR` and `LF` respectively. The usual sequence of both characters is called `CRLF`.

7.1.3 Defining Symbolic Names

One of the certainties of programming is that programs change. Note that our program fragment has used the length of the input buffer in two places. If we ever wanted to change the buffer size in this program we would have to go back, reread the program carefully and find all the places where the number `40` appears as the buffer length and change them. It might be a very difficult task. The expression that we wrote as `5*40-1` might have been written as `237` (in octal, of course), which would conceal its relation to the buffer length even further.

An aid to maintaining programs is to define a symbol whose value is the desired length of the input buffer. Then, instead of writing `40` everywhere, we write this symbol instead. We can define a symbol by writing the symbol name, an equal sign, and the value that we want the symbol to have, e.g., `BUFLEN=40`. This form of definition is called an *assignment*. (This is in contrast to labels that are defined with a colon.) In this program fragment, we will ask MACRO to make such a definition. Further, by writing two equal signs we instruct MACRO to *suppress* this symbol so the debugger will not type it out; as a general rule, if a symbol is not the name of a location, we suppress it.[1] To our program fragment we add the definition of the symbolic name `BUFLEN`, signifying the input buffer length:

[1] See the discussion of DDT, Section 9, page 101.

```
BUFLEN==40                  ;symbol for length of the input buffer.

IBUFR:  BLOCK   BUFLEN      ;reserve space for the input line

        . . .

        HRROI   1,IBUFR     ;destination designator for input buffer
        MOVEI   2,5*BUFLEN-1 ;count of available space in the buffer
        . . .
        RDTTY               ;read one line from the terminal
```

There is one important restriction on the use of symbolic names such as `BUFLEN`. The assembler insists that the argument to the `BLOCK` pseudo–op be completely defined at the time `BLOCK` is seen. This dictates that we must place the definition of `BUFLEN` before the `BLOCK` pseudo–op. (In contrast, MACRO could cope with `BUFLEN` being defined after it appears in the `MOVEI` instruction. The difference is that `BLOCK` reserves space: the assembler must know when it sees `BLOCK` how many locations to reserve.)

7.1.4 Prompting for Input

Before we get around to discussing the final argument for `RDTTY`, we must talk about prompting for input. Whenever a well–written program wants input from the terminal it will prompt with an informative message. Even better, a program should be ready to supply help to the user to make the user better able to decide what function is wanted.

Extensive help facilities are beyond the scope of this program. However, prompting is within our capabilities. We augment the program fragment by adding a prompting message before the `RDTTY` JSYS. The prompt is done by the `PSOUT` JSYS; `PSOUT` requires that register `1` contain a source designator. Therefore, the `PSOUT` must appear before we set up register `1` for `RDTTY`.

7.1.5 The Reprompt Pointer

The `RDTTY` JSYS implements a variety of input line editing functions. From your experience as a user of TOPS–20 you should be aware that the characters `Rubout` (or `Backspace` or `Delete`), `CTRL/W`, and `CTRL/U` can be used to edit an input line, at least up until the point where a carriage return is typed. On a hardcopy terminal, extensive use of these editing characters can result in an unreadable input line. To lessen the confusion caused by editing the line, `RDTTY` allows the user to request that the input line be retyped. The user indicates that he wants the line retyped by typing the character `CTRL/R`. When `RDTTY` sees a `CTRL/R` character, it advances to the next line and retypes a clean copy of the input line.

It is desirable for the retyped line to include the same prompting string as was visible on the original line. Thus, the prompting string must be made known to `RDTTY`. The prompting string is described to `RDTTY` by an argument called the *reprompt pointer*.

The prompting message "Welcome to Echo. Please type a line: " was sent to the user's terminal via `PSOUT`. For this program, we shall set the reprompt pointer to point to this same prompting message. The reprompt pointer is passed to `RDTTY` in register `3`.

The reprompt pointer need not point to the exact last prompt. Rather, it should normally point to the last line of the most recent prompt. For example, if we changed our prompt in this program to

be

```
PROMPT: ASCIZ   /Welcome to Echo.
Please type a line: /
```

Then the last line of the most recent prompt would be "Please type a line: ". Thus, the reprompt should omit the phrase "Welcome to Echo."

To set up the reprompt pointer, again we use the HRROI instruction. The reprompt pointer goes into register 3.

```
BUFLEN==40                  ;the length of the input buffer.

IBUFR:  BLOCK   BUFLEN      ;reserve space for the input line

        . . .
        HRROI   1,PROMPT    ;source designator for prompt message
        PSOUT               ;send prompt message to the terminal
        HRROI   1,IBUFR     ;destination designator for input buffer
        MOVEI   2,5*BUFLEN-1 ;count of available space in the buffer
        HRROI   3,PROMPT    ;the reprompt pointer for RDTTY
        RDTTY               ;read one line from the terminal
        . . .
PROMPT: ASCIZ   /Welcome to Echo.  Please type a line: /
```

7.2 Print the Heading and Echo the Input Line

Next we will discuss how to print the results. We should print a heading before typing the copy of the input line. We have already seen one way to create the text of a heading message and pass it to the PSOUT JSYS. We now examine a more convenient way to do this.

7.2.1 Literals

Now, before we type out our copy of the input line, we will send a message to the terminal that labels the forthcoming output. We could write something like the following:

```
        HRROI   1,HEADER
        PSOUT

        . . .

HEADER: ASCIZ   /The line you typed was: /
```

Instead of writing the message in that manner, we will make use of another assembler feature called a *literal*. A literal saves us from having to write a label and reference the label. Using a literal we would rewrite the sequence that was shown above as:

```
        HRROI   1,[ASCIZ /The line you typed was: /]
        PSOUT
```

The square brackets denote a literal. A literal is a word or group of words that is specified by telling the assembler what the words contain. When you write a literal, MACRO replaces it with the address where it will store the contents that you specified.

In this case, by putting the `ASCIZ` inside a literal you are telling the assembler

- Make a group of words that contain the binary representation of the material found within the square brackets.
- Put those words somewhere.
- In the place where the literal text (i.e., the material in square brackets) appears, store the address where the first word of the literal was put.

A literal is just a way to avoid having to think up a name for a label, and having to write the label twice. More than one line and more than one word can appear in a literal.

It is vital that you always consider the contents of a literal to be a constant. *Never allow the execution of your program to change the contents of a literal.* There are two reasons for this. First, the user should be able to restart the program. If literals have been changed, the program will not be *restartable*. Second, if two literals have the same value, the assembler builds only one copy. The one value is shared among all the places in your program that refer to that value. If some part of a program changes that value, that change affects all other places that refer to that value.

You may place instructions in literals, but the use of literals for vast numbers of instructions is a poor practice; it leads to difficulties in debugging. Literals can be nested. That is, a literal can appear inside another literal.

7.2.2 Echo the Line

Now we must type a copy of the input line. The `RDTTY` JSYS thoughtfully stores a null character at the end of the input string; we recognize this as the same format of text string as is produced by the `ASCIZ` pseudo–op. Therefore, we can use `PSOUT` when we want to print this line. The line appears in the area that we know by the symbolic name `IBUFR`; we load register `1` with a pointer to `IBUFR` and execute a `PSOUT` JSYS.

The program fragment now includes the use of a literal to label the output line, and the instructions to print the output:

```
BUFLEN==40                  ;the length of the input buffer.

IBUFR:  BLOCK   BUFLEN      ;reserve space for the input line

        . . .

        HRROI   1,PROMPT    ;source designator for prompt message
        PSOUT               ;send prompt message to the terminal
        HRROI   1,IBUFR     ;destination designator for input buffer
        MOVEI   2,5*BUFLEN-1 ;count of available space in input buffer
        HRROI   3,PROMPT    ;the reprompt pointer for RDTTY
        RDTTY               ;read one line from the terminal

        . . .

        HRROI   1,[ASCIZ /The line you typed was: /]
        PSOUT
        HRROI   1,IBUFR     ;a pointer to the input line, for PSOUT
        PSOUT               ;type back, i.e., echo, the input line.

        . . .

PROMPT: ASCIZ   /Welcome to Echo.  Please type a line: /
```

7.3 Testing for Special Inputs

We described this program as looping, repeating its function, until a line containing the initial five characters "LEAVE" is seen. We must now implement this loop and end test. The loop is easy. After echoing the input line, a JRST instruction to make the program jump back to the prompt is all that is needed. The end test is slightly more subtle. Since precisely five characters will fit into one computer word, if the word at IBUFR contains the letters "LEAVE" we shall perform a HALTF JSYS to stop the program. We can test IBUFR for the letters "LEAVE" by means of a CAME or CAMN instruction; the choice of a particular instruction is made according to whatever seems most convenient in the program.

```
        MOVE    15,IBUFR          ;Copy the first word at IBUFR to
                                  ;register 15.  This word contains
                                  ;the first 5 characters of input
        CAMN    15,[ASCII/LEAVE/] ;Compare to the 5-character
                                  ;ASCII string "LEAVE".
                                  ;Skip if not equal.
         JRST   STOP              ;The first 5-letters were "LEAVE"
                                  ;Stop running.
```

The ASCII pseudo–op (recall Section 4.8.1, page 39) is used in this case because we know the five letters "LEAVE" occupy precisely one computer word; the nulls added by the ASCIZ pseudo–op are not needed in this comparison.

Note that this test checks for precisely the five upper–case characters "LEAVE" as the first five characters on a line. A better but more complex program would ignore case distinctions and leading

spaces; that better program is just a little beyond our grasp at this moment. It might also be noted that the word "leave" was carefully selected because it has precisely five characters; again, it is somewhat beyond our present capabilities to write a more general program.

The program (our comments have been shortened somewhat) now looks like this:

```
BUFLEN==40                      ;the length of the input buffer.

IBUFR:  BLOCK   BUFLEN          ;reserve space for the input line
        . . .

GETLIN: HRROI   1,PROMPT        ;source of the prompt message
        PSOUT                   ;send prompt to the terminal
        HRROI   1,IBUFR         ;descriptor of the input buffer
        MOVEI   2,5*BUFLEN-1    ;character count of the buffer
        HRROI   3,PROMPT        ;the reprompt pointer for RDTTY
        RDTTY                   ;read one line from the terminal
        . . .
        MOVE    15,IBUFR        ;first five letters of input line.
        CAMN    15,[ASCII/LEAVE/] ;test to see if user typed LEAVE
         JRST   STOP            ;LEAVE was typed. Go halt the program.
        HRROI   1,[ASCIZ /The line you typed was: /]
        PSOUT
        HRROI   1,IBUFR         ;a pointer to the input line, for PSOUT
        PSOUT                   ;type back, i.e., echo, the input line.
        JRST    GETLIN          ;Go get another line.

STOP:   HALTF                   ;stop the program here
        JRST    STOP            ;In case of a CONTINUE command,
                                ;  stay stopped.

PROMPT: ASCIZ   /Welcome to Echo.  Please type a line: /
```

Note that under `HALTF` we have added a jump back to `STOP`. The `HALTF` JSYS stops your program. It doesn't destroy your program. The EXEC's `Continue` command will resume the execution of a stopped program. So, we had better put something underneath every `HALTF` that we write. Looping to the `HALTF` is one choice, looping to `GETLIN` or to `START` are other plausible choices.

7.4 Error Conditions

The possibility that errors might occur must be considered whenever we are attempting to perform input/output functions or otherwise communicate with the operating system. Errors stem from two main sources. First, the program itself can be wrong. In such a case, as TOPS–20 checks the arguments supplied to a JSYS call, it might discover an error. The second source of errors is essentially beyond the control of the programmer. Actual mistakes may occur inside the computer or within external input/output devices. The operating system will indicate that such a failure has occurred. Sometimes it is possible to perform an effective *retry* operation, in which either the program or the operating system repeats the failed operation in the hopes that a second attempt will succeed.

Extensive error handling is beyond the scope of this example. However, to get started along the path of making better programs we must mention two things: first, some JSYS instructions skip

one or more times to indicate their success. A failure of some kind is often indicated by a failure to skip. Second, in addition to the ability of some JSYS instructions to skip to signal their success, the TOPS–20 operating system implements another form of error signal, the `ERJMP` instruction.

`ERJMP` is actually a `JUMP` instruction in which the accumulator field is set to `16`. As we discussed previously, the `JUMP` instruction, unless modified by some condition, e.g., `JUMPGE`, has no effect. So, if it doesn't do anything, what good can it do?

Whenever a TOPS–20 program encounters an error condition, whether it was caused by a failing JSYS call or other illegal instruction, TOPS–20 looks at the instruction following the failing instruction or JSYS. If the instruction after an error is an `ERJMP` then TOPS–20 transfers control to the address specified by the effective address of the `ERJMP` instruction.

`ERJMP` is customarily employed after a JSYS call that might fail for some reason. If all goes well within the JSYS, either the `ERJMP` is skipped or it is executed by the hardware with no effect. However, if something goes wrong, the `ERJMP` tells TOPS–20 that you are prepared to deal with the error; TOPS–20 transfers control to the address specified in your `ERJMP`.[2] If no `ERJMP` is present, TOPS–20 takes some other action such as stopping your program to allow the EXEC to issue an appropriate error message.

Since the meaningful execution of `ERJMP` is conditional on the instruction that precedes it, we remind the reader of this fact by indenting `ERJMP` as if it followed a skip instruction.

The `RDTTY` JSYS can fail, usually from faulty arguments. In this program we perform minimal error handling by informing the user that something is wrong and stopping.

7.5 Sectioning the Program

Although this program is really too simple to require this step, good habits are learned early. We will divide our program into program sections for the purpose of isolating the program and constants from the writeable data. This division is accomplished via the `.PSECT` pseudo–op that we previously used to put the program into a particular place in memory. Now, we use `.PSECT` twice, each time to declare a program section and to locate it in memory.

A program section created by the `.PSECT` pseudo–op is called a *psect*. The psects are not related to the address space sections defined by the extended addressing scheme.

In this example, we place the writeable psect called "DATA" at location 1001000, i.e., page 1 of section 1. Because we know this psect contains only `IBUFR`, we know that one page (1000 locations) is adequate for it. Consequently, when it comes to declaring the read–only psect, "CODE", we can put it at 1002000, i.e., page 2 in section 1. Although a psect can start anywhere, write–protection (or permission) comes in page–size increments: thus we start CODE on a page boundary. We tell MACRO and LINK to write–protect CODE by means of the switch `/RONLY` as a parameter to `.PSECT`.

If, subsequent to some modifications to this program, the DATA psect expanded past one page, the program would still assemble, but LINK would detect a "Psect Overflow" which you would fix by locating CODE at a higher address.

[2] TOPS–20 transfers control to the effective address obtained by interpreting the `I`, `X`, and `Y` fields of `ERJMP` as the hardware would.

7.6 The Finished Program

We display the final program below. We have added the error handling instructions and the `.PSECT` pseudo–ops that were described above. The standard requirements of `TITLE`, `SEARCH`, `END`, and a starting address are included.

```
        TITLE   ECHO INPUT LINE -- Example 3
        SEARCH  MONSYM

BUFLEN==40                      ;the length of the input buffer.

        .PSECT  DATA,1001000    ;location for the writeable portion
IBUFR:  BLOCK   BUFLEN          ;reserve space for the input line

        .PSECT  CODE/RONLY,1002000 ;location for code and constants

START:  RESET                   ;start here.  initialize I/O
GETLIN: HRROI   1,PROMPT        ;source of the prompt message
        PSOUT                   ;send prompt to the terminal
        HRROI   1,IBUFR         ;descriptor of the input buffer
        MOVEI   2,5*BUFLEN-1    ;character count of the buffer
        HRROI   3,PROMPT        ;the reprompt pointer for RDTTY
        RDTTY                   ;read one line from the terminal
         ERJMP  INERR           ;in case of error, print a message
        MOVE    15,IBUFR        ;first five letters of input line
        CAMN    15,[ASCII/LEAVE/]  ;test to see if user typed LEAVE
         JRST   STOP            ;LEAVE was typed. Halt the program.
        HRROI   1,[ASCIZ /The line you typed was: /]
        PSOUT
        HRROI   1,IBUFR         ;point to the input line for PSOUT
        PSOUT                   ;echo (retype) the input line.
        JRST    GETLIN          ;Go get another line.

INERR:  HRROI   1,[ASCIZ/Error from RDTTY.  I give up
/]
        PSOUT                   ;send an error message and stop.
STOP:   HALTF                   ;stop the program here
        JRST    STOP            ;In case of a CONTINUE command,
                                ;  stay stopped.

PROMPT: ASCIZ   /Welcome to Echo.  Please type a line: /

        END     START           ;specify the start address.
```

Chapter 8

The Assembler and Loader

As we have said before, the programs that the computer actually executes are composed of a series of binary words; these words are the instructions and the data for the computer.

At the origins of computer development, programs were created by placing the appropriate binary patterns in memory by hand. Often, results were obtained by examining the binary patterns appearing in particular memory locations.

If we wanted to we could program the DECSYSTEM–20 by dealing only with binary patterns in memory. Rather than do that, it is vastly more convenient to deal with an assembler and loader program that do many of the bookkeeping chores that are necessary in programming.

Given our background of having examined several sample programs already, it is now time to delve somewhat deeper into the functions of the assembler, and to introduce the loader program.

8.1 Overview of Assembly and Loading

Figure 8.1 represents the relationships between the assembler and loader, and between the assembler and cross–reference program.

The inputs to the assembler include the files that contain our source code plus any universal files that we might request, e.g., `SYS:MONSYM.UNV`.

The output of the assembler will be the translation of the source code into a binary relocatable file, and an optional listing of the source code with the assembled data shown with the source code.

The relocatable file is read by the loader program which builds an image of the the program in memory. When the loader has finished, the program resides in memory and is ready to be started. This result is called a *memory–image*.[1] The loader can also produce an *executable file*, a disk file that contains a memory–image, which can be run directly by the EXEC program.

8.2 Assembler Listings

(Author's note: it must be remarked that the need for listings seems vastly diminished. This facility exists, but its utility is limited. This discussion remains largely to illuminate assembler and loader

[1] On occasion, some geezer will say "core–image" evoking the good old days.

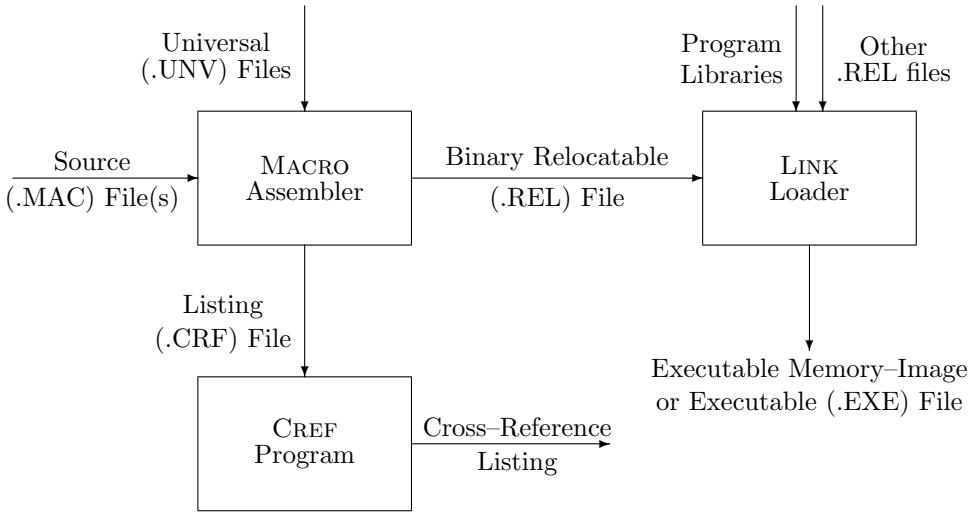


Figure 8.1: Overview of the Assembler and Loader

functions from a different angle.)

The assembler together with the CREF program will produce a listing of a program that contains much useful information. As an example of the CREF output we have prepared a listing of our third example program, ECHO. The listing that follows was obtained by entering the text of the program into the file `EX3.MAC` and then issuing the commands:

```
@COMPILE/COMPILE/CREF EX3.MAC
@CREF
```

The `COMPILE` command runs the MACRO assembler. The command switch `/COMPILE` forces a new assembly even though an up–to–date `REL` file may exist. The `/CREF` switch tells MACRO to create an assembly listing that is augmented by the inclusion of information for the CREF program. Note that this extra information is not meant for people to read; it is included to help CREF make its cross–reference tables. An examination of the file that MACRO makes for CREF will reveal numerous binary characters that can not be printed intelligibly on a line printer.

The `CREF` command runs the CREF program. CREF reads the listing that MACRO produced, reformats the listing, writes the cross–reference tables, and sends the resulting file to the printer.

The resulting listing has three components. The first, shown in Figure 8.2, is the listing of the original source program, augmented by octal numbers representing the location and contents of the assembled instructions and data. Any error messages from MACRO will be present in this portion. This listing also includes line numbers at the left margin; these were added by the CREF program.

The second component of the listing is MACRO's symbol table, shown in Figure 8.3. Each symbolic name defined or used by the programmer appears here, together with its corresponding octal value.

The third component of the listing file is the cross–reference section, shown in Figure 8.4. This is the main contribution of the CREF program. The cross–reference of symbols shows the line number of every occurrence of each symbol name.

By itself, MACRO can produce a listing that includes

```
ECHO INPUT LINE -- Example 3    MACRO EX3     MAC      9-Nov-2004 12:36

     1                                        TITLE   ECHO INPUT LINE -- Example 3
     2                                        SEARCH  MONSYM
     3
     4                     000040       BUFLEN==40                        ;the length of the input buffer.
     5
     6                                        .PSECT  DATA,1001000       ;location for the writeable portion
     7  000000'01                      IBUFR:  BLOCK   BUFLEN             ;reserve space for the input line
     8
     9                                        .PSECT  CODE/RONLY,1002000 ;location for code and constants
    10
    11  000000'02  104 00 0 00 000147   START:  RESET                     ;start here.  initialize I/O
    12  000001'02  561 01 0 00 000024'  GETLIN: HRROI   1,PROMPT          ;source of the prompt message
    13  000002'02  104 00 0 00 000076           PSOUT                     ;send prompt to the terminal
    14  000003'02  561 01 0 00 000000#          HRROI   1,IBUFR           ;descriptor of the input buffer
    15  000004'02  201 02 0 00 000237           MOVEI   2,5*BUFLEN-1      ;character count of the buffer
    16  000005'02  561 03 0 00 000024'          HRROI   3,PROMPT          ;the reprompt pointer for RDTTY
    17  000006'02  104 00 0 00 000523           RDTTY                     ;read one line from the terminal
    18  000007'02  320 16 0 00 000020'           ERJMP  INERR             ;in case of error, print a message
    19  000010'02  200 15 0 00 000000#          MOVE    15,IBUFR          ;first five letters of input line
    20  000011'02  316 15 0 00 000034'          CAMN    15,[ASCII/LEAVE/] ;test to see if user typed LEAVE
    21  000012'02  254 00 0 00 000022'           JRST   STOP              ;LEAVE was typed. Halt the program.
    22  000013'02  561 01 0 00 000035'          HRROI   1,[ASCIZ /The line you typed was: /]
    23  000014'02  104 00 0 00 000076           PSOUT
    24  000015'02  561 01 0 00 000000#          HRROI   1,IBUFR           ;point to the input line for PSOUT
    25  000016'02  104 00 0 00 000076           PSOUT                     ;echo (retype) the input line.
    26  000017'02  254 00 0 00 000001'          JRST    GETLIN            ;Go get another line.
    27
    28                                   INERR:  HRROI   1,[ASCIZ/Error from RDTTY.  I give up
    29  000020'02  561 01 0 00 000042'  /]
    30  000021'02  104 00 0 00 000076           PSOUT                     ;send an error message and stop.
    31  000022'02  104 00 0 00 000170   STOP:   HALTF                     ;stop the program here
    32  000023'02  254 00 0 00 000022'          JRST    STOP              ;In case of a CONTINUE command,
    33                                                                     ;  stay stopped.
    34
    35  000024'02  127 145 154 143 157  PROMPT: ASCIZ   /Welcome to Echo.  Please type a line: /
    36  000025'02  155 145 040 164 157
    37  000026'02  040 105 143 150 157
    38  000027'02  056 040 040 120 154
    39  000030'02  145 141 163 145 040
    40  000031'02  164 171 160 145 040
    41  000032'02  141 040 154 151 156
    42  000033'02  145 072 040 000 000
    43
    44                   000000'                 END     START             ;specify the start address.

NO ERRORS DETECTED

PROGRAM BREAK IS 000000
PSECT 1 BREAK IS 000040 FOR  DATA
PSECT 2 BREAK IS 000051 FOR  CODE
CPU TIME USED 00:01.611

57P CORE USED
```

Figure 8.2: Assembler Listing of the Source Program

```
ECHO INPUT LINE -- Example 3    MACRO EX3     MAC      9-Nov-2004 12:36              SYMBOL TABLE

BUFLEN          000040  spd
DATA            000000  ext
ERJMP   320700  000000  int
HALTF   104000  000170  int
PSOUT   104000  000076  int
RDTTY   104000  000523  int
RESET   104000  000147  int

ECHO INPUT LINE -- Example 3    MACRO EX3     MAC       9-Nov-2004 12:36              Symbol Table for PSect   DATA

CODE            000000  ext
IBUFR           000000'

ECHO INPUT LINE -- Example 3    MACRO EX3     MAC      9-Nov-2004 12:36              Symbol Table for PSect   CODE

GETLIN          000001'
INERR           000020'
PROMPT          000024'
START           000000'
STOP            000022'
```

Figure 8.3: Assembler Listing of the Symbol Tables

```
 Symbol Cross–Reference
BUFLEN       4#      7      15
GETLIN      12#     26
IBUFR        7#     14      19      24
INERR       18      28#
PROMPT      12      16      35#
START       11#     44
STOP        21      31#     32

 Operator Cross–Reference
ERJMP       18
HALTF       31
PSOUT       13      23      25      30
RDTTY       17
RESET       11
```

Figure 8.4: CREF Listing of Cross–Reference to Symbols and Operators

- An informative heading at the top of each page,
- The location counter, and assembled code for each line of the program, and
- A symbol table of all user–defined symbols.

The `/CREF` switch in the `COMPILE` command requests extra output from MACRO. This extra output information is processed by the CREF program to produce a cross–reference listing. The cross–reference listing includes

- Line numbers at the left margin, and
- Two tables at the end of the listing: the first table contains the names of normal user–defined symbols; the second includes all the special operators and macros used in the program.[2]. For each symbol, the line number of every line on which the symbol is referenced is printed. A sharp sign (`#`) is printed next to the line number where a symbol is defined.

The listing that MACRO can produce by itself (via the `/LIST` switch in the `COMPILE` command) is much less useful than the combined efforts of MACRO and CREF. When you need a program listing, make a CREF listing rather than a simple listing of your source.

8.2.1 Page Headings

MACRO produces a page heading on each page of the listing file. A sample heading appears below:

```
ECHO INPUT LINE -- Example 3    MACRO EX3     MAC      9-Nov-2004 12:36
```

The first line of the heading contains the text argument to the `TITLE` statement, the version of MACRO, and the date and time when MACRO was run.

The page numbering corresponds to the page numbers in the source file; each time a form–feed character (i.e., CTRL/L) appears in the source, MACRO starts a new piece of paper and counts the file page number. (This corresponds to the notion of pages in the EDIT program.) When a file page is too long to fit on one piece of paper, MACRO will number subsequent pieces of paper with, e.g., `1-1`, `1-2`, etc. It is a good idea to use separate pages for independent blocks of instructions, e.g., large subroutines.

The second line of the heading shows the name and write–date of the source file.

8.2.2 Listing the Source Lines

From left to right, each line in the listing contains the following:

- The CREF line number. This number counts by one in decimal for each line that MACRO writes. These are line numbers that are referred to in the cross–reference tables that appear after the program listing.
- MACRO's storage location counter. This number shows where MACRO is planning to put the binary code generated for this line. Usually this number counts by one in octal for each line of code. Some pseudo–ops, for example, `BLOCK`, may cause the location counter to advance by

[2]See the discussion of `OPDEF`, Section 13.4, page 179, and the discussion of macros in Section 17, page 235.

more than one. (This isn't visible in the current example because nothing in the DATA psect follows `BLOCK`.)

- The location counter may be flagged with an apostrophe character (`'`) signifying that the value of the counter is *relocatable*. The relocatable symbols and the loader are discussed shortly.

  Following the location counter, if the contents of the line are associated with a named psect, MACRO's psect number will be printed. In this example, MACRO prints `01` for the DATA psect and `02` for the CODE psect. (There is an unnamed psect that we have not used; its number in MACRO is 0, and MACRO avoids printing the number.)

- Next on the line are octal numbers signifying the contents of various fields of the assembled word. For most instructions, five octal fields are printed. These correspond to the `OP`, `AC`, `I`, `X` and `Y` fields of the instruction. For words other than instructions, MACRO displays the data it assembled in some other format. For example, lines 35 through 42 each show the five 7–bit fields that MACRO assembled for the `ASCIZ` pseudo–op.

- Finally, the source text of the line is printed, including comments.

On line 21 we see that the assembler is planning to store a word in location `12'02`; the apostrophe after the 12 signals that this value is *relocatable*. The `02` signifies this location is in the CODE psect. To say that a word or address is relocatable means that the loader is going to add some relocation constant to each of these addresses; this location will be moved to be 12 words after the start of the CODE psect. We will discuss relocation further in the next section; meanwhile, it is important to note that locations such as relocatable 12 are not included among the accumulators, even though it seems to be a small enough address.

Continuing on line 21, the operation code has been assembled as 254; this corresponds to the `JRST` instruction. The accumulator field is 0, as are `I` and `X`. The `Y` field is relocatable 22. This must be the assembler's translation of the symbol `STOP`. You may verify this assumption: line 31 shows the definition, `STOP:` with the location counter set to `22'02`. Thus we see the effect of some of the assembler's bookkeeping functions. The symbol `STOP` is defined as 000022'02 (i.e., relocatable 22 in psect number 2); at places where `STOP` is referenced that definition has been substituted for the symbolic name. On line 32, for example, the reference to `STOP` has also been translated to `22'`.

On line 15, the expression `5*BUFLEN-1` has been assembled to 237. Note that no apostrophe follows the 237 in the code. This means that 237 is not relocatable; symbols and expressions that are not relocatable are said to be *absolute*.

Line 22 displays a literal. Note that MACRO indicates `35'` as the address part of this instruction. Thus, MACRO must have decided to put the assembled code corresponding to this literal at location `35'02`. Note that line 42 in Figure 8.2, the last line printed that consumes a location, uses location `33'02`. This literal has been placed in a location at a higher address than anything else that was seen. But why did MACRO omit location `34'02`? MACRO didn't skip `34'02`: that is where it put `[ASCII/LEAVE/]`, as you can see on line 20. Literals are normally placed after all other material has been assembled.[3]

Lines 35 through 42 show the assembler's translation of the `ASCIZ` pseudo–op. Line 35 shows the location counter, `24'02`, and the octal for the first five characters of the prompting message. The other characters appear on subsequent lines.

Line 24 demonstrates an exception. The reference to `IBUFR` has been rendered as `000000#`. This occurs because `IBUFR` is defined in a psect other than the one for which code is being assembled. The characters `000000#` do not have an intrinsic meaning. Instead, the `#` means that magic has been

[3]See also the discussion of the `LIT` Pseudo–op in [MACRO].

requested: MACRO will tell the loader that it wants location 0 relative to psect 1. Although we've told MACRO to tell LINK to put the DATA psect at `1001000`, MACRO does not, in general, know where psects will be loaded. Thus, it leaves to LINK the task of knitting the inter–psect references together at load time.

8.2.3 Listing the Symbol Table

Examination of Figure 8.3 shows the entire symbol table, with symbol names arranged alphabetically. The symbols are divided into three categories by the different psects in which they were defined.

- The symbols defined in MONSYM are listed in the symbol table for the unnamed psect; the symbol `BUFLEN` is defined here also. (This is becaue the definition of `BUFLEN` appears before the first `.PSECT`.)
- The symbol `IBUFR` is defined as `000000'` in the symbol table for the DATA psect.
- Several symbols, `GETLIN`, *INERR*, etc are defined in the symbol table for the CODE psect. `START` is defined as `000000'` in that symbol table.

The symbol tables are listed separately because they are sent to LINK separately: for each different symbol table a different relocation value is applied. Thus although both `START` and `IBUFR` are defined to be `000000'` they are not the same location because different relocation values are applied in different psects.

Most labels have relocatable values. These include `START`, `GETLIN`, and `IBUFR`. The symbol `BUFLEN` is unique among the user–defined symbols in that it is not relocatable.

Some symbols have 36–bit values. `PSOUT`, for example, is defined as `104000 000076`. You might notice that all the JSYS values start with operation code 104; that is the JSYS instruction itself. The right half value, 76 in the case of `PSOUT`, specifies which system function is wanted.

Some symbol values are followed by a three–letter code. The code "`spd`" means suppressed, and applies to our one suppressed symbol, `BUFLEN`. The code "`int`" means *internal* and applies in this case to those symbols that have been added to MACRO by the effect of the `SEARCH` pseudo–op. Internal symbols are those that are available or visible to other programs that you load.

Finally, `DATA` appears as an "`ext`", external, symbol in the unnamed psect's symbol table, and `CODE` appears as an external in the DATA psect's symbol table. The author is at loss to completely explain these.

8.2.4 Symbol Cross–Reference

In Figure 8.4 the portion that we have labeled "Symbol Cross–Reference" contains a line for every user–defined symbol. These lines report all references to and definitions of each symbol.

The line for the symbol `STOP` reports that `STOP` is referenced on lines 21 and 32. The sharp sign (#) after 31 means that `STOP` is defined on line 31. These numbers refer to the line numbers at the extreme left of the listing.

8.2.5 Operator Cross–Reference

The operator cross–reference has the same format as the symbol cross–reference. Operators (including the JSYS names and `ERJMP`) are listed in this section along with the line numbers where they are used. Operators can be defined by the user, see Section 13.4, page 179; such operators would be listed here also.

8.3 The Loader and Relocatable Code

The assembler reads a text file containing a program and writes a file containing the *binary relocatable* version of the program; this file has file type `REL` signifying *relocatable*. This `REL` file contains a version of the program that is very close to machine language.

However, the `REL` file cannot be run directly. In order to run a program, the program must first be loaded into memory. A special program called a loader is used for this purpose. The name of the loader program is LINK.

LINK reads `REL` files and loads them into memory.

By default, LINK loads the unnamed psect starting at location 140 in section 0. Traditionally, in TOPS–10 the locations in the range from 20 through 137 have been used for communication between the program and the operating system. This region is called the *Job Data Area*, and we shall occasionally make use of some of the locations that are provided there.

There is no default for loading named psects. Either the user communicates the psect origin to `Link` via commands typed to its command line processor or the psect origin is communicated as we have done via MACRO's `.PSECT` pseudo–op.

Normally, MACRO will assemble what is called *relocatable code*. The LINK can place relocatable code anywhere in memory. Symbols that label relocatable code are themselves relocatable values. Because MACRO doesn't know where the program will be put by LINK, `Macro` includes enough information in the `REL` file to allow LINK to *relocate* the program anywhere in the memory space.

In example 1, when we said that MACRO assigned the value `1001004` to the symbol `MESAGE` we did not tell the entire truth.[4] Actually MACRO assigned the value `4'01` to the symbol `MESAGE`. Again, the apostrophe after the `4` means that the value is relocatable; the `01` following the apostrophe names the psect number whose relocation constant will be used when loading the program. The relocation mark (or its absence) is associated with every symbol definition and with every word that MACRO assembles.[5] When MACRO sees a reference to a relocatable symbol, that reference is passed to LINK as a relocatable reference. Specifically, when MACRO saw the line `HRROI 1,MESAGE` it assembled a word containing `561040000004` and associated with this word a right–side relocation mark and psect number 1.

When LINK sees a relocatable value, it adds the *relocation constant* to it to obtain an absolute address; the relocation constant is essentially the address of the first location where LINK started loading the psect. (Yes, LINK keeps a list of all psects and their origins.) In this way, MACRO defers until load–time the decision about where to put the program. In the case of this `HRROI` instruction, LINK adds the psect origin of psect number 1 (the CODE psect), `1001000` to the right half of the word supplied by MACRO. LINK notices that the sum, `1001004` is a section 1 address; it also sees that it is planning to store the result word in section 1. LINK discards the section number from the

[4] Not wishing to label himself a liar in print, the author notes that the awful truth, abridged, was mentioned in a footnote.

[5] Actually, there are two relocation marks (or absences) for each word — one for the left half and one for the right.

sum; it stores the in–section component of the sum, `1004`, in the right half of the target word and proceeds without complaint. Thus, LINK and MACRO contrive to place the value `561040001004` into the appropriate location. The entire program is relocated in this fashion.

Not all symbols are relocatable. Usually labels are relocatable and most other symbols are not. For example, the symbol `PSOUT`, value `104000000076`, is absolute. Absolute symbols and expressions are not relocated by LINK. In our discussion of the assembler listing, we examined a line that said `MOVEI 2,5*BUFLEN-1`. The value of the expression `5*BUFLEN-1` is 237; note that the 237 does not have an apostrophe. Because the 237 is absolute, LINK will not change it when the program is loaded into memory.

Relocatable code is used for several reasons. The predominant reason is that the loader is more flexible than we have yet described. LINK allows your program to be combined with other programs that have been assembled or compiled separately. This allows you to take advantage of subroutines inside of other program sources, and written in other languages. Because the author of a subroutine cannot know what kind of program will call the routine, the he or she cannot be sure where to put the program in memory.

Relocatable code solves the problem of where to put things. Instead of deciding in advance where to put the various pieces of code, such decisions are deferred until the various program modules are loaded together. Relocatable code allows each module to be loaded wherever it's convenient. By contrast, if extensive use is made of absolute locations, conflicts over the use of particular locations can arise when several routines are loaded.

Large programs are built from several separately assembled modules. When one module changes, the program can be reconstructed by reassembling only the changed module. In some circumstances, modules can be developed and debugged independently. This is a great savings in program development time. The use of separately assembled programs is made possible by the linking loader.

Further information detailing the features of the assembler and loader can be found in [MACRO] and [LINK].

Chapter 9

Debugging with DDT

DDT is a program that helps us examine a program and debug it. DDT includes many powerful functions to assist our efforts to understand what the program is doing.

In addition to knowing about DDT, you should be aware of the list of common pitfalls displayed in Appendix E, page 661. When a problem arises, you might want to review that list before attacking the problem with DDT.

9.1 DDT Functions

Among the functions included in DDT are

- **Symbolic Addressing:** The symbol table that MACRO builds is generally loaded into memory along with the program. DDT allows us to reference locations in the program by their symbolic name. When we mention a symbolic name, such as `START`, DDT will look up that name in the symbol table. The name will be translated to the appropriate numeric address.
- **Examine and Deposit:** The first important tool that DDT implements is the ability to examine and deposit locations in memory. The program and its data areas reside in memory; they can be referenced by the symbolic names that were used as labels in the source program.
- **Symbolic Disassembly:** The ability to examine locations in memory is extremely important; DDT would be quite useful even if it were limited to displaying the contents of memory as octal quantities. However, DDT will interpret and display memory locations in any of several different formats.

  The contents of a location can be interpreted and displayed as instructions, ASCII text, octal numbers, decimal numbers, floating-point numbers, bytes, and in several other ways.
- **Symbolic Assembly:** DDT permits us to change the contents of memory by means of depositing new values in specified locations. DDT allows us to express the value that we want to deposit in several different formats.

  DDT understands how to assemble instructions. It can assemble octal, decimal, and floating-point numbers. It assembles ASCII text and some other formats.

- **Breakpoints in the Instruction Stream:** We can use DDT to replace an instruction with a *breakpoint* instruction. The breakpoint instruction is actually a subroutine call to DDT. When the computer executes the breakpoint instruction, control is transferred to DDT. When DDT is entered from a breakpoint, the instruction at which the breakpoint was placed has not yet been executed. We can then examine and deposit locations, place additional breakpoints, and either proceed from this breakpoint or single step.

  DDT uses the keyboard for control. Therefore, it's nearly impossible to use DDT to single step or breakpoint the portion of the program that reads from the terminal.

  Since breakpoints are implemented by storing different instructions in your program, you must avoid placing a breakpoint at any instruction that is used as data elsewhere in the program. For example, the `ERJMP` instruction is implemented by TOPS–20 reading the instruction as data. Thus, you should avoid placing a breakpoint on an `ERJMP`.

- **Single Step Instruction Execution:**

  When DDT reaches a breakpoint, you may single-step the instruction at which DDT is pausing. When an instruction is single stepped, DDT displays the instruction, the operands, and the results. After one instruction has been single stepped, DDT pauses before the next instruction. After any examines or deposits, the next instruction can be single stepped, or DDT can be told to allow the program to proceed at full speed to the next breakpoint.

9.2 Loading and Starting DDT

In TOPS–20, DDT may be added to the program at any time by means of the `DDT` command to the EXEC. Alternatively, DDT can be included with a program by using the EXEC's `DEBUG` command or by use of the `/D` switch in LINK.

Once DDT is present with a program, it can be entered by either:

- the `DDT` command in the EXEC,
- by executing a breakpoint instruction, or
- by executing an instruction that explicitly jumps to DDT.

While in DDT, you can examine and modify the contents of the accumulators and other memory locations. The execution of the program may be started or resumed by:

- the command `adr$G`. That is, type a numeric or symbolic address, the ESCAPE key,[1] and `G` to start execution at the specified address, or
- the `$P` command (type ESCAPE followed by `P`) to proceed from a breakpoint instruction.

After you interrupt the execution of a program by typing `CTRL/C`, you can then enter DDT by typing the EXEC's `DDT` command. *Before* giving the `DDT` command, you should use the `CTRL/T` facility to discover the value of the program counter. You must remember the value of the program counter if you plan to continue the execution of the program after your session in DDT.[2]

[1] If your terminal does not have a key labeled ESCAPE or ESC, you might try ALT-MODE or CTRL/[.

[2] At XKL, the EXEC's `DDT` command tells you the program counter value before it starts DDT; but you must still remember it.

9.3 A Sample Session with DDT

Before we go into the details of all the DDT commands, a brief demonstration of DDT seems appropriate. We will use DDT to examine our program from example 3, Echo Line. In the description that follows, the user-supplied input is shown with underlines.

We begin by using the EXEC's `LOAD` command to assemble and load the file `EX3.MAC`:

```
@load ex3
MACRO:  ECHO
LINK:   Loading

EXIT
@
```

Now, this program could be started via the EXEC's `START` command. Rather than do that, we'll add DDT to our memory space via the `DDT` command. DDT is loaded and started; it responds with the message "`DDT`" to tell us that it is listening:

```
@ddt
DDT
```

The first thing to do is to open this program's symbol table. Remember, the name of the symbol table is taken from the first word of the `TITLE` statement in the program. MACRO tells us the symbol table name as it assembles the program. In this case the name is `ECHO`. We tell DDT to open the symbols for the program named `ECHO` by mentioning the program name and typing the command characters ESCAPE and colon.[3]

Notice that when we type the ESCAPE key, DDT displays a dollar sign character. Throughout this discussion of DDT, the dollar sign characters that appear in the examples represent places where we have typed the ESCAPE key.

We type precisely the six characters "e", "c", "h", "o", ESCAPE, and colon:

```
@ddt
DDT
echo$:
```

DDT types a tab character after our colon to signify that it has accepted our command. The symbols for the `ECHO` program are now available to DDT.

We type a symbol name, `START`, and a slash character `(/)` to *open* the location called `START`. When DDT opens a location, it examines the contents of the location and displays them in the prevailing *type-out mode*. Initially, the display mode is symbolic; locations are interpreted as instructions and symbolic addresses.

```
@ddt
DDT
echo$:   start/   RESET
```

[3]Modern versions of DDT do not insist that you open a symbol table; so long as names that you type are unambiguous, DDT can get by.

The contents of the location called START have been displayed. This is the RESET JSYS. To continue examining locations, type the LINE FEED key.[4] The symbolic name of the next location will be displayed, followed by a slash, and the contents of that location. After the location START is displayed, we type line feed four times:

```
@ddt
DDT
echo$:   start/   RESET                  type line feed
GETLIN/     HRROI 1,PROMPT               type line feed
GETLIN+1/   PSOUT                        type line feed
GETLIN+2/   HRROI 1,IBUFR                type line feed
GETLIN+3/   MOVEI 2,.JBDA+77
```

Note that in addition to displaying the contents of each location as an instruction and symbolic address, DDT has also displayed the location address itself in symbolic form.

Now, at GETLIN+3 we discover one of the failings of DDT. Up to this point, DDT has been doing a fine job of symbolic disassembly. Each word has been interpreted properly as an instruction. However, at GETLIN+3 DDT has stumbled; it has wrongly interpreted the address field as .JBDA+77. In our source program, we wrote 5*BUFLEN-1 as the address field.

Simply stated, the source program cannot be recovered by DDT. As the user of DDT, you must be sufficiently familiar with the program to avoid going astray here.

Because we recognize .JBDA+77 as an anomaly of DDT, we can ask DDT for an alternative interpretation. Type an equal sign, "=", and DDT will respond with the octal equivalent for this word:

```
GETLIN+3/    MOVEI 2,.JBDA+77    =201100,,237
                                 ↑  type equal sign here.
```

It is hoped that the display of the address field as 237 will jog our memory. You should recall from the listing of the program that 237 is the result that MACRO assembled for the expression 5*BUFLEN-1.

Continuing our progress through the program, we type several more line feed characters:

```
GETLIN+3/   MOVEI 2,.JBDA+77   =201100,,237  type line feed
GETLIN+4/   HRROI 3,PROMPT                   type line feed
GETLIN+5/   RDTTY                            type line feed
GETLIN+6/   ERJMP INERR                      type line feed
GETLIN+7/   MOVE 15,IBUFR                    type line feed
GETLIN+10/  CAMN 15,PROMPT+10
```

Let us place a breakpoint at GETLIN+10. To accomplish this, we must mention the address where we want to place the break. In DDT, the symbolic name period, ".", signifies the current location; at this moment, GETLIN+10 is the current location. We type the characters period, ESCAPE, and B to establish a breakpoint here. (We might just as well have typed the command GETLIN+10$B, but .$B is a convenient shorthand.)

[4] If your terminal has no LINE FEED key, type CTRL/J.

```
GETLIN+10/   CAMN 15,PROMPT+10   .$b
```

Here, DDT delivers a raspberry:

```
GETLIN+10/   CAMN 15,PROMPT+10   .$b?not writable
```

Breakpoints modify code. We have placed our code in a read–only psect to keep it from being stepped on. DDT reports, a bit rudely, that it can't place the breakpoint because the code is not writeable.

Type ESCAPE and W This is a command to DDT that tell it to override the protection of the code and put in the breakpoint. (This command gives DDT permission to write where we have protected the program; when DDT is done with its writes, it leaves the program protected as before.) We then repeat the command for installing the breakpoint:

```
GETLIN+10/   CAMN 15,PROMPT+10   .$b?not writable  $w   .$b
```

Now that we have a breakpoint, we can start the program. The command characters ESCAPE and G tell DDT to start the program at its normal starting location; in this case, we begin execution at START:

```
GETLIN+10/   CAMN 15,PROMPT+10   .$b?not writable  $w   .$b   $g
Welcome to Echo.  Please type a line:
```

The program prompts for input, via the PSOUT JSYS. The RDTTY is now being executed: TOPS–20 is collecting a line of input to give to the program. We supply the line, "This is a test." and type return. When we type carriage return, TOPS–20 adds a line feed character and gives the entire line to the program. The RDTTY JSYS returns control to the part of the program that we wrote. Soon, the execution of the program arrives at the instruction where we placed the breakpoint. Instead of executing the instruction, DDT is called. DDT displays the location and contents of the current breakpoint.

```
Welcome to Echo.  Please type a line: This is a test.
$1B>>GETLIN+10/   CAMN 15,PROMPT+10
```

We are once again in DDT. Further examines are now permitted. We type the address of an accumulator, 15, and a slash, to examine what the accumulator contains:

```
$1B>>GETLIN+10/   CAMN 15,PROMPT+10    15/   HLLOS 4,171500(15)
```

The contents of register 15 have been interpreted as an instruction. Since this interpretation is not meaningful, we must think of some better way to interpret this location. Since register 15 has just been loaded from the input buffer, the most likely interpretation for the contents of accumulator 15 is as ASCII text. While register 15 is still open, type the command characters ESCAPE, T, and a semicolon:

```
$1B>>GETLIN+10/   CAMN 15,PROMPT+10    15/   HLLOS 4,171500(15)   $t;This
                                  Type ESCAPE, T, and semicolon   ↑
```

The command characters ESCAPE and T set the display mode to ASCII text. The semi-colon is a command character that tells DDT to retype the previous value. That value is typed again, but in the new mode. We discover that register 15 contains the word "This". Actually, register 15 also contains a blank character following the four letters "This". Because characters like blank can't always be seen, there is an alternative display mode. Type the command characters ESCAPE, 7, the letter O, and semicolon:

```
... 15/   HLLOS 4,171500(15)   $t;This   $7o;124,150,151,163,40,0
```

The command $7O changes the output mode to 7-bit bytes; the semicolon command character requests that the current value be retyped. The consecutive 7-bit fields of the word are displayed. The number 124 corresponds to the letter T, 150 to h, etc. The fifth field, 40, is a blank character. Since five 7-bit fields do not entirely exhaust the 36-bit word, a sixth field is printed; you should be aware, however, that the sixth field represents only one bit.

Type a carriage return to close this location. When carriage return is typed, the display mode reverts to whichever mode has been selected as the *permanent mode*. We have seen two mode changing commands, $T and $7O. To change the output mode permanently, type two ESCAPE characters instead of one, e.g., $$T permanently changes the display mode to show words as text. Of course, a "permanent" change lasts only until the next "permanent" change. By the way, to change the mode back to the symbolic display of instructions, use the command $S or $$S.

We go on now, to examine the contents of the input buffer. Type the symbolic name IBUFR and a slash character:

```
ibufr/   HLLOS 4,171500(15)
```

Again, the display mode is wrong. Type $T and a semicolon to redisplay this word as text:

```
ibufr/   HLLOS 4,171500(15)   $t;This
```

We can go on to examine the remainder of the input buffer by typing consecutive line feed characters:

```
ibufr/   HLLOS 4,171500(15)   $t;This     type line feed
IBUFR+1/   is a                           type line feed
IBUFR+2/   test.                          type line feed
IBUFR+3/
```

Note that typing line feed to examine sequential locations does not change the display mode. The temporary mode that was selected on the line where we examined IBUFR remains in effect until we change it or until we type carriage return.

The last word that we examined, IBUFR+3, is difficult to see properly in text mode. We know what to do to display this word as 7-bit fields: we type $7O;.

```
IBUFR+3/
   $7o;15,12,0,0,0,0
```

From this display we can see that the word at IBUFR+3 contains just the two characters carriage return (the 15) and line feed (12). The remainder of this word, and the remainder of the input buffer contains zero (null) characters.

We have reached a point where we should be comfortable with allowing this program to proceed. Type the command characters ESCAPE and P. The instruction at the breakpoint will be executed, and execution resumes:

```
$p
The line you typed was: This is a test.
Welcome to Echo.  Please type a line:
```

As the program continues, it types a heading and echoes the line that we typed. It loops to prompt again for input. We supply another line. After we type carriage return, the program runs until it hits the breakpoint:

```
Welcome to Echo.  Please type a line: LEAVE now
$1B>>GETLIN+10/   CAMN 15,STOP+2
```

Open register 15, then select text mode, and retype it:

```
$1B>>GETLIN+10/   CAMN 15,STOP+2   15/ SETCMM 2,@153212(10) $t;LEAVE
```

We can see that register 15 contains the text "LEAVE".

An examination of the input buffer area is interesting:

```
ibufr/   SETCMM 2,@153212(10)   $t;LEAVE      type line feed
   FR+1/    now
```

The word at IBUFR+1 includes a carriage return character but no line feed. DDT typed

```
IBUFR+1/    now
```

followed by a carriage return and three spaces. The return set the display cursor to the left margin and the three spaces obliterated "IBU"; the cursor was left blinking under the letter F.

Type carriage return, then period and slash, then $7o;:

```
./   MOVES 7,773432(15)   $7o;40,156,157,167,15,0
```

This shows the characters space, "n", "o", "w", and carriage return in this word. Type line–feed:

```
./   MOVES 7,773432(15)   $7o;40,156,157,167,15,0      type line feed
IBUFR+2/   12,0,163,164,56,0
```

Note that IBUFR+2 contains the line feed and the null character that terminates this string. The remaining characters are left over from the previous string (the "s", "t", and "." of "test.").

Again, we type $P to proceed from the breakpoint. The program detects that "LEAVE" was typed; it executes the HALTF at STOP, and the EXEC prompts for another command:

```
$p
@
```

9.4 Methodical Debugging

Since few complicated programs are written without mistakes, some words about debugging programs systematically seem appropriate. There is no magic recipe for effective debugging. However, there are guidelines that are generally useful. Among these are

- Your most useful debugging tool is located between your ears.
- Plan your debugging when you write the program.
- Divide the program into well-defined loops and subroutines.
- Don't expect anything to work the first time; be skeptical of your code. Test each subroutine and each loop. There's little point in checking the second subroutine until you have verified that the first one works.
- Loops should be verified by a breakpoint before the loop to establish that the initial conditions are properly set up, and a breakpoint following the loop to check that the loop itself has functioned properly. This applies to subroutines as well. In short, know what your program is supposed to do. Know what to expect at various places in your program; examine these places to detect discrepancies between expectation and observed reality.
- If you can, avoid breakpoints in the loop itself, although if the loop has failed, you must investigate the interior. Loops that don't terminate, improper use of index registers, and incorrect indices are all potential sources of trouble inside a loop.
- Failure to properly initialize memory and the accumulators can cause incorrect results that are difficult to reproduce. Among the possible oversights are failure to initialize the stack pointer register, failure to explicitly zero memory locations (and accumulators) in the initialization of the program, and failure to perform a `RESET` JSYS in the startup sequence.
- Erroneous arguments to JSYS calls and subroutines can cause much confusion.
- A breakpoint changes your program. Do not place breakpoints on data. `ERJMP` is both program and data: do not place breakpoints on `ERJMP`.

When a program doesn't work, it is wise to adopt a skeptical attitude towards the code that is there. It makes no difference that someone you trust may have written it. You're debugging it because, no matter how reliable the author, there is a bug. The most difficult bugs to find are those located in the place where you know the program works "perfectly." Another difficult class of bugs are those of omission: it's hard to find what isn't there.

9.5 DDT Command Descriptions

DDT is a large and changing program. It contains many features, some of which are confusing to novice users. The command descriptions that follow are an attempt to present the most widely used commands. A more complete, though terse, description of DDT features appears in Appendix C, page 643.

You might scan the material here once to become familiar with the range of DDT commands. Detailed reading of selected portions of the appendix may be undertaken when it is necessary to apply DDT to debug specific problems.

In the description of DDT commands, the following rules of nomenclature apply:

- The dollar sign character (`$`) signifies places where the ESCAPE key must be typed. This key is labeled as ESC or ALT–MODE on most terminals. When you type the ESCAPE key, DDT will display a dollar sign.
- Numbers are represented by n. Numbers are interpreted as octal, except that digits followed by a decimal point are base ten; if digits follow the decimal point, a floating-point number is assumed.
- A number that follows an ESCAPE, written as `$`n, is interpreted as either octal or decimal, depending on the command to which it applies. Each command that has such an argument should say which radix applies.

9.5.1 Examines and Deposits

In order to examine a location, you must first specify the address of that location. You may specify the location that you wish to examine by any numeric or symbolic address expression. Simple symbolic expressions, such as `TABLE+5`, are allowed. Type the name or number of the memory location (or accumulator) that you wish to examine, followed by one of the command characters, e.g., `TABLE+5/`.

When a location is examined, the contents of that location are displayed. Initially, the *mode* in which locations are displayed is *symbolic*; that is, the contents of locations will be interpreted as instructions. The addresses of locations will be interpreted as labels where possible. The radix for displaying numbers initially is octal. See Section 9.5.2, page 110 for the commands by which you can change the display mode or the radix.

To *open* a location means to read and (usually) display the contents of the location. You may deposit new data into an open location by typing a new value followed by a command that performs a deposit; DDT will store the new value, obliterating the previous contents. Data, instructions, or the contents of the accumulators may all be changed in this way.[5]

Some special symbols exist in DDT. Among the most important of these are the *current location*, called by the symbolic name period (`.`), and the *current quantity* called `$Q`. Additionally, there are special symbols called *masks*. Each mask controls some function within DDT; for example, the *search mask* (`$M`) affects the DDT word searches.

9.5.1.1 Current Location

The character period (`.`) is the symbolic name of the *current location*. Most commands that open locations set the current location to the address that has just been opened.

9.5.1.2 Current Quantity

The symbol `$Q` is the symbolic name of the *current quantity*. The current quantity is either the last value typed by DDT (i.e., the value of the location most recently displayed), or any new address or value that you have typed. Some DDT commands use the right half of the current quantity as an address when no address argument is specified.

The value of the current quantity with right and left halves swapped is accessible by the symbolic name `$$Q`.

[5] If the location you want to deposit into is write–protected, DDT will complain. You can override write–protection in DDT by the command `$W`; `$$W` restores DDT's respect for write–protection.

9.5.1.3 Examine Commands

`addr/` Opens the location specified by the address expression `addr`. The contents of that location are displayed in the current mode. The *current location* (`.`) is set to this address.

If no address expression is mentioned, DDT will open and display the location addressed by the right half of the current quantity; when no address expression is mentioned, DDT will avoid changing the value of "`.`".

`addr[` Opens the specified location; displays its contents as a number in the current radix; DDT will change "`.`" to this address. If no address expression appears, "`.`" is not changed; the address to open is taken from the right half of the current quantity.

9.5.1.4 Deposit Commands

Be careful what you type after you have opened a location for examination. If you type an expression and any of the following commands, you will change the value of an open location. Better to close an open location, by typing carriage return before you type any other expression, before trying new things.

RETURN If a new value has been typed, that value is deposited. The open location is closed. Any temporary display mode that is in effect is cancelled; further displays will default to the prevailing permanent display mode.

LINE–FEED Deposits any new value and closes the open location. Opens `.+1`, that is, the next consecutive location. Displays the contents in the current (temporary) mode.

`^` Deposits any new value and closes the location. Opens `.-1`; displays the contents of that location in the current mode.

BACKSPACE The backspace (`CTRL/H`) command is the same as the `^` command; it deposits any new value and closes the current location. Then `.-1` is opened and displayed in the current mode.

TAB Deposits any new value and closes the current cell. Use the right half of the current quantity as the address of the next cell to open. Does not clear temporary modes. Changes "`.`".

By using only the right half, TAB as a command stays in the current section. If an open location contains an address that you want to follow to another section, you can type, e.g., `$Q/` to use the current quantity as the 30–bit address of the location to open.

9.5.2 DDT Output Modes

In the commands that follow, use the ESCAPE key once to set the mode temporarily. Type the ESCAPE key twice in succession to set the mode "permanently." The temporary mode is cleared by the RETURN command. The permanent mode may be changed by a subsequent command that sets a new "permanent" mode.

`;` The semi-colon character tells DDT to retype the current value in the current display mode. This command usually follows a command that changes the display mode.

`=` If the current radix is octal, the equal sign command makes DDT retype the current value in halfword numeric format. Otherwise, the current value is retyped as a fullword number in the current radix.

`$F` Display quantities as floating-point or decimal integer. DDT scrutinizes the quantity that is being displayed; if it looks as though it might be a normalized floating-point number, DDT will display it as a floating-point number. Otherwise the quantity will be displayed as a decimal integer.

`$`*n*`O` Display quantity as left-justified *n*–bit bytes. The number *n* is interpreted as decimal. If *n* does not evenly divide `36`, then one extra byte value will be output, but that value represents some smaller number of bits. The extra byte value will be displayed with extra zeros added at the *right*.

If *n* is omitted, the value of *n* set by the previous `$`*n*`O` command will be used.

If *n* is zero, the bits that are 1 in the byte mask, $3M, define rightmost bit of each byte position within a word. For example, if the byte mask contains `1061,,1`, then `$0O` would display the numeric values of the opcode, AC, I, X, and Y parts of a word.

`$`*n*`R` Set the display radix for numbers to the decimal value *n*. The radix must be larger than 1. When *n* is larger than 10, the digit value 10 will be typed as `A`, the value 11 will be typed as `B`, etc. Above radix 36, you may have difficulty interpreting the results.

`$S` Display the contents of locations in symbolic mode, i.e., as instructions. `$1S` display in symbolic mode; prefer machine opcode names to user–defined opcode names (`OPDEF`s).

`$T` Display quantities as seven–bit ASCII text. DDT tries to decide if the quantity is left–justified text or right–justified text, and displays the quantity accordingly. Sometimes, DDT guesses wrong, in which case the command `$7O` is helpful.

`$6T` Display quantities as sixbit text.

`$5T` Display quantities as Radix50 text and flags.

`$8T` Display 8–bit bytes as ASCII text.

`$9T` Display 9–bit bytes as ASCII text.

9.5.3 DDT Program Control

DDT allows you to stop the execution of your program at specific instructions by the installation of breakpoints. This section describes breakpoints, single stepping, and other ways by which you can use DDT to control the execution of the program.

`CTRL/Z` Exit from DDT. If you intend to resume debugging the current program, but need to get to the Exec, this is the command. For example, if you want to save a program that includes breakpoints, you must exit from DDT by Ctrl/Z. (If you were to leave DDT by typing Ctrl/C, DDT would not have the opportunity to put the breakpoints where they belong.)

`adr$G` Start execution at the location specified by the address expression. If the address expression is omitted, DDT will start the program at the same address that the Exec's `START` command would use.

`adr2<adr1$`*n*`B` Install a breakpoint at the location defined by `adr1`. (DDT cannot put a breakpoint at location zero.)

The optional decimal number, *n*, is the breakpoint number. If you omit *n*, the first available breakpoint number will be assigned to this new breakpoint. You may specify *n* to recycle an old breakpoint to a new location. If you exhaust DDT's supply of breakpoints (usually limited to eight, numbered from 1 to 8), you will have to select one to overwrite.

The optional address expression `adr2<` specifies the address of the location to automatically open and display whenever this breakpoint is hit. (It is not possible to automatically display location zero.) If you omit the expression `adr2<`, DDT will open `adr1`, the breakpoint location.

A breakpoint replaces the instruction at `adr1` with a jump to DDT. There are two cautions:

- If `adr1` is a read–only location, DDT will refuse to place a breakpoint there. As most instructions are placed in read–only psects, use the `$W` command to tell DDT to override the program's protections when doing deposits, including the store of the breakpoint instruction.
- If `adr1` is read as data, do not replace it with a breakpoint.
  For example, `ERJMP` has dual meaning. When the computer executes `ERJMP` as an instruction, it has no effect. However, when TOPS–20 discovers an error, it peeks at the instruction following the one that caused the error; if that instruction is `ERJMP`, TOPS–20 interprets it as a jump. So, if you were to put a breakpoint at a location that contains an `ERJMP`, you will hide the `ERJMP` from TOPS–20. Instead put one breakpoint at the address to which `ERJMP` would jump and a second breakpoint following the `ERJMP`.

When a breakpoint is installed by a command using two consecutive ESCAPE characters, e.g., `adr1$$B`, then DDT will proceed from the breakpoint automatically. Automatic proceed continues until DDT detects that characters are available from the terminal when the breakpoint is executed. See the description of `$$P` below.

`$B` Remove all breakpoints.

`0$`*n*`B` Remove breakpoint number *n*.

`$P` Proceed from the present breakpoint, or following the previous single stepped instruction. Execution of the program resumes at full speed until another (or the same) breakpoint is executed.

n`$P` Automatically proceed *n* times past this breakpoint, or until terminal input is present.

`$$P` Proceed automatically until the breakpoint is executed while input characters from the terminal are available. That is, this breakpoint will be passed automatically until you type something. Automatic proceed can make the program run very slowly: each time the breakpoint is passed DDT must perform a JSYS to determine if any terminal input is present. This test can add several hundred instruction times to a loop.

`$X` Single step the next instruction. You must be at a breakpoint or you must have previously used `$X` to single step an instruction. Repetitions of `$X` cause subsequent instructions to be single stepped. After single stepping, a `$P` command will resume the normal full-speed execution of the program.

When an instruction is single stepped, the argument and results of the instruction are displayed. Although single stepping is a very slow way to find out what a program is doing, it is worthwhile for novice users who may be uncertain of the effects of particular instructions.[6]

The command n`$X` will single step the next n instructions.

`$$X` Single step automatically and without typeout until the program counter reaches one or two locations beyond the address of the instruction that you are `$$X`-ing.

This command is useful for "single stepping" instructions that call subroutines. Single stepping is a very slow process. If the subroutine that is being `$$X`-ed is complex, it would be better to insert a breakpoint after the call to the subroutine, and then execute the subroutine at full speed.

(Author's note: this book is intended for people who are actually trying to learn this material. But there is the occasional ringer. If you're debugging the TOPS–20 time-sharing monitor, be aware that single–stepping was difficult to get right in user mode. It isn't possible with our instruction set to get it entirely right in executive mode. So, when your monitor runs off to never–never land after you type `$X` or `$$X`, I won't be surprised.)

`instr$X` If the expression `instr` has a non–zero value in bits 0–8, then DDT will execute `instr` as an instruction. If bits 0–8 of `instr` contain zero, then the expression is interpreted as a repeat count, as in the command n`$X`.

9.5.4 DDT Assembly Operations and Input Modes

DDT allows you to deposit new values into memory locations. First, open a location, then type a description of the new value. Finally, type one of the commands that closes a location and deposits a value (e.g., any of the RETURN, LINE–FEED, `^`, etc. commands).

To help you form the new values, several assembler functions are built into DDT. The general instruction format `OP AC,@Y(X)` is recognized and assembled as MACRO would assemble it. Specifically the names of the machine instructions are recognized by DDT. JSYS names that are used by the program are recognized. Symbols defined by the program are available for use by DDT's instruction assembler.

Neither literals, pseudo-ops, nor macros are available in DDT. However, DDT does provide commands for entering text and numbers.

DDT evaluates expressions using integer arithmetic. The operators `+`, `-`, and `*` work as you might expect for addition, subtraction, and multiplication, respectively. Because the slash character is used to open locations, division is signified by the apostrophe character (`'`).

A single comma in an instruction signifies the end of the accumulator field.[7] A pair of commas separates left half and right half quantities.

Parentheses may be used to signify the index register field. Technically, the expression that appears within parentheses is swapped (as in the `MOVS` instruction) and added to the word being assembled.

[6] Prior to Release 4 of TOPS–20, single stepping does not always give the same result as full-speed execution. If a JSYS call is single stepped, an `ERJMP` instruction that follows the JSYS in the normal instruction stream will not be visible to TOPS–20, because DDT copies the instruction being single stepped elsewhere before executing it. If you suspect that your program is acting differently while you single step it, use breakpoints instead.

[7] If an input/output instruction is being assembled, the comma signifies the end of the device number field.

The at-sign character (`@`) sets bit `13`, the indirect bit, in the expression being assembled

The blank character is a separator and adding operator in the instruction assembler.[8]

Numeric input is octal except that digits followed by a decimal point are decimal. If further digits following the decimal point are typed, input is floating-point. A floating-point number may be followed by `E`, an optional plus or minus, and an exponent.

In addition to the input formatting functions described above, special commands exist by which various text formats can be entered:

`"/text/` Left-justified ASCII text, at `5` characters per word. Instead of the slash (`/`) you may use any character that doesn't appear in the text itself. Repeat that character to end the input string.

To enter the sequence `CRLF` you must type only RETURN. To get a LINE–FEED alone, you may type LINE–FEED. RETURN without LINE–FEED can't be had.

For example: `"\this is a sample\`

`"x$` One right-justified ASCII character. Example: `"w$`

9.5.5 DDT Symbol Manipulations

DDT can change the symbol table that MACRO supplies. The changes include adding new symbols, removing symbols, and suppressing symbols.

To say that a symbol is *suppressed* means that the symbolic disassembler in DDT will not consider this symbol name as a possible name to output. However, the definition of a suppressed symbol is available when you use the symbol name as input. Suppressed symbols are sometimes called *half–killed* symbols.

`sym$:` Open the symbol table of the program named `sym`. As we have mentioned, the program name is set from the first six letters of the first word that follows the `TITLE` statement.

Once a program's symbols are opened, they are available for symbolic input and output. The reason that DDT insists that you specify a symbol table name is that in an environment where separately assembled programs have been loaded together there may be repetitions of symbol names among the several programs. Opening one program's symbols serves to eliminate any ambiguity in such cases.

`sym$K` Suppress (i.e., half–kill) the specified symbol. This symbol will no longer be available for output, but it will be recognized on input.

`sym$$K` Kill this symbol. This symbol is removed from the symbol table and will no longer be available for either input or output.

`sym?` Type out the name of each program in which the symbol named `sym` is defined.

`sym:` Define the symbolic name `sym` to have the value of the current location (i.e., "`.`"). If `sym` is already defined, this changes the old definition; otherwise, a new definition is added to the symbol table.

`val<sym:` Define the symbolic name `sym` to have the value specified by the expression `val`.

[8] When used as the adding operator, blank creates an 18–bit sum. For a 36–bit sum, use "`+`".

Chapter 10

Stack Instructions

A *pushdown list* or *stack* is a data structure in which items are removed from it (popped) in the reverse order that they were added to it (pushed). This reversal property, sometimes called *last–in, first–out*, is quite important in some algorithms.

The `PUSH` and `POP` instructions insert and remove full words in a pushdown list. In the PDP–10, a *stack pointer* that describes the location and extent of the area allocated to the pushdown list is usually kept in an accumulator. This accumulator is referenced in the `PUSH` and `POP` instructions.

With extended addressing there are several formats possible for the stack pointer:

- A *local stack pointer* contains a control count in left half and the address of the current stack top in the right half.

  If the control count is negative, the local format stack pointer can be used in any section. The control count is the negative of the number of stack words that are available for additional items.

  In section 0 only, the control count may be non–negative. In this case, the control count is the number of items that are now present on the stack.

- A *global stack pointer* contains the 30–bit address of the current stack top; this should address a non–zero section. This format is valid only in non–zero sections. There is no control count associated with this format.

  A global stack pointer is used when the stack is allocated in an addressing section that is different than the program's address section. This may be because there are multiple address sections that contain code or because the stack is large compared to an address section. (A large stack space is appropriate in situations where subroutines allocate their local variables on the stack.)

10.1 PUSH Instruction

The instruction

```
PUSH    AC,E
```

inserts (pushes) a copy of the word located at the effective address onto the pushdown list that is

defined by the stack pointer contained in the accumulator. This stack pointer is updated to reflect the addition of this item to the stack.

The precise action depends on whether the stack pointer is local or global:

- If the accumulator contains a local stack pointer then its right half addresses the old stack top. A `PUSH` instruction changes the stack pointer by adding `1` to both the left and right halves[1]. The new stack pointer addresses the new stack top at an address one higher than its previous value; the data at the effective address is copied to the new stack top. The stack data is local to the program section.

  If, as a result of the addition, the stack pointer changes sign from negative to positive, a pushdown overflow condition results (but the instruction proceeds to completion). Usually, the left half is initialized to make this warning mechanism effective.

With Local Stack Pointer or in Section Zero	
`PUSH`	C(AC):= C(AC) + <1,,1>; C(CR(AC)) := C(E) Stack overflow trap if C(AC) changes from negative to non–negative

- If the accumulator contains a global format stack pointer, it addresses the old stack top initially. A `PUSH` instruction changes the stack pointer by adding `1` to it. The new stack pointer addresses the new stack top at an address one higher than its previous value; the data at the effective address is copied to the new stack top.

With Global Stack Pointer in a Non–Zero Section	
`PUSH`	C(AC):= C(AC) + 1; C(C(AC)) := C(E)

10.2 Defining the Pushdown List

In the PDP–10, a pushdown list is simply an array of consecutive locations in memory. The `BLOCK` pseudo–op is usually used to allocate space for the pushdown list. If we want to define an array called `PDLIST` of size `PDLEN`, we might write

```
PDLEN==100                      ;define the size of the pushdown list

 . . .

PDLIST: BLOCK   PDLEN           ;allocate space for the pushdown list
```

When we speak of *the* pushdown list, we mean the one area in a program that has been allocated as a stack for general data and subroutine returns (see also the `PUSHJ` and `POPJ` instructions). When

[1] In the KI10 and later processors, any carry from bit `18` to bit `17` is suppressed.

we speak of *a* stack, we mean any such area that is used as a stack. Generally, programs will have only one stack area. However, when complex interactions among the various pieces of a program are possible, multiple stacks may become necessary. For instance, the TOPS–20 operating system requires several different stacks. Each stack requires a unique stack pointer; multiple stack pointers need not be kept in separate accumulators, but a stack pointer must be in an accumulator in order to use these instructions to affect it.

10.3 Initializing the Stack Pointer

From the description of the `PUSH` instruction it is possible to deduce how to initialize a stack pointer. Since the stack pointer always points at the current stack top, the initial pointer, which describes an empty stack, should point to one address before the first location allocated to the stack area. Symbolically, the the stack pointer should be initialized to the address `PDLIST-1`. For a global stack pointer this address must be a 30–bit value; for a local stack pointer, the address is eighteen bits.

The left half of a local stack pointer can be used as a control count. In the most usual case, we initialize the control count to be negative the number of words allocated to the stack area. Symbolically, this quantity is `-PDLEN`. In short, we can initialize the accumulator that will be used to hold the local stack pointer by means of one instruction:

```
MOVE    17,[-PDLEN,,PDLIST-1]
```

10.3.1 IOWD Pseudo–Operator

It happens that there is a pseudo–operator called `IOWD` that assembles this format of descriptor word. The following statement is equivalent to the previous:

```
MOVE    17,[IOWD PDLEN,PDLIST]
```

The `IOWD` pseudo–op assembles the negative of the first argument in the left half, and one less than the second argument in the right half.[2]

10.3.2 Symbolic Names for Accumulators

Programs that use a pushdown list usually make a permanent allocation of one accumulator to hold the stack pointer. Conventionally, register `17` is used for the stack pointer.[3] Just as we wrote `BUFLEN==40` to define a symbolic name for the length of the input buffer in example 3, we can write `P=17` to define the symbolic name `P` (short for pushdown stack pointer) as having the value `17`. Then when we write the symbol `P`,

[2]The name `IOWD` means Input/Output descriptor Word. The PDP–10 block input and output instructions (`BLKO` and `BLKI`) use this kind of descriptor. TOPS–20 "dump–mode"input and output operations use descriptors of this format.

[3]The hardware allows any accumulator to be used as a stack pointer. It is best to avoid register `0`, as sometimes a stack pointer is needed as an index register. The `ERCAL` instruction, that we will discuss later, expects that register `17` is used as the stack pointer.

```
P=17
        . . .

        MOVE    P,[IOWD PDLEN,PDLIST]
```

the assembler will treat the instruction as though we had written

```
        MOVE    17,[IOWD PDLEN,PDLIST]
```

When we define a symbolic name for an accumulator we typically use only one equal sign: the accumulator is a location, we don't usually suppress the names of locations.[4] This makes the symbolic name available to DDT for typeout during symbolic disassembly.

10.4 POP Instruction

The `POP` instruction undoes the effect of `PUSH`. Again there are two cases:

- If the accumulator contains a local stack pointer, or if the instruction is executed in section 0, the contents of the word at the top of the stack (addressed by the right half of the accumulator) are copied to the effective address. Then the stack pointer in the accumulator is decremented by subtracting `1` from both halves.[5]

  If the accumulator changes sign from non–negative to negative as a result of the subtraction, a pushdown overflow results. This condition is actually an underflow, but the hardware calls it overflow anyway. Although this warning mechanism exists, it can be used only at the expense of abandoning the warning available from the `PUSH` instruction. In most cases, the condition of stack overflow from too many pushes is considered to be most likely and most damaging. Therefore, it is far more common to see programs guard against stack overflow than stack underflow.

	With Local Stack Pointer or In Section Zero
`POP`	C(E) := C(CR(AC)); C(AC) := C(AC) –<1,,1> Stack overflow trap if C(AC) changes from non–negative to negative

- If the accumulator contains a global stack pointer and the the instruction is executed in a non–zero section, the contents of the word at the top of the stack (addressed by the accumulator) are copied to the effective address. Then the stack pointer in the accumulator is decremented by subtracting `1` from it.

[4] However, in large programs composed of multiple modules, there are sometimes alternate names for accumulators. Alternate names may be suppressed.

[5] In the KI10 and later processors, any carry from bit `18` to bit `17` is suppressed.

With Global Stack Pointer in a Non–Zero Section

`POP` C(E) := C(CR(AC)); C(AC) := C(AC) – 1

10.5 ADJSP – Adjust Stack Pointer

The `ADJSP` instruction exists in the KL10 and later processors. The `ADJSP` instruction increments or decrements a stack pointer by the immediate value `ER`, the in–section component of the effective address, taken as an 18–bit signed quantity. The precise action depends on whether the stack pointer is global or local:

- If the stack pointer is local, the value `ER,,ER` is added to the accumulator, with carry between bits `18` and `17` suppressed. If the addition of a positive `ER` causes the accumulator to change from negative to non–negative, or if the addition of a negative `ER` causes the accumulator to change sign from non–negative to negative, then pushdown overflow will be signalled.

With Local Stack Pointer or In Section Zero

`ADJSP` C(AC) := C(AC) + <ER,,ER>; Stack Overflow is possible

- If the stack pointer is global, `ER` is sign–extended and added to the accumulator.

With Global Stack Pointer in a Non–Zero Section

`ADJSP` C(AC) := C(AC) + <7777*$ER_{<18>}$,,ER>

If `ER` is positive, this instruction effectively allocates space on the stack. If `ER` is negative, this instruction deallocates space on the stack. These functions are particularly useful in the implementation of subroutines that allocate storage for local variables upon entry and deallocate the space when they exit.

10.6 Examples of PUSH and POP

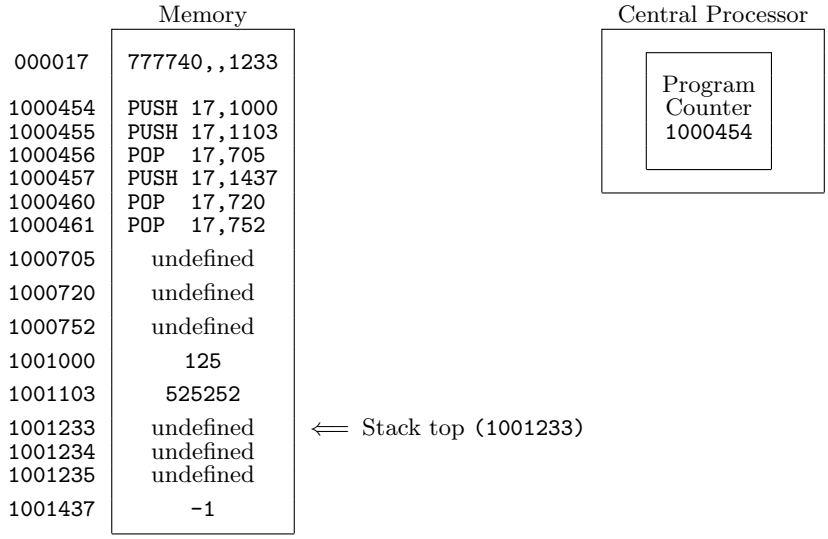
Stacks are useful in recursive subroutines and for temporary storage. Since we are not quite ready to talk about recursive subroutines, for the present we shall discuss only the use of stacks for temporary storage.

Let us examine some pushes and pops. Suppose we have initialized a local stack pointer by executing the instruction

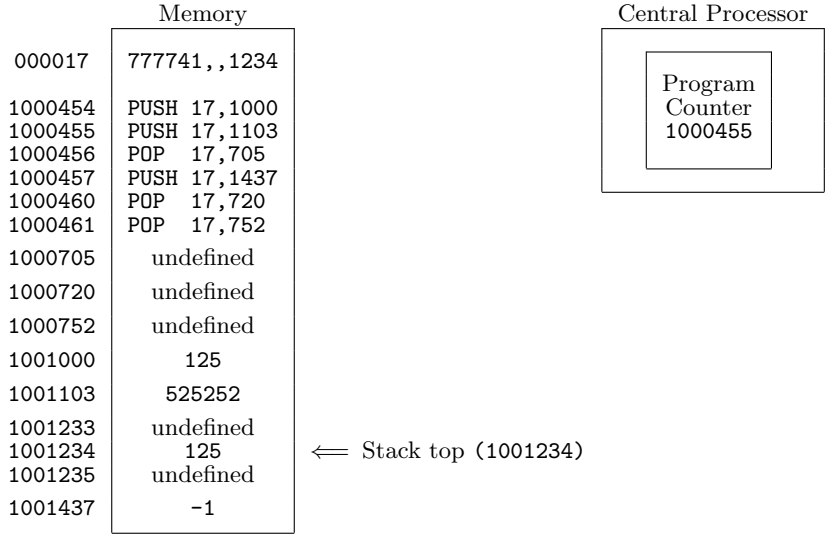
```
MOVE    17,[IOWD 40,1234]
```

Here location 1234 is the first word of an array in memory that we have reserved for our stack. Register 17, the stack pointer, will contain the value -40,,1233 or 777740,,1233.

In the figures that follow, the CPU is "frozen" between instructions, with the PC pointing at the *next* instruction to be executed. We set up memory to contain some data and a short program:

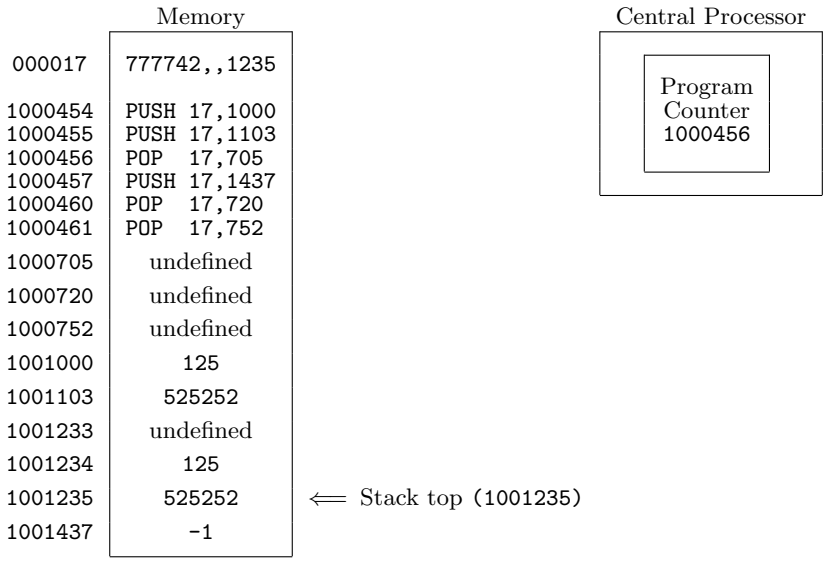


When this program is executed, the PUSH instruction in location 100454 will advance the stack pointer in register 17; register 17 now addresses location 1001234, i.e., in–sectio 1234. This is the first location in the stack. The PUSH instruction goes on to read the word at location 1001000 and copy it to the word addressed by the stack pointer. As a result of executing the PUSH in location 1000454, the CPU and memory now have the following appearance:

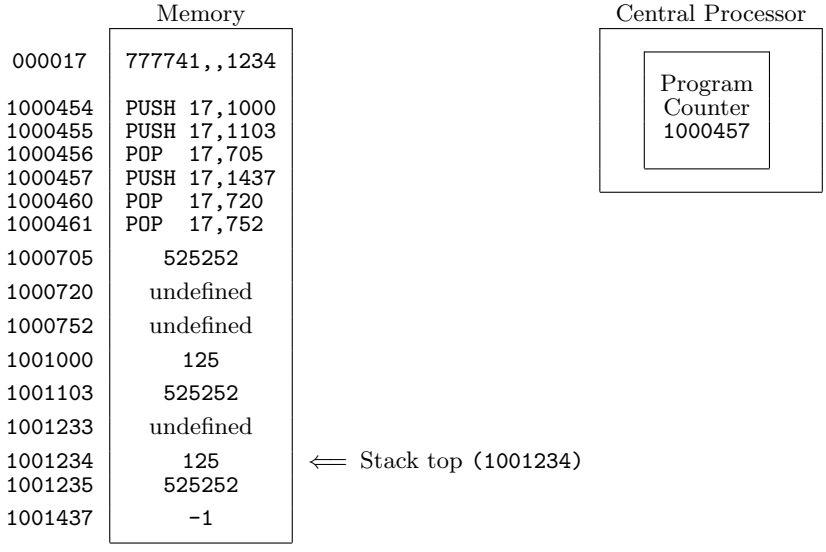


Note that the stack pointer in register 17 has been changed. Also, location 1001234, the current stack top, has been changed to be a copy of the data that is in location 1001000. The stack pointer, in general, addresses the current stack top.

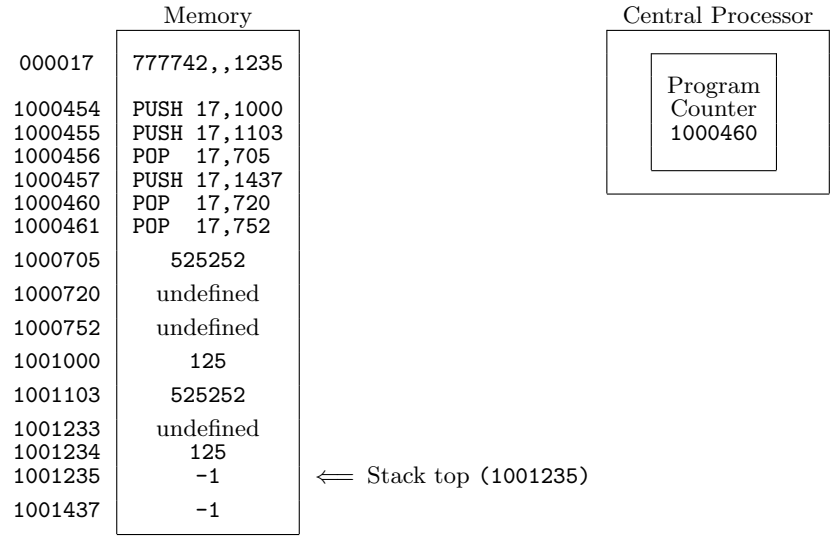
Next, the PUSH in location 1000455 is executed. The stack pointer is advanced and the data at location 1001103 is copied to the stack. The program counter is incremented to address the instruction at location 1000456:



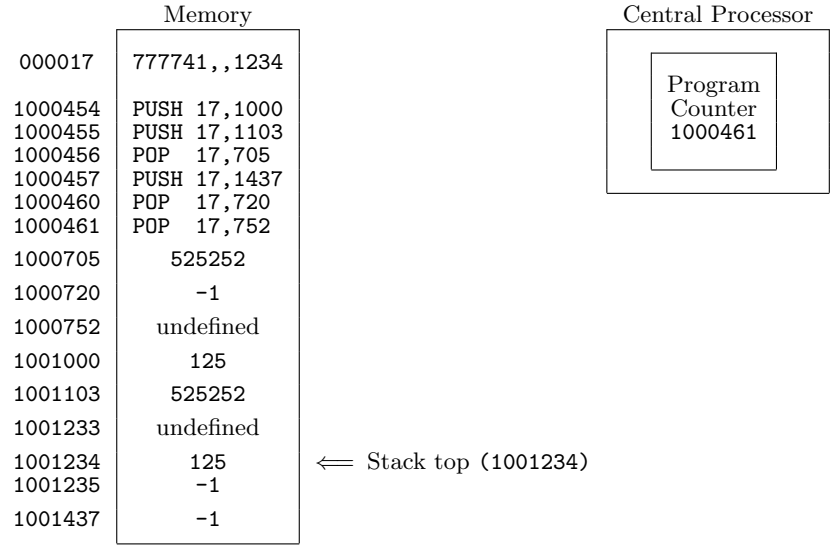
The POP instruction at location 1000456 undoes the previous PUSH by copying data from the stack to location 1000705. The stack pointer is backed up to reflect the removal of an item from the stack. Note that the popped item is still present in the memory allocated to the stack; it is considered to be removed from the stack because the stack pointer no longer includes that item.



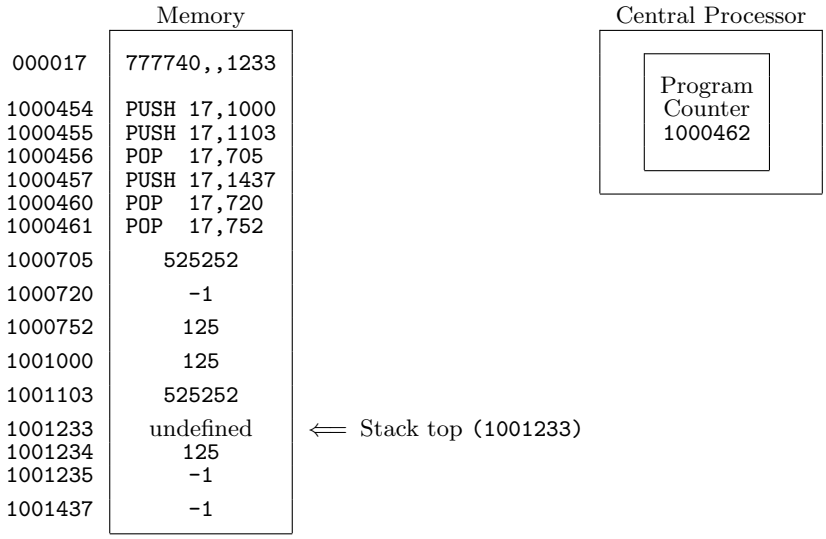
The next PUSH obliterates the stale copy of the item that we just removed from the stack:



Now, we execute the `POP` at location `1000460`. The stack unwinds:



Another `POP` succeeds in emptying the stack:



Again, the essence of a stack is that the last thing added will be the first thing removed. Also, although the operations we perform on the stack are called `PUSH` and `POP`, the data on the stack doesn't move; it is the stack pointer that changes to indicate the current stack top.

In the PDP–10, stacks grow toward higher addresses. As items are pushed, they are placed in consecutively increasing addresses.

A more usual example of the use of stacks appears below. If a block of code is expected to modify some accumulators (or other locations) that you must preserve, one neat place to save them is on the stack:

```
. . .
PUSH    P,A     ;save accumulator A on the stack
PUSH    P,COUNT ;save location COUNT on the stack
. . .
. . .           ;instructions or subroutines that modify
. . .           ;accumulator A and the memory location COUNT.
. . .
POP     P,COUNT ;restore COUNT
POP     P,A     ;restore A
. . .
```

It is important to notice that because `A` is pushed before `COUNT`, it is necessary to pop `COUNT` before popping `A`.

10.7 Example 4–A — Character Reversal

In this example we read a line and reverse it. A pushdown list is used to reverse the order of characters in the line. Since it is necessary to process the characters of the input line one at a time, we turn to the `PBIN` (Primary Byte Input) JSYS for reading the input line. The `PBIN` JSYS reads one character from the terminal and returns it in register `1`.

Let us start with a program fragment in which we use PBIN. Since each execution of the PBIN JSYS reads only one character, to read an entire line it will be necessary to repeat the PBIN in a loop. We begin with a simple fragment:

```
INLOOP: PBIN                    ;read one character
        . . .                   ;process character
        JRST    INLOOP          ;go get another character
```

One immediate problem with this loop is that there is no way to escape from it. Let us think of the appropriate way out. We want to stop reading characters from the terminal after we have read a complete line. A complete line is signified by the presence of the carriage return character. But, when a carriage return is typed, TOPS–20 sends both the carriage return and a line feed character to the program. In order to obtain the complete line, we must read until we find the line feed character.

The character that appears in register 1 is right–justified; that is, it appears in bits 29:35. The CAIE or CAIN instructions are appropriate for making character comparisons. We will augment the program fragment to include this end test. While we are performing this input processing, let us include one additional test. When the carriage return character is seen, it will be discarded.

We will adopt symbolic names for the accumulators. We assign to the symbol A the value 1. Now, when we refer to A the assembler will use the value 1. Symbolic names for accumulators are useful. Often, they have some mnemonic significance; also, symbol names appear in the cross–reference.

```
A=1                                 ;symbolic name for ac 1

INLOOP: PBIN                        ;read one character
        CAIN    A,15                ;skip unless carriage return
         JRST   INLOOP              ;discard carriage return
        CAIN    A,12                ;skip unless end of line
         JRST   INDONE              ;LF was seen. We are done now.
        . . .                       ;process character
        JRST    INLOOP

INDONE:                             ;here at end of line.
```

Earlier we said that we were going to use a pushdown list to effect the reversal of the characters. In order to use a stack, we must set aside one accumulator for the stack pointer. Also, we must allocate some storage space to the stack. Finally, we had better initialize the stack pointer before we get to the code at INLOOP. All of this is accomplished by the following augmentation of the fragment:

```
A=1                                     ;symbolic name for ac 1
P=17                                    ;symbolic for stack pointer

PDLEN==200                              ;define the stack size
PDLIST: BLOCK   PDLEN                   ;reserve space for the stack.

        . . .
        MOVE    P,[IOWD PDLEN,PDLIST]   ;initialize stack pointer
        . . .
INLOOP: PBIN                            ;read one character
        CAIN    A,15                    ;skip unless carriage return
         JRST   INLOOP                  ;discard carriage return
        CAIN    A,12                    ;skip unless end of line
         JRST   INDONE                  ;LF was seen. We are done now.
        . . .                           ;process character
        JRST    INLOOP

INDONE:                                 ;here at end of line.
```

We can now specify the nature of the input processing done in the loop at `INLOOP`. All we have to do is to add each successive character to the pushdown stack. This is accomplished by the instruction `PUSH P,A`, which adds the character in `A` to the stack. Rather than display the entire program as it has developed, we show only the interior of the loop:

```
INLOOP: PBIN                            ;read one character
        CAIN    A,15                    ;skip unless carriage return
         JRST   INLOOP                  ;discard carriage return
        CAIN    A,12                    ;skip unless end of line
         JRST   INDONE                  ;LF was seen. We are done now.
        PUSH    P,A                     ;save character on the stack
        JRST    INLOOP
```

Let us turn now to the problem of outputting the reversed line. As the stack is popped, it will yield the characters in reverse sequence. The `PBOUT` JSYS will send the character in register `1` (symbolic name `A`) to the terminal. We start our fragment of output loop by including an instruction to pop a character into register `1`, followed by a `PBOUT` JSYS:

```
OUTLOO: . . .
        POP     P,A                     ;get one character from stack
        PBOUT                           ;send it to the terminal
        JRST    OUTLOO                  ;loop.
```

Again, we must solve the problem of exiting from a loop at the right moment. There are a variety of ways we might do it. One of the very simplest ways to detect the emptying of the stack is to understand that a sequence of `PUSH` instructions followed by the precise same number of `POP` instructions will return the stack pointer to its original value. Since we know the value to which we initially set the stack pointer, we can use a `CAME` or `CAMN` instruction to test for the stack pointer having returned to that value. We want to pop characters from the stack until the stack is empty. When the stack is empty, the stack pointer will have been restored to its original value.

There are two ways that we can write this loop. Either we can test the stack pointer before we pop

it, or we can test it afterwards. Compare these methods:

Top Test

```
OUTLOO: CAMN    P,[IOWD PDLEN,PDLIST]
         JRST   OUTFIN
        POP     P,A
        PBOUT
        JRST    OUTLOO

OUTFIN:                 ;here when done
```

Bottom Test

```
OUTLOO: POP     P,A
        PBOUT
        CAME    P,[IOWD PDLEN,PDLIST]
         JRST   OUTLOO
        .  .  .         ;here when done
```

These two loops implement control techniques known as *top test* and *bottom test*, so named because of the position of the loop exit test. Note that in this case the bottom test is accomplished in fewer instructions. However, bottom test loops are not satisfactory for all applications. In this case, for example, if the line that is input is empty, i.e., contains no characters apart from a carriage return and line feed, then the bottom test loop will be faulty. This is because the program will arrive at `OUTLOO` with the stack already empty. By popping the stack before testing for empty, we obtain from the stack a word that we never pushed. Moreover, since we are testing for the equality of the stack pointer with its initial value, that condition will never happen.[6] The program will loop, transmitting rubbish to the terminal.

In the absence of further adornment, the bottom test loop could not be used in this program. However, we can add some code in front of the output loop to ensure proper operation of the program. At the same time we can expand the definition of our program to include halting when the input line is empty.

```
        CAMN    P,[IOWD PDLEN,PDLIST]   ;is the stack empty?
        JRST    STOP                    ;empty line, stop running.
OUTLOO: POP     P,A                     ;get one character from stack
        PBOUT                           ;send it to the terminal
        CAME    P,[IOWD PDLEN,PDLIST]   ;is the stack empty now?
        JRST    OUTLOO                  ;not yet.  Loop again.
        . . .                           ;here when done
```

The fragment displayed above is quite adequate for our output loop. The operational behavior of this fragment is superior to that of the top–test, even though, statically, this is one instruction longer than the top–test loop. This fragment provides the desirable feature of stopping the program when an empty line is read.

During our input processing, we removed the characters carriage return and line feed from the original input line. We must put them back into the line after outputting the reversed sequence of characters. We could use either two `PBOUT` calls or one `PSOUT`. The single `PSOUT` is somewhat more convenient:

[6] After some 262,144 characters have been typed, the program will escape from this loop, unless some other error supervenes.

```
        CAMN    P,[IOWD PDLEN,PDLIST]   ;is the stack empty?
        JRST    STOP                    ;empty line, stop running.
OUTLOO: POP     P,A                     ;get one character from stack
        PBOUT                           ;send it to the terminal
        CAME    P,[IOWD PDLEN,PDLIST]   ;is the stack empty now?
        JRST    OUTLOO                  ;not yet.  Loop again.
        HRROI   A,[ASCIZ/
/]
        PSOUT                           ;send CR and LF
```

We add a few more embellishments, such as a starting address, some prompts and headings, the instructions at STOP, and the necessary pseudo–ops. The resulting program appears below:

```
        TITLE   REVERSE - Example 4-A
        SEARCH  MONSYM

Comment $
        Program to reverse the characters on each line of input.
        Program will stop when an empty line is input.

        This program demonstrates the last-in, first-out property
        of push-down stacks.  Also, it shows the PBIN and PBOUT
        JSYS calls.
$


A=1                                     ;symbolic name for ac 1
P=17                                    ;symbolic for stack pointer

PDLEN==200                              ;define the stack size
.PSECT DATA,1001000
PDLIST: BLOCK   PDLEN                   ;reserve space for the stack.
.ENDPS

.PSECT CODE/RONLY,1002000

START:  RESET
        MOVE    P,[IOWD PDLEN,PDLIST]   ;initialize stack pointer
GETLIN: HRROI   A,[ASCIZ/Please type a line: /]
        PSOUT
INLOOP: PBIN                            ;read one character
        CAIN    A,15                    ;skip unless carriage return
         JRST   INLOOP                  ;discard carriage return
        CAIN    A,12                    ;skip unless end of line
         JRST   INDONE                  ;LF was seen. We are done now.
        PUSH    P,A                     ;store character on the stack
        JRST    INLOOP
```

```
;Here at the end of the input line
INDONE: CAMN    P,[IOWD PDLEN,PDLIST]   ;is the stack empty?
         JRST   STOP                    ;empty line, stop running.
        HRROI   A,[ASCIZ/The reversed line:  /]
        PSOUT
OUTLOO: POP     P,A                     ;get one character from stack
        PBOUT                           ;send it to the terminal
        CAME    P,[IOWD PDLEN,PDLIST]   ;is the stack empty now?
         JRST   OUTLOO                  ;not yet.  Loop again.
        HRROI   A,[ASCIZ/
/]
        PSOUT                           ;send CR and LF.
        JRST    GETLIN                  ;repeat

STOP:   HALTF                           ;stop at blank line
        JRST    STOP                    ;stay stopped

        END     START
```

10.7.1 Summary of Example 4–A

The starting code executes a RESET JSYS and initializes the pushdown stack. Register P is used as a stack pointer. As we discussed in the descriptions of PUSH and POP, the stack pointer is initialized using an IOWD pseudo–op. IOWD forms a word that contains -PDLEN in the left half and PDLIST-1 in the right half; this is a local stack pointer. The negative count in the left is used as a control count to indicate when the stack has overflowed the area allocated to it. The address PDLIST-1 in the right half of P points to one word before the first word of the stack. Recall that the first thing that PUSH does is to add one to both halves of P to determine where to store the data that is being pushed. When PDLIST-1 is incremented by one, it will address precisely the first word of the stack area.

The code at GETLIN prompts for a line of input. As characters are read at INLOOP, they are pushed onto the stack. One characteristic of a stack is that the last thing that was pushed is the first thing that will be popped. It is this feature of the stack that accomplishes the reversal of characters.

The loop at INLOOP reads the input line character by character. The PBIN JSYS reads one character from the terminal; the character appears in register 1, which we call A. INLOOP pushes most characters onto the stack; it avoids pushing either the carriage return or line feed characters that terminate the input line. When the line feed is seen, the sequence:

```
CAIN    A,12
 JRST   INDONE
```

will branch to INDONE to leave the input loop.

At INDONE the program tests to see if the stack is empty. An empty stack is indicated by the stack pointer (register P) being equal to its original value. Since the original value was a full–word quantity, it is not possible to use a CAIN instruction here; so CAMN is used instead. If the stack is empty, the sequence at INDONE jumps to STOP; the program will halt.

At OUTLOO, we know the stack is not yet empty. It is safe to POP one character from the stack and send it to the terminal via PBOUT. We have reduced the number of characters on the stack. By means

of the CAME instruction, the stack pointer is tested to see if it now indicates an empty stack. The CAME will skip when the stack is empty. If the stack is not yet empty, the CAME will not skip, and the instruction JRST OUTLOO will be executed to take us around the output loop once more. The program loops through the code sequence at OUTLOO until the stack becomes empty.

When the stack is empty, the CAME instruction will skip; this allows the program to escape from the output loop. After we finish with OUTLOO, the program prints a carriage return and line feed and then jumps to GETLIN to read another line.

It should be noted that these examples are imperfect in some respects. A carefully written version of this program would guard against a line that was too long for the stack size that is given. To show a perfect program, designed to defend against all manner of erratic input, would distract us from the main purpose. We want to show examples of how the instructions fit together to form programs; the extra complication of dealing with error checking would disrupt the presentation of examples.

10.7.2 PBIN

The PBIN JSYS reads a single character from the terminal into register 1. PBIN signifies *Primary Byte INput*, where "byte" means "character" and "primary" usually refers to the terminal that controls the program.

For highly interactive applications, such as occur in some display editors, PBIN may be essential. However, in this application PBIN is not the best way to read from the terminal; we have used PBIN because we are not yet ready to tackle the processing of an entire line of characters.

A programmer should be aware of the problems as well as the benefits of PBIN. The principal benefits of PBIN are

- Every character that is typed goes to the program immediately. This is important in some highly interactive applications.
- No buffer area or buffer management is needed. The characters appear one by one, each time the PBIN function is executed. It is this feature of PBIN that we find attractive in this program.

The problems with PBIN are

- PBIN does no line editing. When RDTTY is used, TOPS–20 provides various facilities for editing what is being typed. For example, the RUBOUT and CTRL/U characters can be used to correct mistakes. In the case of PBIN, the program sees characters as soon as they are typed. Unless the program itself makes provisions for correcting typing errors, the characters RUBOUT, etc. are ineffective for making changes.[7] This problem with PBIN can be overcome by extensive additional programming, such as that found within the RDTTY JSYS.
- When a program calls the operating system to perform a function such as RDTTY, the system stops running the program until an entire line is ready to be processed. While your program is inactive, TOPS–20 allows the other users of the system to run their programs. In RDTTY, TOPS–20 collects input characters until a carriage return is typed. When TOPS–20 sees the carriage return it reactivates (*wakes up*) your program; an entire line is available for your program to process. In contrast, when the PBIN JSYS is used, TOPS–20 reactivates your

[7] Your author was unable to type the following without error: "Doc, note I dissent: a fast never prevents a fatness. I diet on cod."

program each time that a key is typed. Reactivating a program is an expensive operation (compared to simply running it), so the fewer wake–ups that a program requires, the more efficiently the system can run it.

- A third reason to prefer `RDTTY` to `PBIN` is that every system call is relatively expensive, regardless of how little useful work is performed. `RDTTY` moves an entire string from the system to your memory space in one system call. When using `PBIN`, the program must repeat the system call many times to process each line. Each system call includes some number of *overhead* operations that are unavoidable, but that do not contribute to the performance of the desired function. The overhead activity is proportional to the number of system calls that your program executes, and may exceed the amount of useful work done by each JSYS call. Part of the overhead in each JSYS call is the operation called a *context switch*, in which the computer changes from running your program to running the operating system, and then changes back to running your program. Fewer system calls and fewer context switches allow TOPS–20 to run a program more efficiently.

We have used `PBIN` in this example because we are not ready to deal with a string on a character–by–character basis. In example 4–B, after we have demonstrated the byte instructions, we shall show a better way to perform this function.

10.7.3 PBOUT

The `PBOUT`, *Primary Byte OUTput*, JSYS sends one character from register `1` to the "primary output" device, which is usually the terminal.

Some of the efficiency considerations mentioned in the context of the `PBIN` JSYS apply to `PBOUT` as well. In particular, where practicable, the overhead of repetitive JSYS calls can be avoided by building an entire output string and sending it to the terminal with `PSOUT`. This also will be demonstrated after we learn the byte instructions.

Chapter 11

Byte Instructions

In the PDP–10 a *byte* is some number of contiguous bits within one word. Pictorially, a byte within a word is depicted in Figure 11.1. The byte includes consecutive bits starting at bit number 36_{10}`-P-S` through bit number 35_{10}`-P`. The byte includes `S` bits and is located `P` bits to the left of the right end of the word.

The byte instructions allow the programmer to pack and unpack bytes of any length anywhere within a word. Movement of a byte is always between AC and a memory location: a deposit instruction copies a byte of the indicated size from the right–end of AC and inserts it in the memory location; a load instruction unpacks a byte from a memory location and copies the data, right–justified, into AC.

11.1 Byte Pointers

The byte instructions calculate `E` in the standard way. However, `C(E)` is interpreted as a *byte pointer* which is used in turn to locate the byte or the place that will receive the byte.

A byte pointer is a word (or two words) that describes a byte. There are three parts to the description of a byte: the word (i.e., the address) in which the byte occurs, the position of the byte within the word, and the length of the byte.

There are three formats for byte pointers.

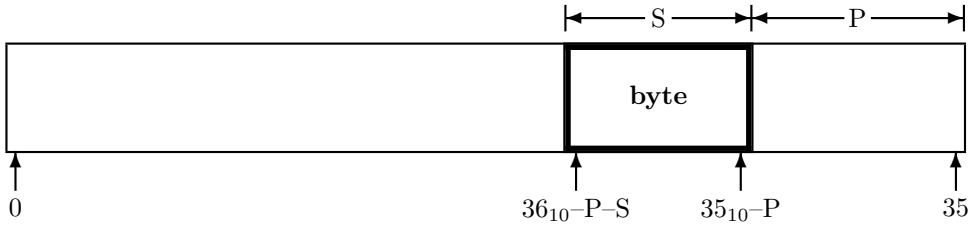


Figure 11.1: A Byte in a Word

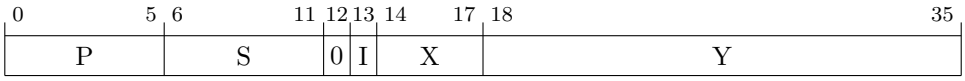


Figure 11.2: One-Word Local Byte Pointer

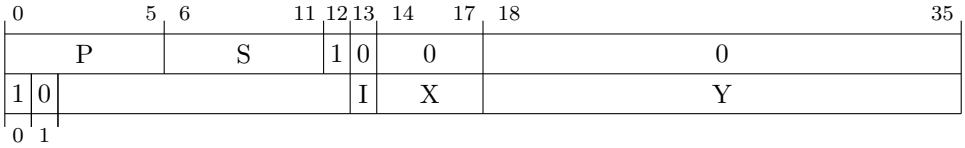


Figure 11.3: Two-Word Byte Pointer (Local Address Word)

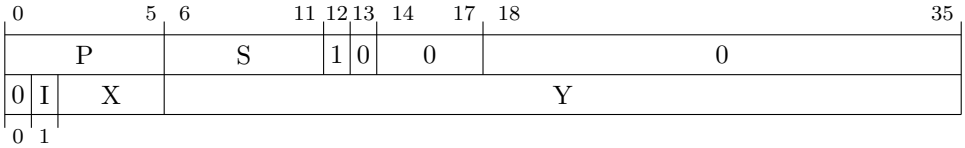


Figure 11.4: Two-Word Byte Pointer (Global Address Word)

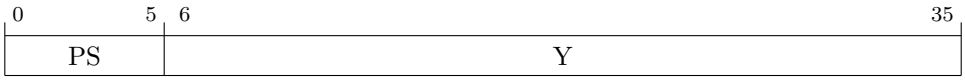


Figure 11.5: One Word Global Byte Pointer

- The traditional format, now called a *one–word local byte pointer*,
- the *two–word byte pointer*, and
- the *one–word global byte pointer*.

The latter two formats are supported in the KL10 and later computers, with the two–word byte pointer being valid only in non–zero sections.

11.1.1 One–Word Local Byte Pointer

We begin with the traditional scheme, the one–word local byte pointer, which has the format depicted in Figure 11.2.

Several fields are present in a one–word local byte pointer:

- The `P` field specifies the byte position within the word. The contents of the `P` field is the count of bits to the right of the desired byte, (i.e, 35_{10} minus the bit number of the rightmost bit in

the byte). The P field may take on values from 0 to 44_8, i.e, to 36_{10}. (Values of P above 44_8 are reserved for use as *one–word global byte pointers*.)

- The S field specifies the byte size in bits. Sizes 0 through 36_{10} are valid.
- Bit 12 must be zero. (In non–zero sections, when bit 12 is 1 it signifies a two–word byte pointer. Although in section 0 bit 12 is ignored by the hardware, for compatibility with the two–word format, programmers should set bit 12 to zero even in section 0.)
- The I, X, and Y fields are the same as in an instruction. These fields are interpreted to provide the effective address of the word in which the byte is contained. Precisely the same rules of effective address calculation in instructions apply to these fields. To initialize the effective address calculation, the byte pointer word is considered to be a local address word in the address section from which it was read.

The word "local" in the name of this byte pointer is slightly misleading. By means of indexed or indirect addressing, this format could access a byte in a section other than the one from which the byte pointer was read.

11.1.2 Two–Word Byte Pointer

A two–word byte pointer may be used only in non–zero sections. The two–word byte pointer is two consecutive words (at E and E+1). The word in location E contains P and S fields in the same places as for one–word local byte pointer. Bit 12 is 1. Bits 13:35 of the word at location E should be zero. The 1 in bit 12 signifies that this byte pointer is in the two–word format. The word at location E+1 contains an address word, either a local format indirect word or a global format indirect word. Effective address computation occurs using this word; the resulting effective address locates the word containing the byte. Two–word byte pointers are depicited in Figure 11.3 and Figure 11.4.

When a two–word byte pointer contains a local format address word, it may generate a local address; when it contains a global format address word, it generates a global address.

11.1.3 One–Word Global Byte Pointer

One–word global byte pointers provide a convenient shorthand by which to express bytes in words that are in address sections remote from the byte pointer. However, only a limited number of sizes and positions are available in this format; these are commonly used sizes and positions.

A one–word global byte pointer is identified by a value in the range (octal) 45 – 76 in bits 0:5 of a byte pointer word. If a value in this range is seen, the value is decoded to provide specific values of P and S; data in bits 6:35 supply the global address of the word containing the byte.

The usual inability of a section 0 program to reference extended addressing notwithstanding, a one–word global byte pointer supplies a 30–bit global address regardless of the address section from which it is read.

The format of the one–word global byte pointer is shown in Figure 11.5. Both the position and the size are encoded in bits 0:5, identified as PS in that figure. The decoding of PS is described in Figure 11.6.

Only popular bytes sizes and their popular (zero–alignment) positions are represented in one–word global byte pointers.[1] Neither indexing nor indirection is provided with this format. For other

[1] Alignment will be explained in Section 11.2.8.

PS	Size	Position	PS	Size	Position	PS	Size	Position
45	6	␣	56	8	15	67	9	␣
46	6	5	57	8	23	70	9	8
47	6	11	60	8	31	71	9	17
50	6	17	61	7	␣	72	9	26
51	6	23	62	7	6	73	9	35
52	6	29	63	7	13	74	18	␣
53	6	35	64	7	20	75	18	17
54	8	␣	65	7	27	76	18	35
55	8	7	66	7	34	77	Illegal	

In this table the PS values are octal; the Size and Position values are decimal. The position value listed denotes the bit number of the rightmost bit of the byte: this is compatible with the representation of position in the `POINT` pseudo–op. It is relates to the `P` field of a byte pointer by the relation `P`$= 35_{10} -$ Position. The position value "␣" denotes the imaginary byte whose rightmost bit is at bit -1, that is, to the left of the leftmost real byte in a word; such a pointer must be incremented before it can be used to address a byte.

Figure 11.6: One–Word Global Byte Pointers: Decode of PS Field

combinations of size, position, and address modes a two–word byte pointer must be used.

The one–word format global byte pointer may be used in a variety of JSYS calls that accept byte pointers as arguments. Be aware, however, that some JSYS calls that accept either a byte pointer or some other data item allow only a limited subset of the one–word global byte pointers. The following table lists JSYS calls that accept the one–word global byte pointers for 7–bit bytes but which do not accept the other byte sizes.

JSYS Name	Location(s) of restricted argument
`CACCT`	1
`LOGIN`	3
`CRJOB`	`.CJACT`
`RCDIR`	2
`SACTF`	2
`CHKAC`	`.CKAUD .CKALD .CKACD`
`ACCES`	`.ACDIR`
`STPPN`	1

11.2 Manipulating Bytes and Byte Pointers

We will discuss six instructions that manipulate byte pointers and bytes and one pseduo–op.[2]

[2]There are additional instructions in the KL10 extended instruction set that manipulate strings that are described by byte pointers. We will not discuss these, but you may consult [SYSREF] for details.

11.2.1 LDB – Load Byte

The contents of the effective address of the `LDB` instruction is interpreted as a byte pointer. The byte described there is loaded, right adjusted, into the accumulator. The other bits in the accumulator are set to zero.[3]

`LDB`	C(AC) := the byte described by C(E)

For example:

```
LDB     5,[270400,,40]
```

loads register `5` with a `4`–bit byte composed of bits `9:12` of the word at location `40`. Bits `9:12` are the accumulator field, so this instruction copies the AC field from location `40` to accumulator `5`. The leftmost `32` bits in register `5` are set to zero.

11.2.2 DPB – Deposit Byte

The contents of the effective address of the `DPB` instruction is interpreted as a byte pointer. The rightmost `S` bits of the accumulator are deposited into the byte specified by the byte pointer at the effective address. The accumulator and the other bits of the word into which the byte is deposited remain unchanged.

`DPB`	The byte described by C(E) := bits 36_{10}–S:35_{10} of C(AC)

11.2.3 IBP – Increment Byte Pointer

To *increment* a byte pointer means to change the pointer to refer to the next byte in sequence following the byte that it presently points to. The next byte is the same size as the current byte; it is immediately to the right of the current byte in this word, if it fits. Otherwise, the next byte is the first `S` bits of the word at the next address.[4] In brief, bytes advance from left to right within a word and from lower addresses to higher addresses.

Figure 11.7 depicts three consecutive bytes. When a byte pointer that points to the byte labeled "A" is incremented, the resulting pointer will point to "B". The byte "B" is adjacent to and immediately to the right of "A". When a pointer to "B" is incremented, then, because there are fewer than `S` bits remainging at the right end of the word, a new pointer will be constructed that points to "C". The byte "C" is in the word immediately following the word that contains "B". Byte "C" includes bits `0` through `S-1` of that word. Note that a byte never crosses a word boundary. When fewer than `S` bits remain at the right end of a word, those bits are ignored.

In detail, a one–word local byte pointer is incremented as follows: a new position field is computed by subtracting `S` from `P`, i.e., `P-S`. If the result is non–negative, it will be stored as the new value of

[3] In the instruction descriptions that follow in this chapter, "`C(E)`" may mean the doubleword `C(E) C(E+1)`.

[4] This description assumes that the byte pointer does not call for indirect addressing. Generally, byte pointers that are incremented do not use indirect addresses.

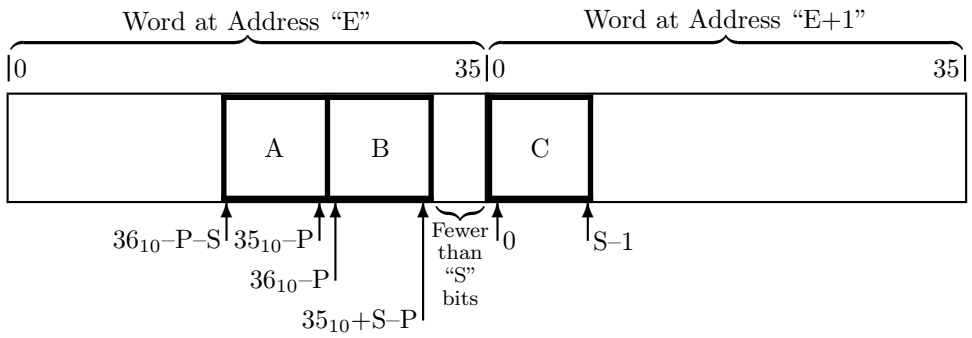


Figure 11.7: Consecutive Bytes

the P field. If the result of the subtraction is negative, no more bytes of size S will fit in the current word; consequently, the Y field of the byte pointer is incremented (to point to the next word) and the new P field is computed from P := 36_{10}-S. The new P field selects the leftmost byte of size S in the word addressed by the new value of the Y field.

When there is no room for another byte in a word, the Y field of the one word local byte pointer is incremented[5]. Because the Y field of a pointer may be changed in this way, the programmer should generally avoid indirect addressing in any byte pointer that is incremented.

A two–word byte pointer is incremented by processing P and S the same way. However, when the address must be incremented, details depend on the format of the address word in E+1. A local format address word has an 18–bit Y field; to this, the computer adds 1 and puts the 18–bit sum into bits 18:35 of the word at E+1. A global format address word has a 30–bit Y field; this is incremented and the sum is stored in bits 6:35 of E+1.

A one word global is "incremented" by table lookup that supplies the new value of PS and a flag based on the old PS field. The new value of PS replaces the old; if the flag is set, the address in bits 6:35 is incremented. For example, if the old PS were 61 the new value would be 62 and no flag; incrementing when PS contains 66 results in 62 with the flag set requesting that the address be incremented.

There are three instructions that increment byte pointers in the manner that we have described. The first of these is the IBP instruction. Although we mention IBP first, generally the two instructions that follow, ILDB and IDPB, are more often used.

The IBP, Increment Byte Pointer, instruction increments a byte pointer as we have described. The accumulator field must be zero in the IBP instruction; a non–zero AC field specifies the ADJBP instruction, as we shall explain below. The IBP instruction will fetch the contents of the effective address. That word (or doubleword) is interpreted as a byte pointer; it is incremented. The updated byte pointer is stored at the effective address, replacing the original byte pointer.

[5]On the old processors, the PDP–6 and the KA10, when Y contains 777777, incrementing the address field produces a carry into the X field. In the case of ILDB and IDPB this causes unpredictable results. No such problem exists on the KI10 or newer processors.

`IBP 0,`	C(E) :=	Incremented byte pointer from original C(E). The AC field of the IBP instruction must be zero.

11.2.4 ILDB – Increment Pointer and Load Byte

Increment the byte pointer contained at the effective address. Then perform a `LDB` function using the updated byte pointer.

`ILDB`	C(E) :=	Incremented byte pointer from original C(E).
	C(AC) :=	Byte described by the new C(E).

11.2.5 IDPB – Increment Pointer and Deposit Byte

Increment the byte pointer contained at the effective address. Then perform a `DPB` function using the updated byte pointer.

`IDPB`	C(E) :=	Incremented byte pointer from original C(E).
	Byte described by the new C(E) :=	Bits 36_{10}–S:35_{10} of C(AC).

The two instructions `ILDB` and `IDPB` are immensely useful in handling character strings and other sequences of data objects.

One of the characteristics of these two instructions is that they advance the byte pointer before loading or depositing a byte. On the whole, this is more convenient than the alternative, but it does provide some peculiarity when initializing for string processing. We shall see that the `POINT` pseudo–op neatly allows us to handle this problem.

11.2.6 POINT Pseudo–operator

For convenience, the assembler has a pseudo–op for creating one–word local byte pointers. The `POINT` pseudo–op has three parameters, the size, the address, and the position. In the `POINT` pseudo–op, the position argument specifies the bit number of the *rightmost* bit in the byte. Note that the position argument to the `POINT` pseudo–op is not the same as the `P` field of a byte pointer. In the `POINT` pseudo–op, both the size and position fields are interpreted as *decimal* (rather than as octal) numbers.

Some examples of the `POINT` pseudo–op follow. Remember, the size and position fields of this pseudo–op are interpreted as *decimal* numbers.

`POINT 7,1000,6`	`350700,,1000` Byte is 7 bits from 0 to 6
`POINT 9,3214,26`	`111100,,3214` Byte is 9 bits, 18 to 26
`POINT 1,67(3),4`	`370103,,67` Selects bit 4

Since the `ILDB` and `IDPB` functions increment a byte pointer *before* performing the load or store operation, it is sometimes necessary to resort to a subterfuge for initializing byte pointers. For example, suppose that the word at (in–section) address `1000` contains several `7`–bit bytes starting at the left end of the word. It would be desirable to use the simplest possible loop to read this sequence of bytes. An `ILDB` instruction in a loop would be satisfactory if only we could determine how to make a byte pointer which, when incremented, addresses the first byte in the word at `1000`.

The byte that we want to process first is described by the pointer `350700,,1000`; this pointer could also be specified as `POINT 7,1000,6`. We also want the first instruction that processes this string to be an `ILDB`. In order to make the first `ILDB` pick up the first byte, we must back up the byte pointer so that when it is incremented it will point to the first byte in this word. Recall that a byte pointer is incremented by subtracting `S` from `P`. If we add the byte size, `7`, to the position, 35_{10}, we get `440700,,1000`. This byte pointer points to the non–existent bits `-7:-1` of word `1000`. It doesn't matter that this byte doesn't exist, so long as we do not try to load or store using this pointer. If the first thing we do is increment the pointer prior to a load or deposit, then everything will be satisfactory. The initialization of a byte pointer to point to a non–existent byte is analogous to the initialization of a stack pointer where we made it point to a word that is outside of the actual area allocated to the stack.

The `POINT` pseudo–op can be used to build a byte pointer that points at the non–existent byte to the left of the first byte in a word. To use `POINT` in this way, simply omit the position field. The byte pointer that is built will have the position field set to octal `44`.

`POINT 7,1000`	`440700,,1000` The non–existent 7–bit byte to the left of all bits at location 1000.

The `POINT` pseudo operator is very commonly used with ASCII strings. The byte instructions are quite useful for string processing.

```
        MOVE    B,[POINT 7,[ASCIZ/This is a string
/]]
LOOP:   ILDB    A,B             ;get a character from the string
        JUMPE   A,LOOPX         ;if null, exit from loop
        . . .                   ;process character
        JRST    LOOP            ;continue - process all characters

LOOPX:  . . .                   ;finished processing.
```

Usually this loop structure will be implemented in a subroutine.

The POINT pseudo–op cannot, by itself, make a two–word byte pointer. To make a two word byte pointer, you may resort to something like the following:

```
<POINT 7,0,13>+1B12     ;Set P and S fields and bit 12.
 local or global format indirect word
```

The pointed brackets "<" and ">" surrounding the POINT pseudo–op cause MACRO to evaluate it as a 36–bit value before attempting the addition that sets bit 12.

11.2.7 Symbols for One Word Global Byte Pointers

The POINT pseudo–op is not used to form one–word global byte pointers. Instead, the MACSYM universal file defines symbols of the form .Psspp where "ss" is the byte size in decimal, expressed as two digits, and "pp" is the byte position, expressed as the two–digit decimal bit number of the right–most bit of the byte (or omitted to mean the imaginary byte at the left). Thus, .P0734 is defined as 66B5 and .P07 is 61B5. MACSYM defines symbols for all of the combinations of byte size and position implemented in the one word global byte pointers.[6]

Use the logical OR operation (in MACRO or at runtime) to include one of the .Psspp symbols with a 30–bit address to form a one–word global byte pointer.

11.2.8 ADJBP – Adjust Byte Pointer

The ADJBP instruction has the same operation code as the IBP instruction; ADJBP is distinguished from IBP by a non–zero accumulator field.[7] In the ADJBP instruction the accumulator contains an adjustment count, either positive or negative. In the execution of this instruction, the computer will fetch the byte pointer at the effective address, adjust it forward or backward by number of bytes specified in the accumulator, and then place the adjusted byte pointer in the accumulator. The original byte pointer is unchanged. This differs from the way that the IBP instruction stores its result.[8]

ADJBP	AC,	C(AC) :=	The byte pointer from C(E) adjusted by the original contents of AC. The AC field of the ADJBP instruction must be non–zero.

ADJBP is not quite the same as iterating IBP. The difference lies in the fact that ADJBP preserves the *byte alignment* across word boundaries. The term byte alignment refers to the bit number of the left–most bit in the left–most byte of a word, as defined by the P and S fields. Numerically, the byte alignment is $(36_{10}$-P) modulo S.

Ordinarily, strings are packed with zero alignment because the IBP, ILDB and IDPB instructions all force the alignment to zero when a byte pointer is incremented and crosses a word boundary. ADJBP, however, preserves byte alignment when crossing a word boundary.

When given a reasonable byte pointer, ADJBP will return a byte pointer that describes a complete byte within a word. For example, if the accumulator contains zero and the byte pointer has a P field

[6] The MACSYM package can be invoked by including "MACSYM" as an additional parameter in the argument list to SEARCH.

[7] ADJBP does not exist in the KI10 and earlier processors.

[8] If the byte pointer in E is a two–word byte pointer, this instruction will read both E and E+1 and it will store a two–word result in AC and AC+1.

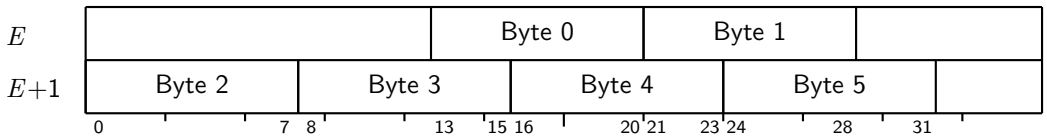


Figure 11.8: 8–bit Byte Sequence in Memory (IBP Action)

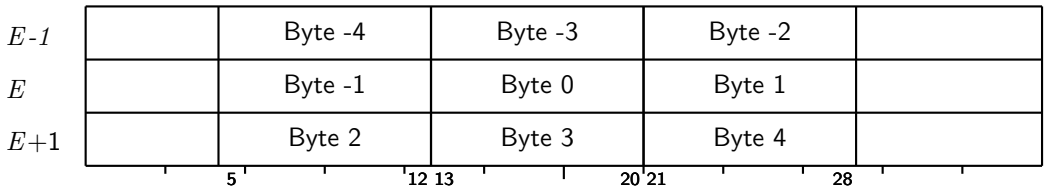


Figure 11.9: 8–bit Byte Array in Memory (ADJBP Action)

of octal `44` (i.e., a byte pointer to the non–existent byte to the left of a word) then `ADJBP` will return a byte pointer to a real byte that is contained within the previous word.

`ADJBP` computes the number of bytes that will fit in a word by the formula:

$$((36_{10}\texttt{-P})\ \texttt{DIV}\ \texttt{S})\ \texttt{+}\ (\texttt{P}\ \texttt{DIV}\ \texttt{S})$$

In this formula, `DIV` means divide and truncate the quotient to an integer. The first portion of this expression is how many bytes, including the specified byte itself, will fit at and to the left of the byte. The second portion is how many bytes will fit to the right of the specified byte.[9]

If the number of bytes per word is zero, the computer sets the divide check flag (see Section 13, page 161) and the instruction avoids changing the accumulator or memory.[10] Otherwise, the adjustment count found in the the accumulator is divided by the specified number of bytes per word. The quotient is added to the effective address of the byte pointer; the remainder specifies an adjustment to make to the `P` field of the resulting byte pointer. (In the process of adjusting `P`, the effective address of the byte pointer may change by one.)

To give an example of difference between the effect of `ADJSP` and `IBP`, consider the following scenario. The byte pointer `POINT 8,E,20` describes an 8–bit byte that occupies bits 13:20 in the word at `E`. This byte has alignment 5. When this byte pointer is incremented by `IBP` (or by `ILDB` or `IDPB`) it becomes `POINT 8,E,28`, i.e., a pointer to bits 21:28 in `E`; this byte also has alignment 5. Increment it again and it becomes `POINT 8,E+1,7`, the pointer that describes bits 0:7 in the word at `E+1`; this byte (and all subsequent ones) have alignment 0 because `IBP` sets the alignment to zero when a word boundary is crossed. These bytes are depicted in Figure 11.8.

In contrast, starting with the same byte pointer but using `ADJBP` to change it, by adjusting +1 we get the same result as `IBP` did. However, adjusting the original by +2 results in `POINT 8,E+1,12`, that is, a byte pointer to bits 5:12 of the next word; this pointer maintains alignment 5 even though

[9] If `S` is zero, the `ADJBP` instruction simply copies `C(E)` to the accumulator.

[10] This can happen only when the given pointer does not describe a real byte, e.g., `POINT 19,0,17`.

it has crossed a word boundary. The adjustment can be either positive or negative as depicted in Figure 11.9. Note that at alignment 5, only three bytes fit in each word.

11.3 Example 4–B — Character Reversal with Byte Instructions

In this version of the reverse program we read an entire line at a time via RDTTY, avoiding the use of the PBIN JSYS. We use the byte instructions to read individual characters from the input line buffer; byte instructions are also used to build an output buffer containing the reversed line. Sending the entire output buffer to the terminal via PSOUT is much more efficient than the repetitions of PBOUT that we resorted to in example 4–A.

We begin by re–examining the input loop from example 4–A:

```
GETLIN: HRROI   A,[ASCIZ/Please type a line: /]
        PSOUT
INLOOP: PBIN                            ;read one character
        CAIN    A,15                    ;skip unless carriage return
         JRST   INLOOP                  ;discard carriage return
        CAIN    A,12                    ;skip unless end of line
         JRST   INDONE                  ;LF was seen. We are done now.
        PUSH    P,A                     ;store character on the stack
        JRST    INLOOP
```

The PBIN JSYS must be replaced by a call to the RDTTY function. We define symbolic names for accumulators 2 and 3. A storage area called IBUFR is reserved for the input line.

```
A=1
B=2
C=3
P=17

BUFLEN==40
IBUFR:  BLOCK   BUFLEN

        . . .

GETLIN: HRROI   A,[ASCIZ/Please type a line: /]
        MOVE    C,A                     ;save the reprompt pointer
        PSOUT                           ;prompt for a line
        HRROI   A,IBUFR                 ;pointer to input buffer
        MOVEI   B,BUFLEN*5-1            ;length of buffer, no flags
        RDTTY
        ERJMP   ERROR
        . . .
```

```
INLOOP: . . .                           ;get a character
        CAIN    A,15                    ;skip unless carriage return
         JRST   INLOOP                  ;discard carriage return
        CAIN    A,12                    ;skip unless end of line
         JRST   INDONE                  ;LF was seen. We are done now.
        PUSH    P,A                     ;store character on the stack
        JRST    INLOOP
```

As in example 3, we have supplied the necessary arguments to RDTTY in accumulators 1, 2, and 3 (herein named A, B, and C, respectively).

We still need a loop to process the input characters; note, however, that the RDTTY has been removed from that loop. The one call to RDTTY will supply an entire line of characters. The processing of the input line is very simple. The basic structure of INLOOP has been preserved. In example 4–A, the PBIN JSYS caused a character to appear in register A. We must perform some instruction that has the same effect. Our input line appears as a sequence of characters starting in the word at IBUFR. It should come as no surprise that we employ the byte instructions for this character processing.

In order to use the byte instructions, we must have a memory location that contains a byte pointer. We must initialize that location to point before the first byte; we can then use the ILDB instruction to read bytes in succession from the buffer.

Under the successful return from RDTTY we install an instruction that initializes the byte pointer. We shall hold the byte pointer in the accumulator that we call B. It is not necessary that the byte pointer be held in an accumulator; it may be in any memory location. Often, we find that holding a byte pointer in an accumulator to be very convenient.

```
        RDTTY
         ERJMP  ERROR
        MOVE    B,[POINT 7,IBUFR]
INLOOP: . . .
```

Recall the explanation of the POINT pseudo–op: the byte pointer that is formed from the expression POINT 7,IBUFR describes the non–existent byte to left of the first real byte in the word at IBUFR. The use of a pointer to a non–existent byte should not disturb you; we shall increment this pointer to make it point to a real byte before we attempt to load a byte. The instruction to replace PBIN at ILOOP should now be obvious. The byte pointer is in B; the byte is wanted in A. An ILDB instruction does the trick:

```
        RDTTY
         ERJMP  ERROR
        MOVE    B,[POINT 7,IBUFR]       ;B := pointer to buffer start
INLOOP: ILDB    A,B                     ;A gets a byte from the buffer
```

We will embellish the program at INLOOP by changing the loop's end test to exit if a null character is seen. Nulls are normally ignored by RDTTY, but if a long line is entered, a null will appear at the end of the buffer instead of a line feed:

```
GETLIN: HRROI   A,[ASCIZ/Please type a line: /]
        MOVE    C,A                     ;save the reprompt pointer
        PSOUT                           ;prompt for a line
        HRROI   A,IBUFR                 ;pointer to input buffer
        MOVEI   B,BUFLEN*5-1            ;length of buffer, no flags
        RDTTY
         ERJMP  ERROR
        MOVE    B,[POINT 7,IBUFR]       ;B := pointer to buffer start
INLOOP: ILDB    A,B                     ;A gets a byte from the buffer
        CAIN    A,15                    ;skip unless carriage return
         JRST   INLOOP                  ;discard carriage return
        CAIN    A,12                    ;skip unless end of line
         JRST   INDONE                  ;LF was seen. We are done now.
        JUMPE   A,INDONE                ;Treat null as end of line.
        PUSH    P,A                     ;store character on the stack
        JRST    INLOOP                  ;go process more.
```

We have now finished making all the changes that are necessary to convert this program from using PBIN to using RDTTY. We wish to undertake one further set of modifications. Since the repeated use of PBOUT suffers from some of the same overhead costs as the use of PBIN, we shall convert this program to use PSOUT by installing an output buffer. Instead of sending each character to the terminal as soon as it has been recovered from the stack, we will store the characters in a buffer area. When the entire output line has been composed, it will be sent to the terminal via one PSOUT.

We must reserve space for the output buffer. Because we add the carriage return, line feed, and null characters to the end of the output line, we make the output buffer one word (i.e., five characters) larger than the input buffer. Prior to entering the loop at OUTLOO we must initialize another byte pointer. Since register B is no longer important to us, we shall discard its previous contents and use it to hold the pointer to the output buffer. We must also add the instructions to place the sequence carriage return, line feed, and null at the end of the string we have composed in OBUFR:

```
OBUFR:  BLOCK   BUFLEN+1        ;room for output line
        . . .

        MOVE    B,[POINT 7,OBUFR]
OUTLOO: POP     P,A
        IDPB    A,B
        CAME    P,[IOWD PDLEN,PDLIST]
         JRST   OUTLOO
        MOVEI   A,15            ;add carriage return to the buffer
        IDPB    A,B
        MOVEI   A,12            ;add line feed to the buffer
        IDPB    A,B
        MOVEI   A,0             ;end with a null byte
        IDPB    A,B             ;to make PSOUT happy
        HRROI   A,OBUFR         ;pointer to the output area.
        PSOUT
```

The instruction sequence at OUTLOO, notably the addition of a null to the end of the string, has built an ASCIZ–format string that is suitable for use with the PSOUT JSYS.

The entire program that we have constructed appears below:

```
        TITLE   REVERSE - Example 4-B
        SEARCH  MONSYM

Comment $
        This is another program to demonstrate the reversal of the
        characters on an input line.

        The structure or organization of this program is similar to
        that found in example 4-A.  In contrast to example 4-A, this
        program uses the byte instructions to make character processing
        easier.

$

A=1                             ;Assign symbolic names to
B=2                             ;the accumulators
C=3
P=17                            ;Symbolic for push down pointer

BUFLEN==40
PDLEN==240

        .PSECT  DATA,1001000
IBUFR:  BLOCK   BUFLEN                  ;buffer for input line
OBUFR:  BLOCK   BUFLEN+1                ;buffer for output line
PDLIST: BLOCK   PDLEN

        .PSECT  CODE/RONLY,1002000
START:  RESET                           ;initialize IO
        MOVE    P,[IOWD PDLEN,PDLIST]   ;initial stack pointer
GETLIN: HRROI   A,[ASCIZ/Reverse a line: /]
        MOVE    C,A                     ;save this prompt for RDTTY
        PSOUT                           ;prompt for a line
        HRROI   A,IBUFR                 ;Setup buffer space for RDTTY
        MOVEI   B,BUFLEN*5-1            ;size of buffer
        RDTTY                           ;read a line
         ERJMP  ERROR
        MOVE    B,[POINT 7,IBUFR]       ;Initial input pointer to line
INLOOP: ILDB    A,B                     ;gobble a byte
        CAIN    A,15                    ;is it carriage return
         JRST   INLOOP                  ;discard return
        CAIN    A,12                    ;is it the line-feed?
         JRST   INDONE                  ;yes.  finished with line
        JUMPE   A,INDONE                ;jump in case of input buffer overrun.
        PUSH    P,A                     ;normal character, save it.
        JRST    INLOOP                  ;loop through line
```

```
INDONE: CAMN    P,[IOWD PDLEN,PDLIST]   ;is the stack empty?
         JRST    STOP                   ;yes, line was empty
        HRROI   A,[ASCIZ/Reversed  line: /] ;tell them what's coming
        PSOUT
        MOVE    B,[POINT 7,OBUFR]       ;Initialize output pointer
OUTLOO: POP     P,A                     ;get a byte from stack
        IDPB    A,B                     ;store the byte
        CAME    P,[IOWD PDLEN,PDLIST]   ;Is the stack unwound now,
         JRST   OUTLOO                  ;no. loop until stack empties
        MOVEI   A,15                    ;stack is now empty.
        IDPB    A,B                     ;add cr at end of line
        MOVEI   A,12
        IDPB    A,B                     ;add lf
        MOVEI   A,0
        IDPB    A,B                     ;end with null for PSOUT
        HRROI   A,OBUFR
        PSOUT                           ;type buffer
        JRST    GETLIN

ERROR:  HRROI   A,[ASCIZ/ERROR in RDTTY.
/]
        PSOUT
STOP:   HALTF                           ;stop at a blank line
        JRST    STOP                    ;stay stopped.

        END     START
```

In case a long line is input, the `RDTTY` JSYS was told to avoid completely filling the input buffer, `IBUFR`. We told `RDTTY` to read up to `BUFLEN*5-1` characters. The extra `-1` in the expression leaves room at the end of the buffer. That remaining character is initially a null, and nothing ever changes it.

If a line longer than `237` (octal, or `159` decimal) characters is typed, `RDTTY` will fill up `IBUFR` with as many characters as will fit, and then return to the program. In such a case neither carriage return nor line feed will be present in the `IBUFR` area. However, the extra instruction in the input loop, `JUMPE A,INDONE` will catch the null at the end of the buffer and exit from `INLOOP`.

The behavior of the program in such conditions will be somewhat strange. The first portion of the line will be reversed; this will be followed by a second prompt, and followed immediately by the second portion of the line, reversed. Since such long lines are rather unlikely, perhaps we should not have bothered about them at all. However, a careful programmer should consider all possibilities to be sure that the program always does something reasonable.

11.4 Example 5 — Character Processing

We shall now build a program in which we read a line of input and type the odd characters (i.e., the first, third, fifth, etc.) on one output line and the even characters on the next line. This program stops when a blank line is typed as input.

```
Type a line: This is a sample input line
             T i   s a s m l   n u   i e
              h s i    a p e i p t l n
Type a line:
```

We begin by writing a fragment to read a line of input from the terminal. The usual definitions of a buffer space, called BUFFER, the length of the buffer area, BUFLEN, and symbolic accumulator names will be present. We label this fragment GETLIN; the program will return to this point to obtain another input line, except when a blank line is present.

```
        . . .                           ;initialize

GETLIN: HRROI   A,PROMPT                ;prompt for input
        PSOUT
        HRROI   A,BUFFER                ;read input to buffer area
        MOVEI   B,BUFLEN*5-1            ;buffer count
        HRROI   C,PROMPT
        RDTTY                           ;wait for user input
         ERJMP  ERROR
        . . .                           ;process the characters
        . . .                           ;decide when to stop, or
        . . .                           ;print the results

        JRST    GETLIN                  ;get another line of input
        . . .
```

If the input line is empty, then the first character in the buffer will be either a carriage return or a line feed. Normally, we would expect to see both the carriage return and line feed, but the user might type a line feed character to end the line, in which case no carriage return appears. We will load the first character from the input buffer and test for the end of line in the following sequence:

```
        LDB     A,[POINT 7,BUFFER,6]    ;read the first character
        CAIE    A,15                    ;skip if it is carriage return
         CAIN   A,12                    ;skip unless it is a line feed
          JRST  STOP                    ;line is empty.  Stop now.
```

The next problem is to produce the odd and even output lines. We have a choice here. Either we can use one output buffer and scan the input line twice, or we can use two output buffers and scan the line once. Generally, if you have a chance to do a function once instead of twice, it is faster to do it only once. On the other hand, space considerations may dictate that that a second pass is preferable. In this case, the space requirements are very modest, so we will scan the line once, producing two output buffers. We have the following general form for this portion of the program:

```
        . . .                           ;initialize line processing
LOOP:   . . .                           ;get an odd character
        . . .                           ;exit loop if end of line
        . . .                           ;store odd char in odd buffer
        . . .                           ;store space in even buffer
        . . .                           ;get an even character
        . . .                           ;exit loop if end of line
        . . .                           ;put even char in even buffer
        . . .                           ;store blank in odd buffer
        JRST    LOOP                    ;get another odd character

LOOPX:                                  ;here at end of line
```

We must initialize three byte pointers. One for the input line, the second for the odd output line, and the third for the even output line. Again, we find it convenient to hold these byte pointers in accumulators. Also, it will be convenient to use one accumulator, B, to hold an ASCII blank character.

```
        MOVE    INP,[POINT 7,BUFFER]    ;input line pointer
        MOVE    ODDP,[POINT 7,OLINE]    ;odd line
        MOVE    EVENP,[POINT 7,ELINE]   ;even line
        MOVEI   B," "                   ;one blank character
```

An ILDB is used to obtain a character from the input buffer:

```
LOOP:   ILDB    A,INP                   ;read an odd character
```

The test for the end of line is somewhat complicated. Usually we expect that a carriage return will be present. However, if the user types a line feed character to end the line, no carriage return will be present. These are the same considerations that we mentioned when we were creating the fragment to test for an empty line. However, there is one additional consideration. If the user supplies a line that is too long, we will run through the entire buffer without finding a return or a line feed.

In case an input line is too long, the entire buffer will be filled with characters. However, we told the system that the buffer was slightly shorter than it really is. The extra character position that we hide from the system contains a null to mark the end of the buffer. Our program will detect the end of the line when any one of carriage return, line feed, or null is seen:

```
LOOP:   ILDB    A,INP                   ;get an odd character
        CAIE    A,15                    ;skip if carriage return
         CAIN   A,12                    ;skip unless line feed
          JRST  LOOPX                   ;leave loop
        JUMPE   A,LOOPX                 ;leave loop if null is seen
```

Next we must add the odd character to the odd buffer and add a blank to the even buffer. The byte pointer in ODDP addresses the odd buffer; the pointer in EVENP addresses the even buffer:

```
        IDPB    A,ODDP                  ;deposit an odd character
        IDPB    B,EVENP                 ;deposit a blank in even line
```

Putting these fragments together, in the framework we described above, we get the following more nearly complete program:

```
;initialize
        MOVE    INP,[POINT 7,BUFFER]    ;input line pointer
        MOVE    ODDP,[POINT 7,OLINE]    ;odd line
        MOVE    EVENP,[POINT 7,ELINE]   ;even line
        MOVEI   B," "                   ;one blank character
LOOP:   ILDB    A,INP                   ;read an odd character
        CAIE    A,15                    ;skip if carriage return
         CAIN   A,12                    ;skip unless line feed
          JRST  LOOPX                   ;leave loop
        JUMPE   A,LOOPX                 ;leave loop if null is seen
        IDPB    A,ODDP                  ;deposit an odd character
        IDPB    B,EVENP                 ;deposit a blank in even line
        . . .                           ;get an even character
        . . .                           ;exit loop if end of line
        . . .                           ;put even char in even buffer
        . . .                           ;store blank in odd buffer
        JRST    LOOP                    ;get another odd character

LOOPX:  . . .
```

We can now write the fragment for dealing with the even characters. It is quite similar to what we have just been through for the odd characters:

```
        ILDB    A,INP                   ;read an even character
        CAIE    A,15                    ;skip if carriage return
         CAIN   A,12                    ;skip unless line feed
          JRST  LOOPX                   ;leave loop
        JUMPE   A,LOOPX                 ;leave loop if null is seen
        IDPB    A,EVENP                 ;put even char in even buffer
        IDPB    B,ODDP                  ;put a blank in odd buffer
```

After the end of the input line is found, we must add the sequence carriage return, line feed, and null to each of the output lines.

```
;here when input line is exhausted.
LOOPX:  MOVEI   B,15                    ;add carriage return
        IDPB    B,ODDP                  ;to odd buffer
        IDPB    B,EVENP                 ;to even buffer
        MOVEI   B,12                    ;add line feed
        IDPB    B,ODDP                  ;to the odd line
        IDPB    B,EVENP                 ;and to the even line
        MOVEI   B,0                     ;finally, add a null
        IDPB    B,ODDP                  ;to both lines,
        IDPB    B,EVENP                 ;to make PSOUT happy.
```

Now that the lines have been created, they must be printed:

```
        HRROI   A,HEAD
        PSOUT                           ;some spaces in front of line
        HRROI   A,OLINE
        PSOUT                           ;send odd line
        HRROI   A,HEAD
        PSOUT                           ;more spaces
        HRROI   A,ELINE
        PSOUT                           ;the even line
```

We can now put all these fragments together and add the several definitions that are necessary to form the complete program for Example 5:

```
        TITLE   EVEN ODD - Example 5
        SEARCH  MONSYM

Comment $
This program reads a line of input and types the odd characters (i.e., the
first, third, fifth, etc.) on one output line and the even characters on
the next line.  This program stops when a blank line is typed as input.

Type a line: This is a sample input line
             T i   s a s m l   n u   i e
              h s i     a p e i p t l n
Type a line:

$

A=1
B=2
C=3
INP=5                                   ;input line pointer
EVENP=6                                 ;even line pointer
ODDP=7                                  ;odd line pointer
```

```
        .PSECT  CODE/RONLY,1002000
START:  RESET                           ;a good way to start
        HRROI   A,[ASCIZ/Welcome to Even-Odd
/]
        PSOUT                           ;send greetings
;obtain an input line
GETLIN: HRROI   A,PROMPT                ;prompt for input
        PSOUT
        HRROI   A,BUFFER                ;read input to buffer area
        MOVEI   B,BUFLEN*5-1            ;buffer count
        HRROI   C,PROMPT
        RDTTY                           ;wait for user input
         ERJMP  ERROR
;determine if the line is empty
        LDB     A,[POINT 7,BUFFER,6]    ;read the first character
        CAIE    A,15                    ;skip if it is a carriage return
         CAIN   A,12                    ;skip unless it is a line feed
          JRST   STOP                    ;line is empty.  Stop now.
;initialize to process the input line
        MOVE    INP,[POINT 7,BUFFER]    ;input line pointer
        MOVE    ODDP,[POINT 7,OLINE]    ;odd line
        MOVE    EVENP,[POINT 7,ELINE]   ;even line
        MOVEI   B," "                   ;one blank character
;process an odd character
LOOP:   ILDB    A,INP                   ;read an odd character
        CAIE    A,15                    ;skip if carriage return
         CAIN   A,12                    ;skip unless line feed
          JRST  LOOPX                   ;leave loop
        JUMPE   A,LOOPX                 ;leave loop if null is seen
        IDPB    A,ODDP                  ;deposit an odd character
        IDPB    B,EVENP                 ;deposit a blank in even line
;process an even character
        ILDB    A,INP                   ;read an even character
        CAIE    A,15                    ;skip if carriage return
         CAIN   A,12                    ;skip unless line feed
          JRST  LOOPX                   ;leave loop
        JUMPE   A,LOOPX                 ;leave loop if null is seen
        IDPB    A,EVENP                 ;deposit even char in even buffer
        IDPB    B,ODDP                  ;deposit a blank in odd buffer
        JRST    LOOP                    ;get the next odd character

;here when the input line is exhausted.
;add carriage return, line feed, and null to each output line
LOOPX:  MOVEI   B,15                    ;add carriage return
        IDPB    B,ODDP                  ;to odd buffer
        IDPB    B,EVENP                 ;to even buffer
        MOVEI   B,12                    ;add line feed
        IDPB    B,ODDP                  ;to the odd line
        IDPB    B,EVENP                 ;and to the even line
        MOVEI   B,0                     ;finally, add a null
        IDPB    B,ODDP                  ;to both lines,
        IDPB    B,EVENP                 ;to make PSOUT happy.
```

```
;print the odd line, then the even line
        HRROI   A,HEAD
        PSOUT                           ;some spaces in front of the line
        HRROI   A,OLINE
        PSOUT                           ;send odd line
        HRROI   A,HEAD
        PSOUT                           ;more spaces
        HRROI   A,ELINE
        PSOUT                           ;the even line
;done with one line.
        JRST    GETLIN                  ;do another line
ERROR:  HRROI   A,[ASCIZ/Error from RDTTY
/]
        PSOUT
STOP:   HALTF
        JRST    STOP

BUFLEN==40
        .PSECT  .DATA,1001000
BUFFER: BLOCK   BUFLEN                          ;input buffer
OLINE:  BLOCK   BUFLEN+1                        ;odd line buffer
ELINE:  BLOCK   BUFLEN+1                        ;even line buffer
HEAD:   ASCIZ   /             /                 ;same length as PROMPT
PROMPT: ASCIZ   /Type a line: /

        END     START
```

11.5 Alternative Techniques

One of the more unfortunate characteristics of the loop at `LOOP` is that the end of line test appears twice. There are at least two basic ways to avoid the repetition. One way is to make a subroutine from the fragment that reads the next input character and tests for end of line; we will discuss this further when we get to subroutines. The second way to avoid the repetition of this fragment is to wrap the two different parts of the loop into one. There are several ways to group these two parts together. We will examine two of them in more detail.

11.5.1 Flags for Control

We can restructure the loop so that a *flag* or *switch* variable controls the action of the program within the loop. Suppose we have a variable called `ODDC` which has value `1` when an odd character is being processed, and value `0` when an even character is being done. We will keep this flag in an accumulator. Then we could write the processing loop as follows:

```
        . . .                   ;initialize
        MOVEI   ODDC,1          ;set to odd character
LOOP:   . . .                   ;get a character,
        . . .                   ;exit loop if end of line
        JUMPE   ODDC,ELOOP      ;if not odd, process even character.
        . . .                   ;process odd character
        MOVEI   ODDC,0          ;set next character is even
        JRST    LOOP

ELOOP:  . . .                   ;process even character
        MOVEI   ODDC,1          ;set next character is odd
        JRST    LOOP
```

It isn't difficult to fill in the blanks. Also, we can take advantage of the SOJA and AOJA instructions to set the ODDC flag and jump in one operation:

```
        MOVE    INP,[POINT 7,BUFFER]    ;input line pointer
        MOVE    ODDP,[POINT 7,OLINE]    ;odd line
        MOVE    EVENP,[POINT 7,ELINE]   ;even line
        MOVEI   B," "                   ;one blank character
        MOVEI   ODDC,1                  ;set to odd character first
LOOP:   ILDB    A,INP                   ;get a character
        CAIE    A,15                    ;skip if carriage return
         CAIN   A,12                    ;skip unless line feed
          JRS    LOOPX                   ;leave loop
        JUMPE   A,LOOPX                 ;leave loop if null is seen
        JUMPE   ODDC,ELOOP              ;jump when doing an even char
        IDPB    A,ODDP                  ;deposit an odd character
        IDPB    B,EVENP                 ;deposit a blank in even line
        SOJA    ODDC,LOOP               ;make even for next time. loop

ELOOP:  IDPB    A,EVENP                 ;deposit an even character
        IDPB    B,ODDP                  ;deposit a blank in odd line
        AOJA    ODDC,LOOP               ;make odd for next time. loop
```

11.5.2 Control Without Flags

We can remove all the extra control logic if we simply remember that each character causes an alternation. By interchanging the byte pointers that are kept in ODDP and EVENP after each character is processed we can force the first character into OLINE, the second into ELINE, the third into OLINE, etc. This interchange is accomplished by means of the EXCH instruction. It should be noted that in using EXCH we confuse the mnemonic significance of the names ODDP and EVENP; we hope that is all we confuse.

```
        MOVE    INP,[POINT 7,BUFFER]    ;input line pointer
        MOVE    ODDP,[POINT 7,OLINE]    ;odd line
        MOVE    EVENP,[POINT 7,ELINE]   ;even line
        MOVEI   B," "                   ;one blank character
LOOP:   ILDB    A,INP                   ;get a character
        CAIE    A,15                    ;skip if carriage return
         CAIN   A,12                    ;skip unless line feed
          JRST  LOOPX                   ;leave loop
        JUMPE   A,LOOPX                 ;leave loop if null is seen
        IDPB    A,ODDP                  ;deposit the char in one place
        IDPB    B,EVENP                 ;and a blank in the other.
        EXCH    ODDP,EVENP              ;exchange odd & even pointers!
        JRST    LOOP
```

11.6 Exercises

11.6.1 Test for an Empty Line

In example 5, we wrote the test for an empty line using the LDB instruction. Why did we avoid the following bad example?

```
;This is a bad example.  What is wrong with it?
        ILDB    A,[POINT 7,BUFFER]      ;read the first character
        CAIE    A,15                    ;skip if it is carriage return
         CAIN   A,12                    ;skip unless it is a line feed
          JRST  STOP                    ;line is empty.  Stop now.
        ...
```

11.6.2 Interleave Program

Create a program to read two lines from the terminal. Then output the characters from the two lines in interleaved order. That is, output characters in the order:

line 1 char 1, line 2 char 1, line 1 char 2, line 2 char 2, line 1 char 3, line 2 char 3, etc.

Your program should be capable of processing many input pairs. That is, after you've successfully output an interleaved line, you should ask for more input.

If the input lines are not of equal length, after interleaving as much as you can, output the remainder of the longer input line.

If the first line contains nothing more than a carriage return and line feed, make the program stop.

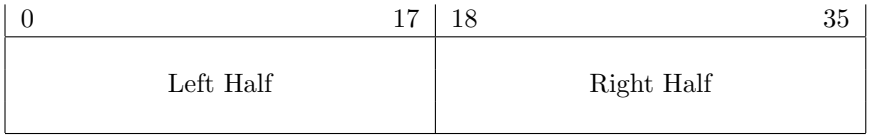
Examples:

```
line1:  ABCDEFG
line2:  1234567
output line:  A1B2C3D4E5F6G7
line1: THIS IS
line2: this is a test
output line: TtHhIiSs  IiSs a test
```

Chapter 12

Halfword Instructions

The halfword class of instructions perform data transmission between one half of an accumulator and one half of an arbitrary memory location. There are sixty–four halfword instructions. Each mnemonic begins with the letter `H` and has four modifier letters. The first modifier selects one half of the source word; the second selects one half of the destination word. The third modifier specifies what change to effect on the other half of the destination. Finally, the fourth modifier specifies the direction of data movement, e.g., from memory to an accumulator or from the accumulator to memory.



The sixty–four halfword instructions are formed as described here:

`H` Halfword from the { `R` Right / `L` Left } half of the source word

to the { `R` Right / `L` Left } half of the destination word,

with { ␣ no change / `Z` Zeros / `O` Ones / `E` sign Extension } to the "other half" of the destination word.

Where Source and Destination are {
- ␣ C(E) to C(AC).
- `I` Immediate : 0, , ER to C(AC).(Except`HLLI`)
- `M` to Memory : C(AC) to C(E).
- `S` to Self : C(E) to C(E), and, if AC is not zero, copy the result to C(AC).

The "other half" of the destination is the halfword not specified by the second "R" or "L".

"Sign extension" means that the most significant bit of the source halfword is copied to the 18–bits of the other half of the destination.

The S, or "self" mode in these instructions is similar to the self mode in the MOVE instructions: the source is the memory operand, the destination is the same word in memory and, if an accumulator other than number 0 is named, the entire destination word will be copied to that accumulator. The computer will generate only one result word: in self mode it may store it in two places.

In the algebraic representation of the halfword instructions that follow, we introduce some additional nomenclature. The notations C0(E) and C18(E) mean the contents of bit 0 of E and the contents of bit 18 of E, respectively. We have used the notation A,,B before; it means the 36–bit word composed of the 18–bit quantity A on the left and the 18–bit quantity B on the right.

The immediate mode operand is 0,,ER in all instructions excepting HLLI. For HLLI, which is called XHLLI for this reason, the immediate mode operand is E, the 30–bit effective address, padded with zeros in bits 0:5.

HRR	C(AC)	:=	CL(AC),,CR(E)
HRRI	C(AC)	:=	CL(AC),,E
HRRM	C(E)	:=	CL(E),,CR(AC)
HRRS	Temp	:=	C(E) := CL(E),,CR(E); if AC>0 then C(AC) := Temp
HRRZ	C(AC)	:=	0,,CR(E)
HRRZI	C(AC)	:=	0,,E MOVEI is preferred
HRRZM	C(E)	:=	0,,CR(AC)
HRRZS	Temp	:=	C(E) := 0,,CR(E); if AC>0 then C(AC) := Temp
HRRO	C(AC)	:=	777777,,CR(E)
HRROI	C(AC)	:=	777777,,E
HRROM	C(E)	:=	777777,,CR(AC)
HRROS	Temp	:=	C(E) := 777777,,CR(E); if AC>0 then C(AC) := Temp
HRRE	C(AC)	:=	777777*C18(E),,CR(E)
HRREI	C(AC)	:=	777777*E18,,E
HRREM	C(E)	:=	777777*C18(AC),,CR(AC)
HRRES	Temp	:=	C(E) := 777777*C18(E),,CR(E); if AC>0 then C(AC) := Temp
HRL	C(AC)	:=	CR(E),,CR(AC)
HRLI	C(AC)	:=	E,,CR(AC)
HRLM	C(E)	:=	CR(AC),,CR(E)
HRLS	Temp	:=	C(E) := CR(E),,CR(E); if AC>0 then C(AC) := Temp
HRLZ	C(AC)	:=	CR(E),,0
HRLZI	C(AC)	:=	E,,0 MOVSI is preferred
HRLZM	C(E)	:=	CR(AC),,0
HRLZS	Temp	:=	C(E) := CR(E),,0; if AC>0 then C(AC) := Temp

HRLO	C(AC)	:=	CR(E),,777777
HRLOI	C(AC)	:=	E,,777777
HRLOM	C(E)	:=	CR(AC),,777777
HRLOS	Temp	:=	C(E) := CR(E),,777777; if AC>0 then C(AC) := Temp
HRLE	C(AC)	:=	CR(E),,777777*C18(E)
HRLEI	C(AC)	:=	E,,777777*E18
HRLEM	C(E)	:=	CR(AC),,777777*C18(AC)
HRLES	Temp	:=	C(E) := CR(E),,777777*C18(E); if AC>0 then C(AC) := Temp
HLR	C(AC)	:=	CL(AC),,CL(E)
HLRI	C(AC)	:=	CL(AC),,0 not useful
HLRM	C(E)	:=	CL(E),,CL(AC)
HLRS	Temp	:=	C(E) := CL(E),,CL(E); if AC>0 then C(AC) := Temp
HLRZ	C(AC)	:=	0,,CL(E)
HLRZI	C(AC)	:=	0
HLRZM	C(E)	:=	0,,CL(AC)
HLRZS	Temp	:=	C(E) := 0,,CL(E); if AC>0 then C(AC) := Temp
HLRO	C(AC)	:=	777777,,CL(E)
HLROI	C(AC)	:=	777777,,0
HLROM	C(E)	:=	777777,,CL(AC)
HLROS	Temp	:=	C(E) := 777777,,CL(E); if AC>0 then C(AC) := Temp
HLRE	C(AC)	:=	777777*C0(E),,CL(E)
HLREI	C(AC)	:=	0
HLREM	C(E)	:=	777777*C0(AC),,CL(AC)
HLRES	Temp	:=	C(E) := 777777*C0(E),,CL(E); if AC>0 then C(AC) := Temp
HLL	C(AC)	:=	CL(E),,CR(AC)
HLLI	C(AC)	:=	EL,,CR(AC). This is XHLLI
HLLM	C(E)	:=	CL(AC),,CR(E)
HLLS	Temp	:=	C(E) := CL(E),,CR(E); if AC>0 then C(AC) := Temp
HLLZ	C(AC)	:=	CL(E),,0
HLLZI	C(AC)	:=	0
HLLZM	C(E)	:=	CL(AC),,0
HLLZS	Temp	:=	C(E) := CL(E),,0; if AC>0 then C(AC) := Temp
HLLO	C(AC)	:=	CL(E),,777777
HLLOI	C(AC)	:=	0,,777777
HLLOM	C(E)	:=	CL(E),,777777
HLLOS	Temp	:=	C(E) := CL(E),,777777; if AC>0 then C(AC) := Temp

HLLE	C(AC)	:=	CL(E),,777777*C0(E)
HLLEI	C(AC)	:=	0
HLLEM	C(E)	:=	CL(AC),,777777*C0(AC)
HLLES	Temp	:=	C(E) := CL(E),,777777*C0(E); if AC>0 then C(AC) := Temp

---

Some of the halfword instructions are very commonly used. For example, in TOPS–20 system calls we have had several occasions to use HRROI to load a register with the address of a string. In system calls, the operating system substitutes the quantity 440700 for the -1 in the left half of a *source* or *destination designator*. The 440700 is the left half of the byte pointer that describes the non–existent 7 bits to the left of bit 0. In other words, the following pairs of system calls are equivalent:

```
HRROI   1,PROMPT        MOVE    1,[POINT 7,PROMPT]
PSOUT                   PSOUT
```

The interpretation of -1,,ADDR as POINT 7,ADDR is performed only by the TOPS–20 operating system software, and only on JSYS arguments that are source or destination designators. In particular, the byte instructions will *not* interpret -1,,ADDR as a reasonable byte pointer.

12.1 Using Halfword Instructions

In the PDP–10, data is often found packed with one item in each half word. The items may be addresses that describe data structures, or other 18–bit items. The halfword instructions are useful for manipulating such data items.[1]

For example, suppose we want to implement a binary tree in which each node consists of two words in the form:

```
word 0:  Address of left subtree,,Address of right subtree
word 1:  Data for this node
```

An example of such a tree appears in Figure 12.1. The pointers are simply the address of word 0 of a node, or if there is no node to point to, the field is zero. When A contains the pointer to some node, the instruction HRRZ A,0(A) changes A to point to the right subtree, or HLRZ A,0(A) would change A to point to the left subtree. (These are similar to LISP's CAR and CDR operations.)

Assuming the address of the topmost node of a tree is held in the location named ROOT, the following code will return the address of the leftmost leaf of the tree:

[1] As addresses now can be wider than a halfword, we recommend that no new code be written in which halfwords are used to hold addresses. For that reason, this example should not be emulated, although it accurately portrays traditional programming techniques.

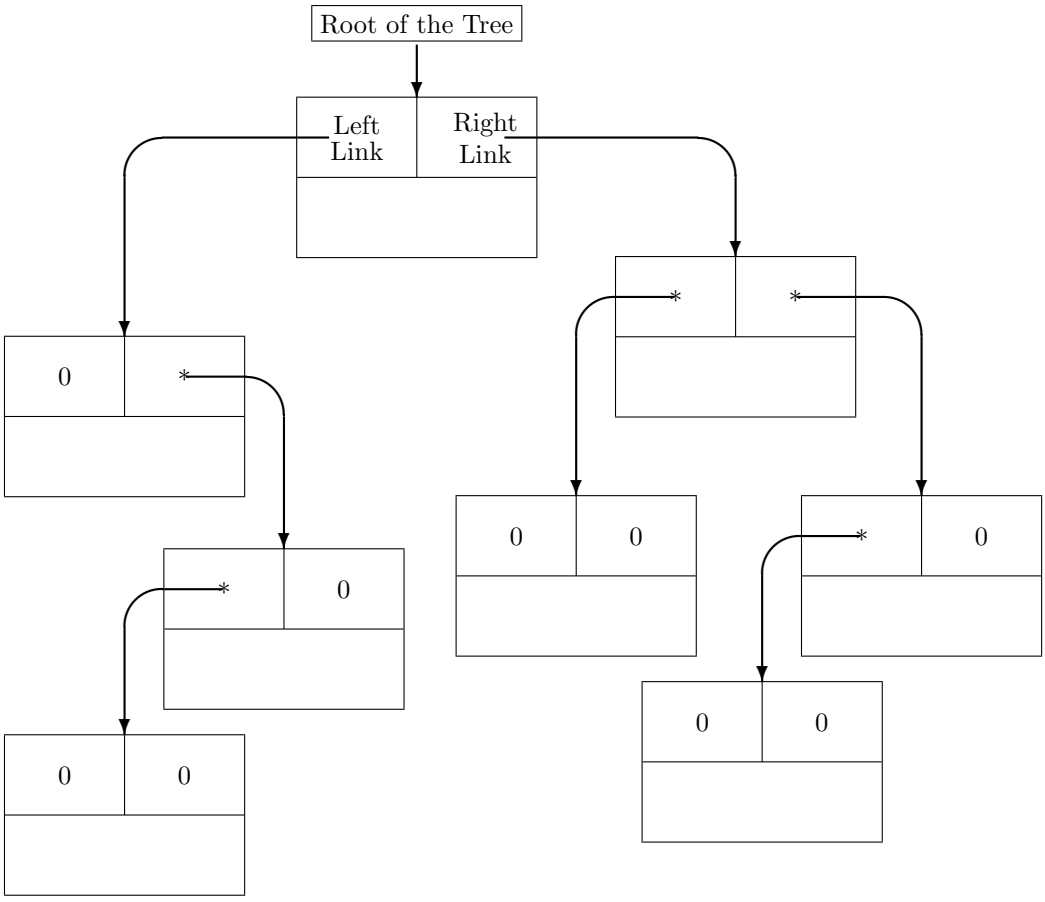


Figure 12.1: Binary Tree with Halfword Links

```
        SKIPN   A,ROOT          ;Get the address of the root node
         JRST   EMPTY           ;there is no tree at all
LOOP:   MOVE    B,A             ;save address of the current node in B
        HLRZ    A,0(A)          ;change A to point to left subtree
        JUMPN   A,LOOP          ;Loop until the left subtree is empty
        MOVE    A,1(B)          ;Load A with datum from leftmost leaf
        . . .                   ;B has the address of leftmost leaf
```

Chapter 13

Subroutines and Program Control

This chapter discusses the instructions that we use to call subroutines and return from them. Also, we will present miscellaneous program control instructions. Before we discuss the benefits of subroutines and the instructions that we use to implement subroutines, we will discuss the program counter in greater detail.

As we have already said, the program counter usually advances through consecutive memory locations as instructions are executed. The skip instructions change the program counter by incrementing it an extra time. The jump instructions supply an entire new value for the program counter. The subroutine calling instructions are similar to jump instructions in that the PC is set to a new value. The subroutine calling instructions save the value of the program counter before jumping to a new address. These instructions, together with some instructions that restore the program counter from a previously saved value, allow us to implement subroutines.

13.1 Program Counter Formats

An instruction that calls a subroutine must store the current program counter before jumping to the subroutine. The stored program counter can then be used by the subroutine to exit and return control to the calling program.

In the traditional machine, the program counter and flags were stored in a single word. In the extended machine, the program counter is 30 bits, so the flags and PC require a double word. However, in many circumstances, programmers do not require that flags be preserved by subroutines, so traditional instructions that formerly stored a word with flags and PC will store only the PC when executed in non–zero sections.

13.1.1 Single Word Flags and Program Counter

In the traditional machine and when executing in section 0, the subroutine call instructions (`PUSHJ`, `JSR`, and `JSP`) store a full word that contains the program counter and the central processor flags. Figure 13.1 depicts the flags and PC word.

The flags reflect some of the state of the computation. If a subroutine intends to be completely transparent to the calling program, it will make an effort to restore these flags. Note that this kind of transparency is more important in the transactions between the operating system and the user

program than in the activities that are wholly within the user program. This transparency is also appropriate when the program uses the pseudo interrupt system as discussed in Section 29.2.

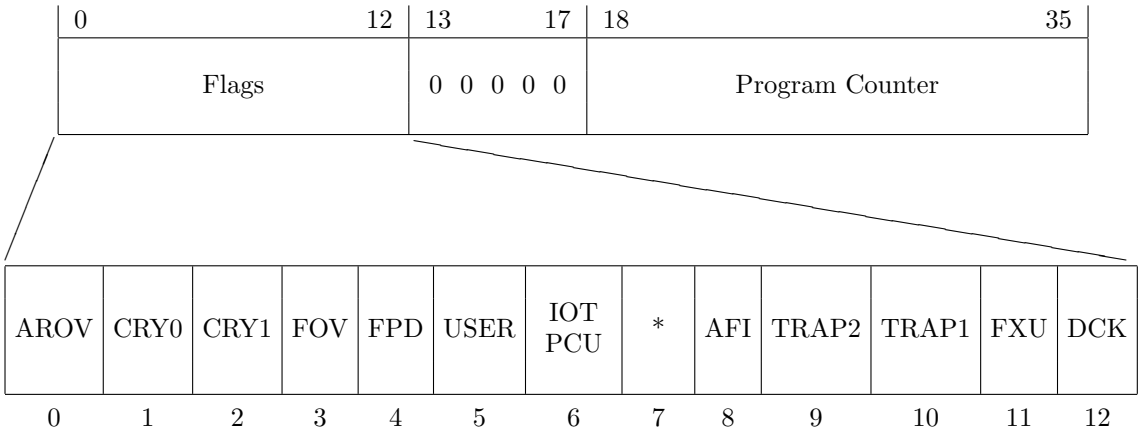


Figure 13.1: Program Counter and Flags (Single Word)

We describe the flags below. Some of these are not entirely meaningful given the typical reader's knowledge base. In time, more will become clear.

- `AROV:` The `AROV` flag indicates that some instruction has caused an arithmetic overflow, either in integer or floating–point arithmetic.
- `FOV:` The `FOV` flag signifies the occurrence of an exponent overflow or underflow in some floating–point arithmetic operation.
- `User:` The `User` flag indicates that the processor is operating in user mode, subject to the applicable restrictions on program behavior. The program must operate within the memory area assigned to it by the operating system. Input/Output instructions are illegal. Some other instructions, such as `HALT`, are illegal.
- `IOT` or `PCU`: When `IOT` is set, user–mode restrictions on privileged instructions are waived. The user–mode program can perform Input/Output instructions, etc. This flag would be set by TOPS–20 to allow a privileged program to perform instructions not generally allowed. (In executive mode, this flag serves a different purpose. There it means "previous context user".)
- Flag bit 7: in the KI10 and KL10, this is the `Public` flag; it is not used by TOPS–20. This flag is not present in the XKL–1. In the XKL–2 processor, this flag is zero in user mode; in executive mode, it reflects the state of the priority interrupt system at the time the flags were stored.
- `AFI:` This flag, address failure inhibit, can be set to allow one instruction to be performed without causing an address trap. Address trapping is one of the debugging techniques available on these computers. When it is used, this flag can be set to boost the computer past an instruction that had previously trapped.
- `TRAP2:` If `TRAP1` is not also set, `TRAP2` signifies that a pushdown overflow has occurred. If traps are enabled, setting this flag immediately causes a trap. (At present no hardware condition sets both `TRAP1` and `TRAP2` simultaneously.) The `TRAP2` flag does not exist in the KA10 and earlier machines.

- `TRAP1:` If `TRAP2` is not also set, `TRAP1` signifies that an arithmetic overflow has occurred. If traps are enabled, setting `TRAP1` immediately causes a trap. (At present no hardware condition sets both `TRAP1` and `TRAP2` simultaneously.) The `TRAP1` flag does not exist in the KA10 and earlier machines.
- `FXU:` The `FXU` flag signifies that a floating exponent underflow has occurred. Some floating–point instruction has computed a result that has an exponent smaller than (decimal) `-128`. (Or, a G–floating operation has made a result with an exponent smaller than (decimal) `-1024`.) The `AROV` and `FOV` flags will be set also.
- `DCK:` The `DCK` flag signifies that a divide check has occurred. Usually this signifies that a division by zero has been attempted. `AROV` will also be set. If the divide check occurs as a result of a floating–point instruction, then `FOV` will be set also.

In the traditional machine or in section zero, the program counter that is stored by a subroutine calling instruction will already have been incremented to point to the instruction immediately following the subroutine call; this is the normal return address. This 18–bit address is stored in bits `18:35` of the PC word. Thus, when a subroutine calling instruction stores a PC word, that word contains the address to which the subroutine should return. Bits `13:17` of the PC word are always stored as zero to facilitate the use of indirect addressing to return from a subroutine.

13.1.2 Double Word Flags and Program Counter

In non–zero sections the PC requires 30 bits. Thus the flags and PC, when both are stored, require a double word. The flags appear in the word at the lower address; the PC follows in the word at the next higher address, as shown in Figure 13.2.

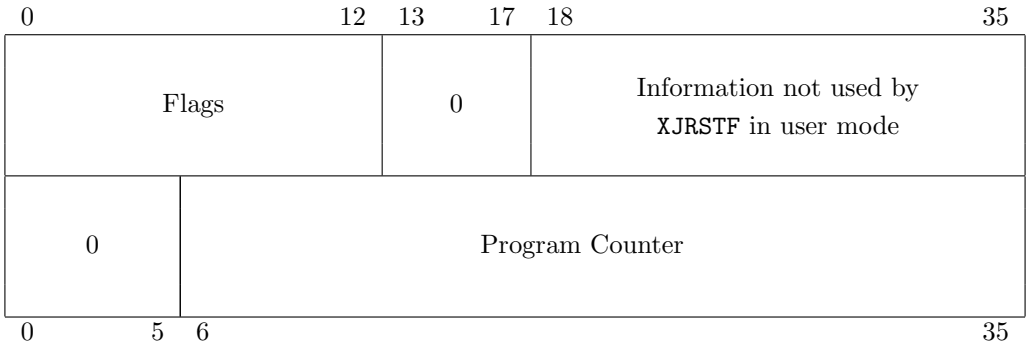


Figure 13.2: Flags and Program Counter Double–Word

13.2 Subroutine Call Instructions

Subroutines are an important programming concept. Subroutines provide at least two important benefits. First, by means of subroutines we can partition a program into manageable subtasks, with clearly defined interfaces between sections. Second, by means of parameter lists, we can cause a

subroutine to be applied to various different cases; one piece of code can be made to serve several functions.

The PDP–10 provides a variety of subroutine calling instructions. Most of the subroutine linkage techniques found in other computers are implemented in the PDP–10. However, current experience shows that only some of them are useful. Of the subroutine calling instructions described below, the most frequently used is `PUSHJ` and its corresponding return instruction, `POPJ`. The subroutine calling instructions `JSR` and `JSP` are used occasionally for special purposes.

(One pair of instructions, the subroutine call `JSA` and associated return `JRA` are now so out of favor that they are discussed in the appendix of obsolete instructions, Appendix D, page 657.)

Several additional forms of the `JRST` instruction will be described.

When executed in the traditional machine or in section zero, the subroutine call instructions store a single word containing the flags and the PC. When executed in a non–zero section, they store only the 30–bit PC.

13.2.1 PUSHJ – Push Return PC and Jump

This instruction uses a stack that is identical in format to the one used by the `PUSH` instruction. `PUSHJ` is very much like the `PUSH` instruction except the data that is stored on the stack is the return address. The effective address specifies the location that the instruction will jump to.

Traditional, i.e., Section Zero Operation

`PUSHJ`	C(AC) := C(AC)+<1,,1>; C(CR(AC)) := <Flags,,PC>; PC := E; Stack Overflow if C(AC) changes from negative to positive.

Non–Zero Section, Local Stack Pointer

`PUSHJ`	C(AC) := C(AC)+<1,,1>; C(CR(AC)) := PC; PC := E; Stack Overflow if C(AC) changes from negative to positive.

Non–Zero Section, Global Stack Pointer

`PUSHJ`	C(AC) := C(AC)+1; C(C(AC)) := PC; PC := E; Stack Overflow can not be detected.

As in the `PUSH` instruction, a pushdown overflow condition occurs if the local stack pointer becomes positive when it is incremented. The saved value of the PC points to the address following the `PUSHJ` instruction. The return from a subroutine called by `PUSHJ` is effected by the `POPJ` instruction.

`PUSHJ` is very useful; it is the most commonly used subroutine call instruction. `PUSHJ` has the disadvantage of requiring that an accumulator be set aside for the stack pointer. This disadvantage

aside, PUSHJ is reentrant and recursive; it allows multiple entry points, and it enforces a last–in, first–out discipline for subroutine calls and returns.

13.2.2 POPJ – Pop Return PC and Jump

The POPJ instruction is the usual return from PUSHJ. In section 0, the POPJ instruction copies the right half of the word at the top of the stack to the right half of the program counter; The left half of the stack top is ignored. In non–zero sections, the bits 6:35 of the word at the top of the stack are copied to PC. The POPJ instruction unwinds the stack by decrementing the stack pointer appropriate to its format. The effective address of the POPJ instruction is ignored.

Traditional, i.e., Section Zero Operation

POPJ	PC := CR(CR(AC)); C(AC) := C(AC)–<1,,1> Stack underflow (reported as "overflow") if C(AC) changes from positive to negative.

Non–Zero Section and Local Format Stack Pointer

POPJ	PC := C(CR(AC)); C(AC) := C(AC)–<1,,1> Stack underflow cannot be detected: a local format stack pointer must have a negative left half.

Non–Zero Section and Global Format Stack Pointer

POPJ	PC := C(C(AC)); C(AC) := C(AC)–1 Stack Overflow (underflow) cannot be detected.

A pushdown overflow (actually an underflow) condition results if the local stack pointer becomes negative when it is decremented.

The same stack that is used with the PUSH and POP instructions can be used with the PUSHJ and POPJ instructions. It is the programmer's responsibility to make sure that a POPJ pops a return PC and not some other data.

13.2.3 Applications of PUSHJ and POPJ

The most straightforward application of PUSHJ and POPJ is to the simple case of calling a subroutine. A stack pointer must be set up before using PUSH or PUSHJ. The subroutine is called by the instruction PUSHJ P,SUBR. P is the symbolic name for the accumulator that contains the stack pointer and SUBR is the symbolic label of the first instruction in the subroutine.

The subroutine may then do whatever computation is desired, and return to the calling program by means of a `POPJ P,` instruction.

```
        PUSHJ   P,SUBR
        . . .           ;SUBR will return to this point.
        . . .

SUBR:   . . .
        . . .
        POPJ    P,
```

Note that it is important to write the comma in the instruction `POPJ P,`; if you forget the comma, accumulator zero will be used as the stack pointer! (We often evade this difficulty by defining `RET` as `POPJ P,`.)

13.2.3.1 Nesting Subroutines

If the subroutine `SUBR` also needs to call subroutines, that can be accomplished by means of `PUSHJ` and `POPJ` also:

```
MAIN:   . . .
        PUSHJ   P,SUBR
        . . .           ;SUBR will return to this point.
        . . .

SUBR:   . . .
        PUSHJ   P,READ
        . . .           ;READ will return to this point.
        POPJ    P,

READ:   . . .
        . . .
        POPJ    P,
```

In this example, `SUBR` calls the `READ` routine. The `READ` subroutine performs its work and executes a `POPJ` instruction. Since the most recently pushed PC is the return address inside `SUBR`, the CPU resumes executing inside `SUBR`. When `SUBR` finishes and executes a `POPJ`, the current stack top contains the address in the main program to which to return.

Any number of `PUSH` or `PUSHJ` instructions can occur within `SUBR` or in the subroutines that it calls. However, an equal number of `POP` and `POPJ` instructions must be used to undo the effects of the `PUSH` and `PUSHJ` instructions. In order for `SUBR` to exit properly to its caller, the stack pointer at the time when `SUBR` exits must be the same as it was when `SUBR` was entered. Generally, keeping the stack straight is not a big problem. For every `PUSH` a `POP` is needed. If a subroutine is called by `PUSHJ` it must return via `POPJ`.

13.2.3.2 Restoring Flags

In section zero, if the program counter flags must be restored, instead of POPJ, you must resort to something like the following (see also the discussion of the JRSTF instruction in Section 13.2.4.1, page 169):

```
POP     P,TEMP          ;pop the return pc and flags
JRSTF   @TEMP           ;restore flags and return.
```

(It is not possible to restore flags from a subroutine called in a non–zero section. In non–zero sections, flags are not stored when subroutines are called.)

13.2.3.3 Skip Returns

If a subroutine called by PUSHJ P, wants to skip over one instruction immediately following the PUSHJ, the following sequence accomplishes that result:

```
AOS     (P)             ;Increment the PC word.  For indexing,
POPJ    P,              ;P must not be accumulator zero.
```

The AOS instruction specifies a zero Y field, no indirection, and uses P as an index register. By the rules of effective address calculation, the effective address will be formed by adding the Y field and the (section appropriate) contents of P. The result, since Y is zero, will be identical to the address contained in (section appropriate portion of) the stack pointer. This is the address of the stack top.

The stack top contains the return PC; the return PC is incremented by means of the AOS instruction. The POPJ instruction will copy this incremented value to the PC. Thus, the instruction immediately following the PUSHJ will be skipped. This is called a *skip return* from a subroutine.

A skip return can be used to indicate the success or failure of a subroutine. For example, a subroutine that reads the next character from a file might skip to indicate that it is returning a valid character; a direct return (also called a non–skip return) might be used to signal that the end of the file has been reached.

13.2.3.4 Recursive Subroutines

A subroutine is said to be *recursive* when it calls itself to perform its computation. In order for it to make sense for a subroutine to call itself, the problem that it is working on must somehow be simplified before making the recursive call.

As an example of the use of PUSHJ to implement recursion, consider the following routine to count the nodes in a binary tree. The format of this tree is discussed in Section 12.1, page 158.

```
        MOVEI   4,0             ;initial count is zero
        MOVE    1,ROOT          ;get the root address
        PUSHJ   P,COUNT         ;count the nodes, return result in 4
        ...

COUNT:  JUMPE   1,CPOPJ         ;nothing to count if empty
        ADDI    4,1             ;count this node
        PUSH    P,1             ;save the address of this node
        HLRZ    1,(1)           ;make a pointer to the left sub-tree
        PUSHJ   P,COUNT         ;count the left sub-tree
        POP     P,1             ;restore pointer to original node
        HRRZ    1,(1)           ;make a pointer to the right sub-tree
        PUSHJ   P,COUNT         ;count the right side
CPOPJ:  POPJ P,
```

Often it is convenient to have one `POPJ` instruction in the program labeled with the name `CPOPJ` for *Constant POPJ*. Then if you have a situation such as exists in the `COUNT` subroutine depicted above, where a conditional jump to a `POPJ` is needed, you'll have one that is already labeled.

The sequence

```
        PUSHJ   P,COUNT
        POPJ    P,
```

might be replaced with simply `JRST COUNT`, provided the `COUNT` routine does not attempt to perform a skip return. (This is an example of "tail–recursion".

13.2.3.5 ERCAL Instruction

`ERCAL` is similar to the `ERJMP` instruction that we have used for error handling. As `ERJMP`, `ERCAL` is not really recognized by the hardware; it is encoded as `JUMP 17,`, i.e., no operation. When any JSYS instruction fails, TOPS–20 looks at the next instruction in the user program. If an `ERCAL` instruction is seen, instead of doing any other processing to handle the failed JSYS, TOPS–20 returns control to the user program by "executing" the `ERCAL` instruction as though it were a `PUSHJ` that uses register `17` as the stack pointer. In other words, when TOPS–20 has an error and finds an `ERCAL` instruction in the program, control passes to the routine addressed by the effective address of the `ERCAL`; a return address (to one past the `ERCAL`) is stored on the stack described by register `17`.

13.2.4 JRST Family

As we have seen before, `JRST` is an unconditional jump instruction. The accumulator field in the `JRST` instruction does not address an accumulator; instead, the accumulator field is decoded to select specific operations, as summarized in the table presented in Figure 13.3.

When the accumulator field is zero, a simple unconditional jump occurs; we have already mentioned that `JRST` is the favorite unconditional jump instruction.

When writing any of these instructions use the indicated mnemonic. The accumulator field is listed only for reference.

Mnemonic	AC	Function
JRST	0	PC := E;
PORTAL	1	PC := E; Portal Instruction
JRSTF	2	PC := E; Restore Flags
HALT	4	PC := E; Halt the Processor
XJRSTF	5	Flags, PC := C(E,E+1); *set 30–bit PC*
XJEN	6	Flags, PC := C(E,E+1); Dismiss Interrupt
XPCW	7	C(E,E+1):= Flags and PC; Flags and PC := C(E+2,E+3)
(none)	10	PC := E; Dismiss Current Interrupt
JEN	12	PC := E; Restore Flags and Dismiss Interrupt
SFM	14	C(E) := Flags
XJRST	15	PC := C(E); *set 30–bit PC*

Figure 13.3: JRST Family

13.2.4.1 JRSTF Jump and Restore Flags

JRSTF is a JRST instruction in which the accumulator field is been set to 2. This is the section zero instruction by which the program counter and flags are restored from one word.

If indirection is used in JRSTF, then the flags are restored from the last address word fetched in the address calculation. If indexing is used with out indirection, the flags are restored from the left half of the specified index register. If neither indexing nor indirection is used in the address calculation the flags are restored from the left half of the JRSTF instruction itself. (That would generally be a mistake.)

A JRSTF will not allow a user mode program to give itself additional privileges. Thus, the User flag cannot be cleared to escape from user mode limitations. Extra privileges can be relinquished; for example, a program can clear the IOT flag.

13.2.4.2 XJRST Jump to Extended Address

XJRST allows a program to jump to another address section. Instead of taking the new PC from the effective address, XJRST takes the new PC from the contents of the word located by the effective address. This is very similar to the effect of JRST @ using a global format address word in a non–zero section. However, because XJRST interprets C(E) as a 30–bit PC regardless of other considerations, this instruction permits a program operating in section zero to jump to a non–zero section.

XJRST	PC := C(E). C(E) is a 30–bit address

13.2.4.3 XJRSTF Jump and Restore Flags, Extended

`XJRSTF` restores the flags and PC from a double word, as depicted in Figure 13.2.

`XJRSTF`	Flags := C(E); PC := C(E+1). C(E+1) is a 30–bit address Bits `0:12` of C(E) supply the replacement flags. User mode restrictions on flag combinations apply.

13.2.4.4 SFM Store Flags in Memory

If you need to access the flags, `SFM` stores them into the word specified by the effective address.

`SFM`	C(E) := Flags. Bits `0:12` of C(E) receive the flags The other bits in C(E) are set to zero.

(In the extended machine in exec mode, the flags may include bits `18:35`.)

13.2.4.5 Other JRSTs

Except for the `PORTAL` instruction, which isn't used in TOPS–20, all of the following forms of `JRST` are illegal in user mode: they are trapped as unimplemented instructions (`MUUO`s).

- `PORTAL` allows entry into a *concealed* program. Normally, if an unconcealed (public) program jumps to a concealed program, the CPU refuses to allow the public program to execute the concealed one. However, if the first instruction taken from the concealed program is a `PORTAL` instruction, the CPU allows the public program to enter concealed mode, and jumps to the effective address specified in the `PORTAL` instruction. It is presumed that a concealed program will contain `PORTAL` instructions only at locations where it is prepared to be entered, e.g., in its starting vector.

  Concealed mode is not supported in TOPS–20. `PORTAL` executes as would an ordinary `JRST`.

- `HALT` sets the PC from `E` and stops the processor when executed in executive mode. In user mode `HALT` traps as an MUUO; the TOPS-20 monitor sets the user PC (for the `Continue` command) from `E` and stops the user process.

- `JRST 10,` dismisses the current priority interrupt. Usually `JEN` or `XJEN` would used for this purpose as this instruction does not restore flags.

- `JEN` is "Jump and Enable interrupts". The `JEN` instruction dismisses the current priority interrupt and restores the PC and flags of the interrupted process. This is the traditional machine's way of exiting from a hardware interrupt, restoring the previous machine state, and re–enabling interrupts.

- `XJEN` is the extended machine's way to exit from a hardware interrupt, restoring the previous machine state, and re–enabling interrupts. The effective address specifies a double word flags and PC block.

13.2.5 JSR – Jump to Subroutine

The `JSR` instruction stores the program counter in the word addressed by the effective address and jumps to the word following the word where the PC is stored. This is the only PDP–10 instruction that stores the PC and flags without modifying any accumulators; however, it is non–reentrant, so `PUSHJ` is favored in most cases. The usual return from a subroutine that was called by a `JSR` is via `JRST` (or `JRSTF`) indirect through the PC word.

In the traditional machine or in section zero, the flags and PC are stored in the word at the effective address; in non–zero sections, only the PC is stored.

Traditional or Section Zero Operation	
`JSR`	C(E) := <flags,,PC>; PC := E+1;

Non–Zero Section Operation	
`JSR`	C(E) := PC; PC := E+1;

Programming example (traditional or section 0):

```
        JSR     SUB1    ;call subroutine SUB1
        ...

SUB1:   0               ;leave room for the PC word
        ...             ;first instruction of SUB1 is here
        JRSTF   @SUB1   ;return to caller, restoring flags
                        ;JRST @SUB1 may be used when it is
                        ;not important to restore the flags
                        ;or in non--zero sections.
```

(In this example, operation in a non–zero section would require that `JRSTF` be replaced with a simple `JRST` as there are no flags saved.

The line where the label `SUB1` appears contains only the number zero. When the assembler sees a number or an arithmetic expression appearing instead of an instruction word, it assembles that number and places it in a word. Thus, the assembler produces a word containing zero corresponding to the label `SUB1`. This zero is a place–holder for the PC word that will be stored by the `JSR SUB1`. When the program is assembled and loaded, a zero word will be present at `SUB1`. After the program is run, a return PC will appear there.

As an alternative we could write:

```
SUB1:   BLOCK   1       ;leave room for the PC word
```

In this example, we use `0` since that is easier to type than `BLOCK 1`.

Finally, note that you can't put this subroutine in a write–protected psect.

13.2.6 JSP – Jump and Save PC

The `JSP` instruction saves the PC in the selected accumulator and jumps. Return can be effected through indirection or by an indexed jump.

Traditional or Section 0 Operation	
`JSP`	C(AC) := <flags,,PC>; PC := E;

Operation in a Non-Zero Section	
`JSP`	C(AC) := PC; PC := E;

Programming example (traditional or section 0; in a non–zero section, `JRSTF` would not be used):

```
        JSP     AC,SUB2         ;call subroutine SUB2
        ...

SUB2:   ...                     ;first instruction of SUB2
        ...
        JRSTF   @AC             ;return, restoring flags, or

        JRSTF   0(AC)           ;restore flags and return, or

        JRST    @AC             ;may be used if flags are unimportant, or

        JRST    1(AC)           ;skip 1 instruction immediately after
                                ;the caller's normal return address
```

`JSP` is somewhat nicer than `JSR` because it is reentrant (i.e., `JSP` avoids storing into the instruction stream). However, `JSP` overwrites an accumulator. `JSP` is convenient in some cases; because the return PC is held in an accumulator, it is easy to effect skip returns. Also, arguments can be placed in the instruction stream immediately following the call; arguments can be picked up by using the specified accumulator as an index register. Finally, on return the argument list can be skipped over, again by using the accumulator as an index register.

When a subroutine has more than one entry point, `JSP` is better than `JSR`. For `JSR`, in order to return, you must know which entry point was called.

JSP may also be used to implement co–routines in a relatively clean way:

```
        JSP     AC,CO1          ;jump to initial co-routine entry addr
         ...
        JSP     AC,(AC)         ;reenter co-routine
                                ;note the effective address, 0(AC), is
                                ;computed before the JSP instruction
                                ;changes the contents of AC
         ...

CO1:     ...
        JSP     AC,(AC)         ;return to original caller,
         ...                    ;second call will resume here
        JSP     AC,(AC)         ;return to caller...
         ...                    ;third call resumes here, etc.
```

13.3 Program Control Instructions

There are four additional control instructions to discuss. These are JFCL, JFFO, XCT, and EXTEND. The first two of these are conditional jumps. The XCT instruction allows an arbitrary machine word to be executed as an instruction, without modifying the actual instruction stream. The EXTEND instruction is similar to XCT in that it causes the word it addresses to be executed as an instruction; the difference is that instructions executed "under" the EXTEND instruction interpret their opcode field differently.

13.3.1 JFCL – Jump on Flag and Clear

The JFCL instruction is another instance in which the accumulator field is decoded to modify the instruction. Each bit from the AC field of the instruction selects one of the PC flag bits. Instruction bits 9:12 select PC flag bits 0:3 respectively.

The JFCL instruction will jump if any PC flag corresponding to a one in the accumulator field is set. All PC flag bits that correspond to ones in the AC field will be set to zero. The mnemonic JFCL means *Jump on Flag and CLear flag*: if any of the selected flags is set, the instruction will jump; the selected flags will be set to zero.

JFCL if (PC[0:3] $\wedge$ IR[9:12]) $\neq$ 0 then PC := E;
PC[0:3] := PC[0:3] $\wedge$ $\neg$(IR[9:12])

JFCL 0, This instruction does not select any PC bits, and so it is a *no–op*; i.e., it performs no operation. JFCL is the most commonly used no–op; on the older processors it was the fastest no–op. On the KL10, TRN is faster. Somehow, it doesn't seem to matter how fast a no–op is executed.

`JFCL 17,` This instruction clears all flags; it will jump if any of the PC flags `AROV`, `CRY0`, `CRY1`, or `FOV` are set.

`JFCL 1,` This instruction, also known as `JFOV`, jumps if the floating overflow flag, `FOV`, is set. The `FOV` flag is cleared by this instruction.

`JFCL 2,` This instruction, known also as `JCRY1`, jumps if the Carry 1 flag, `CRY1`, is set. The `CRY1` flag will be cleared.

`JFCL 4,` The `JCRY0` instruction jumps if the Carry 0 flag, `CRY0`, is set. The `CRY0` flag will be cleared.

`JFCL 10,` This instruction will jump if the arithmetic overflow flag, `AROV`, is set. This instruction, which is also called `JOV`, will clear `AROV`.

`JFCL` is most often used to determine whether the immediately preceding instruction has caused an overflow. The following is one of the ways to use `JFCL`:

```
        JFCL    17,NEXT             ;clear all flags.  Jump to NEXT
NEXT:   inst                        ;instr that may cause overflow
        JOV     OVFLOW              ;jump to handle any overflow
```

The first `JFCL` in this sequence is needed because flags stay set until they are cleared. Any previous unprocessed overflow may leave `AROV` set; the first `JFCL` clears any stray flags.

If you want to be alerted when any arithmetic overflow occurs (as for example, in a program doing scientific or engineering computations), you should probably enable a *trap* as discussed in Chapter 29.

13.3.2 JFFO – Jump if Find First One

The `JFFO` instruction tests the contents of the selected accumulator. If the accumulator contains zero then `AC+1` is set to zero and no jump occurs.[1] If the selected accumulator does not contain zero then `AC+1` is set to the bit number of the leftmost one bit in the accumulator and the processor jumps to the effective address. The contents of the original accumulator are not changed.

`JFFO` If C(AC) = 0 then C(AC+1) := 0 (no jump)
else C(AC+1) := the bit number of the leftmost one in C(AC); PC := E

The `JFFO` instruction is not used very often, but when it is needed there is no plausible substitute. It can be useful in searching arrays of single bits (called *bit tables*).

MACRO implements an unary arithmetic operator that applies the `JFFO` instruction to its operand. In MACRO, when it is necessary to determine the bit number of the leftmost one bit in an expression we can use the `^L` operator. For example, the value of `^L4152` is `30` (i.e., bit `24`). Note that the operator `^L` is two characters, a caret and the letter `L`; it is *not* `CTRL/L`. We will see an application of this peculiar operator in the `FLD` macro described in Chapter 26.

[1] When we speak of `AC+1` we mean (`AC+1` modulo octal `20`). That is, `17+1` is `0`.

13.3.3 XCT – Execute Instruction

The XCT instruction reads the word specified by the effective address (the *target* of XCT) and executes that word as an instruction. If the target instruction stores the PC, that stored PC points to the instruction following the XCT. If a target instruction happens to skip, that skip is relative to the location of the XCT.

The accumulator field of the XCT should be zero.[2]

In Figure 5.6 the label "XCT Continues" means that when the effective address of the target instruction is computed, the calculation begins as local to the address section from which the target instruction was fetched.

XCT	Execute the instruction found in C(E)

The XCT instruction is quite helpful in three particular circumstances.

First, there are situations where it is necessary to compute some portion of an instruction while the program is running. In such a case, the binary image of the instruction is assembled by the program, into memory or into an accumulator. When the instruction is ready, it is executed by means of the XCT instruction.

For example, if the effective address of a MOVE instruction has been computed in accumulator 6, we would use the following sequence to execute the instruction:

```
                                    ;here with effective addr in 6
        HLL     6,[MOVE 1,]         ;set the appropriate left half
        XCT     6                   ;in register 6 and Execute it
```

An alternative to XCT, now considered disreputable, is to store the computed instruction into the stream of instructions. The following is an example. For compatibility with future processors, we strongly recommend *against* storing into the instruction stream:

This is a bad example! Do not change the instruction stream!

```
                                    ;here with effective addr in 6
        HRRM    6,NEXT              ;store addr in the next word
NEXT:   MOVE    1,0                 ;the right half of this
                                    ;instruction gets changed.
```

The second important application of the XCT instruction is to implement a CASE statement. For example:

[2] In exec mode, an XCT with a non–zero accumulator field is called PXCT, Previous Context Execute, and is used to reference data in the user's address space.

```
        MOVE    7,CASNUM                ;copy case number to 7
        XCT     CASTAB(7)               ;execute appropriate case
        ...

CASTAB: PUSHJ   17,CASE0                ;The case table just contains
        SOS     6,J                     ;instructions appropriate for
        MOVE    6,J                     ;each case.
        AOS     6,J
```

A third circumstance in which XCT is important is when the program must execute an instruction with addressing local to an address section other than the section in which the program is running. We defer giving an example until Section 15.1.2.2.

13.3.4 EXTEND – Execute from the Extended Instruction Set

The EXTEND instruction fetches the word specified by the effective address and executes that word as an instruction from the *extended instruction set*. The designers of the PDP–10 ran out of distinct instructions in the 9–bit opcode space. So, they have introduced this mechanism to give access to a set of new instructions. Each of these new instructions is said to execute "under" the EXTEND instruction.

The target instruction has I, X, and Y fields of its own. These specify the effective address to be used by the extended instruction. The extended instruction inherits its accumulator field from the AC field of the EXTEND itself.

Most usually, we see the extended instruction in a literal, as for example,

```
      EXTEND  AC,[GFIX 0,E]     ;any of I,X,Y are legal to form E in GFIX
```

The effective address of the EXTEND instruction is referred to as E0; in the example above, E0 is the address of the literal. The effective address specified by the extended instruction, i.e., by the I, X, and Y fields of GFIX instruction in the example above, is called E1.

Although this takes us a bit ahead of ourselves, the example instruction above interprets the double–word contents of E1 and E1+1 as a "giant" format floating–point number and puts the integer portion of it into AC. Note that the accumulator is selected by the AC field of the EXTEND instruction, not by the GFIX.

EXTEND Execute the extended instruction found in C(E)

Many of the new instructions added under EXTEND perform a variety of string manipulations. In general these are not discussed in this book because the necessary explanation would be tedious; moreover, the applicability of these instructions is quite narrow. They are fully documented in [SYSREF].

13.4 Example 6–A — Subroutines

This example program demonstrates the use of subroutines and skip or non–skip returns. The program will read a line of text from the terminal; the output will be all the even characters followed by the odd characters that are not vowels, followed by the odd characters that are vowels. We would not expect to find any realistic application of this peculiar function, but it does demonstrate some new ideas in a relatively uncomplicated framework.

Sample output:

```
Type a line: This is a test of the vowel extraction program.
hsi  eto h oe xrcinpormT sts ftvwltt rg.iaeeaoa
Type a line:
```

We begin by writing a fragment that reads one line from the terminal. The usual definitions and buffer space declarations are made:

```
        TITLE   EXTRACT - Example 6-A
        SEARCH  MONSYM

Comment $
This program will read a line of text from the terminal.  The output will
be all the even characters followed by the odd characters that are not
vowels, followed by the odd characters that are vowels.  The program will
halt when given an empty line.

Sample session:

Type a line: This is a test of the vowel extraction program.
hsi  eto h oe xrcinpormT sts ftvwltt rg.iaeeaoa
Type a line:

$

A=1
B=2
C=3
P=17                                    ;symbolic for push-down pointer

BUFLEN==40                              ;buffer size
PDLEN==100                              ;stack size
```

```
        .PSECT CODE/RONLY,1002000
START:  RESET                           ;reset i/o
        MOVE    P,[IOWD PDLEN,PDLIST]   ;initialize stack
NEXT:   HRROI   A,PROMPT                ;ask for input
        PSOUT
        HRROI   A,BUFFER                ;setup for RDTTY
        MOVEI   B,BUFLEN*5-1            ;buffer length, no flags
        HRROI   C,PROMPT                ;reprompt
        RDTTY                           ;read input
         ERJMP  ERROR

        . . .

ERROR:  HRROI   A,[ASCIZ/Error from RDTTY
/]
        PSOUT
STOP:   HALTF
        JRST    STOP

PROMPT: ASCIZ   /Type a line: /

        .PSECT  DATA,1001000
BUFFER: BLOCK   BUFLEN
PDLIST: BLOCK   PDLEN
        END     START
```

Now that we have read a line into the buffer area called BUFFER, we must process that line. We will make one pass through the line to move the odd characters to a second buffer, BUFR2, while moving the even characters to the output buffer, OBUFR. After the program makes this pass through the line, all of the even characters will have been moved to the output buffer; all the odd characters will be concentrated in BUFR2 for our perusal on a second pass.

The loop structure that we will adopt for the first pass looks somewhat like the following:

```
        . . .                   ;initialize
INLOOP: . . .                   ;get an odd character
        . . .                   ;exit from loop if end of line
        . . .                   ;store an odd character in BUFR2
        . . .                   ;get an even character
        . . .                   ;exit from loop if end of line
        . . .                   ;store an even character in OBUFR
        JRST    INLOOP          ;continue until end of line

        . . .

OBUFR:  BLOCK   BUFLEN          ;output buffer area
BUFR2:  BLOCK   BUFLEN          ;odd character buffer
```

In our discussion of example 5, we said it would be convenient to have a subroutine to obtain the next character and signal when the end of line is found. We shall write that subroutine for this program.

The most commonly used subroutine calling instruction in the PDP–10 is PUSHJ. We have already

established register P as the stack pointer. We can use the instruction PUSHJ P,GETCHR to call the get character, GETCHR, subroutine. The GETCHR routine is expected to return to the calling program by means of a POPJ P, instruction.

Because PUSHJ and POPJ are used to call and return from subroutines, the alternative mnemonics CALL and RET are often employed. These names are somewhat easier to type than PUSHJ P, and POPJ P,; moreover, the names CALL and RET have greater mnemonic significance to most people.

In order to use CALL and RET, we must define these names for MACRO. This definition is accomplished in each case by means of the OPDEF (operator definition) pseudo–op. In essence, OPDEF is not any different from the other means we have used to make MACRO aware of our symbolic definitions. However, MACRO distinguishes between labels and operators, so we must define operators in a different way.

We use the OPDEF pseudo–op to define the name CALL by writing on one line the word OPDEF, followed by the name we are defining, CALL, followed by a quantity enclosed in square brackets. MACRO defines the operator CALL to have the value found inside the square brackets. The definitions of CALL and RET appear below:

```
OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ P,]
```

These definitions are usually placed following the accumulator name definitions and before the names CALL or RET can be used.[3]

(By the way, these OPDEFs are included in the collection of macros and definitions available via SEARCH MACSYM; we will have occasion later to include MACSYM definitions in our programs.)

Having decided that the subroutine named GETCHR will obtain input characters for us, we can begin to fill in some of the details missing from the fragment we are building:

```
        . . .                           ;initialize
INLOOP: CALL    GETCHR                  ;get an odd character
        . . .                           ;exit from loop if end of line
        . . .                           ;store odd character in BUFR2
        CALL    GETCHR                  ;get an even character
        . . .                           ;exit from loop if end of line
        . . .                           ;store even character in OBUFR
        JRST    INLOOP                  ;continue until end of line
```

Observe that this loop requires us to store characters in BUFR2 and in OBUFR. This suggests that we must have two byte pointers, one for each of these areas. The store character instruction must certainly be an IDPB. Since GETCHR has not yet been written, we are allowed to make assumptions about how it will behave; the assumptions we make now will become the specifications for GETCHR when we come to write it. Let us now say that the GETCHR subroutine will return the next input character in register A.

We can place the IDPB instructions in the loop. Also, we add instructions to initialize registers ODDPTR and OUTPTR with byte pointers to the odd character buffer and the output buffer, respectively. This

[3]There is an obsolete TOPS–10 system call named CALL. The use of the CALL MUUO has lapsed in recent years for a number of good reasons. However, in very old programs you might still find some instances of the CALL MUUO. You must not add the OPDEF CALL to an old program unless you first check to make sure that there are no CALL MUUOs. Note also that when you use CALL instead of PUSHJ P, you must be certain that you add the appropriate OPDEF, lest you get the CALL MUUO instead.

action implies a further specification for the GETCHR routine: GETCHR must not change registers ODDPTR and OUTPTR.

Since we are speculating on the nature of the GETCHR routine, we might as well specify how it handles the end of line condition. We have seen that the end of line can be signaled by carriage return, line feed, or null. Of all these, the simplest to test for is null. Let us specify that GETCHR will return a null character (a zero) in register A to signal the end of the line. We can now completely specify the details of the loop at INLOOP:

```
ODDPTR=6                                ;odd buffer pointer
OUTPTR=7                                ;output buffer pointer

        . . .                           ;initialize
        MOVE    OUTPTR,[POINT 7,OBUFR]  ;pointer to output buffer
        MOVE    ODDPTR,[POINT 7,BUFR2]  ;pntr for storing odd letters
INLOOP: CALL    GETCHR                  ;get an odd character
        JUMPE   A,INDONE                ;exit from loop if end of line
        IDPB    A,ODDPTR                ;store odd character in BUFR2
        CALL    GETCHR                  ;get an even character
        JUMPE   A,INDONE                ;exit from loop if end of line
        IDPB    A,OUTPTR                ;store even character in OBUFR
        JRST    INLOOP                  ;continue until end of line

;here at the end of the first scan.
INDONE: . . .
```

We have made so much progress on the specification of GETCHR that we can now write it. Let us review what we know about GETCHR. First, GETCHR will read characters from the buffer area called BUFFER. This suggests that another byte pointer will be needed. Second, GETCHR is called by a PUSHJ P, instruction; it will have to return via POPJ P,. Third, GETCHR will return the next input character in register A. Fourth, accumulators ODDPTR and OUTPTR must not be changed. Fifth, when GETCHR arrives at the end of the input line, it will return a zero byte in register A.

From our first point, we know we need another byte pointer. Let register INPTR be the pointer to the input line. We must initialize INPTR to point to the entire input buffer area prior to the first call to GETCHR. This is accomplished quite simply by placing the following instruction somewhat before INLOOP:

```
        . . .

INPTR=5

        . . .

        MOVE    INPTR,[POINT 7,BUFFER]  ;initial pntr to input buffer
        . . .
INLOOP:
```

Now, the general function of GETCHR will be to increment this byte pointer to step through the input line. When a character is found that could signal the end of the line, GETCHR will transform that

character to a zero byte:

```
;Get the next character from the input line.
;Return the next character in A, or null at end of line.
GETCHR: ILDB    A,INPTR                 ;get a character from input
        CAIE    A,15                    ;skip if this character is CR
         CAIN   A,12                    ;skip unless this a LF
          MOVEI A,0                     ;LF or CR.  Return null.
        RET                             ;Return to the caller
```

This subroutine steps through the input line, character by character. It returns one character at a time in register `A`. If the input character is a carriage return, or a line feed, or a null, `GETCHR` returns a null in register `A`. `GETCHR` meets all the requirements that we set forth above.

Next, we must design the second pass. The odd characters are concentrated in `BUFR2`. We must read the characters, passing all non–vowels to the output buffer and collecting the vowels in another buffer. When we are done with this pass, the output buffer will contain the even characters followed by the odd non–vowels. The vowel buffer will have all the odd vowels. We send the output buffer to the terminal, followed by the vowel buffer, to which we have added `CR`, `LF`, and null.

```
INDONE: . . .                   ;Jump to STOP if input line is empty
        . . .                   ;initialize for second pass
OLOOP:  . . .                   ;get a character from odd buffer
        . . .                   ;jump to ODONE if odd buffer is empty
        . . .                   ;Is it a vowel?  If so jump to OLOOP1
        . . .                   ;put a non-vowel in the output buffer
        JRST    OLOOP           ;process another

OLOOP1: . . .                   ;deposit a vowel in the vowel buffer
        JRST    OLOOP           ;process another character

ODONE:  . . .                   ;add null to the output buffer
        . . .                   ;add cr lf null to vowel buffer
        . . .                   ;print output buffer; print vowel buf
        JRST    NEXT            ;get another line
```

We can begin to complete the details as follows. The test for an empty input line can check to see if `ODDPTR` has been changed. If the line is empty, there will be no first character; `ODDPTR` will not have been changed. We write the following fragment:

```
INDONE: CAMN    ODDPTR,[POINT 7,BUFR2]  ;skip unless line is empty
         JRST   STOP                    ;empty line.  stop the program
```

Assuming that the line is not empty, after making this test it is important to deposit a null byte at the end of the string of odd letters. We know that register `A` contains a null, so after the test for the empty line, we add the instruction `IDPB A,ODDPTR`. The null byte will be used in the second pass to signal the end of the buffer of odd characters.

The loop at `OLOOP` requires three byte pointers. First, we need a pointer to the odd buffer, which is

being read as input at this time. Second, we need a pointer to the vowel buffer, the place where we store the odd characters that are vowels. Finally, we will continue to use OUTPTR as the pointer to the output buffer.

Since the character buffer is being read as input this time, let us use INPTR as the byte pointer for taking characters out of BUFR2. One of the unfortunate things about giving accumulators names that have significance is that often an accumulator will be used for an entirely different purpose in another part of the program. In such a case, a mnemonic can be misleading. We recycle the accumulator ODDPTR as the pointer to the odd vowels.

Another trick we can make use of is this: we can recycle the buffer space that presently holds the odd characters and make it hold the odd vowels. Let us start by initializing the pointer INPTR to point to the buffer area for the odd characters. Also, we initialize ODDPTR to point to the same space. INPTR will be used as the *take* pointer; ODDPTR will be the *put* pointer.

We can fill in most of the second pass without too much difficulty:

```
INDONE: CAMN    ODDPTR,[POINT 7,BUFR2]  ;skip unless line is empty
         JRST   STOP                    ;empty line.  stop the program
        IDPB    A,ODDPTR                ;store null to end buffer
        MOVE    INPTR,[POINT 7,BUFR2]   ;"take" odd characters
        MOVE    ODDPTR,INPTR            ;"put" odd vowels
OLOOP:  ILDB    A,INPTR                 ;get character from odd buffer
        JUMPE   A,ODONE                 ;jump if odd buffer is empty
        . . .                           ;If not a vowel jump to OLOOP1
        IDPB    A,ODDPTR                ;put a vowel in the vowel bufr
        JRST    OLOOP                   ;process another character

OLOOP1: IDPB    A,OUTPTR                ;put a non-vowel in output buf
        JRST    OLOOP                   ;process another character

ODONE:  IDPB    A,OUTPTR                ;add null to the output buffer
        MOVEI   B,15                    ;add cr lf null to vowel bufr
        IDPB    B,ODDPTR
        MOVEI   B,12
        IDPB    B,ODDPTR
        IDPB    A,ODDPTR
        HRROI   A,OBUFR                 ;print output buffer
        PSOUT
        HRROI   A,BUFR2                 ;print vowel buffer
        PSOUT
        JRST    NEXT                    ;do another line
```

Note how INPTR and ODDPTR are initialized to point to the same buffer area. Using ODDPTR as a deposit pointer will overwrite the contents of BUFR2, the odd–character list. However, reusing that buffer space causes no harm: the byte pointer INPTR is guaranteed to be running ahead of ODDPTR, removing and processing characters before they can be harmed by being deposited onto via ODDPTR. The structure of the loop at OLOOP ensures that a byte is taken by INPTR before anything is deposited by ODDPTR. As soon as any non–vowel is seen, ODDPTR will fall behind INPTR without any possibility of catching up.

The loop at OLOOP takes a character from the odd–character list (via INPTR). For characters other

than null, we will have to determine if the character is a vowel. Non–vowels will be handled by depositing them into the output buffer. A vowel will be deposited, via the byte pointer in ODDPTR into the buffer area at BUFR2. After each character is deposited, the program loops to OLOOP. When a null character appears, OLOOP exits to ODONE; a null signals the end of the odd list.

Finally, at ODONE the program is finished processing the input line. Register A contains zero (because the way we got here was via JUMPE A,ODONE). That zero is deposited via OUTPTR to end the output buffer. Carriage return, line feed, and null are added to the end of BUFR2 (via ODDPTR). Then OBUFR and BUFR2 are printed. The program jumps to NEXT where it hopes to process another line.

One item remains unfinished. We must determine if a character is a vowel. Let us suppose that we could write a subroutine to perform this determination. We can define the characteristics of the subroutine as we choose. In this case we specify three characteristics. First, the input character will be in register A. Second, register A will not be changed by this subroutine. Third, if the given character is not a vowel, the subroutine will return to the instruction immediately following the CALL to it; if the character is a vowel, the subroutine will skip past one instruction immediately following the CALL. Given this specification, we can finish coding OLOOP and then write the subroutine. The name of this subroutine will be ISVOW, meaning, *IS this character a VOWel?*

```
OLOOP:  ILDB    A,INPTR                 ;get a character from odd bufr
        JUMPE   A,ODONE                 ;go to ODONE if odd bufr empty
        CALL    ISVOW                   ;Is it a vowel?
         JRST   OLOOP1                  ; Not a vowel.  Go to OLOOP1
        IDPB    A,ODDPTR                ;put a vowel in vowel buffer
        JRST    OLOOP                   ;process another character
        . . .

;Test character in A; skip if it is a vowel. No skip otherwise.
ISVOW:  CAIE    A,"A"                   ;Skip if "A"
         CAIN   A,"a"                   ;skip if not "a"
          JRST  CPOPJ1                  ;"A" or "a" is a vowel
        CAIE    A,"E"
         CAIN   A,"e"
          JRST  CPOPJ1
        CAIE    A,"I"
         CAIN   A,"i"
          JRST  CPOPJ1
        CAIE    A,"O"
         CAIN   A,"o"
          JRST  CPOPJ1
        CAIE    A,"u"
         CAIN   A,"U"
          JRST  CPOPJ1
        CAIE    A,"Y"                   ;skip to CPOPJ1 if this is "Y"
         CAIN   A,"y"                   ;skip to RET if not vowel
CPOPJ1:   AOS   (P)                     ;perform a skip-return.
        RET
```

The instruction sequence

```
CPOPJ1: AOS     (P)
        POPJ    P,              ;(written as RET above)
```

effects the skip return from ISVOW, as we discussed in Section 13.2.3.3, page 167. (Some programmers favor the name RSKP to label the sequence that effects the skip return.)

The ISVOW subroutine uses nested skip instructions to cut down the number of instructions that we have to write. Two consecutive tests are written as two nested tests and a jump:

```
        CAIE    A,"A"           ;skip if capital A is seen
         CAIN   A,"a"           ;skip unless lower-case A is seen
          JRST  CPOPJ1          ;Here if either "A" or "a"
        ...                     ;here if neither kind of "A" was seen
```

ISVOW contains a number of such nested skips. It is approximately the most straightforward way to test for a vowel that could be devised for this example. When a vowel is found, ISVOW jumps to CPOPJ1, which, as we have already seen, performs a skip return. If no vowel is seen, ISVOW rumbles through all the tests and eventually falls into the RET at the bottom of the routine. When we visit this problem again, we'll see another way that ISVOW can be done.

The complete program for example 6–A appears below:

```
        TITLE   EXTRACT - Example 6-A
        SEARCH  MONSYM

Comment $
This program will read a line of text from the terminal.  The output will
be all the even characters followed by the odd characters that are not
vowels, followed by the odd characters that are vowels.  The program will
halt when given an empty line.

Sample session:

Type a line: This is a test of the vowel extraction program.
hsi  eto h oe xrcinpormT sts ftvwltt rg.iaeeaoa
Type a line:

$

A=1
B=2
C=3
INPTR=5                                 ;input line pointer
ODDPTR=6                                ;odd buffer pointer
OUTPTR=7                                ;output buffer pointer
P=17                                    ;symbol for push-down pointer
```

```
BUFLEN==40                              ;buffer space
PDLEN==100                              ;stack size

OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ P,]

        .PSECT  CODE/RONLY,1002000

START:  RESET                           ;reset i/o
        MOVE    P,[IOWD PDLEN,PDLIST]   ;initialize stack
NEXT:   HRROI   A,PROMPT                ;ask for input
        PSOUT
        HRROI   A,BUFFER                ;setup for RDTTY
        MOVEI   B,BUFLEN*5-1
        HRROI   C,PROMPT
        RDTTY                           ;read input
         ERJMP  ERROR

;prepare for first scan.  separate odd and even characters
        MOVE    INPTR,[POINT 7,BUFFER]  ;pointer for processing input
        MOVE    OUTPTR,[POINT 7,OBUFR]  ;pointer to output buffer
        MOVE    ODDPTR,[POINT 7,BUFR2]  ;pntr for storing odd letters
INLOOP: CALL    GETCHR                  ;get an odd character
        JUMPE   A,INDONE                ;jump if end of line
        IDPB    A,ODDPTR                ;store odd character in buffer
        CALL    GETCHR                  ;get an even character
        JUMPE   A,INDONE                ;jump if end of line
        IDPB    A,OUTPTR                ;store even char for output
        JRST    INLOOP                  ;go on

;here at the end of the first scan.
INDONE: CAMN    ODDPTR,[POINT 7,BUFR2]  ;were there any odd chars?
         JRST   STOP                    ;no.  empty line.  stop now.
        IDPB    A,ODDPTR                ;put null to end odd buffer
;prepare for second scan
        MOVE    INPTR,[POINT 7,BUFR2]   ;fetch odd chars from here
        MOVE    ODDPTR,INPTR            ;store vowels here.
OLOOP:  ILDB    A,INPTR                 ;get a character
        JUMPE   A,ODONE                 ;jump if done
        CALL    ISVOW                   ;is this a vowel?
         JRST   OLOOP1                  ;not a vowel.  type it
        IDPB    A,ODDPTR                ;store vowel
        JRST    OLOOP                   ;get more

OLOOP1: IDPB    A,OUTPTR                ;non-vowel, put in output buf
        JRST    OLOOP                   ;do more
```

```
;here at the end of the second scan.  Print things.
ODONE:  MOVEI   B,15                    ;here when done.
        IDPB    B,ODDPTR                ;add cr, then lf
        MOVEI   B,12                    ;to the vowel list
        IDPB    B,ODDPTR
        IDPB    A,ODDPTR                ;add nulls for psout
        IDPB    A,OUTPTR                ;end output buffer
        HRROI   A,OBUFR                 ;even chrs and odd non-vows
        PSOUT
        HRROI   A,BUFR2                 ;address of odd vowel string
        PSOUT                           ;send it
        JRST    NEXT                    ;time for another input line

;Get the next character from the input line.
;Return the next character in A, or null at end of line.
GETCHR: ILDB    A,INPTR                 ;read a character from input
        CAIE    A,12                    ;test for terminator
         CAIN   A,15                    ;of either kind
          MOVEI A,0                     ;direct return if terminator
        RET

;Test character in A; skip if it is a vowel. No skip otherwise.
ISVOW:  CAIE    A,"A"
         CAIN   A,"a"
          JRST  CPOPJ1
        CAIE    A,"E"
         CAIN   A,"e"
          JRST  CPOPJ1
        CAIE    A,"I"
         CAIN   A,"i"
          JRST  CPOPJ1
        CAIE    A,"O"
         CAIN   A,"o"
          JRST  CPOPJ1
        CAIE    A,"u"
         CAIN   A,"U"
          JRST  CPOPJ1
        CAIE    A,"Y"                   ;skip to CPOPJ1 if this is "Y"
         CAIN   A,"y"                   ;skip to RET if not a vowel
CPOPJ1:   AOS   (P)                     ;perform a skip-return
        RET
```

```
ERROR:  HRROI   A,[ASCIZ/Error from RDTTY
/]
        PSOUT
STOP:   HALTF
        JRST    STOP

PROMPT: ASCIZ   /Type a line: /

        .PSECT  DATA,1001000
BUFFER: BLOCK   BUFLEN                  ;input buffer area
BUFR2:  BLOCK   BUFLEN                  ;odd chars buffer/vowel buf
OBUFR:  BLOCK   BUFLEN                  ;output buffer
PDLIST: BLOCK   PDLEN                   ;stack space
        END     START
```

13.5 Exercises

13.5.1 Change INDONE

It has been proposed that the test at `INDONE` be simplified. Instead of the `CAMN` and `JRST` it has been suggested that the single instruction `JUMPL ODDPTR,STOP` be used instead.

Why does the `JUMPL` work correctly? Can you suggest why it might not be a good idea to use `JUMPL` here? What ways could be used to overcome these objections?

Chapter 14

Tests and Booleans

You might have reached a point where you're tired of learning instructions. Unfortunately, there are many more. Besides the 128 instructions in this section, we haven't even talked about doing arithmetic. Well, relax. You don't have to use every one of them; you only need to know where to look.

The test instructions and the Boolean instructions described in this chapter have this in common: each instruction operates on two 36–bit quantities and produces a 36–bit result in which each bit of the result is a function of the specified operation and the two corresponding bits of the original operands. E.g., bit 19 of the result depends on the function selected, bit 19 of the accumulator and bit 19 of the memory (or immediate) operand. We speak of these operations on an array (or vector) of bits as being *bitwise*.

14.1 Logical Testing and Modification

The test instructions are used for testing and modifying bits in an accumulator. Each mnemonic begins with the letter "`T`" and is followed by three modifiers. The sixty–four test instructions are formed as described here:

`T` Test bits in the accumulator as selected by

- `R` Right immediate mask
- `L` Left immediate swapped mask
- `D` Direct from memory mask
- `S` Swapped from memory mask

- `N` make no change to the AC
- `Z` clear selected AC bits to Zero
- `O` set selected AC bits to One
- `C` Complement (change) selected AC bits

and

- ␣ Do not skip
- `E` Skip if all selected bits in the original AC were Equal to Zero
- `N` Skip if any selected bits in the original AC were Not Zero
- `A` Skip always

The test operation considers two `36`–bit quantities. One of these is the contents of the selected accumulator. The other quantity, called the *mask*, is specified by the first modifier letter. For `R` the mask is `<0,,ER>`; for `L` it is `<ER,,0>`. The letter `D` specifies the contents of the memory word, `C(E)`, as the mask; for `S` the mask is `CS(E)`, the swapped contents of `E`.

The result of the "test" operation is signalled by a conditional skip. When the skip condition `N` is specified, the test instruction will skip if the Boolean AND of the mask and the accumulator operand is Non–zero. The skip condition `E` specifies that the test instruction will skip when the Boolean AND of the mask and the accumulator operand is Equal to zero. The skip condition `A` means always skip; the omitted condition letter means never skip.

When the modification code `Z` appears in a test instruction, bits that are one in mask are set to zero in the accumulator:

`C(AC) := C(AC)` $\wedge$ `(`$\neg$ `mask)`.

When the modification code `O` appears, bits that are one in mask are set to one in the accumulator:

`C(AC) := C(AC)` $\vee$ `mask`.

When the modification code `C` appears, bits that are one in mask are complemented in the accumulator:

`C(AC) := C(AC)` $\oplus$ `mask`.

The modification code `N` means no modification. The accumulator will not be changed by the instruction.

Note that the skip condition is determined on the basis of the contents of the accumulator *prior* to the modification of the accumulator.

The principal use for the test instructions is in testing and modifying single–bit flags that are kept in an accumulator.

Programming Examples:

```
TRO     1,4       ;turn on bit 33 in register 1
TRZ     1,20      ;turn off bit 31 in register 1
TLON    2,400000  ;turn on bit 0 in register 2. Skip if it was
                  ; on before this instruction was executed.
TDZA    4,4       ;Turn off the register 4 bits that are on in
                  ; register 4, i.e., set 4 to zero. Skip.
```

The Test instructions are described as follows:

TRN	No–op
TRNE	If CR(AC) $\wedge$ ER $= 0$ then skip
TRNN	If CR(AC) $\wedge$ ER $\neq 0$ then skip
TRNA	Skip
TRZ	CR(AC) := CR(AC) $\wedge \neg$ER
TRZE	If CR(AC) $\wedge$ER $= 0$ then skip; CR(AC) := CR(AC) $\wedge \neg$ER
TRZN	If CR(AC) $\wedge$ER $\neq 0$ then skip; CR(AC) := CR(AC) $\wedge \neg$ER
TRZA	Skip; CR(AC) := CR(AC) $\wedge \neg$ER
TRO	CR(AC) := CR(AC) $\vee$ ER
TROE	If CR(AC) $\wedge$ ER $= 0$ then skip; CR(AC) := CR(AC) $\vee$ ER
TRON	If CR(AC) $\wedge$ ER $\neq 0$ then skip; CR(AC) := CR(AC) $\vee$ ER
TROA	Skip; CR(AC) := CR(AC) $\vee$ ER
TRC	CR(AC) := CR(AC) $\oplus$ ER
TRCE	If CR(AC) $\wedge$ ER $= 0$ then skip; CR(AC) := CR(AC) $\oplus$ ER
TRCN	If CR(AC) $\wedge$ ER $\neq 0$ then skip; CR(AC) := CR(AC) $\oplus$ ER
TRCA	Skip; CR(AC) := CR(AC) $\oplus$ ER
TLN	No–op
TLNE	If CL(AC) $\wedge$ ER $= 0$ then skip
TLNN	If CL(AC) $\wedge$ ER $\neq 0$ then skip
TLNA	Skip
TLZ	CL(AC) := CL(AC) $\wedge \neg$ER
TLZE	If CL(AC) $\wedge$ ER $= 0$ then skip; CL(AC) := CL(AC) $\wedge \neg$ER
TLZN	If CL(AC) $\wedge$ ER $\neq 0$ then skip; CL(AC) := CL(AC) $\wedge \neg$ER
TLZA	Skip; CL(AC) := CL(AC) $\wedge \neg$ER
TLO	CL(AC) := CL(AC) $\vee$ ER
TLOE	If CL(AC) $\wedge$ ER $= 0$ then skip; CL(AC) := CL(AC) $\vee$ ER
TLON	If CL(AC) $\wedge$ ER $\neq 0$ then skip; CL(AC) := CL(AC) $\vee$ ER
TLOA	Skip; CL(AC) := CL(AC) $\vee$ ER
TLC	CL(AC) := CL(AC) $\oplus$ ER
TLCE	If CL(AC) $\wedge$ ER $= 0$ then skip; CL(AC) := CL(AC) $\oplus$ ER
TLCN	If CL(AC) $\wedge$ ER $\neq 0$ then skip; CL(AC) := CL(AC) $\oplus$ ER
TLCA	Skip; CL(AC) := CL(AC) $\oplus$ ER

`TDN`	No–op
`TDNE`	If C(AC) $\wedge$ C(E) = 0 then skip
`TDNN`	If C(AC) $\wedge$ C(E) $\neq$ 0 then skip
`TDNA`	Skip
`TDZ`	C(AC) := C(AC) $\wedge \neg$C(E)
`TDZE`	If C(AC) $\wedge$ C(E) = 0 then skip; C(AC) := C(AC) $\wedge \neg$C(E)
`TDZN`	If C(AC) $\wedge$ C(E) $\neq$ 0 then skip; C(AC) := C(AC) $\wedge \neg$C(E)
`TDZA`	Skip; C(AC) := C(AC) $\wedge \neg$C(E)
`TDO`	C(AC) := C(AC) $\vee$ C(E)
`TDOE`	If C(AC) $\wedge$ C(E) = 0 then skip; C(AC) := C(AC) $\vee$ C(E)
`TDON`	If C(AC) $\wedge$ C(E) $\neq$ 0 then skip; C(AC) := C(AC) $\vee$ C(E)
`TDOA`	Skip; C(AC) := C(AC) $\vee$ C(E)
`TDC`	C(AC) := C(AC) $\oplus$ C(E)
`TDCE`	If C(AC) $\wedge$ C(E) = 0 then skip; C(AC) := C(AC) $\oplus$ C(E)
`TDCN`	If C(AC) $\wedge$ C(E) $\neq$ 0 then skip; C(AC) := C(AC) $\oplus$ C(E)
`TDCA`	Skip; C(AC) := C(AC) $\oplus$ C(E)
`TSN`	No–op
`TSNE`	If C(AC) $\wedge$ CS(E) = 0 then skip
`TSNN`	If C(AC) $\wedge$ CS(E) $\neq$ 0 then skip
`TSNA`	Skip
`TSZ`	C(AC) := C(AC) $\wedge \neg$CS(E)
`TSZE`	If C(AC) $\wedge$ CS(E) = 0 then skip; C(AC) := C(AC) $\wedge \neg$CS(E)
`TSZN`	If C(AC) $\wedge$ CS(E) $\neq$ 0 then skip; C(AC) := C(AC) $\wedge \neg$CS(E)
`TSZA`	Skip; C(AC) := C(AC) $\wedge \neg$CS(E)
`TSO`	C(AC) := C(AC) $\vee$ CS(E)
`TSOE`	If C(AC) $\wedge$ CS(E) = 0 then skip; C(AC) := C(AC) $\vee$ CS(E)
`TSON`	If C(AC) $\wedge$ CS(E) $\neq$ 0 then skip; C(AC) := C(AC) $\vee$ CS(E)
`TSOA`	Skip; C(AC) := C(AC) $\vee$ CS(E)
`TSC`	C(AC) := C(AC) $\oplus$ CS(E)
`TSCE`	If C(AC) $\wedge$ CS(E) = 0 then skip; C(AC) := C(AC) $\oplus$ CS(E)
`TSCN`	If C(AC) $\wedge$ CS(E) $\neq$ 0 then skip; C(AC) := C(AC) $\oplus$ CS(E)
`TSCA`	Skip; C(AC) := C(AC) $\oplus$ CS(E)

14.2 Boolean Logic

There are sixteen possible Boolean functions of two single–bit variables. The PDP–10 has sixteen instruction classes (each with four modifiers) that perform these operations. Each Boolean function operates on the thirty–six bits of the accumulator and the thirty–six bits of the memory operand as individual bits.

Each of the sixteen instructions shown in Table 14.1 has four modifiers that specify the memory operand and destination of the result.

A blank modifier means the memory operand is `C(E)`; the result will be stored in the accumulator.

The modifier letter `I` means Immediate. The memory operand is `<0,,ER>`. The result is stored in

	0	1	0	1	C(AC) Contents of the Accumulator
	0	0	1	1	C(E) Contents of Memory
SETZ	0	0	0	0	SET to Zero
AND	0	0	0	1	AND
ANDCA	0	0	1	0	AND with Complement of AC
SETM	0	0	1	1	SET to Memory
ANDCM	0	1	0	0	AND with Complement of Memory
SETA	0	1	0	1	SET to AC
XOR	0	1	1	0	eXclusive OR
IOR	0	1	1	1	Inclusive OR
ANDCB	1	0	0	0	AND with Complements of Both
EQV	1	0	0	1	EQuiValence
SETCA	1	0	1	0	SET to Complement of AC
ORCA	1	0	1	1	OR with Complement of AC
SETCM	1	1	0	0	SET to Complement of Memory
ORCM	1	1	0	1	OR with Complement of Memory
ORCB	1	1	1	0	OR with Complements of Both
SETO	1	1	1	1	SET to One

Table 14.1: Boolean Functions

the accumulator. However, in the extended machine, for SETMI only, the immediate operand is the 30–bit value E; for this purpose, the instruction is called XMOVEI.

M as a modifier means store the result in memory; the accumulator is unaffected.

B as a modifier means store the result in both memory and in the accumulator.

Programming examples:

```
MOVEI   1,1400
IORM    1,577      ;turn on 1400 (Bits 26 and 27) in loc 577

MOVSI   2,2000
ANDCAM  2,600      ;turn off bit 7 in location 600

SETZB   3,4        ;store zero in 3 and 4

SETZM   500        ;store zero in location 500
```

The Boolean instruction set is now presented in detail:

SETZ	C(AC)	:=	0
SETZI	C(AC)	:=	0
SETZM	C(E)	:=	0
SETZB	C(AC)	:=	0; C(E) := 0

`AND`	C(AC)	:=	C(AC) ∧ C(E)
`ANDI`	C(AC)	:=	C(AC) ∧ <0,,ER>
`ANDM`	C(E)	:=	C(AC) ∧ C(E)
`ANDB`	Temp	:=	C(AC) ∧ C(E); C(AC) := Temp; C(E) :=Temp
`ANDCA`	C(AC)	:=	¬C(AC) ∧ C(E)
`ANDCAI`	C(AC)	:=	¬C(AC) ∧ <0,,ER>
`ANDCAM`	C(E)	:=	¬C(AC) ∧ C(E)
`ANDCAB`	Temp	:=	¬C(AC) ∧ C(E); C(AC) := Temp; C(E) := Temp
`SETM`	C(AC)	:=	C(E)
`SETMI`	C(AC)	:=	E; this is `XMOVEI`
`SETMM`	C(E)	:=	C(E)
`SETMB`	C(AC)	:=	C(E); C(E) := C(E)
`ANDCM`	C(AC)	:=	C(AC) ∧ ¬C(E)
`ANDCMI`	C(AC)	:=	C(AC) ∧ ¬<0,,ER>
`ANDCMM`	C(E)	:=	C(AC) ∧ ¬C(E)
`ANDCMB`	Temp	:=	C(AC) ∧ ¬C(E); C(AC) := Temp; C(E) := Temp
`SETA`	C(AC)	:=	C(AC)
`SETAI`	C(AC)	:=	C(AC)
`SETAM`	C(E)	:=	C(AC)
`SETAB`	C(AC)	:=	C(AC); C(E) := C(AC)
`XOR`	C(AC)	:=	C(AC) ⊕ C(E)
`XORI`	C(AC)	:=	C(AC) ⊕ <0,,ER>
`XORM`	C(E)	:=	C(AC) ⊕ C(E)
`XORB`	Temp	:=	C(AC) ⊕ C(E); C(AC) := Temp; C(E) := Temp
`IOR`	C(AC)	:=	C(AC) ∨ C(E)
`IORI`	C(AC)	:=	C(AC) ∨ <0,,ER>
`IORM`	C(E)	:=	C(AC) ∨ C(E)
`IORB`	Temp	:=	C(AC) ∨ C(E); C(AC) := Temp; C(E) := Temp
`ANDCB`	C(AC)	:=	¬C(AC) ∧ ¬C(E)
`ANDCBI`	C(AC)	:=	¬C(AC) ∧ ¬<0,,ER>
`ANDCBM`	C(E)	:=	¬C(AC) ∧ ¬C(E)
`ANDCBB`	Temp	:=	¬C(AC) ∧ ¬C(E); C(AC) := Temp; C(E) := Temp
`EQV`	C(AC)	:=	C(AC) ≡ C(E)
`EQVI`	C(AC)	:=	C(AC) ≡ <0,,ER>
`EQVM`	C(E)	:=	C(AC) ≡ C(E)
`EQVB`	Temp	:=	C(AC) ≡ C(E); C(AC) := Temp; C(E) := Temp
`SETCA`	C(AC)	:=	¬C(AC)
`SETCAI`	C(AC)	:=	¬C(AC)
`SETCAM`	C(E)	:=	¬C(AC)
`SETCAB`	Temp	:=	¬C(AC); C(AC) := Temp; C(E) := Temp
`ORCA`	C(AC)	:=	¬C(AC) ∨ C(E)
`ORCAI`	C(AC)	:=	¬C(AC) ∨ <0,,ER>
`ORCAM`	C(E)	:=	¬C(AC) ∨ C(E)
`ORCAB`	Temp	:=	¬C(AC) ∨ C(E); C(AC) := Temp; C(E) := Temp

SETCM	C(AC)	:=	¬C(E)
SETCMI	C(AC)	:=	¬<0,,ER>
SETCMM	C(E)	:=	¬C(E)
SETCMB	C(AC)	:=	¬C(E); C(E) := ¬C(E)
ORCM	C(AC)	:=	C(AC) ∨ ¬C(E)
ORCMI	C(AC)	:=	C(AC) ∨ ¬<0,,ER>
ORCMM	C(E)	:=	C(AC) ∨ ¬C(E)
ORCMB	Temp	:=	C(AC) ∨ ¬C(E); C(AC) := Temp; C(E) := Temp
ORCB	C(AC)	:=	¬C(AC) ∨ ¬C(E)
ORCBI	C(AC)	:=	¬C(AC) ∨ ¬<0,,ER>
ORCBM	C(E)	:=	¬C(AC) ∨ ¬C(E)
ORCBB	Temp	:=	¬C(AC) ∨ ¬C(E); C(AC) := Temp; C(E) := Temp
SETO	C(AC)	:=	777777,,777777
SETOI	C(AC)	:=	777777,,777777
SETOM	C(E)	:=	777777,,777777
SETOB	C(AC)	:=	777777,,777777; C(E) := 777777,,777777

MACRO also recognizes OR, ORI, etc. as alternative names for the IOR instructions.

14.2.1 Boolean (Logical) Operators in MACRO

Beside the arithmetic operators, MACRO implements a selection of Boolean operators. Although MACRO does not provide the entire repertoire of operations that are present in the PDP–10, any of those operations can be implemented by combinations of the operators that are present. Each operand is a 36–bit quantity; the 36–bit result is derived by applying the same operation to each pair of operand bits.

Boolean Operators in MACRO

Operation	Operator Code	Example		
AND	&	5&11	=	1
OR (inclusive)	!	5!11	=	15
XOR	^!	5^!11	=	14
NOT (ones complement)	^-	^-173	=	777777,,777604

14.3 Example 6–B — Extract Vowels

This program performs the same function as the program in example 6–A. Program control techniques that are quite different from those used in 6–A have been chosen to demonstrate these alternatives. The fundamental difference between this program and the one exhibited in example 6–A is that we use the fact that the function required by this program can be described by two loops (in example 6–A, these are called INLOOP and OLOOP) each having the same structure:

```
        initialize
LOOP:   get a character
        exit from loop if end of line or end of odd buffer
        decide on the disposition of this character
        dispose of this character
        JRST LOOP
```

The structure of `INLOOP` in example 6–A conceals this structure by two calls to `GETCHR`. But some examination of that loop should reveal that it fits into the pattern displayed above. Since we have two loops that have the same essential structure, it is possible to fold these two into one loop by the addition of further instructions to repeat the one loop twice, with whatever modifications are necessary the second time. In this program, the loop at `PSTART` (*Pass START*) is executed twice. The first time, `PSTART` executes with the accumulator called `PASS` containing `-1`; the second time through `PSTART`, `PASS` contains zero.

Instead of synthesizing this program — building it in small steps — we shall present the entire program first, and then analyze it.

```
        TITLE   EXTRACT - Alternate Version - Example 6-B
        SEARCH  MONSYM

Comment $
This program will read a line of text from the terminal.  The output will
be all the even characters followed by the odd characters that are not
vowels, followed by the odd characters that are vowels.  The program will
halt when given an empty line.
Sample session:

Type a line: This is a test of the vowel extraction program.
hsi  eto h oe xrcinpormT sts ftvwltt rg.iaeeaoa
Type a line:

$

A=1                                     ;Temp AC, usually a character
B=2
C=3

D=4                                     ;Control AC.  First pass   Second pass
                                        ;       -1    Even Chr     Vowel
                                        ;        0    Odd Chr      Non-Vowel

INPTR=5                                 ;input buffer pointer
OUTPTR=6                                ;Output line pointer
COUNT=7                                 ;character count in each pass
PASS=10                                 ;pass count.  -1 for first pass,
                                        ;              0 for second pass

P=17                                    ;stack pointer
```

```
OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ P,]

        .PSECT  CODE/RONLY,1002000

START:  RESET                           ;Begin execution here
        MOVE    P,[IOWD PDLEN,PDLIST]   ;initialize a stack pointer
        HRROI   A,[ASCIZ/Welcome to Extract
/]
        PSOUT
NEXT:   HRROI   A,PROMPT                ;request an input line
        PSOUT
        HRROI   A,BUFFER                ;setup for RDTTY
        MOVEI   B,BUFLEN*5-1
        HRROI   C,PROMPT
        RDTTY                           ;obtain one line of input
         ERJMP  ERROR
        MOVE    OUTPTR,[POINT 7,OBUFR]  ;initialize output pointer.
        SETO    PASS,                   ;Initialize for pass 1  (PASS := -1)
;Here to start a pass.
PSTART: MOVE    INPTR,[POINT 7,BUFFER]  ;fetch characters from here
        MOVE    B,INPTR                 ;store odd characters here
        MOVEI   COUNT,1                 ;Character Count on this line
LOOP:   CALL    GETCHR                  ;get a character from the input buffer
        JUMPE   A,PEND                  ;end of input buffer. end of pass
        XCT     DECIDE(PASS)            ;decide how to dispose of character
        XCT     DISPOSE(D)              ;Dispose of this character
        AOJA    COUNT,LOOP              ;increment character count, get another.

;End of a pass
PEND:   CAIG    COUNT,1                 ;skip unless no characters were seen.
         JRST    STOP                   ;no character, stop program.
        IDPB    A,B                     ;store 0 to terminate odd/vow list
        AOJE    PASS,PSTART             ;increment PASS, jump to second pass
        IDPB    A,OUTPTR                ;after 2nd pass, store null (A is zero)
        HRROI   A,OBUFR                 ;send even + odd non-vowels
        PSOUT
        HRROI   A,BUFFER                ;send buffer (odd vowels)
        PSOUT
        HRROI   A,CRLF                  ;send crlf
        PSOUT
        JRST    NEXT                    ;get next line

;The next two instructions are a table used to determine the
;disposition of each character
        CALL    PROC1                   ;D 0 if even;  -1 if odd chr
DECIDE: CALL    PROC2                   ;D 0 if non vowel;  -1 if vowel

;The next two instructions are the table called DISPOSE
                                        ;       Pass 1          Pass 2
        IDPB    A,B                     ;D=-1, ODD CHARACTER    VOWEL
DISPOSE:IDPB    A,OUTPTR                ;D=0, EVEN CHARACTER    Non-Vowel
```

```
;PROC1 is the first pass routine to set up D for the disposal of a character.
;During the first pass,  PROC1 returns D=0 to signal an even character,
;                               D=-1 for an odd character.
;
;The SETD routine is returned to from PROC2.  It is used to set D appropriately
;                               depending on whether the character is a
;                               vowel or not.  See also PROC2 and the text
;
PROC1:  TRNN    COUNT,1                 ;skip if character count is odd
SETD:    TDZA   D,D                     ;set D to zero (for even) and skip
          SETO  D,                      ;set D to -1 (odd)
        RET

;Get the next character into A.  INPTR is the pointer to the current input line.
;return a null in A to signal end of line.
GETCHR: ILDB    A,INPTR                 ;Get the next character
        CAIE    A,12                    ;test for end of line
         CAIN   A,15                    ;   either a return or a line feed
          MOVEI A,0                     ;   becomes a null.
        RET

;Process characters on second pass.  PROC2 will cause ISVOW to return
;to SETD (no vowel, D := 0) or to SETD+1 (character is a vowel, D := -1).
;
;ISVOW is prepared to be called by a PUSHJ; it will skip if the character
;in A is a vowel.  Otherwise, no skip.
PROC2:  XMOVEI  C,SETD                  ;get the 30--bit address of SETD
        PUSH    P,C                     ;store SETD as the return address
ISVOW:  MOVE    C,A                     ;copy character to C
        CAIL    C,"a"                   ;test to see if C is lowercase
         CAILE   C,"z"
          TRNA                          ;C is not lower case: skip
           TRZ  C,40                    ;convert lower-case to UPPER
        MOVSI   D,-VOWTLN               ;-length of vowel table,,0
ISVOW1: CAMN    C,VOWTAB(D)             ;Does character match a vowel?
CPOPJ1:  AOSA    (P)                    ;Yes.  Set skip return, exit loop
          AOBJN   D,ISVOW1              ;not yet. Loop unless at end of table
CPOPJ:  RET                             ;return

VOWTAB: EXP "A","E","I","O","U","Y"
VOWTLN==.-VOWTAB

ERROR:  HRROI   A,[ASCIZ/Error in RDTTY
/]
        PSOUT
STOP:   HALTF
        JRST    STOP

PROMPT: ASCIZ   /Type a line: /
```

```
        .PSECT  DATA,1001000
BUFLEN==40
BUFFER: BLOCK   BUFLEN
OBUFR:  BLOCK   BUFLEN
PDLEN==40
PDLIST: BLOCK   PDLEN
CRLF:   BYTE    (7)15,12
        END     START
```

14.3.1 Analysis of Program 6–B

The portions of the program at `START` and `NEXT` should, by now, require no explanation. In these areas, this program is quite similar to example 6–A and earlier examples.

14.3.1.1 Two–Pass Structure

Let us start with the fragment that controls the two passes through `PSTART`.

```
        MOVE    OUTPTR,[POINT 7,OBUFR]  ;initialize output pointer.
        SETO    PASS,                   ;Initialize for pass 1  (PASS := -1)
PSTART: . . .                           ;start a pass.

PEND:   . . .                           ;end a pass

        AOJE    PASS,PSTART             ;increment PASS, jump to second pass
```

We initialize the output buffer pointer, `OUTPTR`, and the pass counter, `PASS`, before entering the loop at `PSTART`. For the first pass, the pass counter is set to `-1` by the instruction `SETO PASS,` that appears before `PSTART`. Note that the comma in the `SETO` instruction is very important; it places `PASS` in the accumulator field. The `SETO` instruction sets an accumulator to all ones (i.e., to `-1`); `SETO` does not affect the location specified by its effective address.

The `AOJE` instruction at `PEND`, *Pass END*, will increment `PASS` and jump back to `PSTART` to start another pass provided that `PASS` becomes zero when it is incremented. Thus, `PSTART` is executed with `PASS` containing `-1` and then with `PASS` containing `0`. After the second pass, `PASS` is incremented to `1` and the `AOJE` instruction will not jump a second time.

14.3.1.2 Inner–Loop Instructions

The instructions inside each pass are also interesting:

```
PSTART: MOVE    INPTR,[POINT 7,BUFFER]  ;fetch characters from here
        MOVE    B,INPTR                 ;store odd characters here
        MOVEI   COUNT,1                 ;Character Count on this line
LOOP:   CALL    GETCHR                  ;get a character from the input buffer
        JUMPE   A,PEND                  ;end of input buffer. end of pass
        XCT     DECIDE(PASS)            ;decide how to dispose of character
        XCT     DISPOSE(D)              ;Dispose of this character
        AOJA    COUNT,LOOP              ;increment character count, get another.
```

```
;End of a pass
PEND:   CAIG    COUNT,1                 ;skip unless no characters were seen.
         JRST   STOP                    ;no character, stop program.
        IDPB    A,B                     ;store 0 byte to terminate odd/vow list
        AOJE    PASS,PSTART             ;increment PASS, jump to second pass
```

At the start of each pass, INPTR is set up to point to BUFFER. On the first pass, BUFFER contains the input line. Accumulator B is initialized to point to BUFFER also. On pass one, B is used as the deposit pointer for the odd characters. The same trick that we first demonstrated in example 6–A is used again: characters are fetched from and stored into the same buffer area. INPTR is the *take* pointer; B is the *put* pointer. Examination of the loop reveals that the take pointer will never fall behind the put pointer, so our recycling of buffer space is without hazard.

Another accumulator, COUNT, is initialized to 1. This register will be used to count the characters in the line as they are processed; at LOOP, COUNT contains the character number of the next character to be processed.

Inside the loop at LOOP, GETCHR is called to obtain a character from the input buffer. GETCHR returns in A the next available character from the input buffer; when no further characters are available, 0 is returned in A. When the input line is exhausted, the program will jump to PEND to terminate this pass.

Once a character has been returned by GETCHR, the program must decide what to do with it, and then dispose of it. After each character is disposed of, the program executes the instruction AOJA COUNT,LOOP to advance COUNT and repeat the loop.

The instruction XCT DECIDE(PASS) will make the determination of what to do with each character. Note that PASS appears as an index register in this instruction. On the first pass, when PASS contains -1, the effective address computed for the execution of this instruction is DECIDE-1. Recall that the XCT instruction performs the instruction found at its effective address. Thus, on pass one, the instruction located at DECIDE-1 is performed to make the decision about each character.

The instruction contained at DECIDE-1 is CALL PROC1. The PROC1 subroutine will set register D to zero if this is an even character; otherwise it will set D to -1 signifying an odd character. Before we examine the instructions in PROC1, let us finish the code at LOOP. Assuming that PROC1 works as described, the next instruction is XCT DISPOSE(D) [1]. Note that the result in D, as returned by PROC1, is used to modify this instruction. If this is an odd character, D will contain -1 and the instruction at DISPOSE-1 will be executed. That instruction is IDPB A,B; the odd character will be deposited in the buffer for odd characters. If this is an even character, D will contain zero; the instruction at DISPOSE will be executed. That instruction, IDPB A,OUTPTR, sends the even character to the output buffer.

The student should review the explanation of the instructions at LOOP thus far. The decision about each character is made by executing an instruction in the table DECIDE; during the first pass, that decision is always made by the PROC1 subroutine. After each decision the character is disposed of by executing one of the instructions in the DISPOSE table. The disposition consists of depositing odd characters into the buffer for odd characters, addressed by B, or depositing even characters into the output buffer, addressed by OUTPTR.

When the end of the line is seen in pass one, the program jumps to PEND. There, the character count, COUNT is tested. If COUNT is greater than one, then characters were present in the input line;

[1] The seven–letter name, DISPOSE, is longer than allowable symbol names; MACRO will shorten the name to DISPOS.

otherwise, the line is empty and the program jumps to STOP. Assuming the line was not empty, the IDPB A,B instruction deposits a zero byte to end the buffer of odd characters. PASS is incremented to zero, and the program jumps to PSTART where the second pass is started.

For the second pass, registers INPTR and B are again initialized to point to BUFFER. Again, INPTR is the take pointer; it takes odd characters. Also, B is used again as the put pointer; it deposits odd vowels.

The description of the action of the code at LOOP is the same for the second pass; the difference is that the instruction XCT DECIDE(PASS) calls the subroutine PROC2 to make the decision about each character. The PROC2 subroutine will return 0 in D for non–vowels, and return -1 in D for vowels. The disposition of characters is the same as in the first pass: when D is 0, non–vowel characters are sent to the output buffer; when D is -1, vowels are stored in the vowel buffer. At the end of the second pass, a zero byte is deposited into the vowel buffer to end it; the AOJE instruction does not jump; the program falls into the output routines where OBUFR, BUFFER, and a carriage return and line feed are printed.

14.3.1.3 PROC1 and PROC2 Subroutines

The PROC1 subroutine is not too complicated:

```
PROC1:  TRNN    COUNT,1         ;skip if character count is odd
SETD:    TDZA   D,D             ;set D to zero (for even) and skip
          SETO  D,              ;set D to -1 (odd)
        RET
```

In PROC1 the instruction TRNN COUNT,1 tests bit 35 of the character count. Bit 35 of COUNT will be one when the program is processing an odd character; it will be zero while processing an even character. The TRNN instruction will skip if bit 35 is a one. If bit 35 is a zero, the TRNN will not skip; the following instruction, TDZA D,D will be executed. Careful reading of the description of the test instructions should convince you that this TDZA will set register D to 0 and skip. If the TRNN skips, the instruction SETO D, is executed; this sets D to -1. Finally, PROC1 returns to its caller. In summary, if COUNT is odd, D is set to -1; when COUNT is even, D is set to 0.

The instruction sequence at PROC2 introduces a new concept. Let us start with a more obvious version of PROC2:

```
PROC2:  CALL    ISVOW               ;Test for vowel.  Skip if Vowel
         TDZA   D,D                 :Not a vowel.  Set D to 0 and skip
          SETO  D,                  ;Vowel.  Set D to -1
        RET
```

This version of PROC2 is similar in structure to PROC1. The ISVOW subroutine (which we describe in detail below) will decide if the character in A is a vowel. If A contains a vowel, ISVOW will skip. Essentially, the instruction CALL ISVOW can be thought of as a conditional skip, like the TRNN instruction in PROC1. The non–skip return from ISVOW causes the instruction TDZA D,D to be executed. The skip return causes the SETO to be executed. When the character in A is a vowel, -1 is returned in D; otherwise, D is set to 0.

Now, this isn't the version of PROC2 that we have used in this program. To understand what we have done, it is necessary to review the details of the PUSHJ instruction. Recall that PUSHJ pushes

the return address onto the stack and jumps. If we use a PUSH instruction to push a data item onto the stack, and then jump (e.g., by means of JRST) to a subroutine, we have simulated the effects of PUSHJ. When the subroutine eventually executes a POPJ (or as we call it, a RET) instruction, then the data item that we pushed will be taken as the return address by the POPJ instruction.

The return address that we push is the address of SETD. This is the address of the TDZA, SETO sequence that follows PROC1.

In the traditional (or section 0) we could rewrite PROC2 as:

```
PROC2:  PUSH    P,[SETD]                ;save return address
        JRST    ISVOW                   ;jump to the ISVOW routine
```

ISVOW will now return to SETD or to SETD+1. We have avoided the repetition of the TDZA, SETO, and RET that was present in our first version of PROC2.

However, in the extended machine, we need the 30–bit address of SETD on the stack, but MACRO will use the 18–bit in–section address of SETD if we write the literal as shown above. We could write

```
PROC2:  PUSH   P,[1,,SETD]
```

which uses our knowledge that the code psect will be placed in section 1. However, this is "cheating" and it can produce an erroneous result if someone should reassign the psect to start in section 34. So, we adopt the following instead:

```
PROC2:  XMOVEI  C,SETD
        PUSH    P,C
```

This sequence forces the correct section number into the left half of C and then pushes that value on the stack as the return address. Accumulator C was chosen by peeking ahead to see what ISVOW would do.

Finally, we eliminate the JRST ISVOW in PROC2 by the trivial expedient of moving PROC2 to immediately precede ISVOW:

```
PROC2:  XMOVEI  C,SETD
        PUSH    P,C
ISVOW:  . . .
```

14.3.1.4 ISVOW Subroutine

The other area that is changed is at ISVOW. A copy of the character is made in register C. By means of nested skips, the program decides whether the character is lower–case; a lower–case character is greater than or equal to "a" and less than or equal to "z". If C contains a lower–case character, that character is changed to upper–case by the instruction TRZ C,40. From the table of ASCII characters, you can see that a lower–case character differs from the corresponding upper–case one by having an additional 40 in the character code. For example, the code for "I" is 111, and that for "i" is 151. By turning off the bit with value 40 (bit 30 in register C) a lower–case letter can be transformed into an upper–case one.

The AOBJN instruction is used to cycle through a table of vowels, comparing the input character to

the table. If a match is seen, the program effects a skip by executing the instruction at CPOPJ1. Otherwise, a non–skip return is taken.

The loop control in the subroutine is accomplished with an AOBJN instruction. Accumulator D is set up via the MOVSI instruction that occurs before the loop ISVOW1. As we will show later, the symbol VOWTLN, *VOWel Table LeNgth*, has value 6. So, D is initialized to -6,,0. The instruction at ISVOW1 compares C to VOWTAB(D). The effective address of this instruction is modified by the right half of D.[2] Initially the right half of D is 0, so the first address compared to is just VOWTAB+0. If the character in C matches VOWTAB+0, the CAMN will not skip, and the instruction at CPOPJ1 is executed. Assuming that C contains something other than the letter "A", the CAMN will skip. The instruction AOBJN D,ISVOW1 will increment both halves of D; D becomes -5,,1. Since the resulting value is negative, the AOBJN instruction will jump back to ISVOW1.

At ISVOW1, this time, D contains -5,,1. So the CAMN references VOWTAB+1. If register C is not an E, the CAMN will skip to the AOBJN. This process continues; if a vowel is present in register C, eventually, the CAMN instruction will find it and avoid skipping; the instruction at CPOPJ1 will be executed. If no vowel is present, eventually, D will contain -1,,5 while ISVOW1 tests to see if C contains the character Y. If the CAMN skips, the AOBJN instruction will increment D to 0,,6. Since this is now a positive number, the AOBJN will not jump back to the ISVOW1; the subroutine executes RET and returns without skipping.

The instruction at CPOPJ1 will increment the return address and skip over the AOBJN to execute the RET, which returns with one skip.

14.3.1.5 The EXP Pseudo–Op

At the label VOWTAB we introduce another of MACRO's pseudo–ops, EXP. This operator takes a list of values and places each one in consecutive words. Otherwise, the six expressions, "A" ... "Y" could have been written on six separate lines.

14.3.1.6 Symbolic Length of a Table

The symbol VOWTLN takes its value from the assignment VOWTLN==.-VOWTAB. The period symbol (".") in the assembler represents the current value of the location counter. After assembling the word containing the character Y (which MACRO places at VOWTAB+5), the location counter will be advanced to point to the next word, VOWTAB+6. The expression .-VOWTAB then has the value 6. This is a neat way to find out how many entries there are in a table. It is especially useful because if something is added to the table the program adjusts itself to the new entry. Thus, the programmer avoids having to search through the program for those instances of the number 6 that signify the count of entries in this table.

14.3.1.7 The BYTE Pseudo–op

This program introduces another pseudo–operator, BYTE, which can be used to generate data words composed of arbitrary fields. In the example that was given, the word at CRLF was defined by:

```
CRLF:   BYTE(7)15,12
```

[2]Only the right half of an index register is significant in address calculations (except see Section 30, page 565).

This defines a word composed of two 7–bit bytes (the remainder of the word has not been specified and is assembled as zero). The byte size of the field was set by the number 7 in parentheses following the word BYTE. The data for each field is supplied by the number or list of numbers that follows the byte size. In this case the data are octal values 15 and 12; these represent carriage return and line feed, respectively.

The BYTE pseudo–op can handle more than one byte size. For example, an instruction word can be defined by the description:

```
BYTE (9)OP (4)AC (1)I (4)X (18)Y
```

Here, the value of the symbol OP will be assembled into bits 0:8 of the word, the opcode field. The value of AC will be assembled into the next four bits, 9:12, etc.

Note that the numbers inside parentheses, the byte sizes, are interpreted as decimal numbers. The data arguments are interpreted in the prevailing radix (usually octal).

When a list of data arguments, all with the same byte size, is written, commas appear between the data items. Note, however, that no comma appears before a new byte size. For example:

```
BYTE (11)X,Y (3)BRT,SIZE (2)MODE,TYPE (4)6
```

In this example, the value of X will be placed in an 11–bit byte, bits 0:10. The value of Y also will be placed in an 11–bit byte, bits 11:21. The fields BRT and SIZE are each three bits, bits 22:24 and 25:27, respectively. The MODE and TYPE fields are two bits each, 28:29 and 30:31. Finally the constant value 6 is placed into the four bits 32:35.

In a BYTE pseudo–op, if a field cannot fit into the remainder of a word, a second (or subsequent) word is started; that field then appears in the left–most byte of the new word. Any unused bits will be zeroed.

14.4 Exercises

14.4.1 Pig Latin

Write a program that will input a line of text and translate it to pig latin. The rules for pig latin are:

- If a word begins with a vowel, then the translation of the word is the word itself, unchanged.
- If a word begins with a consonant, then the translation of that word consists of three parts:
  1. the first vowel in the word, followed by all letters that followed the first vowel in the English word.
  2. the first consonant, followed by all letters up to but excluding the first vowel.
  3. the letters AY.
- If there are no vowels in the word, output the word and the letters AY.

Your program should work for lines of text where spaces or punctuation separate the words. Perhaps the best way to think of this is that letters are parts of words, and anything other than a letter is a word delimiter.

Your program should be able to accept either upper–case, lower–case, or mixed text. The case that you select for your output is not important.

Words with internal punctuation, such as "don't", need not be handled properly.

Example:

```
Input:  I said, ‘‘Oh, what a strange homework assignment!’’
Output: I aidsay, ‘‘Oh, atwhay a angestray omeworkhay assignment!’’

Input:  this is an example of pig latin.
Output: isthay is an example of igpay atinlay.
```

Note that the punctuation has been preserved.

Chapter 15

Block Transfer and Shift Instructions

We continue our exposition of the PDP–10 instruction set by introducing the block transfer instruction, `BLT`, and the various shift instructions.

15.1 BLT Instruction

The `BLT` (BLock Transfer) instruction copies a block of consecutive words from one area of memory to another. In order to specify such a copying operation, three items of information must be given:

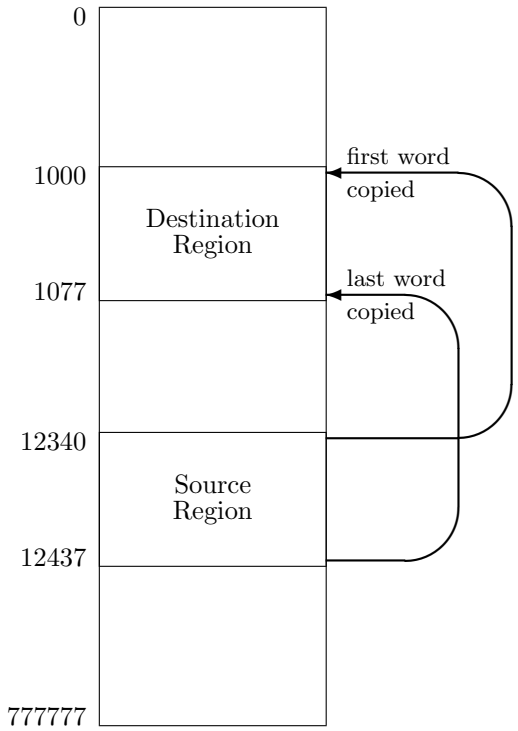
- The first source location,
- the first destination location, and
- how many words to copy.

The accumulator in the `BLT` instruction must be set up to contain the first two of these items of information. The length of the transfer does not appear explicitly; instead, it is encoded by specifying the address of the last word to store into. The final destination address (the last word to store into) appears in the effective address of the `BLT` instruction.

More specifically, the left half of the `BLT` accumulator specifies the first source (in–section) address. The right half of the accumulator is the first destination (in–section) address. The effective address of the `BLT` is the last destination address. In the extended machine, the effective address of `BLT` specifies the section number affecting both the source and destination addresses. Words are copied, one by one, from the source to the destination, until a word is stored at an address greater than or equal to the effective address of the `BLT`. Figure 15.1 shows an example of the BLT instruction.

The `BLT` instruction is somewhat complicated. `BLT` has a loop inside it that controls the copying process. Essentially, you may think of `BLT` as copying from the first source to the first destination. Then both halves of the accumulator are incremented to address the second source and second destination words. The copying process repeats until a word is stored at or above the location specified by the effective address.

`BLT` updates the `AC` as it progresses. Thus, if the instruction were to be interrupted, the `BLT` could restarted, whereupon it actually continues from where it left off. In the KI10 and prior machines,



Copy `100` (octal) words
from locations `12340:12437`
to locations `1000:1077`.

```
MOVE  15,[12340,,1000]
BLT   15,1077
```

Load an accumulator (`15` in this example) with the first source address, `12340`, in the left half and the first destination address, `1000`, in the right half. The effective address, `1077`, the the address of the *last* word to store into. In this example, data are local to the program's address section.

Figure 15.1: BLT Example

the updating of `AC` was left to chance. In the KL10 and later machines, `BLT` deliberately updates `AC` to show the next source address and the next destination address that would have been copied had the `BLT` not stopped itself.

15.1.1 Warnings about BLT

Because there are various things happening during `BLT`, there are circumstances that must be avoided when using a `BLT`. Despite the following list of warnings, `BLT` is quite useful when it is used carefully.

- `BLT` modifies its accumulator. On the KL10, KS10, and TOAD processors, when `BLT` is finished, the accumulator is set to the next source and next destination addresses that would have been transferred. On the earlier processors the contents of the AC are indeterminate after a `BLT` that moves more than one word.
- Because `BLT` modifies its accumulator, you must never allow the `BLT` accumulator to be used as an index register in the address calculation of the `BLT`.

  The reason for this is a little involved. A time–sharing computer system must be able to respond promptly to external events; a long wait for the loop inside `BLT` to finish cannot be tolerated. Therefore, `BLT` has been designed so that its execution can be interrupted between each word that it moves. When a `BLT` instruction is interrupted, it will store an updated source and destination pointer in the accumulator. Because `BLT` updates this pointer, it can resume its execution without repeating any work that it did previously. When the execution of an interrupted `BLT` is resumed, the effective address is calculated again. If the accumulator

that is used by BLT is also used as an index register in the effective address calculation, then a different effective address will be used when the BLT instruction is resumed.

- A BLT that changes anything having to do with its own effective address calculation will be unpredictable. If indexing is used, don't let the index register be among the destination addresses.
- If the source and destination addresses overlap, remember that BLT moves the lowest source word (that is, the source word with the lowest address) first.
- If the destination of the BLT includes the accumulator of the BLT, then the BLT accumulator must be the last destination address.
- If the destination of the BLT includes the BLT instruction itself, then that BLT should be at the last destination address.
- If the effective address of BLT is global, both the source read and the destination store will be to global addresses. Thus, if the source or destination address is in the range 0--17 and the global section is larger than 1, such addresses refer to memory, not to the accumulators.

15.1.2 BLT Programming Examples

We offer a variety of programming examples to demonstrate BLT.

15.1.2.1 Save and Restore Accumulators (Static Local)

The accumulators are very important in programming the PDP–10. Often, their contents must be saved and later restored. An example of this would be a subroutine that wants to avoid changing any of the accumulators in the main program. The BLT instruction is useful for saving and restoring the accumulators.

Define a storage area for accumulators called SAVAC.[1]

```
SAVAC:  BLOCK   20
```

Save all the accumulators in 20 words starting at SAVAC.

```
        MOVEM   17,SAVAC+17   ;save one accumulator first
        MOVEI   17,SAVAC      ;Source is 0, destination is SAVAC
        BLT     17,SAVAC+16   ;Store through SAVAC+16
```

Restore all the accumulators from SAVAC. Note that this stores into the BLT accumulator, but it does so as the last word that is moved.

```
        MOVSI   17,SAVAC      ;Source is SAVAC, destination is 0
        BLT     17,17         ;Store through 17
```

[1] Presume this area is in the same address section as the program.

15.1.2.2 Save and Restore Accumulators (Stack, Global)

We repeat the previous example, but this time we propose to save the accumulators on the stack. Moreover, we assume that the stack is *not* in the same address section as the program. (This implies that a global format stack pointer is being used.) This situation is typical of a programming style in which "local variables" are allocated on the stack when a subroutine is entered and deallocated on exit. (We note that some words that we use are overloaded with conflicting meanings: local variables are private to the routine that creates them, regardless that we need a "global address" to locate them.)

In this example, we presume that `P` is register `17`. This routine doesn't save `P`: the program must maintain `P` as a valid stack pointer in order to exit from the subroutine.

This is a complicated example that exposes the subtleties of extended addressing. The explanation follows the two code fragments.

```
ADJSP   P,20            ;allocate space on stack for 0-16 and 1 extra
MOVEM   16,-1(P)        ;Store one register for use as the BLT AC
MOVE    16,[BLT 16,0]   ;Partial image of instruction to XCT
HRRI    16,-2(P)        ;Set to BLT 16,<local address of last word>
MOVEM   16,0(P)         ;Put BLT on the stack. local to stack's section
MOVEI   16,-17(P)       ;Source is 0, destination is stack
XCT     0(P)            ;perform BLT, addresses local to stack section
ADJSP   P,-1            ;discard the BLT instruction
```

The accumulators are restored as follows:

```
PUSH    P,[BLT 16,16]   ;Put instruction to XCT in stack section
MOVSI   16,-17(P)       ;source is stack, destination is AC 0
XCT     0(P)            ;perform BLT, addresses local to stack section
ADJSP   P,-20           ;deallocate space for ACs and BLT instruction
```

`BLT` is restricted to moving blocks of data within one section. The section is determined by the effective address of `BLT`. However, the most usual way to specify a remote section, e.g., `BLT 16,-1(P)`, results in an effective address that is global. The address property "global" would then be applied to both the source and destination addresses. So if the stack were in section `13`, a source address such as `0` would refer to global `13,,0`, a memory location in section `13`, not an accumulator. So, we cannot use a global address in the effective address of `BLT` to refer to the stack section.

Instead, we use the property of `XCT` that calls for the effective address computation of the target instruction to begin as a local address in the section from which the target instruction was read. Thus, if we can put the correct `BLT` instruction somewhere in the stack section, then its effective address calculation can produce a local result in the stack section.

Let us suppose that `P` initially contains the global stack pointer `13123100`. The code to store the `AC`s on the stack begins by allocating `20` locations on the stack, `17` for accumulators `0--16` and one that will store the `BLT` instruction temporarily. After the `ADJSP` to allocate space, `P` is advanced to `13123120`. We intend to store accumulators `0--16` in locations `12123101`–`12134117`, respectively. We refer to those locations as `-17(P)`–`-1(P)`, respectively. Next, we store `16` on the stack at `-1(P)`; this frees `16` to be the `BLT` accumulator. However, before we use it as such, we create the `BLT` instruction in it. We start by loading `BLT 16,0` into `16`. Then by `HRRI 16,-2(P)` we change the right half of `16`, the `Y` field of the `BLT` instruction to be the intended ending address, local `123116`;

this is the (in–section) address to which we intend 15 to be copied.

The resulting instruction in 16 is

```
BLT 16,123116
```

Note the difference between this and

```
BLT 16,-2(P)
```

As we mentioned above, if BLT 16,-2(P) were executed it would have a global effective address 13123116. When BLT 16,123116 is executed, it will have a local effective address, in–section 123116.

The instruction BLT 16,123116 is stored on the stack by MOVEM 16,0(P). This places the BLT instruction in memory at 13123120.

Accumulator 16 is initialized with the source and destination addresses: the source is zero, the destination is in–section 123101 that was obtained via the immediate address of -17(P). Then, XCT performs the instruction found at the stack top. XCT reads location 13123120 and finds BLT 16,123116 there. The effective address of this instruction is local 123116 in section 13. Because the address is local, references to source addresses 0–15 are references to the accumulators; because the effective address is in section 13, references to destination addresses 123101–123116 are made to the corresponding addresses in that section. Thus, the accumulators are copied to the stack.

Finally, the stack location that contains the BLT is discarded. The body of the subroutine is entered with P containing 13123117.

Restoring the accumulators from the stack is just a bit easier: we don't have to calculate the ending address of BLT, it is the constant 16. Assume that when we arrive at the restore code P contains 13123117. The instruction BLT 16,16 is pushed on the top of the stack (advancing P). The left half of accumulator 16 is initialized to the in–section address of the word to restore from; symbolically, this is the immediate value of -17(P). The right half of 16 is zero, the first destination address. XCT causes the BLT to be performed relative to local addresses in section 13. After the BLT, the space occupied by the saved accumulators and the BLT is deallocated. This restores the stack pointer the value it had at the beginning of the accumulator save fragment, 13123100.

15.1.2.3 Clearing Memory (Setting a Pattern)

Another use for BLT is clearing memory to zero, or storing some other pattern in consecutive memory words. In this example, the program will store zero in 100 words starting at TABLE. We start by using the SETZM instruction (set zeroes to memory) to store a zero at TABLE+0.

Then, by overlapping the source and destination addresses we use BLT to propagate this zero throughout the array TABLE. The first source address is TABLE, the first destination address is TABLE+1. The BLT instruction copies the zero at TABLE to TABLE+1. Then it increments the source and destination addresses, and copies the word at TABLE+1 to TABLE+2. This word is zero, so the BLT instruction keeps moving this zero to higher addresses. The BLT stops when TABLE+76 is copied to TABLE+77.

```
SETZM   TABLE                   ;Set the first to zero
MOVE    AC,[TABLE,,TABLE+1]     ;Source and
BLT     AC,TABLE+77             ; destination overlap
```

A two–word pattern can be replicated in memory by setting the first destination to two words above the first source address. Of course, this technique can be applied to replicate any fixed–size pattern.

15.1.2.4 Backwards Block Move

In this next example, we want to move 76 words from TABLE through TABLE+75 to TABLE+2 through TABLE+77. The BLT instruction cannot be used in this example because the source and destinations overlap in the wrong way.

To perform this copy, we must start by moving TABLE+75 to TABLE+77.[2] This makes room at TABLE+75 for a word (TABLE+73) to be stored there. Of course, we move TABLE+74 to TABLE+76 before moving TABLE+73, but the main idea is that we have to start at the highest source address and move data upwards to the highest destination address. There is no single instruction to do this function; we resort to a loop.[3]

The trick here is to use the POP instruction because POP, unlike most instructions, can move data between two memory locations. The locations are not arbitrary; one location is specified by a stack pointer. We start with a stack pointer that addresses TABLE+75 as the stack top. If we execute a POP instruction, the word at TABLE+75 will be read from the stack top. Now, where to put it?

There is a constant offset of 2 from the stack top to the desired destination address. That is, TABLE+75 is copied to TABLE+77, TABLE+0 is copied to TABLE+2, etc. We can use indexed addressing to help us:

```
        MOVE    A,[400075,,TABLE+75]
LOOP:   POP     A,2(A)          ;Store TABLE+75 into TABLE+77, etc.
        JUMPL   A,LOOP          ;Jump until 76 words have moved.
```

The control count in the left half of the stack pointer is used to tell the JUMPL when to stop.

The first time through LOOP the word at TABLE+75 is popped into TABLE+77. The stack pointer is changed to 400074,,TABLE+74. The second time through LOOP, TABLE+74 is copied to TABLE+76. The last time through the loop, the stack pointer contains 400000,,TABLE. The word at TABLE is copied to TABLE+2; the stack pointer is changed to 377777,,TABLE-1. The JUMPL will not jump.

This use of a stack pointer is completely unrelated to the normal use of stacks or stack instructions; this is an example of the way that instructions can be used for purposes other than the obvious or usual ones.

15.2 XBLT Instruction

The XBLT instruction adds flexibility to the data movement of BLT. XBLT can move data between different address sections. XBLT can move the lowest address in a region first, as BLT does, called a *forward* block transfer. XBLT can move the highest address in a region first, called a *backward* block transfer.

The block size, the location of the source region, and the location of the destination region are defined by the contents of a block of three consecutive accumulators as shown in Figure 15.2. We refer to the three consecutive accumulators as AC, AC+1, and AC+2, where the sums are understood to be taken modulo 20_8.

XBLT performs either a forward or a backward block transfer as follows:

[2] Data are assumed to be local to the program's address section.

[3] Subsequently, the XBLT instruction was invented, rendering this example moot. However, you might find this in an old program.

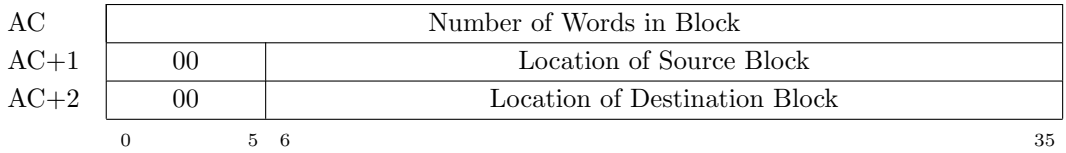


Figure 15.2: Accumulator Usage in XBLT

- If `AC` contains a positive number N, move a block of N words from a source area beginning at the location specified by `AC+1` to a destination area beginning at the location specified by `AC+2` and extending through increasing addresses. At the end, `AC` is cleared to zero, and `AC+1` and `AC+2` respectively contain addresses `1` greater than those of the final source and destination locations referenced.
- If `AC` contains a negative number $-N$, move a block of N words from a source area beginning at a location `1` less than that specified by `AC+1` to a destination area beginning at a location `1` less than that specified by `AC+2` and extending through decreasing addresses. At the end, `AC` is cleared to zero, and `AC+1` and `AC+2` respectively contain the addresses of the final source and destination locations referenced.

The contents of `AC+1` and `AC+2` are interpreted as 30–bit global addresses. This instruction is legal in section zero, and it can reference addresses in non–zero sections when executed in section zero.

(Figure 15.2 notwithstanding, neither the KL10 microcode nor the TOAD–1 microcode enforces the requirement that bits `0:5` of `AC+1` and bits `0:5` of `AC+2` be zero; in these processors, those bits are preserved.)

Under no circumstances should any of the three accumulators be part of either the source or destination block. Because of the possibility of an interrupt or page failure, the contents of these accumulators, even as a source, cannot be guaranteed. `BLT` can store (or load) the accumulators to (or from) any section.

The `XBLT` is one of the instructions executed under `EXTEND`. In code it would appear as

```
EXTEND  AC,[XBLT]
```

where `AC` names the first of the three accumulators that supply parameters to `XBLT`. The I, X, and Y fields in the word containing `XBLT` must be zero.

15.3 Shift Instructions

The following instructions shift or rotate the accumulator or the double word formed by `AC` and `AC+1`. The number of places to shift is specified by the effective address, which is considered to be a signed number modulo decimal `256` in magnitude. That is, the effective shift is the number composed of bit `18` (the sign) of the effective address and bits `28:35` of the effective address. If `E` is positive, a left shift occurs. If `E` is negative, a right shift occurs.

15.3.1 LSH – Logical Shift

The contents of the selected accumulator are shifted as specified by the effective address, E. Zero bits are shifted into the accumulator, as depicted in Figure 15.3.

As you are now aware, MACRO is willing to perform arithmetic using the values of symbols and constants that are known while it is assembling the program. Beside the usual arithmetic operators, MACRO has a logical shift operator, written as an underscore character. Thus the expression 5_11 represents the value 5 left–shifted by nine (octal 11) places, which is equivalent to 5000. To perform a right shift, make the second operand negative.

15.3.2 LSHC – Logical Shift Combined

Combined mode shifts involve the double–word accumulator pair formed from AC and AC+1 as shown in Figure 15.4. The contents of the double–word accumulator, denoted by C(AC AC+1), are shifted as a 72–bit quantity. Zero bits are shifted in. Note that when we speak of AC+1, we mean the accumulator selected by the address (AC+1) modulo 20 (octal). That is, if AC is 17 then AC+1 is accumulator 0.

We offer a short example to demonstrate the LSHC instruction. A subset of the ASCII code is used for various purposes in TOPS–10 and TOPS–20. In this subset, for compactness, each character is stored in six bits instead of seven. This subset of ASCII is called SIXBIT in the PDP–10. In SIXBIT, the ASCII codes in the range octal 40 to 137 are mapped into the range 0 to 77.

The word in register B, representing up to six characters in the SIXBIT code, is translated by this subroutine to a sequence of ASCII characters which are printed:

```
A=1                             ;For PBOUT, A must be 1
B=2                             ;For LSHC, B must be A+1

;Enter with B containing a SIXBIT word.  Convert to ASCII and print
SIXOUT: JUMPE   B,[RET]         ;Return when only blanks remain
        MOVEI   A,0             ;Clear the accumulator on the left
        LSHC    A,6             ;Left-shift a character out of B to A.
        ADDI    A,40            ;Convert from SIXBIT to ASCII
        PBOUT                   ;Print the ASCII character
        JRST    SIXOUT          ;Loop until B is empty.
```

15.3.3 ASH – Arithmetic Shift

In an arithmetic shift, bit 0 is the sign bit; the sign is preserved. In a left–shift, zero bits are shifted into the right end of the accumulator. In a left–shift, if any bit of significance is shifted out of bit 1, then the AROV flag is set to signify that an arithmetic overflow has occurred. (A *bit of significance* is a bit whose value differs from that of the sign–bit.) In a right–shift, bit 0, the sign–bit, is copied into bit 1; this is called sign–extension: providing non–significant bits at the left end of the accumulator. See Figure 15.5.

The ASH instruction can be used to multiply an integer by a power of 2. For example, ASH AC,1 will double the contents of AC. A right–shift will divide a positive integer by a power of 2. A right arithmetic shift of a negative integer does not produce the same result as the corresponding divide operation. For example if a register containing -5 is shifted right one place, the result will be -3,

in contrast to the -2 that would be obtained from a divide instruction. When -1 is right–shifted arithmetically, the result is -1.

15.3.4 ASHC – Arithmetic Shift Combined

In the ASHC instruction, bit 0 of the specified accumulator provides the sign of the result. This bit remains unchanged. If the effective address, E, is non–zero then bit 0 of AC will be copied to bit 0 of AC+1. C(AC AC+1) is shifted as a 70–bit quantity, as shown in Figure 15.6. In a left–shift, zero bits are shifted into the right end of AC+1. In a left–shift, if any bit of significance is shifted out of AC bit 1 then the AROV flag will be set. In a right–shift, AC bit 0, the sign–bit, is extended into AC bit 1.

15.3.5 ROT – Rotate

The 36–bit contents of the selected accumulator are rotated as shown in Figure 15.7. In a left–rotate, bits shifted out of bit 0 are shifted into bit 35. In a right–rotate, bit 35 is shifted into bit 0.

15.3.6 ROTC – Rotate Combined

The data movement for the ROTC instruction is shown in Figure 15.8: AC and AC+1 are rotated as a 72–bit quantity. In a left–rotate, AC bit 0 shifts into AC+1 bit 35; AC+1 bit 0 shifts into AC bit 35. In a right–rotate, AC+1 bit 35 shifts into AC bit 0; AC bit 35 shifts into AC+1 bit 0.

Shift and Rotate Data Movement

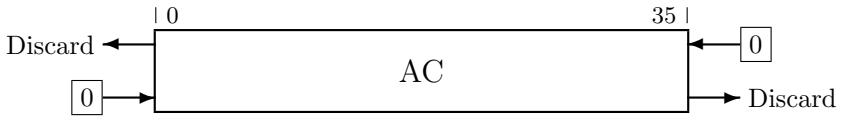


Figure 15.3: `LSH` Data Movement

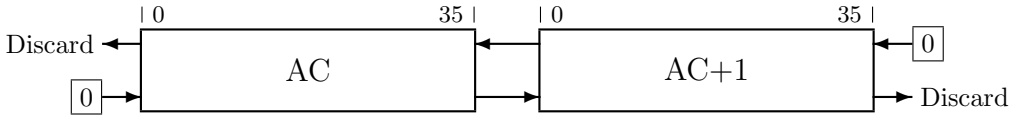


Figure 15.4: `LSHC` Data Movement

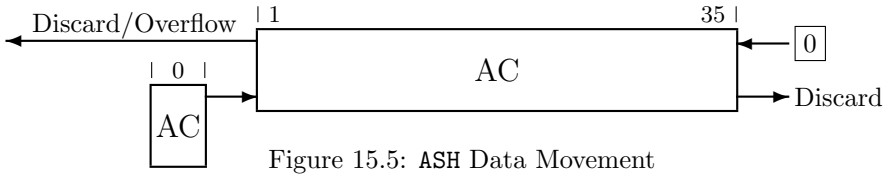


Figure 15.5: `ASH` Data Movement

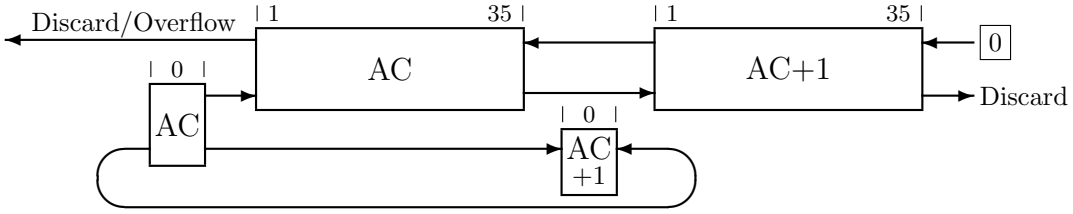


Figure 15.6: `ASHC` Data Movement

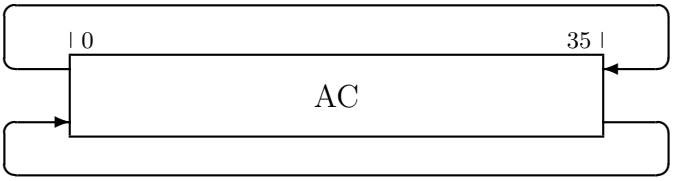


Figure 15.7: `ROT` Data Movement

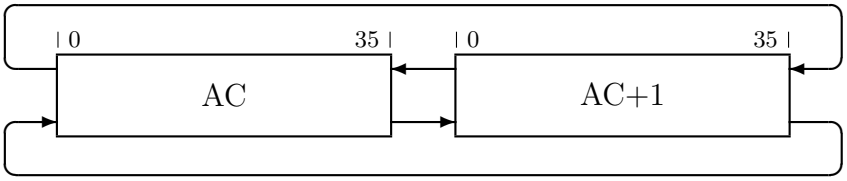


Figure 15.8: `ROTC` Data Movement

Chapter 16

Arithmetic

You might have thought that computers were used for arithmetic. It is true that quite often they are. We have come a long way without resorting to adding or dividing things. But now, the time has come to talk of arithmetic and the instructions that we use to perform numeric calculations.

16.1 Fixed–Point Arithmetic

The instructions that follow deal with fixed–point binary numbers that are (usually) thirty–six bits long. Two's complement binary arithmetic is performed and two's complement results are produced. You might want to take this opportunity to review the discussion of binary arithmetic in Section 4.2, page 30.

The usual convention is to consider that the binary point is placed to the right of bit 35. Using this convention, positive numbers in the range 0 to $2^{35} - 1$ are represented in a straightforward binary pattern. The arithmetic weight of bit number n is given by $2^{(35-n)}$. Negative numbers are in two's complement form: bit 0, the sign bit, has weight -2^{35}.

Other conventions for placing the binary point can be used with fixed–point arithmetic. For example, the operating system internal date format represents integral days in the left half of a word, and fractional days in the right half. For this purpose, the binary point might be said to lie between bits 17 and 18.

16.1.1 ADD Class

The instructions in the `ADD` class form the arithmetic sum of two fixed–point numbers. The operands include the accumulator and the contents of the effective address (or the effective address itself in the case of `ADDI`). The destination of the result is determined by the instruction modifier.

ADD	C(AC)	:= C(AC) + C(E);
ADDI	C(AC)	:= C(AC) + E;
ADDM	C(E)	:= C(AC) + C(E);
ADDB	Temp	:= C(AC) + C(E); C(AC) := Temp; C(E) := Temp;

16.1.2 SUB Class

The SUB instructions compute the difference of the contents of the accumulator minus the memory operand. The destination of the result is determined by the instruction modifier.

SUB	C(AC)	:= C(AC) – C(E);
SUBI	C(AC)	:= C(AC) – E;
SUBM	C(E)	:= C(AC) – C(E);
SUBB	Temp	:= C(AC) – C(E); C(AC) := Temp; C(E) := Temp;

The ADD or SUB class instructions will overflow if the result of the operation, interpreted as a signed, two's–complement number, is greater than or equal to 2^{35} or less than -2^{35}. Overflow is signalled when CRY0 and CRY1 are different. In case of an overflow, the result that is stored will be arithmetically correct, except the sign bit will be wrong. Overflow is signalled to the program via the PC flags AROV and TRAP1.

(Unsigned arithmetic can be performed on the PDP–10 if care is taken: the hardware does not detect overflow automatically. When two unsigned integers are added, CRY0 signifies an overflow. When one unsigned number is subtracted from another, the *absence* of CRY0 signals an underflow.)

16.1.3 IMUL Class

The IMUL instructions are for multiplying numbers when the product is expected to be representable as one word.

IMUL	C(AC)	:= C(AC) * C(E);
IMULI	C(AC)	:= C(AC) * E;
IMULM	C(E)	:= C(AC) * C(E);
IMULB	Temp	:= C(AC) * C(E); C(AC) := Temp; C(E) := Temp;

The IMUL class instructions will overflow if the result of the operation is greater than or equal to 2^{35} or less than -2^{35}. Overflow is signalled by the PC flags AROV and TRAP1.

16.1.4 IDIV Class

The `IDIV` instructions are for divisions in which the dividend is a one–word quantity. The dividend is in the accumulator; the divisor is the memory operand. The quotient will be stored as specified by the instruction modifier. The remainder will have the same sign as the dividend; the remainder is stored in `AC+1` (that is, in `AC+1` modulo `20` octal), except the `IDIVM` instruction does not store a remainder.

If the divisor is zero, the `AROV`, `TRAP1`, and `DCK` (no divide) flags are set in the PC; neither the accumulator nor the memory operand is changed.

`IDIV`	C(AC)	:= C(AC) / C(E); C(AC+1) := remainder;
`IDIVI`	C(AC)	:= C(AC) / E; C(AC+1) := remainder;
`IDIVM`	C(E)	:= C(AC) / C(E);
`IDIVB`	Temp	:= C(AC) / C(E); C(AC+1) := remainder;
	C(AC)	:= Temp; C(E) := Temp;

In division, the sign of the quotient is positive when the signs of both the dividend and divisor are identical. Otherwise, the sign of the quotient is negative. The sign of the remainder is always the same as the sign of the dividend. In all cases the magnitude of the quotient and remainder are the same as though both the dividend and divisor were positive. The relation

Dividend = (Quotient * Divisor) + Remainder

always holds. Some examples follow:

	Dividend	Divisor	Quotient	Remainder
	5	2	2	1
centering	–5	2	–2	–1
	5	–2	–2	1
	–5	–2	2	–1

16.1.5 MUL Class

The `MUL` instructions produce a double–word product. A double–word integer has seventy bits of significance. Bit 0 of the high–order word is the sign bit. In results, bit 0 of the low–order word is the same as bit 0 in the high–order word. `MUL` will set overflow (`AROV` and `TRAP1`) only if both operands are -2^{35}.

`MUL`	C(AC AC+1)	:= C(AC) * C(E);
`MULI`	C(AC AC+1)	:= C(AC) * E;
`MULM`	C(E)	:= high–order word of product of C(AC) * C(E);
`MULB`	C(AC AC+1)	:= C(AC) * C(E);
	C(E)	:= high–order word of product, as stored in AC;

16.1.6 DIV Class

The `DIV` instructions are for divisions in which the divide nd is a two–word quantity (such as produced by `MUL`). The `DIV` instructions will not perform a division if the divisor is zero, or if the divisor is smaller than the contents of `AC`, or if the quotient would overflow a single word (as, for example, dividing -2^{35} by -1). If any of these conditions obtains, `AROV`, `TRAP1`, and `DCK` are set in the PC flags. Bit 0 of the low–order word of the dividend is ignored by this instruction.

```
DIV    C(AC)   :=   C(AC AC+1) / C(E);     C(AC+1) := remainder;
DIVI   C(AC)   :=   C(AC AC+1) / E;     C(AC+1) := remainder;
DIVM   C(E)    :=   C(AC AC+1) / C(E);
DIVB   Temp    :=   C(AC AC+1) / C(E);   C(AC+1) := remainder;
                        C(AC) := Temp;   C(E) := Temp;
```

16.2 Double–Word Memory Operands

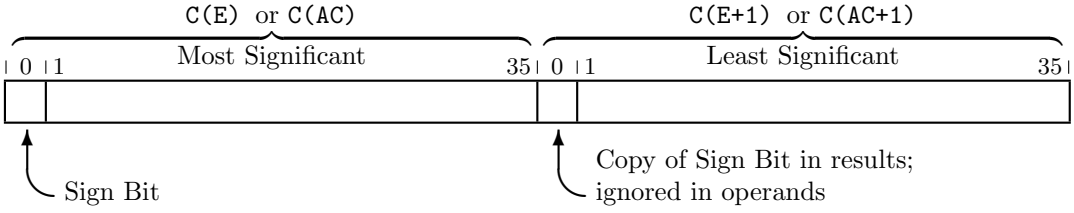
We are about to look at machine instructions that use double–word operands. The computer will generate an effective address `E` for each instruction. The operand will be `C(E)` and `C(E+1)`, with the former considered to be the most significant.

In the extended machine, a special note applies to instructions in which the memory operand is more than one location. When the in–section component of `E` is `777777` there is subtlety to the interpretation of `E+1`. When `E` is local, `E+1` is local `0`, the accumulator. However, when `E` is global, then `E+1` is location `0` in the next higher section.

This distinction reinforces something we should remember from effective address calculation: in local addressing, addresses stay local to the section from the address word was read; in global addressing, the entire memory space of addresses at and above `1000000` is an array, regardless of address section boundaries.

16.3 Double–Precision Fixed–Point Arithmetic

There are four instructions for double–precision fixed–point arithmetic. None of these instructions have any modifier: they all operate on double (or quadruple) accumulators and double–words in memory with results to double (or quadruple) accumulators. These instructions exist on the KL10 and later machines.



The format for a double–word fixed–point number is the same as that produced by `MUL`, i.e., a

seventy–bit integer in two's complement; bit 0 of the most significant word is the sign; in operands, bit 0 of the low–order word is ignored. A quadruple word has one hundred and forty bits; bit 0 of the most significant word is the sign; in operands, bit 0 in all other words is ignored. An instruction that produces a double (or quadruple) arithmetic result will store the same value in bit 0 of the low–order word(s) as it stores in bit 0 of the high–order word.

DADD	C(AC AC+1) := C(AC AC+1) + C(E E+1);
DSUB	C(AC AC+1) := C(AC AC+1) – C(E E+1);
DMUL	C(AC AC+1 AC+2 AC+3) := C(AC AC+1) * C(E E+1);
DDIV	C(AC AC+1) := quotient of C(AC AC+1 AC+2 AC+3) / C(E E+1);
	C(AC+2 AC+3) := remainder of C(AC AC+1 AC+2 AC+3) / C(E E+1)

16.4 Double–Word Moves

There are four double–word move instructions. These are particularly suited for manipulating KI10 and KL10 double–precision floating–point numbers. These instructions do not exist on the KA10 or PDP–6.[1]

DMOVE	C(AC AC+1)	:= C(E E+1)	
DMOVEM	C(E E+1)	:= C(AC AC+1)	
DMOVN	C(AC AC+1)	:= –C(E E+1)	*for double–precision floating–point*
DMOVNM	C(E E+1)	:= –C(AC AC+1)	*for double–precision floating–point*

DMOVE and DMOVEM handle double–word operands without interpreting what they mean.

DMOVN and DMOVNM are designed to negate double–precision (or extended–range) floating–point numbers.

When these instructions are used to manipulate double–precision integers a word of caution is appropriate: the instructions that produce double–precision integers (MUL, DADD, DSUB, DDIV) always set bit 0 of the second word to be the same as the sign bit of the first word. However, the DMOVN and DMOVNM instructions always set bit 0 of the second word to zero (as is appropriate for double–precision floating–point numbers). In double–word integer arithmetic operations, bit 0 of the second word is ignored. But there is no machine instruction for double–word comparison; such a comparison would be fashioned from a collection of single–word comparisons. In such a case, bit 0 of the second word would be looked at. For safety, to negate a double–precision integer, you should use either of the sequences below:

```
SETZB   AC,AC+1         ;a doubleword zero
DSUB    AC,E            ;0-C(E E+1) =  -C(E E+1)
```

or

[1] The DMOVN and DMOVNM instructions must not be used with any KA10–format double–precision floating–point numbers; see also Appendix D, page 657.

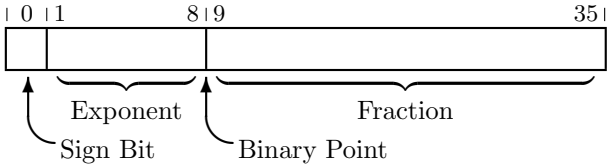


Figure 16.1: Single–Precision Floating–Point Number

```
DMOVN   AC,E            ;negate C(E E+1)
SKIPGE  AC              ;skip if result is positive
 TLO    AC+1,400000     ;set sign bit of second word.
```

16.5 Floating–Point Operations

When it performs floating–point arithmetic operations, the computer hardware takes upon itself the burden of scaling numbers properly to make sensible results. Since the hardware helps in this way, floating–point is more suitable than fixed–point arithmetic for many purposes.

16.5.1 Floating–Point Representations

The PDP–10 offers three formats of floating–point numbers: single–precision, double–precision, and "giant" or "G" format for extended range numbers.[2] We will discuss the representation used for single precision floating–point numbers in great detail. Following our discussion of the single–precision representation, we will briefly explain the double–precision and "G" formats.

16.5.1.1 Single–Precision Floating–Point

Single–precision floating–point numbers are represented in one 36–bit word as depicted in Figure 16.1

In the figure, the field S denotes the sign bit, bit `0`. When S is zero, the sign is positive. When S is one, the sign is negative and the word is in two's complement format. The exponent is held in bits `1:8`. The exponent, of the base two, appears in excess–`200` (octal) notation. The fraction, held in bits 9:35, is interpreted as having a binary point to the left of bit 9.[3]

A floating–point zero is represented by a word in which all bits are zero.

In a properly *normalized*, non–zero floating–point number (see Section 16.5.2, page 226) bit 9 differs from bit 0.[4] In a normalized number, the fraction represents a value whose magnitude is greater

[2]G format is available only in the extended KL10 processor and the TOAD.

[3]By a mistaken analogy to logarithms, the fraction is sometimes called the *mantissa*.

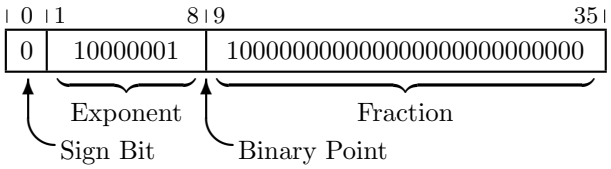
[4]There are two exceptions: zero is represented by a word that is entirely zero, and a negative floating–point number in which bits 0 and 9 are both one is normalized if all the other fraction bits, bits 10:35, are zero. This is because negative numbers are in two's complement form; a normalized positive number in which bits 10:35 are zero has a two's complement in which bits 10:35 are also zero.

than or equal to one–half and less than one: $\frac{1}{2} \leq fraction < 1$. (The largest value of the fraction, `777,,777777` represents $1 - 2^{-27}$ or 0.9999999925494...)

Floating–point numbers can represent numbers with magnitudes within the range from 0.5×2^{-128} to $(1 - 2^{-27}) \times 2^{127}$, and 0. In more familiar notation, the magnitude range is approximately from 1.4×10^{-39} to 1.7×10^{38}, and 0.

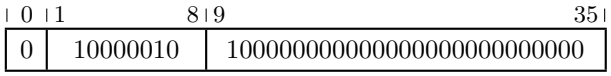
There are infinitely many real numbers in this range, but only a finite number of different representations. This has two consequences: a particular real number may be represented by an approximation, and two distinct real numbers may be represented by the same approximation.

We will look at some examples of floating–point numbers. First, we will examine the number 1.0. To convert a number to floating–point, it must be broken into a fraction that is less than 1 and an exponent. The number 1.0 is equivalent to 1.0×2^0, but the 1.0 is too large a fraction. So we divide the 1.0 by 2, and increase the exponent: 0.5×2^1. Now, we're ready. Write 0.5 as a binary fraction: 0.1. So we build a word that contains the binary pattern shown here:



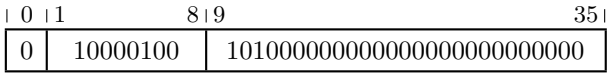
In octal, this would be `201400,,0`. Note that `200` has been added to offset the exponent. This offset is used in lieu of keeping a sign bit for the exponent.

For a second example, let us try to convert 2.0 to floating–point. First, in binary, 2.0 is 10.0. Again the number is too large, so we must make it a fraction. 10.0×2^0 is changed to 0.100×2^2. So we write the binary pattern:



Notice that this is the same as the pattern for 1.0, except the exponent is different. Adding one to the exponent doubles a floating–point number.

Next, we try to represent decimal 10.0. In binary we would have 1010.0 or 0.101×2^4. Thus we write:



As our next example, we will demonstrate the conversion of an internal format floating–point number to decimal notation. For this example, we will convert the internal number `202500,,0` to decimal. The exponent field is `202`, meaning $\times 2^2$, i.e., $\times 4$. The fraction field, in binary is 0.101000. Multiplying the fraction by 4, we obtain the binary number 10.1000. This number is equivalent to binary 101.00 divided by 2. Since 101 in binary is 5, this number must be 2.5. To check our work, we can compare the original octal pattern, `202500,,0` to the octal pattern corresponding to (decimal) 10.0. The floating–point representation of 10.0 is `204500,,0` which differs from our number by 2 in the exponent. A difference of 2 in the exponent means a factor of 4 difference in the values represented. Indeed, $\frac{10.0}{4}$ is 2.5.

As a final example, we shall try a hard one. We will convert $\frac{4}{3}$ to a floating–point number. We begin by converting the numerator and denominator to binary, and doing long division:

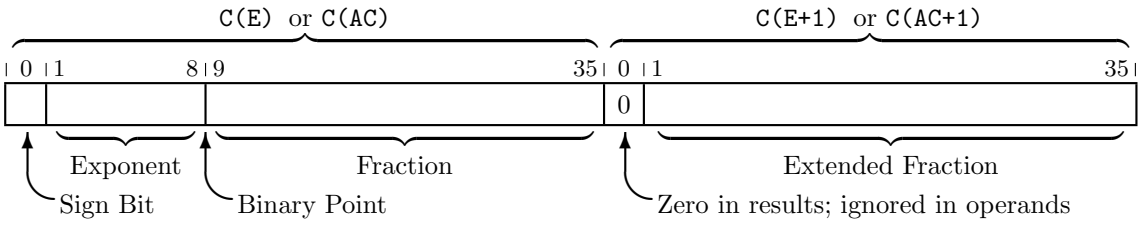
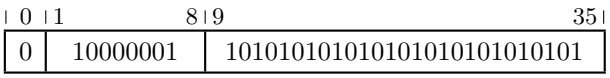


Figure 16.2: Double–Precision Floating–Point Number

```
    1.010101...
11 ) 100.000000...
     -11
       1.00
       -11
          100
          -11
            100
            -11
              1...
```

It should be apparent that the pattern `01` will repeat. The resulting binary floating–point number is



16.5.1.2 Double–Precision

For many applications, the single–precision format produces insufficiently accurate results. A second format of floating–point numbers is available for circumstances where extra precision is required. Increased precision comes from the inclusion of a second word which contains thirty–five additional fraction bits in bits 1:35; bit 0 of the second word is always zero as shown in Figure 16.2. The additional fraction bits do not materially affect the range of representable numbers, rather they extend the precision to approximately one part in 2^{62}, typically 18 decimal digits. Double–precision floating–point is available in the KI10 and later processors. In memory operands, if `E` addresses the most significant word, then the second word is found at `E+1`; similarly, an operand in the accumulators is specified by naming `AC` and the second word is found at `AC+1` modulo (octal) 20.

The instructions whose names begin with `DF` are used to manipulate the double–precision floating–point objects. These instructions have no modes: operands are always `C(AC AC+1)` and `C(E E+1)` with results stored in the accumulators. `DMOVEx` and `DMOVNx` are suitable for moving and negating these numbers.

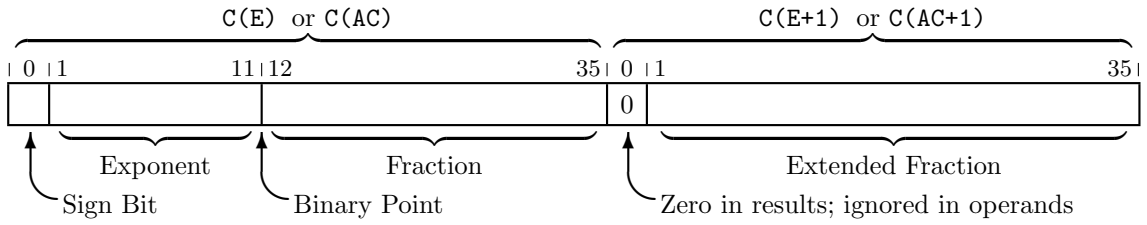


Figure 16.3: Giant–Format Floating–Point Number

16.5.1.3 Giant–Format — Extended Range Floating Point

In extended–range (or "*giant*") floating–point numbers, the exponent field is expanded by three bits at the expense of losing bits in the fraction, as depicted in Figure 16.3. For this small loss in precision, the range has been greatly extended. The resulting exponent (in excess 2000 octal) can take on values from 2^{-1024} to 2^{1023}. Numbers whose magnitudes are in the range from 2.78×10^{-309} to 8.98×10^{307} and zero are representable.

16.5.2 Floating–Point Arithmetic Operations

When the computer does addition (or subtraction) involving floating–point numbers, it is necessary to make the exponents of both numbers the same before adding them. Suppose we add 2.0 to the representation of $\frac{4}{3}$ that we computed earlier.

The representations of both numbers are shown below:

```
2.0         0|10 000 010|100 000 000 000 000 000 000 000 000 000
1.3333...   0|10 000 001|101 010 101 010 101 010 101 010 101
```

Before adding, the PDP–10 performs a pre–normalization process to make the smaller number (i.e, the operand of smaller magnitude) have the same exponent as the larger one. The fraction is shifted right, halving the value at each shift; simultaneously, the exponent is incremented, doubling the value at each shift. Except for loss of precision that occurs if significant bits are shifted off the right end of the fraction, pre–normalization does not change the value of the original operand; the process stops when the smaller number has been changed to have an exponent that is identical to the exponent of the larger operand. Since this shifting takes place inside the CPU, a result that is longer than one word can be kept. A guard bit, essentially bit 36, remembers the most significant of the bits that were shifted out during pre–normalization.

```
2.0         0|10 000 010|100 000 000 000 000 000 000 000 000 000
1.333...    0|10 000 010|010 101 010 101 010 101 010 101 010|1 (after shift)
                                                             ↑
                                                             guard bit
```

The addition proceeds, resulting in a new number. If, as is usual, addition with *rounding* is requested, the presence of a one in the guard bit modifies the result. If the guard bit is a one, rounding is accomplished by adding one to the least significant position. Note the effect this has on the result in this case:

```
2.0         0|10 000 010|100 000 000 000 000 000 000 000 000
1.333...    0|10 000 010|010 101 010 101 010 101 010 101 010|1   (after shift)

3.333...    0|10 000 010|110 101 010 101 010 101 010 101 011     (sum with rounding)
```

It would be possible to verify that the result is equivalent to (approximately) 3.333...

Because there are a limited number of bits available to represent the floating–point number, some loss of precision occurs on conversion from external numbers to internal format. Further loss of precision results from performing arithmetic operations. For example, if we now subtract 3.0 from this sum, the result can be computed as follows:

```
3.333...     0|10 000 010|110 101 010 101 010 101 010 101 011
3.000        0|10 000 010|110 000 000 000 000 000 000 000 000

0.333...     0|10 000 010|000 101 010 101 010 101 010 101 011  (difference)
```

This result differs from previous floating–point numbers in the following way. In all examples thus far, bit 9 (in positive numbers) has been a one. Making bit 9 a one is a goal of the hardware. The result of this subtraction is said to be *unnormalized*; a normalized result has a significant bit in position 9. The PDP–10 hardware would not be satisfied to leave this result alone. It would *post–normalize* the result by shifting the fraction left and decrementing the exponent. A left shift of the fraction doubles the fraction; decreasing the exponent by one halves the value of the number. Thus, when these actions are coupled, there is no change to the value of the number, but the number becomes *normalized*. Normalization is desirable for at least two reasons. First, normalization eliminates non–significant bits at the left, making room for guard bits to be shifted in at the right. Second, two floating–point numbers that are both normalized can be compared with the same `CAM` class instructions that are used to compare integers; unnormalized numbers cannot be compared readily.

When we shift this number three places and subtract three from the exponent the result is

```
0.333...    0|01 111 111|101 010 101 010 101 010 101 011 000
```

Note that subtracting `3` from the exponent field of `202` leaves us an exponent field containing `177`. This number corresponds to $0.10101010101010101010101011000 * 2^{-1}$.

Pay special attention to the fact that as a result of these manipulations the low–order four bits taken together have the value 8; a more nearly correct result would be 5. Arithmetic done on imprecise quantities produces even less precise results. Care must be taken when considering the results from extensive floating–point calculations. The area of computer science called *numerical analysis* concerns itself with questions of precision and effective algorithms for a variety of mathematical problems. Further information about algorithms to implement floating–point arithmetic can be found in [KNUTH 2].

16.5.2.1 Special Cautions

A number in which bit 0 is one and bits 9:35 are zero can produce an incorrect result in any floating–point operation. A word with a zero fraction and non–zero exponent can produce extreme loss of precision if used as an operand in a floating–point addition or subtraction.

16.5.2.2 Floating–Point Exceptions

Under some circumstances, floating–point operations will produce incorrect results. If an attempt is made to compute a number that is smaller in magnitude than $0.5*2^{-128}$ then the FOV and FXU (Floating OVerflow and Floating eXponent Underflow) flags will be set in the PC. TRAP1 and AROV will also be set.

If a number that is larger in magnitude than 2^{127} is computed, FOV will be set along with AROV and TRAP1.

In cases where FOV is set, you may expect that the fraction portion of the result is arithmetically correct; however, the exponent will be wrong by decimal 256. In case of overflow, the exponent will be too small by 256; for underflow, the exponent is too large by 256.

For the G–format operations, overflow occurs when the result exceeds 2^{1023} in magnitude; underflow occurs when the result is smaller than 2^{-1025} in magnitude; the exponent of the result will be wrong by 2048.

16.5.3 Floating–Point Instruction Set

Table 16.1 sets forth the PDP–10 floating–point arithmetic instruction set. The PDP–10 also includes instructions to perform conversion between the fixed–point and floating–point formats.

Most of the floating–point arithmetic instructions are straightforward and need no detailed explanation. However, the immediate mode instructions are somewhat unusual. Just as the TL instructions swap the effective address into the left halfword of the mask, the floating point immediate instructions swap the effective address into the left halfword to form the sign, exponent and high–order fraction bits of a floating point operand. In an immediate mode floating–point instruction, the memory operand is <E,,0>. Often you may see an instruction written as:

```
FMPRI   AC,(10.0)
```

This instruction makes use of an assembler trick. Since the X field of an instruction is in the right end of the left halfword, the assembler processes the notation (B) by evaluating the expression B, swapping the halves of the result, and then adding the swapped result to the word that is being assembled. In the case of an index register, a small number, perhaps 0,,2, is swapped to make 2,,0; this is added to the word being assembled, setting the X field to 2.

In this example, the 10.0 is evaluated to octal 204500,,0. When 10.0 appears in parentheses where MACRO is expecting an address field, MACRO swaps this number (getting 0,,204500). The swapped number is then added to the word being assembled. The result is a word containing FMPRI AC,204500. When executed, this instruction has the effect of multiplying AC by floating–point 10.0. Another common version of the same trick is to use a MOVSI to load an accumulator with a floating–point number, e.g.,

```
MOVSI   AC,(1.0)
```

Naturally, one should be certain that the floating–point number in question has only zeros in the right halfword.

(Rather than copy the examples above, we now prefer to use the macro named MOVX to handle many cases of loading a constant into an accumulator. However, when we get to macros, we will find that MOVX will make use of parentheses to swap the left and right halves of an operand.)

`F` Floating { `AD` Add; `SB` Subtract; `MP` Multiply; `DV` Divide } {

`R` Rounded { ␣ Result to AC; `I` Immediate Operand E, , 0. Result to AC; `M` Result to Memory; `B` Result to both AC and Memory }

without rounding { ␣ Result to AC; `M` Result to Memory; `B` Result to both AC and Memory }

}

`DF` Double Floating, `GF` Giant Floating } { `AD` Add; `SB` Subtract; `MP` Multiply; `DV` Divide } Memory Operand from C(E E + 1) AC Operand from, and Result to, C(AC AC + 1)

␣ Single to single, `G` Giant to single, `GD` Giant to Double } `FIX` Convert Floating–Point to Fixed–Point { ␣ without rounding; `R` with Rounding }

␣ Single to single, `G` Single to Giant, `DG` Double to Giant } `FLTR` Convert Fixed–Point to Floating–Point, with Rounding.

␣ Single or double, `G` Giant } `FSC` Floating Scale (immediate).

`GSNGL` Giant to Floating (single) Convert.

`GDBLE` Floating (single) to Giant Convert.

Table 16.1: Floating–Point Instruction Set

The instructions that follow are used to convert between fixed and floating formats, and to scale floating–point numbers.

16.5.3.1 FIX — Convert Floating–Point to Fixed–Point

FIX will convert a floating–point number to an integer. If the exponent of the floating–point number in C(E) is greater than (decimal) 35 (i.e., an exponent field larger than octal 243) then this instruction will set the arithmetic overflow flag, AROV and TRAP1, and not affect the accumulator. Otherwise, the FIX instruction will convert the floating–point number at the effective address to fixed–point by the following procedure: Move C(E) to the accumulator, extending the sign bit, bit 0 of C(E), into bits 1:8 of accumulator. Then perform an arithmetic shift on the contents of the accumulator by EXP-233 bits, where EXP is the exponent field from bits 1:8 of C(E).

FIX will always truncate towards zero, i.e., 1.9 is fixed to 1 and −1.9 is fixed to −1. This truncation is that used in the Fortran language for conversion of real to integer. For positive numbers, bits shifted off the right–end are ignored. For negative numbers, if any "1" bits are shifted off the right–end then 1 is added to bit 35 to make the result closer to zero.

GFIX is similar to FIX. The "giant" format floating–point number in C(E E+1) is converted to a single integer in C(AC). GFIX is available only under the EXTEND instruction. Write it as

```
EXTEND  AC,[GFIX 0,E]    ;any of I,X,Y are legal to form E
```

GDFIX is similar to FIX. The result, in C(AC AC+1) is a double–precision integer. GDFIX will overflow if the effective exponent is greater than (decimal) 70 (octal 2106 in the exponent field). GDFIX is available only under the EXTEND instruction. Write it as

```
EXTEND  AC,[GDFIX 0,E]   ;any of I,X,Y are legal to form E
```

FIX	C(AC)	:= fixed–point version of the floating number C(E)
GFIX	C(AC)	:= fixed–point version of the floating number C(E E+1)
GDFIX	C(AC AC+1)	:= double–length fixed–point version of the floating number C(E E+1)

16.5.3.2 FIXR — Fix and Round

The FIXR instruction will convert a floating–point number to an integer by rounding. **Note:** "rounding" in FIXR differs from the rounding employed in the other arithmetic instructions; see FLTR in Section 16.5.3.3, page 231 for the more typical rounding.

If the exponent field of the floating–point number in C(E) is greater than octal 243 (meaning an effective exponent greater than decimal 35) then this instruction will set AROV and not affect the accumulator. Otherwise, C(E) is converted to fixed–point by the following procedure: move C(E) to the accumulator, extending the sign bit into bits 1:8 of AC. Then arithmetic shift the accumulator by EXP-233 bits (where EXP is the exponent (or, in the case of a negative number, the one's complement of the exponent) from bits 1:8 of C(E)). If EXP-233 is non–negative then no rounding will take place.

When EXP-233 is negative, the word in the accumulator has been shifted right. The rounding process

will consider the most significant bit that was shifted off the right end of the accumulator. If the last bit shifted off the right end of the accumulator was a one, then one will be added to bit 35 of the result.

In `FIXR` rounding is always in the positive direction:

1.4	$\Longrightarrow$	1	–1.4	$\Longrightarrow$	–1
1.5	$\Longrightarrow$	2	–1.5	$\Longrightarrow$	–1
1.6	$\Longrightarrow$	2	–1.6	$\Longrightarrow$	–2

This rounding procedure is the Algol language standard for real to integer conversion.

The `GFIXR` instruction will convert a G–format floating–point number to an integer by rounding. If the exponent field of the floating–point number in `C(E E+1)` is greater than octal `2043` (meaning an effective exponent greater than decimal 35) then this instruction will set `AROV` and not affect the accumulator. Otherwise, `C(E E+1)` is converted to fixed–point by the following procedure: copy `C(E E+1)` to an internal double–word register. Extend the sign bit into bits 1:11 of the high–order word of that register. Then arithmetic shift (`ASHC`) the double–word register by `EXP-2030` bits (where `EXP` is the exponent from bits 1:11 of `C(E)`). The rounding process will consider the data bit to the right of of bit 35 in the high–order word. If that bit is a one, then one will be added to bit 35 of the result. Rounding is always in the positive direction.

`GFIXR` is available only under the `EXTEND` instruction. Write it as

```
EXTEND  AC,[GFIXR 0,E]   ;any of I,X,Y are legal to form E
```

The `GDFIXR` instruction will convert a G–format floating–point number to a double–precision integer by rounding. If the exponent field of the floating–point number in `C(E E+1)` is greater than octal `2106` (meaning an effective exponent greater than decimal 70) then this instruction will set `AROV` and not affect the accumulator. Otherwise, `C(E E+1)` is converted to fixed–point by the following procedure: copy `C(E E+1)` to an internal double–word register. Extend the sign bit into bits 1:11 of the high–order word of that register. Then arithmetic shift (`ASHC`) the double–word register by `EXP-2073` bits (where `EXP` is the exponent from bits 1:11 of `C(E)`). If `EXP-2073` is non–negative then no rounding will take place.

If `EXP-2073` is negative then the double–word register was shifted to the right. The rounding process will consider the last data bit that was shifted off the low–order word. If that bit is a one, then one will be added to bit 35 of the low–order word. The double–word register is stored in `C(AC AC+1)`. Rounding is always in the positive direction.

`GDFIXR` is available only under the `EXTEND` instruction. Write it as

```
EXTEND  AC,[GDFIXR 0,E]  ;any of I,X,Y are legal to form E
```

`FIXR`	C(AC)	:= fixed, rounded version of the floating number C(E)
`GFIXR`	C(AC)	:= fixed, rounded version of the giant floating number C(E E+1)
`GDFIXR`	C(AC AC+1)	:= double–length fixed, rounded version of the giant floating number C(E E+1)

16.5.3.3 FLTR — Float and Round

The FLTR instruction will convert an integer in C(E) to a floating–point number in C(AC).

The FLTR instruction uses the normalize and round "subroutine" in the CPU that is common to many of the floating–point operations. [5] Because normalize and round is important to so many instructions, we describe the actions of FLTR in excruciating detail below:

The integer data from C(E) is copied to a CPU internal register. FE (the CPU's internal floating exponent register) is loaded with octal 243, signifying an effective exponent of (decimal) 35. To simplify the normalization and rounding steps that follow, a negative number would be negated at this point, with the CPU remembering that it must negate the normalized, rounded result before storing it.

If bits 1:8 are non–zero, the number is right–shifted (and the exponent in FE increased) until bit 9 is one and bits 1:8 are all zero. After this shift, if the last bit right–shifted out of bit 35 is one, then one is added to bit 35 to effect rounding: rounding moves results away from zero. (In the rare case that bits 9:35 were all ones, the add de–normalizes the result which is then shifted one more place to the right with an increment to FE.) The FE is copied into bits 1:8. If the original number was negative, this result is negated again. Finally, the result is stored in AC.

If bits 1:8 are zero, then a test is made to see if the entire fraction is zero. If so, a zero is stored in AC. Otherwise, the fraction is left–shifted (with corresponding decrease to FE) until bit 9 becomes one. No rounding takes place: the conversion is exact. Then, as described above, the FE is copied into bits 1:8. If the original number was negative, this result is negated again. Finally, the result is stored in AC.

The GFLTR instruction will convert an integer in C(E) to a G–format floating–point number in C(AC AC+1). This instruction doesn't actually do rounding because every single–precision integer has an exact representations in G–format.

The instruction performs as follows. The accumulator AC+1 is set to zero. The data from C(E) is copied to AC where it is shifted right 11 places as a double–word, extending the sign (ASHC). The exponent 2040 is inserted into bits 1:11; if the number is negative, the exponent field is set to the one's complement of 2040, 5737. The resulting number is normalized until bit 12 becomes significant.

GFLTR is available only under the EXTEND instruction. Write it as

```
EXTEND  AC,[GFLTR 0,E]   ;any of I,X,Y are legal to form E
```

The DGFLTR instruction will convert a double–precision integer in C(E E+1) to a G–format floating–point number in C(AC AC+1).

The instruction performs as follows. The data from C(E E+1) is copied to AC AC+1 where it is shifted right 11 places as a double–word, extending the sign (as in ASHC), retaining the bits that are shifted out. The exponent 2073 (or its one's complement) is inserted into bits 1:11; The resulting number is normalized by left–shifting until bit 12 becomes significant. The left–shift may restore some or all of the bits shifted right initially. If any of the bits shifted right remain outside the double–word result, then if the leftmost of those bits is one, the result is modified by adding one to bit 35 of the low–order word of the fraction.

DGFLTR is available only under the EXTEND instruction. Write it as

[5] The word "subroutine" appears in quotation marks because some machines have subroutines built in their hardware, and others have subroutines in their microcode.

```
EXTEND  AC,[DGFLTR 0,E]  ;any of I,X,Y are legal to form E
```

FLTR	C(AC)	:= floating, rounded version of the fixed number C(E)
GFLTR	C(AC AC+1)	:= G–floating, exact version of the fixed number C(E)
DGFLTR	C(AC AC+1)	:= G–floating, rounded version of the double–precision fixed number C(E E+1)

16.5.3.4 FSC — Floating Scale

The FSC instruction will add E (i.e., an immediate quantity) to the exponent of the number in AC and normalize the result. This is useful for multiplying or dividing a single–precision floating–point number by a power of two. Each unit added to the exponent doubles the number. FSC AC,2 would add two to the exponent, multiplying the number in AC by four. Similarly, FSC AC,-3 would effectively divide by eight. FSC will set AROV, FOV, and TRAP1 if the resulting exponent exceeds decimal 127. FXU (and all the flags set by overflow) will be set if the exponent becomes smaller than −128.

In the KA10 and earlier processors, FSC is sometimes used to convert an integer to floating–point. The FLTR instruction that is available in the KI10 and newer processors is more general, so FLTR is more frequently used. To use FSC to float a small integer, copy the integer to an accumulator. Perform the instruction FSC AC,233 (excess 200 and shift the binary point 27 bits). The integer being floated must not have more than 27 significant bits.

FSC may be used for various "tricks" not having anything to do with floating–point numbers. For example, the following fragment of code converts a bit–mask (of up to 27 bits) to a single number that identifies the leftmost one in the mask: (This presumes the mask is not zero.)

```
MOVE    A,MASK
FSC     A,32    ;insert "exponent" and count down any leading zeros
LSH     A,-33   ;extract the exponent only
```

The result in A will be (decimal) 35 – (the bit number of the leftmost mask bit). This is similar to the effects of the JFFO instruction, although it numbers the result differently.

FSC can also be used to scale (i.e., change the exponent of) a *normalized* double–precision floating–point number. If the double–precision number is in C(AC AC+1) then FSC AC,1 will double the value. Note that the operand must be normalized for this to work correctly.

The GFSC instruction can be used to scale a G–Format number. The immediate operand is added to the exponent found in bits 1:11 of AC and the double–word operand in C(AC AC+1) is normalized. Overflow (AROV, FOV, and TRAP1) occurs if the resulting exponent exceeds octal 3777 (+1023 decimal). Underflow (FXU and all the flags for overflow) occurs if the resulting exponent is smaller than zero (–1024 decimal).

GFSC is available only under the EXTEND instruction. Write it as

```
EXTEND  AC,[GFSC 0,E]    ;any of I,X,Y are legal to form E
```

FSC	C(AC)[1:8]	:= C(AC)[1:8]+E; normalize C(AC)
GFSC	C(AC)[1:11]	:= C(AC)[1:11]+E; normalize C(AC AC+1)

16.5.3.5 GSNGL — Convert G–Format to Single–Precision Floating–Point

The GSNGL instruction converts a G–format quantity (in C(E E+1)) to a single–precision floating–point number in C(AC). Overflow or underflow is possible, in which case the appropriate flags are set and the accumulator is not affected.[6]

GSNGL is available only under the EXTEND instruction. Write it as

```
EXTEND  AC,[GSNGL 0,E]    ;any of I,X,Y are legal to form E
```

GSNGL	C(AC) := C(E E+1) converted from G to F

16.5.3.6 GDBLE — Convert Single–Precision Floating–Point to G–Format

The GDBLE instruction converts a single–precision floating–point quantity (in C(E)) to a G–format quantity in C(AC AC+1). This conversion is exact. GDBLE is available only under the EXTEND instruction. Write it as

```
EXTEND  AC,[GDBLE 0,E]    ;any of I,X,Y are legal to form E
```

GDBLE	C(AC AC+1) := C(E) converted from F to G

16.6 Exercises

16.6.1 Date and Time Conversion

At the beginning of this section we stated that in TOPS–20 the date and time is represented in one word in which the left half represents integral days, and the right half word represents a fraction of

[6] In KL10 microcode version 2.1[442] conversion of a G–format number whose exponent is in the range 1570 to 1577 sets underflow and stores a result.

a day.

Often it's useful to know the difference between two times, as in the case where a program wants to know how long a user has been logged–in. Write a subroutine that takes two times, `STIME` and `ETIME` (i.e., starting time and ending time) and computes their difference in seconds.

That is, write the conversion from the day and fraction format to seconds.

Hint: the resulting subroutine should be quite short, perhaps as few as four to six instructions. Also, beware of overflows! You may want to use DDT to verify that your subroutine works properly.

16.6.2 Series and Convergence

Form the sum

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots - \frac{1}{1000}$$

Do so by combining the larger terms first, i.e., evaluate the expression stated above from left to right.

Repeat, this time combining the smallest terms first, i.e., from right to left.

Why do the two sums differ?

Chapter 17

Macros and Conditionals

The assembler is a text processor. It reads text, makes some straightforward translations, and outputs those translations. In the process of making translations it frequently *looks up* symbolic names to find their translation. Thus far, the only kind of translation that we have discussed is from a symbol name to a number. However, the assembler is capable of translating from a symbolic name to another text string; the new text string is then processed as though it, rather than its symbolic name, had appeared in the original file.

17.1 Macros

The idea of a *macro* is to give a name to a block of text. Then when we mention the name, the corresponding text appears. To give a specific example, suppose we frequently need the following code fragment:

```
MOVEI   B,15
IDPB    B,A
MOVEI   B,12
IDPB    B,A
MOVEI   B,0
IDPB    B,A
```

This code adds carriage return, line feed, and null to the end of a string that is addressed by a byte pointer in `A`.

It would be tiresome to write these six lines repeatedly in various parts of the program.[1] The assembler provides a tool that enables us to avoid this drudgery. We define a macro by means of the pseudo–op `DEFINE`. In this case we shall associate the name `ACRLF`, *Add CR and LF*, with this block of text:

[1] In this example, calling a subroutine would probably be a better choice. However, we shall see that macros with arguments permit extra flexibility not found in the usual subroutine call.

```
DEFINE ACRLF <
        MOVEI   B,15
        IDPB    B,A
        MOVEI   B,12
        IDPB    B,A
        MOVEI   B,0
        IDPB    B,A
>;End of ACRLF
```

The word `DEFINE` is followed by the name of the macro that is being defined. The *macro body*, the actual text that we associate with this definition, is enclosed in pointed brackets following the macro name. The comment that follows the closing pointed bracket is not necessary, but, like other comments, it is a good idea.

After making this definition, when the name `ACRLF` appears in a context where it is recognized as an identifier, MACRO will *expand* the name into the six lines corresponding to the macro definition. This action is referred to as *invoking the macro*.

Please understand that all processing of macros (and of conditional assembly functions that we shall discuss below) occurs inside the assembler. By the time your program runs, the macros themselves have been completely forgotten; only their effects remain.

17.1.1 Arguments to Macros

In the previous example, the macro definition specified that the byte pointer to the string be set up in location `A` (which might or might not be one of the accumulators). Also, the code generated by the expansion of this macro will change register `B`. This lack of flexibility might cause problems when this macro is invoked. `B` may be valuable, or the byte pointer may be somewhere other than in `A`. We can generalize the usefulness of a macro by adding *arguments*.

The following definition of `ACRLF` allows the specification of the accumulator and byte pointer to use. In the definition, after the name of the macro, write the names of the *formal parameters* in parentheses. The assembler treats the formal parameters, `ACC` and `BYP`, as placeholders for the text that will be supplied as *actual arguments* when the macro is invoked. Each formal parameter must be an identifier; in the definition the use of that identifier as a formal parameter overrides any other definition that uses the same name. Once the definition has been completed, the names by which the formal parameters are known disappear.

```
DEFINE ACRLF (ACC,BYP)<
        MOVEI   ACC,15
        IDPB    ACC,BYP
        MOVEI   ACC,12
        IDPB    ACC,BYP
        MOVEI   ACC,0
        IDPB    ACC,BYP
>
```

When this macro is invoked, the name `ACRLF` should be followed by two arguments. The first argument should correspond to the accumulator to use; the second should be the byte pointer. The

actual arguments in a macro invokation may be in parentheses, or not, at your option. The macro could be invoked as

```
ACRLF C,PTR-1(Y)
```

This would expand to

```
MOVEI   C,15
IDPB    C,PTR-1(Y)
MOVEI   C,12
IDPB    C,PTR-1(Y)
MOVEI   C,0
IDPB    C,PTR-1(Y)
```

Note that the actual argument `C` has replaced each instance of the formal parameter named `ACC`; similarly, the actual argument string `PTR-1(Y)` has replaced each instance of the formal parameter `BYP`. This example shows how an arbitrary string can be used as an actual argument.

17.2 Conditional Assembly

The assembler provides *conditional assembly* features for at least two broad purposes:

- to customize one source file for slightly different applications. For example, to allow one source file to be assembled for TOPS–10 or for TOPS–20.
- to allow the assembler to make decisions based on symbolic values. We'll have examples of this in the macros `TX..`, `MOVX`, `LOAD`, and `STOR` that we shall visit in later chapters.

We shall construct a program below that can be assembled for either TOPS–10 or TOPS–20. Although these are different operating systems, they both use the same machine instructions; only the operating system calls are different. Although in some cases the way things are done are so different between systems that it would be difficult to have only one program source, in many cases the differences are so minor that it is possible to have one source file that assembles for either system.

17.2.1 The Arithemetic Conditionals

One basic conditional assembly construction in MACRO is the arithmetic `IF` statement. The `IF` statement comes in six basic variations (corresponding to six of the eight modifiers for the `SKIP` or `JUMP` instruction): `IFG`, `IFGE`, `IFE`, `IFN`, `IFL`, and `IFLE`. The usual form of an arithmetic conditional is

```
IFx exp,<
        text to be assembled if the the value of the (integer)
        expression "exp" bears relation "x" to zero
>
```

In this example, `IFx` should be replaced by one of the six arithmetic `IF`s. The selected `IF` is followed by an arithmetic expression consisting of defined symbols. MACRO calculates the value of the expression. If the value bears the appropriate relation to zero, the condition is said to be *satisfied*. A satisfied condition will assemble the text that follows the expression. The extent of the text subject to the condition is defined by the paired angle brackets, "`<`" and "`>`".

Important note: an unsatisfied condition causes all text enclosed in the angle brackets to be discarded. Because normal assembly functions are suspended while the text of an unsatisfied conditional is being discarded, you must be careful not to have any superfluous angle brackets lying around, even if they seem to be in comments. In an unsatisfied conditional, MACRO discards text counting only bracket pairs to locate the end of the discarded text.

Blocks of conditional code can be nested, as parentheses can be nested.

To give some specific examples, suppose the symbol `TOPS10` has value 1. Then,

```
IFN TOPS10,<     MOVEI   1,2 >
IFE TOPS10,<     MOVEI   2,3 >
IFL TOPS10,<     MOVEI   3,4 >
IFG TOPS10,<     MOVEI   4,5 >
```

would have the effect of assembling the following:

```
        MOVEI     1,2
        MOVEI     4,5
```

17.2.2 The Definitional Conditionals

Another conditional assembly feature of MACRO is the ability to give an otherwise undefined symbol a default value. The conditional `IFNDEF` is satisfied if it is followed by the name of a symbol that has no definition. `IFNDEF` means *IF Not DEFined*. (There is also a conditional named `IFDEF`, meaning *IF DEFined*.)

In the program that follows, an `IFNDEF` conditional will be used to assign a default value to the symbol named `TOPS10`. The symbol `TOPS10`, which defaults to `0`, signifies which operating system to assemble code for. When `TOPS10` is set to `1`, code for the TOPS–10 system is assembled; when set to `0`, TOPS–20 code is made. A symbol that is used to control conditional assembly is frequently referred to as an *assembly switch* or just a *switch*.

To override the default setting of this switch, merely edit the file and insert a line containing

```
TOPS10==1
```

before the line with the `IFNDEF`. It is also possible to have switch settings present in a *header file*, so that various assemblies can be made without editing the principal source file.

17.2.3 Macros to Control Conditional Assembly

Sprinkling `IFN TOPS10` and `IFE TOPS10` throughout a program makes the program more difficult to read. To avoid this complication, we can define a pair of macros, `T10` and `T20`. The `T10` macro will

expand to `IFN TOPS10,`, which is satisfied when TOPS–10 code is being assembled. The `T20` macro expands to `IFE TOPS10,`, which is satisfied during the assembly of the TOPS–20 version.

The TOPS–10 portion of the following program appears without explanation. If you find it necessary to program for TOPS–10, you should refer to the DECsystem–10 System Calls Manual for an explanation of the Monitor UUOs that are used as system call instructions.

17.3 Example 7 — Numeric Evaluator

The following program is by far the most complicated example that we have yet encountered. This is our first example in which we need to do more arithmetic than just counting. We shall make extensive use of subroutines to divide the program into smaller, more understandable, portions. Among the useful subroutines that will be demonstrated are routines for scanning the textual representation of numbers to convert the text to internal binary numbers. We also demonstrate the reverse process: converting an internal integer to an external text string in conventional notation for decimal integers; the decimal output subroutine is an important example of the usefulness of a recursive subroutine. Additionally, some realistic examples of assembler macros and conditional assembly appear in this program.

This program will read an arithmetic expression composed of numbers and the operators `+`, `-`, `*`, and `/`. The program will compute the value of the expression and output the result.

This program has only a very limited set of features. For one thing, expressions containing spaces and parentheses are illegal. Another limitation is that strict left–to–right evaluation is used; operator precedence, as found in most expression evaluators, is absent from this example. A third limitation is that no testing for integer overflow occurs at any point in the calculation.

The range for representable integers is from decimal `-34359738368` to `+34359738367`, or, in octal, from `400000000000` to `377777777777`. Another limitation of this program is that the largest representable negative number (`-34359738368`) cannot be printed out.

This program uses conditional assembly and macros to make the TOPS–10 and TOPS–20 sources more nearly alike. It must be stated that this program does not present the TOPS–10 input/output facilities in the best possible light; rather it is written with the idea that the two different systems should be accommodated with the least total effort.

17.3.1 Synthesis of the Main Program

We begin the development of this program by creating some definitions that will be useful in controlling the conditional assembly of the program.

```
IFNDEF TOPS10,<TOPS10==0>               ;Default to TOPS-20 Assembly
DEFINE  T10     <IFN TOPS10,>
DEFINE  T20     <IFE TOPS10,>
```

The first line ensures that the symbol `TOPS10` will have some value. By default, the value will be zero, selecting the TOPS–20 assembly version. The next two lines define the `T10` and `T20` macros that we shall use to signify code for TOPS–10 and TOPS–20 respectively.

We continue the development of this program by creating a main program that exhibits the general structure that we want.

```
START:  RESET
RESTRT: MOVE    P,[IOWD PDLEN,PDLIST]  ;initialize stack (after error)
NEXT:   HRROI   A,PROMPT               ;prompt for a line of input
        TTYSTR                         ;macro for string output to terminal
        CALL    GETLIN                 ;read entire line into buffer
        MOVE    W,[POINT 7,BUFFER]     ;initialize scan of line
        CALL    EVAL                   ;evaluate line, result to A, skip
        JRST    STOP                   ;empty line.  exit now.
PRINT:  CALL    DECOUT                 ;print result
        HRROI   A,CRLF                 ;print new line
        TTYSTR
        JRST    NEXT                   ;get another line.

STOP:
T10<    EXIT    >                      ;stop program (TOPS-10)

T20<    HALTF                          ;stop program (TOPS-20)
        JRST    START >                ;in case of CONTINUE (T20)
```

The main program consists of initialization, a prompt for input, a call to GETLIN, which reads a line from the terminal, a call to EVAL to evaluate the expression, and finally a call to DECOUT which prints the result. After printing a result, the program loops to NEXT, where another prompt is given. The program loops through these functions until an empty line is input, whereupon, the program jumps to STOP where it stops running.

The label RESTRT will be referenced later as part of the error recovery in the program. The label PRINT isn't needed by this program; we include it because we will use it as a reference point in our discussion of DECOUT.

The label STOP appears before conditional code. In the TOPS–20 case, STOP labels a HALTF JSYS. For TOPS–10, the code at STOP is an EXIT UUO to terminate the execution of the program.

The main program is intentionally very short. It reveals the connection between the major subroutines (GETLIN, EVAL, and DECOUT) without supplying all the details that would obscure the structure of the program. By placing most of the functionality of this program into subroutines, we have provided the reader with a short and easy to understand main program. At the same time we have divided the program into a group of parts or subprograms each less complicated than the entire program. We can now focus our attention on the problem of writing each subprogram. In this form of problem solving, we first divide the original problem into more manageable subproblems, and then attack each of these smaller problems in turn.

17.3.2 Terminal Input and Output

The GETLIN subroutine is used for terminal input. Two macros, TTYSTR and TTYCHR are used for terminal output.

The GETLIN subroutine reads one line of terminal input into the buffer area called BUFFER. In the TOPS–20 version, this is accomplished by the RDTTY JSYS. In the TOPS–10 version, a line is read by repeated calls on the INCHWL UUO. The characters read by INCHWL are deposited into the buffer area to simulate the results of a RDTTY JSYS. Again, please note that if this program were intended for only the TOPS–10 environment, the input and output structure would be somewhat different.

The TTYSTR macro sends a string to the terminal. For TOPS–20 the string descriptor is placed in

register A by the caller; PSOUT sends the string to the terminal. For TOPS–10, the OUTSTR UUO wants the address of the first word of the ASCIZ string. This is obtained by using indexed addressing, in which the -1 in the left half of A is ignored.

The TTYCHR macro sends one character, in A, to the terminal. PBOUT does this in TOPS–20; for TOPS–10 an OUTCHR UUO is used.

Macros are used for these output instructions because the resulting code fragment is too short to make sense as a subroutine; the macro can be made to differ with the selection of a particular operating system. All system–dependant definitions are concentrated in this area of the program.

```
T10<
GETLIN: MOVE    W,[POINT 7,BUFFER]      ;Pointer to buffer to store things
        MOVEI   B,BUFLEN*5-1            ;Number of characters avail in buffer
INLOOP: INCHWL  A                       ;Get a character
        IDPB    A,W                     ;Store in buffer
        CAIE    A,12                    ;LF seen yet?
        SOJG    B,INLOOP                ;No, loop unless buffer full.
        MOVEI   A,0                     ;Add null to end string
        IDPB    A,W
        RET

DEFINE  TTYSTR  <OUTSTR  (A)>           ;Output string to terminal

DEFINE  TTYCHR  <OUTCHR  A>             ;Output a character to terminal

>;T10 IO routines

T20,<
GETLIN: HRROI   A,BUFFER                ;setup for RDTTY
        MOVEI   B,BUFLEN*5-1            ;
        HRROI   C,PROMPT                ;
        RDTTY
        ERJMP   ELIN
        RET

ELIN:   HRROI   A,[ASCIZ/Error from RDTTY.
/]
        PSOUT
        JRST    STOP

DEFINE  TTYSTR  <PSOUT>                 ;Output string to terminal

DEFINE  TTYCHR  <PBOUT>                 ;Output a character to terminal

>;T20 IO routines
```

17.3.3 Decimal Output and Recursive Subroutines

The DECOUT routine is used to convert an internal format binary number to a string of characters corresponding to the equivalent decimal number. The main idea used in this routine is the same as one presented in our earlier discussion of conversion between number systems, Section 4.6, page 35.

Essentially, when we divide a number, X, by decimal 10, the remainder is the units digit of the decimal representation of X; the quotient is a number that represents all the more significant digits of X. When we divide the quotient by 10, the remainder is the tens digit; the second quotient contains all the other more significant digits. We stop forming remainder digits when the quotient becomes zero.

Observe that this process, in which we successively divide the input number by 10, produces the result digits in reverse order. That is, the first remainder is the units digit. In conventional printing, the units digit should be output last. In order to reverse the digits we will use a pushdown stack.

With these remarks as a preface, we can now introduce the DECOUT routine. It is somewhat complicated, and somewhat subtle. We shall explain further below. The notation ^D10 that appears here is the way we tell MACRO that a number (in this case 10) is decimal. Please note that ^D represents the two distinct characters ^ and D; it is *not* a representation of CTRL/D.

```
;DECOUT - decimal output printer.  Call with number to be printed in A.
DECOUT: JUMPGE  A,DECOT1                ;Jump unless printing negative
        PUSH    P,A                     ;Save the number
        MOVEI   A,"-"                   ;Load a minus sign
        TTYCHR                          ;Send the minus to the terminal
        POP     P,A                     ;Restore the number
        MOVN    A,A                     ;Make argument positive
DECOT1: IDIVI   A,^D10                  ;Quotient to A, Remainder to B
        PUSH    P,B                     ;Save remainder
        SKIPE   A                       ;Skip if we have divided enough
        CALL    DECOT1                  ;Must divide some more
        POP     P,A                     ;Pop a remainder digit
        ADDI    A,"0"                   ;Convert digit to an ASCII character
        TTYCHR                          ;Print character
        RET
```

DECOUT starts innocently enough. The JUMPGE tests to see whether the argument, register A, is non–negative. When A is non–negative, the program jumps to DECOT1. When A contains a negative number, register A is pushed onto the stack for temporary safe–keeping; a minus sign is loaded into A and printed; the data item is popped back into A and negated (i.e., becomes positive). Now that A is positive, the program falls into DECOT1.[2]

At DECOT1, A contains a non–negative number. Divide A by decimal 10. As we have discussed above, the result of this division is a remainder that is the least significant (i.e., rightmost) digit of the decimal number. The other digits of the result can be formed from the quotient.

The remainder, which appears in B (that is, A+1), is pushed onto the stack. As we saw in the versions of example 4, stacks are a good way to reverse the order of characters. We shall use this reversing property of the stack when it is time to print the result. Bear in mind, we calculate answer digits in a right–to–left order, but we must print them in a left–to–right sequence.

The quotient is left in A. If A hasn't yet become zero, another call to the routine DECOT1 is made. That call will cause the digits of the present quotient to be calculated and printed. Eventually, when this call to DECOT1 returns we will then pop the remainder digit from the stack and print it.

When A is reduced to zero the unwinding begins. The most recent remainder is popped. The remainder is a number in the range from 0 to 9. Now, remember that numbers are not characters;

[2]It is the failure of the MOVN to change -34359738368 to a positive number that accounts for the problem, mentioned above, of not being able to print that number correctly.

the remainder must be converted to the corresponding character. To give a specific example, if the remainder were the number 5 then the character "5" should be printed. We accomplish this conversion by the simple expedient of adding the ASCII code for the character "0" to the remainder digit.

Perhaps at this point a specific example with some diagrams would be helpful. At PRINT, register P, the stack pointer, contains -PDLEN,,PDLIST-1; nothing is presently on the stack. Suppose register A contains the octal number 173 (decimal 123). The instruction PUSHJ P,DECOUT is executed (we wrote this instruction as CALL DECOUT). The stack pointer in P is advanced to 1-PDLEN,,PDLIST; the return address, PRINT+1, is pushed onto the stack (at location PDLIST); the program jumps to DECOUT.[3]

```
PDLIST: PRINT+1          ⟵ P
```

At DECOUT, A is positive; the program jumps to DECOT1. The contents of register A are divided by the immediate constant decimal 10. In this case, the quotient is octal 14; the remainder is 3. The remainder that is in register B is pushed onto the stack. The stack now looks like this:

```
PDLIST: PRINT+1
        3                ⟵ P
```

Since A now contains 14, the instruction SKIPE A fails to skip. The instruction PUSHJ P,DECOT1 is executed. That instruction pushes the return address, DECOT1+4, onto the stack, and jumps to DECOT1:

```
PDLIST: PRINT+1
        3
        DECOT1+4         ⟵ P
```

The program is again at DECOT1; A contains octal 14. The number in A is divided by decimal 10. The quotient is 1; the remainder is 2. The 2 is pushed on the stack. Because A has non–zero contents, the instruction CALL DECOT1 is executed again. The return address, DECOT1+4, is pushed; the program jumps to DECOT1. The stack is now like this:

```
PDLIST: PRINT+1
        3
        DECOT1+4
        2
        DECOT1+4         ⟵ P
```

Once more, the program finds itself at DECOT1. This time, A contains 1. Dividing by 10 yields a quotient of 0 and a remainder of 1. The 1 is pushed onto the stack.

[3] If executed in section 0 the program flags and PC are pushed in a single word. The diagrams that follow presume execution in a non–zero section. A symbolic expression such as PRINT+1 should be understood as a 30–bit value.

```
PDLIST: PRINT+1
        3
        DECOT1+4
        2
        DECOT1+4
        1                ⟵ P
```

Now, A contains 0. The SKIPE A will skip over the call to DECOT1; the program begins unwinding the stack. The instruction POP P,A will remove the most recent remainder from the stack. The last remainder that was computed is 1; that 1 is popped from the stack. The number 1 is converted to the ASCII character "1" (by adding the value of the ASCII character "0" to it); the resulting character is sent to the terminal.

After sending this result, the program executes the RET instruction. The stack looks like this:

```
PDLIST: PRINT+1
        3
        DECOT1+4
        2
        DECOT1+4          ⟵ P (before the RET)
        1                 (this was popped and output first)
```

The RET instruction (that is, POPJ P,) sets the program counter to DECOT1+4 and decrements the stack pointer. The program jumps to DECOT1+4 with the stack looking like this:

```
PDLIST: PRINT+1
        3
        DECOT1+4
        2                 ⟵ P (after the first RET)
        DECOT1+4          (this has already been popped)
        1                 (this was popped and output first)
```

At DECOT1+4 for the second time, another remainder (this time it is 2) is popped into A; it is converted to a character and sent to the output line. When the RET is executed, the program counter is set once more to DECOT1+4. At DECOT1+4, after the second RET, the stack looks like this:

```
PDLIST: PRINT+1
        3                  ⟵ P (after second RET)
        DECOT1+4           (this has already been popped)
        2                  (This was popped and printed second)
        DECOT1+4           (This has been popped)
        1                  (This was popped and printed first)
```

The third remainder to be popped is 3. After the character "3" is sent to the output line, the final RET exits from the DECOUT routine and returns to the caller at PRINT+1.

17.3.3.1 Recursion

DECOUT (DECOT1) is an example of a *recursive subroutine*. Recursion is a technique in which a subroutine calls itself in order to solve a problem. Although it may seem unlikely that such a technique could be an effective computational tool, recursion is extremely useful in many cases.

As a computational technique, a recursive procedure requires that at least two conditions be met. First, if a routine calls itself recursively, it must simplify the problem before making the recursive call. The meaning of *simplify* is rather broad. In the case of DECOUT, the complexity of the problem is essentially the number of digits that remain to be printed. Each call reduces by one the number of digits to print.

The second requirement of a recursive procedure is that it must contain some simplest case in which the result of the computation is known without resorting to further recursive calls. In this example, this simple case is when the answer consists of precisely one digit.

If a procedure cannot simplify the problem, a recursive call will not help anything. If a procedure cannot recognize the simple case that it can handle without further recursion, it cannot terminate.

We shall have further occasion to talk of recursion. In every case you may expect an explanation of how the procedure simplifies the problem, and what the simplest case might be.

17.3.4 Expression Evaluation

The EVAL routine scans the input line for numbers and for operators. The object of EVAL is to collapse the input line into one number that represents the value of the expression that it has scanned. Recall that in this program, evaluation is performed on a strict left–to–right basis, without regard for what is called the *precedence* of the different arithmetic operators.

For example, when presented with the expression 6-4*3 the program will call the decimal input routine, DECIN, which we will investigate below. The number 6 and the operator "-" will be returned. The first partial result is 6, the "-" is saved for later. The second call to DECIN results in the number 4 and the operator "*". The two partial results 6 and 4 are combined via the "-" operator to form 2. The 2 is the second partial result. That partial result and the "*" operator are saved for later. The third call to DECIN will locate the number 3 and the line feed character as the delimiter. The 3 is combined with the previous result (2) by means of the "*" operator. The result, 6, is the third partial result. The line feed is not recognized as an arithmetic operator, and signals the end of the line. The last partial result, 6, is the value of this expression.

The EVAL routine that accomplishes this scanning and evaluation is show below:

```
EVAL:   CALL    DECIN                   ;Get a number and an operator
        TRNN    FL,DIGF                 ;Was a number present?
        JRST    EVAL2                   ;No. error unless end of line.
        AOS     (P)                     ;Number seen.  EVAL will skip
EVAL1:  MOVEM   B,OP1                   ;Save as first operand
        MOVSI   B,-OPTLNG               ;Lookup the operator in OPTAB
        CAME    A,OPTAB(B)              ;Compare to one in OPTAB
        AOBJN   B,.-1                   ;No match, keep scanning.
        JUMPGE  B,EVAL2                 ;Jump if AOBJN exhausted.
                                        ;    (operator unknown)
        MOVE    B,OPINS(B)              ;Get the Instruction to XCT
        MOVEM   B,X1                    ;Save instruction to XCT
        CALL    DECIN                   ;Get the second operand/operator
        EXCH    B,OP1                   ;2nd operand to mem, first to AC
        XCT     X1                      ;Execute instr to do first operation.
        JRST    EVAL1                   ;Value of (first OP1 second) in B,
                                        ;   second operator in A.  Loop.

EVAL2:  CAIE    A,12                    ;End of line here?
        JRST    ERR                     ;No. unrecognized operator.
        MOVE    A,OP1                   ;End of line.  Return result in A
        RET

OPTAB:  "+"                             ;table of the known operator characters
        "-"
        "*"
        "/"
OPTLNG==.-OPTAB                         ;length of OPTAB table

OPINS:  ADD     B,OP1                   ;table of instructions to execute
        SUB     B,OP1                   ;corresponding to OPTAB
        IMUL    B,OP1
        IDIV    B,OP1
```

The `EVAL` routine calls `DECIN` to get a number and an operator. If no first number is found, the flag `DIGF` will be zero; the program will jump to `EVAL2`, signifying an error or a blank line. If a first number is seen, the `AOS (P)` is executed to cause an eventual skip return. At `EVAL1`, the result that `DECIN` returned in register `B` is saved in `OP1`; this is the number that `DECIN` has scanned.

The operator (delimiter) that `DECIN` found following the number is returned in register `A`. That operator is looked up in the table of operators (`OPTAB`, the *OPerator TABle*). This lookup is much the same as the `ISVOW` routine we have seen before. If the operator character is one of the known operators, the corresponding instruction from the `OPINS` (*OPerator INStruction*) table is selected and saved in `X1`. The corresponding instruction is selected by using the same index value to index `OPINS` as was used to match an entry in `OPTAB`.

`DECIN` is called to read the second operand. The second operand is exchanged with `OP1` (subtract and divide are not commutative, so the second argument must be in memory and the first in the accumulator). Then the instruction stored in `X1` is executed. This instruction performs the selected operation on the two operands (in `B` and `OP1`), leaving the result in `B`. The program loops to `EVAL1` where `B` is stored once more in `OP1` while the second operator is looked up.

When the delimiter character cannot be found in the table, the program jumps to `EVAL2`. If `A`

contains a line feed, all is well; the contents of OP1 are returned in A. If A is some character other than a line feed, then an unrecognized delimiter has been seen. The program will jump to ERR and complain.

17.3.5 Macros for Data Structures

The operators known to EVAL are +, -, *, and /. A table containing the ASCII values of these characters is built at OPTAB. This table is searched via AOBJN as we saw in example 6–B. There is a difference this time: we want to connect different characters with different operations. We must connect "+" with ADD, "/" with IDIV, etc.

We have shown how to build these two tables by hand: the first, OPTAB, containing the ASCII characters; the second, OPINS, containing instructions. When we have two tables such as these it is possible, by inadvertance, to break the correspondence between characters and operations.

To avoid any problem in which this correspondence is upset, we can use the macro capability of the assembler to help us keep these tables straight. The following lines of assembler instructions are equivalent to the OPINS and OPTAB tables that we defined above. Note that by using macros we have placed each operator character next to the corresponding instruction. This makes it much harder for us to accidentally scramble the tables.

```
;Macros to construct OPTAB (operator character table) and
;                    OPINS (corresponding instruction table).
;
DEFINE OPMAC<
        XX(<"+">,<ADD B,OP1>)
        XX(<"-">,<SUB B,OP1>)
        XX(<"*">,<IMUL B,OP1>)
        XX(<"/">,<IDIV B,OP1>)
>
DEFINE XX(A,B)<A>               ;Set XX to select the character
OPTAB:  OPMAC                   ;Define the OPerator character TABle
OPTLNG==.-OPTAB                 ;OPerator Table LeNGth
DEFINE XX(A,B)<B>               ;Set XX to select the instruction
OPINS:  OPMAC                   ;Define the OPerator INStruction table
```

The OPMAC macro contains one line for each operation character. These lines are themselves calls to another macro named XX. Each line in OPMAC clearly shows the correspondence between an operation character and a PDP–10 instruction.

First, the XX macro is defined to expand to its first argument. Then the mention of the name OPMAC calls XX four times to build the table called OPTAB. At this point, OPTLNG, the length of the table, is defined. Then XX is redefined; the new definition expands to the second argument to XX. Then when OPMAC is expanded again, four expansions of XX happen; these expansions of the XX macro produce a table of instructions at OPINS.

Although using these macros makes the text of the program more difficult to understand, these macros make it easier to expand or shrink the table without fear of breaking the program. Programs last a long time; we need to build them to withstand some tinkering.

17.3.6 Decimal Input Routine

The purpose of the DECIN routine is to read a series of characters from the input line and translate those characters to an internal format binary number.

Suppose the number 123 is seen. If we were scanning from right to left, we would know to multiply the 3 times 1, the 2 times 10 and the 1 times 100. However, it is more natural to scan from left to right, so we must think of some other way to perform this conversion. Suppose we have a register, B, that contains the number we have scanned thus far. It is natural to imagine that B contains successively the values 1, 12, and 123. These values are related to each other; of course, they are also related to the number 123 that we are scanning.

The plan is this: Initialize B to contain 0. Every time a digit is seen, multiply the old contents of B by 10 and add the new digit. This loop is implemented by the code at DECIN1. An abbreviated version of this loop appears here:

```
DECIN:  . . .
        MOVEI   B,0                     ;Accumulate the result here
DECIN1: ILDB    A,W                     ;Get a character from input
        CAIL    A,"0"                   ;Skip if not a digit
        CAILE   A,"9"                   ;Skip if this is a digit
        JRST    DECIN2                  ;Not a digit
        IMULI   B,^D10                  ;Multiply old accumulation by 10
        ADDI    B,-"0"(A)               ;Add the number corresponding to chr
        . . .
        JRST    DECIN1                  ;Look for the rest of the number.
```

With B being 0, the character "1" is seen. The old contents of B are multiplied by 10 (the result is 0). The character "1" must be converted to a number (by subtracting the character "0" from it); the number is added to B. Register B now contains the number 1. Next, the character "2" is seen. B becomes 10 and 2 is added, making 12 (stored in binary of course). When the character "3" comes in, the 12 is multiplied by 10 to make 120 and 3 is added, to form 123. If some character, not a digit, follows the "3", the DECIN routine will return with register B containing the binary for 123.

The conversion of the character in A to a digit and the addition of this digit to the partial result in B is accomplished in one instruction, ADDI B,-"0"(A). This instruction uses the effective address computation in an interesting way. The Y field of this instruction is assembled as 777720, that is, an 18–bit representation of octal −60, the negative of the value corresponding to the character "0". If register A contains a character that represents a digit, say the character "5", then the effective address will be 777720 + 65. Thus, the effective address is precisely the *number* corresponding to the character in A. This number is the immediate operand of the ADDI instruction. So, this instruction adds to B the number corresponding to the character in A.

The remainder of DECIN is concerned with things like deciding if a minus sign means a negative number or a subtraction operator, etc. DECIN shows us another example of the use of the test instructions. The flags, in the right half of register FL, are called NEGF and DIGF, for *NEGative Flag* and *DIGit seen Flag* respectively.

```
;Define Flags for FL right half.
NEGF==1                                 ;Negative sign has been seen
DIGF==2                                 ;A digit has been seen
```

```
;DECIN - read decimal number from input stream
;Return the value in B, the delimiter character in A.
DECIN:  TRZ     FL,NEGF!DIGF            ;Neither "-" nor digits seen yet
        MOVEI   B,0                     ;Accumulate the result here
DECIN1: ILDB    A,W                     ;Get a character from input
        CAIL    A,"0"                   ;Skip if not a digit
        CAILE   A,"9"                   ;Skip if this is a digit
        JRST    DECIN2                  ;Not a digit
        IMULI   B,^D10                  ;Multiply old accumulation by 10
        ADDI    B,-"0"(A)               ;Add the number corresponding to chr
        TRO     FL,DIGF                 ;Set flag that we saw a digit
        JRST    DECIN1                  ;Look for the rest of the number.

DECIN2: CAIN    A,15                    ;Here with something other than digit
        JRST    DECIN1                  ;Throw out CR
        CAIN    A,"-"                   ;Minus or negative sign?
        JRST    DECIN3                  ;Yes, "-" seen.  Think harder.
        TRNN    FL,DIGF                 ;Have any digits been seen?
        RET                             ;No.  Just return the delimiter
DECIN4: TRZE    FL,NEGF                 ;Was unary "-" seen?
        MOVN    B,B                     ;Yes,  negate the result
        RET

;here when a minus sign is seen
DECIN3: TRNE    FL,DIGF                 ;Have any digits been seen yet?
        JRST    DECIN4                  ;Yes.  Must be subtract operator
        TRON    FL,NEGF                 ;No.  Must be negative number
        JRST    DECIN1                  ;Collect the rest of the number
ERR:    HRROI   A,[ASCIZ/Illegal Expression
/]
        TTYSTR
        JRST    RESTRT                  ;Restart
```

Both flags are cleared by the `TRZ` instruction at `DECIN`. The exclamation mark in the expression `NEGF!DIGF` means the inclusive `OR` of the two flag values. The `TRZ` instruction will clear both flags. When a digit is seen, `DIGF` is set.

When a non–digit is seen, the code at `DECIN2` behaves as follows. First, the carriage return is thrown away. If a minus sign is seen, the routine jumps to `DECIN3`.

If the character was not a minus sign, then that character is the delimiter. If no digits have yet been seen, the character is returned. If a digit was seen, then the `NEGF` flag is examined. If `NEGF` is set, register `B` is negated. The delimiter is returned in register `A`.

At `DECIN3`, if digits have already been seen, the minus character is a delimiter. The program jumps to `DECIN4` to return the delimiter in `A` and the result in `B`. If no digits have yet been seen, `NEGF` is set, signifying that a negative number is being scanned. If `NEGF` has already been set, that is an error condition: two minus signs appear at the start of a number.

17.3.7 Complete Program for Example 7

```
        TITLE   NUM - NUMERIC EVALUATOR - Example 7

IFNDEF TOPS10,<TOPS10==0>               ;Default to TOPS-20 Assembly
DEFINE  T10     <IFN TOPS10,>
DEFINE  T20     <IFE TOPS10,>

T20<    SEARCH  MONSYM                  ;add TOPS-20 JSYS definitions >;T20


FL=0                                    ;define symbolic names. FL is flags
A=1
B=2
C=3
D=4
W=5
X=6
Y=7
P=17

OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ P,]

;FLAG NAMES (FL RIGHT)
NEGF==1                                 ;Negative sign has been seen
DIGF==2                                 ;A digit has been seen

T20<    .PSECT  CODE/RONLY,1002000      ;Set for code >;T20
T10<    TWOSEG   400000                 ;Set code into high segment >;T10

START:  RESET
RESTRT: MOVE    P,[IOWD PDLEN,PDLIST]   ;initialize stack (after error)
NEXT:   HRROI   A,PROMPT                ;prompt for a line of input
        TTYSTR                          ;string output to terminal
        CALL    GETLIN                  ;read entire line into buffer
        MOVE    W,[POINT 7,BUFFER]      ;initialize scan of line
        CALL    EVAL                    ;evaluate line, result to A, skip
        JRST    STOP                    ;empty line.  exit now.
PRINT:  CALL    DECOUT                  ;print result
        HRROI   A,CRLF                  ;print new line
        TTYSTR
        JRST    NEXT                    ;get another line.

STOP:
T10<    EXIT    >                       ;stop program (TOPS-10)

T20<    HALTF                           ;stop program (TOPS-20)
        JRST    START >                 ;in case of CONTINUE (T20)
```

```
T10<
GETLIN: MOVE    W,[POINT 7,BUFFER]      ;Pointer to buffer to store things
        MOVEI   B,BUFLEN*5-1            ;Number of characters avail in buffer
INLOOP: INCHWL  A                       ;Get a character
        IDPB    A,W                     ;Store in buffer
        CAIE    A,12                    ;LF seen yet?
        SOJG    B,INLOOP                ;No, loop unless buffer full.
        MOVEI   A,0                     ;Add null to end string
        IDPB    A,W
        RET

DEFINE  TTYSTR  <OUTSTR  (A)>           ;Output string to terminal

DEFINE  TTYCHR  <OUTCHR  A>             ;Output a character to terminal

>;T10 IO routines

T20<
GETLIN: HRROI   A,BUFFER                ;Setup for RDTTY
        MOVEI   B,BUFLEN*5-1            ;
        HRROI   C,PROMPT                ;
        RDTTY
        ERJMP   ELIN
        RET

ELIN:   HRROI   A,[ASCIZ/Error from RDTTY.
/]
        PSOUT
        JRST    STOP

DEFINE  TTYSTR  <PSOUT>                 ;Output string to terminal

DEFINE  TTYCHR  <PBOUT>                 ;Output a character to terminal

>;T20 IO routines

;DECOUT - decimal output printer.  Call with number to be printed in A.
DECOUT: JUMPGE  A,DECOT1                ;Jump unless printing negative
        PUSH    P,A                     ;Save the number
        MOVEI   A,"-"                   ;Load a minus sign
        TTYCHR                          ;Send the minus to the terminal
        POP     P,A                     ;Restore the number
        MOVN    A,A                     ;Make argument positive
DECOT1: IDIVI   A,^D10                  ;Quotient to A, Remainder to B
        PUSH    P,B                     ;Save remainder
        SKIPE   A                       ;Skip if we have divided enough
        CALL    DECOT1                  ;Must divide some more
        POP     P,A                     ;Pop a remainder digit
        ADDI    A,"0"                   ;Convert digit to an ASCII character
        TTYCHR                          ;Print character
        RET
```

```
EVAL:   CALL    DECIN                   ;Get a number and an operator
        TRNN    FL,DIGF                 ;Was a number present?
        JRST    EVAL2                   ;No. error unless end of line.
        AOS     (P)                     ;Number seen.  EVAL will skip
EVAL1:  MOVEM   B,OP1                   ;Save as first operand
        MOVSI   B,-OPTLNG               ;Lookup the operator in OPTAB
        CAME    A,OPTAB(B)              ;Compare to one in OPTAB
        AOBJN   B,.-1                   ;No match, keep scanning.
        JUMPGE  B,EVAL2                 ;Jump if AOBJN exhausted. op'tor unknown
        MOVE    B,OPINS(B)              ;Get the Instruction to XCT
        MOVEM   B,X1                    ;Save instruction to XCT
        CALL    DECIN                   ;Get the second operand/operator
        EXCH    B,OP1                   ;2nd operand to mem, first to AC
        XCT     X1                      ;Execute instr to do first operation.
        JRST    EVAL1                   ;Value of (first OP1 second) in B,
                                        ;   second operator in A.  Loop.

EVAL2:  CAIE    A,12                    ;End of line here?
        JRST    ERR                     ;No. unrecognized operator.
        MOVE    A,OP1                   ;End of line.  Return result in A
        RET

;Macros to construct OPTAB (operator character table) and
;                    OPINS (corresponding instruction table).
;
DEFINE OPMAC<
        XX(<"+">,<ADD B,OP1>)
        XX(<"-">,<SUB B,OP1>)
        XX(<"*">,<IMUL B,OP1>)
        XX(<"/">,<IDIV B,OP1>)
>

DEFINE XX(A,B)<A>
OPTAB:  OPMAC
OPTLNG==.-OPTAB
DEFINE XX(A,B)<B>
OPINS:  OPMAC

;DECIN - read decimal number from input stream.
;Return the value in B, the delimiter character in A.
DECIN:  TRZ     FL,NEGF!DIGF            ;Neither "-" nor digits seen yet
        MOVEI   B,0                     ;Accumulate the result here
DECIN1: ILDB    A,W                     ;Get a character from input
        CAIL    A,"0"                   ;Skip if not a digit
        CAILE   A,"9"                   ;Skip if this is a digit
        JRST    DECIN2                  ;Not a digit
        IMULI   B,^D10                  ;Multiply old accumulation by 10
        ADDI    B,-"0"(A)               ;Add the number corresponding to chr
        TRO     FL,DIGF                 ;Set flag that we saw a digit
        JRST    DECIN1                  ;Look for the rest of the number.
```

```
DECIN2: CAIN    A,15                    ;Here with something other than digit
        JRST    DECIN1                  ;Throw out CR
        CAIN    A,"-"                   ;Minus or negative sign?
        JRST    DECIN3                  ;Yes, "-" seen.  Think harder.
        TRNN    FL,DIGF                 ;Have any digits been seen?
        RET                             ;No.  Just return the delimiter
DECIN4: TRZE    FL,NEGF                 ;Was unary "-" seen?
        MOVN    B,B                     ;Yes,  negate the result
        RET

DECIN3: TRNE    FL,DIGF                 ;Have any digits been seen yet?
        JRST    DECIN4                  ;Yes.  Must be subtract operator
        TRON    FL,NEGF                 ;No.  Must be negative number
        JRST    DECIN1                  ;Collect the rest of the number
ERR:    HRROI   A,[ASCIZ/Illegal Expression
/]
        TTYSTR
        JRST    RESTRT                  ;Restart

CRLF:   BYTE    (7)15,12
PROMPT: ASCIZ   /Type an arithmetic expression: /

T20< .PSECT DATA,1001000 ;data into a separate psect >;T20
T10<    RELOC                           ;data into the low segment >;T10
OP1:    0
X1:     0
BUFLEN==40
BUFFER: BLOCK   BUFLEN
PDLEN==40
PDLIST: BLOCK   PDLEN

        END     START
```

The lines that define the labels `OP1` and `X1` specify initial values of zero. It is necessary to put something there to force the assembler to leave room for one word at each of the locations `OP1` and `X1`.

17.4 Exercises

17.4.1 Recursive Computation of the Sine Function

The following subroutine makes use of the trigonometric identity:

$$\sin(x) = 3 \times \sin(x/3) - 4 \times [\sin(x/3)]^3$$

to compute values of the sine function.[4] Discuss this program. Give convincing evidence that this procedure is effective, i.e., that it actually works properly.

[4]This subroutine is a modified version of the subroutine appearing in item 158 of Beeler, Gosper, and Schroeppel, *HAKMEM*, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo number 239.

```
SIN:    MOVM    B,A             ;absolute value of argument
        CAMG    B,[0.00017]     ;for small X, sin(X) = X
        RET                     ;so return sin(A) in A
        FDVRI   A,(3.0)         ;compute A/3
        CALL    SIN             ;Compute sin(A/3).  Result in A.
        MOVN    B,A             ;Negative sin(A/3) in B
        FMPR    B,A             ;-[sin(A/3) squared]
        FSC     B,2             ;-4.0*[sin(A/3) squared]
        FADRI   B,(3.0)         ;3.0-4.0*[sin(A/3) squared]
        FMPR    A,B             ;A gets 3*sin(A/3)-4[sin(A/3) cubed]
        RET
```

17.4.2 Russian Multiplication

Visitors to Russia in the nineteenth century discovered that the peasants there used an unusual method to multiply numbers. The Russian multiplication algorithm, applicable to positive numbers, is described below.[5]

Begin by writing three columns. Put the original multiplicand at the top of the first column. Put the original multiplier at the top of the second column. The third column will contain the numbers that are discarded during the multiplication steps, as we shall further explain.

Each multiplication step occurs as follows. If the entry in the multiplier column is an odd number, copy the number from the multiplicand column to the discard column. Next, write a new row: double the present entry in the multiplicand column and enter that value as the next multiplicand; halve the multiplier entry and place that value at the bottom of the multiplier column. When halving the multiplier entry discard any fraction that might be generated.

Repeat the multiplication step until the multiplier becomes zero.

Add together all the numbers that were placed in the discard column. This sum is the product of the original multiplier and multiplicand.

Example:

Multiplicand	Multiplier	Discard
312	57	312
624	28	
1248	14	
2496	7	2496
4992	3	4992
9984	1	9984
19968	0	
		17784

- By hand, work through the problem 54 times 79 using this method. Verify that you have the

[5] Claims to the contrary notwithstanding, the Russians did not invent this method. It was used by Egyptian mathematicians in 1800 B.C.

correct result.

- Write a program that accepts a multiplicand and a multiplier from the terminal and performs this algorithm, printing a table of results similar to the one shown above. You may wish to print a fourth column containing the running total of the discarded values. Use the `ASH` instruction to effect the doubling and halving. To help format the output into right–justified columns, you might look at the `DECFIL` subroutine in example 8.
- Explain how this method works; give a convincing discussion that shows that these steps produce arithmetically correct results. Is this algorithm analogous to anything we have discussed concerning the representation of data in the computer?
- Discuss how this algorithm can be modified to cope with negative numbers.

17.4.3 Efficient Exponentiation

Modify the "Russian Multiplication" algorithm, explained above, to apply it to exponentiation instead of multiplication.

17.4.4 Expand the Decimal Printout Routine

In the text we noted that `DECOUT` fails to convert `-34359738368` to ASCII characters. Fix `DECOUT` to handle this case.

Chapter 18

Local UUOs

The PDP–10 sets aside thirty–one operation codes that may be defined by the program for any purpose. Each of these codes is an LUUO, *Local Unimplemented User Operation*. These codes are numbered from 001 through 037 octal. The opcode appears right–adjusted in bits 0:8 of an instruction word.

There are differences between how the CPU handles LUUOs depending on whether they occur in in section 0 or in a non–zero sections.

- In the traditional machine (or in section zero), when one of the LUUO instructions is executed, the CPU performs the following operations. First, the effective address specified in the instruction is calculated. Then a copy of the instruction and its effective address is stored in location 40 of the program. The CPU then executes the instruction that it finds in location 41.

  The instruction image that is stored in location 40 has the same bits 0:12 as the instruction that was executed. Bits 13:17 are zero, and bits 18:35 reflect the results of the effective address calculation. This saves the program from having to calculate E.

  The instruction in 41 is presumed to be a subroutine call to the program's LUUO handling routine; either a JSR or a PUSHJ instruction may be used for this purpose. The instruction in 41 is executed as though, instead of an LUUO, an XCT 41 instruction had been performed. Thus, the program counter that is stored will point to the address following the LUUO.

- When an LUUO is executed in a non–zero section the CPU's action is more complicated: because the effective address of the LUUO is 30 bits, it can not be stored in a word that also contains the image of the LUUO's bits 0:12. An entirely different scheme is used.

  Prior to executing any LLUOs, the program must tell TOPS–20 the address of a four–word block of memory that it will use as the LUUO trap block. This is accomplished via the SWTRP% JSYS. The program initializes the fourth word of the block to contain the 30–bit address of the LUUO handler. Thereafter, when an LUUO is executed, the CPU stores three words of information about the LUUO in the block; then it jumps to the address specified in the fourth word. The LUUO block is depicted in Figure 18.1.

  The LUUO block contains the LUUO Opcode and AC, the old PC and flags (stored in a format compatible with the use ofXJRSTF to exit from the LUUO handler), and the effective address of the LUUO. Instead of providing an instruction to execute to jump to the LUUO handler, the programmer supplies the 30–bit address of the handler.

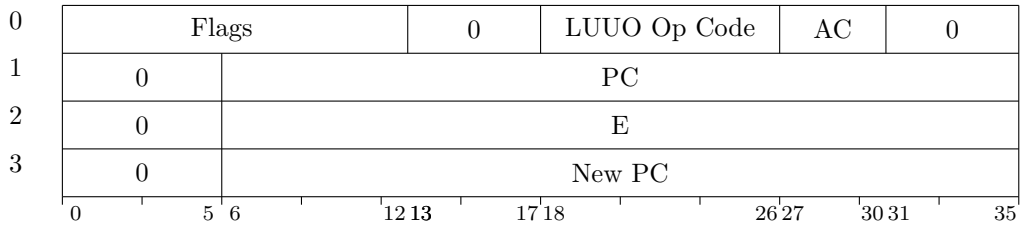


Figure 18.1: LUUO Block for Non–Zero Sections

No accumulators are stored in the process of getting to the LUUO handler. Also, no decoding of the operation code is performed. These functions are the responsibility of the LUUO handler.

Local UUOs provide a flexible (although somewhat costly) subroutine calling mechanism. Because of the high cost of getting into an LUUO, saving the accumulators and decoding the desired operation, an LUUO should perform some substantial amount of work.

Essentially, an LUUO is a convenient subroutine call in which the operation code describes which subroutine you want. This leaves the address field of the LUUO free to point to arguments. An LUUO may select a subfunction based on the accumulator field that is used in the LUUO; or it may find an argument or return a result in the accumulator that is specified. The address field of the LUUO may point to an argument or to an argument block, or it may encode a further function selection. The three fields, the accumulator, the address, and the opcode are at the disposal of the author of the LUUO handler to decode as s/he sees fit.

As there is only set of locations associated with the LUUO, the LUUOs are not perfectly reentrant. If recursive LUUOs are required then

- The LUUO handler is responsible for saving the contents of the first three locations in the LUUO block (or location `40`) before attempting a recursive LUUO.
- On exit from one LUUO, those locations must be restored.
- If the program uses the pseudo interrupt system (see Chapter 29), each interrupt level that intends to use LUUOs must save these locations on entry to the level and restore them upon exit.

In the example that follows, a Local UUO handler is provided to make the calling sequences for terminal input and output functions somewhat more palatable.

18.1 Example 8–A: Floating–Point Input and Output

Besides taking this opportunity to introduce an example of a Local UUO handler, we will demonstrate two useful subroutines for floating–point input and output. Although this is presented in the guise of a complete program, it really should be thought of as two useful subroutines, plus a typical LUUO handler. (The routines presented here are servicable, but they are not the very best way to perform these functions.)

As we have done before, we shall present the entire program first and then subject it to analysis.

This program presents the traditional or section zero program. It makes use of locations 40 and 41 in section 0. It omits .PSECT, etc. (A section of this chapter follows that will show how an LUUO handler can be implemented in a non–zero section.)

```
        TITLE   FLIO    Floating-point Input and Output (Example 8-A)
        SEARCH  MONSYM
        EXTERN  .JB41,.JBUUO

FL=0
A=1
B=2
C=3
D=4
W=5
P=17

;Flags for the right side of FL
DPOINT==1                               ;Decimal point has been seen
SIGNED==2                               ;+ or - seen already
NEG1==4                                 ;Minus number
NEG2==10                                ;Minus exponent
DIGF==20                                ;Digits have been seen

OPDEF   ERROR   [1B8]                   ;User-defined LUUOs.
OPDEF   TTYSTR  [2B8]
OPDEF   TTYCHR  [3B8]

OPDEF   CALL    [PUSHJ  P,]
OPDEF   RET     [POPJ   P,]

PDLEN==200
BUFLEN==100
PDLIST: BLOCK   PDLEN
BUFFER: BLOCK   BUFLEN
OUTBUF: BLOCK   BUFLEN
UUACAD: 0
PROMPT: 0

        SUBTTL  FLINP - Floating-point Input

;FLINP:
;Convert characters to internal format floating-point numbers.
;Call with a byte pointer to the input text string in W.
;Returns the number in B, the trailing delimiter in A.
;Accepts numbers of the form  "sdd.ddesdd",
;where "s" is a sign, +, or -, or absent,
;      "dd" are decimal digits, and
;      "e" is the letter "E" or "e".
;This routine may fail to produce an accurate result in certain cases.
```

```
FLINP:  SETZ    B,                      ;Accumulate result here
        MOVSI   C,(1.0)                 ;Divisor for digits after the DP
        TRZ     FL,DIGF!DPOINT!NEG1!NEG2!SIGNED         ;clear flags
FLINP1: ILDB    A,W                     ;get a byte of input
        CAIE    A,"+"                   ;plus or
         CAIN   A,"-"                   ;minus sign here?
          JRST  FLINSN                  ;Yes, go process sign characters
        CAIN    A,"."                   ;decimal point?
         JRST   FLINPT                  ;Handle Decimal point
        CAIE    A,"E"                   ;Exponent to come next?
         CAIN   A,"e"
          JRST  FLINEX                  ;Handle exponent
        CAIL    A,"0"                   ;Well, is it a digit?
         CAILE  A,"9"
          JRST  FLINRT                  ;nothing we know, return.

;note the similarity to the integer scan to read a decimal number
        TRNE    FL,DPOINT               ;Has a decimal point been seen yet?
         FMPRI  C,(10.0)                ;Yes.  Make Divisor Larger
        SUBI    A,"0"                   ;convert a character to a number
        FSC     A,233                   ;float it.  (the old way)
        FMPRI   B,(10.0)                ;floating immediate uses E,,0 as operand
        FADR    B,A                     ;Add new digit to previous accumulation
        TRO     FL,DIGF                 ;set we have seen a digit
        JRST    FLINP1

FLINRT: TRZE    FL,NEG1                 ;Here to return a number.
         MOVN   B,B                     ;Negate it if a minus sign was seen
        FDVR    B,C                     ;Divide to account for digits after DP
CPOPJ:  RET

;Here when a plus or minus sign is seen.
FLINSN: TRNE    FL,DIGF                 ;Have any digits been seen yet?
         JRST   FLINRT                  ;Yes. Return this as an operator.
        CAIN    A,"-"                   ;Is it a minus?
         TRO    FL,NEG1                 ;Yes. set negative flag.
        TRON    FL,SIGNED               ;A sign has been seen.  Skip if losing.
         JRST   FLINP1                  ;Get more stuff.
        ERROR   [ASCIZ/Two signs in the number/]

;Here when a decimal point is seen.
FLINPT: TRON    FL,DPOINT               ;Flag DP has been seen. Skip if losing
         JRST   FLINP1                  ;go eat more
        ERROR   [ASCIZ/Two decimal points in the number/]
```

```
;Here to process the exponent.
FLINEX: TRNN    FL,DIGF                 ;have digits been seen
         ERROR  [ASCIZ/Exponent seen, but no number/]
        CALL    FLINRT                  ;apply DP divisor and sign, so far.
        ILDB    A,W                     ;Get next character
        CAIN    A,"+"                   ;look for signs
         JRST   FLEX1                   ;got one.  ignore plus sign
        CAIE    A,"-"                   ;
         JRST   FLEX2                   ;character had better be a digit
        TRO     FL,NEG2                 ;set negative exponent, get another chr
FLEX1:  ILDB    A,W                     ;This had better be a digit
FLEX2:  CAIL    A,"0"                   ;verify we have a digit
         CAILE  A,"9"
          ERROR [ASCIZ/No digits follow "E" in number/]
        MOVEI   D,-"0"(A)               ;Accumulate exponent in D
FLEX3:  ILDB    A,W                     ;this is a standard decimal scan
        CAIL    A,"0"
         CAILE  A,"9"
          JRST  FLEX4                   ;exit, exponent in D
        IMULI   D,12                    ;decimal 10
        ADDI    D,-"0"(A)               ;add current digit.
        JRST    FLEX3

;Here we apply the exponent to adjust the result.
FLEX4:  JUMPE   D,CPOPJ                 ;exit if the exponent is zero.
        TRZE    FL,NEG2                 ;Is exponent negative?
         JRST   FLEX6                   ;yes.  apply appropriate divides.
FLEX5:  FMPRI   B,(10.0)                ;apply a positive exponent
        SOJG    D,FLEX5                 ;multiply by 10 and decrease exp.
        RET

FLEX6:  FDVRI   B,(10.0)                ;apply negative exponent.
        SOJG    D,FLEX6
        RET


        SUBTTL  FLOUTP - Floating-Output

;FLOUTP, Internal floating-point format to character conversion routine.
;Call with a floating-point number in A, a byte pointer in W.

FLOUTP: JUMPGE  A,FLOUT1                ;If A is positive, don't do
        MOVEI   B,"-"                   ;a minus sign.
        IDPB    B,W
        MOVN    A,A                     ;make A positive.
FLOUT1: CAMGE   A,[0.1]
         JUMPN  A,FLOUTS                ;Output a small, but non-zero, number
        CAML    A,[10000.0]
         JRST   FLOUTL                  ;Output a large number.
```

```
;Here to output a "normal" sized number including zero
FLOUTN: FIX     B,A                     ;Get the integer part into B
        FLTR    C,B                     ;Float the integer part
        FSBR    A,C                     ;Compute the fraction only into A.
        PUSH    P,A                     ;Save the fraction.
        MOVE    A,B
        CALL    DECOUT                  ;Print the integer part
        MOVEI   A,"."                   ;print a decimal point
        IDPB    A,W
        POP     P,A                     ;retrieve the fraction
        FMPR    A,[10000.0]             ;Multiply to make some integer part
        FIX     A,A                     ;Take integer part (0 to 9999)
        MOVEI   C,4                     ;output 4 digits
        MOVEI   D,"0"                   ;with zero as the fill character
        JRST    DECFIL                  ;Jump to DECFIL.  RET from DECFIL

;Here to output a small number (less than 0.1)
FLOUTS: MOVNI   D,1                     ;initial exponent. -1
FOUTS1: FMPRI   A,(10.0)                ;multiply by ten
        CAMGE   A,[1.0]                 ;is the result large enough yet?
         SOJA   D,FOUTS1                ;no.  decrement exp and loop.
        JRST    FOUTL2                  ;go print what was done.

;Here to output a large number.
FLOUTL: MOVEI   D,1                     ;initial exponent
FOUTL1: FDVRI   A,(10.0)                ;make number smaller
        CAML    A,[10.0]                ;is it small enough?
         AOJA   D,FOUTL1                ;No, increment exp and loop until good
FOUTL2: PUSH    P,D                     ;Save the exponent.
        CALL    FLOUTN                  ;Print an acceptably sized number.
        MOVEI   A,"E"                   ;print an E
        IDPB    A,W
        POP     P,A                     ;retrieve the exponent.
;Fall into DECOUT

DECOUT: MOVEI   C,0                     ;Decimal output, no fill.
DECFIL: JUMPGE  A,DECFL1                ;jump if positive.  same old trick
        MOVEI   B,"-"
        IDPB    B,W                     ;write minus sign
        MOVN    A,A
DECFL1: IDIVI   A,12                    ;decimal 10
        PUSH    P,B                     ;save remainder on stack
        SUBI    C,1                     ;count a digit will be printed
        JUMPE   A,DECFL3                ;exit if done with dividing
        CALL    DECFL1                  ;make more remainders.
DECFL2: POP     P,B                     ;get a digit from the stack.
        ADDI    B,"0"
        IDPB    B,W                     ;stuff it.
        RET

DECFL3: JUMPLE  C,DECFL2                ;Jump if no (more) fill is needed.
        IDPB    D,W                     ;store fill character
        SOJA    C,DECFL3                ;Count fill char has been sent.
```

```
        SUBTTL  MAIN PROGRAM

START:  RESET                           ;Initialize IO
        MOVE    P,[IOWD PDLEN,PDLIST]   ;establish a stack
        MOVE    A,[CALL UUOHAN]
        MOVEM   A,.JB41                 ;Store call to UUO trap handler
        TTYSTR  [ASCIZ/
Floating-point I-O Example
/]

NEXT:   TTYSTR  [ASCIZ/Type a number, any number: /]
        HRROI   A,BUFFER                ;read input to buffer area
        MOVEI   B,BUFLEN*5-1            ;buffer count
        MOVE    C,PROMPT                ;pickup reprompt
        RDTTY                           ;wait for user input
         ERJMP  [ERROR [ASCIZ/Error in RDTTY/]]

        MOVE    W,[POINT 7,BUFFER]      ;a byte pointer to buffer
        CALL    FLINP                   ;Read a number
        TRNN    FL,DIGF                 ;were any digits seen?
         JRST   DONE                    ;no.  exit
        MOVE    W,[POINT 7,OUTBUF]      ;Pointer to the output buffer
        MOVE    A,B                     ;Setup A with the number to write
        CALL    FLOUTP                  ;convert internal to external
        MOVEI   A,15                    ;add CR LF NULL
        IDPB    A,W
        MOVEI   A,12
        IDPB    A,W
        MOVEI   A,0
        IDPB    A,W
        TTYSTR  [ASCIZ/You typed: /]    ;and print results.
        TTYSTR  OUTBUF
        JRST    NEXT

DONE:   TTYSTR  [ASCIZ/Done!
/]
        HALTF
        JRST    START
```

```
        SUBTTL  Handle Local UUOS - Section Zero

UUOHAN: PUSH    P,UUACAD                ;save AC base addr in case of recursion
        PUSH    P,0                     ;Save AC number Zero
        HRRZM   P,UUACAD                ;address of AC zero on stack
        ADJSP   P,16                    ;make room for 1 to 16
        MOVEI   0,-15(P)                ;address at which to store 1
        HRLI    0,1                     ;1,,-15(P)  (source,,destination)
        BLT     0,0(P)                  ;save ACs on stack
        MOVE    0,-16(P)                ;in case you need caller's 0 in 0
        CALL    DOUUO                   ;perform the UUO function
         TRNA                           ;non-skip from UUO
          AOS   -20(P)                  ;pass skip to caller
        MOVSI   16,-16(P)               ;-16(P),,0  (source,,destination)
        BLT     16,16                   ;restore 0 thru 16
        ADJSP   P,-17                   ;remove storage for 0 to 16
        POP     P,UUACAD                ;restore AC base addr
        RET                             ;return to caller

DOUUO:  LDB     A,[POINT 9,.JBUUO,8]    ;the OP code
        LDB     B,[POINT 4,.JBUUO,12]   ;the AC field
        HRRZ    C,.JBUUO                ;the effective address
        CAIL    A,UUOTLN                ;is the uuo number in range?
         MOVEI   A,0                    ;No.  0 is a loser too.
        MOVE    D,UUOTAB(A)             ;Get the dispatch address
        JRST    (D)                     ;Return via POPJ or CPOPJ1

UUOTAB: ILLUUO
        ERRUUO
        OTSUUO
        OTCUUO
UUOTLN==.-UUOTAB

ILLUUO: ERROR   [ASCIZ/Illegal Local UUO detected
/]

ERRUUO: HRROI   A,[ASCIZ/Error: /]      ;The ERROR UUO
        ESOUT                           ;Clear typeahead.  Write ?
        HRRO    A,C
        PSOUT
        HRROI   A,[ASCIZ/
/]
        PSOUT
        HALTF
        JRST    START

;Process TTYSTR UUO.
OTSUUO: HRRO    A,C                     ;output a string
        MOVEM   A,PROMPT                ;save last typeout as the prompt.
        PSOUT
        RET
```

```
; Process TTYCHR LUUO.  Note that by convention, an AC is an acceptable
; place for a character argument (but not usually for a string), so
; this takes special care to test for an argument in the ACs.
OTCUUO: CAIGE   C,17                    ;look for a reference to an AC
         ADD    C,UUACAD                ;reference to an AC.  Look on stack
        MOVE    A,(C)
        PBOUT
        RET

        END     START
```

This program has three areas of special interest, the floating–point input scan, the floating–point output conversion, and the LUUO handler. In addition to these main points of interest, we have introduced another pseudo–op, `SUBTTL` and another JSYS, `ESOUT`. These will be explained in the discussion that follows.

18.1.1 SUBTTL Pseudo–Operator

The `SUBTTL` pseudo–op that we have introduced in this example adds a subtitle to the assembly listing file. The text that follows the word `SUBTTL` will be printed as a subheading on each page of the listing file following the occurrence of the `SUBTTL` pseudo–op. It is usual to place `SUBTTL` statements above large routines or groups of routines that serve a common purpose. A long program consists of many different components; often a set of related components are gathered under one descriptive subtitle.

Another organizational technique is the use of page marks in the source file. A page mark (or form–feed character) should be used to separate functionally separate components of the program. Often the first thing on a new page is a `SUBTTL` command; all routines on the page may be related. The various text editors have commands for adding a form–feed character to a file; the assembler will start a new page of output listing for each form–feed that is seen. As a matter of style, programmers should probably avoid writing code that falls off the end of one page into the beginning of another. If it can't be avoided, document it: "falls through to next page", "here from previous page", etc.

18.1.2 FLINP — Floating–Point Input Scan

The `FLINP` routine is an extension of the `DECIN` routine that we examined in example 7. The essential difference is that `FLINP` does all of its computations using floating–point numbers. The essence of `DECIN`, where the old number is multiplied by 10 and the new digit added to the result, appears in `FLINP`. Besides the change to floating–point instructions, `FLINP` has several additional problems to worry about. When scanning a real number, decimal points and exponents have to be dealt with.

18.1.2.1 Processing the Decimal Point

The decimal point is dealt with in the following manner. Accumulator `C` is initialized with the constant `1.0`. When a decimal point is seen, the code at `FLINPT` sets the flag `DPOINT`. After `DPOINT` is set, each time a digit is seen, `C` is multiplied by `10.0`; other than multiplying `C` by `10.0`, digits after the decimal point are processed in the same way as the ones that come before the point. After all, the difference between scanning `123.0` and scanning `1.23` is simply a matter of dividing by the right

power of 10. The multiplication of C by 10.0 for each digit after the decimal point will compute the proper divisor in C.

The code at FLINRT effects the division of the number accumulated in B by the proper divisor that was accumulated in C. Also, at FLINRT, the proper sign is applied to the result.

(Another approach would be to initialize a counter to zero; increment it for each digit past the decimal point. After the scan of digits is complete, this count can be used to index a table of powers of ten to locate an appropriate divisor. The table of powers of ten may be more accurate—and less work in total—than the repetitive multiplication by ten.)

18.1.2.2 Processing the Exponent

The FLINP routine will jump to FLINEX when the character E, signifying an exponent, is seen. FLINEX will call the FLINRT routine to apply the sign and decimal point divisor to the number seen thus far. Then, with code that is very similar to DECIN, it will accumulate the exponent in D. When the end of the exponent string is found, the exponent is applied to the number in B, by multiplying (or dividing) by 10.0 an appropriate number of times.

(A table of powers of ten would be more efficient and perhaps more accurate than this loop. The exponent and the count of digits following the decimal point could be combined to make one lookup and one multiplication or division.)

18.1.3 FLOUTP — Floating–Point Output Processing

The FLOUTP routine converts a floating–point number in A into a character string. When FLOUTP is first entered, a minus sign is sent to the output string if A is negative; register A is then made positive. A three–way branch is made: if the number is 10000.0 or larger, the program jumps to FLOUTL (for large numbers); if the number is smaller than 0.1, but not zero, the program jumps to FLOUTS (for small numbers); numbers in the range from 0.1 $\leq$ C(A) < 10000.0 and zero are handled at FLOUTN.

18.1.3.1 FLOUTN

The FLOUTN routine is called with register A containing a number that is in a range such that it is reasonable to output characters before the decimal point and four digits after the decimal point.

The integer portion of A is obtained in B by the instruction FIX B,A. The floating equivalent of the integer portion of A is then created in register C by FLTR C,B. The fraction portion of the number in A is then obtained by subtracting C from A. The fraction is saved on the stack.

The integer portion of the original number is copied to A and output via DECOUT; DECOUT acts like a standard decimal print routine. A decimal point is picked up in A and output. Then the fraction part of the original number is retrieved by popping the stack. The fraction is multiplied by 10000.0. The integer portion of this product, a number from 0000 to 9999, represents the four digits following the decimal point. The integer portion is obtained by applying the FIX instruction to the product in A. This fixed–point quantity is now printed with leading zero fill in four columns by the DECFIL, *DECimal output with FILl*, routine (see below). Thus, the number 99 would be printed as 0099, to cause the proper number of zeros to appear after the decimal point.

One trick has been played here that was mentioned in the discussion of the PUSHJ instruction. At the end of the FLOUTN routine, the instruction JRST DECFIL appears. This instruction has the same

effect as the sequence

```
CALL    DECFIL
RET
```

The JRST instruction is faster. Instead of having DECFIL return to FLOUTN which would return to its caller, the JRST instruction causes the final RET in DECFIL to return instead to the caller of FLOUTP or FLOUTN.[1]

18.1.3.2 FLOUTS and FLOUTL

The FLOUTS and FLOUTL routines are used to normalize a number that would otherwise be out of the range that can be printed by FLOUTN. The FLOUTL routine is called with a large number in A. FLOUTL initializes register D to 1 and divides A by 10.0. If the result is small enough for FLOUTN to handle, the CAML at FOUTL1+1 will skip to FOUTL2. If the number in A isn't yet small enough, register D is incremented and another divide occurs.

The loop at FOUTL1 will divide A by 10.0 until A becomes smaller than 10.0; D counts the number of divisions that were needed. At FOUTL2, the number in A is now within an acceptable range. The exponent, D, is pushed on the stack. The number in A is printed via FLOUTN. Then, the letter E is printed; the exponent is popped from the stack and then printed by falling into the DECOUT routine.

Falling into DECOUT takes the trick mentioned above one step further. Instead of replacing the CALL and RET sequence with a JRST, we avoid even the JRST by the simple expedient of placing the code for DECOUT immediately after the FOUTL2 code.

The loop at FLOUTS is essentially the same as the one at FLOUTL. FLOUTS makes the number in A larger, while decrementing the exponent in D.

(Again, a different approach is possible. Given that you have a table of powers of ten and their reciprocals, i.e., negative powers of ten, a binary search of the table of powers of ten to find an appropriate divisor—or multiplier—is expected to be faster and more accurate than the repetitive multiplications or divisions.)

18.1.3.3 DECFIL — Decimal Output with Leading Fill

The code at DECFIL is a modification to the recursive decimal printer that we examined in example 7. The modification consists mainly of having another accumulator, in this case C, that initially contains a count of how many columns to print. Each time a remainder is stored on the stack, C is decremented. When it is time to start unloading remainders from the stack, the code at DECFL3 is called to add as many fill characters before the digits as are necessary. At DECFL3, C contains precisely the number of fill characters to add. The instruction JUMPLE C,DECFL2 will jump when no more fill characters are needed. If we get to DECFL3 with C being zero or negative then no filling at all will occur. When filling is needed, the character in D is deposited in the output buffer by means of the byte pointer in W. Then the fill count in C is decremented and the program loops back to DECFL3 until adequate filling has been accomplished. After enough fill characters have been sent, the program jumps to DECFL2, rejoining the normal recursive decimal printer.

[1] Sometimes you may see the mnemonic CALLRET or PJRST used to signify a JRST to a routine, the return from which will effect the return from the routine in which the CALLRET (or PJRST) appears. In these cases CALLRET is actually the same machine operation as JRST.

18.1.4 Local UUO Processing in Section Zero

This program includes an example showing how Local UUOs can be used in a program. There are several important concepts that we must explain here. Among these are external symbols, the definition of LUUOs, and the function of the LUUO handler itself.

18.1.4.1 External Symbols

Symbols that are defined in one program and referred to from another are called *global symbols*. Often, global symbols are names of subroutines that are defined in separate `REL` files or libraries. The pseudo–ops `INTERN` and `ENTRY` will make a local symbol (i.e., a symbol that is defined in the current program) available to other programs as a global symbol. The `EXTERN` pseudo–op informs the assembler that particular symbols referred to by the current program are defined elsewhere. This example program refers to the symbols `.JBUUO` and `.JB41` which are defined elsewhere. Thus, these two symbols appear as arguments to the `EXTERN` pseudo–op.

The assembler cannot output a value for an external symbol. Instead, it outputs a request for help from the loader. As the loader reads the file where a global symbol is defined, it becomes aware of the value of that symbol. Following the directions received from the assembler, the loader will place the correct value for each global reference into the program that is loaded.

The symbols `.JBUUO` and `.JB41` are defined in the file `SYS:JOBDAT.REL`. This file is normally loaded with every program; it contains definitions for the TOPS–10 job data area (`JOBDAT`). Even though in TOPS–20 few `JOBDAT` symbols are meaningful, all the `JOBDAT` symbols definitions applicable to TOPS–10 are loaded. As we had occasion to mention earlier, the `JOBDAT` area extends from location `20` through `137`; LINK accommodates the job data area by loading most programs starting at location `140`.

The particular symbols `.JBUUO` and `.JB41` are defined as locations `40` and `41` respectively. The location `.JBUUO` is where the image of the LUUO instruction is stored. The location `.JB41` contains the instruction to execute when an LUUO occurs.

18.1.4.2 Definitions of Local UUOs

Apart from our usual definitions of `CALL` and `RET`, three additional `OPDEF` pseudo–ops are included in the program. The notation `1B8` represents another form of number in the assembler. The `B` notation means that the expression in front of the letter `B` should be shifted so that the least significant bit of the expression occupies the specified bit in a word. Thus the expression `1B8` means the value `1` adjusted so that the least significant bit occupies bit `8` in a word. This is equivalent to `001000,,0`. It should be noted that the number that follows the `B` is interpreted as a *decimal* number.

The three `OPDEF`s define new operation codes for the "instructions" `ERROR`, `TTYSTR` and `TTYCHR`. When these instructions are written in a program, the corresponding binary patterns are assembled. Unlike the definitions of `CALL` and `RET` that we examined earlier, these binary patterns do not correspond to hardware instructions. Instead, these patterns are executed as LUUOs.

18.1.4.3 Initialization of the LUUO Handler

In order to execute an LUUO successfully, it is necessary to store an appropriate instruction in `.JB41`. In this program, we use a `PUSHJ P,UUOHAN` to call the LUUO handler. It should be obvious

that P, the stack pointer, as well as .JB41 must be set up properly before attempting to execute an LUUO.

18.1.4.4 The LUUO Handler

The code at UUOHAN and DOUUO comprise the LUUO handler. UUOHAN is concerned with establishing an environment in which the LUUO can be processed. The routine DOUUO sets up particular accumulators with arguments and dispatches to the appropriate UUO service routine.

UUOHAN is called via the instruction PUSHJ P,UUOHAN executed in location 41 as a result of performing any LUUO. The return program counter that is stored on the stack reflects the address of the LUUO itself (and not, for example, 41 or 42).

The code at UUOHAN allows recursive LUUOs. This means that any temporary storage that is needed for an LUUO must be allocated on the stack. This is usually accomplished by pushing local storage onto the stack.

The first item of local storage that is pushed is the contents of UUACAD. Inside the LUUO handler, UUACAD contains the address where we stored the previous context's accumulators. If an LUUO is called from the normal program context, UUACAD contains nonsense; however, when an LUUO is called recursively, UUACAD must be saved and restored.

The instruction PUSH P,0 saves accumulator 0 on the stack. The HRRZM P,UUACAD that follows, stores the address where 0 was stored. Next, an ADJSP instruction is used to allocate stack space. Room for accumulators 1 through 16 is obtained. Then the accumulators 1 through 16 are copied onto the stack via a BLT instruction. Following the BLT, accumulator 0, which had been used as the BLT accumulator, is restored from the stacked copy of the previous context's 0.

DOUUO performs the UUO function. If DOUUO skips, the skip return is passed to the caller of UUOHAN by means of the instruction AOS -20(P). The remainder of the code in UUOHAN restores the previous context's accumulators, restores the original value of UUACAD, unwinds the stack, and returns via a POPJ P, instruction.

The code at DOUUO loads register A with the LUUO opcode. Register B is set to the accumulator field of the LUUO image; C is set to the effective address from the LUUO. The opcode in A is now tested to see if it is within the range of defined local UUOs. If A is out of range, it is set to zero. Then the dispatch address from UUOTAB is copied to D, and the program jumps to the specific routine to service the particular LUUO.

As a sample routine, consider OTSUUO. This routine performs a HRRO instruction to get −1 into the left of A and the address of the string into the right. This result is stored in PROMPT, where it could be used as the reprompt for RDTTY. Then OTSUUO performs the PSOUT instruction and returns.

It might be argued that to rumble through all the code in UUOHAN and DOUUO for the purpose of performing only a PSOUT is wasteful. There is some truth to such an argument. However, consider the following arguments in favor of using a local UUO.

First, the calling sequence is more compact than the code would have been had an LUUO not been used. Sometimes this compactness is a great convenience. For example, the several places where ERROR is used, it appears as a one–line (and one–word) instruction. This is important because several places use skips over the ERROR UUOs. Second, the relatively expensive entry sequence for an LUUO may be forgiven when the function that is performed is either infrequent or intrinsically expensive. PSOUT may be expensive; in any event, it is relatively infrequent (in relation to computer speeds), since PSOUT implies that a human is reading the output.

An example of the use of UUACAD occurs in OTCUUO. The TTYCHR LUUO permits the argument to the LUUO to be in the accumulators.[2] To account for the possibility that the argument is in one of the accumulators that has been changed by DOUUO, the code at OTCUUO examines the effective address of the LUUO. If the address, in C, falls within the accumulators, then the contents of UUACAD are added to the accumulator address. Since UUACAD contains the address where accumulator 0 is stored, the sum of UUACAD and an accumulator number corresponds to the address where the appropriate accumulator was saved.

The word at the effective address is fetched into A; a PBOUT is executed to write one character.

18.1.4.5 ESOUT JSYS

The ESOUT, *Error String OUTput*, JSYS is used to send error typeout to the terminal. ESOUT discards any terminal input that may have been typed ahead. Then ESOUT types a question mark preceding the text of the message. A special JSYS exists for this purpose primarily to make the format and behavior of error messages more uniform.

18.2 Example 8–B: LUUO Handler for a Non–Zero Section

Example 8–B shows how to change Example 8–A's LUUO processing to make it work in a non–zero section. Instead of presenting a complete program, only the changes are shown.

The file of differences is the result of using the SRCCOM program to compare two source files. SRCCOM is presented here as a useful tool for tracking the differences that separate two text sources. It is especially helpful in looking at programs.

The command by which this comparison file was made is

```
@srccom ex8.dif=ex8a.mac,ex8b.mac
```

The resulting file, EX8.DIF, is presented below, with the explanatory text interspersed.

```
;Comparison of TOED:<GORIN.DOC.T20ASM>EX8A.MAC.1 and
;              TOED:<GORIN.DOC.T20ASM>EX8B.MAC.4
;Options are    /3;   at 24-Nov-2004 09:16:45
```

To begin with, SRCCOM announces itself by saying what files it is comparing, what options were requested, and when the file was made. The option reported as /3 (a default value) signifies that SRCCOM must find three identical lines in both files that follow an area of differences in order for it to think that one group of differences has ended.

[2] By convention, strings aren't passed in the accumulators, so we avoid this code at OTSUUO.

```
**** FILE TOED:<GORIN.DOC.T20ASM>EX8.MAC.1, 1-1 (0)
        TITLE   FLIO    Floating-point Input and Output (Example 8-A)
        SEARCH  MONSYM
        EXTERN  .JB41,.JBUUO
**** FILE TOED:<GORIN.DOC.T20ASM>EX8B.MAC.4, 1-1 (0)
        TITLE   FLIO    Floating-point Input and Output (Example 8-B)
        SEARCH  MONSYM
***************
```

Each group of differences is headed with the name of the file from which the report originates, the page number and line number in the file and, in parentheses, the character number of the first character of the line where the difference is noted. As these files differ on line 1 of page 1, the first character position is zero.

A group of differences ends with a short row of asterisks.

Although the lines `SEARCH MONSYM` are identical, the line that follows differs: the non–zero section program has omitted the `EXTERN` line. So, that difference is reported with line 1's difference as one group of differences.

```
**** FILE TOED:<GORIN.DOC.T20ASM>EX8.MAC.1, 1-28 (677)
**** FILE TOED:<GORIN.DOC.T20ASM>EX8B.MAC.4, 1-28 (662)
        .PSECT  DATA,1001000
***************
```

`.PSECT` was added to locate the data area.

(A criticismy of SRCCOM is that some of its reports lack sufficient context information to locate the point where an insertion is called for. The indicated character position can be helpful. SRCCOM can also be told to report additional context information.)

```
**** FILE TOED:<GORIN.DOC.T20ASM>EX8.MAC.1, 1-35 (785)
**** FILE TOED:<GORIN.DOC.T20ASM>EX8B.MAC.4, 1-36 (800)
LUUBLK: BLOCK   4                       ;Block for LUUO information
        .PSECT  CODE/RONLY,1002000
***************
```

The definition of `LUUBLK` was added to the data area. A second ṖSECT line was added to locate the code and constants.

SRCCOM omits blank lines from the listing; although this output shows these two lines as consecutive, the actual source file separates them with a blank line.

```
**** FILE TOED:<GORIN.DOC.T20ASM>EX8.MAC.1, 1-199 (9034)
        MOVE    A,[CALL UUOHAN]
        MOVEM   A,.JB41                 ;Store call to UUO trap handler
**** FILE TOED:<GORIN.DOC.T20ASM>EX8B.MAC.4, 1-204 (9164)
        XMOVEI  A,UUOHAN
        MOVEM   A,LUUBLK+3              ;set address of LUUO handler in block
        MOVEI   A,.FHSLF                ;This fork
        MOVEI   B,.SWLUT                ;set LUUO block address
        XMOVEI  C,LUUBLK                ;address of LUUO Block
        SWTRP%
         ERJMP  [HRROI A,[ASCIZ/Error from SWTRP%
/]
                ESOUT
                JRST STOP]
***************
```

The initialization of the LUUO handler is very different. Instead of storing an instruction in location 41, the non–zero section code stores the extended address of the LUUO handler in the LUUO block and uses the SWTRP% JSYS to introduce LUUBLK as the LUUO trap block.

For this purpose, SWTRP% requires three arguments. The argument in accumulator 1 is .FHSLF, *Fork Handle to Self*. When the program wishes to talk about itself to TOPS–20, .FHSLF is often used as an identifier. It serves the same purpose as the word "me" as the object of the verb in a sentence. The argument in accumulator 2 specifies which function of SWTRP% we want; .SWLUT is the code by which we request that TOPS–20 set the LUUO block address for the specified process. Finally, accumulator 3 specifies the 30–bit address of the LUUO block.

If SWTRP% should fail for any reson the ERJMP that follows jumps to a routine that does *not* use the ERROR LUUO defined in the program. The reason for avoiding an LUUO should be obvious.

```
**** FILE TOED:<GORIN.DOC.T20ASM>EX8.MAC.1, 1-229 (10306)
        HALTF
        JRST    START
        SUBTTL  Handle Local UUOS - Section Zero
UUOHAN: PUSH    P,UUACAD                ;save AC base addr in case of recursion
        PUSH    P,0                     ;Save AC number Zero
        HRRZM   P,UUACAD                ;address of AC zero on stack
**** FILE TOED:<GORIN.DOC.T20ASM>EX8B.MAC.4, 1-243 (10600)
STOP:   HALTF
        JRST    START
        SUBTTL  Handle Local UUOS - Non-Zero Section
UUOHAN: PUSH    P,LUUBLK                ;preserve LUUO flags (and image)
        PUSH    P,LUUBLK+1              ;preserve LUUO 30-bit PC
        PUSH    P,UUACAD                ;save AC base addr in case of recursion
        PUSH    P,0                     ;Save AC number Zero
        MOVEM   P,UUACAD                ;address of AC zero on stack
***************
```

A label, STOP: has been added to the HALTF JSYS. This label is referred to by the code that handles any error from SWTRP%.

We now turn to the changes at UUOHAN. In the new program, the CPU effect a jump to UUOHAN (instead of arriving via a PUSHJ). To preserve the return address (and the flags) in the event of a recursive LUUO, the program pushes both the flags and PC that were stored in LUUBLK.

(If restoring the flags is not important to the program, you may omit pushing them and push only the return address. In that case, on return to the program that executed the LUUO, you may use RET instead of the two POPs and XJRSTF that are found below.)

UUOHAN stores the entire word P in UUACAD. Regardless of whether P is a local stack pointer or global, this is a good choice. The value in UUACAD is used to access the caller's ACs, stored on the stack. If it is a local stack pointer, the negative value in the left half will not affect the calculation; if it is a global stack pointer, the left half is needed. (Note: this program assumes that the stack is in the program's address section. The discussion of BLT showed the general case of storing and retrieving ACs using a stack that isn't local to the program's section.)

```
**** FILE TOED:<GORIN.DOC.T20ASM>EX8.MAC.1, 1-249 (11397)
        RET                             ;return to caller
DOUUO:  LDB     A,[POINT 9,.JBUUO,8]    ;the OP code
        LDB     B,[POINT 4,.JBUUO,12]   ;the AC field
        HRRZ    C,.JBUUO                ;the effective address
        CAIL    A,UUOTLN                ;is the uuo number in range?
         MOVEI   A,0                    ;No.  0 is a loser too.
        MOVE    D,UUOTAB(A)             ;Get the dispatch address
        JRST    (D)                     ;Return via POPJ or CPOPJ1
UUOTAB: ILLUUO
        ERRUUO
        OTSUUO
        OTCUUO
**** FILE TOED:<GORIN.DOC.T20ASM>EX8B.MAC.4, 1-265 (11816)
        POP     P,LUUBLK+1              ;restore LUUO PC (possibly incremented)
        POP     P,LUUBLK                ;restore LUUO Flags
        XJRSTF  LUUBLK                  ;restore caller's flags and PC
DOUUO:  LDB     A,[POINT 9,LUUBLK,26]   ;the LUUO OP code
        LDB     B,[POINT 4,LUUBLK,30]   ;the LUUO AC field
        MOVE    C,LUUBLK+2              ;the 30--bit effective address
        CAIL    A,UUOTLN                ;is the uuo number in range?
         MOVEI   A,0                    ;No.  0 is a loser too.
        JRST    @UUOTAB(A)              ;dispatch.  Return via POPJ or CPOPJ1
UUOTAB: 1B0!ILLUUO                      ;dispatch addresses are IFIWs
        1B0!ERRUUO
        1B0!OTSUUO
        1B0!OTCUUO
***************
```

At the top of this group of differences note that RET has been replaced by two POPs and XJRSTF. This combination restores the program flags so that they are unaltered by the LUUO.

At DOUUO, the opcode and AC fields are extracted from the right side of the first word of the LUUO block instead of from the left side of location 40.

The table of dispatch addresses, UUOTAB has been changed by setting bit 0 to 1 in each entry. This gives each word the form of a local format indirect word. The dispatch has been changed slightly to use an indirect JRST instead of one that is indexed.

```
**** FILE TOED:<GORIN.DOC.T20ASM>EX8.MAC.1, 1-284 (12475)
OTCUUO: CAIGE   C,17                    ;look for a reference to an AC
         ADD    C,UUACAD                ;reference to an AC.  Look on stack
        MOVE    A,(C)
**** FILE TOED:<GORIN.DOC.T20ASM>EX8B.MAC.4, 1-302 (13080)
OTCUUO: TDNE    C,[7776,,777760]        ;look for a reference to an AC
         JRST   OTCUU1                  ;Not an AC reference
        TLZ     C,1                     ;clear section 1 from AC Address
        ADD     C,UUACAD                ;reference to an AC.  Look on stack
OTCUU1: MOVE    A,(C)
***************
```

Finally, the code at OTCUUO uses a different test to see whether or not the LUUO's effective address is an accumulator. If `E` is an accumulator, the CPU will store the accumulator's global address, that is, `1000000` plus the accumulator number. The `TDNE` tests for no section bits other than section 1 being set and no in–section address bits higher than `17` being set. If the test determines that `E` is an AC, the program clears the section number and the AC number is added to the value in `UUACAD`.

These changes complete the conversion of the traditional style LUUO handler to one that works in the extended machine.

18.3 Exercises

18.3.1 Report Overflow, Underflow

Demonstrate that either example program ignores floating point exceptions: overflow and underflow.

Augment the floating point input routine to report floating overflow and floating underflow.

18.3.2 A Bad Idea

Instead of the sequence given above for restoring the flags and PC, the following is proposed. It is a bad idea. Can you say why?

```
        Good                    Bad

     POP P,LUUBLK+1         ADJSP P,-2      ;correct the stack pointer
     POP P,LUUBLK           XJRSTF 1(P)     ;restore flags and PC
     XJRSTF LUUBLK
```

(This question is only a little bit unfair: it makes more sense at the end of Chapter 29.)

18.3.3 Simulate the ADJBP Instruction

Write an LUUO handler that simulates the effect of the `ADJBP` instruction. The explanation of `ADJBP` given here is probably not sufficient for you to write a completely accurate simulation. Although you may not be able to find any complete description of `ADJBP`, do not lose hope: since the computer

that you are using has an ADJBP instruction, you can test the implementation whenever questions arise about how the instruction works.

Pay particular attention to the following points:

- Test your simulation on reasonable cases and on boundary cases.
- Do you get the right result when S is zero?
- Does your program work when S or P is greater than 36_{10}?
- How about when zero is the adjustment count?
- Does your program work no matter which accumulator is used to hold the adjustment count in the LUUO?
- Does it simulate IBP when AC 0 is selected?
- Does it return a divide check condition in the same circumstances as the real instruction does?

This does not have to be a long program, but does require some care and patience to make it work right in all cases.

18.3.4 Create the Inverse of ADJBP, SUBBP

Create an LUUO that undoes the effect of ADJBP. Specifically, compute the difference between two byte pointers: given a byte pointer in C(AC) and a second byte pointer in C(E) compute the adjustment count that would have been needed in C(AC) to make the ADJBP instruction return in the AC the byte pointer that you have been given there. Return this count in the accumulator. For example:

```
MOVE    A,[POINT 7,1000,13]
SUBBP   A,[POINT 7,1000]
. . .                          ;result in A should be 2
```

Note: after performing your SUBBP, Subtract Byte Pointer, operation, you should be able to reconstruct the original contents of the AC (that is, the contents prior to the SUBBP operation) by doing an ADJBP operation.

Some argument pairs are incompatible: if the byte size and byte alignment are not identical then the operation is meaningless. Halting the program with a message would be appropriate in such a case.

Does your code work for all formats of byte pointer?

Chapter 19

Operating System Facilities

At this point our discussion of the hardware instruction set is essentially complete (although we have yet to discuss extended addressing, and we shall not discuss the string instructions). There are just a few instructions that have not been discussed; some of these we will speak of later. From this point on, we shall concentrate our attention on putting these instructions together to form useful programs.

Although our discussion of the hardware instruction set is nearly complete, we have barely scratched the surface in our exploration of the operating system. This section is an overview of the operating system facilities that we will discuss in detail in the sections that follow.

Complete information about the operating system facilities is found in [MCRM] and [MCUG].

19.1 Files and JFNs

The *file* is an extremely useful data structure provided by the operating system. Files on the disk represent a particularly important facility. While we are logged in to the system, the computer remembers many things about us. But when we leave the computer, the computer forgets nearly all that it knows about what we have been doing. The important exception, the information that is remembered from session to session, is our disk files.

Mostly when we talk of files in general we shall mean disk files. However, TOPS–20 tries to hide the differences between devices. Thus, without too much effort, a program that we write to manipulate disk files may be made to manipulate tape files, etc.

A file is a collection of information that has a name (and other attributes). Ultimately the name of the file identifies how to find the file. In TOPS–20 a file name such as `PS:<SYSTEM>MONITR.EXE.4` specifies a path by which the system can find the file. In order to obtain access to a file, you must know its name.

A file name is a bulky thing. Each component may have up to thirty–nine characters. Although the string of characters in the file name does identify the file, it would be helpful to have some shorthand for talking about any particular file. A JFN, *Job File Number*, is such a shorthand notation.

A JFN is a small number that is assigned within one job to be the *handle* by which the file can be manipulated. A JFN can be obtained from the operating system by means of the `GTJFN` JSYS. In `GTJFN` the program specifies the name of the file; the system returns a JFN that subsequently can

be used to reference that file.

A variety of system calls accept and manipulate JFNs. Among these are

`GTJFN` GeT a Job File Number

`GTJFN` accepts control flags in register `1`. A string pointer or a pair of source and destination JFNs are in register `2`. When successful, `GTJFN` returns the new JFN in the right half of register `1`; optional JFN flags may be returned in the left half of register `1`.

The JFN flags will be zero unless flag bits in the call to `GTJFN` request that JFN flags be returned. The JFN flags are not used too frequently, except in conjunction with the `GNJFN` call. Most of the JSYS calls that are mentioned below require as an argument a JFN in the right half of register `1`. In many cases the left half of the JFN argument word should be zero or data other than the JFN flags.

`OPENF` OPEN File

Requires a JFN in register `1`. Register `2` must be set up to contain the file byte size and the desired access modes.

After a `GTJFN` call, a file must be opened before input or output can be done. `OPENF` allows the program to specify either input or output or both. (There are more exotic options that we will not have an opportunity to discuss.)

`SOUT` String OUTput

Requires a JFN in register `1`, a source descriptor in `2`, and a byte count in `3`. When a positive byte count is specified, `SOUT` also uses register `4` as the break character.

`BOUT` Byte OUTput

The `BOUT` JSYS sends one character, held in register `2`, to the file corresponding to the JFN in `1`.

`SIN` String INput

Requires a JFN in register `1` and a destination designator in `2`. The length of the input buffer area is set up in register `3`. When the byte count given in register `3` is positive, a break character is required as an argument in register `4`.

`BIN` Byte INput

`BIN` will return in register `2` the next character from the file corresponding to the JFN in register `1`.

`GTSTS` GeT STatuS

Requires a JFN in `1`; returns the JFN status word in `2`. The status word contains information reporting conditions such as end of file or data errors.

`CLOSF` CLOSe File

Requires a JFN in the right half of register `1` and flags for `CLOSF` in the left half. Closing a file usually releases the JFN also. Closing the file is necessary if output has been done that must be preserved.

It is possible to close an output JFN with flags set to abort the creation of a new file. It is necessary to close an open file before releasing the corresponding JFN; usually `CLOSF` will release the JFN; see `RLJFN` below.

`GNJFN` Get Next JFN

This JSYS allows the program to step, file by file, through the list of files corresponding to a file name that contains *wildcards*. A wildcard is a file name that possibly specifies several files. For example, the file name `*.MAC` contains a wildcard, the asterisk; this name matches all files of type `MAC`.

The JFN and flags that were obtained by `GTJFN` are required in register in `1`. The JFN must not correspond to an open file (you must `CLOSF` first, if a previous file was open). `GNJFN` skip returns, having stepped the JFN to the next file specified by the original wildcard description. When the file list implied by the original specification to `GTJFN` has been exhausted, no skip occurs, and `GNJFN` releases the exhausted JFN.

`JFNS` Convert JFN to String

A destination pointer is required in `1`; the JFN and optional JFN flags should be placed in register `2`; JFNS control flags are put in register `3`. The `JFNS` JSYS writes the name of the file corresponding to the JFN from register `2`, to the destination specified in register `1`. The flags in register `3` control the format of the resulting file name.

`RLJFN` ReLease JFN

Requires a JFN in the right half of register `1`. The JFN should not have a file open at the time it is released. Releasing a JFN breaks the connection between the JFN and the file.

`PMAP` Page Map control.

The `PMAP` JSYS provides for a program to control the contents of its memory map (page map). Among the operations provided are

- Map a file page to a memory page, thus effecting file input.
- Map a memory page to a file page, thus effecting file output.
- Map a memory page to another process' memory page, effecting page sharing between two or more processes.

There are numerous JSYS calls in addition to these that accept JFNs as arguments. As we have occasion, we shall explain each of these JSYS calls in greater detail. Consult [MCRM] for the complete explanation of each JSYS.

19.1.1 Predeclared JFNs

There are two predeclared JFNs that are of special interest. These JFNs are permanently assigned numbers corresponding to the names `.PRIOU` and `.PRIIN`. These are the primary output and primary input JFNs, respectively. Ordinarily, primary input and output are directed to the terminal. In any circumstance where an output JFN is acceptable, the symbol `.PRIOU` may be used. Similarly, `.PRIIN` may be used as an input JFN. For example, the following two program fragments have precisely the same effect:

```
MOVEI   1,.PRIOU        ;select primary output JFN
HRROI   2,MESAGE        ;a source designator
MOVEI   3,0             ;message ends with a zero byte
SOUT                    ;string output (to the terminal)
```

Compared to

```
HRROI 1,MESAGE          ;source designator
PSOUT                   ;implies SOUT to .PRIOU,
                        ; end at the first zero byte
```

19.2 Other Operating System Features

A variety of other facilities exist in the TOPS–20 operating system. We will briefly mention some of these now, and defer our detailed discussion of these to later sections.

19.2.1 Information about the Environment

TOPS–20 provides functions that obtain the date and time, status of the system and other information pertinent to the environment in which a program is executing.

19.2.2 Data Conversion Routines

Simple functions like numeric input and output routines (`NIN` and `NOUT`) are provided to simplify programming. The `NIN` and `NOUT` JSYS calls are similar in function to the subroutines `DECIN` and `DECOUT` that we have already examined. We have made a point to explain `DECIN` and `DECOUT`, because most systems do not provide such functions. TOPS–20 provides similar functions for single–precision floating–point input and output: `FLIN` and `FLOUT`, and for double–precision floating–point: `DFIN` and `DFOUT`.

TOPS–20 provides routines that perform date and time input and output, `IDTIM` and `ODTIM`, respectively. One of the important reasons for using these JSYS calls is that they provide a consistent format between different programs.

19.2.3 Process Creation and Control

Every program in TOPS–20 is run within its own process or *fork*. TOPS–20 provides system calls by which a program can create a new fork, load another program into that fork and start it. The originating process, called the *superior* process, can exert a variety of controls over the created (*inferior*) process. In the usual TOPS–20 environment, the EXEC program is superior to any other process run within a job.

Each process has its own address space that may or may not be shared with other processes in the same job. When a new process is created, the creating process (the superior process) may elect to share a portion of its address space with the newly created process.

Each process is scheduled independently of other processes.

Processes within one job share such things as job–wide logical names, the connected directory, and the pseudo interrupt (software interrupt, or PSI) system.

19.2.4 Command Scanning

We have already seen that RDTTY provides a consistent set of line editor functions. Although every program could do its own line editing, RDTTY provides a convenient and uniform interface for all programs.

In the same spirit as RDTTY, the COMND JSYS provide a uniform method of command scanning and a consistent user interface. The COMND JSYS provides such functions as command and file name recognition, CTRL/R repetition of the command prompt and typed line, and CTRL/W to erase fields. The interface to COMND is rather complex; we shall discuss it in Section 26, page 469.

19.2.5 Pseudo Interrupts and Traps

The *PSeudo Interrupt System* (PSI System) provides a means for a process (or for a superior process) to transfer control to an interrupt routine upon the occurrence of any of a variety of events. For example, a program can request that an interrupt occur whenever a particular character is typed. Thereafter, when that character is typed, normal execution will be suspended and the interrupt routine will be run. The ability to obtain interrupts is quite useful; the alternative, which is quite wasteful, requires periodic checking by the program to see if the specific event has occurred.

The trap system is somewhat similar to the PSI system. However, the trap system can detect and respond only to arithmetic and pushdown overflow conditions. After the program has enabled the trapping of overflows, when an overflow occurs the normal execution of the program is suspended and the trap service routine is run. The trap service routine will be provided with the program counter that points to the instruction following the one that caused the overflow. By examining the instruction and its result, the program can determine how to proceed. For example, the accumulator in which the incorrect result appears might be changed before resuming the trapped–from program. The alternative to using the trap mechanism is to place a JFCL instruction to test for arithmetic overflow after every arithmetic operation.

19.2.6 Interprocess Communication

The *InterProcess Communication Facility* (IPCF) allows messages to be sent between cooperating processes. In connection with the PSI system, IPCF provides a useful means for processes to communicate requests and responses.

Chapter 20

File Output

This chapter offers two programs to exhibit file output. The first is kept as simple as possible. In the second, we exhibit some further techniques.

However, before we get to the main event, we will find it helpful to add to our collection of tools. As we find ourselves dealing with additional TOPS–20 system calls, it is appropriate to introduce some of the macros that can be used to simplify the appearance of our programs.

20.1 MACSYM Macro Package

The universal file `MACSYM` contains macro definitions and run–time support code which implement a number of useful machine–language coding facilities used in the monitor and system program sources. These facilities include variable naming and storage conventions and data structure techniques similar in principle to capabilities found in some implementation languages.

We introduce `MACSYM` in the following programs by including the name `MACSYM` as an argument to the `SEARCH` pseudo–op.

`MACSYM` contains several convenient instruction abbreviations. Also, it contains macros to help construct the complex data structures required by various system calls.[1]

(`MACSYM` includes `OPDEF`s for `CALL` and `RET`. When we use `MACSYM`, we can omit defining `CALL` and `RET`.)

20.1.1 MOVX Macro

For the time being, we focus on only one macro definition from `MACSYM`. When we use a great variety of JSYS calls, it is easy to become perplexed about what flags go in which half of the accumulator. `MACSYM` provides a set of macros to deal with this problem.

The `MOVX` macro will assemble any one of `MOVEI`, `MOVSI`, `HRROI`, `HRLOI`, or `MOVE` depending on the argument. For example,

[1] Be aware that `MACSYM` has many more macros and facilities than we discuss here. You may find your program symbols conflicting with macro names defined in `MACSYM`. Such conflicts often result in mysterious assembly–time errors from what appears to be perfectly reasonable source code.

```
                                        ;Assembles as:
        MOVX    AC,GJ%OLD!GJ%SHT        ;  MOVSI   AC,(GJ%OLD!GJ%SHT)
        MOVX    AC,<FH%SLF,,777777>     ;  HRLOI   AC,FH%SLF
```

The definition of MOVX is lengthy but straightforward. MOVX includes the use of several features of MACRO that are new to us. These will be discussed below.

```
 DEFINE MOVX (AC,MSK)<
 ..MX1==MSK                             ;;Evaluate the given expression.
 .IFN ..MX1,ABSOLUTE,<                  ;;If the expression isn't absolute
        MOVE AC,[MSK]>                  ;;We must use a literal.
 .IF ..MX1,ABSOLUTE,<                   ;;If it is absolute, find a good way
        ..MX2==0                        ;;Set flag. We Haven't done it yet.
        IFE <..MX1>B53,<                ;;If the left half is zero,
            ..MX2==1                    ;;do it as a MOVEI
            MOVEI AC,..MX1>             ;;LH is zero.
        IFE ..MX2,<                     ;;If we haven't done it yet, and
          IFE <..MX1>B17,<              ;;if the Right half is zero, then
            ..MX2==1                    ;;do it as a MOVSI
            MOVSI AC,(..MX1)>>          ;;RH 0, do as LH only
        IFE ..MX2,<                     ;;If we still haven't done it,
          IFE <<..MX1>B53-^O777777>,<    ;;and the left half is -1
            ..MX2==1                    ;;we can do it as
            HRROI AC,<..MX1>>>          ;;HRROI, with the LH -1
        IFE ..MX2,<                     ;;If we haven't done it yet,
          IFE <<..MX1>B17-^O777777B17>,< ;;maybe the right half is -1
            ..MX2==1                    ;;if so, do it via
            HRLOI AC,(..MX1-^O777777)>>  ;;HRLOI
        IFE ..MX2,<                     ;;After all that, if we haven't found a
          MOVE AC,[..MX1]>              ;;better way, give up and use a literal.
        >                               ;;End of .IF ..MX1 ABSOLUTE
 PURGE ..MX1,..MX2                      ;;Discard temporary symbols
 >                                      ;End of MOVX
```

.IF and .IFN mean *if* and *if not*, respectively. These are conditional operators that extend MACRO's conditional assembly capabilities by allowing the program to ask questions about the *properties* of a symbol, not just the value of a symbol.

One of these conditional operators is followed by an expression, a comma, a *qualifier*, another comma, and, enclosed in brackets, < and >, the text to be included if the condition is satisfied. If the argument expression has the property specified by the qualifier, .IF is satisfied. If the argument doesn't have the stated property, .IFN is satisfied.

In the definition of MOVX, the *absolute* property of a symbol is being tested; a symbol is absolute if its value is known and is not relocatable. If the symbol ..MX1 is an absolute quantity, i.e., if it is locally defined and not relocatable, then the main body of the MOVX macro will be assembled. If ..MX1 is not absolute, MOVX assembles a MOVE of a literal without trying any of the optimized forms.

The notation ^O makes MACRO interpret the number that follows as octal. This appears in the macro definition because the programmer may have changed the prevailing radix to be other than octal. Note that ^O is two characters; it is *not* CTRL/O.

We saw the character B used as a shifting operator in our example of LUUOs. An expression such as

<..MX1>B53 means the value of the expression ..MX1 shifted so that the least significant bit appears in "bit 53". Of course, there is no bit numbered 53. However, position 53 is eighteen places to the right of the rightmost bit, bit 35. Thus, this expression right shifts the value ..MX1 by eighteen bits. This moves the left half of the value ..MX1 to the right half of the expression <..MX1>B53.

The symbol ..MX2 is a flag. Initially, it is set to zero. As soon as any one of the optimized forms of MOVE can be generated to perform this MOVX, this flag is set to one. After the flag is set to one, no further attempts to generate instructions will be made. If all attempts to make an optimized form fail, MOVX will generate a simple MOVE of a literal.

A new form of comment is also present in this definition. By using two semicolons, we tell MACRO that these comments are to be deleted from the body of the macro. These comments will appear with the definition of the macro, but they will be absent from the expansion when the macro is called. This produces neater listings. Also it saves space inside of MACRO: otherwise, these comments would be stored as part of the macro definition.

PURGE is a pseudo–op that causes MACRO to forget the name and value of the symbols specified as arguments. The symbols ..MX1 and ..MX2 are temporary and have no importance outside of the macro; therefore, they are removed from the symbol table by the PURGE pseudo–op. It is not an error to attempt to purge a symbol that does not already exist.

20.2 Example 9 — File Output

The following is a very simple example of file output. In a sense, we are reverting to a program as simple as the one in our very first example. This is done so we can concentrate on the issue at hand: writing a file.

```
        TITLE   FILE OUTPUT - Example 9
        SEARCH  MONSYM,MACSYM

A=1                                     ;Define symbolic names for ACs
B=2
C=3
.PSECT  CODE/RONLY,1002000
START:  RESET

;Obtain a Job File Number (JFN) for the output file SAMPLE.OUT.
        MOVX    A,<GJ%SHT!GJ%FOU>       ;short form, output file
        HRROI   B,[ASCIZ/SAMPLE.OUT/]   ;pointer to file name
        GTJFN                           ;get a JFN (job file number)
         ERJMP  ERROR                   ;error
        MOVEM   A,OJFN                  ;remember JFN of output file

;Open the file for output
        HRRZ    A,OJFN                  ;Get the JFN (avoid any flags)
        MOVE    B,[070000,,OF%WR]       ;7-bit bytes
        OPENF                           ;open the file for output
         ERJMP  ERROR                   ;error
```

```
;Send some data to the file
        HRRZ    A,OJFN                  ;JFN in 1, String pointer in 2
        HRROI   B,[ASCIZ/This is sample data to be written to the file
/]
        MOVEI   C,0                     ;Stop at null byte
        SOUT                            ;Send data to the file
         ERJMP  ERROR                   ;Jump in case of an accident

;Close the file and release the JFN
        HRRZ    A,OJFN                  ;JFN needed in register 1
        CLOSF                           ;Close the JFN
         ERJMP  ERROR

;Stop
        HRROI   A,[ASCIZ/Done
/]
        PSOUT
        HALTF
        JRST    START

;here in case of an error
ERROR:  HRROI   A,[ASCIZ/Error: /]      ;send error typeout
        ESOUT
        MOVEI   A,.PRIOU                ;send error msg to terminal
        HRLOI   B,.FHSLF                ;this fork, most recent error
        MOVEI   C,0                     ;no limit to byte count
        ERSTR                           ;convert last error -> string
         JFCL                           ;TWO possible error returns
         JFCL
        HRROI   A,CRLF
        PSOUT
        HALTF
        JRST    START

CRLF:   BYTE(7)15,12

.PSECT  DATA,1001000
OJFN:   0                               ;output JFN

        END     START
```

Most of this program is concerned with putting the output into a file instead of printing it on the terminal.

20.2.1 Getting a JFN

The sequence following `START` obtains a JFN (job file number) for the output file:

```
MOVX    A,<GJ%SHT!GJ%FOU>
HRROI   B,[ASCIZ/SAMPLE.OUT/]
GTJFN
 ERJMP  ERROR
MOVEM   A,OJFN
```

The flag bit GJ%SHT indicates that we are using the short form of the GTJFN JSYS. There is a long form of GTJFN that requires many arguments and a lengthy explanation; the short form is much easier to use. (On the other hand, the short form lacks such amenities as reprompting when CTRL/R is typed, etc.).

The flag bit GJ%FOU indicates that a JFN for an output file is wanted; the JFN that is assigned will include a new, higher, generation number if the specified file name already exists. GJ%FOU is usually seen in output JFNs (although for special effects it might be omitted).

These flags are set in register 1 (our A) by

```
MOVX    A,<GJ%SHT!GJ%FOU>
```

Because the expression GJ%SHT!GJ%FOU has bits set only in the left half, MOVX will construct

```
MOVSI A,(GJ%SHT!GJ%FOU)
```

This repeats the trick that we first mentioned in connection with floating–point immediates. The parentheses cause the left and right halves of the expression to be swapped and placed in the instruction.

The absence of a bit, GJ%FNS, indicates that the contents of register 2 (our B) will be interpreted as a pointer to a string that specifies the file. Register 2 is set up to point to the ASCIZ string SAMPLE.OUT, the file name of our output file.

It is interesting to note that when GJ%FNS is set in the call to GTJFN, register 2 is interpreted as containing a pair of JFNs. Text is read from the file specified by the input JFN in the left half of 2. Text is echoed to the output JFN, specified in the right half of 2. The text that is read from the input file should contain the name of the file for which a new JFN is sought by this call to GTJFN. In other words, it is easy to tell the system to read the file name from the terminal (or from another file that is open for input).

Assuming the call to GTJFN goes well, a JFN is returned in register 1, and GTJFN will skip one instruction. The JFN is stored in OJFN, *Output JFN*, for future use. If GTJFN encounters any problem then it will fail to skip and TOPS–20 will find the ERJMP instruction and interpret it as an unconditional jump.

20.2.2 Opening the File for Output

After the JFN is obtained, the file must be *opened* for output. The following sequence accomplishes this:

```
        HRRZ    A,OJFN                  ;Get the JFN (avoid any flags)
        MOVE    B,[070000,,OF%WR]       ;7-bit bytes
        OPENF                           ;open the file for output
         ERJMP  ERROR                   ;error
```

Although the call to GTJFN did not ask for flags to be returned, the instruction HRRZ A,OJFN is a good idea because it allows this fragment to be used in other contexts where JFN flags may be present. It is a good practice to reduce a JFN to a halfword quantity in all cases where flags are undesirable. In a later example, we shall demonstrate the use of these flags; in most cases the flags are not desired.

The JFN is a *handle* on the file; to possess a handle gives us some power over the file. However, for reading, writing, modifying, or appending to the file, a handle alone is insufficient. We must inform the operating system of our intended actions regarding the file by means of the OPENF, *OPEN File*, JSYS. OPENF requires a JFN in register 1 and a description of our intentions in register 2.

In OPENF, register 2 contains the byte size and intended access mode. In this example, the byte size is set to 7, which is the standard size for character input and output. The byte size is set in bits 0:5 of register 2. In the right half of register 2, the bit OF%WR informs the system that we intend to write the file.

If all goes well, the system will add the appropriate structures describing the new file to the directory where it is being written. OPENF will skip to signify success. On failure, the ERJMP to ERROR is executed. Normally, write access will be granted only if there is no other program that is already writing on this file. Moreover, once write access to this file has been granted, other programs are locked out from being able to write on this file. Generally, this is exactly what is wanted. This mode of access for file writes is called *frozen access*.

20.2.3 Sending Characters to the File

The SOUT JSYS will write a string of characters to a file:

```
        HRRZ    A,OJFN
        HRROI   B,[ASCIZ/This is sample data to be written to the file
/]
        MOVEI   C,0
        SOUT
         ERJMP  ERROR
```

The SOUT JSYS requires a JFN in register 1. Register 2 is set up with a string descriptor. As we have so often seen before, the HRROI instruction can make an adequate string descriptor. The SOUT JSYS is somewhat more flexible than PSOUT in determining when the string ends. Register 3 describes the string termination condition. In this case the string ends with a null byte, so we set register 3 to zero.

It might be noted that as an alternative to placing a zero in register 3, the SOUT JSYS will accept a negative byte count instead. Or, if register 3 is set to a positive byte count, then the SOUT will send that many bytes or stop as soon as a byte is sent that matches the byte in register 4.

20.2.4 Closing the File

After all output has been sent to the file, the file must be closed.

```
        HRRZ     A,OJFN
        CLOSF
         ERJMP   ERROR
```

This code fragments picks up the JFN of the output file into register 1 (our `A`) and closes the file via the `CLOSF` JSYS. Closing a file makes the file permanent. The operating system performs whatever functions are needed to make certain the file exists in a form that can be located by other programs. Until the `CLOSF` is performed, the file is susceptible to being obliterated if the program terminates abnormally. The `CLOSF` JSYS also releases the JFN. After releasing the JFN, if any attempt is made to use the number in `OJFN` as a JFN, only errors will result.

20.2.5 Error Handling

One more useful code fragment is the one called `ERROR`.

```
ERROR:  HRROI   A,[ASCIZ/Error: /]   ;send error typeout
        ESOUT
        MOVEI   A,.PRIOU             ;Send error msg to terminal
        HRLOI   B,.FHSLF             ;this fork, most recent error
        MOVEI   C,0                  ;no limit to byte count
        ERSTR                        ;convert last error -> string
         JFCL                        ;Two possible error returns.
         JFCL
        HRROI   A,CRLF
        PSOUT
        HALTF
        JRST    START
```

`ERROR` starts out with an `ESOUT` to clear any type–ahead and to give the error message a uniform appearance. Then an `ERSTR` JSYS is used to convert a TOPS–20 error number to a string. In this example, `.PRIOU` is specified as the destination for the resulting string; this just makes the message show up on the terminal. Register `B` is initialized to contain `.FHSLF` in the left; this is the *fork handle* of the the current process (i.e., the *fork*). The `HRLOI` instruction will place a -1 in the right half of `B`. This -1 signifies the most recent error. Thus, the `ERSTR` will print the message corresponding to the most recent error in the current fork.

Register 3, `C`, is initialized to zero, signifying no limit to the size of the string that may be written by `ERSTR`; the terminal device can hold anything that `ERSTR` cares to dish out.

`ERSTR` permits two error returns! The normal return skips two instructions following the `ERSTR`. It doesn't seem too worthwhile to install any error handling to deal with errors from the `ERSTR` JSYS.

Although our error handling is still primitive, at least now we print a standard message describing what has gone wrong. As we advance our study of file input and output, the number of possible errors that we must cope with will grow enormously.

20.3 Example 10 — Long–Precision Fixed–Point Output

This example writes the decimal numbers 0 through 69 and the corresponding powers of 2 into the file SAMPLE.OUT.

The portion of the program in the immediate vicinity of LOOP does the computations. The routine DECPBG is explained following the text of this program; DECPBG uses the DECFIL routine that was explained in Example 8.

The output activity is initialized in the region following the label START. The output activity terminates in the vicinity of DONE. The routine PUTLIN is the basic output routine. To eliminate repetitive system calls, this program buffers a line at a time; it performs a SOUT to write each line to the file.

```
        TITLE   Long-precision Decimal Output & File Output.  Example 10
        SEARCH  MONSYM,MACSYM
A=1
B=2
C=3
D=4
P=17
PDLEN==100

        .PSECT  CODE/RONLY,1002000

START:  RESET
        MOVE    P,[IOWD PDLEN,PDLIST]
        CALL    OPNOUT                  ;Open the Output File
        MOVE    A,[POINT 7,LINBUF]
        MOVEM   A,OUTPNT                ;pointer to line buffer
        MOVE    B,[POINT 7,[ASCIZ/Sample Results file

/]]
        CALL    COPYST                  ;copy string to output line.

                                        ;Perform the computations here
        SETZM   COUNT                   ;the counter
LOOP:   MOVE    A,COUNT
        MOVEI   C,4                     ;Print 4 columns
        CALL    DECFIL                  ;decimal output with blank fill.
        MOVE    B,[POINT 7,[ASCIZ/  /]]
        CALL    COPYST
        MOVEI   B,1                     ;load the double-precision
        MOVEI   A,0                     ;constant 1 into A and B
        ASHC    A,@COUNT                ;double shift: 70-bit result
        CALL    DECPBG                  ;Print BIG number
        MOVE    B,[POINT 7,CRLF]
        CALL    COPYST                  ;copy CRLF to output
        CALL    PUTLIN                  ;write line.
        AOS     A,COUNT                 ;increment counter
        CAIGE   A,^D70                  ;enough yet?
         JRST   LOOP                    ;no
```

```
;Here after printing the first 70 lines.  Print the last result
        MOVE    B,[POINT 7,[ASCIZ/The largest double integer is:
      /]]
        CALL    COPYST                  ;some final things to output
        HRLOI   A,377777                ;largest positive number to A
        MOVE    B,A                     ;and a copy to B.
        CALL    DECPBG                  ;print the largest positive number
        MOVE    B,[POINT 7,CRLF]
        CALL    COPYST
        CALL    PUTLIN

;Finished with computations.  Close the file
DONE:   HRRZ    A,OJFN                  ;get the JFN
        CLOSF                           ;close and release it
         ERJMP  ERROR
        HALTF
        JRST    START

;Subroutine to obtain a JFN and open it for an output file.
OPNOUT: MOVX    A,<GJ%SHT!GJ%FOU>       ;short form, output file
        HRROI   B,[ASCIZ/SAMPLE.OUT/]   ;pointer to file name
        GTJFN                           ;get a JFN (job file number)
         ERJMP  ERROR                   ;error
        MOVEM   A,OJFN                  ;remember JFN of output file
        HRRZ    A,A                     ;a good habit.  remove JFN flags
        MOVE    B,[070000,,OF%WR]       ;7-bit bytes.  Output of course
        OPENF                           ;open the file for output
         ERJMP  ERROR                   ;error
        RET

;Copy a string pointed to by B to the output buffer
COPYST: ILDB    A,B                     ;copy string from B to
        JUMPE   A,CPOPJ                 ;the output pointer
        IDPB    A,OUTPNT
        JRST    COPYST

;Transmit the output buffer to the file
PUTLIN: HRRZ    A,OJFN                  ;Place one line into the output file
        HRROI   B,LINBUF                ;string pointer to the line
        MOVEI   C,0                     ;stop on zero.
        IDPB    C,OUTPNT                ;store the zero in the output line
        SOUT                            ;send it.
         ERJMP  ERROR
        MOVE    A,[POINT 7,LINBUF]      ;reset the output line pointer
        MOVEM   A,OUTPNT
        RET
```

```
;Decimal output with fill
DECZFL: SKIPA   B,["0"]              ;fill character is zero
DECFIL:  MOVEI   B," "               ;fill character is space
        SKIPGE  A                    ;is operand negative?
         TLO     B,400000            ;Yes, remember that
        MOVEM   B,FILLCH
        JUMPGE  A,DECFL1             ;jump ahead if operand is positive
        SUBI    C,1                  ;count a column for the sign character
        MOVM    A,A                  ;and make argument positive
DECFL1: IDIVI   A,12                 ;this is fairly standard
        PUSH    P,B                  ;save a remainder
        SUBI    C,1
        JUMPE   A,DECFL3             ;jump if it's time to do the fill
        CALL    DECFL1
DECFL2: POP     P,A                  ;retrieve a saved remainder
        ADDI    A,"0"
        IDPB    A,OUTPNT
CPOPJ:  RET

DECFL3: JUMPLE  C,DECFL4             ;jump if no need to pad
        MOVE    A,FILLCH             ;Pickup the fill character.
        IDPB    A,OUTPNT             ;store fill character
        SOJG    C,.-1                ;loop until done filling.
DECFL4: JUMPG   A,DECFL2             ;jump unless flagged as negative
        MOVEI   A,"-"                ;get a minus sign
        IDPB    A,OUTPNT             ;and put it in the output
        JRST    DECFL2               ;now, go print the number.
```

```
;Print double-length integer
;The comments in the program are keyed to the discussion in the text.
DECPBG: MOVE    C,B                     ;Save the low part  "b"
        JUMPGE  A,DECPB0                ;jump ahead unless negative
DECPB0: IDIV    A,[^D10000000000] ;break up the high part "s" "t"
        DIV     B,[^D10000000000] ;break up the low parts "u" "v"
        JUMPE   A,DECPB2                 ;less work for no very big stuff
        TDNE    B,[377777777777]         ;is "u" zero?
         SUBI   A,1                      ;no, so make A ones complement
        DIV     A,[^D10000000000] ;handle the big parts "s" "u"
        MOVM    C,C                      ;low parts in positive form
        MOVM    B,B
        JRST    DECPB4                   ;join the rest of the code

DECPB0: IDIV    A,[^D10000000000] ;break up the high part "s" "t"
        DIV     B,[^D10000000000] ;break up the low parts "u" "v"
        JUMPE   A,DECPB2                 ;less work for no very big stuff
        DIV     A,[^D10000000000] ;handle the big parts "s" "u"
DECPB4: PUSH    P,C                     ;Save lowest part "v"
        PUSH    P,B                     ;save smaller stuff "x"
        MOVEI   C,4                     ;4 spaces enough for this part "w"
        CALL    DECFIL                  ;fill with blanks
        POP     P,A                     ;the second part "x"
        MOVEI   C,12                    ;12 digits
        CALL    DECZFL                  ;Decimal print, fill with leading 0
DECPB1: POP     P,A                     ;the third part "v"
        MOVEI   C,12
        JRST    DECZFL                  ;print with leading zero fill

DECPB2: MOVE    A,B                     ;high part. "u"
        JUMPE   A,DECPB3                ;Jump if high part is zero
        MOVM    C,C
        PUSH    P,C                     ;save low part
        MOVEI   C,16                    ;14 spaces
        CALL    DECFIL                  ;decimal print.  fill space.
        JRST    DECPB1                  ;write low part ("v") in 10 columns.

DECPB3: MOVE    A,C                     ;write the low part in 24 columns
        MOVEI   C,30                    ;blank fill.
        JRST    DECFIL
```

```
ERROR:  HRROI   A,[ASCIZ/Error: /]      ;send error typeout
        ESOUT
        MOVEI   A,.PRIOU                ;output error message to terminal
        HRLOI   B,.FHSLF                ;this fork, most recent error
        MOVEI   C,0                     ;no limit to byte count
        ERSTR                           ;convert error number to a string
         JFCL                           ;would you believe TWO error returns?
         JFCL
        HRROI   A,CRLF
        PSOUT
        HALTF
        JRST    START

CRLF:   BYTE(7)15,12

        .PSECT  DATA,1001000
PDLIST: BLOCK   PDLEN
COUNT:  0                               ;index variable, 0 to 69
FILLCH: 0                               ;fill character for DECFIL
LINBUF: BLOCK   40                      ;output line buffer
OUTPNT: 0                               ;byte pointer for output line
OJFN:   0                               ;output JFN
        END     START
```

A portion of the resulting output file, `SAMPLE.OUT`, is shown below:

```
Sample Results file


   0                          1
   1                          2
   2                          4
   3                          8
   4                         16

  ...

  67     147573952589676412928
  68     295147905179352825856
  69     590295810358705651712
The largest double integer is:
        1180591620717411303423
```

This program has two items of interest to us. One is a double–precision integer printout routine. The second is a further example of output to a file. In this example we apply some of our previously developed ideas of buffering output to the situation of output to a file.

The double–precision integer printer is an extension, although not an obvious one, of the standard decimal printer. The section that follows explains the mathematical (well, arithmetic) basis of the routine. Chiefly, the routine changes double–precision fixed–point binary numbers into numbers that are multiples of large powers of 10.

20.3.1 Mathematical Basis of the DECPBG Routine

Suppose we have a very large positive number to print. The number is in a double word; call the parts a on the left and b on the right. The original number has the value $a \times 2^{35} + b$.

In general, we want to compute remainders by dividing this number by 10. However, the `DIV` instruction requires that the divisor be larger than the a part of the number.[2] Since the a part can range up to $2^{35} - 1$, we must choose a very large divisor. Suppose we choose 10^{10} as the divisor; 10^{10} is the largest power of 10 less than 2^{35}. If we divide $a \times 2^{35} + b$ by 10^{10}, the remainder would be the low–order ten digits of the number, and the quotient would be the more significant digits. This seems like a good way to start. However, there is a catch. Since a can be larger than 10^{10}, we cannot be assured of successfully dividing the quantity $a \times 2^{35} + b$ by 10^{10}.

To solve this problem, we resort to some tricks by which we rewrite the quantity $a \times 2^{35} + b$ to reveal a more useful structure in terms of polynomial factors of radix 10^{10}. We start by rewriting a as $a = s \times 10^{10} + t$. Then, the original number can be written as

$$a \times 2^{35} + b = (s \times 10^{10} + t) \times 2^{35} + b = s \times 2^{35} \times 10^{10} + t \times 2^{35} + b.$$

The number $t \times 2^{35} + b$ can be rewritten as $u \times 10^{10} + v$. There is no problem using `DIV` to divide $t \times 2^{35} + b$ by 10^{10} because t is less than 10^{10}: it is the remainder resulting from having divided a by 10^{10}.

So the entire number can be rewritten thus:

$$a \times 2^{35} + b = s \times 2^{35} \times 10^{10} + u \times 10^{10} + v = (s \times 2^{35} + u) \times 10^{10} + v,$$

where

s	=	the quotient of $a \div (10^{10})$;	$s < 10$, since $a < 2^{35} < 10 \times 10^{10}$
t	=	the remainder of $a \div (10^{10})$;	$t < 10^{10}$
u	=	the quotient of $(t \times 2^{35} + b) \div (10^{10})$;	$u < 2^{35}$
v	=	the remainder of $(t \times 2^{35} + b) \div (10^{10})$;	$v < 10^{10}$

It should be clear that the portion called v contains the ten least significant digits of the result. The portion $s \times 2^{35} + u$ can be attacked by dividing by 10^{10}. We know that s is smaller than 10; our earlier caution about avoiding the `DIV` instruction no longer applies. The number $s \times 2^{35} + u$ is rewritten as $w \times 10^{10} + x$. Therefore the original number has been decomposed into

$$a \times 2^{35} + b = w \times 10^{20} + x \times 10^{10} + v,$$

where, in conjunction with the definitions of s, t, u and v above, we define

w	=	the quotient of $(s \times 2^{35} + u) \div (10^{10})$
x	=	the remainder of $(s \times 2^{35} + u) \div (10^{10})$

The number w is printed first, followed by x in ten digits with leading zeros, followed by v in ten digits with leading zeros. The comments in the `DECPBG` routine reflect the discussion above. The first instructions in `DECPBG` calculate the s and t given the high–order portion a. The double–word containing t and b is then broken into u and v. The v portion, in register `C`, is pushed.

If the s part is zero, the program jumps to `DECPB2`; the number is somewhat easier to print. If the s part is non–zero, the double–word s and u is divided by 10000000000. The quotient, w is printed in four digits. The remainder, x, is printed in ten digits. Finally, the v part is printed in ten digits.

The instructions at `DECPB2` and `DECPB3` are used for printing smaller numbers. Appropriate numbers of leading spaces are output.

[2] Yes, we could have used the `DDIV` instruction to avoid this problem. But then, suppose you wanted to print 140–bit numbers.

The points made in the discussion above are generally applicable to negative numbers, however, a few things have to be changed. In double word negative numbers, the most significant word contains the ones complement of a, unless the least significant word, b, is zero, in which case the most significant word conatins the twos complement of a. (The low part being 0 is represented by a word in which bit 0 is 1 and all the other bits are zero.)

The first `IDIV` instruction results in a twos complement quotient and a remainder in ones or twos complement depending on whether the original operand was ones or twos complement. In any case, the remainder is correct for the next `DIV` instruction. But following that instruction, if u is non-zero, then the value of s in twos complement notation must be changed to ones complement by subtracting 1 prior to the third division.

The most significant portion of the result carries the sign, other portions are changed to contain their positive magnitude. `DECFIL` has been changed to print negative numbers.

20.3.2 Sending Characters to the File

An area called `LINBUF`, and an associated byte pointer, `OUTPNT`, are used by the program to buffer file output on a line–by–line basis. (Other arrangements are possible; alternatives will be demonstrated in later examples.) Single characters are deposited into the line buffer via the byte pointer `OUTPNT`. Character strings are copied to the output buffer by the `COPYST` subroutine.

`COPYST` (*COPY STring*) is a completely straightforward example of using byte pointers in a loop to copy a string from one place to another. `COPYST` is called with a source pointer in register `B` and a destination pointer in `OUTPNT`. Each byte loaded via `B` is deposited via `OUTPNT`. A zero byte in the source string terminates the loop; the zero byte is not deposited in the output string.

It might be noted that an optimization of `COPYST` is possible. This removes one instruction from that loop, a savings of 25%:

```
CPYST1:  IDPB   A,OUTPNT                ;send a byte
COPYST:  ILDB   A,B                     ;get a byte
         JUMPN  A,CPYST1                ;loop unless null
         RET                            ;return
```

The `PUTLIN` routine should be called with some frequency to effect the transfer of data from `LINBUF` to the file. At `PUTLIN`, register 1 is loaded with the JFN, register 2 is loaded with a string pointer to the line buffer. Register 3 is loaded with a zero to signify that a zero byte will terminate the string. Then, to make sure that a zero byte is present in the string, a zero byte is deposited via `OUTPNT`.

The `SOUT` JSYS will write a string, in this case terminated by a null byte, to the output file specified by the `JFN` in register 1. When the `SOUT` succeeds, `OUTPNT` will be reset to the beginning of `LINBUF` and the `PUTLIN` routine returns.

Chapter 21

Arrays

An *array* is a *data structure*, an organized collection of data items, in which each item is uniquely identified by one or more *index* numbers. We begin our discussion with the simplest case, the one–dimensional array. Later in this chapter we will discuss two–dimensional and multi–dimensional arrays.

At the outset we confine our discussion to arrays that reside within the same address section as the program. In srefExtArray we shall consider arrays in address sections remote from the program.

21.1 One–Dimensional Arrays

The one–dimensional array is a collection of data items that are stored in consecutive memory locations. Each data item is identified (or *addressed*, or *accessed*) by an integer index number that represents the item's position in the collection. For example, suppose we build an array called `TABLE` that contains 100 (decimal) items, corresponding to the Pascal language declaration `TABLE: ARRAY [0..99]`. The legal index values by which we can refer to `TABLE` are the integers in the range from 0 to 99.

In assembly language we can allocate this array by means of the `BLOCK` pseudo–op:

```
TABLE:  BLOCK   ^D100
```

The easiest way to reference an element in this list, e.g., `TABLE[67]`, is to use an index register. Simply load any one of the accumulators from 1 to 17 with the index value, `^D67`, and write an indexed instruction:

```
MOVEI   15,^D67          ;load the index (item number)
MOVE    3,TABLE(15)      ;reference to TABLE[67]
```

Of course, there is little point in using an index register if the index number is a constant. Rather, the power of arrays and index registers becomes apparent when the references appear in loops.

For example, the following program fragment will locate the minimum and maximum elements of the array called `TABLE`. The index value for the minimum will be placed in `MINLOC`; the index value for the maximum will be placed in `MAXLOC`.

```
MINLOC=5                              ;Index of smallest element
MAXLOC=6                              ;Index of largest element
INDEX=7                               ;Index to scan the array
TEMP=10                               ;AC to hold each array item in scan
          MOVSI   INDEX,-^D100        ;Load INDEX with -^D100,,0
                                      ;-^D100 is a control count (item
 count)
                                      ;The zero is the initial index value.
          SETZB   MAXLOC,MINLOC       ;Initial indices for maximum and minimum
LOOP:     MOVE    TEMP,TABLE(INDEX)   ;Get an array element.
          CAMLE   TEMP,TABLE(MAXLOC)  ;Skip unless larger than old maximum
           HRRZ   MAXLOC,INDEX        ;Save index to new maximum element.
          CAMGE   TEMP,TABLE(MINLOC)  ;Skip unless smaller than old minimum
           HRRZ   MINLOC,INDEX        ;Save index to new minimum element
          AOBJN   INDEX,LOOP          ;Scan through the entire array.
```

In this example the register that we have called `INDEX` steps through all the values from 0 to 99 (in the right half); these values are used to access items in the array called `TABLE`.

Zero–origin indexing is the name of the method we have used to access the array named `TABLE`. Fundamentally, all arrays in the computer use zero as the lower index number. Sometimes it is inconvenient in an algorithm to use zero as the lower bound for an array; the Heapsort algorithm that we discuss later (in Section 23.7, page 390) is an example of a process in which the computations would be painfully more complex if we were restricted to using only zero–origin indexing.

To escape from the constraint of zero–origin indexing in those cases where it is convenient to do so, we can resort to a variety of techniques to shift the origin of the array. For example, if you write a program to deal with the values of the gross national product for the years 1968 through 1990, you would want to have an array corresponding to the declaration `GNP: ARRAY [1968..1990]`. We can accomplish this as follows:

```
GNPLNG==^D1990-^D1968+1          ;define the length of the array

GNP:    BLOCK   GNPLNG           ;allocate storage for this array

GNPORG==GNP-^D1968               ;define the virtual origin.
```

Here we have defined a storage area for the array called `GNP`. Note that the length of the array is always given by the formula

$$\text{length of the array} = \text{highest index} - \text{lowest index} + 1$$

Sometimes it is easy to forget to add the one in this formula; the one is necessary because even when the lower index and upper index are identical you still have to allocate one word.

The other thing we have done is to define a new symbol called `GNPORG` that is the address of the non–existent array element `GNP[0]`. The usefulness of the symbol `GNPORG` will be evident from inspection of the three code fragments that follow; each has the same effect:

```
        MOVE    INDEX,[-GNPLNG,,^D1968]
LOOP:   MOVE    3,GNP-^D1968(INDEX)
        CALL    PRINT
        AOBJN   INDEX,LOOP
```

compared to

```
        MOVE    INDEX,[-GNPLNG,,^D1968]
LOOP:   MOVE    3,GNPORG(INDEX)
        CALL    PRINT
        AOBJN   INDEX,LOOP
```

compared to

```
        MOVE    INDEX,[-GNPLNG,,^D1968]
LOOP:   HRRZ    3,INDEX
        SUBI    3,^D1968
        MOVE    3,GNP(3)
        CALL    PRINT
        AOBJN   INDEX,LOOP
```

In these fragments, the register called `INDEX` holds an `AOBJN` pointer for stepping through the entire `GNP` array. The value of each item is picked up into register 3 and the `PRINT` subroutine is called. We can assume that `PRINT` will use the value in register 3. Let us also assume that `PRINT` will use the year number found in the right half of `INDEX`. Perhaps a table of years and corresponding GNPs is being printed.

In detail, the first two of these code fragments assemble the same thing. They differ only in that the second fragment uses the symbol `GNPORG` instead of the equivalent expression `GNP-^D1968`. The reason we prefer to use the symbol `GNPORG` is that it is easier to type, and it is more likely to be used correctly.

The third code fragment is inferior to the other two. In it, the index value is copied from the right half of `INDEX`; from this value we subtract the lowest array index. This subtraction normalizes the index value, producing the effect of zero–origin indexing. The result of the subtraction is sometimes called an *effective index* (or, an *effective subscript*). Since this method involves three instructions to access the array, as compared to the one instruction used in the other cases, it is less favored than the other versions. Where possible, do your constant subtractions once, in the assembler, instead of at run time.

21.1.1 Example 11 — Factorials to 100!

The program that follows demonstrates the use of an array in the computation of factorials. Factorials are important in various computations of probabilities and permutations. The factorial of a positive integer k, written as $k!$, is defined by the expression $k! = k \times (k-1)!$. The value 0! is defined by convention to be 1. Thus we can display a small sample of factorials:

```
0!  =           1  (by definition)
1!  =   1*0! =   1
2!  =   2*1! =   2
3!  =   3*2! =   6
4!  =   4*3! =  24
5!  =   5*4! = 120
6!  =   6*5! = 720
```

Factorials grow very rapidly; the number 100! requires more than 150 digits to represent. In order to calculate the value of 100!, we must use an array to hold all the digits. In this example program, a 200–word array called `ACCUM` is defined. Each word of `ACCUM` holds one decimal digit of the number we are calculating. The first word of the array, address `ACCUM+0`, is considered to be the units digit; `ACCUM+1` is the tens digit, `ACCUM+2` is the hundreds digit, etc.

The word `FACT` will count from 0 to 100. The basic calculation consists of multiplying the number represented in the array `ACCUM` by `FACT`, then storing the results back into the array, printing intermediate results, and repeating. After printing the value of 100!, `FACT` reaches 101 and the program exits.

One further control word is used. The number held in `NDIG` represents the index value needed to address the most significant digit in the array `ACCUM`. The number of significant digits increases throughout this calculation; keeping track of the index value of the most significant digit eliminates some effort.

The general outline of this program is quite simple:

```
         Initialize the ACCUM array, FACT, and NDIG to zero.
         Set ACCUM to 1, representing 0!
FLOOP:   Print the current value of FACT and the number in ACCUM.
         Increment FACT; if the result is greater than 100, go to DONE.
         Compute the next factorial.
         Go to FLOOP
DONE:
```

We begin by displaying the definition of the data areas that we have discussed thus far.

```
ACCUM:   BLOCK     ^D200               ;Room for 200 digits
NDIG:    0                             ;index number of most significant digit
FACT:    0                             ;how far we've gotten
```

We will start the computation by setting the array `ACCUM` to contain the representation of the value 1, the value of 0!. `FACT` and `NDIG` will both be set to zero.

When this program is assembled and first loaded into memory, the array `ACCUM` will be set to zero. Perhaps it is unnecessary to zero the array in the initialization of the program. However, if the program were ever restarted, the results of prior computations would be present in this array; the presence of that data would cause the program to produce erroneous results. Therefore, it seems better, more certain, to initialize our data structures each time the program is started.

We will zero the array `ACCUM`, and zero the words `FACT` and `NDIG`. Then, we will set `ACCUM+0` to one. `ACCUM+0` represents the units digit of the result; the entire array then represents the number 1, the

value of 0!.

Zeroing the array ACCUM is accomplished by means of BLT instruction; you may wish to review the discussion in Section 15.1, page 207. The word at ACCUM+0 is set to zero. Register A is then set with a source and destination address pair to control the BLT instruction. The address of ACCUM+0 is the source; the destination is ACCUM+1. A literal is used to hold these two addresses; that literal is copied to register A.

The BLT instruction will copy the word in ACCUM+0, which was set to zero, to ACCUM+1; the new contents of ACCUM+1 are then copied to ACCUM+2. This process continues, propagating zero words throughout the array. The BLT instruction specifies FACT as the ending address; all the words from ACCUM+1 through FACT are set to zero. From the way that we set up the data area, this BLT includes the entire array ACCUM and the words NDIG and FACT.

After the BLT, the AOS instruction increments the zero held in ACCUM+0 to make it 1; thus, the ACCUM array contains the representation of the value 1 for 0!.

```
SETZM   ACCUM                ;zero the array of digits
MOVE    A,[ACCUM,,ACCUM+1]
BLT     A,FACT               ;200 digits, ACCUM thru ACCUM+199
                             ;also zero NDIG and FACT.
AOS     ACCUM                ;start with 0! = 1. ACCUM+0 set to 1
```

The proper functioning of this instruction sequence depends on the definition of the data area. If FACT were moved to be a lower address than ACCUM, or if NDIG were moved somewhere else, then this sequence would not work properly. To guard against such accidental changes, we will add a comment to the place where the data area appears.[1] Remember that the assumptions you make while writing a program are not always obvious to a person who subsequently reads the program. That subsequent reader, who is trying to fix the mistakes or expand the functionality of the program, needs all the help you can give. That unfortunate person who has to fix your program may well be yourself.

FACT and NDIG now contain zero; ACCUM contains the representation of 0!. The desired initialization steps are complete. In order to write the rest of the main program, we will assume that a subroutine called PRINT can be called to print the results. Another subroutine, MULT, will be called to perform the multiplication of the old factorial value by the new value of FACT to give the next factorial value. Of course, MULT and PRINT don't exist yet; we'll have to write them. But we have made progress; the main program can now be written:

```
FLOOP:  CALL    PRINT     ;Print the current value of FACT and the
                          ;corresponding factorial.
        AOS     Y,FACT    ;increment FACT.  Result to Y and to FACT
        CAILE   Y,^D100   ;have we reached the last value yet?
         JRST   DONE      ;yes. go finish up.
        CALL    MULT      ;perform the multiplication
        JRST    FLOOP     ;go print result, etc.
DONE:
```

Now we must talk about the process of multiplication. Suppose that at FLOOP, FACT contains the value 6; the ACCUM array must (if the program is working properly) contain a representation of

[1] Another approach is to define symbols such as ZBEG and ZEND which are the boundaries of the space that is zeroed at startup.

6!, 720. Specifically, ACCUM+2 contains 7, the hundreds digit; ACCUM+1 contains 2, the tens digit; ACCUM+0 contains 0, the units digit. After the PRINT subroutine prints the line 6! = 720, the value in FACT is incremented to 7. The MULT routine is called to perform the multiplication of 7 times 720. When MULT is called, our data structures look like:

ACCUM+3	ACCUM+2	ACCUM+1	ACCUM+0	NDIG	FACT
0	7	2	0	2	7

Recall from our previous discussion that NDIG represents the index number of the most significant digit of the array ACCUM. Since ACCUM contains a representation of decimal 720, the most significant digit resides in ACCUM+2; therefore, NDIG takes the value 2.

In order to perform this multiplication, we process the number in ACCUM, starting at its least significant digit. For each digit, we multiply the old digit by the number in FACT. This product forms the new digit that is stored back in the ACCUM array. Since it is possible for a product to be larger than 9, we divide each product by decimal 10. The remainder is the desired digit; the quotient is the carry that we must add to the next higher product.

Let us now perform the multiplication of 720 by 7. We start by fetching the contents of ACCUM+0 and multiplying by the contents of FACT. ACCUM+0 contains 0; the product is 0. This product is stored back into ACCUM+0. Next, we fetch the contents of ACCUM+1; the value is 2. The product is decimal 14; this is too large to fit back in ACCUM+1, so we divide 14 by 10. The remainder, 4, is small enough to fit in ACCUM+1, so we store it there. The quotient, 1, is carried to the next step of the calculation.

ACCUM+3	ACCUM+2	ACCUM+1	ACCUM+0	NDIG	FACT
0	7	4	0	2	7

Carry = 1. Next digit to process is in ACCUM+2.

In the next step of the calculation, ACCUM+2 is fetched. This value, 7, is multiplied by the contents of FACT. To the product, 49, we must add the carry from the previous step. The result, 50, is too large to fit into ACCUM+2. We divide it by 10. The remainder, 0, is stored in ACCUM+2. The quotient, 5, is carried to the next step.

ACCUM+3	ACCUM+2	ACCUM+1	ACCUM+0	NDIG	FACT
0	7	4	0	2	7

Carry = 5. Next digit to process is in ACCUM+3.

In the final step of the calculation, the number in ACCUM+3 is multiplied by the contents of FACT. The result is 0, but we add the carry, 5, to it. This sum is now stored into ACCUM+3:

ACCUM+3	ACCUM+2	ACCUM+1	ACCUM+0	NDIG	FACT
5	7	4	0	2	7

The multiplication process can now be stopped. We know we can stop because there is no carry out and the current index value, 3, because we stored into ACCUM+3, is greater than the contents of NDIG. We end by updating NDIG to contain the highest index we have used. In this case, NDIG is set to 3.

You might think from the length of the explanation that the resulting program will be long and complicated. Indeed not! It is only a few instructions. In the subroutine that follows, register A contains the carry from the previous stage. Register C is used as an index register; C starts at zero and counts up through at least NDIG to access each element of the previous factorial value. Register A is also initialized to zero, to signify no carry from the previous stage. At each step the program will bring forward a carry from the previous stage. In every case, the sum of the new product plus the previous carry is divided by 10 to produce a new result digit and a new carry.

```
MULT:   SETZB   A,C             ;carry out of previous digit is zero
                                ;C is the index to the current digit
                                ;Start at least significant digit
MLOOP:  MOVE    B,ACCUM(C)      ;get one digit from the ACCUM array
        IMUL    B,FACT          ;mult by the current FACT value
        ADD     A,B             ;add the product to the previous carry
        IDIVI   A,^D10          ;divide to get this digit and carry
        MOVEM   B,ACCUM(C)      ;store one digit.  Carry in A.
        SKIPG   A               ;Skip if any carry is present
         CAMGE  C,NDIG          ;no carry. are we at end?
          AOJA  C,MLOOP         ;must go on.  advance C to next digit
        MOVEM   C,NDIG          ;End. Store the new value of NDIG
        RET
```

Let us re–examine the example we have been using and relate it to this subroutine. After executing the SETZB A,C instruction at MULT, the various data items are as follows:

ACCUM+3	ACCUM+2	ACCUM+1	ACCUM+0	NDIG	FACT	A	B	C
0	7	2	0	2	7	0		0

(The contents of register B are not defined at this point)

At MLOOP, register B is loaded from ACCUM(C). Since C contains 0, this address expression specifies ACCUM+0, a word that contains 0. The 0 in B is multiplied by FACT. This product is added to the contents of register A, another 0. This sum is divided by 10. The remainder, a 0 in register B, is stored in ACCUM+0. The quotient, another 0, is left in register A to be carried to the next step. The SKIPG instruction does not skip because the carry is 0. The CAMGE instruction compares the 0 in register C to the 2 in NDIG. The CAMGE doesn't skip, so the AOJA instruction is executed. The AOJA increments C to the next index value, 1, and jumps back to MLOOP. At MLOOP now for the second time we find data as follows:

ACCUM+3	ACCUM+2	ACCUM+1	ACCUM+0	NDIG	FACT	A	B	C
0	7	2	0	2	7	0	0	1

Since C now contains 1, the instruction at MLOOP loads B from ACCUM+1. The value loaded is 2. This is multiplied by the contents of FACT; the product, decimal 14, is added to the previous carry in A.

The result, 14, is divided by 10. The remainder in B, the number 4, is stored into ACCUM+1. The quotient in A, 1, is carried to the next step. Because the carry out is non–zero, the SKIPG instruction will skip to the AOJA. Register C is incremented again, and the program arrives at MLOOP for the third time, with its data areas appearing as

ACCUM+3	ACCUM+2	ACCUM+1	ACCUM+0	NDIG	FACT	A	B	C
0	7	4	0	2	7	1	4	2

It is interesting to note that registers A and B contain a representation of the number 14 which was the sum (also the product) from the previous step. Since C now contains 2, register B is loaded with the 7 from ACCUM+2. This is multiplied by 7, and added to the carry in register A. Register A now contains decimal 50, the product of 7×7 plus the carry. This 50 is divided by 10. The remainder in B is stored in ACCUM+2. The quotient, 5, is left in A to be carried to the next step. Once more, because the carry is non–zero, the SKIPG will skip to the AOJA; the AOJA increments C and jumps to MLOOP again. At MLOOP for the fourth time, the data areas now contain

ACCUM+3	ACCUM+2	ACCUM+1	ACCUM+0	NDIG	FACT	A	B	C
0	0	4	0	2	7	5	0	3

The zero at ACCUM+3 is fetched and multiplied by the contents of FACT. This product, zero, is added to the carry in register A. The result, 5, is then divided by 10. The remainder, 5, is stored in ACCUM+3. The quotient, zero, represents the carry to the next step. However, as we shall see, there is no next step. The SKIPG instruction will not skip, since the carry out of this step is zero. Then, the CAMGE instruction is executed. The CAMGE compares the number in register C, 3, to the contents of NDIG. Because register C is now larger than NDIG, the CAMGE skips, avoiding the AOJA. Essentially, the meaning of this end test is that there are no more significant digits to process in the array ACCUM. Register C now contains the index to the most significant digit; that value is stored in NDIG, and the MULT routine returns to its caller. The result in memory looks like this:

ACCUM+3	ACCUM+2	ACCUM+1	ACCUM+0	NDIG	FACT	A	B	C
5	0	4	0	3	7	0	5	3

The control structure for the loop at MLOOP is somewhat more complicated than any that we have seen thus far. Instead of being just a counter, the control is accomplished by the logical OR of two conditions. If either a non–zero carry is present, or if not enough steps have yet been taken, the loop is repeated. You ought to be able to convince yourself that this method insures that precisely the right number of multiplication steps take place each time.

We come now to the PRINT routine. We are fortunate that PRINT is both simple and simply explained. In the representation of data that we use, each word in the ACCUM array holds exactly one digit of the result. We know that NDIG is the index number of the most significant digit. We know also that 0 is the index number of the least significant digit. The Pascal loop,

```
FOR i := ndig DOWNTO 0 DO ...
```

is the simple model for the printing routine. The interesting part of the print routine appears below:

```
;Print current factorial value.  NDIG is the index to most significant digit

        MOVE    D,NDIG          ;index to most significant digit
TLOOP:  MOVE    A,ACCUM(D)      ;get a digit, MSD through LSD
        ADDI    A,"0"           ;convert number to an integer
        IDPB    A,Z             ;store it in the output
        SOJGE   D,TLOOP         ;run down through digit at ACCUM+0
```

Register `D` is initialized to the value contained in `NDIG`. Each array element is processed by fetching it, adding the value of the ASCII character "0" to it, and depositing the resulting character into the output line buffer, via the `IDPB` instruction. The `SOJGE` instruction decrements register `D`, and loops to `TLOOP` while `D` remains non–negative. The `SOJGE` provides for `TLOOP` being executed when `D` contains zero, to print the least significant digit.

The entire program for calculating and printing factorials appears below. We hope this explanation has been sufficient so there will be no parts of the program that you find mysterious. There are some small sections that we haven't discussed here. Routines such as `DECFIL` and `COPYST` have appeared in earlier examples. The particular versions of these routines differ slightly from the previous examples; the structure of these routines is similar to what we have seen before.

```
        TITLE   Factorials up to 100!  Example 11

        SEARCH  MONSYM,MACSYM           ;Add the TOPS-20 JSYS definitions

A=1
B=2
C=3
D=4
W=5
X=6
Y=7
Z=10
P=17

PDLEN==100

        .PSECT  DATA,1001000
PDLIST: BLOCK   PDLEN
OBUFR:  BLOCK   ^D50                    ;output text buffer

;the next three lines must be kept together because they are zeroed by a BLT
ACCUM:  BLOCK   ^D200                   ;Room for 200 digits
NDIG:   0                               ;index number of most significant digit
FACT:   0                               ;how far we've gotten
        .ENDPS
```

```
        .PSECT  CODE/RONLY,1002000

START:  RESET                           ;initialize I/O
        MOVE    P,[IOWD PDLEN,PDLIST]   ;initial stack
        HRROI   A,[ASCIZ/Factorials up to 100
/]
        PSOUT                           ;Send Heading

;Initialization

        SETZM   ACCUM                   ;zero the array of digits
        MOVE    A,[ACCUM,,ACCUM+1]
        BLT     A,FACT                  ;200 digits, ACCUM thru ACCUM+199
                                        ;also, zero NDIG and FACT
        AOS     ACCUM                   ;start with 0! = 1.  ACCUM+0 set to 1.

;Print Current FACT and corresponding Factorial value
FLOOP:  CALL    PRINT                   ;Print the current value of FACT and
                                        ;the corresponding factorial.
;Increment FACT.  Test to see if we're done, if not, go make next factorial
        AOS     Y,FACT                  ;increment FACT
        CAILE   Y,^D100                 ;reached last value yet?
         JRST   DONE                    ;yes. go finish up
        CALL    MULT                    ;perform the multiplication
        JRST    FLOOP                   ;print result, etc.

DONE:   HRROI   A,[ASCIZ/

Done!
/]
        PSOUT
        HALTF                           ;Stop here.
        JRST    START                   ;In case of Continue command

        SUBTTL  Multiplication

;Multiply existing factorial value by the number that's in FACT.

MULT:   SETZB   A,C                     ;carry out of previous digit is zero
                                        ;C is the index to the current digit
                                        ;Start at least significant digit
MLOOP:  MOVE    B,ACCUM(C)              ;get one digit from the ACCUM array
        IMUL    B,FACT                  ;mult by the current FACT value
        ADD     A,B                     ;add the product to the previous carry
        IDIVI   A,^D10                  ;divide to get this digit and carry
        MOVEM   B,ACCUM(C)              ;store one digit.  Carry in A.
        SKIPG   A                       ;Skip if any carry is present
         CAMGE  C,NDIG                  ;no carry. are we at end?
          AOJA  C,MLOOP                 ;must go on.  advance C to next digit
        MOVEM   C,NDIG                  ;End. Store the new value of NDIG
        RET
```

```
        SUBTTL  Print the value of FACT and the corresponding factorial

PRINT:  MOVE    Z,[POINT 7,OBUFR]       ;pointer to output line
        MOVE    W,FACT                  ;the number to print
        MOVEI   Y,3                     ;number of digits
        CALL    DECFIL                  ;decimal output with blank fill.
        MOVE    Y,[POINT 7,[ASCIZ/! = /]]
        CALL    COPYST                  ;Copy string thru Z
;Print current factorial value.  NDIG is the index to most significant digit
        MOVE    D,NDIG                  ;index to most significant digit
TLOOP:  MOVE    A,ACCUM(D)              ;get a digit, MSD through LSD
        ADDI    A,"0"                   ;convert number to an integer
        IDPB    A,Z                     ;store it in the output
        SOJGE   D,TLOOP                 ;run down through digit at ACCUM+0
;Print end of line.
        MOVE    Y,[POINT 7,CRLF]        ;add CRLF to end of line
        CALL    COPYST
        MOVEI   A,0                     ;end line with a null for ASCIZ
        IDPB    A,Z
        HRROI   A,OBUFR                 ;send output line
        PSOUT
        RET

        SUBTTL  Other useful subroutines.

;Decimal output with fill.
;Call with W, the number to print, Y the number of columns to print
;       and A, the fill character.
DECFIL: MOVEI   A," "                   ;Fill with spaces
DECOUT: IDIVI   W,^D10                  ;divide to compute remainder digit
        SUBI    Y,1                     ;decrement number of fill spaces
        PUSH    P,X                     ;store remainder digit
        JUMPE   W,DECF1                 ;are we done with divides?
        CALL    DECOUT                  ;no.  divide some more
DECF0:  POP     P,A                     ;get a remainder digit
        ADDI    A,"0"                   ;convert to characters
        IDPB    A,Z                     ;stuff it in the buffer
CPOPJ:  RET

;Here to perform the leading fill.
DECF1:  SOJL    Y,DECF0                 ;Decrease Y, jump if all filled
        IDPB    A,Z                     ;Stuff a fill character
        JRST    DECF1                   ;loop until filled.

;copy string.  Source in Y, destination in Z.
COPYST: ILDB    A,Y                     ;Copy a string from Y to Z
        JUMPE   A,CPOPJ                 ;get a byte from Y, exit if zero
        IDPB    A,Z                     ;stuff thru Z and loop.
        JRST    COPYST

CRLF:   BYTE(7)15,12

        END     START
```

21.1.2 .ENDPS Pseudo–Op

We introduce the `.ENDPS` pseudo–op that we use to mark the end (or the end for now) of a psect. MACRO maintains a stack of psect "activations". Each .PSECT pushes the named psect onto the stack and makes it active. `.ENDPS` is the directive to cause MACRO to pop the stack. In this case MACRO pops to the unnamed psect.

21.1.3 Exercises

These exercises are intended to provide the student with an opportunity to experiment with one–dimensional arrays.

21.1.3.1 Compute Pascal's Triangle

The structure depicted below is called Pascal's triangle. It displays a table of binomial coefficients, the coefficients of the expansion of $(X + 1)^n$.

```
                    1
                  1   1
                1   2   1
              1   3   3   1
            1   4   6   4   1
          1   5  10  10   5   1
        1   6  15  20  15   6   1
      1   7  21  35  35  21   7   1
```

In this triangle, each number is the sum of the two numbers immediately above it.

Write a program to compute the first eleven (decimal) rows of Pascal's triangle. Format your output so that it is similar to what is shown above. Note that numbers are right–justified into columns; odd and even rows are offset to create a pleasant appearance. You will have to leave more space between the numbers than was provided in the portion depicted.

Output the result both to a file and to the terminal.

21.1.3.2 Compute e, the Base of Natural Logarithms

Compute an approximation to e, the base of natural logarithms, by evaluating the first 101 terms of the series:

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}$$

For this problem, let k take on the values from 0 to (decimal) 100.

Produce an output file, `ESUM.OUT`, that displays the value $1/k!$ for each value of k and the sum that approximates e for each value of k. Display exactly 120 digits after the decimal point. Compute the result using a total of 200 digits; one digit to the left, and 199 digits to the right of the decimal point.

Assuming that at some earlier point you have computed the fraction $1/(k-1)!$, then to compute $1/k!$, simply divide the previously obtained value by k. Remember that 0! is defined to be 1. So $1/0! = 1$ gives you a starting point.

Some examples:

```
1/0! = 1.000000000              (a given, initial value)
1/1! = (1/0!)/1 = 1.000000      (probably you have to compute this)
1/2! = (1/1!)/2 = 0.500000
1/3! = (1/2!)/3 = 0.166666
```

Given that you have computed 1/3! you can compute 1/4! by long division, as you were taught in grammar school.

```
1/4! = (1/3!)/4

    0.0414444
  4)0.1666666
   -0.0
    0.16
   -0.16
    0.006
      -4
       26
      -24
         26
        -24
           2 etc.
```

Long division is accomplished as follows:

- For each digit of the dividend produce one quotient digit. The quotient digit is simply the quotient of the (dividend digit plus any remainder from the previous digit) divided by the divisor.
- Any remainder from this division is multiplied by 10 and carried along to be added to the next digit of the dividend.

After each quotient is formed, add it to the sum that you are making. Your output should look like the following, only to greater precision:

```
1/  0! = 1.0000000
Sum    = 1.0000000
1/  1! = 1.0000000
Sum    = 2.0000000
1/  2! = 0.5000000
Sum    = 2.5000000
1/  3! = 0.1666666
Sum    = 2.6666666
1/  4! = 0.0416666
Sum    = 2.7083332
 ....
1/100! = 0.0000000
Sum    = 2.7182818
```

You may find it helpful to review the examples of file output and example 11 in which factorials were computed.

21.2 Two–Dimensional Arrays

Frequently the one–dimensional array is not an adequate tool for the computations that we find necessary. Two–dimensional arrays have important applications in scientific, engineering, and commercial calculations. Since the computer contains a one–dimensional memory, we have to use more complicated programming techniques than simple indexed addressing to create a two–dimensional structure.

A two–dimensional array is a rectangular structure in which each element is identified by a *row* number and a *column* number. An `M` × `N` array, called `Q`, defined by `Q: ARRAY [1..M,1..N]` conventionally appears as

```
Q[1,1] Q[1,2] Q[1,3] ... Q[1,N]
Q[2,1] Q[2,2] Q[2,3] ... Q[2,N]

...

Q[M,1] Q[M,2] Q[M,3] ... Q[M,N]
```

The lines across the page are called *rows*; the vertical lines are called *columns*.

When an array is stored in sequential memory locations, it is customary to choose either *row major form* or *column major form* to map array elements into the one–dimensional memory. In row major form the elements of each row are stored in consecutive locations; elements of row 2 appear after all the elements of row 1.

```
Q[1,1] Q[1,2] ... Q[1,N] Q[2,1] Q[2,2] ... Q[2,N] ...
         Q[M,1] Q[M,2] ... Q[M,N]
```

Row major form is seen in Pascal implementations. In column major form, the standard for Fortran, each column is stored in consecutive locations:

```
Q[1,1] Q[2,1] ... Q[M,1] Q[1,2] Q[2,2] ... Q[M,2] ...
         Q[1,N] Q[2,N] ... Q[M,N]
```

There are two basic ways to deal with a two–dimensional array. You may use indirect addressing via a structure called a *side–table* or you may calculate *address polynomials*. Both of these techniques are useful; the address polynomial technique is seen more frequently.

21.2.1 Array Addressing via Side–Tables

Suppose we want to store an `M` × `N` array, called `Q`, defined by `Q: ARRAY [1..M,1..N]`. In the program we can think of this as `M` one–dimensional arrays, `Q1`, `Q2`, `... QM`. Each of these one–dimensional arrays holds an entire row of the original array `Q`. The row index, the first index in a subscript pair, selects the appropriate one–dimensional array. The second index, the column number, selects one element from within the one–dimensional array. Thus, a reference to `Q[5,7]` would be handled by the program as a reference to `Q5[7]`.

The side–table is the data structure we use to translate from a specific row index (the first index) to the address of a specific row array, one of `Q1`, `Q2`, etc. The side–table will hold the address of the origin of each row.

Let us be specific. Suppose we want to access an array defined by `Q: ARRAY [1..8,1..20]`. First we must allocate the space for the eight one–dimensional row–arrays that will hold `Q`. A simple repetition of `BLOCK` pseudo–ops will suffice:

```
;Eight one-dimensional arrays, each one is a row of Q: ARRAY [1..8,1..20]
Q1:     BLOCK ^D20              ;space for row 1
Q2:     BLOCK ^D20              ;space for row 2
Q3:     BLOCK ^D20              ;space for row 3
Q4:     BLOCK ^D20              ;space for row 4
Q5:     BLOCK ^D20              ;space for row 5
Q6:     BLOCK ^D20              ;space for row 6
Q7:     BLOCK ^D20              ;space for row 7
Q8:     BLOCK ^D20              ;space for row 8
```

Next, we can build a side–table that we will call `QX`. Each of the eight entries is one of the row origin addresses. That is, consecutive entries in the `QX` table will contain the addresses of `Q1[1]`, `Q2[1]`, through `Q8[1]`.[2]

[2]If `Q1` ... `Q8` are in the same psect as `QX` and in the same assembly, this code produces 18–bit values in `QX`. If they are in different psects or in different assemblies, the assembler and loader will supply 30–bit address in the `QX` table.

```
QX:     Q1                  ;address of row 1, Q1[1]. At QX+0
        Q2                  ;address of row 2, Q2[1]. At QX+1
        Q3                  ;address of row 3, Q3[1]. At QX+2
        Q4                  ;address of row 4, Q4[1]. At QX+3
        Q5                  ;address of row 5, Q5[1]. At QX+4
        Q6                  ;address of row 6, Q6[1]. At QX+5
        Q7                  ;address of row 7, Q7[1]. At QX+6
        Q8                  ;address of row 8, Q8[1]. At QX+7
```

Now, suppose the first index (the row number) is in register `X`, and that the second index (the column number) is in register `Y`. Then, to access the specified array element, we perform the following steps:

```
MOVE    A,QX-1(X)            ;specific row origin address
ADD     A,Y                  ;add the column number
MOVE    B,-1(A)              ;access the specified array element.
```

The first instruction loads the address of a specific row array into register `A`. The expression `QX-1` appears in this instruction to compensate for the word at `QX+k` addressing row `k+1`. In another view, the word at address `QX-1+k` contains the base address of row `k`. Thus, given a specific row number, `k`, contained in register `X`, the address expression `QX-1(X)` addresses the word containing the origin of row `k`. This instruction will load register `A` with the address of the origin of the row whose number is in `X`.

Adding the index value contained in `Y` to the address contained in `A` gives nearly the correct address of the desired element. Again, we are off by one. To be specific, suppose `X` contains 5 and `Y` contains 3. Then the first instruction above has loaded register `A` with the data found at `QX+4`; that data is the address of `Q5`. Adding the contents of register `Y` to this produces in `A` the address `Q5+3`. Unfortunately, since `Q5+0` addresses `Q[5,1]`, the address `Q5+3` must correspond to `Q[5,4]`. But, we are very close. All we have to do is subtract one from the contents of `A` and we have the right address. Rather than do this subtraction explicitly, we incorporate an offset, `-1`, in the next instruction, the one that actually loads register `B` with the desired data item.

This process can be refined in several ways. We can modify the structure of the side–table to take advantage of the PDP–10's effective address calculation. We shall use both indexing and indirection to perform the array addressing calculation; this is an example where indirect addressing has no reasonable substitute. We redefine `QX` by making two changes. First, we adjust each address by one to account for the offset in the column addresses. (If the first column number were zero instead of one, this offset wouldn't be necessary.). Our second change to `QX` is to include register `Y` in the index field of each word in `QX`. We shall see what difference these changes accomplish below.

For extended addressing, we must know whether `Q1` and `QX` are in the local address section of the program. For this example, we they are. Include the definition `OPDEF IFIW [1B0]` somewhere near the top of the program. Using it forces bit 0 to be 1, that is, it creates an instruction format indirect word.[3]

[3]Some programmers have noticed that `SETZ` is equal to `1B0`; so they use `SETZ` for this purpose.

```
QX:     IFIW Q1-1(Y)    ;IFIW with Y,,address of Q[1,0].  At QX+0
        IFIW Q2-1(Y)    ;IFIW with Y,,address of Q[2,0].  At QX+1
        IFIW Q3-1(Y)    ;IFIW with Y,,address of Q[3,0].  At QX+2
        IFIW Q4-1(Y)    ;IFIW with Y,,address of Q[4,0].  At QX+3
        IFIW Q5-1(Y)    ;IFIW with Y,,address of Q[5,0].  At QX+4
        IFIW Q6-1(Y)    ;IFIW with Y,,address of Q[6,0].  At QX+5
        IFIW Q7-1(Y)    ;IFIW with Y,,address of Q[7,0].  At QX+6
        IFIW Q8-1(Y)    ;IFIW Y,,address of Q[8,0].  At QX+7
```

Now, if `X` contains the row number, and `Y` contains the column number, an array element can be accessed by a single instruction:

```
MOVE    B,@QX-1(X)
```

Let us examine the action of this instruction carefully. Recall how effective address calculations are performed. Suppose that `X` contains 5 and `Y` contains 3; we are trying to reference `Q[5,3]`. Remember that we have transformed `Q[5,3]` to `Q5[3]`, which is stored at address `Q5+2`. The expression `QX-1(X)` is familiar from the previous description; this evaluates to `QX+4`. Because indirect addressing is specified by this instruction, the local format address word at `QX+4` is fetched, and the effective address calculation is continued, using the address fields of that word.

The local format word at `QX+4` contains the address `Q5-1` in the right half; it also contains the address of register `Y` in the index field. The contents of register `Y` are added to the address `Q5-1`. In this case the result is `Q5+2`, as we desired.

The use of side–tables can produce very fast–running programs. Side tables allow certain operations, such as the permutation of pairs of rows, to be accomplished very quickly. Other data structures, such as ragged arrays in which the rows have different lengths, can be implemented with this approach.

The main disadvantage of side–tables is that they use some extra space. A second requirement, in some cases a disadvantage, is that when indirect addressing techniques are used, the same index register must be available at all places where the array is accessed.

We have described an example of row major form; similar techniques can be applied to storage in column major form.

In the example given above, we assumed that `QX` and `Q1` were local to the program's address section. We briefly examine an alternative: `Q1` ... `Q8` are in a different address section, but `QX` is local to the program's address section. In this case we need a global format indirect word for each entry in `QX`. The first entry would look like this:

```
QX:     <BYTE(1)0,0(4)Y>+Q1-1 ;GIW with index Y and address of Q[1,0].  At QX+0
```

In this we have assumed that `Q1` is defined in some other psect.

MACRO, being very much a creature of the un–extended addressing era, has a strong desire to store addresses in halfwords. MACRO would like to use 18 bits for an address expression such as `Q1-1` appearing in `<BYTE(1)0,0(4)Y>+Q1-1`. However, if `Q1` is external to the current module or if it is in a different psect, MACRO will direct LINK to use a full word for the address expression.

We did not write `Q1-1` in the `BYTE` pseudo–op even though, logically, it belongs there. MACRO and LINK can store into left– or right–halfwords or into full words; storing into other subfields of a word

is beyond the scope of the language in which "fixups" are defined.

Before leaving the subject of side–tables, we should mention that we can make MACRO do most of the work necessary to define the side–tables. We observe that the each of the labels Q2, Q3, etc. differs from its predecessor by precisely decimal 20. We can take advantage of this constant offset by using the REPEAT macro operator to expand a code fragment several times:

```
Q:      BLOCK    ^D20*^D8       ;room for 8 x 20 items, Q[1..8,1..20]

QX:                             ;Base of QX, the side table
IIII==0                         ;A temporary variable
REPEAT ^D8,<                    ;repeat the following 8 times
        IFIW Q+IIII-1(Y)        ;analog of IFIW Qk-1(Y)
        IIII==IIII+^D20         ;advance to next row
>                               ;end of repeated material
```

Room for the entire array is reserved by the BLOCK pseudo–op at the label Q. The side–table is defined following the label QX. A temporary variable called IIII is initialized to have the value 0. This variable will be increased by decimal 20 after each entry is made in the side table.

The REPEAT operator causes two lines to be repeated. One line will make an entry into the side–table; the other line advances the variable IIII. Each entry in the side table differs from its predecessor by decimal 20. The REPEAT operator performs this function eight times, building the entire side–table.

21.2.2 Address Polynomials

We have already noted that the address contained in QX+1 is decimal 20 larger than the address in QX. Similarly, the address in QX+2 was 20 larger than the address in QX+1. The number 20 in this case is just the number of columns. Simple arithmetic can replace the entire side table, since the contents of QX+k are precisely 20×k larger than the contents of QX.

Therefore, if Q0 is the address of Q[1,1], the address of Q[i, j] is given by the formula

$$\texttt{Q0} + (i - 1) \times 20 + (j - 1)$$

More generally, if we have an array defined by S: ARRAY [K..L,M..N], we can define an accessing system that will be suitable. Let i be the row index, and j be the column index. A valid access to S[i, j] must satisfy the following constraints on i and j:

$$(\texttt{K} \leq i \leq \texttt{L}) \wedge (\texttt{M} \leq j \leq \texttt{N})$$

Let S0 be the address of S[K,M]. Then the address of S[i, j] is given by the expression

$$\texttt{S0} + (i - \texttt{K}) \times (\texttt{N} - \texttt{M} + 1) + (j - \texttt{M})$$

The expressions $(i - \texttt{K})$ and $(j - \texttt{M})$ represent *normalized* or *effective* subscripts. By subtracting the lower row bound from i we have made a number, $(i - \texttt{K})$, that ranges from 0 to L-K; the normalized row number is multiplied by the number of words per column (given by $\texttt{N} - \texttt{M} + 1$) to produce the offset to address the selected row. Finally, the normalized column subscript is added to address the specific word.

The expression

$$(i - \texttt{K}) \times (\texttt{N} - \texttt{M} + 1) + (j - \texttt{M})$$

is called the address polynomial. The generalization of this expression to a true polynomial must await our discussion of multi–dimensional arrays in Section 21.3, page 334.

There are some important optimizations to consider when writing a program that accesses an array by an address polynomial. Several constant terms appear in the expression given for addressing a specific element of the array. These constants should be computed once, when the array is allocated, and saved for accessing the array later. We rewrite the expression given above for the address of S[i, j], to collect constants:

S0 − K × (N − M + 1) − M + i × (N − M + 1) + j

The expression

S0 − K × (N − M + 1) − M

represents S_{00}, the address where S[0,0] is (or would be) allocated. There need not be a [0,0] element present. Also, the expression (N − M + 1) is simply L_C, the number of columns. For the sake of rapid access to the array during the running of the program, both of these expressions should be computed when the array is allocated. Then the polynomial to address S[i, j] becomes

$S_{00} + i \times L_C + j$

Using these formulas, it is possible to write simple code sequences to access the array:

```
MOVE    1,I         ;row index
CAML    1,K         ;perform array index boundary checks
 CAMLE  1,L
  CALL  AIDXER      ;array index error
MOVE    2,J         ;column index
CAML    2,M         ;bounds check
 CAMLE  2,N
  CALL  AIDXER      ;an indexing error
IMUL    1,LC        ;row index times the number of columns
ADD     1,2         ;row * number of columns + column number
MOVE    3,S00(1)    ;read an array element
```

If you are certain that no indexing errors can occur, the sequence above can be abbreviated:

```
MOVE    1,I         ;row index
IMUL    1,LC        ;row index times the number of columns
ADD     1,J         ;row * number of columns + column number
MOVE    3,S00(1)    ;read an array element
```

The advantage of address polynomials is that they are readily generalized to multi–dimensional arrays. A programmer should be conscious, however, that the multiplications required by this method are costly. In many cases, the use of indirect addressing through a side–table would result in a faster running program.

The next example shows the use of an address polynomial to access a two–dimensional array that represents a piece of paper on which we draw several figures.

21.2.3 Example 12 — Plot Program

We are going to build a program that uses a two–dimensional array to represent a piece of paper. We will write into the array to "color" areas of the paper. When we finish, we print the array onto a real piece of paper, set the array to zero, signifying blankness, and draw another picture.

This program is quite crude in a number of respects. Again, this is not intended as an example of the very best way to do this function. Rather, it is an illustration of techniques that have many useful applications. To refine the program would require the addition of further materials that would confuse, rather than clarify, the issue of array access.

One side issue has been allowed to creep in: we demonstrate how to access the Fortran library of useful mathematical subroutines. In particular we shall use the `SIN` function for the computation of sines and cosines.

21.2.3.1 Defining the Array

We begin by writing the necessary definitions for the array into which we will be plotting. For plotting, it is natural to think of the array as representing a portion of the X–Y coordinate space. Conventionally, increasing X moves towards the right; increasing Y moves towards the top of the page. The array will represent a rectangular region of the X–Y plane, centered at (0,0). The following definitions appear in the program:

```
;The following parameters describe XY: ARRAY [XMIN..XMAX,YMIN..YMAX]

XMAX==  ^D40            ;maximum X value
XMIN==  -XMAX           ;minimum X value
XSIZE== XMAX-XMIN+1     ;total number of allowable X values
XMUL==  40.0            ;the multiplier to spread (-1.0 to +1.0)
                        ;normalized data across the width of the array

YMAX==  ^D20            ;maximum Y value
YMIN==  -YMAX           ;minimum Y value
YSIZE== YMAX-YMIN+1     ;number of allowable Y values
YMUL==  20.0            ;the multiplier to spread normalized data
                        ;  through the entire height of the array.

ARYSIZ== XSIZE*YSIZE    ;total size of the array

XY:     BLOCK   ARYSIZ  ;Allocate space for the array.
```

The selection of the particular sizes of the array are governed in this case by two considerations. First, the "paper" must be filled as full as possible. Second, a *square aspect ratio* must be established. Each character on the paper is printed in a rectangular box. On line printers, usually six characters appear in a vertical inch, and ten appear in a horizontal inch. To write a square, the ratio of six vertical characters to ten horizontal characters must be maintained. Different display terminals have different aspect ratios. It happens that the typeface we use for figures in this book has nearly a 1:2 vertical to horizontal ratio. So, in the array we have built there are two positions in the X–direction for each one in the Y–direction.

21.2.3.2 Accessing the Array

From our discussion of address polynomials for accessing two–dimensional arrays, we have the following formula for accessing a particular element of `XY`.[4] The address of `XY[x,y]` is given by:

```
XY + (y-YMIN)*XSIZE + x-XMIN
```

Next, we collect the constants:

```
XY - YMIN*XSIZE - XMIN + y*XSIZE + x
```

Since the constant, `XY-YMIN*XSIZE-XMIN` will occur frequently, we name it `XYORG`, the virtual origin of the `XY` array:

```
XYORG== XY-YMIN*XSIZE-XMIN
```

Now, we can rewrite our access formula. The address of `XY[x,y]` is given by

```
XYORG + y*XSIZE + x
```

In this program, plotting is accomplished by generating a sequence of coordinate pairs. For each pair, a subroutine named `SETXY` is called to "paint" the specified point. Each coordinate is a floating–point number within the range from `-1.0` to `+1.0`.

In `SETXY`, the range of each coordinate must be expanded into the full width or height of the array. The constant `XMUL` that was defined above is the floating–point number corresponding to the maximum X–coordinate in the array. By multiplying the given X–coordinate by `XMUL` we transform it into a floating–point number in the range from `-XMAX` to `+XMAX`. Of course, the resulting number is still floating–point, so it must be made an integer. We use the `FIXR` instruction for this purpose. A similar transformation is done to the Y–coordinate. The resulting expanded integer values for X and Y (in registers `X` and `Y`) are treated by the formula we developed above to transform them into an address in the `XY` array. Finally, the "paint" that we are using to make our picture is deposited into the array.

[4] In matrix manipulation, it is customary to consider the first subscript a row number. In the X–Y coordinate system, it is conventional to write the X–value first, which is a column number. Because of this interchange of the position of the row and column subscripts, the formula that is given looks like column major form. Actually, it is row major form, where a constant Y–value determines one row.

```
;SETXY, deposit a paint color into the array XY.
;       Call with:
;       X/      X-coordinate, between -1.0 and 1.0
;       Y/      Y-coordinate, between -1.0 and 1.0
;       Z/      Color of paint, a right-adjusted ASCII character
;
;       X and Y are changed by this routine.

SETXY:  FMPR    X,[XMUL]          ;expand the width of X coordinate
        FMPR    Y,[YMUL]          ;and the height of the Y coordinate
        FIXR    X,X               ;convert both to integer
        FIXR    Y,Y
        IMULI   Y,XSIZE           ;Y*XSIZE
        ADD     Y,X               ;Y*XSIZE + X
        MOVEM   Z,XYORG(Y)        ;deposit the paint
        RET
```

21.2.3.3 Plotting Figures

This program will plot a circle and a Lissajous figure. Both of these require that sines and cosines of angles be computed. To draw a circle, we generate a series of coordinate pairs and plot them. This form of plotting makes use of parametric equations that describe a curve as if plotting positions of a point as a function of time. The circle is drawn by a program analogous to the Pascal fragment:

```
FOR i := 0 TO stabln-1 DO
 BEGIN
 x := COS(2 * 3.14159265 * i / stabln);
 y := SIN(2 * 3.14159265 * i / stabln);
 setxy(x,y)
 END
```

The Lissajous figure is draw by a similar fragment:

```
FOR i := 0 TO stabln-1 DO
  BEGIN
  x := COS(3 * 2 * 3.14159265 * i / stabln);
  y := SIN(4 * 2 * 3.14159265 * i / stabln);
  setxy(x,y)
  END
```

In both of these fragments, the symbol `STABLN` represents the *resolution*, the fineness, of the step size. For larger values of `STABLN`, a larger number of smaller steps are taken to complete the figure. The iteration variable, `i`, represents sequential time intervals in the parametric equations.

Instead of calling the `SIN` or `COS` function for each step that we take, we will build two tables. `SINTAB` will be an array that contains `STABLN` elements. `SINTAB[I]` will be the value of $\sin(2 \times \pi \times \mathtt{I}/\mathtt{STABLN})$ for values of `I` in the range $0 \leq \mathtt{I} < \mathtt{STABLN}$. The table `COSTAB` is similarly defined. Then, our programs for the circle and Lissajous figure become

```
(* Circle *)
FOR i := 0 TO stabln-1 DO
  BEGIN
  x := costab[i];
  y := sintab[i];
  setxy(x,y)
  END
```

```
(* Lissajous Figure *)
FOR i := 0 TO stabln-1 DO
  BEGIN
  x := costab[(4 * i) MOD stabln];
  y := sintab[(3 * i) MOD stabln];
  setxy(x,y)
  END
```

These fragments are readily translated into assembly language. The program for drawing a circle is very simple:

```
CIRCLE: MOVEI   Z,"o"           ;select the "color" of paint
        MOVSI   W,-STABLN       ;set up AOBJN pointer for STABLN steps
CIRL:   MOVE    X,COSTAB(W)     ;Set up X
        MOVE    Y,SINTAB(W)     ;Set up Y
        CALL    SETXY           ;Deposit "paint" in the array
        AOBJN   W,CIRL          ;Loop through all values 0 to STABLN-1
        CALL    PUTARY          ;Write the resulting array.
        RET
```

To make the MOD function efficient, we select a power of 2 for the value of STABLN. Then the value STABLN-1 is a mask that can be ANDed with any value to produce the desired residue. The program for drawing the Lissajous figure is then

```
LISAJ:  MOVEI   Z,"#"         ;select the color of paint
        MOVSI   W,-STABLN     ;prepare for STABLN steps
LISAJ1: HRRZ    X,W           ;the index value, "i"
        HRRZ    Y,W           ;the index value, "i"
        IMULI   X,3           ;3*i
        IMULI   Y,4           ;4*i
        ANDI    X,STABLN-1    ;(3*i) mod STABLN;  assumes STABLN is
        ANDI    Y,STABLN-1    ;(4*i) mod STABLN;    a power of two
        MOVE    X,COSTAB(X)   ;x := costab[(3*i) MOD stabln]
        MOVE    Y,SINTAB(Y)   ;y := sintab[(4*i) MOD stabln]
        CALL    SETXY         ;color the array
        AOBJN   W,LISAJ1      ;loop through all values, 0 to STABLN-1
        CALL    PUTARY        ;write the results.
        RET
```

21.2.3.4 Constructing SINTAB and COSTAB

The SINTAB table is constructed by this simple loop:

```
FOR i := 0 TO stabln-1 DO
        sintab[i] := SIN(2 * 3.14159265 * i / stabln)
```

In assembly language, this becomes a fairly straightforward loop. We begin by using an AOBJN to run the index upwards from zero to STABLN-1:

```
INIT:   MOVSI   W,-STABLN               ;build the sine table
INITL:  . . .
        AOBJN   W,INITL
        . . .
```

The right half of register W will count upwards from 0 to STABLN-1. However, since we need a floating–point number, we must copy this value and float it. We add at INITL:

```
INITL:  HRRZ    A,W                     ;copy the integer index to A
        FLTR    A,A                     ;float it
        FMPR    A,[6.2831853]           ;multiply by 2 PI
```

Since STABLN has been chosen to be a power of two, the division by STABLN can be most rapidly accomplished by the FSC instruction. Division of a floating–point number by a power of two is equivalent to a subtraction from the exponent of the dividend. In order to tell what number to subtract from the exponent, we will define STABLN in terms of a new symbol called LOGSTL, the logarithm (base two) of the sine table's length.

The definition of STABLN in terms of LOGSTL is accomplished by using MACRO's shifting operator, the underscore character. LOGSTL is defined to be octal 11, decimal 9. STABLN is defined to be the value 1 shifted by LOGSTL places. The result in binary is a one followed by nine zeroes, octal 1000:

```
LOGSTL==11              ;LOGarithm of the Sine Table's Length
STABLN==1_LOGSTL        ;the Sine TABle's LeNgth.
```

Again the expression 1_LOGSTL means the value one shifted by LOGSTL bits. LOGSTL is the number by which the exponent must be reduced in order to effect the division by STABLN. We write the instruction

```
        FSC     A,-LOGSTL       ;Divide by STABLN
```

to perform this division.

The result in A is the value 2*PI*I/STABLN. This is the argument to the SIN function. We store this value in the location called SINARG. Now, we load register 16, whose symbolic name is AP (the *Argument Pointer*), with the address of the argument list. Then we call the Fortran Library's SIN routine. The SIN routine returns its result in register 0. That result is stored into the SINTAB array. The entire code segment for building SINTAB appears below:

```
INIT:   MOVSI   W,-STABLN               ;build the cosine/sine tables
INITL:  HRRZ    A,W                     ;the multiplier
        FLTR    A,A                     ;float it
        FMPR    A,[6.2831853]           ;2*PI*I
        FSC     A,-LOGSTL               ;divide by size of table
        MOVEM   A,SINARG                ;save arg to SIN routine
        MOVEI   AP,ARGLST
        CALL    SIN                     ;compute the sine
        MOVEM   0,SINTAB(W)             ;store it.
        AOBJN   W,INITL
```

The SIN function is obtained from the Fortran Library by two steps. First, by declaring the name SIN to be an external symbol, via the EXTERN statement, we tell MACRO and LINK to look for SIN outside of this program. Second, by including the .REQUEST statement, we cause MACRO to tell LINK to search the specified file. In this case, the requested file is the binary relocatable file that holds the Fortran library. These two statements are summarized below:

```
.REQUEST SYS:FORLIB       ;Ask LINK to look in SYS:FORLIB.REL
EXTERN SIN                ;for the Fortran SIN routine
```

In the DECSYSTEM–20 all Fortran functions and subroutines have a standardized way to pass arguments and results.[5] Before calling one of these functions, register 16 must be initialized to contain the address of the argument list. This is called the argument pointer. An argument list has a specific form. The word immediately preceding the word addressed by the argument pointer contains the count of how many arguments are present. The argument pointer then holds the address of the first argument descriptor.

In this simple case, the argument descriptor is a word in which the number 4 appears in the accumulator field, and in which the address of the argument appears in the right halfword. The accumulator field describes the argument type; the number 4 signifies a floating–point (i.e., Real) argument. The word will be used as an instruction format (local) indirect word, so we arrange that bit 0 is set by writing SETZ as the opcode; SETZ is operation code number 400, so this sets bit 0. The author doesn't necessarily recommend this, but it often is done.

The argument list for this program appears below:

```
        -1,,0
ARGLST: SETZ 4,SINARG       ;IFIW, Real value (AC = 4), Address
```

The word containing -1,,0 has the negative of the number of arguments in the left half. Even though the argument list is constant, we place it in the DATA psect because it points – with an 18–bit address – to a data item in that psect.

Finally, we turn our attention to the table of cosine values. We could repeat this loop to build and store an equivalent table of cosines. We could call the Fortran COS routine and store results in COSTAB. Instead of doing anything so simple, we take advantage of our knowledge of mathematics. From the identity

$\cos(x) = \sin(x + \pi/2)$

[5] For further information, consult the appendices to the DECSYSTEM–20 Fortran Reference Manual.

we know that we already have computed three–quarters of the cosine table. By overlapping the cosine table into the same space as the sine table, we observe the following relationships:

SINTAB + 0	$\sin(0)$		
. . .	. . .		
SINTAB + <STABLN/4>	$\sin(\pi/2)$	$\cos(0)$	COSTAB + 0
. . .	. . .		
SINTAB + <STABLN/2>	$\sin(\pi)$	$\cos(\pi/2)$	
. . .	. . .		
SINTAB + 3*<STABLN/4>	$\sin(3\pi/2)$	$\cos(\pi)$	
. . .	. . .		
SINTAB + STABLN -1	$\sin(2\pi - \varepsilon)$	$\cos(3\pi/2 - \varepsilon)$	

All that we need to do is copy the first quarter of the sine table to the end of the cosine table. We accomplish this operation by the following data definitions and program segment:

```
SINTAB: BLOCK   <5*STABLN>/4         ;25% more space for cosine table
COSTAB==SINTAB+<STABLN/4>            ;origin of the cosine table,
                                     ;corresponding to SIN(PI/2).

;copy the first quarter of SINTAB to the last quarter of COSTAB
        MOVE    A,[SINTAB,,SINTAB+STABLN]  ;copy start of SINE table
        BLT     A,COSTAB+STABLN-1          ;to end of COSINE table
```

This code fragment goes at the end of the initialization code, after the sine table is built. After this BLT instruction is executed, the sine and cosine table has been augmented:

SINTAB + 0	$\sin(0)$		
. . .	. . .		
SINTAB + <STABLN/4>	$\sin(\pi/2)$	$\cos(0)$	COSTAB + 0
. . .	. . .		
SINTAB + <STABLN/2>	$\sin(\pi)$	$\cos(\pi/2)$	
. . .	. . .		
SINTAB + 3*<STABLN/4>	$\sin(3\pi/2)$	$\cos(\pi)$	
. . .	. . .		
SINTAB + STABLN	$\sin(0)$	$\cos(3\pi/2)$	
. . .	. . .		
SINTAB + 5*<STABLN/4>-1	$\sin(\pi/2 - \varepsilon)$	$\cos(2\pi - \varepsilon)$	

21.2.3.5 Writing the Array to a File

The array is written to the file by the PUTARY subroutine. This routine starts at the maximum Y position (corresponding to the top of the paper) and scans each row of the array. Since the array is stored so that each row (a constant Y–value) appears in consecutive locations, an abbreviated form of address calculation is used.

Each word in the array contains either a character to print, or zero. For zero values, we print a blank. At the end of scanning each row, a new line is started by sending carriage return and line feed to the file. Characters are sent to the file by means of the BOUT, *Byte OUTput*, JSYS in the PUTCHR routine.[6]

[6]This is a reasonable opportunity to display the BOUT JSYS. However, if this program were intended for frequent and realistic usage, we would eschew BOUT because the overhead of multiple JSYS calls is too high. We would prefer SOUT or the PMAP JSYS that is described in Section 25.

```
PUTARY: MOVEI   W,YMAX          ;Outer loop index.  YMAX down to YMIN
PUTNTY: MOVE    Y,W             ;Here to start one row. (Y-value)
        IMULI   Y,XSIZE         ;compute row-origin
        ADD     Y,[XMIN]        ;offset to true row-origin
        MOVSI   X,-XSIZE        ;step column (X) thru all values.
PUTNTX: SKIPN   A,XYORG(Y)      ;skip if square was painted
         MOVEI  A," "           ;unpainted, use "blank" paint
        CALL    PUTCHR          ;store a character in the file
        ADDI    Y,1             ;advance to next column.
        AOBJN   X,PUTNTX        ;advance to the next column
        MOVEI   A,15            ;add carriage return
        CALL    PUTCHR
        MOVEI   A,12            ;and line feed, to the file.
        CALL    PUTCHR
        CAMLE   W,[YMIN]        ;have we finished the last row?
        SOJA    W,PUTNTY        ;No, decrement row number and loop
;Clear the array.
CLRARY: SETZM   XY              ;zero the array.
        MOVE    A,[XY,,XY+1]
        BLT     A,XY+ARYSIZ-1
        RET

;Put one character, in A, into the output file.
PUTCHR: PUSH    P,B             ;Save register B
        MOVE    B,A             ;copy character to B
        HRRZ    A,OJFN          ;JFN to register A
        BOUT                    ;Send one byte
         ERJMP  ERROR           ;in case of error
        POP     P,B             ;Restore original contents of B
        RET
```

21.2.3.6 The Completed Plot Program

The entire plotting program that we have discussed appears below. Among the portions that we have not talked about are the file output routines, error handling, and the `CLRARY` routine. All of these are similar to portions that have been discussed earlier.

```
        TITLE   Plot Program - Example 12

        SEARCH  MONSYM,MACSYM
        .REQUEST SYS:FORLIB             ;Ask LINK to look in SYS:FORLIB.REL
        EXTERN SIN                      ;for the Fortran SIN routine
        .TEXT "/REDIRECT:DATA:CODE"     ;put SIN in CODE, not in high segment
A=1                                     ;the usual definitions
B=2
C=3
D=4
W=5
X=6
Y=7
Z=10

AP=16                                   ;argument pointer, for FORLIB
P=17                                    ;this is the stack for FORLIB also

PDLEN==100
LOGSTL==11                              ;LOG (base 2) of the Sin Table's Length
STABLN==1_LOGSTL                        ;Sine TABle LeNgth.

        .PSECT  DATA,1001000
OJFN:   0
;argument list for the Fortran SIN routine (even though constant)
        -1,,0                   ;-number of arguments,,0
ARGLST: SETZ 4,SINARG           ;IFIW, Real (4 as the AC), address of argument
SINARG: 0                       ;one word to hold the argument to SIN
PDLIST: BLOCK   PDLEN

;Define the Sine/Cosine Table
SINTAB: BLOCK   <STABLN*5>/4            ;25% longer to make room for cosines
COSTAB==SINTAB+<STABLN/4>               ;Cosine table 1/4 shifted from sines.
```

```
;The following parameters describe XY: ARRAY [XMIN..XMAX,YMIN..YMAX]

XMAX==^D40              ;maximum X value
XMIN==-XMAX             ;minimum X value
XSIZE==XMAX-XMIN+1      ;total number of allowable X values
XMUL==40.0              ;the multiplier to spread normalized (-1 to 1) data
                        ;across the width of the array

YMAX==^D20              ;maximum Y value
YMIN==-YMAX             ;minimum Y value
YSIZE==YMAX-YMIN+1      ;number of allowable Y values
YMUL==20.0              ;the multiplier to spread normalized (-1 to 1) data
                        ;across the height of the array

ARYSIZ==XSIZE*YSIZE     ;total size of the array

XY:     BLOCK   ARYSIZ

;the formula for accessing the array is:
;       XY[x,y] is at XY+(y-YMIN)*XSIZE+x-XMIN
;
;       collecting constants:
;                       XY-YMIN*XSIZE-XMIN+y*XSIZE+x
;       rename the constant XY-YMIN*XSIZE-XMIN as XYORG

XYORG==XY-YMIN*XSIZE-XMIN

;now we have the accessing formula:
;       XY[X,Y] is at XYORG+Y*XSIZE+X

        .ENDPS



        .PSECT   CODE/RONLY,1012000

CRLF:   BYTE(7)15,12

;Main program
START:  RESET                           ;Always start this way
        MOVE    P,[IOWD PDLEN,PDLIST]   ;Initialize a stack
        CALL    INIT                    ;Build the sine/cosine tables
        CALL    SETFIL                  ;establish an output file
        CALL    CLRARY                  ;Clear the array
        CALL    CIRCLE                  ;draw a circle
        MOVEI   A,14                    ;start a new page
        CALL    PUTCHR
        CALL    LISAJ                   ;draw a Lissajous figure
        CALL    CLSFIL                  ;close the output file.
        HALTF                           ;done.
        JRST    START                   ;in case of a continue command
```

```
;Initialize the Sine and Cosine Tables
; FOR i := 0 TO stabln-1 DO sintab[i] := SIN(2 * 3.14159265 * i / stabln)
;
INIT:   MOVSI   C,-STABLN               ;build the cosine/sine tables
INITL:  HRRZ    A,C                     ;the multiplier
        FLTR    A,A                     ;float it
        FMPR    A,[6.2831853]           ;2*PI*I
        FSC     A,-LOGSTL               ;divide by size of table
        MOVEM   A,SINARG                ;save arg to SIN routine
        XMOVEI  AP,ARGLST
        CALL    SIN                     ;compute the sine
        MOVEM   0,SINTAB(C)             ;store it.
        AOBJN   C,INITL
;Copy the first quarter of the sine table to be the last quarter of
;the cosine table.
        MOVE    A,[SINTAB,,SINTAB+STABLN]       ;copy start of SINE table
        BLT     A,COSTAB+STABLN-1               ;to end of COSINE table
        RET

;Select a file for output.  Obtain a JFN, open the file for writing.
SETFIL: MOVX    A,<GJ%SHT!GJ%FOU>       ;short form, output file
        HRROI   B,[ASCIZ/PLOTS.OUT/]    ;pointer to file name
        GTJFN                           ;get a JFN (job file number)
         ERJMP  ERROR                   ;error
        MOVEM   A,OJFN                  ;remember JFN of output file
        HRRZ    A,A                     ;a good habit.  remove JFN flags
        MOVE    B,[070000,,OF%WR]       ;7-bit bytes
        OPENF                           ;open the file for output
         ERJMP  ERROR                   ;error
        RET

;Error handling
ERROR:  HRROI   A,[ASCIZ/Error: /]
        ESOUT                           ;start of error message.
        MOVEI   A,.PRIOU                ;output error message to terminal
        HRLOI   B,.FHSLF                ;this fork, most recent error
        MOVEI   C,0                     ;no limit to byte count
        ERSTR                           ;convert error number to a string
         JFCL                           ;would you believe TWO error returns?
         JFCL
        HRROI   A,CRLF                  ;end message with end of line
        PSOUT
        HALTF
        JRST    START
```

```
;Draw a circle.
;       FOR i := 0 TO stabln-1 DO SETXY(COSTAB[i],SINTAB[i])
;
CIRCLE: MOVEI   Z,"o"                   ;select the "color" of paint
        MOVSI   W,-STABLN               ;set up AOBJN pointer for stabln steps
CIRL:   MOVE    X,COSTAB(W)             ;Set up X
        MOVE    Y,SINTAB(W)             ;Set up Y
        CALL    SETXY                   ;Deposit "paint" in the array
        AOBJN   W,CIRL                  ;Loop through all values 0 to stabln-1
        CALL    PUTARY                  ;Write the resulting array.
        RET

;Draw a Lissajous Figure
;       FOR i := 0 TO stabln-1 DO
;               SETXY(COSTAB[(3*i) MOD stabln],SINTAB[(4*i) MOD stabln])
;
LISAJ:  MOVEI   Z,"#"                   ;select the color of paint
        MOVSI   W,-STABLN               ;prepare for stabln steps
LISAJ1: HRRZ    X,W                     ;the index value, "i"
        HRRZ    Y,W                     ;the index value, "i"
        IMULI   X,3                     ;3*i
        IMULI   Y,4                     ;4*i
        ANDI    X,STABLN-1              ;(3*i) MOD stabln; assumes STABLN is a
        ANDI    Y,STABLN-1              ;(4*i) MOD stabln;        power of two
        MOVE    X,COSTAB(X)             ;x := costab[(3*i) MOD stabln]
        MOVE    Y,SINTAB(Y)             ;y := sintab[(4*i) MOD stabln]
        CALL    SETXY                   ;color the array
        AOBJN   W,LISAJ1                ;loop through all values, 0 to stabkn-1
        CALL    PUTARY                  ;write the results.
        RET

;SETXY, deposit a paint color into the array XY.
;       Call with:
;       X/      X-coordinate, between -1.0 and 1.0
;       Y/      Y-coordinate, between -1.0 and 1.0
;       Z/      Color of paint, a right-adjusted ASCII character
;
;       X and Y are changed by this routine.
SETXY:  FMPR    X,[XMUL]                ;expand the width of X coordinate
        FMPR    Y,[YMUL]                ;and the height of the Y coordinate
        FIXR    X,X                     ;convert both to integer
        FIXR    Y,Y
        IMULI   Y,XSIZE                 ;Y*XSIZE
        ADD     Y,X                     ;Y*XSIZE + X
        MOVEM   Z,XYORG(Y)              ;deposit the paint
        RET
```

```
;Scan the array. & Output it.  By convention the maximum Y value is
;output first. (top of the paper).  After the array is written, clear it.
; FOR y := ymax DOWN TO ymin DO
;   FOR x := xmin TO xmax DO writecharacter(xy[x,y])
PUTARY: MOVEI   W,YMAX                  ;Outer loop index.  From YMAX down to YMIN
PUTNTY: MOVE    Y,W                     ;Here to start one row. (Y-value)
        IMULI   Y,XSIZE                 ;compute row-origin
        MOVE    X,[-XSIZE,,XMIN]        ;step column (X) thru all values.
PUTNTX: HRRE    Z,X                     ;obtain column number, extend sign
        ADD     Z,Y                     ;add row origin.
        SKIPN   A,XYORG(Z)              ;skip if square was painted
         MOVEI  A," "                   ;unpainted, use "blank" paint
        CALL    PUTCHR                  ;store a character in the file
        AOBJN   X,PUTNTX                ;advance to the next column
        MOVEI   A,15                    ;add carriage return
        CALL    PUTCHR
        MOVEI   A,12                    ;and line feed, to the file.
        CALL    PUTCHR
        CAMLE   W,[YMIN]                ;have we finished the last row?
        SOJA    W,PUTNTY                ;No, decrement row number and loop
;Clear the array.
CLRARY: SETZM   XY                      ;zero the array.
        MOVE    A,[XY,,XY+1]
        BLT     A,XY+ARYSIZ-1
        RET

;Put one character, in A, into the output file.
PUTCHR: PUSH    P,B                     ;Save register B
        MOVE    B,A                     ;copy character to B
        HRRZ    A,OJFN                  ;JFN to register A
        BOUT                            ;Send one byte
         ERJMP  ERROR                   ;in case of error
        POP     P,B                     ;Restore original contents of B
        RET

CLSFIL: HRRZ    A,OJFN                  ;close the output file
        CLOSF                           ;close and release the JFN
         ERJMP  ERROR
        RET

        END     START
```

21.2.4 .TEXT Pseudo–Op

This program introduces the `.TEXT` pseudo–op. This is used to pass commands to LINK. Note that the commands pend, unread, until the next time that LINK is ready to read commands; generally, this is after LINK has finished loading the present file. Therefore, such commands don't generally affect how the present file is loaded.

The word `.TEXT` is followed by a delimiter character, the text to pass to LINK, and a copy of the same delimiter. The format is very much the same as for `ASCIZ`.

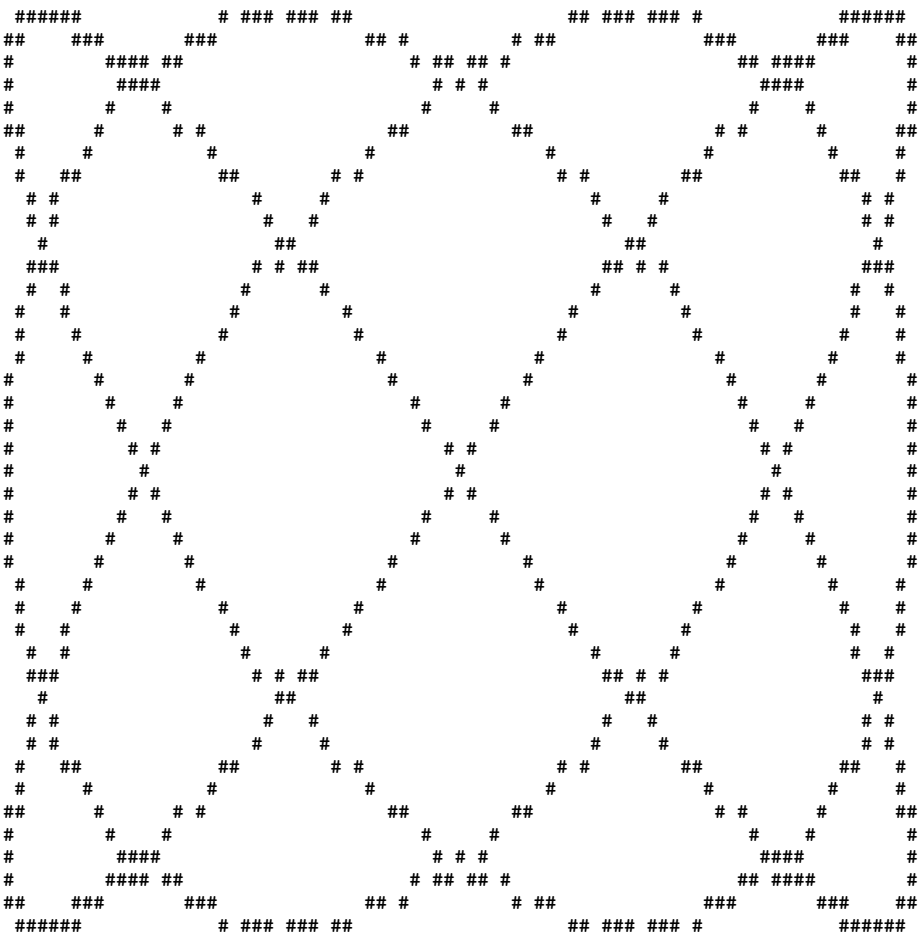


Figure 21.1: Sample Output from the LISAJ Subroutine

The command passed to LINK is `/REDIRECT:DATA:CODE`. The Fortran Library was created for TOPS–10 in which its components would have been loaded into the *high–segment*, an early analog of today's read–only psect. The `/REDIRECT` command tells LINK to put anything aimed at the low segment into the DATA psect and anything aimed at the high segment into the CODE psect. The effect is that when LINK gets around to loading the SIN function from the Fortran library, it will append it to the CODE psect instead of loading it in section 0 at addresses starting at 400010. (This is a further example of how, having decided to use extended addressing in this program we must systematically shun section zero.)

21.2.5 Fortran Library SIN Function

(Since 1981, when this book was first published, the Fortran library has changed. The new library provides routines that are reentrant; they allocates writeable variables on the stack. However, the version below works also.)

In case you find yourself curious, the following is a listing of the `SIN` routine that is found in version 5A of the DECSYSTEM–20 Fortran Library. The code comes from `SYS:FORLIB.REL`. This author used DDT to disassemble the code, to which he added comments.

This is not the very best computation of $\sin(x)$, but it illustrates some of the influence of mathematics on programming. This example is presented without a full explanation; some parts are quite intricate, but the comments are more extensive than usual.

By the way, if you fail to find yourself curious about the mathematical basis of the `SIN` function, you may skip ahead to Section 21.3, page 334.

This program displays some antique foibles and crotchets. Some of these are explicable; others defy explanation. Among thos explicable:

The sequence

```
Label:  CAIA
         PUSH P,CEXIT.
```

commemorates an earlier FORTRAN compiler that made use of the `JSA` instruction to call subroutines. `JSA` is now considered obsolete; it cannot be used for intra–section jumps because it truncates the saved PC to 18 bits. Moreover `JSA` has the unattractive property of writing into the instruction stream: as with `JSR`, `JSA` writes into `C(E)` and jumps to `E+1`.

This routine allows for the possibility of being called from old compiled code; if entered by `JSA`, the `CAIA` instruction would be overwritten and the `PUSH` would be executed. A subsequent `POPJ` causes the routine to jump to the special exit sequence that repairs the clobbered entry instruction and returns to the caller. The present program does not make use of `JSA`: the subroutine is part of a read–only psect.

The sixbit labels help a person navigate when there are no symbols. In TOPS–10 it was customary to omit the symbols from the running program, because without virtual memory, they are costly. Moreover, there was no way by which DDT could be loaded dynamically. Even when the program was loaded including DDT, it was customary to omit symbols from libraries because it was assumed that the libraries were already debugged.

Instructions in self–mode that reference only one accumulator, e.g., `MOVNS 1` would by preference be replaced with `MOVN 1,1`. The effect is identical, but the latter is faster in all known implementations.

The `JFCL` instructions following some floating point operations are no–ops when executed by the

hardware. However, if a floating point instruction were to trap, the Fortran runtime environment would handle the trap, examine the trap address and find the JFCL. The runtime environment would accept the JFCL as notice to suppress its own messages and to ignore the trap condition. This was, in effect, an early form of ERJMP.

Among the mysteries: a person can only speculate about the programmer's intent when staring at POPJ 17,1, just below the label S3A. The effective address field of POPJ is unused. People have been known to put 18–bit constants in the right half of a POPJ, but this value doesn't seem to be used anywhere.

```
        TITLE SIN Sine, Cosine and related functions from FORLIB
        ENTRY SIN,COS,SIND,COSD         ;mark subroutine entry points
        TWOSEG                          ;make a 2-segment program

Comment $

The Maclaurin series expansion for sin(X) (for X near zero) is:

                3      5     7     9
               X      X     X     X
sin(X) =  X -  ---  + --- - --- + --- - ...
               3!     5!    7!    9!

This subroutine transforms the given angle by dividing it by pi/2.
The transformed angle is normalized into the first or fourth
quadrants (between -1.0 and +1.0) before expanding this series.
Bringing the angle into the first or fourth quadrant assures
that five terms of the series are sufficient for an accurate result.

What we compute is
                                  2    2    2
sin(X) =     (c1+(c3+(c5+(c7+c9*x' )*x' )*x' )*x'

where x' = X/(pi/2) and transformed to quadrant 1 or 4, and

                        3             5                 7             9
                   (pi/2)        (pi/2)            (pi/2)        (pi/2)
c1 = pi/2;  c3 = - ------;  c5 = -------;   c7 = - ------;  c9 = -------
                     3!            5!                7!            9!
---------

Calling sequence:
        MOVEI   16,ARGLST       ;address of argument list
        PUSHJ   17,SIN
        return here with result in register 0
        register 1 has been used as a temporary

        -1,,0                   ;length of list, a Fortran standard
ARGLST: 4,ARG                   ;argument list points to the word containing
ARG:    0                       ;the argument.

---------
```

```
Alternate calling sequence, for compatibility with F40, an older Fortran

        JSA     16,SIN
        JUMP    ARG
        return here with result in register 0
        register 1 has been used as a temporary

---------

In addition to SIN, entries are provided for COS, cosine; SIND, sine of
an argument given in degrees rather than radians; and  COSD, cosine of
an argument given in degrees.

$

        SIXBIT  /COSD/          ;Cosine of argument in degrees
COSD:   CAIA                    ;new-style entry.  Skip.
         PUSH    17,CEXIT.      ;save "return" address for old style entry
        MOVE    1,@0(16)        ;fetch argument
        FADR    1,CD1           ;add 90.0 degrees
        FDVR    1,SCD1          ;divide by 57.295 degrees per radian
        JFCL
        JRST    S1

        SIXBIT  /SIND/          ;Sine of argument in degrees
SIND:   CAIA                    ;new-style entry.  Skip.
         PUSH    17,CEXIT.      ;save "return" address for old style entry
        MOVE    1,@0(16)        ;get argument
        FDVR    1,SCD1          ;divide to convert to radians
        JFCL
        JRST    S1

        SIXBIT  /COS/           ;Cosine of argument in radians
COS:    CAIA                    ;new-style entry.  Skip.
         PUSH    17,CEXIT.      ;save "return" address for old style entry
        MOVE    1,@0(16)        ;get the argument
        FADR    1,PIOT          ;add pi/2.  COS(X) = SIN(X+pi/2)
        JRST    S1

        SIXBIT  /SIN/
SIN:    CAIA                    ;new-style entry.  Skip.
         PUSH    17,CEXIT.      ;save "return" address for old style entry
        MOVE    1,@0(16)        ;fetch the argument
S1:     MOVEM   1,SX            ;save the argument
        MOVMS   1               ;absolute value
        CAMG    1,SP2           ;for arguments close to zero, approximate
         JRST   S3A             ; Sin(X) = X.  go return X
        MOVEM   2,SC            ;save one accumulator
        FDV     1,PIOT          ;divide by pi/2
        CAMG    1,ONE           ;is argument in the first quadrant?
         JRST   S2              ;yes. - easy to do
```

```
;Here if the magnitude of the original argument exceeds pi/2.
;
;The number in register 1 is abs(x)/(pi/2).    2   |   1
;A value from 0 to 1 represents quadrant 1         |
;             1 to 2 represents quadrant 2   ------+-------
;             2 to 3 represents quadrant 3         |
;             3 to 4 represents quadrant 4     3   |   4
;above 4, take this argument modulo 4.
        MULI    1,400           ;exponent to 1, fraction to 2
        LSH     2,-202(1)       ;fix the point between bits 2 and 3
                                ;cast out eights.
        TLZ     2,400000        ;cast out fours.
;register 2 now contains a fixed point quantity, 0 <= C(2) < 4.0
;the binary point is located between bits 2 and 3.
        MOVEI   1,200           ;exponent is excess 200
        ROT     2,3             ;move the integer part to bits 33:35
        LSHC    1,33            ;shift 27 fraction bits into 1
;An un-normalized floating point number is present in register 1.
;It is x0 = [abs(X)/(pi/2)] modulo 1.0.  "How far into the quadrant"
;Register 2 contains the (quadrant number-1) which has just been shifted
;up from bits 34:35 to bits 7:8 (2000,,0 and 1000,,0). The rest of register
;2 is zero.
        FAD     1,SP3           ;add zero to normalize the number
;The following code will arrive at S2 with register 1 containing "x"
;a normalized number, less than 1.0 in magnitude, defined in terms of x0
;as follows:
;       let q  = (abs(X)/(pi/2) - x0) modulo 4 in bits 7:8 of register 2
;       (in this transformed system, 1.0 represents pi/2)
;       q = 0, quadrant 1, x = x0       sin(x)         = sin(x)
;       q = 1, quadrant 2, x = -(x0-1)  sin(x + pi/2)  = - sin(x - pi/2)
;       q = 2, quadrant 3, x = -x0      sin(x + pi)    = - sin(x)
;       q = 3, quadrant 4, x = x0-1     sin(x + 3pi/2) = sin(x -  pi/2)
;
        JUMPE   2,S2            ;jump if quadrant 1.
        TLCE    2,1000          ;skip if quadrant 3.
         FSB    1,ONE           ;quad 2 or 4, reflect to previous quad. x0-1
        TLCE    2,3000          ;skip if quad 2 - go negate x
         TLNN   2,3000          ;Quad 3 or 4. skip if Q4
          MOVNS 1               ;Quad 2 or 3, negate argument
                                ;fall into S2
```

```
S2:     SKIPGE  SX              ;test sign of original argument
         MOVNS  1               ;if negative, negate adjusted arg again.
        MOVEM   1,SX            ;save x, = X/(pi/2), adjusted, reflected
        FMPR    1,1             ;                      x^2
        MOVE    0,SC9           ;                   c9
        FMP     0,1             ;                   c9*x^2
        FAD     0,SC7           ;                c7+c9*x^2
        FMP     0,1             ;               (c7+c9*x^2)*x^2
        FAD     0,SC5           ;            c5+(c7+c9*x^2)*x^2
        FMP     0,1             ;           (c5+(c7+c9*x^2)*x^2)*x^2
        FAD     0,SC3           ;        c3+(c5+(c7+c9*x^2)*x^2)*x^2
        FMP     0,1             ;       (c3+(c5+(c7+c9*x^2)*x^2)*x^2)*x^2
        FAD     0,PIOT          ; pi/2+(c3+(c5+(c7+c9*x^2)*x^2)*x^2)*x^2
S2B:    FMPR    0,SX            ;(pi/2+(c3+(c5+(c7+c9*x^2)*x^2)*x^2)*x^2)*x
                                ;(pi/2)*x + c3*x^3 + c5*x^5 + c7*x^7 + c9*x^9
        SKIPA   2,SC            ;restore saved copy of AC 2 and skip.
S3A:     MOVE   0,SX            ;for small X, Sin(X) = X
        POPJ    17,1            ;return to caller, unless called by
                                ; a JSA instruction, in which case, jump
                                ; to CEXIT.+1.

SC3:    -0.64596371             ;-((pi/2)^3)/3!
SC5:    0.079689680             ; ((pi/2)^5)/5!
SC7:    -0.0046737656           ;-((pi/2)^7)/7!
SC9:    0.00015148418           ; ((pi/2)^9)/9!
SP2:    170000,,0               ;a number larger than 0.001953
SP3:    0                       ;a floating point zero
CD1:    90.0                    ;90.0 degrees for SIND, COSD
SCD1:   57.295779               ;57.295 degrees per radian for SIND, COSD
PIOT:   1.5707963               ;pi/2
ONE:    1.0                     ;constant 1.0

        RELOC                   ;to low segment
SX:     0                       ;room to store the (adjusted) argument
SC:     0                       ;save accumulator 2 here

;Special exit to restore things if any of these routines is called by the
;old JSA instruction.
CEXIT.: 0,,CEXIT.+1             ;address of special exit if called by JSA
        HLRM    16,CEXIT1       ;store the entry address in the EXCH instr
        HRLI    16,(<CAIA>)     ;LH of 16 gets a CAIA instruction
        HRRM    16,CEXIT2       ;store the return address in the JRST
CEXIT1: EXCH    16,0            ;store CAIA back at the entry point,
CEXIT2: JRST    0               ;restore original value of 16 and return.

        END
```

21.3 Multi–Dimensional Arrays

The use of indirect addressing through side–tables is possible but complicated as the number of dimensions increases. Address polynomials also become more complex, but the increased complexity

is manageable in a systematic manner.

If we have an array S with k dimensions, defined by

S: ARRAY [L_1..U_1,L_2..U_2, ... ,L_k..U_k]

then the address of a specific element, S[i_1,i_2, ... ,i_k], is given by the formula

$$S0 + (...(E_1 \times D_2 + E_2) \times D_3 + E_3...) \times D_k + E_k$$

where S0 is the address of S[L_1,L_2, ... ,L_k], D_j is the length of the j–th dimension, given by the expression U_j - L_j + 1, and E_j is the j–th normalized subscript, given by i_j - L_j.

When this formula is applied, it is usual to gather the constant terms together into one term that represents the address of S[0,0,...,0]. This address replaces S0 above, and the normalized subscripts are replaced by the actual subscripts. Although it is somewhat cumbersome to express this formula in writing, it is relatively simple to compute. Address polynomials are used by many language compilers for accessing arrays.

21.4 Arrays in Remote Address Sections

If the size of the array is known in advance, it may be allocated statically in a psect devoted to it (and to other large data structures):

```
        .TEXT   "/PVBLOCK:PSECT:CODE/SYMSEG:PSECT:CODE"

        .PSECT  BIG,2000000
BIGARY: BLOCK   400000
        .ENDPS

        .PSECT  CODE/RONLY,1002000
        ...

        MOVE    2,@[<BYTE (1)0,0(4)1>+BIGARY]
        ...

        .ENDPS
```

In the extended addressing machine the only access to data in a different address section is by indexed or indirect addressing. Consequently, any reference to the array BIGARY will use one or the other of these addressing forms.

Shown above is an indirect, indexed method. Presume that accumulator 1 contains the index to select one item in BIGARY. Then the instruction MOVE 2,@[<BYTE (1)0,0(4)1>+BIGARY] suffices to access the desired item. Because BIGARY is defined in one psect and used in another, MACRO and LINK will use the 30–bit address of BIGARY in the expression in the literal. The remainder of the literal assembles a global–format indirect word that specifies register 1 as an index register. By the rules of effective address calculation, this combination accesses the word at the address of BIGARY plus the contents of accumulator 1.

Alternate code forms that could be used include either of the following, where `idxval` stands for a location containing the desired index. In either case the literal `[BIGARY]` is expected to contain the 30–bit address of the array.

```
MOVE   1,idxval              MOVE   1,[BIGARY]
ADD    1,[BIGARY]            ADD    1,idxval
MOVE   2,(1)                 MOVE   2,(1)
```

21.4.1 Program Data Vector

This fragment uses `.TEXT` to pass two commands to LINK.

The first command, `/PVBLOCK:PSECT:CODE`, specifies the name of the psect into which to place the *Program Data Vector*. The program data vector, also known as the PDV, contains information about the program that has been collected by LINK. Schematically, the PDV contents and related data are displayed in Figure 21.2. (Additional information about the format of the PDV is found with the documentation of the `PDVOP%` JSYS.)

21.4.2 Symbol Table

The second command in `.TEXT`, `/SYMSEG:PSECT:CODE`, directs LINK to put the symbol table in the CODE psect. The format of the symbol table is discussed in Section 29.2.4.

21.4.3 Dynamic Arrays

If, instead of a static array, the array is allocated dynamically, we cannot use MACRO and LINK to build address words. We must build the address word(s) ourselves.

Although the details of dynamic memory allocation are beyond the scope of the present example, we can say that, given the 30–bit address of array entry [0] in accumulator `1` (with 0 in bits `0:5`) and the index value in accumulator `2`, the following fragment builds a global indirect word in `1` and uses it to access an array entry.

```
        MOVE    1,arybas      ;get the 30--bit address of array[0]
        MOVE    2,idxval      ;set the index value in register 2
        TXO     1,2B5         ;set GIW index field to 2
        MOVE    3,@1          ;access array element. Index by AC 2 implicit
```

In an extension of this scheme, build a table of global indirect words, each using a different index register field:

```
        MOVSI   4,-20         ;count thru the index registers
        MOVE    1,arybas      ;the 30--bit address of array[0]
LOOP:   DPB     4,[POINT 4,1,5]  ;set the GIW index field
        MOVEM   1,ARYACC(4)   ;store GIW to access using X register N
        AOBJN   4,LOOP        ;   in ARYACC+N
```

Program Data Vector Content

Word Number (relative to start of PDV)	Content	Meaning
0	16	length of the PDV, including this word
1	pointer	address of ASCIZ program name string
2	0	address of the export information vector
3	0	
4	0	program version number
5	pointer	address of the memory descriptor block
6	pointer	address of the program symbol vector
7	date & time	of module assembly
10	0	version number of the compiler
11	date & time	when loaded into memory
12	100700,,2425	Version of LINK, i.e., 7(2425)-1
13	0	
14	0	
15	0	Address of a user–defined data block

(The program version number is more fully described in Section 26.5.1.)

Memory Descriptor Block Contents

Word Number (relative to start of MDB)	Content	Meaning
0	7	Length of MDB, including this word
1	3	flags,,Length of descriptor
2	1,,2000	Low address in this region
3	1,,2276	Highest address in this region
4	40000,,3	flag (writeable),,Length of descriptor
5	2,,0	Low address in this region
6	2,,377777	Highest address in this region

Program Symbol Vector Contents

Word Number (relative to start of PSV)	Content	Meaning
0	7	Length of PSV, including this word
1	10000,,136	flags (defined symbols),,length
2	1,,2141	lowest address in symbol table
3	0	
4	20000,,0	flags (undefined symbols),,length
5	1,,2141	lowest address in undefined symbol table
6	0	

Figure 21.2: Program Data Vector and Associated Descriptors

The 20–word table ARYACC is prepared once. Thereafter, use when the array must be accessed. Just select the entry that corresponds to the register that you want to use:

```
MOVE    3,idxval      ;load 3 with an index value
MOVE    7,@ARYACC+3   ;Index register 3 is implicit.
```

21.5 Efficiency Considerations

The choice between row major and column major forms can result in significant performance differences. Essentially, you should match the form of the array to the type of processing being done. For example, if an inner loop of the program steps among elements of one row (i.e., changes only the column number), while an outer loop is used to step from row to row, then row major order is indicated. In the DECSYSTEM–20 there are two specific reasons why this is appropriate.

The first reason is the *cache memory*. The cache is a high–speed buffer memory that is much faster than the main memory. The cache stores words that have recently been used; also, it anticipates the future need for some words.[7] Specifically, because of this anticipation, assuming that you store `Q: ARRAY [1..8,1..20]` in row–major order, after you access `Q[5,3]`, three other array elements in the "neighborhood" of `Q[5,3]` will be present. These might be `Q[5,2]`, `Q[5,4]` and `Q[5,5]` (the actual selection depends on the precise address of `Q[5,3]`). Thus, if you happen to be processing row 5 of this array, you will obtain some benefit from the cache, since several words from this row have now appeared.

The second reason is the TOPS–20 virtual memory. In TOPS–20 your program's memory space is divided into pages of decimal 512 words each. A page must reside in main memory while it is being referenced by the program. A *dormant page* is a page that contains useful data or pieces of program, but that has not been accessed recently. In order to conserve main memory, TOPS–20 will move dormant pages to the disk. A *page fault* occurs when the program requests a word from a page that is not present in main memory. When a page fault occurs, TOPS–20 suspends the program until that page can be brought into memory.

Virtual memory interacts with array storage in the following way. In row major form, all of one row of the array is allocated in contiguous space. Contiguous space will span no more than one page more than the minimum number of pages needed to allocate the row. This means that references along one row will cause only a small number of pages to be referenced. This reference pattern minimizes the occurrence of page faults, and keeps the program's *working set*, the set of actively referenced pages, to a small size. Both of these characteristics diminish the program's demand for system resources, and generally contribute to increased system–wide efficiency and throughput.

To give a very specific example, Pascal stores arrays in row major form. Suppose an array called `s` has been declared by `s: ARRAY [1..100,1..100] OF INTEGER;`. Then, the following two program fragments perform the same function, but have remarkably different working set requirements (and page fault behavior).

[7] The cache that is described here is the one present in the 2050 and 2060 configurations; other configurations may act somewhat differently.

```
(* Fragment "A" *)
FOR i:=1 TO 100 DO
  FOR j := 1 TO 100 DO
    s[i,j] := s[i,j] + 1;
```

```
(* Fragment "B" *)
FOR j:=1 TO 100 DO
  FOR i := 1 TO 100 DO
    s[i,j] := s[i,j] + 1;
```

In fragment "A", the column index, `j`, is varied by the inner loop; the row index remains constant while all columns are scanned. This fragment references words that have been stored in sequential locations. The entire array, containing 10,000 words, spans about 20 pages. The characteristic of this program is that the pages comprising the array are referenced sequentially. Once the first page has been referenced, the program performs all its work on that page. When it touches the second page, it has finished all its work on the first page. Each of the 20 pages is touched in succession, and processed entirely, and then becomes dormant. The working set of this program can be limited to the program page, and one data page.[8]

In contrast, fragment "B" varies the row index, `i`, in the inner loop. Since each row occupies 100 words, the elements `s[1,1]` and `s[7,1]` are 600 words apart; these must be on different pages. Each time through the inner loop, this fragment references all of the 20 pages that the array spans. Then, for the next value of the column index, this fragment repeats these references to the 20 data pages. When the system that runs fragment "B" has sufficient resources, the working set size of this fragment will be nearer 21 pages (20 for data, one for program) than the 2 pages used by fragment "A".

If the particular computer system that runs fragment "B" is pressed for main memory space, the page containing `s[1,1]` may be forced out to the disk to make room for the page containing `s[90,1]`. When, shortly thereafter, the program references `s[1,2]`, which is the word adjacent to `s[1,1]`, that page will just have to be brought in again. Very little real work will get done between page faults.

This behavior, where very little progress is made because of frequent page faults, is called *thrashing*. Thrashing is generally characteristic of an operating system, system configuration, and the set of programs active on the system. Thrashing is not usually the result of a particular user program. However, a program that uses memory in an incautious or wasteful fashion can cause an operating system (and specific hardware configuration) to exhibit thrashing.

The use of fragment "B" instead of fragment "A" causes a needless demand on system resources. Although real–life situations are usually less clear–cut than this example, the programmer should consider his or her program's effect on system performance.

21.6 Array Exercises

These exercises are an opportunity to practice using two–dimensional arrays.

21.6.1 Magic Square

A magic square of order N consists of an array of numbers from 1 to N^2 arranged in the form of an $N \times N$ square, so that the sum of each column, of each row, and of each of the two main diagonals is identical.

[8] This is an oversimplification, but the principle is correct.

There are many simple methods for constructing a magic square of a specific order. The methods used in constructing magic squares are usually specific to the parity of the square; that is, one set of methods is used for constructing squares with an odd order, and a different set of methods is used to construct even–order squares.

Your assignment is to write a program which constructs and prints magic squares. For each square, your program should read in the order of the square from the terminal, and generate the square using the De La Loubere method, described below. The De La Loubere method works for squares of odd order only.

Since it is difficult to print a square larger than order 20 on the terminal screen, you may assume that no square larger than order 19 will be required.

Your program should ask the terminal user for the order of the square to be printed. It should reject bad responses, i.e., even numbers and numbers larger than 19. The program should terminate when given zero as the order. Your program should label each magic square with a header telling its order.

The output should be sent to both the terminal and to an output file. Turn in a listing of your program and the output produced for squares of order 5, 11, and 19.

The De La Loubere method can be used to generate a magic square of any odd order. To illustrate the method, we will follow through the steps involved in creating an order 5 magic square.

1. Place the number 1 in the center cell of the top row.
2. Move to the next cell in a diagonal manner, going one to the right and up one. In this case, the movement results in going off the top of the array. Whenever you step off the top, it is necessary to go instead to the bottom row. Place the next number, 2 in this case, in the bottom row, one to the right of the center column.
3. Now move diagonally right and upwards again. Place the next number, 3, in the cell you find there.
4. Moving up and to the right again, forces you off the right side of the array. Go instead to the leftmost column of the array. Place the number 4 there.
5. Once again move up and to the right; place the number 5 there.
6. If you were to move up and to the right again, you would find the square is already occupied with the number 1. Instead of moving up and right, move down one cell. Place the next number, 6, under the cell that has the 5 in it. In a square of order N, this downward step is needed after each group of N elements has been placed.
7. Repeat these steps until the array is filled. Whenever the diagonal movement results in stepping off the top of the array, re–enter the array at the bottom. If you step off the right side of the array, re–enter at the left. Whenever a collision occurs, when by a diagonal step you select a cell that is already occupied, step down instead.

Here is the magic square that is produced by following these steps:

```
17  24   1   8  15
23   5   7  14  16
 4   6  13  20  22
10  12  19  21   3
11  18  25   2   9
```

The sum of each of the five rows, each of the five columns, and each of two main diagonals is 65 which is $(N^3 + N)/2$.

Extra credit: this exercise was designed to give you practice with arrays. However, there really isn't any need to use an array here. Develop a closed–form expression for the value of a position, given three parameters: the order of the square, a row number, and a column number.

21.6.2 Tic–Tac–Toe

Write a program to play tic–tac–toe with a person at the terminal.

Tic–tac–toe is played on a 3 by 3 open grid of squares. The two players alternate turns. At each turn, a player claims one of the empty squares. The symbol `X` is placed in the squares claimed by one player; the other player uses the symbol `O` to mark his squares. A player wins the game by filling an entire row, column, or diagonal with his symbols. If neither player has won and all the squares are filled, the game ends with no winner.

The user will enter his moves through the terminal, and the computer will indicate its response by displaying an updated game board after each of its moves. Your tic–tac–toe board might look like the example pictured here:

```
  OO  |      |  OO
 O  O |      | O  O
O    O|      |O    O
 O  O |      | O  O
  OO  |      |  OO
------+------+------
      |X    X|
      | X  X |
      |  XX  |
      | X  X |
      |X    X|
------+------+------
      |      |X    X
      |      | X  X
      |      |  XX
      |      | X  X
      |      |X    X
```

The computer's move will be represented by an `O`; the user's symbol is `X`.

The computer will continue playing games of tic–tac–toe until the user decides to stop. After each game, the computer will announce the result of the game and ask if the user wants to play again. The first move will go alternately to the user and to the computer. In the first game, the user gets the first move. In the second game, the computer gets the first move, etc.

Although it is possible to write a tic–tac–toe program that never loses, you are not required to do that. Your program must be able to do the following:

1. All moves made by the computer and by the user must be legal tic–tac–toe moves. If the user enters an illegal move, the program must reject it and ask for another move.
2. The computer must recognize when the game is over and print an appropriate message.

3. If the computer has a chance to win on a given turn, it must do so. In other words, if a row, column or diagonal has two `O`'s and a blank, the computer must move to that blank to win.
4. If the computer cannot win on a given turn, but the user has a chance to win on his next turn, the program must thwart that winning move, if possible.
5. If the computer's move is not forced by the above rules, then the computer may make any legal move.

You may allow the user to enter his moves in any form that you want. One possibility is to let each square on the board be represented by a pair of integers corresponding to the indices of a 3×3 matrix:

(1,1)	(1,2)	(1,3)
(2,1)	(2,2)	(2,3)
(3,1)	(3,2)	(3,3)

When prompted, the user would indicate his move by typing two indices separated by a space.

After each move by the computer, the program will display the tic–tac–toe board containing all the moves made thus far during this game. The board you display may be any size that you like (be reasonable!) but it must use more than four characters to represent each `X` and each `O`. The board depicted above is one example of an acceptable output format.

Output the play both to the terminal and to a file. When the user declines the invitation to play another game, close the output file. Turn in the program and the file displaying the results of at least three games.

21.6.3 Triangular Matrices

Often a matrix can be reduced to a *triangular form* in which all elements below (or above) the main diagonal are zero. All non–zero elements are clustered in the upper (or lower) triangle. An $N \times N$ upper triangular matrix can be stored in $[N \times (N + 1)]/2$ locations. Develop an addressing formula for such a matrix.

Chapter 22

File Input

We come now to the problem of how to read data from an external file into the program's memory space. Several of the JSYS operations that we have already studied are relevant. These include `GTJFN`, `OPENF`, and `CLOSF`. In addition to these, we will need to learn the `SIN`, *String INput*, JSYS, and the `GTSTS`, *GeT STatuS*, JSYS. To increase our ability to handle errors in an intelligent and tolerant way, we will also employ the `GETER`, *GET ERror*, JSYS.

String input is quite similar to string output, or to the `RDTTY` functions that we are familiar with. The major new problem we face when doing input processing is how to detect the end of file.

However, before we get to the gist of the matter, we digress to introduce some additional macro helpers.

22.1 TX Macro Family

In a manner somewhat akin to the `MOVX` macro, the `TXNN` macro will generate one of `TRNN`, `TLNN`, or `TDNN`, as appropriate to its second actual parameter. This relieves the programmer of remembering which halfword a particular symbolic name relates to. (It also frees the programmer to rearrange such symbols if necessary.)

`TXNN` is one of a family of sixteen macros provided in `MACSYM`. The family includes all of the macros listed in Figure 22.1.

By analogy with the `MOVX` macro, it should be clear what function is performed here. If you think you want a `TRON` instruction, but you're not certain that you are testing a right–half flag, use the `TXON` macro. `TXON` will generate either a `TRON`, a `TLON`, or a `TDON` instruction, whichever is appropriate for

```
TXN    TXO    TXZ    TXC
TXNE   TXOE   TXZE   TXCE
TXNN   TXON   TXZN   TXCN
TXNA   TXOA   TXZA   TXCA
```

Figure 22.1: TX Macro Family

testing the mask that you specify.[1] We will not display the definition of of these macros: you can find them in `MACSYM.MAC` as distributed to DECSYSTEM–20 installations.

22.2 SIN JSYS

The `SIN` JSYS requires a JFN in register 1. Register 2 contains a string pointer to the input buffer area. Register 3 contains either a negative character count, zero, or a positive character count.

When the `SIN` function is executed with register 3 containing a negative number, precisely that number of characters will be read into the input buffer area. A short count can happen only if some error condition occurs. At the conclusion of this `SIN` call, register 3 will be updated towards zero by the number of characters actually read.

When the `SIN` function is executed with register 3 containing a positive character count, then input stops either when this count is exhausted, or when a character is input that matches the character found in register 4. Again, register 3 will be updated towards zero by the number of characters actually read.

When register 3 contains zero and `SIN` is called, `SIN` reads until it finds a zero byte in the input stream. Unless the programmer is certain that a zero byte is present, this has the potential of overrunning the space allocated in the program's memory space for the input buffer.

22.3 GTSTS JSYS

The `GTSTS` JSYS allows us to determine the status of any JFN and its associated file. `GTSTS` requires one argument, the JFN, in register 1. It returns the status of that JFN in register 2.

In the example program we will use the `GTSTS` JSYS to analyze any error from the `SIN` JSYS. The error that we expect from `SIN` is end of file.

22.4 GETER JSYS

The `GETER` JSYS allows us to determine the *error number* of the most recent error. We shall use this JSYS as part of the error recovery process when the user types an incorrect file name.

22.5 Example 13 — Search Program

The program that we shall use as an example of file input performs a modestly useful function. Through a dialog at the terminal, the user is allowed to specify the names of an input file and an output file. Also, the user tells the program the text of a search string.

The program reads the input file. Each time it finds a line that contains the specified search string, it copies that line to the output file. Headings and a summary are also sent to the output file.

The following is a sample session demonstrating this program. The user's type–in is underlined.

[1] In some special circumstances, e.g., left half of the mask all ones, or the entire mask ones, macros in the `TX` family may generate `ORCMI`, `ANDI`, `EQVI`, or even `CAIE` or `CAIN`.

```
@execute ex13
MACRO:  FILE
LINK:   Loading
[LNKXCT FILE Execution]
File name for input: acct:<acct1>win78-master.list
File name for output: tty:
Type the search string on one line:
*gorin

Search of ACCT:<ACCT1>WIN78-MASTER.LIST for
gorin
------------------------------------------------------------------------
Ralph Gorin ( Overhead. ) L G.GORIN
Ralph Gorin ( Other. ) S D.DEMO

There were 2 matches found in the file
File name for input: acct:<acct1>win78-master.list
File name for output ex13.dat
Type the search string on one line:
*overhead
File name for input: _
Goodbye
@type ex13.dat

Search of ACCT:<ACCT1>WIN78-MASTER.LIST for
overhead
------------------------------------------------------------------------
* * ( Overhead. ) S 3.3-documentation
* * ( Overhead. ) S 3.3-system
* * ( Overhead. ) S 3.3-subsys
* * ( Overhead. ) S 3-sources
Queenette Baur ( Overhead. ) L Q.QUEENIE
Michael Byron ( Overhead. ) L B.BYRON
Ralph Gorin ( Overhead. ) L G.GORIN
J. Q. Johnson ( Overhead. ) L J.JQJOHNSON

There were 8 matches found in the file
@
```

22.5.1 Structured Programming

This program is the most complex that we have seen thus far; as programs become more complex, our need to divide them into manageable modules is greater. There are techniques that are especially applicable to managing large programs; these techniques are collectively referred to as *structured programming*. Among the main ideas in structured programming are the division of problems into subroutines and the selection of appropriate control structures. In this example we have divided the

program into a relatively large number of understandable subroutines.

The main program appears below. It is extremely compact; nearly all the work is done by calling subroutines:

```
;Main Program
START:  RESET
        MOVE    P,[IOWD PDLEN,PDLIST]   ;Set up the stack
NEXT:   CALL    GTINPF                  ;Get an input file
         JRST   DONE                    ;No file was given: exit
        CALL    GTOUTF                  ;Get an output file
        CALL    GTSTRG                  ;Get the search string
        CALL    HEADER                  ;Write a heading
        CALL    FIND                    ;find the matches
        CALL    FIN                     ;finish up, write trailer
        JRST    NEXT                    ;go do some more

DONE:   HRROI   A,[ASCIZ/Goodbye
/]
        PSOUT                           ;print ending message
        HALTF                           ;stop
        JRST    START                   ;restart if continued
```

There are several compelling reasons why this style of programming has been adopted. First, partitioning the program into subroutines makes the program more readily understandable. The structure of the process should be apparent to any reader: get an input file, get an output file, get a search string, write the header, read the file and find the matches, finish up, and repeat.

Another reason to adopt this structure is that the subroutines that are used may prove to be useful components that we can carry into other programs.

A third reason is ease of debugging. A subroutine that performs a well–defined function can be debugged independently of other portions of the program. The more pieces that can be carved off and debugged by themselves, the less there will be that is hard to debug.

A fourth reason is ease of writing the program. Often the entire structure will be apparent from the beginning. By deferring the messy details until later, more progress can be made establishing the framework of the overall structure. Whenever you're stuck for a solution, instead of getting bogged down, call a subroutine.

A fifth reason is that a subroutine interface can act as a *firewall*. A firewall keeps bugs contained inside small modules. For example, a subroutine could check for the validity of its inputs and outputs. When some discrepancy occurs, it can blow the whistle: either it has produced a faulty output, or its caller has been passing incorrect arguments. The more checking that goes on (and the smaller the modules that are checked) the easier it will be to pinpoint faulty modules and effect corrections.

A sixth reason (we could go on) is that in all large programming projects subroutines, firewalls, and detailed interface specifications are mandatory. Since a large program is the work of many individual intellects, an overall structure must be imposed and adhered to. The most useful structure yet discovered is to divide problems into subroutines.

The text of the program itself appears below. As we have done before, we present the entire program here; the analysis of the individual subroutines will follow.

```
        TITLE   File Input/Output and Search,  Example 13
        SEARCH  MONSYM,MACSYM

A=1
B=2
C=3
D=4
W=5
X=6
P=17

;Main Program
.PSECT  CODE/RONLY,1002000      ;set program into section 1

START:  RESET
        MOVE    P,[IOWD PDLEN,PDLIST]   ;Set up the stack
NEXT:   CALL    GTINPF                  ;Get an input file
         JRST   DONE                    ;No file was given: exit
        CALL    GTOUTF                  ;Get an output file
        CALL    GTSTRG                  ;Get the search string
        CALL    HEADER                  ;Write a heading
        CALL    FIND                    ;find the matches
        CALL    FIN                     ;finish up, write trailer
        JRST    NEXT                    ;go do some more

DONE:   HRROI   A,[ASCIZ/Goodbye
/]
        PSOUT                           ;print ending message
        HALTF                           ;stop
        JRST    START                   ;restart if continued

;Search the file.  FIND and its subroutines do most of the work
FIND:   SETZM   MATCH                   ;count of matches so far
LOOP:   CALL    GETLIN                  ;read one line from the file
         RET                            ;end of file.  return now.
        MOVE    A,[POINT 7,IBUF]        ;pointer to input data
        CALL    LOOK                    ;look for a match in the line.
         JRST   LOOP                    ;not found
FOUND:  AOS     MATCH                   ;found.  count a match
        HRROI   B,IBUF                  ;pointer to the input line
        CALL    DOOUT                   ;output the relevant line
        JRST    LOOP

;Routine to read one line from the input file into IBUF.
GETLIN: HRRZ    A,IJFN                  ;read one line from the file
        HRROI   B,IBUF                  ;into IBUF
        MOVEI   C,BUFLEN*5-1            ;maximum size of buffer
        MOVEI   D,12                    ;end of string character
        SIN                             ;gobble string
         ERJMP  EOFTST                  ;possible eof
        MOVEI   A,0                     ;make sure input string ends with null
        IDPB    A,B                     ;(for SOUT in DOOUT; called from FOUND)
        JRST    CPOPJ1
```

```
EOFTST: HRRZ    A,IJFN                  ;here on error from SIN,
        GTSTS                           ;get file status, check for eof
        TXNN    B,GS%EOF                ;end of file here?
         JRST   ERROR                   ;no.  some other error
        RET                             ;non-skip return from FIND

;Search one line of the input file for a match.
;Enter with: A/ Byte pointer to entire input line
;            BUFFER/ search string
LOOK:   MOVE    B,A                     ;copy pointer to input string
        MOVE    C,[POINT 7,BUFFER]      ;pointer to search string
SLOOP1: ILDB    X,C                     ;get char from search string
        JUMPE   X,CPOPJ1                ;end of search string = win
        CALL    GTINCH                  ;get a character from input string
         RET                            ;end of line = lose
        CAMN    W,X                     ;are characters the same?
         JRST   SLOOP1                  ;yes.  advance to next pair
        IBP     A                       ;no. advance to next character of
        JRST    LOOK                    ; of the input line and try again

;Get one character from the input line.
;Call with a byte pointer to the input line in B.  Return character in W.
;Skip unless end of line.  Lower-case is converted to upper-case.
GTINCH: ILDB    W,B                     ;Get a character from input file
        CAIE    W,12                    ;test for either end of line character
         CAIN   W,15
          RET                           ;End of line.  Non-skip return.
        JUMPE   W,CPOPJ                 ;end if null.  (A long line was input)
        CAIL    W,"a"                   ;test for lower-case
         CAILE  W,"z"
          TRNA                          ;not lower-case: skip
           TRZ  W,40                    ;convert to upper-case
CPOPJ1: AOS     (P)                     ;Perform skip return
CPOPJ:  RET
```

```
;This is the cleanup.  Close the files, write the results.
FIN:    HRRZ    A,IJFN                  ;close input file
        CLOSF
         ERJMP  ERROR
        HRROI   B,[ASCIZ/
There were /]
        MOVE    C,MATCH                 ;number of matches found
        CAIN    C,1                     ;special for 1 match
         JRST   ONEMAT                  ;special
        CALL    DOOUT                   ;other than 1 match
        SKIPG   B,MATCH                 ;count of matches.  2 or more?
         JRST   ZMATCH                  ;no matches found
        MOVEI   C,^D10                  ;decimal radix
        HRRZ    A,OJFN
        NOUT
         ERJMP  ERROR
FINMES: HRROI   B,[ASCIZ/ matches found in the file
/]
FINMS1: CALL    DOOUT
        HRRZ    A,OJFN                  ;close and release the output JFN
        CLOSF
         ERJMP  ERROR
        RET

ZMATCH: HRROI   B,[ASCIZ/no/]
        CALL    DOOUT
        JRST    FINMES

ONEMAT: HRROI   B,[ASCIZ/
There was one match found in the file
/]
        JRST    FINMS1

;Error handling
ERROR:  CALL    ERPRIN                  ;Print an error message
        HALTF
        JRST    START

ERPRIN: HRROI   A,[ASCIZ/Error: /]
        ESOUT                           ;start of error message.
        MOVEI   A,.PRIOU                ;output error message to terminal
        HRLOI   B,.FHSLF                ;this fork, most recent error
        MOVEI   C,0                     ;no limit to byte count
        ERSTR                           ;convert error number to a string
         TRN                            ;would you believe TWO error returns?
         TRN
        HRROI   A,CRLF                  ;end message with end of line
        PSOUT
        RET
```

```
;Get input file name from terminal, open it for 7-bit bytes.  JFN in IJFN
GTINPF: HRROI   A,[ASCIZ/File name for input: /]
        PSOUT
        MOVX    A,<GJ%OLD!GJ%SHT!GJ%FNS!GJ%CFM> ;Short form, old file
        MOVE    B,[.PRIIN,,.PRIOU]      ;scan for file name from these JFNs
        SETZM   IJFN                    ;initially, no JFN
        GTJFN
         ERJMP  GTINER                  ;special handling for error
        MOVEM   A,IJFN                  ;save the jfn
        HRRZ    A,A                     ;keep only right half of the JFN
        MOVE    B,[070000,,OF%RD]       ;7 bit bytes, read access
        OPENF
         ERJMP  GTINE1                  ;can't open file. handle error
        JRST    CPOPJ1                  ;return with a skip

GTINER: MOVEI   A,.FHSLF                ;error from GTJFN
        GETER                           ;get the error number
        HRRZ    B,B                     ;keep only the error number
        CAIN    B,GJFX33                ;"filename was not specified"?
         RET                            ;user typed return: exit now
GTINE1: CALL    ERPRIN                  ;print the name of this error
        HRRZ    A,IJFN                  ;get just the JFN number
        JUMPE   A,GTINPF                ;no jfn was gotten. try again
        RLJFN                           ;release jfn that was gotten
         ERJMP  .+1                     ;ignore errors here.
        JRST    GTINPF                  ;make another attempt

;Get output file name from terminal, open it for 7-bit bytes.  JFN in OJFN
GTOUTF: HRROI   A,[ASCIZ/File name for output: /]
        PSOUT
        MOVX    A,<GJ%FOU!GJ%FNS!GJ%SHT!GJ%CFM> ;Short form of GTJFN,
                                        ;output file gets next generation num
                                        ;AC2 is JFN for reading the file name
                                        ;Require confirmation if user uses
                                        ;file name recognition
        MOVE    B,[.PRIIN,,.PRIOU]      ;read file name from terminal
        SETZM   OJFN                    ;initially, no JFN
        GTJFN
         ERJMP  GTOTER                  ;error
        MOVEM   A,OJFN                  ;save output JFN
        HRRZ    A,A                     ;Right half of JFN only
        MOVE    B,[070000,,OF%WR]       ;7 bit bytes, write access
        OPENF
         ERJMP  GTOTER                  ;error
        RET

GTOTER: CALL    ERPRIN                  ;print the name of this error
        HRRZ    A,OJFN                  ;get just the JFN number
        JUMPE   A,GTOUTF                ;no jfn was gotten. try again
        RLJFN                           ;release jfn that was gotten
         ERJMP  .+1                     ;ignore errors here.
        JRST    GTOUTF                  ;make another attempt
```

```
;get search string from terminal.  put it in BUFFER.
GTSTRG: HRROI   A,[ASCIZ/Type the search string on one line:
*/]
        PSOUT                           ;note that the reprompt
        HRROI   A,BUFFER                ;is different from the prompt
        MOVEI   B,BUFLEN*5-1
        HRROI   C,[ASCIZ/*/]            ;reprompt
        RDTTY
         ERJMP  ERROR
;Convert lower-case to upper.  Place null at end of string.
        MOVE    B,[POINT 7,BUFFER]      ;pointer to start of string
GTSSCN: ILDB    A,B                     ;get a byte from the search string
        CAIE    A,12                    ;is this the end of the string?
         CAIN   A,15                    ;or this?
          MOVEI A,0                     ;either CR or LF is made into null
        CAIL    A,"a"                   ;lower-case
         CAILE  A,"z"                   ;
          TRNA                          ;
        TRZ     A,40                    ;is transformed to upper-case
        DPB     A,B                     ;the transformed byte is stored
        JUMPN   A,GTSSCN                ;and we loop, unless we stored
        RET                             ;the null to mark the end of the string

;Write heading to the output file
HEADER: HRROI   B,[ASCIZ/
Search of /]
        CALL    DOOUT                   ;send "search of"
        HRRZ    A,OJFN                  ;destination designator, a JFN
        HRRZ    B,IJFN                  ;JFN of the input file
        MOVEI   C,0                     ;default format
        JFNS                            ;write file name here
         ERJMP  ERROR
        HRROI   B,[ASCIZ/ for
/]
        CALL    DOOUT                   ;"for"
        HRROI   B,BUFFER                ;send the text of the
        CALL    DOOUT                   ;search string
        HRROI   B,[ASCIZ/
-----------------------------------------------------------------------
/]                                      ;a separation
DOOUT:  HRRZ    A,OJFN                  ;Output text.  call with B=source des.
        MOVEI   C,0                     ;string stops with a zero.
        SOUT
         ERJMP  ERROR
        RET

CRLF:   BYTE (7) 15,12
.ENDPS
```

```
        .PSECT  DATA,1001000
PDLEN==100
BUFLEN==100
PDLIST: BLOCK   PDLEN
BUFFER: BLOCK   BUFLEN
IBUF:   BLOCK   BUFLEN
MATCH:  0                               ;count of the number of matches
IJFN:   0                               ;input jfn
OJFN:   0                               ;output jfn

        END     START
```

Now we must discuss some of the routines in detail.

22.5.2 GTINPF – Get Input File

This subroutine prompts for an input JFN. It obtains the JFN from the terminal and opens the input file. This is our first example of doing file input, so several details must be mentioned.

First, the flag bits in GTJFN are somewhat different than the cases in which an output file was wanted. The bit GJ%SHT still means the short form of GTJFN. The bit GJ%OLD indicates that the JFN to be gotten must correspond to an old, i.e., already existing, file. GJ%CFM means that when the user types an escape character to request file name recognition, then the program will ask for a confirming carriage return before proceeding.[2]

As was mentioned when we first discussed GTJFN, the GJ%FNS bit means that register 2 contains an input JFN from which the file name will be read, and an output JFN to which the name and any recognition typeout will be sent. In this case, register 2 will be set up to contain .PRIIN and .PRIOU as the input and output JFNs, respectively. These predefined JFNs are for primary input and output; usually these are the terminal.

The effect of the GTJFN JSYS will be to scan for a file name from the terminal. The JFN corresponding to the file name that is typed in will be returned in register A. The JFN is stored in IJFN (for input JFN).

```
GTINPF: HRROI   A,[ASCIZ/File name for input: /]
        PSOUT
        MOVX    A,<GJ%OLD!GJ%SHT!GJ%FNS!GJ%CFM> ;Short form, old file
                                                 ;file name from jfn
        MOVE    B,[.PRIIN,,.PRIOU]
        SETZM   IJFN                             ;initially, no JFN
        GTJFN
         ERJMP  GTINER                           ;error from GTJFN
        MOVEM   A,IJFN                           ;save the jfn
```

After obtaining the JFN, the GTINPF routine will open the input file. This call to OPENF is similar to the one seen in previous examples. The difference is that the bit OF%RD is used instead of OF%WR. The OF%RD bit signifies our desire to read this file.

[2] If GJ%CFM is not set and escape is typed, the GTJFN will return the JFN immediately; this does not give the user an opportunity to respond to TOPS–20's selection of a file name.

The EDIT program is one of the standard editors that DEC supplies with TOPS–20. EDIT adds line numbers to the text of each file that it edits. These line numbers are used during editing to help the user find his or her way around in the file. TOPS–20 has a useful feature when dealing with such files: TOPS–20 can filter these line numbers out of the input stream. Therefore unless a program goes to extra trouble to see them, EDIT line numbers will not be seen when the SIN JSYS is used to read the file.[3] This is a very useful feature: it allows us to omit writing a code fragment to discard these numbers. Moreover, this code fragment is omitted from most programs that read files via the SIN JSYS.[4]

The code for this portion of GTINPF looks like this:

```
HRRZ    A,A                     ;keep only right half of JFN
MOVE    B,[070000,,OF%RD]       ;7 bit bytes, read access
OPENF
 ERJMP  GTINE1                  ;can't open file
JRST    CPOPJ1                  ;successful return
```

When reading file names from the terminal, the programmer must be aware that the user is fallible: he or she may not type a valid file name every time. By careful programming, we will increase this program's ability to cope with errors from the user. This generally will make the program easier to use. In case of an error from GTJFN the program will jump to GTINER; if an error occurs in OPENF, the program skips some of the code in GTINER by jumping to GTINE1.

At GTINER the program checks for a specific error called GJFX33. This error means "file name was not specified." This error occurs when, for example, the user types carriage return instead of a file name. If this error occurs, we interpret the error to mean that the user wants to exit from the program. For all other errors, including any errors from OPENF, the program will call ERPRIN (the error print subroutine) to print an informative message. The input JFN, if present, is released and the program loops to GTINPF to allow the user another chance to specify the file name.

In order to determine whether the error was GJFX33, the GETER JSYS is used. This JSYS returns in register 2 the error code corresponding to the most recent error in the process.

```
GTINER: MOVEI   A,.FHSLF                ;error from GTJFN
        GETER                           ;get the error number
        HRRZ    B,B                     ;keep only the error number
        CAIN    B,GJFX33                ;"filename was not specified"?
         RET                            ;user typed return: exit now
GTINE1: CALL    ERPRIN                  ;print the name of this error
        HRRZ    A,IJFN                  ;get just the JFN number
        JUMPE   A,GTINPF                ;no jfn was gotten. try again
        RLJFN                           ;release jfn that was gotten
         ERJMP  .+1                     ;ignore errors here.
        JRST    GTINPF                  ;make another attempt
```

The GTOUTF routine is very nearly the same as GTINPF. GTJFN flag bits appropriate to output have been used. In OPENF, as we have seen before, the bit OF%WR signifies that write access is requested.

One of the disadvantages of the short form of GTJFN is that it does not provide for a reprompt in

[3] If you want to see these numbers, set the bit OF%PLN in the call to OPENF.

[4] For those of you who are desperate to know how to omit line numbers, we shall provide the code in Section 25.1.3.

case CTRL/U or CTRL/R is typed.[5] When a program is to be used by a large number of people, it would be worthwhile to make the user interface, including prompts, reprompts, and error messages, as perfect as possible.

We have not yet reached the point in these examples where we want to produce such carefully crafted programs; to do so would distract us from our present purpose. So long as the programs you write serve yourself alone, some amount of sloppiness in the terminal interactions is tolerable. When you write programs to serve others, you will discover that highest standards of perfection are demanded in the command processing and error handling portions of a program. In Section 26 we will discuss how to construct better user interfaces.

22.5.3 GTSTRG – Get Search String

The GTSTRG routine reads a search string from the terminal. The search string is placed in the area called BUFFER. This routine provides an example of something we mentioned in our earliest discussion of RDTTY: the first prompt for input is more than one line. In such a case, the CTRL/R reprompt should consist of only those characters that appear on the last line of the original prompt.

```
GTSTRG: HRROI   A,[ASCIZ/Type the search string on one line:
*/]
        PSOUT                           ;note that the reprompt
        HRROI   A,BUFFER                ;is different from the prompt
        MOVEI   B,BUFLEN*5-1
        HRROI   C,[ASCIZ/*/]            ;reprompt
        RDTTY
         ERJMP  ERROR
```

After obtaining the search string, the GTSTRG routine converts the search text to upper–case. Whatever end–of–line character the user typed is replaced at this time with a null. This null will be important later on. This fragment is similar to those we have seen in earlier examples of terminal input processing.

```
;Convert lower-case to upper.  Place null at end of string.
        MOVE    B,[POINT 7,BUFFER]      ;pointer to start of string
GTSSCN: ILDB    A,B                     ;get a byte from search string
        CAIE    A,12                    ;is this the end of string?
         CAIN    A,15                   ;or this?
          MOVEI   A,0                   ;change CR or LF to null
        CAIL    A,"a"                   ;lower case?
         CAILE   A,"z"                  ;lower case?
          TRNA                          ;No.
           TRZ  A,40                    ;is transformed to upper-case
        DPB     A,B                     ;store the (transformed) byte
        JUMPN   A,GTSSCN                ;loop, unless we stored null
        RET                             ;to mark the end the string
```

[5] Reprompt is provided in the *long form* of GTJFN.

22.5.4 HEADER

The HEADER routine sends the first string to the output file. This string contains a heading explaining what file is being searched and for what key. HEADER starts things by sending the string ``Search of '' to the output file via the DOOUT routine. The DOOUT routine contains just the necessary setup for a SOUT JSYS to send a string to the output file.

```
HEADER: HRROI   B,[ASCIZ/
Search of /]
        CALL    DOOUT
```

Next, HEADER adds the name of the input file to the output stream by using the JFNS, *JFN to String*, JSYS. JFNS converts a JFN to a string that contains the name of the file corresponding to the JFN. In JFNS, register A is set up with a destination designator. The destination designator could be a string pointer, or, as in this example, it may be an output JFN. Then register B is set up with the JFN whose file name we want. Register C holds format control flags; zero defaults to a reasonable format.

This instance of the JFNS JSYS will send the name of the file corresponding to IJFN to the output file:

```
HRRZ    A,OJFN                  ;destination designator, a JFN
HRRZ    B,IJFN                  ;JFN of the input file
MOVEI   C,0                     ;default format
JFNS                            ;write file name here
 ERJMP   ERROR
```

HEADER continues by sending the string "for", and a new line to the output file. The text of the search string, from BUFFER, is also sent to the output file. HEADER exits by falling into DOOUT, which sends the final string consisting of a series of hyphens to separate the heading from the output. DOOUT then returns to the caller of HEADER.

22.5.5 FIND

The subroutine FIND depends on subroutines to do most of the work; the structure of FIND is essentially simple:

```
FIND:   SETZM   MATCH                   ;count of matches so far
LOOP:   CALL    GETLIN                  ;read one line from the file
         RET                            ;end of file.  return now.
        MOVE    A,[POINT 7,IBUF]        ;pointer to input data
        CALL    LOOK                    ;look for a match in the line.
         JRST    LOOP                   ;not found
FOUND:  AOS     MATCH                   ;found.  count a match
        HRROI   B,IBUF                  ;pointer to the input line
        CALL    DOOUT                   ;output the relevant line
        JRST    LOOP
```

The variable called MATCH is initialized to zero; in this cell, the program will count the number of

times the search string is found in the input file. The call to GETLIN at LOOP will read a line from the file. GETLIN will skip unless the end of file has been seen. When GETLIN skips, a new line from the input file will be present in IBUF.

The LOOK subroutine will test the input line to see if any match can be found. LOOK requires that accumulator A be set up to contain a byte pointer to the line from the input file. If LOOK fails to find a match on the current line, it will return without skipping. The instruction JRST LOOP following the call to LOOK brings the program back to LOOP where another line will be read. When LOOK finds a match, it will perform a skip return. At FOUND, the counter called MATCH is incremented. Then the input line that contains this match is copied from IBUF to the output file. The program continues at LOOP, seeking more input.

At LOOP, eventually the call to GETLIN will result in the end of file being detected. GETLIN will avoid the skip return; the RET instruction at LOOP+1 will return to the main program.

22.5.6 GETLIN, EOFTST, and the SIN JSYS

The GETLIN routine is where the actual file input occurs. GETLIN will use the SIN JSYS to read a line from the file into the buffer called IBUF. The SIN JSYS requires that register 1 (which we call A) contain the JFN of the input file. For us, that is IJFN. Register 2 (our B) is set up with a destination designator. We use the familiar HRROI instruction to establish IBUF as the destination.

In this case we will set register 3 (our C) to the positive length of the input buffer (measured in number of characters). Because register 3 is positive, the SIN JSYS will look in register 4 for the *break character*. Input will terminate either when the buffer area fills up, when the break character is seen in the input stream, or when end of file is reached. We will use line feed, octal 12, as the break character. Our set up and call to the SIN JSYS looks like this:

```
GETLIN: HRRZ    A,IJFN                  ;read one line from the file
        HRROI   B,IBUF                  ;into IBUF
        MOVEI   C,BUFLEN*5-1            ;maximum size of buffer
        MOVEI   D,12                    ;end of string character
        SIN                             ;gobble string
         ERJMP  EOFTST                  ;possible eof
```

In the normal course of events, the SIN JSYS will cause text to be read into IBUF. Register 2 (B) will be updated to point to the line feed character that terminates the input. Register 3 (C) will be updated (towards zero) to account for the number of characters that were stored in the buffer.

If all goes well, the ERJMP following the SIN will be avoided. Using the updated byte pointer in register B, a zero is deposited into the buffer following the line feed. This zero terminates the string in the buffer. This zero is necessary because if a match is found, this line will be output via a SOUT that expects the null to be present. Finally, GETLIN performs a skip return:

```
        MOVEI   A,0                     ;make input string end w/ null
        IDPB    A,B                     ;(for SOUT in DOOUT,
        JRST    CPOPJ1                  ;           called from FOUND)
```

If a line is too long to fit in the buffer, as much as will fit is placed there. The rest of the line will appear the next time GETLIN is called. It must be acknowledged that this example program will not handle long lines correctly: it fails to match a string that is split across buffers in this way.

A variety of problems may occur when input is attempted. One problem that we must deal with is the eventual end of file. Other problems may arise such as hardware–detected data errors, improper JSYS arguments, etc. In any of these cases, the SIN will fail and the ERJMP will jump to EOFTST.

At EOFTST, we are hoping that the problem is simply the end of file condition. The GTSTS JSYS is used to query the status of the JFN. GTSTS requires that we load the JFN into register 1 (our A). The status is returned in register 2. The left–half bit, GS%EOF, will be set in register 2 (our B) if the end of file has been reached.

We test for this bit via the TXNN macro which, becase GS%EOF is defined in the left half, generates TLNN. If TXNN skips, we have reached the end of file; EOFTST will perform the non–skip return from GETLIN. If the problem is anything other than end of file, EOFTST jumps to the catch–all error handler.

```
EOFTST: HRRZ    A,IJFN                  ;here on error from SIN,
        GTSTS                           ;get file status,
        TXNN    B,GS%EOF                ;at end of file here?
         JRST   ERROR                   ;no.  some other error
        RET                             ;return from FIND
```

We must note here that there is another deficiency in this program. Although it is conventional in text files to place a carriage return and line feed sequence at the end of each line, some editors (such as EMACS allow text files to be created in which the last line omits the CRLF. We have noted already that MACRO requires that the END pseudo–op be followed with a CRLF. This example is another program that expects the last line to end with a CRLF.

This behavior is explained as follows. If a file ends with a line feed, when SIN reads the last line it will stop at the line feed. The next SIN will start after that line feed and read nothing because there are no more characters. The end of file will be detected by EOFTST and the program will stop trying to read the file. Contrast this behavior to the situation where the last character in the file is not a line feed. Some previous SIN has read up to the last line feed. The next (and last) SIN will read the last line; because there is no line feed, SIN will not stop until it has read past the end of the file. Then, EOFTST will determine that the end of file has been reached, and GETLIN will return without allowing the last line to be processed. (There are ways to avoid this problem that we will demonstrate in a later example.)

22.5.7 LOOK and GTINCH

Before we discuss the details of LOOK, let us briefly mention the GTINCH, *GeT INput CHaracter*, subroutine. GTINCH is another example of the kind of character processing that we have seen in terminal input examples. Using register B as a byte pointer to the input line, GTINCH loads a byte into register W. If the end of the input line is found, GTINCH performs a non–skip return. Lower–case characters are converted to upper–case.[6]

[6]TOPS–20 conventionally treats “foo”, “Foo”, and ‘FOO” as if they are all the same word, i.e., differences only in letter case are insignificant. Contrast to a UNIX system, in which those three strings are unrelated. When building programs in the TOPS–20 environment, we try to adhere to TOPS–20 conventions.

22.5.7.1 LOOK

The LOOK subroutine accomplishes one of the major purposes of this program. LOOK determines whether the search string is present in one line of input.

```
LOOK:   MOVE    B,A                     ;copy pointer to input string
        MOVE    C,[POINT 7,BUFFER]      ;pointer to search string
SLOOP1: ILDB    X,C                     ;get char from search string
        JUMPE   X,CPOPJ1                ;end of search string = win
        CALL    GTINCH                  ;get a char from input string
         RET                            ;end of line = lose
        CAMN    W,X                     ;are characters the same?
         JRST   SLOOP1                  ;yes.  advance to next pair
        IBP     A                       ;no. advance to next character
        JRST    LOOK                    ;  of input line and try again
```

When LOOK is first called register A points to the beginning of the input line. As the search progresses, A advances. At each point in LOOK, register A contains a pointer to what we hope is the beginning of a match. As each potential match is discarded, A is advanced to try again on the remainder of the input line. All input characters that are to the left of where A points have already been examined; no match was found. LOOK copies the byte pointer from A to B; B will be used as a temporary pointer to see if the string that starts at A matches the search string. Register C is initialized to be a byte pointer to the search string.

The search really begins at SLOOP1. A character is loaded into X from the search string. The end of the search string is indicated by a null byte. If the search string becomes exhausted, we have found a match. The LOOK routine exits via CPOPJ1, indicating that a match was found. Assuming we have not yet gotten to the end of the search string, the code at SLOOP1 continues by calling GTINCH for a character from the input line. When GTINCH exhausts the input line, it returns without skipping. If we find the end of the line, we have failed to find a match. LOOK returns without skipping.

If there are characters remaining in the input line, GTINCH has returned one in W. Now, if W and X are the same then the first character of the input string matches the first character of the search string. In this case, the program loops to SLOOP1. At SLOOP1, as we discussed already, the program progresses along the search string (byte pointer in C) and along the input line (pointer in B). Unless the CAMN instruction detects a difference between the characters, both the search pointer and the input line pointer will advance. A match is found when the search pointer runs off the end of the search string.

When the CAMN instruction detects that the characters in W and X are different, then the string that starts at A does not match the search string. Therefore, we must advance A. This is accomplished by an IBP instruction. The program continues by looping back to LOOK, where we start once more at the front of the search string.

At this point, perhaps an example is needed. Let us suppose that the search string (at BUFFER) consists of the word FIND. Also, let the input line (in IBUF) be the string "FINISH FINDING".

At the first call to LOOK, register A is set to point to the string "FINISH FINDING". Register B will be copied from A; C will be set to point to FIND.[7]

[7] In the diagrams that follow, the arrows indicate the next character that will be read by the indicated byte pointer. It would be more accurate (but perhaps more confusing) to show all pointers addressing the previous character, because all characters are read via ILDB instructions.

```
              C                      B
              ↓                      ↓
BUFFER:       FIND          IBUF:    FINISH FINDING
                                     ↑
                                     A
```

At SLOOP1, X will be loaded with the character F from FIND; W will be loaded with the F from FINISH. Because these characters are identical, the program will jump back to SLOOP1. Note that C and B are advanced past these characters:

```
               C                      B
               ↓                      ↓
BUFFER:        FIND          IBUF:    FINISH FINDING
                                      ↑
                                      A
```

At SLOOP1 for the second time, characters are taken from the search string and from the input line. The characters, I, are identical once more; the program finds itself at SLOOP1 again. Yet again, the characters, N, are identical. At SLOOP1 for the fourth time the picture now stands

```
                 C                      B
                 ↓                      ↓
BUFFER:          FIND          IBUF:    FINISH FINDING
                                        ↑
                                        A
```

This time, a D is taken from the search string, but the character I is found in the input line. This difference causes the CAMN instruction to skip; A will be advanced. At LOOK, B is then copied from A; register C is reset to the beginning of the search string. At SLOOP1 again, the picture is

```
              C                       B
              ↓                       ↓
BUFFER:       FIND          IBUF:     FINISH FINDING
                                      ↑
                                      A
```

The F in FIND doesn't match the I in FINISH so the program again advances the input pointer in A and jumps to LOOK. At LOOK, B is copied from A and C is set to point to the front of the search string:

```
              C                       B
              ↓                       ↓
BUFFER:       FIND          IBUF:     FINISH FINDING
                                        ↑
                                        A
```

This process, incrementing A, is repeated. Eventually, at SLOOP1, A points to the F in FINDING.

```
              C                              B
              ↓                              ↓
BUFFER:       FIND           IBUF:   FINISH FINDING
                                             ↑
                                             A
```

The F in FIND matches the one in FINDING, so the program finds itself at SLOOP1, without having advanced A. Registers B and C have been advanced to point to the I characters.

```
               C                              B
               ↓                              ↓
BUFFER:        FIND           IBUF:   FINISH FINDING
                                             ↑
                                             A
```

The program continues looping at SLOOP1. After the D in FIND has matched the one in FINDING, the program returns again to SLOOP1. The picture looks like

```
                 C                                B
                 ↓                                ↓
BUFFER:          FIND           IBUF:   FINISH FINDING
                                             ↑
                                             A
```

This time, following the ILDB X,C, the program will discover that it has run off the end of the search string. Because the end of the search string has been reached, the program knows that the characters starting at the point described by A match the search string. The LOOK routine returns with a skip.

It should be evident that if the program fails to find a match on a line, then eventually B runs off the end of the input line. When that happens, the LOOK routine returns without skipping.

22.5.7.2 An Optimization of LOOK

The LOOK subroutine that was discussed above was written for simplicity. By carefully examining the way that routine works, we can find some opportunities to optimize the loop. The loop in which LOOK seeks the first matching character is 8 instructions, including a call to GTINCH. Some of these instructions can be removed by restructuring the program.

In the version of LOOK that follows, a new loop, LOOK1 has been added to speed the search for an input character to match the first character of the search string. The search for the first matching character has been reduced to three instructions, including a call to the same subroutine. An additional accumulator, called Y, is needed:

```
LOOK:   LDB     Y,[POINT 7,BUFFER,6]    ;first character from search string
        JUMPE   Y,CPOPJ1                ;an empty search string matches anything
LOOK0:  MOVE    B,A                     ;copy pointer to input string
LOOK1:  CALL    GTINCH                  ;get a character from the input string
         RET                            ;end of input means no match found.
        CAME    W,Y                     ;does input match the first search chr?
         JRST    LOOK1                  ;no.
        MOVE    A,B                     ;first ch. matches.  copy pointer to A
MOVE    C,[POINT 7,BUFFER,6]    ;Pointer so ILDB gets second char
SLOOP1: ILDB    X,C                     ;get char from search string
        JUMPE   X,CPOPJ1                ;end of search string = win
        CALL    GTINCH                  ;get a character from input string
         RET                            ;end of line = lose
        CAMN    W,X                     ;are characters the same?
         JRST   SLOOP1                  ;yes.  advance to next pair
        JRST    LOOK0                   ;No.  Seek new start of match string.
```

For utmost speed in searches different techniques can be applied. These techniques are beyond the scope of this book.

22.5.8 FIN – Finish Routine

The `FIN` routine is responsible for writing a trailer into the output file and closing the JFNs. First, the input JFN is closed (and released). This is done, as for an output JFN, just by using the `CLOSF` JSYS. Next, a trailer is written into the output file. The trailer will say one of three possible messages:

```
There were no matches found in the file

There was one match found in the file

There were nn matches found in the file
```

In the case of more than one match, a decimal number is output to the file where "`nn`" appears above. We could add the `DECOUT` routine to this program; instead, we choose to use the `NOUT`, *Numeric OUTput*, JSYS for this purpose.

`NOUT` is used for fixed point numeric output. Register 1 (our `A`) must be set up with a destination designator; in this case we use the output JFN. Register 2 (`B`) is loaded with the number to output. Register 3 specifies the format, field width, and radix. In this case we specify only that the radix is decimal. The following example of `NOUT` is similar to what appears in this program:

```
        HRRZ    A,OJFN                  ;destination designator
        MOVE    B,MATCH                 ;count of matches.
        MOVEI   C,^D10                  ;base 10, no control flags
        NOUT
         ERJMP   ERROR
```

After writing the concluding messages, `FIN` closes the output JFN and returns.

22.6 Exercises

22.6.1 Maze

Simulate the movement of a rat searching for food in a maze. A maze will consist of a rectangular grid of squares much like a checkerboard. Each square is either a wall or not a wall. Exactly one square in the maze is designated as the start; exactly one square is designated as the finish.

The object of the program is to find a path for the rat to take from the starting square to the finishing square. This path must contain only adjacent squares that are not walls. The rat can move horizontally and vertically, but it cannot move diagonally.

Such a path is called a solution to the maze. The solution can be described as a sequence of squares in which the starting square is first, the finishing square is last, and in which all the other squares in the sequence are connected by either a horizontal or a vertical path to both the previous square and to the next square in sequence.

You are required to find a path through the maze. It need not be the shortest path, but it must not contain any square twice.

The input will be a file that describes a sequence of mazes. Each maze will be described by a pair of numbers on the first line that defines the size of the maze (number of rows, number of columns). That line will be followed by a sequence of lines describing each row. Four possible characters will define each square in the maze:

```
blank   a "not wall"
W       a wall
S       the starting point
F       the finishing point.
```

Thus, a maze description might appear as

```
5 6
  WF
W WWW
    W
W W
 SW W
```

Notice that the maze is not surrounded by a border. This means that your program must check to see that it is not leaving the maze as it looks for a solution.

When you have solved the maze, your program should print the maze with the solution shown on the maze. You can draw the maze and your solution any way that you want, but be sure that the output is readily understandable. A border should be printed around the maze. An acceptable example of a solution and the output for this maze is

```
BBBBBBBB
B  WF**B
BW WWW*B
B ***W*B
BW*W***B
B SW W B
BBBBBBBB
```

Asterisks have been used to indicate the solution path.

It is possible that a given maze has no solution. In such a case you should print a message to that effect, and print the maze without any solution indicated.

Your instructor will tell you where to find the data for this program. The data file will contain several mazes. End of file will indicate the end of the data for mazes. You may assume that no maze will be larger than 20×20.

Hint: in order to find a solution for the maze, your program should begin at the starting square and successively examine neighboring squares. It should never visit a square that it has visited before. Therefore, you must keep track of which squares you have visited. The program must also keep track of the path over which it has moved. If you hit a dead end, back up until you find another a new path to investigate. This is only a suggestion; you may do this problem any way that you want.

22.6.2 Saddle Points in an Array

A matrix is said to have a *saddle point* if some position is the smallest value in its row and the largest value in its column.

You will be given a data file that contains several problem sets. Each set will begin with the integer dimensions `I` and `J` of the matrix. These will be followed by the necessary number of integer data items, presented such that `A[1,1]` is first, followed by `A[1,2]` ... `A[1,J]`, `A[2,1]`, etc.

Print the matrix. Locate all saddle points (if any) within the array. Print a list of the locations of the saddle points.

The end of the input file signifies the end of the problem sets.

22.6.3 Crossword Puzzle

The object of this exercise is to write a program to "draw" a crossword puzzle diagram.[8] This is an example of an application where the problem of correctly formatting the output is nearly as challenging as the other computations.

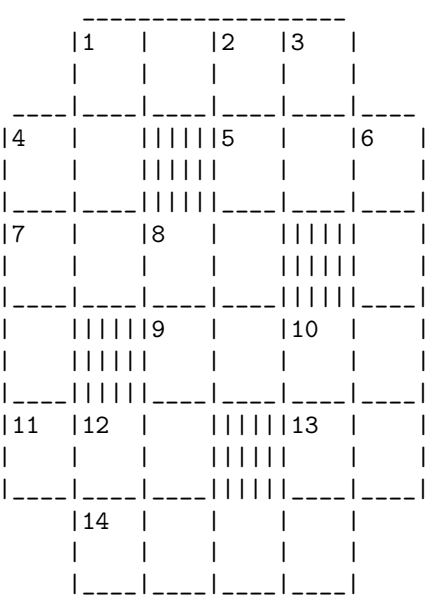
You will be given as input a matrix of zeros and ones. An entry of zero indicates a white square; a one indicates a black square. The output should be the diagram of the puzzle, with the appropriate white squares numbered for words *across* and *down*.

For example, given the matrix

[8] This exercised is adapted from [KNUTH1, 1.3.2 #23.], with permission from Addison Wesley.

```
1 0 0 0 0 1
0 0 1 0 0 0
0 0 0 0 1 0
0 1 0 0 0 0
0 0 0 1 0 0
1 0 0 0 0 1
```

the corresponding puzzle diagram would be as shown in the following figure:



A square is numbered if it is a white square and either (a) the square below is white and there is no white square immediately above, or (b) the adjacent square on the right is white and there is no white square immediately to the left.

If black squares are given at the edges they should be removed from the diagram as illustrated in this example: the squares at the corners have been removed. A simple way to accomplish this is to artificially insert rows and columns of -1's at the top, bottom, and sides of the given input matrix, and then change every $+1$ that is adjacent to a -1 into -1 until no $+1$ remains that is next to any -1. Think carefully about the process by which you change black squares to border squares: avoid using an algorithm that is inefficient.

The following method should be used to print the final diagram: each box of puzzle should correspond to five columns and three rows of the output page. These positions should be filled with one of the following diagrams, as appropriate:

```
␣␣␣␣|     Unnumbered white squares
␣␣␣␣|     "␣" represents a blank space
____|

nn␣␣|         n␣␣␣|
␣␣␣␣|  or    ␣␣␣␣|     Numbered white squares
____|         ____|

|||||
|||||     Black squares
|||||
```

−1 Squares, Depending on neighboring −1 Squares:

```
␣␣␣␣␣      ␣␣␣␣|      ␣␣␣␣␣      ␣␣␣␣␣      ␣␣␣␣|
␣␣␣␣␣      ␣␣␣␣|      ␣␣␣␣␣      ␣␣␣␣␣      ␣␣␣␣|
␣␣␣␣␣      ␣␣␣␣|      _____      _____      ____|
```

Each puzzle will appear in the input data as a 23×23 matrix of zeros and ones. Each row of the matrix will be written on one line of an input file. Read 23 rows of data and print the crossword diagram. Repeat this until the end of file is reached. The first line of the file that made the example shown above was written as:

```
10000111111111111111111
```

Chapter 23

Directory Processing

23.1 Example 14 — File Directory and Sort

The program that follows is an example of how to read and process a file directory. Each file directory is itself a file in TOPS–20. Although the detailed structure of the directory file is somewhat complicated, the TOPS–20 operating system provides a convenient mechanism for stepping, file–by–file, through the contents of a directory.

This program will gather the file name and other information about each file in the directory. This information is collected in the main memory space of the program, in an area of memory that is dynamically allocated as the program runs. After all the information has been gathered, the program sorts the directory information by the date of each file's most recent reference. Finally, the program will print the sorted directory listing.

23.2 Directory Processing

The first requirement for reading a directory file is to obtain a special kind of JFN, called an *indexable file handle*, which names a collection of files. The collection of files in a directory is called by the name `*.*.*`; as a user of TOPS–20 you should be familiar with this example of a *wildcard file specification*. The indexable file handle is obtained by asking the `GTJFN` JSYS for a JFN for the name `*.*.*`. The indexable file handle differs from a normal JFN in two respects. First, it can be stepped from one file specification to another, via the `GNJFN` JSYS. Second, the left half of the JFN word returned by `GTJFN` contains flags that are necessary to `GNJFN`. We get the indexable file handle by executing the following fragment from the beginning of the `INIT` subroutine:

```
INIT:   MOVX    A,<GJ%OLD!GJ%IFG!GJ%SHT> ;return flags, permit wildcards
        HRROI   B,[ASCIZ/*.*.*/]          ;All file names, types, generations
        GTJFN                             ;Obtain the JFN
         ERJMP   ERROR
        MOVEM   A,DIRJFN                  ;save entire JFN and flags
```

The new feature of this call to `GTJFN` is the use of the flag called `GJ%IFG`. This control flag specifies that wildcards are acceptable in the name and that JFN flags should be returned by `GTJFN`. The

JFN flags are a necessary part of an indexable JFN. Register B contains a source pointer to the string *.*.*, which specifies all file names, all file types, and all generation numbers. In the normal course of events, GTJFN will return an indexable file handle in register A. The entire JFN, including the flags in the left halfword, is saved in the word called DIRJFN.

The file directory is stored in alphabetical order by file names and by file types. GTJFN returns a JFN to the very first of these names. We can obtain the name of the first file by executing the following fragment from the DOFILE subroutine:

```
DOFILE: HRROI   A,NAMBUF                ;Extract the file name from the JFN
        HRRZ    B,DIRJFN                ;output to NAMBUF.
        MOVEI   C,0                     ;default flags
        JFNS                            ;store the file name string in NAMBUF
         ERJMP   ERROR
```

The name of the first file will be written in NAMBUF by the JFNS JSYS. It is important to omit the JFN flags from register B in the call to JFNS; if you include them, you will get back the name *.*.*.

After each file has been processed, the JFN is stepped to the next file by using the GNJFN JSYS. This fragment is part of the main program:

```
LOOP:   CALL    DOFILE                  ;Perform functions for one file.
        MOVE    A,DIRJFN                ;step to the next file in sequence
        GNJFN                           ;note the full JFN & flags word is used
         JRST   NOMORE                  ;there are no more files
        JRST    LOOP                    ;we have another file, process it.

NOMORE: HRROI   A,CRLF
        PSOUT                           ;There are no more files to do.
```

The entire JFN word, including flags, is loaded into register A for the call to the GNJFN JSYS. GNJFN will step the JFN to make it point to the next file that matches the original file specification. Normally, GNJFN will skip. After all the files in the directory have been processed, GNJFN will release the JFN and return without skipping.

Two further aspects of GNJFN are important. First, if the JFN argument to GNJFN is not an indexable file handle, nothing bad will happen; GNJFN will just release the JFN and return without skipping. Second, GNJFN returns a flag word in register A. These flags contain information about which part of the file specification was stepped. There are flags to indicate that the generation number was stepped, that the file type was stepped, or that the file name was stepped; for details, consult [MCRM]. Some programs, notably the EXEC, make use of this information for formatting their directory output. It is important to note that this result word is *not* a new JFN; it should *not* be used to replace the original JFN obtained from GTJFN.

23.3 Dynamic Space Allocation

The DOFILE routine collects information about each file in a fixed location; that information is then moved to an area whose size grows as the program runs. In the preceding examples, we have dealt with static regions of memory whose size and extent were known when the program was being written. In this program we do not know the extent of the information that we must store; the

amount of storage depends on the number of files in the directory. Therefore, a *dynamic storage area* must be used to collect this data. A dynamic storage area or *free space area* is a storage region whose size or location is not known when the program is written.

Although we could guess at some maximum number of files, guessing is not a good idea. If there are fewer files than we guessed, much of this program's address space would be wasted. If the guess is too small, the program won't work. There are many situations where there is just too little information to make an intelligent guess beforehand. In such cases, dynamic allocation of storage is an important tool. Dynamic and flexible allocation of memory space is one of the particular attractions of assembly language programming; many high–level languages do not provide adequate tools for storage management.

In order to perform dynamic allocation, the programmer must identify an area of the program's address space where there isn't any other piece of program or data. The address of that area provides an initial boundary for the dynamic area. It is necessary for the program to keep track of the boundary and extent of its dynamic space; this is to allow the program to know what range of addresses it has stored data into.

LINK helps us identify such an area of our address space; However, details of the story differ depending on wheather or not the program is loaded into section zero.

23.3.1 Section Zero Memory Layout

When LINK finishes loading a program into section zero, it sets the right half of the job data area cell named `.JBSA` to be the starting address of the program. LINK places the first free address above the program in the left half of `.JBSA`.

It would be difficult for us to compute this number if `LINK` didn't give it to us. The address space that we use contains more than just our program. The job data area, any library subroutines, patch space, and our symbol table are all present in the address space. The number that LINK provides in the left half of `.JBSA` accounts for all these factors. LINK has loaded nothing at or above this address. (However, some section zero programs are partitioned, as if for TOPS–10, into a low segment and a high segment. In this case, the value in the left half of `.JBSA` defines the first available address in the low segment. The low segment should not attempt to grow past the start of the high segment.)

The value in the left half of `.JBSA` may be taken as the first free address available to the the program. Unless otherwise constrained, the free area extends to address `777777`. Among the constraints might be the existence of a high segment, the presence of the TOPS–10 compatibility package, the presence of DDT if the machine is not extended.

For a section zero program we would use the left half of `.JBSA` to initialize two variables, `FFLOC` and `FRELOC`. `FFLOC` is permanently kept as a copy of the origin of free space. `FRELOC` is used to indicate the next free address in free space. The initialization would be performed by the following fragment placed in the `INIT` subroutine:

```
HLRZ    A,.JBSA         ;Left half of .JBSA is the first
MOVEM   A,FFLOC         ;free location above the program and
MOVEM   A,FRELOC        ;symbols.
```

23.3.2 Memory Layout for Extended Adressing Programs

LINK creates a memory map of all the psects; it places the map following the Program Data Vector (PDV) and it stores a pointer in the PDV that supplies the address of the memory map. To find a free location for dynamic allocation, a programmer must locate the program's PDV, locate the map, and interpret the map.

These tasks are little off the point of the present demonstration, so we shall apply a slightly different tactic. We write the program using psects defined in section 1. Then, we allocate an entire section, section 2, for our dynamic area. Although the program is free to touch any page in its address space and have TOPS–20 create that page if it does not exist, TOPS–20 draws a line at creating a page in an address section that does not exist. Consequently, the programmer must make an explicit request to add each new address sections that it needs.

For this task, we shall use the SMAP% JSYS to create a private section 2. We shall then initialize FFLOC and FRELOC to be used as described above for the section zero program.

```
DSECT==2                        ;Define section for dynamic allocation

        SKIPE   A,FFLOC         ;Does FFLOC have a value already?
         JRST   INIT1           ;Yes, we must have done SMAP% previously
        MOVE    B,[.FHSLF,,DSECT]   ;Create private section
        MOVE    C,[SM%RD!SM%WR!1]   ;1 section with read and write access
SMAP%                   ;Request that TOPS-20 make the section
         ERJMP  ERROR            ;report an error
        MOVSI   A,DSECT          ;first address in the new section
        MOVEM   A,FFLOC          ;first address at which to allocate
INIT1:  MOVEM   A,FRELOC         ;first address not yet allocated
```

When we desire to use free space, the contents of FRELOC will indicate the first available word. After storing data into free space, we will update FRELOC to address the word beyond the last word that we used. Thus, when we next need to use free space, the updated FRELOC will account for our previous usage, and the program will avoid writing on top of good data that had previously been stored.

Often in other computer systems, the program must negotiate with the operating system to expand its memory size. In TOPS–20 that negotiation is not required on a page–by–page basis. Subject to the caution that you must avoid overwriting the program, the program may use any portion of its address space corresponding to existing sections. In TOPS–20, the program must explicitly create new sections.

23.4 Bubble Sort

The Bubble sort algorithm has been chosen in this example because it is one of the simplest sorts known. Unfortunately, Bubble sort is also one of the slowest sorts; we will present a faster but more complicated sort in Section 23.7.

Bubble sort works by comparing adjacent items in an array. If the first item, i.e, the one with the lower index, is less than or equal to the second item, then the program just goes on to the next pair of items (i.e., the second and third items). When a pair is found that is out of order, the items are interchanged and a flag is set to indicate that an interchange has occurred. Each interchange repairs one pair that is wrongly ordered. After the interchange, the program goes on to compare the next

pair.

One pass through the data is completed after the next–to–last and last items have been compared (and interchanged if necessary). At the end of each pass, the program tests to see if any interchanges have been made. If any interchanges were necessary, the program starts a new pass. It clears the flag that records interchanges and begins the next pass by comparing the first and second items. When the program reaches the end of a pass in which no interchanges have occurred, the data is sorted.

A Pascal procedure to do Bubble sort appears below:

```
CONST n = 50;

TYPE table = ARRAY[1..n] OF INTEGER;

PROCEDURE Bubblesort(VAR data:table);
        VAR: interchange:BOOLEAN;
             i,j: INTEGER;
        BEGIN
        REPEAT                          (* Start a pass *)
         interchange:=FALSE;            (* No interchanges yet *)
         FOR i:=1 TO n-1 DO             (* Compare each pair *)
            IF data[i] > data[i+1] THEN BEGIN
                j := data[i];           (* A pair is out of order *)
                data[i] := data[i+1];   (* So interchange the data *)
                data[i+1] := j;         (* And signal that an *)
                interchange := TRUE     (* interchange was made *)
                END
        UNTIL NOT interchange           (* repeat passes until no *)
        END;                            (* interchanges.  Then finished *)
```

We shall defer our detailed discussion of Bubble sort until after we have seen the entire program. One problem that we face in translating this algorithm into assembly language is that the records we are sorting have different lengths. Since Bubble sort requires an array of consecutive items, each the same size, the program that appears below actually sorts an array of *pointers* to the data records, rather than the data records themselves. To build the array of pointer records, the program calls the `BUILDA` subroutine; after `BUILDA` is finished, the array of record pointers is sorted by the `SORT` subroutine.

23.5 Directory I/O and Sort Program

The entire text of this example program is presented below. The detailed explanation of the special features of this program will follow.

The conditional assembly flag `EXADFL` would be set to zero for a section zero program, or to a non–zero value for the extended addressing version. This flag is referred to by the macros `EXTADR` and `ZADR`.

```
        TITLE   Directory I/O and Bubble Sort - Example 14

EXADFL==1                       ;Flag, zero for section 0, non-zero for NZS
DEFINE EXTADR <IFN EXADFL,>     ;macro to invoke extended addr code
DEFINE ZADR   <IFE EXADFL,>     ;macro to invoke section zero code.

        SEARCH  MONSYM,MACSYM
ZADR<
        EXTERN  .JBSA
>;ZADR

A=1
B=2
C=3
D=4
W=5
X=6
Y=7
Z=10
P=17

PDLEN==200

EXTADR<
DSECT==2                                ;Define section for dynamic allocation
        .PSECT  DATA,1001000
>;EXTADR
PDLIST: BLOCK   PDLEN
DIRJFN: 0
FDBADR: BLOCK   .FBLWR+1                ;array for reading the FDB
NAMBUF: BLOCK   50                      ;plenty of room for a file name
LINPTR: 0
FFLOC:  0
FRELOC: 0
TBLEND: 0
TBLSTT: 0
EXTADR<
        .ENDPS
        .PSECT  CODE/RONLY,1002000
>;EXTADR
```

```
;Main program
START:  RESET                           ;normalize all I/O activity
        MOVE    P,[IOWD PDLEN,PDLIST]   ;establish a stack
        HRROI   A,[ASCIZ/Directory and Sort
/]
        PSOUT                           ;send a friendly greeting
        CALL    INIT                    ;obtain JFN. Setup storage
;Loop through all the files in the directory
LOOP:   CALL    DOFILE                  ;Perform functions for one file.
;You might restore the next two lines to see the alphabetical listing
;       HRROI   A,NAMBUF                ;Print the name of this file.
;       PSOUT
;After each file, step to the next file.
        MOVE    A,DIRJFN                ;step to the next file in sequence
        GNJFN                           ;note the full JFN & flags word is used
         JRST   NOMORE                  ;there are no more files
        JRST    LOOP                    ;we have another file, process it.

;There are no no more files in the directory
NOMORE: HRROI   A,CRLF
        PSOUT
;Build an array of pointers to the data records kept in memory
        CALL    BUILDA                  ;Build the array
;Proceed to sort the array
        MOVE    A,TBLEND                ;Get the end of the array
        SUB     A,TBLSTT                ;minus start = number of things (N)
        JUMPE   A,STOP                  ;jump if table is empty.
        CALL    SORT                    ;sort things.  A/number of elements
;Print the sorted results
        CALL    PRNTBL                  ;print table
        JRST    STOP                    ;finish.
```

```
INIT:   MOVX    A,<GJ%OLD!GJ%IFG!GJ%SHT> ;return flags, permit wildcards
        HRROI   B,[ASCIZ/*.*.*/]
        GTJFN
         ERJMP  ERROR
        MOVEM   A,DIRJFN                ;save entire JFN and flags
ZADR<
;initialize the pointers to the free-space array
        HLRZ    A,.JBSA                 ;Left half of .JBSA is the first
        MOVEM   A,FFLOC                 ;free location above the program and
        MOVEM   A,FRELOC                ;symbols.
>;ZADR
EXTADR<
        SKIPE   A,FFLOC                 ;Does FFLOC have a value already?
         JRST   INIT1                   ;Yes, must have done SMAP% previously
        MOVE    B,[.FHSLF,,DSECT]       ;Create a private section
        MOVE    C,[SM%RD!SM%WR!1]     ;1 section with read and write access
        SMAP%                           ;Request that TOPS-20 make the section
         ERJMP  ERROR                   ;report an error
        MOVSI   A,DSECT                 ;first address in the new section
        MOVEM   A,FFLOC                 ;first address at which to allocate
INIT1:  MOVEM   A,FRELOC                ;first address not yet allocated
>;EXTADR
CPOPJ:  RET

;here for each file.  Store the file name in the data area
DOFILE: HRROI   A,NAMBUF                ;Extract the file name from the JFN
        HRRZ    B,DIRJFN                ;output to NAMBUF.
        MOVEI   C,0
        JFNS
         ERJMP  ERROR
        MOVEI   B," "                   ;add some spaces after the name
        IDPB    B,A
        IDPB    B,A
FILUP1: IDPB    B,A                     ;fill with spaces until the byte pointer
        HRRZ    C,A                     ;addresses NAMBUF+5, that is, fill to a
        CAIG    C,NAMBUF+4              ;minimum of 26 columns
         JRST   FILUP1                  ;loop until something is stored in +5
```

```
        MOVEM   A,LINPTR                ;save pointer to line thus far.
        HRRZ    A,DIRJFN                ;the JFN again.
        MOVSI   B,.FBLWR+1              ;number of words to read,,zero offset
        MOVEI   C,FDBADR                ;plunk them here.
        GTFDB                           ;read the file descriptor block
         ERJMP  ERROR
        MOVE    A,LINPTR                ;byte pntr to the line we're building
        HRRZ    B,FDBADR+.FBBYV         ;size of file in pages
        MOVE    C,[NO%LFL!<6,,^D10>]    ;flags for NOUT. Leading blank fill
        NOUT                            ;6 spaces, decimal radix.
         ERJMP  ERROR
        MOVEI   B,11                    ;add a tab
        IDPB    B,A
        MOVE    B,FDBADR+.FBREF         ;date and time of last reference
        MOVEI   C,0                     ;flags for ODTIM
        JUMPE   B,FILNVR                ;jump if Never referenced.
        ODTIM                           ;convert date and time to a string.
         ERJMP  ERROR                   ;add the string to the line we have
        JRST    FILUP2

FILNVR: MOVE    B,[POINT 7,[ASCIZ/Never/]]
        CALL    CPYSTR                  ;Copy string, from B to A.
FILUP2: MOVEI   B,15                    ;add cr-lf-null to the line
        IDPB    B,A
        MOVEI   B,12
        IDPB    B,A
        MOVEI   B,0                     ;end the line with a null
        IDPB    B,A

;Store this record in dynamic memory.  FRELOC is the address of the first
;free location.  The record format is:
;       first word:             the file's reference date
;       second word:            the file's size in pages
;       third and following:    the ASCIZ string containing the file name
;                               and other file information
;The end of the record is determined by the null byte at the end of the
;ASCIZ string.   FRELOC is updated to point to the word beyond the word
;where the ASCIZ null was stored, i.e., FRELOC is made to point to the next
;free location.
```

```
        MOVE    A,FRELOC                ;get address of first free location.
        MOVE    B,FDBADR+.FBREF         ;get the ref. date.
        MOVEM   B,0(A)                  ;store reference date
        HRRZ    B,FDBADR+.FBBYV         ;get the file size in pages
        MOVEM   B,1(A)                  ;store file size.
ZADR<
        ADD     A,[POINT 7,2]           ;convert free pointer to byte pntr
>;ZADR
EXTADR<
        ADD     A,[.P07!2]              ;Convert free pointer to OWGBP
>;EXTADR
        MOVE    B,[POINT 7,NAMBUF]      ;pointer to the string we made.
        CALL    CPYSTR                  ;Copy string from B to A
        MOVEI   B,0                     ;Add a null to end the text buffer
        IDPB    B,A
        ADDI    A,1                     ;advance to next word
ZADR<
        HRRZM   A,FRELOC                ;store as new free location.
>;ZADR
EXTADR<
        TXZ     A,77B5                  ;clear OWGBP bits
        MOVEM   A,FRELOC                ;store address of next free locn
>;EXTADR
        RET

CPYSTR: ILDB    C,B                     ;read from NAMBUF
        JUMPE   C,CPOPJ                 ;at null, exit.
        IDPB    C,A                     ;store in free space.
        JRST    CPYSTR                  ;loop until null is read

;The BUILDA routes makes one pass through the data area to build an array
;of pointers to beginning of each record.
;The data area starts at C(FFLOC) and extends to C(FRELOC)-1.
;Copy FRELOC to TBLSTT; TBLSTT is the address where the array of
;record pointers will be built.
;The first record address is given by FFLOC.  Other record addresses
;are computed by finding the null at the end of the ASCIZ string in
;each record.

;In this routine, register A is the address of the next available word
;that we can store a record address pointer into.  Initially, this is the
;same as TBLSTT.  It is incremented after each record pointer is stored.
```

```
BUILDA: MOVE    A,FRELOC                ;first free location becomes...
        MOVEM   A,TBLSTT                ;the address of the start of array
        MOVE    B,FFLOC                 ;base of string data area.
;At BLDLP, B is the address of the start of a record.
;When B equals or exceeds FRELOC, we have reached the end of all records.
BLDLP:  CAML    B,TBLSTT                ;are we done yet?
         JRST   BLDONE                  ;yes
        MOVEM   B,@FRELOC               ;store address of record
        AOS     FRELOC                  ;increment address for next record pntr
ZADR<
        ADD     B,[POINT 7,2]           ;make record address into string pointer
>;ZADR
EXTADR<
        ADD     B,[.PO7!2]             ;make record address into string pointer
>;EXTADR

BLDLP1: ILDB    C,B                     ;look for the end of the text
        JUMPN   C,BLDLP1                ;loop until a null is seen.
ZADR<
        HRRZ    B,B                     ;keep only the address part of byte pntr
>;ZADR
EXTADR<
        TXZ     B,77B5                  ;keep only the address part of byte ptr
>;EXTADR
        AOJA    B,BLDLP                 ;Increment to addr of next record. loop

;As we exit, copy the current value for FRELOC to TBLEND, one beyond the
;end of the record pointer array.
BLDONE: MOVE    A,FRELOC                ;copy FRELOC to signify
        MOVEM   A,TBLEND                ;the end address of the table
        RET
```

```
;time to sort stuff.  The table starts at TBLSTT and ends just before TBLEND.
;Call SORT with A/ Element Count
SORT:   SOJLE   A,CPOPJ                 ;decrease N by 1, exit if none left.
ZADR<
        MOVN    A,A                     ;negate it
        HRLZ    A,A                     ;-(N-1),,0
        HRR     A,TBLSTT                ;-(N-1),,First data address.
SORT0:  MOVE    B,A                     ;copy of the pointer
        MOVEI   C,0                     ;number of exchanges, so far
SORT1:  MOVE    D,@0(B)                 ;get the data item (date)
        CAMG    D,@1(B)                 ;compare to next item
         JRST   SORT2                   ;this pair is ok.
        ADDI    C,1                     ;count an exchange was made
        MOVE    D,0(B)                  ;exchange pointers
        EXCH    D,1(B)
        MOVEM   D,0(B)
SORT2:  AOBJN   B,SORT1                 ;loop thru all pairs of items
>;ZADR
EXTADR<
SORT0:  MOVE    B,TBLSTT                ;point to record [1]
ADDI B,1 ;advance to [2].  compare [k] vs [k-1]
        MOVEI   C,0                     ;number of exchanges, so far
SORT1:  MOVE    D,@-1(B)                ;get a data item (date) [k-1]
        CAMG    D,@0(B)                 ;compare to next item.  [k]
         JRST   SORT2                   ;this pair is ok.
        ADDI    C,1                     ;count an exchange was made
        MOVE    D,0(B)                  ;exchange pointers.
        EXCH    D,-1(B)                 ;
        MOVEM   D,0(B)
SORT2:  ADDI    B,1
        CAMGE   B,TBLEND ;within the table still?
 JRST   SORT1 ;yes, check next parirl
>;EXTADR
        JUMPG   C,SORT0                 ;jump if any exchanges were done
        RET
```

```
PRNTBL: MOVE    B,TBLSTT                ;first item in table
PRNTB1:
ZADR<
        HRRO    A,0(B)                  ;-1,,address of date word
        ADDI    A,2                     ;advance to the string itself
>;ZADR
EXTADR<
        MOVE    A,0(B)                  ;fetch pointer to record
        ADD     A,[.PO7!2]              ;make byte pointer to record field
>;EXTADR
        PSOUT                           ;send string to the terminal
        ADDI    B,1                     ;advance to next item in the table
        CAMGE   B,TBLEND                ;have we reached the end?
         JRST   PRNTB1                  ;no.  keep writing.
        HRROI   A,CRLF
        PSOUT
        RET

CRLF:   BYTE(7)15,12

;Error handling
ERROR:  HRROI   A,[ASCIZ/Error: /]
        ESOUT                           ;start of error message.
        MOVEI   A,.PRIOU                ;output error message to terminal
        HRLOI   B,.FHSLF                ;this fork, most recent error
        MOVEI   C,0                     ;no limit to byte count
        ERSTR                           ;convert error number to a string
         JFCL                           ;would you believe TWO error returns?
         JFCL
        HRROI   A,CRLF                  ;end message with end of line
        PSOUT
STOP:   HALTF
        JRST    START

        END     START
```

23.6 Discussion of this Program

As is usual in real programming situations, we have chosen to partition the program into several subroutines in order to make the overall structure more obvious. The main program commences at `START`. It resets any input/output operations and initializes the stack pointer. Other initialization functions are performed by the `INIT` subroutine.

```
START:  RESET                           ;normalize all I/O activity
        MOVE    P,[IOWD PDLEN,PDLIST]   ;establish a stack
        HRROI   A,[ASCIZ/Directory and Sort
/]
        PSOUT                           ;send a friendly greeting
        CALL    INIT                    ;obtain JFN. Setup storage
```

The loop called `LOOP` processes each file by calling `DOFILE`. The program advances from one file to

the next by means of the GNJFN JSYS. This JSYS will advance DIRJFN to the next file available in the list specified by the initial GTJFN. If another file is available, GNJFN will skip. The program will jump to LOOP to process the next file. When the list of files is exhausted, GNJFN will not skip; the program will jump to NOMORE.

A JFN is a *file handle* that the operating system provides to the program. The operating system knows at all times what file the particular JFN refers to. When the GNJFN JSYS is executed, the system advances the JFN to the next file. There is no need for the program to store an updated JFN; when the JFN is used again, the system knows that the JFN has been stepped to the next file. That is, TOPS–20 makes the original JFN point to a different file.

```
;Loop through all the files in the directory
LOOP:   CALL    DOFILE                  ;Perform functions for one file.
                                        ;After each file, step to the next file.
        MOVE    A,DIRJFN                ;step to the next file in sequence
        GNJFN                           ;note the full JFN & flags word is used
         JRST   NOMORE                  ;there are no more files
        JRST    LOOP                    ;we have another file, process it.
```

By the time the program gets to NOMORE, the JFN has been released by GNJFN. The routine BUILDA is called to build an array of pointers in memory. BUILDA will set up the locations called TBLSTT, *TaBLe STarT*, and TBLEND, *TaBLe END*, that define the size of the pointer array. If the array is empty then the program jumps to STOP; there is nothing to sort and nothing to print.

```
;There are no no more files in the directory
NOMORE: HRROI   A,CRLF
        PSOUT
;Build an array of pointers to the data records
        CALL    BUILDA                  ;Build the array
;Proceed to sort the array
        MOVE    A,TBLEND                ;Get the end of the array
        SUB     A,TBLSTT                ;minus start = number of things (N)
        JUMPE   A,STOP                  ;jump if table is empty.
        CALL    SORT                    ;sort things.  A=number of elements
;Print the sorted results
        CALL    PRNTBL                  ;print table
        JRST    STOP                    ;finish.
```

Assuming the table isn't empty, the SORT routine is called to sort the table. In this program the sorting is done to show files ordered by their date of most recent reference, with the least recently referenced file first. After SORT returns, PRNTBL is called to print this array. After printing the result, the program jumps to STOP where execution terminates.

23.6.1 DOFILE

DOFILE processes each file in the directory. First, DOFILE uses a JFNS JSYS to translate the JFN into a file name. Only the right half of DIRJFN is used in the JFNS call. The file name from JFNS is stored in the array called NAMBUF. On return from JFNS, register A contains a byte pointer to the last byte in NAMBUF that was used for storing the file name. Two blanks are deposited following the file name.

```
DOFILE: HRROI   A,NAMBUF                ;Extract the file name from the JFN
        HRRZ    B,DIRJFN                ;output to NAMBUF.
        MOVEI   C,0
        JFNS
         ERJMP  ERROR
        MOVEI   B," "                   ;add some spaces after the name
        IDPB    B,A
        IDPB    B,A
```

The loop at FILUP1 continues to deposit blanks until a blank has been deposited in (or after) the word at NAMBUF+5. The intention of this code is to fill the file name with trailing spaces through 26 columns. If the file name is longer than 26 columns, a total of three blanks will appear after the name for spacing. The resulting byte pointer is saved in LINPTR.

```
FILUP1: IDPB    B,A                     ;fill with spaces until the byte pointer
        HRRZ    C,A                     ;addresses NAMBUF+5, that is, fill to a
        CAIG    C,NAMBUF+4              ;minimum of 26 columns
        JRST    FILUP1                  ;loop until something is stored in +5
        MOVEM   A,LINPTR                ;save pointer to line thus far.
```

Next, we obtain the file size in pages and the date of the most recent reference to the file by using the GTFDB, *GeT File Descriptor Block*, JSYS.[1] The file descriptor block, or FDB, is a part of the file directory that contains further information about the file.

The GTFDB JSYS requires a JFN in register 1. Register 2 is set to contain (in the left) the number of words to read, and in the right the offset from the start of the FDB. This format allows you to select only one or two words from the middle of the FDB if that is all you want. In this program, we request words 0 through the word called .FBLWR. Register 3 contains the address of where we want the data stored; in this program, GTFDB is told to store the FDB data in the array called FDBADR.

```
        HRRZ    A,DIRJFN                ;the JFN again.
        MOVSI   B,.FBLWR+1              ;number of words to read,,zero offset
        MOVEI   C,FDBADR                ;plunk them here.
        GTFDB                           ;read the file descriptor block
         ERJMP  ERROR
```

The right half of the word at FDBADR+.FBBYV contains the size of the file in pages[2]. Register 1 is loaded with the byte pointer that we saved above. Register 2 is loaded with the size of the file in pages. Register 3 is loaded with a word containing the control flags for the NOUT JSYS; NOUT is used to convert the number of pages in the file to a text string addressed by register 1.

The NOUT JSYS, as we saw in example 13, effects the same conversion of a number to a character string as does the DECOUT routine that we first examined in example 7. Often NOUT is used to write to a file, but, as we have done here, it can also write its results to an arbitrary byte pointer. In the NOUT JSYS, the format of the number is controlled by the flags in register 3. The flag NO%LFL requests that leading fill (with blanks) be provided. The number 6 in the left half of register 3 is the number of columns to print; the value in the right half is the radix to use in printing the number. NOUT returns an updated byte pointer in register 1 (i.e., A). That pointer is used to deposit a tab

[1] The JFNS JSYS could obtain this information also, but we want to use the date (and perhaps the size in some other version of this program) as the sorting key. GTFDB obtains this for us in the most useful format.

[2] The definitions for the symbols .FBBYV, .FBLWR, etc. come from the universal definitions in MONSYM.UNV. Consult [MCRM] for a detailed discussion of the names and meanings of the FDB data items.

character following the file size.

```
        MOVE    A,LINPTR                ;byte pntr to the line we're building
        HRRZ    B,FDBADR+.FBBYV         ;size of file in pages
        MOVE    C,[NO%LFL!<6,,^D10>]    ;flags for NOUT. Leading blank fill
        NOUT                            ;6 spaces, decimal radix.
        ERJMP   ERROR
        MOVEI   B,11                    ;add a tab
        IDPB    B,A
```

The date of the most recent reference is loaded into `B` from our copy of the `.FBREF` word in the FDB. Register `A` still has a byte pointer. Register `C` is set to zero to request default format control from the `ODTIM`, *Output Date and TIMe*, JSYS. `ODTIM` converts the reference time to a character string. The characters are stored using the byte pointer in `A`.

If the reference date word is zero, the program jumps to `FILNVR`. Rather than include a meaningless date, 16–Nov–1858, the program adds the word "Never" to the line being constructed. The word "Never" is added to the line by calling the `CPYSTR` subroutine. This version of `CPYSTR` is similar to others that we have examined.

The characters carriage return, line feed, and null are added to the end of the line that is being composed in `NAMBUF`.

```
        MOVE    B,FDBADR+.FBREF         ;date and time of last reference
        MOVEI   C,0                     ;flags for ODTIM
        JUMPE   B,FILNVR                ;jump if Never referenced.
        ODTIM                           ;convert date and time to a string.
        ERJMP   ERROR                   ;add the string to the line we have
        JRST    FILUP2

FILNVR: MOVE    B,[POINT 7,[ASCIZ/Never/]]
        CALL    CPYSTR                  ;Copy string, from B to A.
FILUP2: MOVEI   B,15                    ;add cr-lf-null to the line
        IDPB    B,A
        MOVEI   B,12
        IDPB    B,A
        MOVEI   B,0                     ;end the line with a null
        IDPB    B,A
```

The subroutine `DOFILE` thus far has constructed a string, starting at `NAMBUF`, that describes the current file. The string includes the file name, its size and reference date. Next, `DOFILE` builds a record in memory for the current file. The record built for each file in the directory consists of a word containing the reference date, a word containing the size of the file in pages, and a series of words containing a copy of the text string that we composed in `NAMBUF`.

The address contained in `FRELOC` is picked up into `A`; this value is the address of the first free word in memory. We will copy the current file record to the words following this address. The program gets the reference date from our copy of the FDB. The reference date is stored in the address that `A` points to; again, this is the lowest available address in memory, as specified by `FRELOC`. The file size in pages is copied from the FDB to the second word of free space, `1(A)`.

```
;Store this record in dynamic memory.  FRELOC is the address of the first
;free location.  The record format is:
;       first word:             the file's reference date
;       second word:            the file's size in pages
;       third and following:    the ASCIZ string containing the file name
;                               and other file information
;The end of the record is determined by the null byte at the end of the
;ASCIZ string.   FRELOC is updated to point to the word beyond the word
;where the ASCIZ null was stored, i.e., FRELOC is made to point to the next
;free location.
        MOVE    A,FRELOC                ;get address of first free location.
        MOVE    B,FDBADR+.FBREF         ;get the ref. date.
        MOVEM   B,0(A)                  ;store reference date
        HRRZ    B,FDBADR+.FBBYV         ;get the file size in pages
        MOVEM   B,1(A)                  ;store file size.
```

Next, we convert the address in `A` to be a byte pointer.

In the section 0 program, the address contained in `A` is local; a one–word local byte pointer will do: add the constant `POINT 7,2` to the address in register `A`. This constant has the octal value 440700,,2. The result in `A` is a byte pointer for 7–bit bytes that addresses the third word (counting 0, 1, 2) of the file record.

In the extended addressing program, `A` contains a 30–bit address that is not local to the program's section. We convert `A` to a one–word global byte pointer by adding the constant `.p07!2`. This constant adds 2 to the address in `A` and decorates the high bits of `A` to make it a one–word global byte pointer (OWGBP in the comments).

```
ZADR<
        ADD     A,[POINT 7,2]           ;convert free pointer to byte pntr
>;ZADR
EXTADR<
        ADD     A,[.P07!2]              ;Convert free pointer to OWGBP
>;EXTADR
```

The resulting byte pointer is used to augment the file record with a copy of the text string that resides in `NAMBUF`. Register `B` is loaded with a byte pointer to the string in `NAMBUF`; the copy is made by the `CPYSTR` routine. After making this copy, a null byte is added to terminate the string in the file record.

```
        MOVE    B,[POINT 7,NAMBUF]      ;pointer to the string we made.
        CALL    CPYSTR                  ;Copy string from B to A
        MOVEI   B,0                     ;Add a null to end the text buffer
        IDPB    B,A
```

After copying the string from `NAMBUF` to the file record, the address portion of the byte pointer contains the address of the word into which we stored the null byte to end the string. The program advances that address by adding one to it. This is the address of the next free word in memory. For the section zero program, the address is 18 bits in the right of `A`; this is stored as the new value of `FRELOC`. In the extended addressing program, the address is contained in the rightmost 30–bits of `A`; the program clears the leftmost six bit of `A` and stores the result as the new value of `FRELOC`. In either case, the next time free storage is needed, `FRELOC` will point past the record that we have just made.

```
        ADDI    A,1                     ;advance to next word
ZADR<
        HRRZM   A,FRELOC                ;store as new free location.
>;ZADR
EXTADR<
        TXZ     A,77B5                  ;clear OWGBP bits
        MOVEM   A,FRELOC                ;store address of next free location
>;EXTADR
        RET
```

23.6.2 BUILDA

We want to present the file records in order of their reference dates, earliest reference first. To order the records we must sort them. It would be possible to rearrange the records themselves, but that is complicated: each record is potentially a different length. Rearranging the records would involve a lot of copying of the record data.

Rather than rearrange the file records themselves, we choose to create pointers to the records and then to rearrange the pointers. Although this adds a step to create the pointers, once we have them, they are easier to sort as each pointer is just one word. Rearranging them consisists or rearranging pairs of words instead of pairs of records.

The address of a record is usually called a *pointer* to that record. In `BUILDA` the program will construct an array of record pointers. `BUILDA` is called after all the file records are built in memory by `DOFILE`.

`BUILDA` starts by copying the contents of `FRELOC` to the word called `TBLSTT`, *Table Start*. When `BUILDA` is called, `FFLOC` contains the address of the first file record and `FRELOC` contains an address that is one greater than the last word of the last file record. The value copied to `TBLSTT` represents the boundary between the area of memory that contains the file records and the array that is going to contain pointers to the file records. Specifically, the value in `TBLSTT` signifies two things: first, it is the address into which we will store the first word of the array of record pointers; second, it signifies the highest address of the file record space. This is shown in Figure 23.1.

The program will now scan through the file record space, identifying the addresses where file records start. When a file record is located, its address will be copied to the array that we are constructing. As shown, file records are contained in the region starting at the address given in `FFLOC` and ending just prior to the address given in `TBLSTT`. Register `B` is initialized from `FFLOC`. Each time the program reaches `BLDLP`, register `B` will contain the address of the start of some file record.

```
BUILDA: MOVE    A,FRELOC                ;first free location becomes...
        MOVEM   A,TBLSTT                ;the address of the start of array
        MOVE    B,FFLOC                 ;base of string data area.
;At BLDLP, B is the address of the start of a record.
BLDLP:
```

The loop at `BLDLP` copies the address in `B` into the array of record pointers, and then computes a new value for `B` by locating the next file record following the one that `B` points to. After all the file records have been scanned, the program leaves this loop.

In detail, the loop at `BLDLP` checks to see that `B` is smaller than the value in `TBLSTT`. The address in `B` is advanced by this loop. When it matches the value in `TBLSTT` we have finished building the

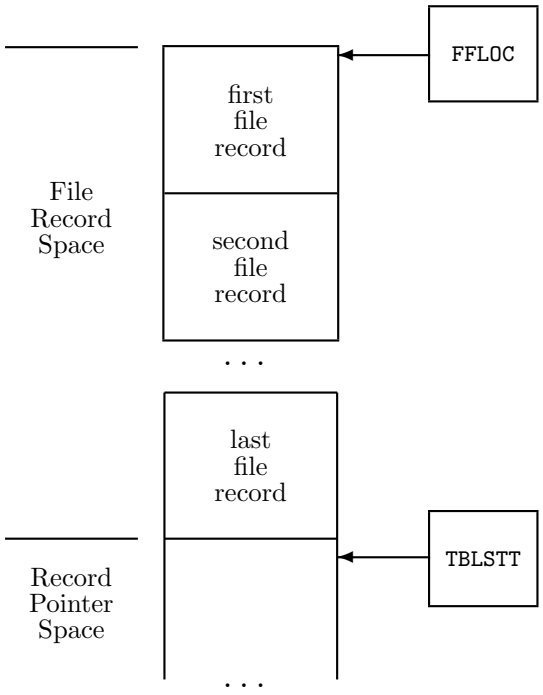


Figure 23.1: File Record and Pointer Space in Memory

array of record pointers; the program jumps to BLDONE. If we have not yet finished processing all the records, register B is the address of a file record. It is stored in the word addressed by FRELOC. Then FRELOC is incremented. The first time through BLDLP, FRELOC will be the same as TBLSTT. Thus, FRELOC in general points to the next available word to be used when adding things to the table of record addresses.

```
BLDLP:  CAML    B,TBLSTT                ;are we done yet?
         JRST   BLDONE                  ;yes
        MOVEM   B,@FRELOC               ;store address of record
        AOS     FRELOC                  ;increment address for next record pntr
```

B contains the address of one of the file records. For the section 0 program, adding POINT 7,2 to the address in B give us a byte pointer to the text in the file record; for the extended addressing program, adding .P07!2 accomplishes the same. At BLDLP1, using B as a byte pointer, the text portion of the file record is scanned until the null that terminates the string is found. The byte pointer in B then contains the address of the last word of the file name string. In the section 0 program, the right half of B is the address of the last word of the present file record; in the extended addressing program, 30 bits of B are that address, so the program clears the high six bits of B. With B now the address of the last word of a file record, the program increments B to be the address of the next file record and it loops to BLDLP.

```
ZADR<
        ADD     B,[POINT 7,2]           ;make record address into string pointer
>;ZADR
EXTADR<
        ADD     B,[.P07!2]              ;make record address into string pointer
>;EXTADR

BLDLP1: ILDB    C,B                     ;look for the end of the text
        JUMPN   C,BLDLP1                ;loop until a null is seen.
ZADR<
        HRRZ    B,B                     ;keep only the address part of byte pntr
>;ZADR
EXTADR<
        TXZ     B,77B5                  ;keep only the address part of byte ptr
>;EXTADR
        AOJA    B,BLDLP                 ;Increment to addr of next record. loop
```

As a result of this subroutine, a table whose starting address is contained in TBLSTT is built. The data items in the table are the addresses of (i.e., pointers to) the individual file records. This scheme is used because it was not possible to know at the beginning how many files there would be, or how long each file record would be.

When the file records are exhausted, the loop at BLDLP exits to BLDONE. At this point FRELOC contains the address of the next word that could have been used for the table of pointers. That address is copied to TBLEND to mark the end of the table of pointers.

```
;As we exit, copy the new value for FRELOC to TBLEND.  This is one beyond the
;end of the record pointer array.
BLDONE: MOVE    A,FRELOC                ;copy next free address to
        MOVEM   A,TBLEND                ;signify the end of the table
        RET
```

23.6.3 SORT

The main program computes the number of file records by subtracting TBLSTT from TBLEND. If the result is zero there is nothing to sort and nothing to print; the main program exits to STOP. Assuming the directory was not empty, the SORT routine is called with A containing the number of elements to sort.

When SORT is entered it decreases the item count in A by one. If the result is zero, SORT returns immediately: one item is already sorted. (Although the caller of SORT has already tested for the case of no items to sort, the instruction is SOJLE instead of SOJE. It is slightly more general this way.)

```
SORT:   SOJLE    A,CPOPJ                ;decrease N by 1, exit if none left.
```

The SORT routine shows the greatest difference in the section zero and extended addressing versions. The AOBJN instruction that was used in the section zero code is unsuited to accessing an array in another section: an index register with a negative left half provides local addressing in an instruction; in a global format address word, bits 6:17 of the index register are part of the address. So, the details of the code in the section zero program do not have great influence on the extended addressing version

23.6.3.1 Sorting in Section Zero

For the section 0 code, the sort continues as described here. The number in A is negated and moved to the left half of A. The address of the start of the table is copied to the right half of A. The reader should recognize a negative control count in the left half and an address in the right half is suitable for use with the AOBJN instruction. Register A will remain unchanged by the remainder of SORT. Note that if there are N data items, register A will contain the negative of $N-1$ in the left; this is the assembly language analog of the FOR loop in which an index was run from 1 to $N-1$.

```
        MOVN    A,A                    ;negate it
        HRLZ    A,A                    ;-(N-1),,0
        HRR     A,TBLSTT               ;-(N-1),,First data address.
```

At SORT0, the AOBJN pointer is copied from A to B. Register C is initialized to zero; C will count the number of interchanges that are done on each pass through the data.

```
SORT0:  MOVE    B,A                    ;copy of the pointer
        MOVEI   C,0                    ;number of exchanges, so far
```

Bubble sort compares adjacent data elements. If any pair of elements is out of order, the program will interchange them. A pass through the data consists of comparing the first and second elements, then the second and third, then the third and fourth, and so on, until elements numbered $N-1$ and N have been compared. If, at the end of a pass, no interchanges have been necessary, the data is now sorted. Otherwise, another pass is needed.

When there are N items to sort each pass will require $N-1$ comparisons. In the worst case, $N-1$ passes will be required. Even in the average case, approximately $N/2$ passes will be needed. Therefore, on the average you might expect that Bubble sort will require $(N-1)\times N/2$ comparisons. Such a sort is said to be of *order* N^2. The N^2 stems from the fact that for large N, the size of the quantity $(N-1)\times N/2$ is dominated by growth of the N^2 portion. As we shall demonstrate in the next example, N^2 is much too high a price to pay for sorting.

At SORT1, the right half of B contains the address of some element in the table. If 0(B) addresses some particular element number, say k, in the table, then 1(B) will address element $k+1$. Recall that the table consists of *pointers* to the actual file records that are being compared. Therefore, the effective address computed by 0(B) will be the data item (i.e., the reference date) of record number k.

At SORT1 the reference date of one record is loaded into D. The reference date of the next record is compared with the contents of D. If the items are in order, the CAMG instruction doesn't skip and the program jumps to SORT2. If the items are out of order, the pointers to these two records are interchanged. Each interchange is counted in C.

```
SORT1:  MOVE    D,@0(B)                 ;get the data item (date)
        CAMG    D,@1(B)                 ;compare to next item
         JRST   SORT2                   ;this pair is ok.
        ADDI    C,1                     ;count an exchange was made
        MOVE    D,0(B)                  ;exchange pointers
        EXCH    D,1(B)
        MOVEM   D,0(B)
```

At SORT2, the current pair is correct; either it was correct to start with or the elements have been interchanged. Register B is advanced to address the next pair by means of the AOBJN instruction. The AOBJN jumps back to SORT1, where the comparison process is repeated. At the conclusion of a pass, the control count, kept in the left half of B, will become exhausted and the AOBJN will fail to jump. At this point, if C shows that interchanges have been necessary, the program initiates another pass through the entire array by jumping to SORT0. When a pass is completed in which no interchanges were necessary, the SORT routine is done.

```
SORT2:  AOBJN   B,SORT1                 ;loop thru all pairs of items
        JUMPG   C,SORT0                 ;jump if any exchanges were done
        RET
```

23.6.3.2 Sorting in Extended Addresses

For extended addressing, we still use bubble sort. Each pass through the array of size N will require $N-1$ comparisons of consecutive records. Where the section zero program counted from 1 to $N-1$, this program will, in effect, count from 2 to N, because this makes the test for the end of pass a bit simpler. Where the section zero program compared record k and record $k+1$, this program will compare records k and $k-1$.

The label SORT0 marks the start of one pass of comparisons. C, the count of interchanges is initialized to zero. B, which will be used as the "pointer to record k" is initialized from TBLSTT and then immediately advanced by one. This initializes B to point to the second record.

```
SORT0:  MOVE    B,TBLSTT                ;point to record [1]
ADDI B,1 ;advance to [2].  compare [k] vs [k-1]
        MOVEI   C,0                     ;number of exchanges, so far
```

The label SORT1 denotes where a pair of consecutive records is compared. If B is thought of as pointing to record k, the comparison involves that record and record $k-1$. As B names a location that contains the address of a file record, data in the file record is accessed by indirect addressing.

If the pair of elements are correctly ordered, the CAMG doesn't skip and the program jumps to SORT2

where it will advance B. If the records are not correctly ordered, the code interchanges the pointers to the records, in effect, interchanging the record. When an interchange is required, the program increments C.

```
SORT1:  MOVE    D,@-1(B)                ;get a data item (date) [k-1]
        CAMG    D,@0(B)                 ;compare to next item.  [k]
         JRST   SORT2                   ;this pair is ok.
        ADDI    C,1                     ;count an exchange was made
        MOVE    D,0(B)                  ;exchange pointers.
        EXCH    D,-1(B)                 ;
        MOVEM   D,0(B)
```

At SORT2 the program advances B. If the address in B now matches TBLEND, a pass is complete. If the pass is not complete, the program loops to SORT1 to continue with the next pair of comparisons. At the end of a pass, if C is not zero, interchanges were made and another pass is necessary. When a pass finishes and no interchanges were done, the sort is complete.

```
SORT2:  ADDI    B,1
        CAMGE   B,TBLEND ;within the table still?
 JRST   SORT1 ;yes, check next parirl
        JUMPG   C,SORT0                 ;jump if any exchanges were done
        RET
```

23.6.4 PRNTBL

Results are printed by the PRNTBL routine. Register B is loaded from TBLSTT, the starting address of the table.

```
PRNTBL: MOVE    B,TBLSTT                ;first item in table
```

Again there are differences between the section zero program and the extended addressing program. In section zero, we can get by with an argument to PSOUT that is just -1 in the left half and the in–section address of a string in the right half. In the extended addressing program, we must provide the 30–bit address of the string and give an actual one–word global byte pointer.

For the section 0 program, code at PRNTB1 loads the address of the file record into the right side of A and sets the left side of A to −1. Since the first two words of the file record are not part of the text, we add 2 to A to make it point directly at the text string within the file record. This text is sent to the terminal via the PSOUT JSYS.

```
PRNTB1: HRRO    A,0(B)                  ;-1,,address of date word
        ADDI    A,2                     ;advance to the string itself
        PSOUT                           ;send string to the terminal
```

For the extended addressing program, the word at 0(B) is the address of a file record. To this address, the program adds .P07!2. The resulting sum is a one–word global byte pointer that points to the text string within the file record.

```
PRNTB1: MOVE    A,0(B)                  ;Address of the file record
        ADD     A,[.PO7!2]              ;Convert to OWGBP, point at text
        PSOUT                           ;send string to the terminal
```

Following the `PSOUT`, `B` is advanced by one. While `B` remains smaller than `TBLEND` the program will loop to `PRNTB1`.

```
        ADDI    B,1                     ;advance to next item in the table
        CAMGE   B,TBLEND                ;have we reached the end?
         JRST   PRNTB1                  ;no.  keep writing.
```

After all the file records have been output, `B` will be equal to `TBLEND`. The `PRNTBL` routine will print one more CRLF and return.

23.7 Example 15 — Heapsort

We shall make amends for the bubble sort shown in example 14 by introducing a better sorting technique. This better sorting algorithm is called Heapsort. Heapsort was discovered by J.W.J. Williams [*CACM* 7 (1964), 347-348.]; an efficient approach to heap creation was suggested by R.W. Floyd [*CACM* 7 (1964), 701.]. A full description and analysis of Heapsort appears in [Knuth 3].

Heapsort is much faster than bubble sort. Bubble sort requires on the order of N^2 operations to sort N data items, but Heapsort requires only on the order of $N \times \log(N)$ operations. Since $log(N)$ is much smaller than N (and grows much more slowly than N) the quantity $N \times \log(N)$ is much smaller than N^2. Heapsort is more complicated than Bubble sort, however. For this reason, when only a small number of items are being sorted, a Bubble sort may be preferable. The area of computer science called *Analysis of Algorithms* studies the mathematical basis for selecting which algorithm is best for a particular case.

Heapsort makes use of a data structure called a *tree*. An element in the tree may have any number of *descendants*. Every element has precisely one *ancestor* element, except for the one element at the *root* of the tree which has no ancestor. Often we refer to the descendants of an element as *sons*, and the ancestor as the *father*. We examined a special kind of tree, the *binary tree*, in the examples of the halfword instructions and `PUSHJ` (see Section 12.1, page 158). The binary tree is a tree in which an element may have at most two descendants.

In this version of Heapsort, the *heap* is a binary tree in which the data contained in the sons of any node are numerically less than or equal to data contained at their father node. This implies that the root of the tree is larger than (or equal to) every other node in the tree; one of the two sons of the root node is the second largest item in the tree, etc. A sample heap is depicted in Figure 23.2

This diagram shows a representative heap in which there are twelve elements. Note that at each node in the tree, the father is numerically greater than either son. Half of the nodes (the bottom row) will have no sons at all.

There are three things that we have yet to explain about the heap. First, we should explain how to represent this tree–like structure in computer memory. Second, we must describe the process by which unsorted data is formed into a heap. Finally, we must describe how to extract the sorted information from the heap.

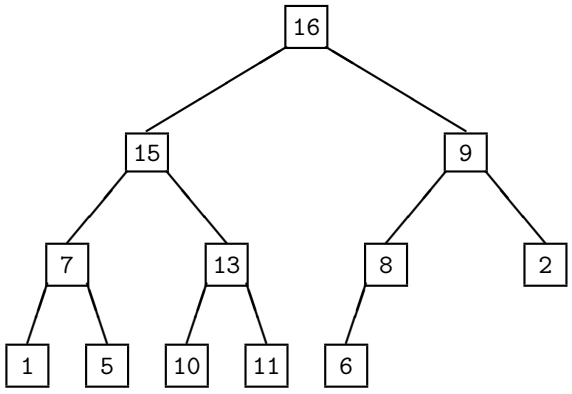


Figure 23.2: Example of a Heap

23.7.1 Machine Representation of a Heap

A heap is a binary tree. Usually, tree–like data structures require pointers (i.e., from the father to each son) to help us find our way around in the structure. Pointers are sometimes cumbersome. Remarkably, the Heapsort algorithm very neatly does away with the explicit pointers that tree structures usually require. The way to represent the heap is really quite simple. Let the N elements of the heap be held in an array `HEAP[1..N]`. Then the root of the tree, the father of all nodes, is `HEAP[1]`. The sons of node K are at $2 \times K$ and $2 \times K + 1$. Thus the sons of `HEAP[1]` are at `HEAP[2]` and `HEAP[3]`, the sons of `HEAP[5]` are `HEAP[10]` and `HEAP[11]`, etc.

23.7.2 Building a Heap

If randomly ordered elements are present in the array `HEAP`, how do we organize those elements to form a heap? The algorithm is not intrinsically difficult, but it takes a while to describe; please bear with us.

First, by the definition of a heap, the elements `HEAP[`$N/2 + 1$`]` through `HEAP[`N`]` have no sons.[3] Therefore, they cannot be out of order with respect to their sons. These elements form the initial bottom level of the heap. Nothing at all has to be done to these elements to place them into the initial heap.

Now, we will add elements one by one to the heap. We will work backwards, from `HEAP[`$N/2$`]` to `HEAP[1]`, adding one element at a time and reshuffling the data as necessary so that all the properties of the heap are maintained. When we add a new element to the heap, it is possible that it won't be in the right place. So, when some particular new item is added to the heap, say `HEAP[`X`]`, we examine the sons of `HEAP[`X`]` to determine if `HEAP[`X`]` is larger than both sons. If `HEAP[`X`]` is larger than both of its sons, we leave it at `HEAP[`X`]`. If, as is often the case, `HEAP[`X`]` fails to be larger than both of its sons, we select the larger of its sons, say, `HEAP[`Y`]`, where Y is either $2 \times X$ or $2 \times X + 1$, and interchange `HEAP[`X`]` with `HEAP[`Y`]`. Since the original `HEAP[`Y`]` is larger than the the original `HEAP[`X`]`, `HEAP[`Y`]` ought to be moved to be the father of `HEAP[`X`]`. This movement is accomplished by the interchange of `HEAP[`X`]` and `HEAP[`Y`]`.

[3] When we speak of $N/2$ in this discussion, we mean the integer portion of the quotient, as the `IDIV` instruction would compute.

Having accomplished one interchange, we have made things better, but not necessarily perfect. The new element, originally placed in `HEAP[`X`]` and now residing in `HEAP[`Y`]` may still be wrongly placed in the heap. We must examine the sons (if any) of the newly installed `HEAP[`Y`]` to see if further motion is needed. This is accomplished exactly as described above. The variable X that is the element number of the newcomer is changed by copying the value of Y to it. The program loops back to the point where it tries to decide if `HEAP[`X`]` is correctly placed. Again, an element is correctly placed when it is larger than both of its sons. If it has no sons, it cannot be wrongly placed with respect to its sons.

Let us examine the process of building a modest heap. Suppose, we start with nine numbers (to keep from making the example too tedious). They might be arranged as: 1, 9, 2, 8, 3, 7, 4, 6, and 5.

With nine items, items that are numbered $9/2+1$ (that is, 5) through 9 form the initial bottom row of the heap. Items 1 to $9/2$ (i.e., 4) are set aside because they have yet to be added to the heap. We use asterisks to mark where future items will be placed in the heap.

```
remaining: 1, 9, 2, 8
                    *
                  *   *
                *   3  7   4
               6 5
```

Working backwards through the series of remaining data items, 1, 9, 2, 8, we add 8 to the fourth position in the heap:

```
remaining: 1, 9, 2
                    *
                  *   *
                8   3  7   4
               6 5
```

Since 8 is larger than its two sons, 6 and 5, we are satisfied with this arrangement.

Next we add the 2 to the heap:

```
remaining: 1, 9
                    *
                  *   2
                8   3  7   4
               6 5
```

This item is not positioned properly. Both of its sons are larger than 2. We select the larger of the two sons, in this case it is 7, and interchange items:

```
remaining: 1, 9
                    *
                  *   7
                8   3  2   4
               6 5
```

We examine the new situation; since 2 now has no sons it is not wrongly placed.

We go on to the next element; the 9 is added to the heap:

```
remaining: 1
                      *
                    9      7
                  8   3  2   4
                 6 5
```

There is no difficulty placing the 9. It is larger than its two sons, so no motion is needed.

When we add the last element, the 1, there will be several dislocations:

```
none remain           1
                    9      7
                  8   3  2   4
                 6 5
```

Since 1 is smaller than its two sons, it is interchanged with the 9.

```
none remain           9
                    1      7
                  8   3  2   4
                 6 5
```

Examining the new situation, again we discover that 1 is wrongly placed. It is interchanged with the larger of its sons:

```
none remain           9
                    8      7
                  1   3  2   4
                 6 5
```

The 1 still is not correctly placed. We interchange it with the larger son:

```
none remain           9
                    8      7
                  6   3  2   4
                 1 5
```

Having driven the 1 all the way down to the bottom row, it now has no sons, and is correctly placed.

Thus, we have constructed a heap from unsorted data.

23.7.3 Removing Sorted Data from the Heap

By the nature of the heap, the largest item in the heap will be found at the top of the heap. We can remove the item at `HEAP[1]` with assurance that no other item is larger. When we remove an item the heap becomes smaller. Also, since there is now no data item at `HEAP[1]`, we no longer have a heap.

We rebuild the heap in the following expedient manner. Let Z represent the present size of the heap. First, remove HEAP[1]. Copy HEAP[Z] to HEAP[1]. Then, the new size of the heap is $Z - 1$. Of course, there is no assurance that HEAP[Z] was the right element to put at the top of the heap. So, using the same procedure that we used when building the heap originally, we shuffle the heap until the former HEAP[Z] (that we had placed at HEAP[1]) sinks to its proper location. Meanwhile, by this process, some new item has risen to become HEAP[1]. This will be the largest item in the smaller heap.

Repeat this process. Each time an element is removed from the heap, the heap gets smaller. Eventually, all of the elements have been removed, with the largest elements having been removed earliest.

23.7.4 Intermediate Storage for Heapsort

We have yet to explain why we choose to build the heap so that the largest item comes out first. Superficially, it would be more natural to have the smallest item come out first. However, there is a good reason behind this unusual choice: when the heap shrinks, it leaves room at the high end of the array HEAP. When HEAP[Z] (in terms of the description above) is copied to HEAP[1] and the heap size is set to $Z - 1$, a hole is left at HEAP[Z]. The array element HEAP[Z] does not belong to the heap of size $Z - 1$, so we can store anything at all in HEAP[Z]. What we choose to store there is the original HEAP[1] which was the largest item in the heap of size Z. Thus, as the heap shrinks, it leaves room in the array. We fill the array with the data extracted from the heap. Since the heap shrinks in size towards HEAP[1] it is natural to want to fill the space at the end of the HEAP array with the largest items first. At the end of the process HEAP[N] will be the largest item, HEAP[$N - 1$] will be the second largest, and so forth, with HEAP[1] being the smallest.

Since we have explained this trick, we might as well admit to the rest. In the description of how the heap was constructed originally, the places where we showed the asterisks actually held the data that was labeled "remaining". Since these items were not being considered as part of the heap, the fact that they were sitting there in the array HEAP was irrelevant to the description. In the implementation of the actual program, however, we can use the space in the array HEAP for these data items, to avoid tying up any other space in memory.

23.7.5 High–Level Representation of Heapsort

The following is a representation of the Heapsort algorithm in the Pascal language.

```
PROGRAM heap;

TYPE heaptype = INTEGER;
     heapsize = 1..100;
     heaparray = ARRAY [heapsize] OF heaptype;

VAR theheap:heaparray;
    zz,num:heapsize;
```

```
PROCEDURE heapsort(n:heapsize; VAR heap:heaparray);
    VAR z:heapsize;
        temp:heaptype;

    FUNCTION mustmove(x,size:heapsize):BOOLEAN;
        BEGIN (* Decide if heap[x] is properly placed *)
        mustmove:=FALSE;
        IF 2*x <= size THEN IF heap[x] < heap[2*x] THEN mustmove:=TRUE;
        IF 2*x+1 <= size THEN IF heap[x] < heap[2*x+1] THEN mustmove:=TRUE
        END;

    FUNCTION select(x,size:heapsize):heapsize;
        BEGIN (* Select which son of heap[i] to exchange *)
        select:=2*x;
        IF 2*x < size THEN IF heap[2*x] < heap[2*x+1] THEN select:=2*x+1
        END;

    PROCEDURE enheap(x,size:heapsize);
        VAR y:heapsize;
            t:heaptype;
        BEGIN (* Correctly place a new element, heap[x], into the heap *)
        WHILE mustmove(x,size) DO
            BEGIN
            y:=select(x,size);
            t:=heap[x];
            heap[x]:=heap[y];
            heap[y]:=t;
            x:=y
            END
        END;

BEGIN
(* Build the heap *)
FOR z:=n DIV 2 DOWNTO 1 DO Enheap(z,n);
(* Shrink the heap *)
FOR Z:=n DOWNTO 2 DO
        BEGIN
        temp:=heap[1];
        heap[1]:=heap[z];
        heap[z]:=temp;
        enheap(1,Z-1)
        END
END; (* of heapsort procedure *)

BEGIN
(* Place the main program here *)
(*         ...                 *)
(*                             *)

END.
```

23.7.6 Heapsort Subroutine

The following routine implements the Heapsort algorithm. This routine can be "plugged into" example 14, replacing the Bubble sort that is there. When adding this to example 14, three writeable memory locations, `N`, `HEAPX`, and `HEAPY` must be defined also.

For the extended addressing version of the program, the writeable locations must be placed in the DATA psect. There is no need to collect the text defining the writeable variables in one place in the program. Instead, from the middle of the CODE psect, we can invoke the DATA psect, describe the variables we need, and resume the CODE psect when we say `.ENDPS`.

```
        SUBTTL  Heapsort Subroutine - Example 15

EXTADR<
        .PSECT  DATA            ;add writeable variables to DATA psect
>;EXTADR
N:      0                       ;Index to highest element number in heap
HEAPX:  0                       ;Pointer by which we access HEAP[X]
HEAPY:  0                       ;Pointer by which we access HEAP[Y]
EXTADR<
        .ENDPS                  ;resume the interrupted psect
>;EXTADR

;Call SORT with A/ item count.
;TBLSTT contains the address of the first item, HEAP[1]
```

```
SORT:   MOVEM   A,N             ;Save the number of things to sort
        MOVE    Y,TBLSTT        ;Address of the first pointer element.
        SUBI    Y,1             ;Offset for 1-origin indexing
ZADR<
        HRLI    Y,X             ;X,,address of HEAP[0]
        MOVEM   Y,HEAPX         ;Reference HEAP[X] via @HEAPX
        HRLI    Y,Y             ;Y,,address of HEAP[0]
        MOVEM   Y,HEAPY         ;Reference HEAP[Y] via @HEAPY
>;ZADR
EXTADR<
        TXO     Y,<<X>B5>       ;GIW indexed by X, base is address of HEAP[0]
        MOVEM   Y,HEAPX         ;Reference HEAP[X] via @HEAPX
        TXC     Y,<<X^!Y>B5>    ;Y,,address of HEAP[0]
        MOVEM   Y,HEAPY         ;Reference HEAP[Y] via @HEAPY
>;EXTADR

;HEAPX is set up so that if an element number is placed in X then @HEAPX
;will fetch HEAP[X].  Similarly, @HEAPY can be used to to reference HEAP[Y].
        MOVE    Z,N             ;Number of elements to sort.
        LSH     Z,-1            ;Form N/2
        JUMPE   Z,CPOPJ         ;If N=1 (i.e., N/2=0) then sort is done.
SORT1B: MOVE    X,Z             ;This loop will build the heap
        CALL    ENHEAP          ;Insert HEAP[X] into heap
        SOJG    Z,SORT1B        ;Loop. End when HEAP[1] is placed.
;Now the heap is built.  start removing things from it.
SORT1C: MOVE    Y,N             ;Index to last item (OLD N)
        SOSG    Z,N             ;Make heap smaller.  Decrement N.
         RET
        MOVEI   X,1             ;Element number for HEAP[1]
        MOVE    B,@HEAPY        ;Exchange HEAP[1] with HEAP[old N].
        EXCH    B,@HEAPX
        MOVEM   B,@HEAPY
        CALL    ENHEAP          ;Reposition the new HEAP[1] in the heap. X=1
        JRST    SORT1C          ;Loop.  Reduce size of heap.

;Call ENHEAP with X=element number to add to heap,
;                 N=number of elements in the heap
ENHEAP: CALL    MOVSEL          ;Decide what to move.  Return largest son in Y.
         RET                    ;Item is well-placed.  Move nothing.
        MOVE    B,@HEAPY        ;Exchange HEAP[X] with HEAP[Y]
        EXCH    B,@HEAPX        ;
        MOVEM   B,@HEAPY        ;
        MOVE    X,Y             ;Now, we must be sure that the new HEAP[Y]
        JRST    ENHEAP          ;  is properly placed.
```

```
;Call MOVSEL with X = element number of an element to check, and
;                N = number of elements in the heap.
;       If HEAP[X] is properly placed with respect to its sons,
;               return without skipping.
;       Otherwise, skip, returning in Y the element number of the larger son.
;
MOVSEL: MOVE    Y,X             ;Element number of target element
        LSH     Y,1             ;2*X
        CAMLE   Y,N             ;Skip if first son exists. 2*X <= N
         RET                    ;No sons.  Element is placed ok.
        MOVE    B,@HEAPX        ;B:=address of the record for HEAP[X]
        MOVE    B,(B)           ;The data item
        MOVE    C,@HEAPY        ;Address of HEAP[2*X]
        CAMGE   B,(C)           ;Compare HEAP[X] and HEAP[2*X]
        AOJA    Y,MOVSL1        ;HEAP[2*X] is larger.  Must exchange.
        CAML    Y,N             ;Skip if 2*X < N (that is, 2*X+1 <= N)
         RET                    ;Only one son. HEAP[X] is in the right place.
        ADDI    Y,1             ;Element number of HEAP[2*X+1] (second son)
        MOVE    C,@HEAPY        ;Address of HEAP[2*X+1]
        CAMG    B,(C)           ;Skip if HEAP[X] > HEAP[2*X+1]
CPOPJ1:  AOS    (P)             ;Not properly placed, return Y = 2*X+1
        RET                     ;HEAP[X] is properly placed.

;Here HEAP[2*X] is larger than HEAP[X].  Examine second son.  Y has 2*X+1
MOVSL1: CAMLE   Y,N             ;Is there a second son?  2*X+1 <= N?
        SOJA    Y,CPOPJ1        ;No.  Set Y to 2*X; must swap HEAP[X], HEAP[Y].
        MOVE    B,(C)           ;HEAP[2*X].  C was set from above.
        MOVE    C,@HEAPY        ;Address of HEAP[2*X+1]
        CAMLE   B,(C)           ;Compare HEAP[2*X] to HEAP[2*X+1].
        SUBI    Y,1             ;HEAP[2*X] is larger, return Y = 2*X
        JRST    CPOPJ1          ;(If the CAMLE skips, return Y = 2*X+1)
```

23.7.7 Discussion of the Heapsort Subroutines

These routines are written to approximate the structure of the Pascal version of Heapsort. The major difference is that the functions `mustmove` and `select` have been combined into the `MOVSEL`, *MOVe SELection*, function.

The Heapsort routine is entered at `SORT`. Register `A` is stored as `N`, the number of items to sort. Register `Y` is set up to have the contents of `TBLSTT` minus one. The word `TBLSTT` contains the address of the first item in the array that we want to sort. In most algorithms we would treat the first item as item number zero; in Heapsort, we want to treat the first item as item number one. Therefore, register `Y` is being set up to contain the address of the non–existent item number zero.

In the section zero program, the left half of `Y` is set to `X`, the name of an index register. This quantity is stored in `HEAPX`. `HEAPX` will be used as a indirect address word to access data items in the heap. When `X` contains the index number (between 1 and N) of some object in the heap, the program can fetch or store that object by using the address expression `@HEAPX`. When the computer encounters the address expression `@HEAPX` in an instruction, it fetches `HEAPX` and performs the address calculation using the data it finds there. The right half of `HEAPX` contains the address of `HEAP[0]`; bits `14:17`, the index field, of `HEAPX` name `X`. Therefore, the contents of `X` will be added to the address of `HEAP[0]`

to form the address of HEAP[X], the address of the desired object.

The location HEAPY is set up in a similar fashion, but register Y appears in HEAPY as the index register. A reference to @HEAPY will obtain the item from HEAP[Y]. The code for the section zero program is

```
SORT:   MOVEM   A,N             ;Save the number of things to sort
        MOVE    Y,TBLSTT        ;Address of the first pointer element.
        SUBI    Y,1             ;Offset for 1-origin indexing
ZADR<
        HRLI    Y,X             ;X,,address of HEAP[0]
        MOVEM   Y,HEAPX         ;Reference HEAP[X] via @HEAPX
        HRLI    Y,Y             ;Y,,address of HEAP[0]
        MOVEM   Y,HEAPY         ;Reference HEAP[Y] via @HEAPY
>;ZADR
```

In the extended addressing program, the same idea is followed except Y contains the 30–bit address of HEAP[0]. Then instead of storing the index register number (X or Y) in bits 14:17, we must store them in bits 2:5 to form a global indirect word (GIW) in which both an index register and a base address are provided.

We've used TXO to set a value in bits 2:5 before. This code uses a trick to set the value of Y into that field. After TXO, the GIW index field contains X. Instead of clearing the field and setting it to Y, we can XOR a value into that field that will change X to Y. The instruction that does the XOR is TLC (via the TXC macro). The value X^!Y when XORed with X will change X to Y. Thus, we write TXC Y,<<X^!Y>B5> to set the index field of the GIW to contain Y. In all, this differs from the section zero code in only two instructions:

```
EXTADR<
        TXO     Y,<<X>B5>       ;GIW indexed by X, base is address of HEAP[0]
        MOVEM   Y,HEAPX         ;Reference HEAP[X] via @HEAPX
        TXC     Y,<<X^!Y>B5>    ;GIW indexed by Y, base is HEAP[0]
        MOVEM   Y,HEAPY         ;Reference HEAP[Y] via @HEAPY
>;EXTADR
```

The program continues by setting up Z with the value $N/2$. If $N/2$ is zero, the SORT subroutine exits: there is only one element to sort. Register Z will count down, from $N/2$ to 1 as items are added to the heap. The loop at SORT1B calls ENHEAP with X set up as a copy of Z; this is the element number to add to the heap portion of the array. After each item is added, Z is decremented and the program loops back to SORT1B. SORT1B is executed until, after adding HEAP[1] to the heap, Z becomes zero.

At SORT1C it is time to shrink the heap. The current value of N is copied to register Y. Then N is decremented, copying the new, smaller value of N to Z. When N is reduced to zero, the routine exits. For larger values of N, the constant 1 is copied to X, and the original value of N (in Y) is used to govern the interchange of HEAP[1] with the last element of the heap.

The exchange places the largest element of the heap at the end of the array. The element that was displaced is inserted into the heap at HEAP[1]. The smaller heap, in which HEAP[1] is wrongly placed, is rebuilt as a heap by calling ENHEAP, with X, the item number to place in the heap still set to one. The program loops to SORT1C and the heap continues to shrink. As the heap shrinks, the space in the array vacated by the heap is filled with properly sorted data.

The ENHEAP routine moves one element, addressed by X, into the proper position within the heap.

ENHEAP first calls MOVSEL to see what adjustment of the heap is necessary. If MOVSEL returns without skipping, no further adjustment is needed. If MOVSEL skips, the item HEAP[X] must be interchanged with one of its sons. The son to interchange with is specified by the value of Y that MOVSEL returns. HEAP[X] and its son, HEAP[Y], are interchanged.

After this interchange, the item that we just moved further down into the heap may not yet be placed correctly. Therefore, we copy the item number of the new location of the item we are working on from Y into X, and loop back to ENHEAP where we call MOVSEL to see if this item has come to a satisfactory resting place; if not, the process repeats until the item finds a proper home.

The MOVSEL routine combines the MUSTMOVE and SELECT functions from the Pascal version. MOVSEL is entered with X containing the item number of an element; MOVSEL will determine if the item is wrongly placed in the heap. An item is wrongly placed if either of its sons is larger than itself. At MOVSEL the index to the first son, twice the contents of X, is calculated in Y. If this index is larger than N, the number of items in the heap, then MOVSEL gives the non–skip return; the item in question is correctly placed because it has no sons. If the item has at least one son, the father and the first son are compared. It is important to remember that the data items in this heap are pointers to the file records described in example 14.

things. First, the field specification (the first column number and the length of the field). Second, whether to sort in ascending or descending sequence. Third, the name of the input file. Lastly, the name of the output file.

Assume the field contains only numeric data. Read the input file. Select the keys necessary to perform the sort. Sort the data. Write the output file.

Among the optional extensions that you could make to this program are

- Handle non–numeric data. Treat the field as a string of characters instead of as a number.
- Allow several field specifications. If two lines of input have identical first fields, then sort them based on the second field. If both the first and second fields are identical, sort them based on the third field, etc.

Chapter 24

Lists and Records

In example 14, the directory and sort program, we dealt with a data structure called a *record*. A record is a block of memory with a particular structure that is defined by the programmer. In example 14 the structure was

```
Word 0:         File Reference Date
Word 1:         File Size in Pages
Word 2:         The ASCIZ text of the file name
```

This description should be augmented by stating that the length of the text string is variable.

This block of memory is an example of a record; the structure of the record, as defined by the program, includes fixed portions in fixed places, and it includes a variable length portion that begins at a fixed place. Each of the data items within a record is usually called a *field*. Records may be vastly more complex than this.

One of the particular benefits of using records is that a record keeps all the information pertaining to a particular item in one place. In example 14, the items are files; the data that is kept includes the file name, the date and time of the most recent reference, and the size of the file. The address of the record in memory is the single handle by which all the data that is known about the item can be retrieved. As we have mentioned, this handle is sometimes called a *record pointer* or, more simply, a *pointer*.

One problem with using records is keeping track of all the records that a program might construct. The only handle that we have on a record is its address in computer memory; if we forget the address of a record (and if we can not recompute it somehow) we will lose the information that the record contains.

We did not worry too much about remembering the addresses of the records while they were being built in example 14. Instead, the program remembered the range of addresses in which all records were kept; moreover, the program had a technique by which it could determine where one record ended and another began.

Later in example 14, an array was built that contained the address of every record. Although arrays can be used for remembering record pointers, an array is not always suitable. The array in example 14 was built after all the records were constructed; this was done because there was no way to know in advance how many different records there would be. Space for the array of record pointers could not have been allocated until the number of records was known.

A more general technique for dealing with records is to include space in every record to point to another record. The data structure that results from records pointing to other records is called a *list*. Lists permit extremely powerful programming techniques to be used. A list is useful where a number of records are to be organized for sequential processing. Records that are constructed dynamically are an extremely valuable data structure for applications where the number of data items cannot be predicted. Lists are a useful way to organize dynamic records.

The important points to remember about a list are

- A list is composed of items called records. These items are often identical in structure (but they need not be). (When the records are not identical in structure, each record must somehow identify what kind of structure it is.)
- Every record contains at least one field that is a record pointer.
- A particular value of the pointer field signifies the end of the list. This value is sometimes called the *null pointer* or *NIL* (both are terms from the programming language LISP). In the DECSYSTEM–20 we shall often find it convenient to use zero as this special value.
- A word, often in some fixed location, contains the pointer to the first record in the list. This is called the *list head*. Sometimes the list head of one list may be a field of a record belonging to another list.

24.1 Example 16 — Dictionary Program

This program allows the user to select an input file and an output file. The program reads the input and copies it to the output. Each word in the input file is placed into a dictionary that is built in memory. An occurrence count is kept for each word. After reading the entire input file, this program will output the dictionary in alphabetical order, printing also each word's occurrence count. Finally, the program will output the words in occurrence count order, with the least frequently used words appearing first.

This program attempts better error recovery, especially from errors in the user's file specifications, than has previously been demonstrated. This program also demonstrates the use of a *hash table* to make dictionary searches go faster; lists and record data structures are shown. Finally, this program includes a reasonably fast sorting algorithm suited for linked lists.

This program is our longest and most complicated example thus far. In assembly language programming there is an inescapable tendency towards long programs. Composing and debugging a long program need not be difficult if approached properly. We begin with a plan, a general idea of the structure of the program, that breaks the program into manageable subroutines. As we write the code to implement the plan, we call these subroutines. Eventually, rather vague descriptions of subroutines will be made very specific and then they will be coded.

A long program should contain many, relatively short, subroutines. The process of debugging a long program consists of verifying and debugging each of the subroutines. If the subroutines have been designed properly and debugged thoroughly, then when the subroutines work, no other problems will remain.

The major structural elements of this program are quite straightforward. The main program begins by calling for an input file and an output file from routines `GTINPF` and `GTOUTF` that are quite similar to what we have seen before:

```
START:  RESET
        MOVE    P,[IOWD PDLEN,PDLIST]   ;Initialize stack
        CALL    GTINPF                  ;Get input file
        CALL    GTOUTF                  ;Get output file
```

We continue in the main program by calling the DINIT subroutine to initialize the dictionary. Since we have not yet discussed the form of the dictionary, the meaning of "initialize the dictionary" is somewhat vague. Obviously, we should expect that some initialization will be necessary. When we decide on the format of the dictionary, we will place the appropriate initialization steps in DINIT.

Next, we must read the input file and copy the input to the output. For each group of characters that we think might form an English word, we will increment the occurrence count for that word.

It seems most reasonable to divide this processing into two subroutines. The first, GETWRD, will read from the input and copy to the output. When GETWRD finds a sequence of alphabetic characters that form an English word, it stores them as a string and returns to the main program. The PROCWD subroutine processes the word. By "processing the word," we mean that each input word will be looked for in the dictionary. If the word is found, its occurrence count will be incremented. If the word can't be found in the dictionary, a new dictionary entry will be made for it.

```
        CALL    DINIT                   ;Initialize dictionary

MAIN:   CALL    GETWRD                  ;Get a word from input
        JUMPE   B,INEOF                 ;Jump if now at end of file
        CALL    PROCWD                  ;Process word
        JRST    MAIN                    ;continue in file
```

We may assume that GETWRD will signal that the end of file has been encountered in some characteristic way. As we have yet to write GETWRD we are allowed to specify any protocol that we desire. We shall write GETWRD so it returns a non–zero quantity in register B to signify that a word has been found. At end of file, GETWRD will return a zero in register B. The main loop, shown above, makes use of some of these assumptions.

When the end of the file is found, we must close the input file, sort the dictionary to make it alphabetical, sort the dictionary again by occurrence count order, and then print the two sorted versions.

At this point it is necessary to make further steps towards specifying some of the characteristics of the dictionary. Each unique English word will be represented by a record in memory. A dictionary record includes the text of the English word and a field in which the occurrence count is kept. Lists are used to connect the various records together. For the moment we may assume that each record appears on two different lists. These lists that pass through each record are called the *name list* and the *count list*.

The name list threads through every record in the dictionary. As the dictionary is built, words are simply added to the front of the name list. However, after all the words have been added, the dictionary will be sorted alphabetically. At the end of the sort, the name list will contain all the records in alphabetical order.

While the dictionary is being built, the count list is constructed in the same way as the name list. In fact, until the two sorts occur, the count list is identical to the name list. The second sort rearranges

the count list so that records appear on it in ascending order of occurrence counts.

Let us assume that the head of the name list will be in the location called NSHEAD, meaning *Name Sort HEAD*. The head of the count list will be CSHEAD, the *Count Sort HEAD*. In order to sort a list, we must copy the contents of the list head to a register and call the appropriate sort routine. We shall see that the sort routine will return the new list head in the same register.

The following fragment closes the input file and calls both of the sorts:

```
;Here at end of file on input.
INEOF:  HRRZ    A,IJFN                  ;close input file
        CLOSF
        ERJMP   .+1                     ;Closf normally skips.
        SKIPE   A,NSHEAD                ;sort the name list, by name
         CALL   NSSORT
        MOVEM   A,NSHEAD                ;new head of name sort list
        SKIPE   A,CSHEAD                ;sort count list, by counts
         CALL   CSSORT
        MOVEM   A,CSHEAD                ;new head of the csort list
```

Note that the instruction SKIPE A,NSHEAD picks up the name list head into register A. Normally, we would not expect the instruction to skip, but in the unusual event that there are no dictionary words (i.e., the name sort list is empty) we must avoid calling the sort routine, NSSORT. The result of NSSORT is a new list; the pointer to the first element of that list is returned in register A. That list pointer is stored as the new value of NSHEAD. Similarly, the count list is sorted and new value set in CSHEAD.

After the sorting is finished, we call the routine PRDICT to print the dictionary in two different formats. The program concludes by calling the FINISH subroutine and halting. We may assume that FINISH will end any output activity and close the output file.

```
        CALL    PRDICT                  ;print dictionary, both ways
        CALL    FINISH                  ;clean up output activity
        HALTF                           ;stop execution here
        JRST    START                   ;for CONTINUE, start over
```

This explanation of the general structure of the program may have left several unanswered questions. We have yet to discuss the exact format of the dictionary, or the details of GETWRD and PROCWD. The sort routines are presently shrouded in mystery. Bear with us, and we shall reveal all.

24.2 Dictionary Records

In this program each unique English word is represented by a record in memory. In example 14 we saw a record in which the computer words represented the reference date, the file size, and the file name string. In this example, the records are somewhat more complicated.

The record includes the text of the English word and a field in which the occurrence count is kept. Lists are used to connect the various records together. In addition to the two lists that we have already described, the name list and the count list, each record appears on a third list, called the

hash list. As a consequence of being on three lists, each record contains three pointer fields.[1]

Just as with the name list and the count list, the hash list is built as each word is added to the dictionary. The purpose of the hash list is quite different from the purpose of other two lists; the hash list helps the program perform speedy searches through the dictionary as each input word is processed. We will discuss the hash list and the hash search technique shortly.

We write the description of a prototypical record to define the field names and relative positions of the fields within each record. In MACRO we can define the fields of the dictionary record as shown below. (The line containing the definition of `WRDBUF` that appears below is not part of the record definition. It is included here because it is mentioned in the explanation of the `PHASE` pseudo–op.)

```
WRDBUF: BLOCK   40                      ;The word buffer

DEFS:!  PHASE   0                       ;definitions of record fields
WCOUNT:!        0                       ;occurrence count
HSHLNK:!        0                       ;hash-linkage
NSLINK:!        0                       ;name-sort linkage
CSLINK:!        0                       ;Count-sort linkage
NAMBLK:!                                ;entry name starts here
        DEPHASE
        .ORG    DEFS
```

This definition is somewhat complex; it introduces several new pseudo–ops, `PHASE`, `DEPHASE`, and `.ORG`, and the concept of suppressed labels.

24.2.1 Suppressed Labels

We use the characters `:!` in defining some of the labels above. In MACRO, `:!` defines a *suppressed label*. A suppressed label is similar to an ordinary label that is defined with a colon; however, as with a suppressed symbol defined via `==`, a suppressed label is not available for output by DDT's symbolic disassembler. These labels are suppressed in this program because, as we shall see, they have values that would conflict with our accumulator definitions. (Usually, a label corresponds to a location. But `WCOUNT` and others are not actual locations.)

24.2.2 PHASE and DEPHASE Pseudo–Operators

The definition of `WRDBUF` as `BLOCK 40` sets the current location counter in MACRO to `WRDBUF+40`. When the symbol `DEFS` appears as a label, it is assigned the value `WRDBUF+40`. `DEFS` is the last symbol to be defined prior to the `PHASE` pseudo–op.

The `PHASE` pseudo–op has a very peculiar effect; it establishes a second location counter, called the *phase location counter*. The phase location counter is initialized to the value of the expression that follows the word `PHASE`.

After the `PHASE` pseudo–op is seen, assembled words, code and data, will continue to be placed in memory locations as before, as if no `PHASE` had occurred. But, labels that are defined while a `PHASE` pseudo–op is in effect are given a value that is controlled by the phase location counter. Each time

[1] This example is lavish in its use of memory, perhaps only to show that one record can be on more than one list at a time. This example could be recoded with only one pointer field, and one set of lists at a time.

a word is assembled and stored, the assembler increments both the usual *storage location counter* (that points to where things get stored) and the phase location counter.

If no PHASE had appeared on the line with DEFS:!, then the next line would define WCOUNT to be the same as DEFS. However, since PHASE 0 appeared, the phase location counter is set to zero. Then the symbol WCOUNT is defined to have the value zero. The word containing the constant data zero, that appears on the line with WCOUNT, is assembled and stored in memory at DEFS. After storing the zero data word at DEFS, MACRO increments the storage location counter DEFS+1 and the phase location counter to 1. Next, HSHLNK is defined to have the value one; another zero is stored into memory at DEFS+1. The phase location counter advances to 2; the storage location counter advances to DEFS+2.

Similarly NSLINK and CSLINK are defined as 2 and 3, respectively, storing zeros at DEFS+2 and DEFS+3. Finally, NAMBLK is defined as 4.

When we define the labels WCOUNT, HSHLNK, etc., we make them suppressed labels because they take on values that would conflict with our accumulator names.

The DEPHASE pseudo–op makes both the storage location counter and the phase location counters the same: the value of the storage location counter is copied to the phase location counter; things revert to normal.

While under control of a PHASE pseudo–op, the symbol "." has the value of the phase location counter.

The purpose of all of this is somewhat subtle. We could have defined WCOUNT==0, HSHLNK==1, and so forth. The reason we avoided that method of defining symbols was to establish a picture of the record; to visually and unmistakeably show the relations among the fields within the record definition. Because we have used this technique, it would be possible to reshuffle these symbols, for example to put HSHLNK after CSLINK, without struggling to make sure all definitions have been changed. Again, we have forced MACRO to do more bookkeeping for us, with the aim of creating a program that is more readily changeable.

24.2.3 .ORG Pseudo–Operator

The .ORG pseudo–op resets the storage location counter (and the phase location counter) to the value specified by its argument. In this program, we set the location counter back to point to DEFS, where our prototype record definition appeared. The next thing that is assembled and stored will be located at WRDBUF+40, the value of DEFS.

The .ORG pseudo–op reclaims the space consumed by the record definition. Since we do not in fact need any of that space, we re–use it by telling the assembler to go back and put something else there.

Because we release this space for something other than the record definitions, the symbol DEFS has been suppressed.

24.3 Searching by Hash Code

In PROCWD, processing a word consists of finding the dictionary record for the word and incrementing that record's occurrence count. If no dictionary entry for this word exists, one must be created. For each word in the input file, the dictionary must be searched to locate the record for that word. This search is called a *look up* operation.

There are several ways that the dictionary look up process could be carried out. If we have a list of dictionary words, we might look at the first dictionary entry to see if it matches the input word. If it doesn't match, we could advance to the second dictionary word to see if that is a match. This process can be repeated; eventually either a match for the input word is found in the dictionary, or all the words in the dictionary have been examined without finding a match. This process is easy to implement but it is not very efficient for large dictionaries.

Another technique that comes to mind is to find the word based on its alphabetical (lexicographic) order in the dictionary. This is the technique that people use when trying to find a word in a printed dictionary. This approach could be programmed; it would be superior to the first technique that was discussed. The alphabetical lookup can be quite useful in cases where insertions into dictionary are uncommon. However, depending on the data structure that is chosen for the dictionary, insertion may be expensive. It is possible to use a linked tree data structure for the dictionary; insertions are not too burdensome. It is quite interesting to program the tree search and tree insertion routines. But, trees are not what we have chosen to demonstrate at this time.

In many circumstances, the hash code search that we will describe is superior to the tree search algorithms. The basic principle of searching by hash codes is to partition the search space. The *search space* is the set of records that must be examined to locate the object of the search. If the search space were divided into several regions, where the desired object is known to be present in one specific region (unless it does not exist at all), then the program can concentrate on searching in that specific region. For example, if there are two thousand objects in the search space, and if the search space were divided into one hundred regions, you might expect to find about twenty objects in each region. So, instead of having to search through two thousand objects, you might need to examine only about twenty.

An arithmetic function, called a *hashing function*, is applied to the search object; the result is a number that is called the *hash code* or *hash index*. The search space is partitioned into some number of regions called *hash buckets*. All objects that have the same hash index are dropped into the same hash bucket. For example, suppose that the English word "cocoanuts" has a hash code of `16`. Then "cocoanuts" would be dropped into bucket number `16`. When the word "cocoanuts" is presented to the search routine, the program understands that "cocoanuts" will be in bucket `16` or else not be present at all; it would be be pointless to look in bucket `1` or bucket `12`.

The hashing function always gives the same result when applied to identical objects. The hashing function is often designed so that the set of objects is approximately evenly distributed among the various buckets. The number of buckets is selected based on the application; as the expected number of objects in the search space becomes larger, the number of buckets should grow also.

In this program a hash bucket is a list that threads through all the records that belong in one bucket. The array `HASHTB` holds the list head for each hash bucket.

Other implementations of hash code searches avoid using linked lists. In such cases the problem of *collision*, i.e., two objects that have the same hash code, must be dealt with. The program described here expects collisions; objects that collide are linked onto a list. This list forms the hash bucket.

In the specific implementation of hash coding that we will use, the array `HASHTB` contains pointers to `HSHTLN` different hash buckets:

```
HSHTLN==177                             ;hash table size
HASHTB: BLOCK   HSHTLN                  ;the hash table
```

The entire hash table is initialized to zero by the `DINIT` subroutine that is called by the main program prior to processing the input file. Since we have described part of the initialization of this

program, we might as well tell about the other initialization steps. The list head of the name list, NSHEAD, and the head of the count list, CSHEAD are both set to zero. In the section zero program, the memory location called FREEAD is set to the first address above the program space. In the extended addressing program, we use SMAP% to create a private section and we initialize FREEAD to be the first address in that section.

```
DINIT:  SETZM   NSHEAD                  ;empty head of Name sort list
        SETZM   CSHEAD                  ;empty head of count sort list
        SETZM   HASHTB                  ;clear out the hash table
        MOVE    A,[HASHTB,,HASHTB+1]
        BLT     A,HASHTB+HSHTLN-1
ZADR<
        HLRZ    A,.JBSA                 ;address of free space
        MOVEM   A,FREEAD                ;save as free address
>;ZADR
EXTADR<
        SETZ    A,                      ;request a private section
        MOVE    B,[.FHSLF,,DSECT]       ;This fork, section number DSECT
        MOVE    C,[SM%RD!SM%WR!1]     ;1 section with read and write access
        SMAP%                          ;Request that TOPS-20 make the section
         ERJMP  [HRROI A,[ASCIZ/Cannot make section/]
                JRST ESTOP]             ;report an error
        MOVSI   A,DSECT                 ;first address in the new section
        MOVEM   A,FREEAD                ;first address at which to allocate
>;EXTADR
        RET
```

Initially, the dictionary is empty. The hash buckets, which are used while searching the dictionary, are empty. The count sort and name sort lists are empty.

24.3.1 PROCWD

The object of the PROCWD routine is to locate the dictionary record for the English word contained in WRDBUF. Once the record is found, the occurrence count field will be incremented. Of course, it may happen that the word does not yet exist in the dictionary. In that case, a dictionary record for this word will be created and added to the dictionary.

The way the hash search works is basically simple. The code at PROCWD calls HSHFUN to compute the hash code for the input word. The hash code selects one of the hash buckets. Each hash bucket is a list whose head is contained in the array HASHTB. If the input word is present in the dictionary, it will be found on this list.

A loop is necessary to search the hash bucket list. The loop will advance from one record to the next and decide if the new record matches the input word. If a match is found, the search terminates successfully. If the current record on the list doesn't match the input word, it is necessary to follow the hash bucket list to the next record. If this process reaches the end of the list without finding the input word, then that word is not yet in the dictionary; it will be added.

We begin coding by imagining that we are in the middle of the search. Suppose register B contains the address of a record that failed to match the input word. At PROCW1 we will advance from this record, called the *previous record*, to the next record.

The field HSHLNK of the previous record will give the address of the next record. We copy HSHLNK(B) to register A. If the previous record (pointed to by B) is the last record in this hash bucket, then register A will now contain a zero. If this happens, the program will jump to PROCW3 where we execute the code necessary to add a new record to the dictionary; PROCW3 uses B as the address of the last record on the list.

```
;Enter PROCW1 with B/ Address of Previous Record
PROCW1: SKIPN   A,HSHLNK(B)           ;link to next item
         JRST   PROCW3                ;Jump if end of list. Add item
```

Assuming now that we haven't exhausted the hash bucket, register A contains a pointer to the record following the one addressed by B. This is the address of the *current record*. We must now decide if the current record matches the input word. We accomplish this by loading registers B and C with byte pointers. One pointer addresses the input word in WRDBUF; the other points to the current record's name field, NAMBLK, where the text of this dictionary entry is stored.

The NAMCMP, *NAMe CoMParison*, subroutine is called. If the input name and the current record's name field are identical, NAMCMP will skip. We have identified the dictionary entry for the input word; the WCOUNT field of the current record is incremented and PROCWD returns.

If the names are different, NAMCMP returns without skipping; the JRST to PROCW2 is executed. At PROCW2 register A, the pointer to the current record, is copied to register B. The program loops back to PROCW1 with register B now containing the address of the previous, i.e., most recently examined, record. This loop repeats, examining all the records in one hash bucket until a match is found. When all the records have been examined without having found a match, the program jumps to PROCW3 where it builds a new record for this word.

```
;Enter PROCW1 with B/ Address of Previous Record
PROCW1: SKIPN   A,HSHLNK(B)             ;link to next item
         JRST   PROCW3                  ;Jump if end of list. Add item
        MOVE    B,[POINT 7,WRDBUF]      ;byte pointers
        MOVE    C,[POINT 7,NAMBLK(A)]   ;pntr to wd in current record
        CALL    NAMCMP                  ;compare input wd to record wd
         JRST   PROCW2                  ;names are not equal.
        AOS     WCOUNT(A)               ;Found. +1 to occurrence count
        RET

;Not found yet, continue search through the hash bucket
PROCW2: MOVE    B,A                     ;not equal.  B:= current rec
        JRST    PROCW1                  ;go to next rec in this bucket
```

Now it is time to show how we got to PROCW1 the first time. At PROCWD, the program calls HSHFUN to compute the hash code of the input word. The hash code is returned in register A. The word at HASHTB(A) contains the pointer to the first record in the hash bucket (or zero if the bucket is empty). Rather than write a program fragment to handle the special case of processing the first record, we will trick PROCW1 into doing that job for us.

The way we trick PROCW1 is a bit subtle. We made PROCW1 so that it expects that register B contains a record pointer to the previous record. Although there is no record previous to the first record in the hash list, we shall pretend that there is. We load register B with the address HASHTB-HSHLNK(A):[2]

[2]In the section zero program, XMOVEI has the effect of MOVEI, i.e., the appropriate behavior.

```
PROCWD: MOVE    B,[POINT 7,WRDBUF]      ;Pointer to the input word
        CALL    HSHFUN                  ;compute hash code for it.
        XMOVEI  B,HASHTB-HSHLNK(A)      ;initial link addr. B/prev rec
PROCW1: SKIPN   A,HSHLNK(B)             ;link to the next item
```

When we get to `PROCW1` the first time, the `SKIPN` there will use `B` as a pointer to a record. The `SKIPN` will advance to the next record by fetching the field addressed by `HSHLNK(B)`. The instruction to set up register `B` at `PROCW1-1` offsets the obvious address expression, `HASHTB(A)`, by subtracting `HSHLNK` from it. The `HSHLNK` part gets added back at `PROCW1`, canceling the effect of the first offset.

The first time the instruction at `PROCW1` is executed, register `A` is loaded from the list head for the hash bucket. If the list head is empty (i.e., zero) then the program jumps to `PROCW3`; the word in `WRDBUF` must be added to the dictionary. Register `B` contains a pointer to the last "record" in the hash chain.

If the word cannot be found on this hash bucket, it does not exist in the dictionary. We must add it. The program arrives at `PROCW3` with register `B` containing the address of the last record in the hash bucket. We will add the new record at the end of the list. This addition to the list will be accomplished by making a new record for the current input word. Then the hash link out of the record addressed by `B` will be set to point to the new record.

```
;Here if this word doesn't exist in the dictionary.  Add it.
PROCW3: PUSH    P,B                     ;addr of last item in hash list
        MOVE    B,[POINT 7,WRDBUF]      ;byte ptr to new dict word
        CALL    BLDBLK                  ;build record, return A/record addr
        POP     P,B                     ;address of last record.
        MOVEM   A,HSHLNK(B)             ;put new addr into last rec
        AOS     WCOUNT(A)               ;count occurrence of new item
        RET
```

At `PROCW3` register `B` contains the address of the last record in the hash bucket; this address is saved on the stack. `BLDBLK` is called with a byte pointer to the input word in `B`; `BLDBLK` builds a dictionary record for this word and returns the address of the new record in `A`. Register `B` is restored from the stack. The address of the new record is stored in `HSHLNK(B)`, extending the old hash bucket to include this new record. The occurrence count field of the new record is incremented to one.

24.3.1.1 HSHFUN

Given a byte pointer to an English word in register `B`, the subroutine `HSHFUN` will compute a hash index for that word. A polynomial function of the letters of the word is computed (compare this to the `DECIN` subroutine). This result is divided by the number of buckets; the absolute value of the remainder is taken as the hash index.[3] This number will be in the range from zero to the number of buckets minus one. When the number of buckets is a prime, this remainder is relatively evenly distributed over all of its possible values. This makes for hash lists that are nearly the same length.

[3] After enough letters, the `IMULI` can overflow, providing a negative value. Hence the need to take the magnitude of the remainder.

```
HSHFUN: MOVEI   A,0                     ;accumulate hash value here.
HASH1:  ILDB    C,B                     ;get a character
        JUMPE   C,HASH2                 ;jump if end of string
        IMULI   A,35                    ;multiply by some constant
        ADDI    A,-"A"(C)               ;add chr to the result
        JRST    HASH1

HASH2:  IDIVI   A,HSHTLN                ;div result by length of the
        MOVM    A,B                     ;hashtable.  Rem is hashcode
        RET                             ;in range 0 to HSHTLN-1
```

24.3.1.2 NAMCMP

The `NAMCMP` subroutine determines if two strings are lexically equal. At `NAMCMP`, one byte is read from each string. If the two characters read are unequal, this routine returns without skipping. If the characters are not both null, the program loops to `NAMCMP`. When two nulls are found simultaneously, this routine returns with a skip.

```
NAMCMP: ILDB    W,B                     ;compare names.  Get 1 byte
        ILDB    X,C                     ;from each of them
        CAME    X,W                     ;are they the same?
         RET                            ;not the same.  No skip.
        JUMPN   W,NAMCMP                ;the same. loop until null
        JRST    CPOPJ1                  ;both end at the same place.
```

24.3.1.3 BLDBLK

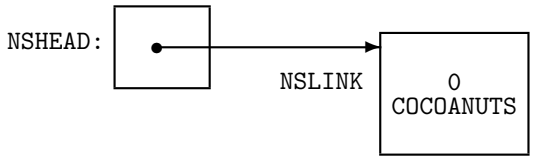
The purpose of the `BLDBLK` routine is to build a new dictionary record. The address of the first available word in free space is copied from `FREEAD` to register `A`; this will be the address of the new record. The `HSHLNK` and `WCOUNT` fields are set to zero. Register `C` is set up to point to the string portion of the new record. The text of the new word is copied to the new record. After copying the string, register `C` points to the last word in which text was stored; incrementing `C` results in a pointer to the next free word in memory; this is stored as the new value `FREEAD`.

```
BLDBLK: MOVE    A,FREEAD                ;addr of first free wd in mem
        SETZM   HSHLNK(A)               ;zero hash link out of here
        SETZM   WCOUNT(A)               ;and the occurrence count
        MOVE    C,A                     ;get address of new record.
ZADR<
        ADD     C,[POINT 7,NAMBLK]      ;make pntr to strng in new rec
>;ZADR
EXTADR<
        ADD     C,[.P07!NAMBLK]         ;make OWGBP to string in new record
>;EXTADR
BLDBL1: ILDB    W,B                     ;load from the word string
        IDPB    W,C                     ;put in new rec's word area
        JUMPN   W,BLDBL1                ;loop until the null is stored
        ADDI    C,1                     ;advance to next free word
ZADR<                                   ;in section 0, address is a halfword
        HRRZM   C,FREEAD                ;store as next free wrd in mem
>;ZADR
EXTADR<
        TXZ     C,77B5                  ;clear OWGBP bits, leave address
        MOVEM   C,FREEAD                ;store as next free word in memory
>;EXTADR
```

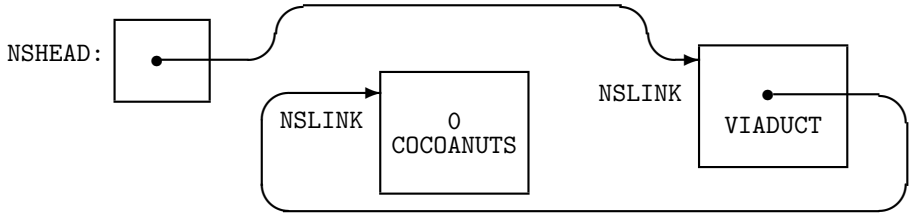
The head of the name–sort list, `NSHEAD`, is picked up at `BLDBL2`. This is the address of the most recently added record. The pointer to this other record is stored in the `NSLINK` and `CSLINK` portions of the new record. The address of the new record is stored as `NSHEAD` and `CSHEAD`. These instructions thread each new record onto two lists. Although these are called the name–sort and count–sort lists, they are not sorted until later. The object of these instructions is simply to build two lists, each containing every dictionary record.

```
BLDBL2: MOVE    C,NSHEAD                ;old head of name-sort list
        MOVEM   C,NSLINK(A)             ;add to front of nsort list
        MOVEM   C,CSLINK(A)             ;and to count-sort list
        MOVEM   A,NSHEAD                ;store this as new head of the
        MOVEM   A,CSHEAD                ;name-sort & count-sort lists
        RET
```

Since we have not yet had much discussion of list structures, we digress from the explication of this program to give an example of how these lists are manipulated. Suppose the very first word processed by this program is `COCOANUTS`. Initially `NSHEAD` is zero. The instructions at `BLDBL2` store the old head of the name–sort list in `NSLINK` of the `COCOANUTS` record. The address of the `COCOANUTS` record is then stored in `NSHEAD`. Similar activities occur with the count–sort list. The result, ignoring the count–sort list, looks like this:



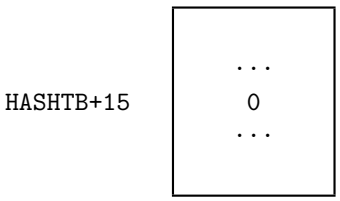
When another new word arrives, BLDBLK will make a dictionary entry for it. Suppose the new word is VIADUCT. Then the contents of NSHEAD will be copied to the NSLINK of the VIADUCT record. The address of the VIADUCT record will then be stored in NSHEAD. This changes the picture:



Each new record is added to the front of the name–sort list; NSHEAD will always point to the most recently added record.

Unless the reader is familiar with pointers, it is easy to become lost. We continue our digression by giving a specific example of searching by hash code. Two words, HARPO and CHICO, both have the same hash code, octal 15.[4]

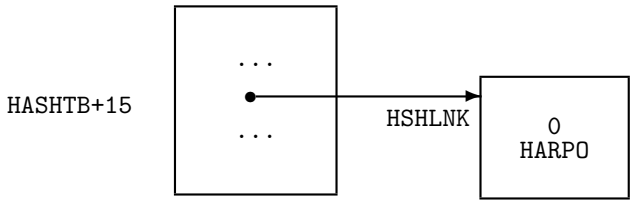
To start with, suppose that the dictionary is empty. The word HARPO appears in the input file, and is stored in WRDBUF by GETWRD. At PROCWD, the HSHFUN subroutine is called; HSHFUN eventually returns the hash index, 15, in A.



Register B is loaded with the address HASHTB-HSHLNK(A). In this case, with register A containing 15, this is equivalent to the expression HASHTB-HSHLNK+15. At PROCW1, the right half of the word addressed by HSHLNK(B) is fetched. Since B contains HASHTB-HSHLNK+15, this fetches from the right half of the word HASHTB+15. This word is the list head for hash bucket 15. Since the dictionary is empty to start with, this word is zero. The program jumps to PROCW3 where an entry is built for the word HARPO.

At PROCW3, B still contains the expression HASHTB-HSHLNK+15. This expression is saved on the stack. BLDBLK is called to build a dictionary record. The pointer to this record is returned in register A. Register B is restored from the stack. The address of the new record is stored in the word addressed by HSHLNK(B); the effective address of this instruction is HASHTB+15. Hence, the address of the new record for HARPO is stored in HASHTB+15. Bucket 15 is no longer empty.

[4]The hash code for GROUCHO is 143.



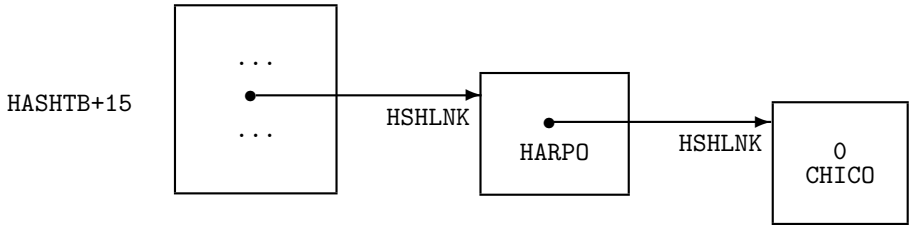
Subsequent to all of this, the word CHICO appears for the first time. PROCWD calls HSHFUN; the hash index, 15, is returned in A.

Again, register B is loaded with the address HASHTB-HSHLNK(A); this is equivalent to the expression HASHTB-HSHLNK+15. At PROCW1, the right half of the word addressed by HSHLNK(B) is fetched; the effective address is HASHTB+15. This word, the list head for hash bucket 15, now addresses the record for HARPO. The address of the HARPO record is loaded into register A.

The NAMCMP routine is called to compare the name CHICO in WRDBUF to the name found in the record addressed by A. Naturally, HARPO doesn't match CHICO; NAMCMP returns without skipping. The program jumps to PROCW2. Note that if the input word were HARPO, a match would have been found, and the occurrence count in the record addressed by A would be incremented.

At PROCW2, register A addresses a record that did not match the input word. The address of this record is copied from A to B; the program jumps back to PROCW1 with B containing the address of the HARPO record. At PROCW1, the word addressed by HSHLNK(B) is fetched; this is the link from the HARPO record to the next record in this hash bucket. There is no next record; this link word contains a zero. The program jumps to PROCW3.

The contents of register B, the address of the HARPO record, are pushed onto the stack at PROCW3. BLDBLK is called to build a dictionary record. The pointer to the new record for CHICO is returned in register A. Register B is restored from the stack. The address of the CHICO record is stored in the word addressed by HSHLNK(B); the effective address of this instruction is the link word in the HARPO record. Thus, the address of the new record for CHICO is stored in the HARPO record.



From this example it should be clear how the search at PROCW1 works. If the search fails, the program adds a new dictionary record at PROCW3.

24.3.1.4 Efficiency Improvements

In situations where searches are frequent, adding some extra complexity to the program can make a further improvement. Often, a small number of objects are looked up with far greater frequency than most other objects. An example where this would happen is in the MACRO assembler. MACRO make frequent searches though the symbol table. Some symbols, such as MOVE and JRST, are accessed more frequently than others (HRLEM, TSCE, etc.). Also, accumulator names tend to be accessed more frequently than other user–defined symbols.

Though in many cases even better techniques exist, one simple technique can often cut down search time dramatically: whenever a record is found, remove it from the list and reinsert it at the beginning of the list. In this way, frequently accessed records cluster near the beginning of each hash bucket list. This technique is especially useful in assemblers and compilers because of the tendency for a programmer to access each particular variable in bursts of activity. That is, for any variable, there may be only one or two small regions of the program where the variable is used. While it is being used, it will be near the front of its hash bucket list. As other symbols are referenced, an inactive symbol will drift toward the back of the list.

24.3.2 GETWRD

The GETWRD subroutine assembles an English word in WRDBUF. At GETWRD, the program scans the input stream looking for a letter. If end of file is detected, GETWRD returns with B set to zero. Non–letters are discarded. The call to GETCHR will also copy each input character to the output file; see the discussion of GETCHR that follows. Letters are detected by the ISLET subroutine. (The ISLET subroutine also converts lowercase letters to uppercase.)

```
GETWRD: MOVEI   B,0                     ;indicate no word so far
        CALL    GETCHR                  ;Get a character
         RET                            ;EOF
        CALL    ISLET                   ;is this a letter?
        JRST    GETWRD                  ;not a letter.  look for start of word
                                        ;A letter is seen: a word begins here
```

As soon as a letter is seen, register B is initialized to point to WRDBUF. The instruction at GETWD1 deposits the letter from A into WRDBUF and continues. At GETWD1 the program is in the middle of assembling a word. GETCHR is called to obtain another character; if end of file is detected, this loop exits to GETWD3 where a null is added to the end of WRDBUF. If the incoming character is a letter, the program loops to GETWD1 where that letter is deposited in WRDBUF.

```
        MOVE    B,[POINT 7,WRDBUF]      ;A word begins here, with a letter.
GETWD1: IDPB    A,B                     ;store characters of the word.
        CALL    GETCHR                  ;get another character
         JRST    GETWD3                 ;return a word and eof
        CALL    ISLET                   ;is this a letter?
         JRST   GETWD2                  ;not a letter.  word ends (probably)
        JRST    GETWD1                  ;stuff letter. get more

GETWD3: MOVEI   A,0                     ;end of word.  terminate WRDBUF
        IDPB    A,B                     ;by stuffing a null in.
        RET
```

When an input character that is not a letter arrives, the loop exits to GETWD2. At GETWD2 we are most likely at the end of the word. Unless the incoming character is an apostrophe, the program will jump to GETWD3 where a null is added at the end of the word; GETWRD returns.

If an apostrophe is seen at GETWD2, the program checks to see if a contraction (e.g., “don’t”) has been input. If a letter follows the apostrophe, then both the apostrophe and the letter are deposited in WRDBUF. The program resumes its search for the end of the word at GETWD1.

```
GETWD2: CAIN    A,"'"                   ;is non-letter an apostrophe?
         CALL    GETCHR                 ;get character following apostrophe
          JRST    GETWD3                ;Return word. (eof or no apostrophe)
        CALL    ISLET                   ;does a letter follow apostrophe?
         JRST    GETWD3                 ;not a letter.  it must be 'word'
        PUSH    P,A                     ;save the letter.
        MOVEI   A,"'"                   ;put the apostrophe back.
        IDPB    A,B                     ;"don't forget to write"
        POP     P,A                     ;get the letter back from the stack
        JRST    GETWD1                  ;and handle it as though normal.
```

24.4 Buffered Input and Output

This program displays a different approach to file input and output. Instead of doing line by line input via the SIN JSYS as we saw in example 13, this program fills a fixed–size buffer from the input on each call to SIN. Output is done in a similar fashion. Instead of calling SOUT to send each line to the output file, this program fills a buffer before calling SOUT. Our intention here is to perform fewer system calls by buffering larger amounts of data for each call. By increasing the amount of useful work done by each JSYS, we reduce the time spent performing unavoidable overhead operations. We gain in efficiency and decrease the running time of the program. However, a certain amount of extra bookkeeping is needed to deal with these buffers.

24.4.1 Buffered Input

The subroutines that relate to buffered input are GTINPF, GETCHR and GETBUF.

This version of GTINPF is very similar to the version seen in example 13. One additional GTJFN flag is used. The GJ%MSG flag may be used in conjunction with the GJ%CFM flag. When filename recognition is requested by the user typing the escape key, a message is typed following the recognized name. This message is in the form of a comment that describes the status of the file, e.g., old file, new file, new generation, etc. This message is for information purposes only.

```
GTINPF: HRROI   A,[ASCIZ/File name for input: /]
        PSOUT
        SETZM   IJFN
        MOVX    A,<GJ%OLD!GJ%SHT!GJ%FNS!GJ%CFM!GJ%MSG>
                                        ;Short form, old file,
                                        ;file name from JFNs in B
                                        ;confirmation and messages
        MOVE    B,[.PRIIN,,.PRIOU]
        GTJFN
         ERJMP  [HRROI A,[ASCIZ/Illegal input file specification/]
                CALL ECOM
                JRST GTINPF]
        MOVEM   A,IJFN                  ;save the JFN
```

Another change is that instead of halting on all errors, this version of GTINPF will retry the GTJFN after typing an appropriate error message. The routine ECOM accepts a string pointer in A. It types

"?ERROR: " and the text of the program–supplied error message. Then it types the system's error message for this occasion. If GTJFN fails, the error recovery code jumps back to GTINPF. However, if a failure occurs during OPENF, the error recovery code must first release the JFN before attempting to get another.

Finally, the GTINPF routine zeros the word called ICOUNT. This word signifies how many unprocessed characters remain in the input buffer; initially, there are none.

```
        HRRZ    A,A                     ;keep only right half
        MOVE    B,[070000,,OF%RD]       ;7 bit bytes, read access
        OPENF
        ERJMP   [HRROI A,[ASCIZ/Cannot Open Input file/]
                 CALL ECOM
                 HRRZ A,IJFN
                 RLJFN
                 ERJMP GTINPF
                 JRST GTINPF]
        SETZM   ICOUNT                  ;no bytes left in the input buffer
        RET
```

The GETCHR routine is designed to obtain one character from the input file. The word ICOUNT contains the count of how many characters remain in the input buffer. This count is decremented by GETCHR. If the count becomes negative, the program jumps to GETCH0 where the input buffer is refilled.

Assuming that ICOUNT remains non–negative, a character is obtained via an ILDB instruction. Null characters, if any, are discarded. The input character is copied to the output file by calling PUTCHR. The upper–case version of the input character is returned in register A; GETCHR returns with a skip.

```
GETCH0: CALL    GETBUF                  ;here if buffer is empty.  refill it
        SKIPG   ICOUNT                  ;Did I get anything?
         RET                            ;no.  this must be the end of file.
GETCHR: SOSGE   ICOUNT                  ;decrement buffer count
         JRST    GETCH0                 ;none left in buffer, go refill it
        ILDB    A,IPOINT                ;gobble a character from the buffer
        JUMPE   A,GETCHR                ;throw away any nulls that may be there
        CALL    PUTCHR                  ;copy the input to the output.
        CAIL    A,"a"
         CAILE  A,"z"
          JRST  CPOPJ1                  ;not lower-case, exit with skip
        TRZ     A,40                    ;convert lower-case to upper-case
        JRST    CPOPJ1                  ;perform the skip return
```

When ICOUNT becomes negative, the input buffer must be refilled. This happens the very first time that GETCHR is called because ICOUNT is initialized to zero. Also, this occurs when, in the course of running the program, all the characters have been removed from the buffer and more characters are wanted.

The buffer is refilled by jumping to GETCH0 where the GETBUF routine is called. GETBUF will refill the input buffer and reinitialize IPOINT and ICOUNT. If ICOUNT is positive (greater than zero) when GETBUF returns, the code at GETCH0 will fall into GETCHR and the next character will be returned. GETBUF will return a zero in ICOUNT to signify that the end of file has been reached; in this case, GETCHR will return without skipping.

The GETBUF routine sets up the usual JFN and string pointer for the SIN JSYS. It sets the character count in C to -IBUFLN*5, meaning that we want to read precisely that many characters. (When we set C negative, there is no need to specify a break character in register D.)

If the ERJMP following the SIN is executed, then either end of file has been reached or some other error has occurred. The code at GBERR tests for end of file. If end of file occurs, it might be the case that a partial buffer was read. The code jumps to GETBF1 where the character count is computed by adding IBUFLN*5 to the contents of register C. When SIN returns, C has been updated (towards zero) to reflect how many characters were read. If C is returned as zero, the buffer was completely filled. If C is smaller than zero, then C shows the amount of room that yet remains. The sum of the contents of C plus the buffer size, IBUFLN*5, is the number of characters that were read. This result is stored in ICOUNT. IPOINT is initialized to address the first character of the input buffer. GETBUF returns.

```
GETBUF: PUSH    P,B                     ;save some ACs
        PUSH    P,C
        HRRZ    A,IJFN                  ;JFN in A,
        HRROI   B,IBUF                  ;string pointer in B
        MOVNI   C,IBUFLN*5              ;negative character count in C
        SIN                             ;read characters
         ERJMP  GBERR                   ;error or end of file.
GETBF1: ADDI    C,IBUFLN*5              ;compute how many characters were read.
        MOVEM   C,ICOUNT                ;store that as ICOUNT
        MOVE    C,[POINT 7,IBUF]        ;set up IPOINT
        MOVEM   C,IPOINT
        POP     P,C                     ;restore ACs and leave
        POP     P,B
        RET

GBERR:  HRRZ    A,IJFN                  ;error from SIN, get status
        GTSTS
        TXNE    B,GS%EOF                ;end of file?
         JRST   GETBF1                  ;go compute character count of partial
                                        ;       buffer
        HRROI   A,[ASCIZ/Input error/]  ;this program doesn't go any further.
        JRST    ESTOP
```

In the case of end of file with a partial buffer, the SIN in which end of file is first detected will cause GETBUF to return a short, but non–zero, character count. These characters will be processed as usual. When the input buffer becomes empty, GETBUF will be called again. This time, SIN will detect end of file without transferring any more characters. Register C will be returned unchanged, containing -IBUFLN*5. Adding IBUFLN*5 to this will result in zero. The zero in ICOUNT will be detected at GETCH0+1 as signifying the end of the file.

24.4.2 Buffered Output

Buffered output is implemented using five routines: GTOUTF, OSET, PUTCHR, PUTOUT, and FINISH.

The GTOUTF has error recovery features similar to GTINPF. As with the GTINPF routine, GTOUTF has some special initialization tasks to perform for buffer management. After obtaining and opening the output JFN, GTOUTF falls into the OSET subroutine which initializes the output buffer:

```
OSET:   MOVE    A,[POINT 7,OBUF]        ;initialize output buffer
        MOVEM   A,OPOINT                ;set up output byte pointer
        MOVEI   A,OBUFLN*5              ;and character count
        MOVEM   A,OCOUNT
        RET
```

OSET sets up the output buffer. It initializes OCOUNT, the number of characters that may be placed in the output buffer, as OBUFLN*5. The byte pointer, OPOINT, is initialized to point just before the first character position in the output buffer area.

Character output is performed by the PUTCHR routine. PUTCHR is called with A containing the character to send to the output file. First, PUTCHR avoids putting null bytes in the output file, by returning immediately if A happens to contain a zero. If the byte in A isn't null, PUTCHR will decrement OCOUNT. If the SOSGE skips, there is room in the buffer for this character: the character is deposited in the buffer via IDPB A,OPOINT and PUTCHR returns.

If OCOUNT becomes negative, the buffer is too full to contain this new character. The program jumps to PUTCH0, where PUTOUT is called. PUTOUT will send the current buffer to the output file and reinitialize OPOINT and OCOUNT so the character that is in A can be placed in the buffer. Assuming PUTOUT performs properly, a fresh buffer will be made available; when PUTOUT returns, the program falls back into PUTCHR once more and places the character in A into the output buffer.

```
PUTCH0: CALL    PUTOUT                  ;send current buffer, reinitialize
PUTCHR: JUMPE   A,CPOPJ                 ;no nulls into buffer
        SOSGE   OCOUNT                  ;decrement byte count
        JRST    PUTCH0                  ;no room available
        IDPB    A,OPOINT                ;store character in the buffer
        RET
```

The PUTOUT routine is called from two places. The usual call is from PUTCH0 where the output buffer has filled up. In this case, OCOUNT is negative one; the full buffer must be sent. In the other case, PUTOUT is called from the FINISH subroutine just before closing the output file. This call is necessary because, in general, there will be a partial output buffer that must be transmitted to the file. In this call to PUTOUT, OCOUNT contains the amount of space left in the buffer; only those characters that have been newly deposited in the output buffer will be sent to the file.

The character count for the buffer, -OBUFLN*5, is loaded into C. If OCOUNT is negative, no change to this number is made. Otherwise, OCOUNT is the amount of empty space remaining in the buffer; this is added to the number in C (moving the number in C closer to zero). If the result is zero, then the buffer is completely empty; the SOUT is avoided.[5] Otherwise, when the SOUT is executed, C will contain precisely the negative of the number of valid characters in the buffer.

After performing the SOUT, the OSET routine is called to reinitialize the OPOINT and OCOUNT variables.

[5]This occurs only if no characters at all have been written to the output file.

```
PUTOUT: PUSH    P,A                     ;save accumulators over this call
        PUSH    P,B
        PUSH    P,C
        HRRZ    A,OJFN                  ;JFN
        HRROI   B,OBUF                  ;string pointer to the buffer
        MOVNI   C,OBUFLN*5              ;-number of bytes to write
        SKIPL   OCOUNT                  ;skip if full buffer (see PUTCHR)
         ADD    C,OCOUNT                ;otherwise, decrease buffer count
                                        ;by the number of chrs remaining
        SKIPE   C                       ;avoid SOUT if the buffer is empty.
         SOUT
          ERJMP [HRROI A,[ASCIZ/Output Error/]
                 JRST ESTOP]
        CALL    OSET                    ;reinitialize pointers and counters
        POP     P,C
        POP     P,B
        POP     P,A
        RET
```

The FINISH routine is called from the main program to finish all output activity. Mostly, FINISH just closes the output file, but prior to closing it, FINISH calls PUTOUT to send the last partial buffer to the output file.

```
FINISH: CALL    PUTOUT                  ;send last output buffer
        HRRZ    A,OJFN                  ;close output file
        CLOSF

        RET
```

24.5 Dictionary and Sort Program

Here is the entire dictionary and sort program. We have yet to discuss sorting and printing the results. This discussion will follow.

```
        TITLE   Dictionary and Sort Program - Example 16
        SEARCH  MONSYM,MACSYM

EXADFL==1                       ;Extended address flag.
                                ;0 for section zero, otherwise extended
DEFINE EXTADR <IFN EXADFL,>     ;macro to assemble for ext addresses
DEFINE ZADR   <IFE EXADFL,>     ;macro to assemble for section 0

ZADR<
        EXTERN  .JBSA
>;ZADR

A=1
B=2
C=3
D=4
W=5
X=6
Y=7
Z=10
P=17

PDLEN==40                               ;stack length
HSHTLN==177                             ;hash table size
OBUFLN==200                             ;output buffer length
IBUFLN==200                             ;input buffer length

EXTADR<
.PSECT  DATA,1001000
>;EXTADR
FREEAD: 0                               ;first free address in memory

NSHEAD: 0                               ;head of the name-sort list
CSHEAD: 0                               ;head of the count-sort list

OPOINT: 0                               ;pointer to the output buffer
OCOUNT: 0                               ;count of room avail. in output buffer
OJFN:   0                               ;output JFN

IPOINT: 0                               ;pointer to the input buffer
ICOUNT: 0                               ;Count of chrs avail. in input buffer
IJFN:   0                               ;input JFN

PDLIST: BLOCK   PDLEN                   ;the stack
HASHTB: BLOCK   HSHTLN                  ;the hash table
OBUF:   BLOCK   OBUFLN                  ;the output buffer
IBUF:   BLOCK   IBUFLN                  ;the input buffer
WRDBUF: BLOCK   40                      ;The word buffer
```

```
;The notation SYMNAM:! in MACRO means the definition of a suppressed
;label.  It is similar to the SYM==expression construct.

;This is the definition of the record made for each distinct word:

DEFS:!  PHASE   0                       ;definitions of record fields
WCOUNT:!        0                       ;occurrence count
HSHLNK:!        0                       ;hash-linkage
NSLINK:!        0                       ;name-sort linkage
CSLINK:!        0                       ;Count-sort linkage
NAMBLK:!                                ;entry name starts here
        DEPHASE
        .ORG    DEFS
EXTADR<
.ENDPS
.PSECT  CODE/RONLY,1002000

DSECT==2                                 ;section for dynamic allocation
>;EXTADR

        SUBTTL  MAIN PROGRAM

START:  RESET
        MOVE    P,[IOWD PDLEN,PDLIST]   ;Initialize stack
        CALL    GTINPF                  ;Get input file
        CALL    GTOUTF                  ;Get output file
        CALL    DINIT                   ;Initialize dictionary
MAIN:   CALL    GETWRD                  ;Get a word from input
        JUMPE   B,INEOF                 ;Jump if this is the end of file
        CALL    PROCWD                  ;Process word
        JRST    MAIN                    ;continue in file

;Here at end of file on input.
INEOF:  HRRZ    A,IJFN                  ;close input file
        CLOSF
         ERJMP  .+1                     ;Closf normally skips.
        SKIPE   A,NSHEAD                ;sort the name-sort list, by name
         CALL   NSSORT
        MOVEM   A,NSHEAD                ;new head of the name sort list
        SKIPE   A,CSHEAD                ;sort the count-sort list, by counts
         CALL   CSSORT
        MOVEM   A,CSHEAD                ;store the new head of the csort list
        CALL    PRDICT                  ;print dictionary, both ways
        CALL    FINISH                  ;clean up the output activities
        HALTF                           ;stop execution here
        JRST    START                   ;in case of CONTINUE, start over
```

```
        SUBTTL  GET A WORD
;read a word from the input file.  Return it in WRDBUF.
;If GETWRD returns with B = 0, it signifies end of file.

GETWRD: MOVEI   B,0                     ;indicate no word so far
        CALL    GETCHR                  ;Get a character
         RET                            ;EOF
        CALL    ISLET                   ;is this a letter?
         JRST   GETWRD                  ;not a letter.  look for start of word
        MOVE    B,[POINT 7,WRDBUF]      ;A word begins here, with a letter.
GETWD1: IDPB    A,B                     ;store characters of the word.
        CALL    GETCHR                  ;get another character
         JRST   GETWD3                  ;return a word and eof
        CALL    ISLET                   ;is this a letter?
         JRST   GETWD2                  ;not a letter.  word ends (probably)
        JRST    GETWD1                  ;stuff letter. get more

GETWD2: CAIN    A,"'"                   ;is non-letter an apostrophe?
         CALL   GETCHR                  ;get character following apostrophe
          JRST  GETWD3                  ;Return word. (eof or no apostrophe)
        CALL    ISLET                   ;does a letter follow apostrophe?
         JRST   GETWD3                  ;not a letter.  it must be 'word'
        PUSH    P,A                     ;save the letter.
        MOVEI   A,"'"                   ;put the apostrophe back.
        IDPB    A,B                     ;"don't forget to write"
        POP     P,A                     ;get the letter back from the stack
        JRST    GETWD1                  ;and handle it as though normal.

GETWD3: MOVEI   A,0                     ;end of word.  terminate WRDBUF
        IDPB    A,B                     ;by stuffing a null in.
        RET

;Test for a letter.  Skips if the character in A is a letter.
;If the character in A is a lowercase letter, it is converted to uppercase

ISLET:  CAIL    A,"a"
         CAILE  A,"z"
          JRST  ISLET1                  ;not lowercase
        SUBI    A," "                   ;convert lowercase to uppercase
        JRST    CPOPJ1                  ;exit with a skip

ISLET1: CAIL    A,"A"
         CAILE   A,"Z"
          RET
CPOPJ1: AOS     (P)                     ;return with a skip.
CPOPJ:  RET
```

```
        SUBTTL  Initialize IO, Error Handling

;Get input file name from terminal, open it for 7-bit bytes.  JFN in IJFN
;Initialize input buffer as empty.
GTINPF: HRROI   A,[ASCIZ/File name for input: /]
        PSOUT
        SETZM   IJFN
        MOVX    A,<GJ%OLD!GJ%SHT!GJ%FNS!GJ%CFM!GJ%MSG>
                                        ;Short form, old file,
                                        ;file name from JFNs in B
                                        ;confirmation and messages
        MOVE    B,[.PRIIN,,.PRIOU]
        GTJFN
         ERJMP  [HRROI A,[ASCIZ/Illegal input file specification/]
                CALL ECOM
                JRST GTINPF]
        MOVEM   A,IJFN                  ;save the jfn

        HRRZ    A,A                     ;keep only right half of JFN
        MOVE    B,[070000,,OF%RD]       ;7 bit bytes, read access
        OPENF
         ERJMP  [HRROI A,[ASCIZ/Cannot Open Input file/]
                CALL ECOM
                HRRZ A,IJFN
                RLJFN
                 ERJMP GTINPF
                JRST GTINPF]
        SETZM   ICOUNT                  ;no bytes available in input buffer
        RET

;Get output file name from terminal, open it for 7-bit bytes.  JFN in OJFN
;Initialize output buffer as entirely available
GTOUTF: HRROI   A,[ASCIZ/File name for output: /]
        PSOUT
        SETZM   OJFN
        MOVX    A,<GJ%FOU!GJ%FNS!GJ%SHT!GJ%CFM!GJ%MSG>  ;Short form of GTJFN,
                                        ;output file gets next generation num
                                        ;AC2 is JFN for reading the file name
                                        ;Require confirmation if user uses
                                        ;file name recognition
        MOVE    B,[.PRIIN,,.PRIOU]      ;read file name from terminal
        GTJFN
         ERJMP  [HRROI A,[ASCIZ/Illegal Output File Specification/]
                CALL ECOM
                JRST GTOUTF]
        MOVEM   A,OJFN                  ;save output JFN
```

```
        HRRZ    A,A                     ;sometimes ugly flags are in left
        MOVE    B,[070000,,OF%WR]       ;7 bit bytes, write access
        OPENF
         ERJMP  [HRROI A,[ASCIZ/Cannot Open Output file/]
                CALL ECOM
                HRRZ A,OJFN
                RLJFN
                 ERJMP GTOUTF
                JRST GTOUTF]
OSET:   MOVE    A,[POINT 7,OBUF]        ;initialize output buffer
        MOVEM   A,OPOINT                ;set up output byte pointer
        MOVEI   A,OBUFLN*5              ;and character count
        MOVEM   A,OCOUNT
        RET

;error handler.  call with A = string pointer to an error message.
ECOM:   PUSH    P,A                     ;save string pointer
        HRROI   A,[ASCIZ/Error: /]      ;print standard heading
        ESOUT
        POP     P,A                     ;restore caller supplied string
        PSOUT                           ;print caller's string
        HRROI   A,CRLF
        PSOUT                           ;print crlf
        MOVEI   A,.PRIOU
        HRLOI   B,.FHSLF
        MOVEI   C,0
        ERSTR                           ;print system's version of error
         JFCL
         JFCL
        HRROI   A,CRLF
        PSOUT
        RET                             ;return to caller

ESTOP:  CALL    ECOM                    ;here to print message and stop
ESTOP1: HALTF
        JRST    ESTOP1                  ;stay stopped

        SUBTTL  Low-Level IO routines

;Put character in output buffer.  Call PUTCHR with character in A.

PUTCH0: CALL    PUTOUT                  ;send current buffer, reinitialize
PUTCHR: JUMPE   A,CPOPJ                 ;no nulls into buffer
        SOSGE   OCOUNT                  ;decrement byte count
         JRST   PUTCH0                  ;no room available
        IDPB    A,OPOINT                ;store character in the buffer
        RET
```

```
;Force current contents of output buffer, if any, to output file.
;called from PUTCHR if buffer fills up, or from the close routine to
;send the last buffer

PUTOUT: PUSH    P,A
        PUSH    P,B
        PUSH    P,C
        HRRZ    A,OJFN                  ;JFN
        HRROI   B,OBUF                  ;string pointer to the buffer
        MOVNI   C,OBUFLN*5              ;-number of bytes to write
        SKIPL   OCOUNT                  ;skip if full buffer (see PUTCHR)
         ADD    C,OCOUNT                ;otherwise, decrease buffer count
                                        ;by the number of chrs remaining
        SKIPE   C                       ;avoid SOUT if the buffer is empty.
         SOUT
          ERJMP [HRROI A,[ASCIZ/Output Error/]
                 JRST ESTOP]
        CALL    OSET                    ;reinitialize pointers and counters
        POP     P,C
        POP     P,B
        POP     P,A
        RET

;Finish output.  Do the necessary clean up operations.
FINISH: CALL    PUTOUT                  ;send last output buffer
        HRRZ    A,OJFN                  ;close output file
        CLOSF
         ERJMP  .+1
        RET

;Read one character from the input file into A.  Skips unless end of file.
;CALL GETCHR

GETCH0: CALL    GETBUF                  ;here if buffer is empty.  refill it
        SKIPG   ICOUNT                  ;get anything?
         RET                            ;no.  this must be the end of file.
GETCHR: SOSGE   ICOUNT                  ;decrement buffer count
         JRST   GETCH0                  ;none left in buffer, go refill it
        ILDB    A,IPOINT                ;gobble a character from the buffer
        JUMPE   A,GETCHR                ;throw away any nulls that may be there
        CALL    PUTCHR                  ;copy the input to the output.
        CAIL    A,"a"
         CAILE  A,"z"
          JRST  CPOPJ1                  ;not lower-case
        TRZ     A,40                    ;convert lower-case to upper-case
        JRST    CPOPJ1                  ;perform the skip return
```

```
;Here to get a buffer full of stuff from the input file.
;CALL GETBUF.  Returns with ICOUNT and IPOINT setup.  ICOUNT=0 means EOF.

GETBUF: PUSH    P,B                     ;save some ACs
        PUSH    P,C
        HRRZ    A,IJFN                  ;JFN in A,
        HRROI   B,IBUF                  ;string pointer in B
        MOVNI   C,IBUFLN*5              ;negative character count in C
        SIN                             ;read characters
         ERJMP  GBERR                   ;error or end of file.
GETBF1: ADDI    C,IBUFLN*5              ;compute how many characters were read.
        MOVEM   C,ICOUNT                ;store that as ICOUNT
        MOVE    C,[POINT 7,IBUF]        ;set up IPOINT
        MOVEM   C,IPOINT
        POP     P,C                     ;restore ACs and leave
        POP     P,B
        RET

GBERR:  HRRZ    A,IJFN                  ;error from SIN, get status
        GTSTS
        TXNE    B,GS%EOF                ;end of file?
         JRST   GETBF1                  ;go compute character count of partial
                                        ;       buffer
        HRROI   A,[ASCIZ/Input error/]  ;this program doesn't go any further.
        JRST    ESTOP

        SUBTTL  Process Word, Hashing

;LOOKUP this word.  Increment counter if it is in the dictionary.
;                   Build a dictionary entry for it, otherwise.

PROCWD: MOVE    B,[POINT 7,WRDBUF]      ;Pointer to the input word
        CALL    HSHFUN                  ;compute hash code for it.
        XMOVEI  B,HASHTB-HSHLNK(A)      ;initial link address (B=prev record)
;Enter PROCW1 with B = Address of Previous Record
PROCW1: SKIPN   A,HSHLNK(B)             ;get link to next item, if any
         JRST   PROCW3                  ;End of list.  Add item.
        MOVE    B,[POINT 7,WRDBUF]      ;byte pointers
        MOVE    C,[POINT 7,NAMBLK(A)]   ;byte pointer to word in current record.
        CALL    NAMCMP                  ;compare input word and record word
         JRST   PROCW2                  ;names are not equal.
        AOS     WCOUNT(A)               ;increment occurrence count
        RET
```

```
;Not found yet, continue search through the hash bucket
PROCW2: MOVE    B,A                     ;not equal. restore current record to B
        JRST    PROCW1                  ;advance to next rec in this bucket

;Here if this word doesn't exist in the dictionary.  Add it.
PROCW3: PUSH    P,B                     ;addr of the last item in hash bucket
        MOVE    B,[POINT 7,WRDBUF]      ;byte pointer new dictionary word
        CALL    BLDBLK                  ;build a record.  record address to A
        POP     P,B                     ;address of last record.
        MOVEM   A,HSHLNK(B)             ;store addr of new item in last record
        AOS     WCOUNT(A)               ;count occurrence of new item
        RET


;Routine to compare two words.  Skips if they are equal.

NAMCMP: ILDB    W,B                     ;compare names.  Get a byte from one
        ILDB    X,C                     ;and a byte from the other
        CAME    X,W                     ;are they the same?
         RET                            ;not the same.  No skip.
        JUMPN   W,NAMCMP                ;both the same. loop unless null.
        JRST    CPOPJ1                  ;both end at the same place.

;The hash computation.  CALL HSHFUN with a byte pointer in B.
;returns the hash code corresponding to this word in A.
;the objective of the hashing function is to distribute data evenly among the
;various hash buckets.  Of course the same word will hash identically each time.

HSHFUN: MOVEI   A,0                     ;accumulate hash value here.
HASH1:  ILDB    C,B                     ;get a character
        JUMPE   C,HASH2                 ;jump if end of string
        IMULI   A,35                    ;multiply by some strange constant.
        ADDI    A,-"A"(C)               ;add chr to the accumulating result
        JRST    HASH1

HASH2:  IDIVI   A,HSHTLN                ;divide the result by the length of the
        MOVM    A,B                     ;hashtable.  Remainder is the hashcode.
        RET                             ;remainder is in range 0 to HSHTLN-1
```

```
;Build entry for a word.  Call BLDBLK with a byte pointer to the new word in B.

BLDBLK: MOVE    A,FREEAD                ;address of first free word in mem
        SETZM   HSHLNK(A)               ;zero hash link out of here
        SETZM   WCOUNT(A)               ;and the occurrence count
        MOVE    C,A                     ;get address of new record.
ZADR<
        ADD     C,[POINT 7,NAMBLK]      ;make byte pointer to string in new rec
>;ZADR
EXTADR<
        ADD     C,[.PO7!NAMBLK]         ;make OWGBP to string in new record
>;EXTADR
BLDBL1: ILDB    W,B                     ;load from the word string
        IDPB    W,C                     ;store in the new record's word area
        JUMPN   W,BLDBL1                ;loop until the null is stored
        ADDI    C,1                     ;advance to next free word
ZADR<
        HRRZM   C,FREEAD                ;store as the next free word in mem.
>;ZADR
EXTADR<
        TXZ     C,77B5                  ;remove OWGBP bits
        MOVEM   C,FREEAD
>;EXTADR
BLDBL2: MOVE    C,NSHEAD                ;the old head of the name-sort list
        MOVEM   C,NSLINK(A)             ;add this to the front of the nsort list
        MOVEM   C,CSLINK(A)             ;and to the front of the count-sort list
        MOVEM   A,NSHEAD                ;store this as the new head of the
        MOVEM   A,CSHEAD                ;name-sort and count-sort lists.
        RET
```

```
;Once-only initialization of the dictionary.
;  Zero all list heads and hash buckets
;  Set up the address of the origin of free space.

DINIT:  SETZM   NSHEAD                  ;no head of the Name sort list
        SETZM   CSHEAD                  ;no head of the count sort list
        SETZM   HASHTB                  ;clear out the hash table
        MOVE    A,[HASHTB,,HASHTB+1]
        BLT     A,HASHTB+HSHTLN-1
ZADR<
        HLRZ    A,.JBSA                 ;address of free space
        MOVEM   A,FREEAD                ;save as free address
>;ZADR
EXTADR<
        SETZ    A,                      ;request a private section
        MOVE    B,[.FHSLF,,DSECT]       ;This fork, section number DSECT
        MOVE    C,[SM%RD!SM%WR!1]       ;1 section with read and write access
        SMAP%                           ;Request that TOPS-20 make the section
         ERJMP  [HRROI A,[ASCIZ/Cannot make section/]
                JRST ESTOP]             ;report an error
        MOVSI   A,DSECT                 ;first address in the new section
        MOVEM   A,FREEAD                ;first address at which to allocate
>;EXTADR
        RET

        SUBTTL  Print the dictionary two ways

PRDICT: MOVEI   A,14                    ;form-feed to make the output
        CALL    PUTCHR                  ;start on a new page
        MOVE    B,[POINT 7,HEADR1]      ;print a heading
        CALL    PUTSTR
        MOVEI   W,NSHEAD-NSLINK         ;traverse the list by name-sort order
PRDIC1: SKIPN   W,NSLINK(W)             ;get link to next
         JRST    PRDIC2                 ;none left
        CALL    PRNREC                  ;print the record
        JRST    PRDIC1                  ;loop

PRDIC2: MOVEI   A,14                    ;another form feed
        CALL    PUTCHR
        MOVE    B,[POINT 7,HEADR2]      ;another heading
        CALL    PUTSTR
        MOVEI   W,CSHEAD-CSLINK         ;traverse by count sort order
PRDIC3: SKIPN   W,CSLINK(W)             ;advance to next
         RET                            ;end of list.  we are done
        CALL    PRNREC                  ;print this
        JRST    PRDIC3                  ;continue traverse
```

```
;routine to print one record.  Record address in W.
PRNREC: MOVE    B,[POINT 7,NAMBLK(W)]   ;pointer to the word name
        CALL    PUTSTR                  ;print the word as a string
        MOVEI   A,11
        CALL    PUTCHR                  ;send a tab character
        MOVE    A,WCOUNT(W)             ;get the count
        CALL    PUTDEC                  ;print the count
        MOVE    B,[POINT 7,CRLF]        ;add crlf to line.  Fall into PUTSTR
;Send a string to the output file.  Call with B=byte pointer to string
PUTSTR: ILDB    A,B                     ;get a byte from source string
        JUMPE   A,CPOPJ                 ;return when source string ends
        CALL    PUTCHR                  ;print characters via PUTCHR
        JRST    PUTSTR                  ;loop

PUTDEC: IDIVI   A,12                    ;decimal print to output file.
        PUSH    P,B                     ;save remainder on stack
        SKIPE   A                       ;skip if all digits have been computed
         CALL   PUTDEC                  ;compute more remainders
        POP     P,A                     ;get a digit
        ADDI    A,"0"                   ;make it a character
        JRST    PUTCHR                  ;send it to the output.

        SUBTTL  SORT

;The following is not a simple sort routine.
;Call with A/ pointer to the (non-null) list to sort.
;Return+1 always with
;          A/ pointer to sorted list.
;
;Sort works by dividing the list in two until the subdivision contains
;only one element.  Then it merges the sublists and rebuilds
;a new list that is sorted and that contains all the original
;elements.  Sort is recursive; it requires N*LOG(N) time plus LOG(N)
;extra stack space.

NSSORT: SKIPN   B,NSLINK(A)             ;get link to next guy
         RET                            ;no next guy. this list is sorted
        MOVE    C,B                     ;tail of the B-list
        MOVE    D,A                     ;tail of the A-list
NSORT1: MOVE    W,NSLINK(C)             ;link out of the B-list
        MOVEM   W,NSLINK(D)             ;store in tail of the A-list
        SKIPN   D,W                     ;skip unless done
         JRST   NSORT2                  ;none left
        MOVE    W,NSLINK(D)             ;link out of the A-list
        MOVEM   W,NSLINK(C)             ;store in tail of the B-list
        SKIPE   C,W                     ;skip if done
         JRST   NSORT1                  ;not done yet.  keep dividing the list
NSORT2: PUSH    P,B                     ;save the B-list
        CALL    NSSORT                  ;sort the A-list
        EXCH    A,(P)                   ;get B-list, save sorted A-list
        CALL    NSSORT                  ;sort the B-list
        POP     P,B
```

```
;Now, the situation is that there are two lists, A and B, both sorted.
;Merge them.  (This really does the hard work).
        MOVEI   D,C-NSLINK              ;list head of result will be C
NSMERG: CALL    NSCOMP                  ;compare Head(A) and Head(B)
         EXCH   A,B                     ;B was smaller
        MOVEM   A,NSLINK(D)             ;store link out of new list.
        MOVE    D,A                     ;advance tail pointer
        SKIPE   A,NSLINK(A)             ;advance in a list.  skip if empty
         JRST   NSMERG                  ;loop.  reduce both lists
        MOVEM   B,NSLINK(D)             ;store rest of B-list in tail
        MOVE    A,C                     ;return sorted result in A
        RET

;Comparison routine for Name Sort.  Skip if the head of the A-list
;alphabetically precedes the head of the B-list.

NSCOMP: MOVE    W,[POINT 7,NAMBLK(A)]   ;pointer to the word at head of A-list
        MOVE    X,[POINT 7,NAMBLK(B)]   ;pointer to the word at head of B-list
NSCOM1: ILDB    Y,W                     ;a byte from the head of the A-list
        ILDB    Z,X                     ;a byte from the head of the B-list
        CAMN    Y,Z                     ;are these characters the same?
        JUMPN   Y,NSCOM1                ;yes: get next char, unless end of both

;Usually Y and Z will be the first different characters of the two strings.
;Compare to see which string is smaller.   Strings end with a null byte, so
;the comparison of "CAT" and "CATASTROPHE" will compare NULL to "A";  NULL
;is smaller.  We also get here if both strings are identical; we don't
;expect to see identical strings at this point in the program.  This routine
;skips for identical strings.

        CAMG    Y,Z                     ;skip if A is larger
         AOS    (P)                     ;A is smaller or equal.  Give a skip.
        RET                             ;B is smaller
```

```
;This sort is essentially identical to the sort at NNSORT.
;The difference comes from using the CSLINK through the records
;and from the different comparison criteria that are applied.

CSSORT: SKIPN   B,CSLINK(A)             ;get link to next guy
         RET                            ;no next guy. this list is sorted
        MOVE    C,B                     ;tail of the B-list
        MOVE    D,A                     ;tail of the A-list
CSORT1: MOVE    W,CSLINK(C)             ;link out of the B-list
        MOVEM   W,CSLINK(D)             ;store in tail of the A-list
        SKIPN   D,W                     ;skip unless done
         JRST   CSORT2                  ;none left
        MOVE    W,CSLINK(D)             ;link out of the A-list
        MOVEM   W,CSLINK(C)             ;store in tail of the B-list
        SKIPE   C,W                     ;skip if done
         JRST   CSORT1                  ;not done yet.  keep dividing the list
CSORT2: PUSH    P,B                     ;save the B-list
        CALL    CSSORT                  ;sort the A-list
         EXCH   A,(P)                   ;get B-list, save sorted A-list
        CALL    CSSORT                  ;sort the B-list
        POP     P,B

;The situation is that there are two lists, A and B, both sorted.
;Merge them.  (This really does the hard work).
        MOVEI   D,C-CSLINK              ;list head of result will be C
CSMERG: MOVE    W,WCOUNT(A)             ;compare Head(A) and Head(B)
        CAME    W,WCOUNT(B)             ;are they equal?
         JRST   CSMRG0                  ;no.  then no secondary key.
        CALL    NSCOMP                  ;alphabetical order if counts identical
        EXCH    A,B                     ;A was larger.  It won't be after this.
        JRST    CSMRG1

CSMRG0: CAMLE   W,WCOUNT(B)             ;Is A the smaller one?
         EXCH   A,B                     ;No, A was larger.
CSMRG1: MOVEM   A,CSLINK(D)             ;store link out of new list.
        MOVE    D,A                     ;advance tail pointer
        SKIPE   A,CSLINK(A)             ;advance in a list.  skip if empty
         JRST   CSMERG                  ;loop.  reduce both lists
        MOVEM   B,CSLINK(D)             ;store rest of B-list in tail
        MOVE    A,C                     ;return sorted result in A
        RET


CRLF:   BYTE(7)15,12
HEADR1: ASCIZ/Dictionary and Counts   Word Order

/

HEADR2: ASCIZ/Dictionary and Counts   Count Order

/

        END     START
```

24.5.1 NSSORT

The NSSORT and CSSORT sort subroutines are especially suited for sorting a list of records. Each routine works by dividing the given list in two until the subdivision contains only one element. Then the sort subroutine will merge the resulting sublists to build a sorted list that contains all the original elements. These sort routines are recursive procedures; they require N*Log(N) time plus Log(N) extra stack space.

These sort routines are quite similar in structure and intent; rather than discuss both NSSORT and CSSORT, we will focus on NSSORT only. As with all recursive subroutines, in order to terminate, NSSORT must be given a simpler problem each time it's called. In NSSORT the complexity of the problem is measured by the length of the list to be sorted. The simplest case is a list of only one element: a list that contains only one element is already sorted.[6] The simplification consists of creating two lists, each with half as many elements as the first list. Then NSSORT sorts each of these shorter lists. This result, two sorted lists, is not satisfactory. These two sorted lists are combined into one sorted list by a process known as *merging*.

NSSORT starts by testing the list to see if it contains one item. If register A points to a list that contains only one item then NSLINK(A) will be zero. If it is zero, NSSORT returns immediately.

```
NSSORT: SKIPN   B,NSLINK(A)             ;get link to next guy
         RET                            ;no next guy. this list is sorted
```

If the list addressed by A contains more than one item, NSSORT will build two shorter lists. The first list will contain all the odd elements from the original list. The second list will contain the even elements. These lists will be approximately half the size of the original input list; the odd list will have either the same number of items as the even list or just one more item than the even list.

The code between NSORT1 and NSORT2 partitions the input list. The odd list head is in A and the even list head is in B. To build the list quickly, there are pointers to the list tails (i.e., the places to which new items will be appended). The list tail pointers are kept in C and D.

When the input list is exhausted, each of the A–list and the B–list will be approximately half the length of the input list. Now, since the A-list and B–list are each simpler (i.e., shorter) than the original list, we call NSSORT twice more: once to sort the A–list and once to sort the B–list.

```
        MOVE    C,B                     ;tail of the B-list
        MOVE    D,A                     ;tail of the A-list
NSORT1: MOVE    W,NSLINK(C)             ;link out of the B-list
        MOVEM   W,NSLINK(D)             ;store in tail of the A-list
        SKIPN   D,W                     ;skip unless done
         JRST   NSORT2                  ;none left
        MOVE    W,NSLINK(D)             ;link out of the A-list
        MOVEM   W,NSLINK(C)             ;store in tail of the B-list
        SKIPE   C,W                     ;skip if done
         JRST   NSORT1                  ;not done yet.  keep dividing the list
NSORT2: PUSH    P,B                     ;save the B-list
        CALL    NSSORT                  ;sort the A-list
        EXCH    A,(P)                   ;get B-list, save sorted A-list
        CALL    NSSORT                  ;sort the B-list (result in A)
        POP     P,B                     ;restore the A-list to B.
```

[6] Zero length lists are handled by SKIPE A,NSHEAD outside the first call to NSSORT.

As NSSORT is recursive, we cannot finish explaining how it works without the reader suspending disbelief and supposing that it *does* work. Having called NSSORT twice for the two short lists, we must suppose that we now have two short, sorted, lists. But we didn't want two lists, so we must combine the two short lists into one list. This is called *merging* the lists, that is, we shall mix items from both lists together into one longer, sorted, list.

It is easy to merge two sorted lists to produce one long sorted list. The reason it is easy is this: the head of the A–list is smaller than any other element in the A–list; the head of the B–list is smaller than anything else in the B–list. Therefore, the smallest element in the merged list must be the smaller of the head of the A–list and the head of the B–list.

We select whichever list head is smaller, remove it from the list that it was on, and add it to the end of the merged result list. If the list from which we have just removed the head is not yet empty, the merge process repeats; the next smallest element will be in one of the list heads. As soon as one list becomes empty, the entire other list is tacked onto the end of the merged result list.

In the code at NSMERG, register D will contain the address of the list tail. It is initialized (at NSMERG-1) to cause register C to become the list head. The code at NSMERG calls NSCOMP to compare the first element in the A–list to the first element in the B–list. If the B–list has the smaller first element, then NSCOMP doesn't skip; the instruction EXCH A,B will be executed: A and B are interchanged so that at NSMERG+2 register A will be the head of the list that has the smaller first element.

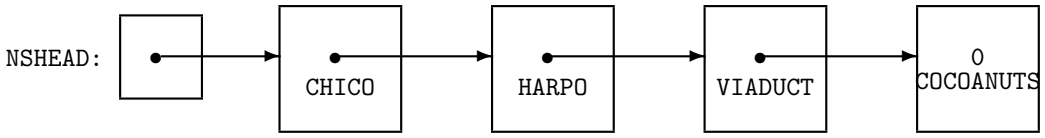
The first element of the A–list is removed and appended (via D) to the end of the output list, whose list head is in register C. D is updated so that it always points to the tail of the C–list. Register A is advanced past the head of the A–list by means of the instruction SKIPE A,NSLINK(A). If the SKIPE doesn't skip, there are more items left on the A–list; the program jumps back to NSMERG. The NSLINK field of the last item in the A–list will be zero. After the last item from the A–list is added to the C–list, this SKIPE instruction will skip. Then the remaining elements of the B–list are appended to the C–list; Register C is copied to A and NSSORT returns.

```
;The situation is that there are two lists, A and B, both sorted.
;Merge them.  (This really does the hard work).
        MOVEI   D,C-NSLINK              ;list head of result will be C
NSMERG: CALL    NSCOMP                  ;compare head(A) and head (B)
         EXCH   A,B                     ;B was smaller, after EXCH, A is smaller
        MOVEM   A,NSLINK(D)             ;store link out of new list.
        MOVE    D,A                     ;advance tail pointer
        SKIPE   A,NSLINK(A)             ;advance in A-List.  skip if empty
         JRST   NSMERG                  ;loop.  reduce both lists
        MOVEM   B,NSLINK(D)             ;store rest of B-List in tail
        MOVE    A,C                     ;return sorted result in A
        RET
```

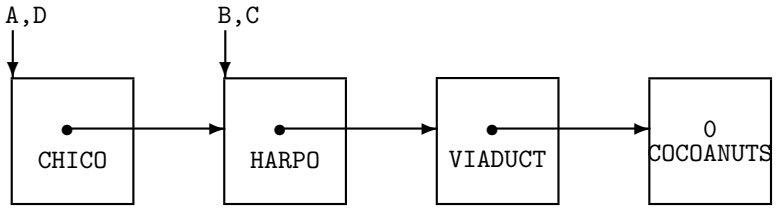
At this point we hope it is clear how NSSORT works. NSSORT divides the input list into two lists, each being half the length of the original. A pair of recursive calls to NSSORT further subdivides these lists. When a list is made that has only one element, NSSORT returns that list.

When NSSORT obtains two short, sorted, lists resulting from the recursive calls to NSSORT, it builds up one longer sorted list by merging these two lists. Eventually, all the pieces are brought back together into one list that contains all the items from the original list in sorted order.

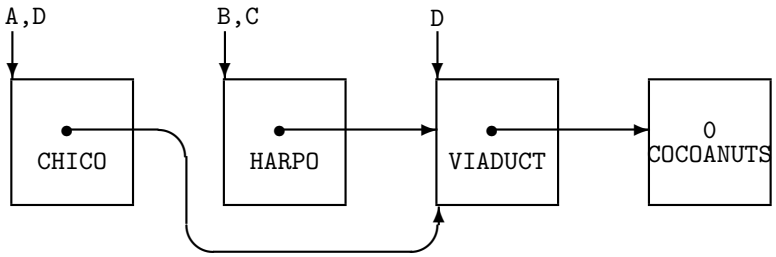
Suppose the original, unsorted list in NSHEAD is



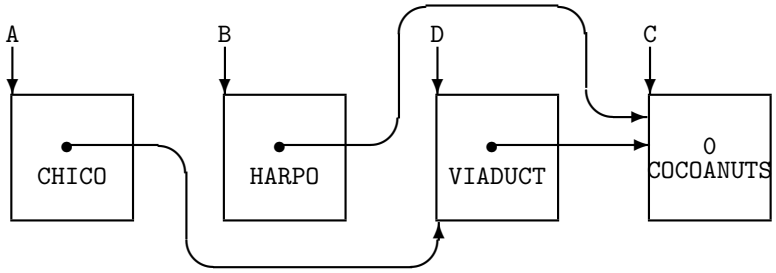
This diagram omits all but the NSLINK and NAMBLK fields of the records. Initially, register A points to the CHICO record. Since the link out of CHICO is not empty, the instructions at NSORT1 will be executed. The first time the program arrives at NSORT1, registers A, B, C, and D will be set up as indicated:



At NSORT1, register W gets a copy of the link out of the record that C addresses. C points to the HARPO record; W is loaded with a pointer to the VIADUCT record. The pointer in W is stored in the record addressed by D. Register D addresses the CHICO record; this changes the link out of the CHICO record to be a pointer to the VIADUCT record. Then register D is changed to point to VIADUCT. All of this is summarized in the following diagram:

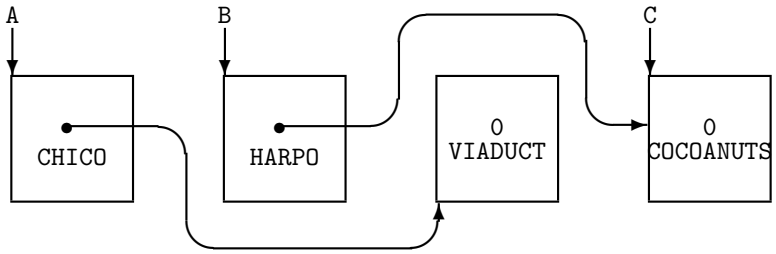


Next, W is loaded with a copy of the link out of the record that D addresses. Register D points to VIADUCT; W is loaded with a pointer to COCOANUTS. Then W is stored as the link out of the record that C addresses. This changes the link out of HARPO to point to COCOANUTS. Register C is loaded with a copy of W, a pointer to COCOANUTS.

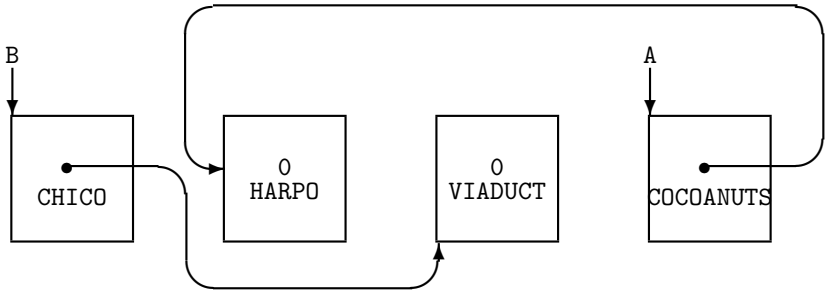


Back at NSORT1, W is loaded from the link out of the COCOANUTS record that is addressed by C. This link is zero, signaling the end of the list. This zero is stored in the VIADUCT record addressed by D. Because W is now zero, the instruction SKIPN D,W will not skip; the program jumps to NSORT2. The

relevant registers, A and B, address the odd and even elements of the original list. The important thing about this result is that each of the lists addressed by A and B are half the size of the original list.



At NSORT2, register B, the pointer to the even list, is pushed on the stack. The odd list, addressed by register A, is sorted by a recursive call to NSSORT. The pointer to the sorted odd list is exchanged with the pointer to the even list that was stored on the stack top. The even list is sorted by a recursive call to NSSORT. Then the pointer to the sorted odd list is popped into register B. The relevant list structures are now as depicted below:

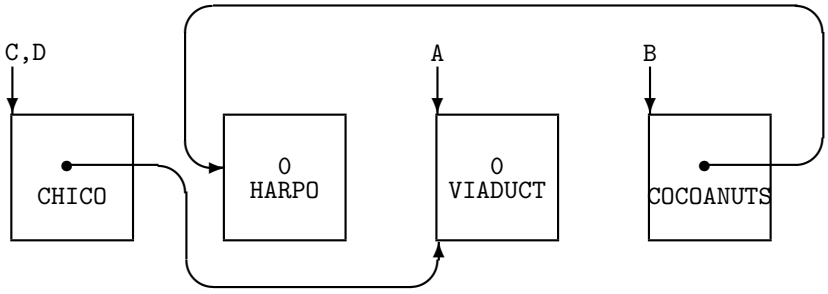


It is unimportant that A and B have exchanged meanings; all that matters is that both are pointers to short sorted lists.

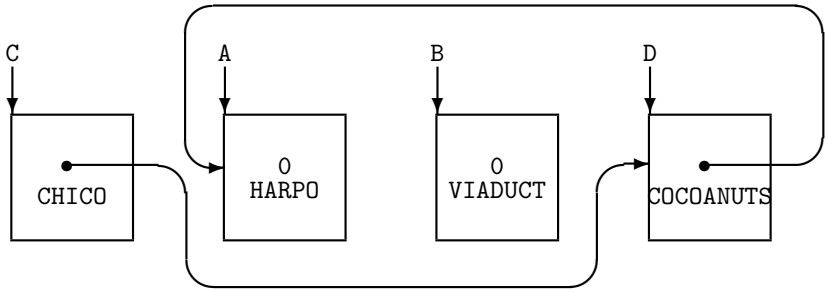
Prior to falling into the code at NSMERG, register D is initialized to the address C-NSLINK. This is another version of the same initialization trick that we first saw in PROCWD. In the loop at NSMERG, D will address the last record in the sorted result list. This initial value will cause register C to be loaded with the address of the first record in the result list.

NSMERG calls NSCOMP; NSCOMP looks at the records addressed by A and B. If register A addresses a record that is alphabetically less that the record that B addresses, then NSCOMP will skip. Otherwise, NSCOMP will return without skipping. The first time that NSCOMP is called, it decides that CHICO is smaller than COCOANUTS; it returns without skipping. The non–skip return from NSCOMP causes A and B to be exchanged. As a result, A now addresses the smaller record, CHICO.

The pointer in register A (to CHICO) is stored in the NSLINK field of the record addressed by D. D was initialized to C-NSLINK. This instruction stores the pointer to CHICO in C. D is then changed to point to CHICO. Then, A is advanced to a copy of the pointer out of CHICO, making A point to VIADUCT. Because the list that A points to has not yet been exhausted, the program jumps to NSMERG again.

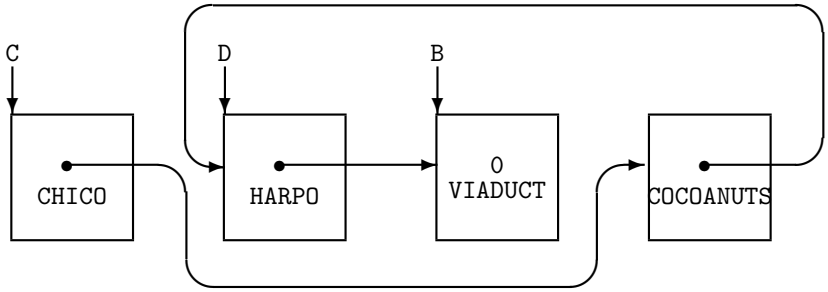


At NSMERG, again the records pointed to by A and B are compared. The COCOANUTS record, addressed by B, is smaller; A and B are exchanged again. The pointer to COCOANUTS is stored as the link out of the CHICO record addressed by D. Register D is then changed to point to COCOANUTS. Register A is advanced to point to the HARPO record following COCOANUTS. Again the program loops to NSMERG.



NSMERG calls NSCOMP to compare the records addressed by A and B. This time, A addresses the smaller one, HARPO, so no exchange is needed. The address of the HARPO record is stored in the link out of the COCOANUTS record that D addresses. (The address of the HARPO record was already present in the COCOANUTS record, but this program does not understand that.) Register D is changed to point to HARPO.

At this point, A is loaded from NSLINK(A), that is, the NSLINK field of the HARPO record. Because the link out of the HARPO record is zero, the program avoids jumping back to NSMERG. Register A no longer points to a list; there is nothing to compare at NSMERG. Instead, register B points to the remaining list of records. This pointer is stored into the HARPO record that register D now addresses.



Now, register C is a pointer to the entire sorted list. It is copied to register A, where it is returned. We hope that the explanation and example are sufficient to display the method by which this sort routine works.

The performance of this method is analyzed as follows: each data item participates once in each of several different "levels" of merge, where the level is defined by the length of the resulting list. Thus, each record participates in one level–2 merge, one level–4 merge, etc., until a level–N merge completes the task of sorting N data items. Each record participates in $\log_2(N)$ distinct levels of merge. Each level of merge eventually involves every data item, and therefore requires on the order of N comparisons. Thus, this sort is another example of sorting that operates in order of $N \times \log(N)$ time.

To contrast this list–oriented merge sort with Heapsort, we can make several observations. Heapsort is advantageous because it does not require any complex linkages between data items. On the other hand, Heapsort requires that the data items (or pointers to them) be placed in a compact array. Building the array for Heapsort is an extra overhead chore that is avoided in the merge sort. Generally, individual circumstances will dictate the which technique to employ. Techniques can be intermixed; for example, one program uses Heapsort to guide the merge of ten sorted lists.

24.5.2 PRDICT

The sorted dictionaries are printed by the `PRDICT` routine. `PRDICT` sends an octal `14`, a form–feed character, to the output file to start a new page. Then a heading is copied to the output.

```
PRDICT: MOVEI   A,14                    ;form-feed to make the output
        CALL    PUTCHR                  ;start on a new page
        MOVE    B,[POINT 7,HEADR1]      ;print a heading
        CALL    PUTSTR
```

The fragment at `PRDIC1` copies the dictionary in name–sort order to the output file. At `PRDIC1-1` register `W` is initialized with the address `NSHEAD-NSLINK`. This is similar to the technique employed at `PROCWD`; an offset is needed the first time through the loop. At `PRDIC1`, register `W` is advanced to point to the next record in the list; if there is no next record, the `SKIPN` will not skip and the program jumps out of the loop to `PRDIC2`. Assuming there is another record, `W` contains its address. The `PRNREC` subroutine is called to print the record that `W` points to. Upon return from `PRNREC`, the code loops back to `PRDIC1` where the program advances to process the next record from the name–sort list.

```
        MOVEI   W,NSHEAD-NSLINK         ;traverse the list by name-sort order
PRDIC1: SKIPN   W,NSLINK(W)             ;get link to next
         JRST   PRDIC2                  ;none left
        CALL    PRNREC                  ;print the record
        JRST    PRDIC1                  ;loop
```

The code at `PRDIC2` and `PRDIC3` is a repetition of the code that prints the name–sort list. The changes reflect the use of a different list head, `CSHEAD`, and a different field name, `CSLINK`. This loop prints the dictionary records in count–sort order.

The `PRNREC` subroutine prints one dictionary record. It prints the string in the `NAMBLK` field of the record addressed by `W`. Then it prints a tab character (octal `11`), the number of occurrences, and a carriage return and line feed.

24.6 Exercises

24.6.1 Token Scanning

A *token* is an identifier name. Write a program to identify and print the tokens on each line of a MACRO language source program.

An identifier in MACRO consists of a "letter" followed by "letters" or "digits." In MACRO a *letter* is any alphabetic character, plus the special symbols period (.), percent (%), and dollar–sign ($). MACRO treats lower–case letters as upper–case letters. Identifiers in MACRO are limited to six characters.[7] We can take advantage of the limitation in the length of identifiers and the small size of the character set to store each identifier name in one computer word by using the sixbit code.

Sample output appears below. This output has been edited to remove some lines that aren't interesting.

```
    1           TITLE   Lexical analysis of MACRO program
TITLE
    2
    3   A=1
A
    4   B=2
B
   14
   15   OPDEF   CALL    [PUSHJ P,]
OPDEF   CALL    PUSHJ   P

   21   PDLIST: BLOCK   PDLEN
PDLIST  BLOCK   PDLEN
   22   IBUF:   BLOCK   200
IBUFH   BLOCK
   23   OBUF:   BLOCK   200
OBUFH   BLOCK

   46   EXTERN  .JBSA
EXTERN  .JBSA
   47
```

[7] A version of MACRO that distinguishes longer token names is in development.

```
   49   START:  RESET
START   RESET
   50           MOVE    P,[IOWD PDLEN,PDLIST]           ;Initialize stack
MOVE    P       IOWD    PDLEN   PDLIST
   51           MOVSI   A,(GJ%OLD!GJ%SHT!GJ%FNS!GJ%CFM!GJ%MSG)
MOVSI   A       GJ%OLD  GJ%SHT  GJ%FNS  GJ%CFM  GJ%MSG
   52           MOVE    B,[.PRIIN,,.PRIOU]              ;Get JFN from TTY
MOVE    B       .PRIIN  .PRIOU
   53           GTJFN                                   ;Get it
GTJFN
   54           ERJMP   JFNERR
ERJMP   JFNERR
   63           CALL    GIFILE
CALL    GIFILE
   66           SETZM   LINNUM                          ;initial line number
SETZM   LINNUM

   68   MAIN:   CALL    DOLINE                          ;process one line
MAIN    CALL    DOLINE
   73   INEOF:  CALL    PRCLIN
INEOF   CALL    PRCLIN
   75           MOVE    A,IJFN
MOVE    A       IJFN
   76           CLOSF
CLOSF
   79           HALTF
HALTF
   81

   82           SUBTTL  GETCHR, PUTCHR, DOLINE, TOKENIZATION
SUBTTL
   83
   84   DOLINE: AOS     A,LINNUM
DOLINE  AOS     A       LINNUM
   85           MOVEI   C,5
MOVEI   C
   86           CALL    DECFIL                  ;print 5 characters
CALL    DECFIL
   87           MOVEI   A,11
MOVEI   A
   88           CALL    PUTCHR
CALL    PUTCHR
```

```
  149
  150   ERREOF: HRROI   A,[ASCIZ/Error- unexpected end of file/]
ERREOF  HRROI   A       ASCIZ
  154   SPCTOK: MOVSI   D,-SPTKLN
SPCTOK  MOVSI   D       SPTKLN
  155   SPCTK1: CAMN    B,SPTKTB(D)
SPCTK1  CAMN    B       SPTKTB  D
  156           JRST    CPOPJ1
JRST    CPOPJ1
  157           AOBJN   D,SPCTK1
AOBJN   D       SPCTK1
  158           RET
RET
  159
  160   SPTKTB: 'COMMEN'
SPTKTB
  161           'ASCIZ '
  162           'ASCII '
  163           'SIXBIT'
  164   SPTKLN==.-SPTKTB
SPTKLN  .       SPTKTB
  165
```

Notice that items that appear in single or double quotes are not tokens. Also, some pseudo–ops need special handling. When you see any of the pseudo–ops `COMMENT`, `ASCIZ`, etc. you must scan to the end of the string without identifying tokens. Comments that follow semicolons should not be looked at while you are identifying tokens. The arguments to `TITLE` and `SUBTTL` should be ignored. There may be some other things you have to do to make your program work properly.

Your program should be able to read a MACRO source file and generate an output file that contains each line from the original file, a line number (just a sequential number to identify the source lines), and, on the output line after each source line, a list of tokens that appear on the source line.

The following is a fragment of a subroutine that you might find useful. It reads characters (using the byte pointer `W`) and assembles sixbit identifier names. It skips leading spaces and tabs. It returns in `B` the identifier that it found, and in `A` the ASCII character that terminates the identifier.

```
GTOKEN: MOVEI   B,0
        MOVE    C,[POINT 6,B]
GTOKE1: ILDB    A,W                     ;get a character
        CAIE    A," "
        CAIN    A,11
        JUMPE   B,GTOKE1                ;toss out leading spaces and tabs
        CAIL    A,"a"
        CAILE   A,"z"
        JRST    GTOKE2                  ;not lower-case
        TRZ     A,40                    ;convert to upper-case
        JRST    GTOKE3                  ;go store this character
```

```
GTOKE2: CAIL    A,"A"
        CAILE   A,"Z"
        CAIN    A,"."                   ;not upper-case.  skip if not "."
        JRST    GTOKE3                  ;upper-case letter or "."
        CAIE    A,"%"
        CAIN    A,"$"
        JRST    GTOKE3                  ;allow "%" and "$"
        CAIL    A,"0"
        CAILE   A,"9"
        RET                             ;return.  A is a delimiter
        JUMPE   B,GTOKE4                ;Digit seen. Jump unless inside a token.
GTOKE3: SUBI    A,40                    ;convert to sixbit
        TLNE    C,770000                ;skip if 6 characters seen already
        IDPB    A,C                     ;store another character in B
        JRST    GTOKE1                  ;go get more.

;Here, you must decide what to do with a digit.
;It must be time to accumulate a number...
GTOKE4:
        ;it seems as though there's something missing.
        ; . . .
```

24.6.2 Cross–Reference Program

Convert the token–scanning program into a "cross–reference" program.

Instead of printing a list of tokens following each line of MACRO source input, collect the tokens and line numbers so that at the end of the source input you can print an alphabetical list of the tokens that you saw, and for each token, a list of line numbers where it was seen.

The desired output is

- A listing of the MACRO language source file that was read by your program. This listing should be augmented by the addition of line numbers to identify each line.
- An alphabetical list of all the different tokens that were seen. Each token should be followed by a list of line numbers showing where that token appears in the input file. Print ten line numbers per line of cross–reference output, in eight columns each. If more that ten instances of a token appears, print extra occurrences ten to a line, aligned with the first line of output.

Sample:

```
A           4      10      32      34      35      36     100 .....
          294     302     312     321    1047
A1         69      74     123
B           5      12   .....
```

24.6.3 KWIC Index Program

A Key–Word In Context (KWIC) index is an alphabetical listing of words (the key–words) including the context in which the key–words appear. Usually, by *context* we mean the entire line in which

the word appears.

Often KWIC indices are used to list the titles of scientific articles. In this example we will display a short selection of science fiction titles.

```
I Will Fear No Evil
Stranger in a Strange Land
Glory Road
The Man Who Sold the Moon
The Moon is a Harsh Mistress
Time Enough for Love
```

Usually in a KWIC index there are specified words that are never considered as key–words. These words include common articles and prepositions such as “a”, “the”, “for”, “to”, etc. The words that remain (shown below in all capitals) are words that should be indexed:

```
I WILL FEAR NO EVIL
STRANGER in a STRANGE LAND
GLORY ROAD
The MAN WHO SOLD the MOON
The MOON is a HARSH MISTRESS
TIME ENOUGH for LOVE
```

Various formats for a KWIC index exist. The program that you should write places every key–word in the same position in the center of the line:

```
                  Time ENOUGH for Love
          I Will Fear No EVIL
                 I Will FEAR No Evil
                        GLORY Road
           The Moon is a HARSH Mistress
                        I Will Fear No Evil
  Stranger in a Strange LAND
        Time Enough for LOVE
                     The MAN Who Sold the Moon
    The Moon is a Harsh MISTRESS
   The Man Who Sold the MOON
                     The MOON is a Harsh Mistress
            I Will Fear NO Evil
                  Glory ROAD
            The Man Who SOLD the Moon
          Stranger in a STRANGE Land
                        STRANGER in a Strange Land
                        TIME Enough for Love
                The Man WHO Sold the Moon
                      I WILL Fear No Evil
```

Write a program to read a file and produce a KWIC index of the lines contained there. The first several lines of the file will start with the word `STOP` and a colon. The remainder of each line is a list of words that you should ignore when it comes to making the index. These words are called the

stop list. The remainder of the file, after the various stop lines, will be a list of titles or phrases to index. Write an output file containing the KWIC index.

A sample input file appears below:

```
STOP: in a
STOP: the is a for of
I Will Fear No Evil
Stranger in a Strange Land
Glory Road
The Man Who Sold the Moon
The Moon is a Harsh Mistress
Time Enough for Love
```

No input line will be longer than `60` characters. Place the key–word that you are indexing starting in column `55` of the output line.

The extra capitalization that appears in the example above was added for emphasis. Your program need not (although it may) capitalize the key–words. Your program should not change other capitalization.

If two lines appear with the same key–word, (e.g., MOON above) it doesn't matter which line is printed first.

Your instructor will tell you what file to read as input.

24.6.4 Set Operations

Write a program to read a file containing set definitions and commands. Write an output file that contains the original commands plus the output requested by the print command.

The program must recognize two commands, set assignment and print.

- In set assignment a set name will be followed by an equal sign. The text to the right of the equal sign will be either a *literal* set in which the elements are enclosed in braces, "`''` and `` ` ``, and explicitly stated, or an expression consisting of two sets (either may be a named set or a literal set) and one set operator. Evaluate the right side of the assignment and build a set with the given name that contains the specified elements.
- The print command will consist of the word `PRINT` followed by a set name, a literal set, or an expression such as found in an assignment. The expression or set following `PRINT` should be evaluated and the members of the resulting set should be printed.

There are three set operations that you must implement. Set intersection, denoted by the ampersand character (`&`); set union, denoted by a plus sign (`+`); and set difference denoted by a minus sign (`-`).

Implement each set as a linked list.

Assume that the members of a set are not sets.

The following is an example of the output file from this program. This is identical to the input file, except for the program's response to each print command:

```
ANIMALS = DOG CAT ZEBRA MOOSE
PEOPLE = GEORGE TOM RICK MOOSE BOB
AP = ANIMALS & PEOPLE
APP = ANIMALS + PEOPLE
PRINT APP
        DOG CAT ZEBRA MOOSE GEORGE TOM RICK BOB
PRINT ANIMALS - ZEBRA BADGER
        DOG CAT MOOSE
```

Chapter 25

Efficient Disk I/O

Files are organized as 512–word pages on the disk; programs are organized as 512–word pages in virtual memory. We can take advantage of the similar representations of programs and files when doing file input and output by *mapping* a selected page of a file into the memory space of the program. The program can read the data from the file by examining the memory locations where the file is mapped. Also, assuming that the program has requested and been granted write access to the file, data that is deposited into the mapped memory area becomes part of the file.

A file page can be mapped to a memory page within a program by means of the `PMAP` JSYS. Since the data structure that the system uses to represent a disk file is similar to the structure that describes a running process, the only function that the system performs in response to the `PMAP` JSYS is to change pointers that describe the process or file. The actual input or output operations are handled by the virtual memory facility within TOPS–20. The operating system does not process the data in any way. Thus, `PMAP` is the most efficient mechanism for disk input and output. In fact, all input and output to the disk is implemented via `PMAP`; when the program performs a `BIN`, a `BOUT`, a `SIN`, or a `SOUT` JSYS, TOPS–20 eventually resorts to `PMAP` to actually move the data between disk and main memory.

Page mapping is used whenever a program is run from an `EXE`, *executable memory–image*, file.[1] Running a program from an `EXE` file is accomplished by mapping file pages into the memory space of a process. When the process is started, the disk pages that contain the program are found to be present at the right places in the process address space. Usually, an `EXE` file is arranged so that if several people are using the same file, they all share the same physical pages in main memory. Typically, an `EXE` file is set up so that when a program attempts to change one of the pages, a private copy of that page is made for that program to change. The other users, if any, who are sharing that page are unaffected.

25.1 Using PMAP for Input

Only files that are resident on the disk can be read with `PMAP`. If a program must be sufficiently general to handle the case of input from devices other than the disk, it would be possible to include extra code after the `GTJFN` to see if the JFN belongs to a disk file or not.[2] If the JFN is not for a

[1] Occasionally, one of us geezers will slip and call this a "core–image" file. Just smile tolerantly.
[2] See the discussion of the `DVCHR` JSYS in [MCRM].

disk file, the input code that we have already seen in example 16 can be used. For simplicity, the code that appears below assumes that the input file will be on the disk.

For input, the PMAP JSYS requires that the programmer specify the following things:

- The JFN of a disk file that is open for reading. As is usual in other forms of input, reading a file page requires that we must request at least read access (OF%RD) in the OPENF call.
- The page number of the file page; it is our responsibility to keep track of where we are in the file.
- The fork handle and page number within the fork where you want the data to appear. Often, we shall use .FHSLF, fork handle to self, to make the data appear in our process. The memory area into which a disk page is read must be aligned on a memory page boundary. Pages are 512 words long; they begin at addresses that are multiples of octal 1000.
- The desired access to the mapped file page in memory. In the call to PMAP, we shall request read access and *copy–on–write* access. Copy–on–write access means that when we write onto the page, we shall get a private copy of the page; the file itself will be unaffected. The filter that we use to remove EDIT line numbers, as explained below, requires that we be able to write on the memory page that holds the input data. (If we did not use this filter, it would be more appropriate to specify only read access to this file page.)
- It is possible to specify a repetition count so that several file pages can be mapped into consecutive memory pages via one PMAP. Using multiple–page PMAP is the most efficient way to do input and output, but we shall keep this example as simple as possible by doing only one page at a time.

To demonstrate the PMAP JSYS (and the extra care that must be taken with its use), we offer the following program fragments to replace portions of example 16.

25.1.1 Initialization

We will augment the initialization code from example 16 to satisfy the additional requirements that PMAP places on us. The initialization code will be added to GTINPF, following the SETZM ICOUNT that is there already.

GTINPF will obtain a JFN and open the file for read access as before.[3] To keep track of our progress through the file we will use the variable IPAGEN; this variable holds the page number of the file page that we want to read. We will initialize IPAGEN to be −1, and increment it before reading each page. Thus the first page we read will be page zero.

```
ISETUP: SETZM   ICOUNT                  ;Count of chars left in page
        SETOM   IPAGEN                  ;"current" page of input file
                                        ;this will be incremented to
                                        ;page 0 before the first PMAP
```

The next requirement is to have a word containing a fork handle and a page number that represents the memory page into which to read the data. Because the input operation is effected by changing the map, an entire page is changed in one operation. The data area being selected for PMAP must be

[3]The byte size specified in OPENF is ignored for input via PMAP.

an entire page; it must be aligned as a page, i.e., starting at an address that is a multiple of octal 1000.

25.1.1.1 Page Alignment in Section Zero

There are several ways to force page alignment. A very simple technique is to place page buffers at pre–determined memory addresses. You could accomplish this by some assignments such as:

```
INPAGN==600                     ;page number of the input page
INADDR==INPAGN_^D9              ;shift page number to form an address
```

This assigns a specific page number to the symbol INPAGN; this page number is then used in PMAP to identify the buffer page. The symbol INADDR is the address of the first word of the buffer page; it is computed by left–shifting the page number by nine bits.

These assignments accomplish the simplest management of page buffers. Since programs and their data are usually loaded into the low addresses in memory, we typically see relatively large numbers being used for page buffer addresses. Be aware that DDT and the TOPS–10 compatibility package (if needed) are resident in addresses above 700000; it would be wise to avoid colliding with them. The disadvantage of this simple approach is that when programs contain dynamically growing regions, or when they must be loaded with other programs, permanently assigned buffer pages can get in the way of other uses of the address space.

A related scheme is to allocate a page buffer among the relocatable components of the program. When we do this, we surrender our ability to specify exactly where the buffer is. To assure page alignment, allocate 1777 words, i.e., one word short of two pages. Somewhere, within the allocated region is a 1000–word block aligned to a page boundary.

```
IBUFRG: BLOCK 1777              ;somewhere within is an aligned page.
INADDR=<<IBUFRG+777>&777000>    ;the address of the aligned page
INPAGN==INADDR_-^D9             ;shift address to form a page number
```

At assembly time, MACRO will not be able to compute values for INADDR or INPAGN. Instead, MACRO will pass instructions to LINK that will be evaluated when the actual location of IBUFRG is known. Of course, LINK will fix the locations in our program where we use INADDR or INPAGN.

In this scheme, if multiple page–aligned objects are required, we would allocate them in consecutive memory locations so that the space needed to achieve alignment is wasted only once.

For this program, we choose to use another way to allocate page buffers. The program takes the left half of .JBSA as the first free location in memory. That location is rounded up to the next multiple of octal 1000 by the sequence:

```
                                ;Section Zero code
HLRZ    A,.JBSA                 ;addr of 1st free memory loc
TRZE    A,777                   ;skip if a multiple of 1000
 ADDI   A,1000                  ;round up to next multiple
                                ;  of 1000.
```

The TRZE instruction will clear the low–order nine bits of A, making A contain a multiple of 1000. If all of those nine bits were zero, the TRZE will skip. Otherwise, the TRZE has truncated A, making it

a smaller number; in this case, 1000 is added to A to advance it to the next multiple of 1000 beyond the first free address.

25.1.1.2 Page Alignment in Non–Zero Sections

One of the difficulties in the section zero program was predicting to what address the data would be relocated. As we control where each psect is loaded, it is natural to align psects to page boundaries and to allocate page–aligned data regions at the beginning of a psect. So, immediately following the (first) .PSECT statement for DATA, allocate one or more pages of buffer space aligned to a page boundary.

```
        .PSECT DATA,1001000     ;start the psect at a page boundary
IPAGE:  BLOCK 1000              ;allocate a page buffer at a page boundary
```

Thereafter, if the code and data are in the same address section the instruction XMOVEI A,IPAGE will load IPAGE's 30–bit virtual address into A. If the code and data are in different address sections (or if this is not known), instead use MOVE A,[IPAGE] to load the address of IPAGE into A.

25.1.1.3 Arguments to PMAP

For either section zero code or extended addressing code continue as follows:

Given that register A now contains the address of the first word on a page, B is loaded from A and converted to the corresponding page number by a 9–bit right shift. The fork handle, .FHSLF, is placed in the left half of this page number word. The combination of a fork handle in the left and a page number in the right is call a *page handle*. The page handle is one of the arguments to PMAP; this is saved as INPAG.

```
        MOVE    B,A                     ;copy to B.
        LSH     B,-^D9                  ;convert to a page number
        HRLI    B,.FHSLF                ;.FHSLF,,Page number in memory
        MOVEM   B,INPAG                 ;an argument to PMAP
```

Extended Addressing Note: In the TOAD systems, a fork's page number may exceed 18 bits. For such a case, the XKL version of TOPS–20 provides an alternate scheme for expressing a page handle. An *Extended Page Handle (XPH)* is a double word. The first word contains a fork handle or JFN in the left half.[4] The second word contains optional access flags (PM%RD, PM%WT, PM%EX, PM%EPN) and an extended page number (XPN), which is a 21-bit page number, in bits 15–35. To use an XPH, place the 30–bit address of its first word in the accumulator, replacing the usual page handle. In the case of the PMAP JSYS, flags in register 3 must be set to announce the use of an XPH: set PM%SXP to indicate that AC1 contains the address of an XPN; set PM%DXP to indicate the same for AC2.

Other JSYSes affected by extended page numbers include XRMAP%, and XRPAC%.

For the present example, we continue with the traditional form of page handle.

In the section zero program, register A still contains the address of the first word on the page. This is converted to a byte pointer for 7–bit bytes and stored in INPNTI. The use of parentheses to swap the quantity POINT 7,0 into the right half of the HRLI instruction is similar to techniques we have

[4] At present, files are limited to 2^{18} pages, so the use of an XPH with a JFN is not necessary. As a future TOPS–20 might allow larger files, the XPH with a JFN is permitted.

applied to floating–point numbers (see Section 16.5.3, page 227). Next, A is advanced past the end of this page by adding 1000 to it; the result is stored back in the left half of .JBSA as the new value of the first free address available to the program.

```
                                ;Section Zero Code
HRLI    A,(<POINT 7,0>)         ;a byte pointer to input page
MOVEM   A,INPNTI                ;initial input pointer
ADDI    A,1000                  ;make room for input page by
HRLM    A,.JBSA                 ;advancing free addr past it
```

For the non–zero section program, if the buffer page is in the same address section as the code, register A can be transformed to a byte pointer as described above. Otherwise, transform A to be a one–word global byte pointer by TXO A,.P07 and store the result in INPNTI:

```
                                ;Non Zero Section Code
TXO     A,.P07                  ;make a byte pointer to input page
MOVEM   A,INPNTI                ;initial input pointer
```

The end of file test that we have used with SIN and BIN cannot be used with PMAP. Therefore, we are forced to use a different method to determine where the end of the file occurs. The SIZEF JSYS is used to obtain the number of pages in the input file; this quantity is stored as IMAXPG. Pages start at page number 0, hence, IMAXPG contains a number that is one larger than the largest page number in the file.[5]

```
HRRZ    A,IJFN
SIZEF
 ERJMP  [HRROI A,[ASCIZ/SIZEF Failure/]
        JRST ESTOP]
MOVEM   C,IMAXPG                ;page count of file.
RET
```

We can remove the definitions of IBUF and IBUFLN from example 16. The following variables must be added:

```
INPNTI: 0                       ;byte pntr to input page
INPAG:  0                       ;.FHSLF,,memory page number
IMAXPG: 0                       ;page count of input file
IPAGEN: 0                       ;input file page number
```

25.1.2 GETCHR

The GETCHR subroutine is very much like the one shown in example 16. It uses ICOUNT as the count of characters that are available in the page buffer. Each time GETCHR is called, it decrements this count. If the count falls below zero, the program jumps to GETCH0 where GETPAG is called to read a new page.

[5]In TOPS–20, it is possible to write a file that contains *holes*, i.e., non–existent pages. For example, a five page file containing pages 0, 1, 37, 602, and 1252 can be written. This program will not be able to read such a file properly; the techniques for dealing with such files are beyond the scope of the present discussion.

```
GETCH0: CALL    GETPAG                  ;time to refill page
         RET                            ;end of file
GETCHR: SOSGE   ICOUNT                  ;decrement input count
         JRST   GETCH0                  ;must read another page
        ILDB    A,IPOINT                ;get next char from this page
        JUMPE   A,GETCHR                ;ignore null characters.
                                        ;process the character.
```

Register A is loaded with a character from the mapped copy of the input page. If the input character is a null (nulls are used by the EDIT program for padding at the ends of some lines), the null is discarded by jumping back to GETCHR. When a non–null character is seen, GETCHR behaves as it did in example 16; the character is sent, via PUTCHR, to the output file. Then the character is converted to upper–case and GETCHR returns with a skip.

25.1.3 GETPAG

GETPAG is called from GETCHR when the current input page has been exhausted. Recall that the initialization sets ICOUNT to zero so that the first call to GETCHR results in a call to GETPAG. GETPAG starts by saving two registers. Then it increments the file page counter, IPAGEN; when this counter reaches the same value as IMAXPG, the program will jump to GETPG1 where it handles the end of file condition. Assuming that file pages remain to be read, the left half of register A is loaded with the JFN; the right half of A already has the next file page number from the incremented IPAGEN.

Register B is loaded with the fork handle and page number that describes where to put the data. If some other page is already mapped to this process page, that mapping will be cleared by PMAP before anything new is mapped in.

Register C is set up with access bits. PM%RD asks for read access; PM%CPY asks for copy–on–write access.

```
GETPAG: PUSH    P,B                     ;get another page of input
        PUSH    P,C
        AOS     A,IPAGEN                ;advance to the next page
        CAML    A,IMAXPG                ;less than maximum page?
         JRST   GETPG1                  ;end of file
        HRL     A,IJFN                  ;JFN,,FILE PAGE
        MOVE    B,INPAG                 ;.FHSLF,,PAGE NUMBER in Memory
        MOVX    C,<PM%RD!PM%CPY>        ;read access and copy on write
        PMAP                            ;map in the page.
         ERJMP  [HRROI A,[ASCIZ/PMAP failure/]
                JRST ESTOP]
```

Assuming that all goes well when the PMAP JSYS is performed, the new file page will be mapped into the process page. When the program examines the contents of that page, it will find data from the file there.

IPOINT is set from INPNTI; ICOUNT is set to octal 5000, being 5 characters per word and 1000 words in this page.

```
        MOVEI   A,5000                  ;count of bytes on this page
        MOVEM   A,ICOUNT
        MOVE    A,INPNTI                ;byte pointer to this data
        MOVEM   A,IPOINT
```

Because this program uses PMAP, it will see all of the data in the file. You will recall that the SIN JSYS filters the EDIT line numbers out of the input stream that the program sees. PMAP does no such filtering; therefore, if we want it done, we must do it ourselves.

An EDIT line number is a 36–bit word that contains five ASCII digits and which has bit 35 set to one. The first character in the word following a line number is a tab character that is also considered to be part of the line number rather than part of the regular data.[6]

This fragment scans the entire input page and zeroes any line numbers that may be present; the explanation follows. For this fragment, the data must be in the same address section as the code.

```
; This code is for section zero or for extended addressing
; when the data is in the same address section as the code.
;
                                        ;scan page remove EDIT line numbers
        HRLI    A,-777                  ;-777,,first address on page
        MOVEI   B,1                     ;Mask to select Bit 35
GETPG0: TDNE    B,(A)                   ;skip if word is not line number
         JRST   GETPG2                  ;This is a line number word
GETPG3: AOBJN   A,GETPG0                ;loop to next wd of this page
        POP     P,C                     ;end of loop. restore
        POP     P,B                     ;accumulators and return
        JRST    CPOPJ1                  ;with a skip.

GETPG2: SETZB   B,(A)                   ;zero the line number word
        DPB     B,[POINT 7,1(A),6]      ;zero the tab that follows
        AOJA    B,GETPG3                ;Restore B.  Go get next word on page
```

We enter this code with register A containing a one–word local byte pointer to the first byte on this page; the right half of A contains the (in–section) address of the first word of this page. The left half of A is set to −777 octal; this creates an AOBJN pointer in A by which we will scan the entire page.[7]

Register B is loaded with the constant 1 which is a mask that selects bit 35, the bit that is present only in line number words.

The loop at GETPG0 selects a word from the page and tests it. In the usual case, the word is not a line number; the TDNE will skip to the AOBJN which advances to the next word of this page and

[6]The rest of the story about line numbers includes an obscure but important restriction: the line number word and the word that follows it must be on the same file page. Otherwise, this program fragment (and many like it) would fail to discard the tab. For compatibility with TOPS–10 (where disk buffers are 200_8 words), the line number word is further constrained: it must not appear as word 177, 377, 577 or 777 in the page; such words being at the end of TOPS–10 buffers. If EDIT were faced with placing a line number in one of these positions, it would fill the word with zero and put the line number in the following word. EDIT delimits pages with a *page mark* that consists of a word containing five blanks with bit 35 set followed by a word that contains a carriage return, a form feed, and three null bytes. Finally, implicit in files written by EDIT is the notion that null bytes ought to be ignored when seen as input.

[7]The count is short by one to avoid looking at the very last word of the page; that word must not be the start of a line number.

loops back to the TDNE. When the control count in register A is exhausted, the scan is complete. At the end of this loop, the registers that were stored at the entry to GETPAG are restored, and GETPAG returns with a skip.

When a word containing a line number is found, the TDNE at GETPG0 will not skip and the program will jump to the code at GETPG2. At GETPG2 the right half of register A will contain the address of the line number word. The SETZB will zero both the line number word that A points to and clear B. Then the zero in register B is deposited in the first character of the word following the line number; this sets the tab character to zero. Having replaced the line number and its tab with null characters, the program uses AOJA to restore the value 1 to B and return to the AOBJN at GETPG3. The effect of this fragment is to replace all line numbers on this page with null bytes. Recall that GETCHR was programmed to discard null bytes.

For extended addressing programs in which the data page is in a different address section than the code, a slightly different approach is used.

```
; This code is for extended addressing when the file data page and the code
; are in different address sections.
;
; Enter with A/ one-word global byte pointer to the first byte on the page
;
                                        ;scan page remove EDIT line numbers
        TXZ     A,77B5                  ;Clear OWGBP bits
        MOVEI   B,1                     ;Mask to select Bit 35
GETPG0: TDNE    B,(A)                   ;skip if word is not line number
         JRST   GETPG2                  ;This is a line number word
GETPG3: ADDI    A,1                     ;advance to next word of this page
        TRNE    A,777                   ;Skip if crossed to the next page
         JRST   GETPG0                  ;continue in this page
        POP     P,C                     ;end of loop. restore
        POP     P,B                     ;accumulators
        JRST    CPOPJ1                  ;return with a skip.

GETPG2: SETZB   B,(A)                   ;zero B and the line number word
        DPB     B,[POINT 7,1(A),6]      ;zero the tab that follows
        AOJA    B,GETPG3                ;Restore B, get next word on the page
```

This code is similar to the section zero example. AOBJN has been replaced with the sequence ADDI, TRNE, and JRST. This code also looks at word 777 on the page, a word that the section zero code avoided. Another possibility is

```
        TXZ     A,77B5                  ;Clear OWGBP bits
XMOVEI  B,776(A)                ;compute the last address to look at
        PUSH    P,B                     ;save final address on stack top
         ...                            ;(Same as above)
GETPG3: CAMGE   A,0(P)                  ;at or past the ending address?
         AOJA   A,GETPG0                ;No.  advance in this page
        ADJSP   P,-1                    ;discard data from stack
        ...                             ;(Same as above)
```

The code at GETPG1 handles the end of file condition. End of file is detected when the file page

counter, IPAGEN, reaches IMAXPG. It is necessary to *unmap* the last file page before attempting to close the input file. To unmap a page, set register A to −1, and register B to the fork handle and page number of the process page being unmapped. The access bits in register C are set to zero. The −1 in register A tells PMAP to clear any mapping corresponding to the specified process page. The page disappears from the process. This routine returns without skipping to signal that the end of file has been reached.

```
GETPG1: SETO    A,                      ;here at end of file. A:=-1
        MOVE    B,INPAG                 ;unmap the present file page
        MOVEI   C,0
        PMAP
         ERJMP  .+1
```

25.1.4 Example 16–A — PMAP for Input

Here we summarize all of the changes necessary to make example 16 use page mapping for file input.

For extended addressing, add IPAGE: BLOCK 1000 immediately following the .PSECT that starts the DATA psect.

Remove the definitions of IBUF and IBUFLN from example 16. Add:

```
INPNTI: 0                               ;created byte pointer to input page
INPAG:  0                               ;.FHSLF,,memory page number
IMAXPG: 0                               ;page count of input file
IPAGEN: 0                               ;current page number of input file
```

Augment GTINPF by adding the following code after SETZM ICOUNT:

```
ISETUP: SETZM   ICOUNT                  ;Count of characters remaining in page
        SETOM   IPAGEN                  ;"current" page of input file
ZADR<
        HLRZ    A,.JBSA                 ;get the first free address
        TRZE    A,777                   ;round to a page boundary
         ADDI   A,1000
>;ZADR
EXTADR<
MOVE    A,[IPAGE]
>;EXTADR
        MOVE    B,A                     ;copy to B.
        LSH     B,-^D9                  ;convert to a page number
        HRLI    B,.FHSLF                ;.FHSLF,,Page number in Memory
        MOVEM   B,INPAG                 ;an argument to PMAP
```

```
ZADR<
        HRLI    A,(<POINT 7,0>)         ;a byte pointer to the input page
        MOVEM   A,INPNTI                ;initial input pointer
        ADDI    A,1000                  ;make room for the input page
        HRLM    A,.JBSA                 ;by advancing free addr past it.
>;ZADR
EXTADR<
        TXO     A,.P07                  ;a byte pointer to the input page
        MOVEM   A,INPNTI                ;initial input pointer
>;EXTADR
        HRRZ    A,IJFN
        SIZEF
         ERJMP  [HRROI A,[ASCIZ/SIZEF Failure/]
                JRST ESTOP]
        MOVEM   C,IMAXPG                ;page count of file.
        RET
```

Replace the existing versions of GETCHR and GETBUF:

```
GETCH0: CALL    GETPAG                  ;time to refill page
         RET                            ;end of file
GETCHR: SOSGE   ICOUNT                  ;decrement input count
         JRST   GETCH0                  ;must read another page
        ILDB    A,IPOINT                ;get the next character
        JUMPE   A,GETCHR                ;flush null bytes.
        CALL    PUTCHR                  ;send a byte to the output
        CAIL    A,"a"                   ;convert to upper-case
         CAILE  A,"z"
          JRST  CPOPJ1                  ;not lower-case
        TRZ     A,40                    ;convert lower-case to upper-case
        JRST    CPOPJ1                  ;perform the skip return

GETPAG: PUSH    P,B                     ;here to get another page of input
        PUSH    P,C                     ;save some registers
        AOS     A,IPAGEN                ;advance to the next page
        CAML    A,IMAXPG                ;less than maximum page?
         JRST   GETPG1                  ;end of file
        HRL     A,IJFN                  ;JFN,,file page
        MOVE    B,INPAG                 ;.FHSLF,,Memory page number
        MOVX    C,<PM%RD!PM%CPY>        ;read access and copy
        PMAP                            ;map in the page.
         ERJMP  [HRROI A,[ASCIZ/PMAP failure/]
                JRST ESTOP]
        MOVEI   A,5000                  ;the count of bytes on this page
        MOVEM   A,ICOUNT
        MOVE    A,INPNTI                ;the byte pointer to this data
        MOVEM   A,IPOINT
```

```
ZADR<
                                        ;scan page; remove EDIT line numbers
        HRLI    A,-777                  ;-777,,first address on page
>;ZADR
EXTADR<
        TXZ     A,77B5                  ;remove OWGBP bits
        XMOVEI  B,776(A)                ;Almost the end of the file page
        PUSH    P,B                     ;put final memory address on stack top
>;EXTADR
        MOVEI   B,1                     ;Mask to select Bit 35
GETPG0: TDNE    B,(A)                   ;skip unless this word is a line number
         JRST   GETPG2                  ;This is a line number word
GETPG3:
ZADR<
AOBJN   A,GETPG0                ;loop to next word of this page
>;ZADR
EXTADR<
CAMGE   A,0(P)                  ;at the end of this page yet?
         AOJA   A,GETPG0                ;no, advance to next word in page
        ADJSP   P,-1                    ;end of loop.  discard ending address
>;EXTADR
        POP     P,C                     ;end of loop. restore
        POP     P,B                     ;accumulators and return
        JRST    CPOPJ1                  ;with a skip.

GETPG2: SETZB   B,(A)                   ;zero the line number word
        DPB     B,[POINT 7,1(A),6]      ;zero the tab that follows
        AOJA    B,GETPG3                ;restore B.  Get next word of the page

GETPG1: SETO    A,                      ;here at end of file.
        MOVE    B,INPAG                 ;unmap the present file page. (A=-1)
        MOVEI   C,0
        PMAP
         ERJMP   .+1
        POP     P,C                     ;restore ACs, no skip for
        POP     P,B                     ;end of file.
        RET
```

25.2 Using PMAP for Output

Many of the considerations attendant to the use of PMAP for input apply also to its use for output. The complexity of dealing with EDIT line numbers in GETCHR is missing from PUTCHR. However, there is new complexity at FINISH for closing the output file.

There are basically two ways to perform file output via PMAP. The most natural way is to use the function of PMAP by which a process page is mapped to a file. Although this use of PMAP is straightforward, it is much less efficient than the method we will use.

We will perform file output by mapping a non–existent file page into the memory space of the program. We will specify read and write access to this page. When we write into the memory page, the file page will be created and the data that we store in memory will appear in the file.

25.2.1 OSET

OSET is analogous to ISETUP. It allocates one page for the mapped image of an output page; it establishes a fork handle and page number in the word at OUPAG. OSET makes an initial byte pointer called OPNTI. It sets the current output page number, OPAGEN, to −1. When done with these initialization chores, OSET falls into the code at OMAP to map file page zero of the output file.

For section zero programs, we again look to .JBSA to identify a memory location suitable for our use. For extended addressing, we imagine that a page–aligned space in the DATA psect (or some other writeable psect) has been reserved for us.

```
OSET:
ZADR<
        HLRZ    A,.JBSA                 ;first free address in memory
        TRZE    A,777                   ;round up to page boundary
         ADDI   A,1000                  ;must add to get to next page
>;ZADR
EXTADR<
        MOVE    A,[OPAGE]               ;get the address of the data page
>;EXTADR
        MOVE    B,A                     ;copy of the page address
        LSH     B,-^D9                  ;convert to a page number
        HRLI    B,.FHSLF                ;add fork handle to SELF
        MOVEM   B,OUPAG                 ;.FHSLF,,Mem page num for PMAP
        HRLI    A,(<POINT 7,0>)         ;mem addr becomes a byte pntr
        MOVEM   A,OPNTI                 ;use in OSET  Init'l Byte Pntr
ZADR<
        ADDI    A,1000                  ;advance to start of next page
        HRLM    A,.JBSA                 ;this is next free mem addr
>;ZADR
        SETOM   OPAGEN                  ;Current output page := -1
                                        ;fall into OMAP to map first
                                        ;output page
```

25.2.2 OMAP

The OMAP subroutine maps a non–existent file page into memory. When we write on the corresponding memory page, TOPS–20 will create the file page and add it to the file. OMAP increments the output file page count, OPAGEN, to advance to a new page. If the file page number equals or exceeds 2^{18}, which is the TOPS–20 maximum file size, the TLNE will not skip and the program will stop with an error message. The output JFN is brought into the left half of register A. Registers B and C are loaded with the fork page pointer and the access bits, respectively. The PMAP JSYS will establish the correspondence between the program's memory page and the file page. OMAP concludes by initializing the byte pointer, OPOINT, and the byte count, OCOUNT.

```
OMAP:   AOS     A,OPAGEN                ;map next non-existent page in
        TLNE    A,-1                    ;skip unless file is too long
         JRST   [HRROI A,[ASCIZ/Output file is too long for TOPS-20
/]
                PSOUT
                JRST ESTOP1]            ;file is too long.  Stop now.
        HRL     A,OJFN                  ;incr file pg cnt,  A gets JFN
        MOVE    B,OUPAG                 ;.FHSLF,,our Memory page num
        MOVX    C,<PM%RD!PM%WR>         ;access with both read & write
        PMAP
         ERJMP  [HRROI A,[ASCIZ/Error from PMAP for output/]
                JRST ESTOP]
        MOVE    A,OPNTI                 ;POINT 7,addr of Memory page
        MOVEM   A,OPOINT                ;save as Output byte pointer
        MOVEI   A,5000                  ;output count =1000 wds/page
        MOVEM   A,OCOUNT                ;  times 5 characters/word
        RET
```

OMAP is called from the initialization routine, OSET, to map file page zero to memory. Also, OMAP is called from the PUTCHR subroutine when the current output page buffer fills up. In this case, the page that has just been filled is unmapped; it no longer appears in the program's memory space, but the data that was there is left in the file page. A blank page is then made available to the program for further output.

25.2.3 FINISH

The FINISH routine is generally responsible for unmapping the last file page and closing the file. There are a number of special bookkeeping chores pertaining to mapped output that must be performed by FINISH.

First, FINISH unmaps the last file page. As we have seen in the case of PMAP for input, unmapping is accomplished by setting register A to −1. Then the process page described by register B is unmapped by the PMAP call.

```
FINISH: SETO    A,                      ;Set A to unmap
        MOVE    B,OUPAG                 ;and B to select the file page
        MOVEI   C,0                     ;no flags
        PMAP                            ;unmap the last file page
         ERCAL  [HRROI A,[ASCIZ/Can't Unmap the last file page./]
                JRST ECOM]
```

Having unmapped the file, we can now close it. It is not possible to close the file while any of its pages are still mapped. We want the file closed, but we must retain the JFN for further operations. We can close the file and retain the JFN by setting the flag CO%NRJ in the left half of A before doing the CLOSF JSYS.

```
HRRZ    A,OJFN                  ;Now, close the file, but
TXO     A,CO%NRJ                ;keep the JFN in CLOSF
CLOSF                           ;close file; keep JFN
 ERCAL  [HRROI A,[ASCIZ/Can't Close the file/]
        JRST ECOM]
```

In addition to the data contained within a file, a file has various *file properties*. Among these properties are such things as the creation date, the name of the user who last wrote the file, the byte size, the number of file pages, the byte count, and many more A file's properties are stored in the file's directory in a structure called a *File Descriptor Block* or FDB.

When a file is read sequentially via the `SIN` or `BIN` JSYS calls, TOPS–20 uses the file's byte count to determine where end of file occurs. The byte count is among the items of information that is kept in the file's FDB. Although the file byte count is updated automatically by TOPS–20 when we use `SOUT` or `BOUT`, no such updates are made when we do output via `PMAP`.

Before we declare that we've finished with this file, we must explicitly change the file byte count to reflect the actual size of the file that we wrote. Unless we fix the byte count, programs that use `SIN` and `BIN` will be unable to read this file.[8]

Before changing the file byte count, it is necessary to determine what the correct value should be. The variable `OPAGEN` contains the page number of the last output page of the file. The first output page is page `0`, so the total number of pages in the output file is one greater than `OPAGEN`. Each page holds `5000` (octal) characters, so `(OPAGEN+1)*5000` is the maximum character count. This must be reduced by the contents of the variable `OCOUNT` because `OCOUNT` is the number of unfilled bytes on the last page. This value is computed into register `C`.

```
AOS     C,OPAGEN                ;last page number used + 1
IMULI   C,5000                  ;times characters per page
SUB     C,OCOUNT                ; - bytes left in last page
```

The file byte count is changed by the system call `CHFDB`, *CHange File Descriptor Block*. This call takes an FDB index in the left half of `A`; the FDB index is the name of the word that we want to change. In this case, the file byte count is held in a word called `.FBSIZ`. The right half of register `A` is the JFN for the file. Register `B` contains a mask that contains a one at each bit position where we want to effect a change. By means of zeros in the mask, we can avoid changing some fields. In this case, the mask is all ones; we want to change the entire word. Register `C` is set up with the new value for the FDB word and the `CHFDB` JSYS is executed to set the file byte count.

```
MOVSI   A,.FBSIZ                ;addr of byte count wd in FDB
HRR     A,OJFN                  ;the JFN
SETO    B,                      ;The mask: change all bits
CHFDB                           ;Change FDB. C has new count
 ERCAL  [HRROI A,[ASCIZ/CHFDB failure/]
        JRST ECOM]
```

Similarly, the file byte size, `7`, must also be set. The name of the appropriate FDB word is `.FBBYV`.

[8]Note that in our example of `PMAP` for input we made no use of this byte count, so that style of input would work correctly.

This time, we are going to change only one six–bit field in that word. Accordingly, B is set to a mask containing six ones. Then C is set to contain 7 in the appropriate field position. This call to CHFDB sets the byte size. TOPS–20 will not allow some fields in the FDB to be changed; it is an error to specify a mask that includes such fields. Finally, the JFN is released via the RLJFN JSYS.

```
MOVSI   A,.FBBYV                ;file byte size word in FDB
HRR     A,OJFN
MOVX    B,77B11                 ;mask denoting bits to change
MOVX    C,07B11                 ;set file byte size to 7
CHFDB
 ERCAL  [HRROI A,[ASCIZ/CHFDB failure/]
        JRST ECOM]
HRRZ    A,OJFN                  ;release the output JFN
RLJFN
 ERCAL  [HRROI A,[ASCIZ/Can't release the output JFN./]
        JRST ECOM]
RET
```

25.2.4 Example 16–B — PMAP for Output

We summarize the changes necessary to transform example 16 to use PMAP for output.

For extended addressing, we assume that OPAGE: BLOCK 1000 appears in the page–aligned region following .PSECT DATA, before any data blocks that are not whole pages are allocated.

Delete the definitions of OBUFLN and OBUF. Add the following definitions:

```
OPAGEN: 0                       ;output file page number
OPNTI:  0                       ;output file initial byte pointer
OUPAG:  0                       ;.FHSLF,,Memory Page number.
```

In GTOUTF, change the argument to OPENF to include read access also:

```
MOVEI   B,OF%WR!OF%RD           ;read and write access
OPENF                           ;byte size is irrelevant
```

Replace the routine OSET as follows:

```
OSET:
ZADR<
        HLRZ    A,.JBSA                 ;first free address in memory
        TRZE    A,777                   ;round up to page boundary
         ADDI   A,1000                  ;must add to get to next page
>;ZADR
EXTADR<
        MOVE    A,[OPAGE]               ;address of the page for output
>;EXTADR
        MOVE    B,A                     ;copy of the page address
        LSH     B,-^D9                  ;convert to a page number
        HRLI    B,.FHSLF                ;add fork handle to SELF
        MOVEM   B,OUPAG                 ;.FHSLF,,Memory page number for PMAP
ZADR<
        HRLI    A,(<POINT 7,0>)         ;convert mem address to a byte pointer
        MOVEM   A,OPNTI                 ;save for OSET.  Initial byte pointer
        ADDI    A,1000                  ;advance to start of next page
        HRLM    A,.JBSA                 ;save this as the next free addr in mem
>;ZADR
EXTADR<
        TXO     A,.PO7                  ;convert mem address to OWGBP
        MOVEM   A,OPNTI
>;EXTADR
        SETOM   OPAGEN                  ;Current output page is page number -1

OMAP:   AOS     A,OPAGEN                ;map next non-existent page in.
        TLNE    A,-1                    ;skip unless file is too long
         JRST   [HRROI A,[ASCIZ/Output file is too long for TOPS-20
/]
                PSOUT
                JRST ESTOP1]            ;file is too long.  Stop now.
        HRL     A,OJFN                  ;bump the file pg count, JFN to LH of A
        MOVE    B,OUPAG                 ;.FHSLF,,Our Memory page number
        MOVX    C,<PM%RD!PM%WR>         ;access with both read and write
        PMAP
         ERJMP  [HRROI A,[ASCIZ/Error from PMAP for output/]
                JRST ESTOP]
        MOVE    A,OPNTI                 ;POINT 7,address of Memory page
        MOVEM   A,OPOINT                ;save as Output deposit pointer
        MOVEI   A,5000                  ;output count is 1000 words/page times
        MOVEM   A,OCOUNT                ; 5 characters/word
        RET
```

Replace the existing PUTCHR and PUTOUT with new routines:

```
;Enter at PUTCHR to output one byte.  Calls OMAP when page fills up.

PUTCH0: PUSH    P,A                     ;Output page is now full.  Save ACs
        PUSH    P,B                     ;Map the next non-existent page
        PUSH    P,C                     ;of the output file into this memory
        CALL    OMAP                    ;space.  Initialize pointers and counts
        POP     P,C                     ;restore ACs.  PUTCHR now has a new
        POP     P,B                     ;output page to work with
        POP     P,A
PUTCHR: JUMPE   A,CPOPJ                 ;Throw out nulls
        SOSGE   OCOUNT                  ;Decrement character count for this page
         JRST   PUTCH0                  ;no room left.  send this page
        IDPB    A,OPOINT                ;store character in the page
        RET
```

Replace the FINISH routine as indicated.

```
FINISH: SETO    A,                      ;First, unmap the last file page.
        MOVE    B,OUPAG
        MOVEI   C,0
        PMAP
         ERCAL  [HRROI A,[ASCIZ/Can't Unmap the last file page./]
                JRST ECOM]
        HRRZ    A,OJFN                  ;Now, close the file, but
        TXO     A,CO%NRJ                ;keep the JFN in CLOSF
        CLOSF                           ;close file; keep JFN
         ERCAL  [HRROI A,[ASCIZ/Can't Close the file/]
                JRST ECOM]

;the actual number of bytes written to the file is given by the expression:
;       (OPAGEN+1)*5000 - OCOUNT.
;This is because OPAGEN+1 is the actual number of file pages used,
;at 5000 bytes/page, and OCOUNT is the number of unused bytes on the last page.
        AOS     C,OPAGEN                ;last page used +1 = # of file pages.
        IMULI   C,5000                  ;times characters per page
        SUB     C,OCOUNT                ;less avail bytes from the last page.
```

```
;now change the file byte size to reflect this computed value
        MOVSI   A,.FBSIZ                ;address of the byte count word in FDB
        HRR     A,OJFN                  ;the JFN
        SETO    B,                      ;The mask - change all bits there
        CHFDB                           ;Change FDB.  Register C has new count
         ERCAL  [HRROI A,[ASCIZ/CHFDB failure/]
                JRST ECOM]
        MOVSI   A,.FBBYV                ;file byte size word in FDB
        HRR     A,OJFN
        MOVX    B,77B11                 ;mask denoting bits to change
        MOVX    C,07B11                 ;set file byte size to 7
        CHFDB
         ERCAL  [HRROI A,[ASCIZ/CHFDB failure/]
                JRST ECOM]
        HRRZ    A,OJFN                  ;release the output JFN
        RLJFN
         ERCAL  [HRROI A,[ASCIZ/Can't release the output JFN./]
                JRST ECOM]
        RET
```

25.3 PMAP for Update–in–Place

PMAP has more uses than those we have mentioned thus far. PMAP can be used to access a file in a mode called *update–in–place*. Update–in–place is unlike other forms of file writes in which a new generation of the file is created. In this mode of access, the individual pages of an existing file are changed; no backup of an old generation is kept. Where appropriate, update–in–place is faster and more efficient than recopying the file to make changes.

To obtain update–in–place access to a file, get a JFN for an existing file; specify both read and write access in the call to OPENF. Then a PMAP call that specifies both read and write access will map an existing file page into the memory space of the process. Any changes made by writing on the mapped–in page will be made to the file page.

A related use of PMAP makes it possible for several processes to be updating the same file simultaneously. In fact, simultaneous update to a shared file page is a very efficient way to pass information between two processes, even when the processes are in diffrent jobs. We will discuss how this can be done in Section 28.3, page 522.

25.4 PMAP for Sharing Process Pages

PMAP also can be used to share information between several processes of one job. Discussion of this aspect of PMAP will be deferred to Section 27. Within one process, sometimes it is necessary to map one page into a second location. This can be done with PMAP. For example, the following fragment makes page 740 a copy of page 0:

```
MOVSI   1,.FHSLF                ;Source is this fork, page 0
MOVE    2,[.FHSLF,,740]         ;Destination is page 740
MOVX    3,<PM%RD!PM%WR>         ;Request read & write access
PMAP
```

After this PMAP, a reference to a location on page 740 will actually refer to that word on page 0. Thus, if location 405 contains the number 5, the instruction AOS 740405 will change the contents of 405 to 6. The locations are precisely equivalent: a change to one appears in both places.

A reference to 740010 will reference the memory word corresponding to word 10 on page 0, but this is not accumulator 10. There are simply sixteen words on page zero that are normally inaccessible; these words are not related to the actual accumulators.

(The accumulators cannot be mapped elsewhere in the address space, although in the extended machine the global addresses in the range 1000000 through 1000017 are aliases for the accumulators.)

25.5 Random Access Input and Output

It should be evident from the examples of using PMAP for input and output that we have effected sequential access to the pages of file by counting the page number variables, IPAGEN and OPAGEN. Random access can be obtained by using PMAP with arbitrary page numbers. That is, it is possible to read page 17 of file with one PMAP, without first having to read pages 0 through 16. It is also possible to write files in which pages do not appear in sequence. As PMAP is the most efficient form of disk input and output, we recommend it also for random access to files.

It is also possible to do random access via the string and byte I/O calls. The *file byte number* or *file pointer* is kept internally by TOPS–20 as the pointer to the current file location. Each time a byte is read or written, this pointer is advanced. On input, end of file occurs when the file pointer equals or exceeds the byte count of the file. In an output file, the file byte count will be set from the maximum value that the file pointer reaches during the process of writing the file. Various JSYS operations exist that allow the program to manipulate the file byte pointer. Random access I/O can be effected by changing this pointer.

Please note that there is only one file pointer for each JFN; it is shared for both input and output. When a program is doing random input and output to the same file, it will be necessary to set the file pointer appropriately as the program switches between input and output.

The JSYS calls RIN and ROUT are analogous to BIN and BOUT. The RIN JSYS requires a *file byte number* in register 3 and a JFN, as usual, in register 1. The specified file byte is read into register 2. Moreover, subsequent input via SIN or BIN will continue sequentially from this position. The ROUT JSYS requires a file byte number in register 3, and a JFN in register 1. The specified file byte is written from register 2. Subsequent output via SOUT or BOUT will continue sequentially from this position.

Two more JSYS calls that are relevant to random access byte I/O are RFPTR, *Read File Pointer*, and SFPTR, *Set File Pointer*. Given a JFN in register 1, RFPTR returns the current file pointer in register 2; RFPTR normally skips. The SFPTR JSYS allows the file pointer to be positioned at any point in the file. Load register 1 with the JFN and register 2 with the desired file pointer. After SFPTR returns, subsequent input or output will start at that position. Register 2 can be set to −1 to advance the pointer to the current end of file. Consult [MCRM] for further details.

Read 4096 Words of ASCII Text

Multiple–page PMAP costs:	Single page PMAP costs:
1 JSYS overhead	8 JSYS overhead
8 Page map times	8 Page map times
SIN (words) costs:	SIN (characters) costs:
1 JSYS overhead	1 JSYS overhead
8 Page map times	8 Page map times
4096 Word copy times	20480 Character copy times
BIN (words) costs:	BIN (characters) costs:
4096 JSYS overheads	20480 JSYS overhead times
8 Page map times	8 Page map times
4096 Word copy times	20480 Character copy times.

Table 25.1: Alternative I/O Techniques

25.6 Multiple–Page PMAP

The most efficient way to use PMAP for disk file input and output operations is to have it map several pages by one JSYS call. To specify a multiple–page operation, the flag PM%CNT is included with the access flags in the left half of register 3 and a repetition count is placed in the right half of that register. Successive file pages are mapped to (or from) successive memory pages. The programming needed to cope with multiple–page PMAP is only slightly more complex than the examples we have been through; it is left as an exercise.

In general, all input and output operations are expensive. It is important that the programmer choose the most appropriate JSYS and mode. Suppose we had to write a program to read one 4096–word record of ASCII characters. There are six alternatives that immediately spring to mind. These are compared in Table 25.1.

Observe that the best choices involve PMAP, particularly the multiple–page PMAP. This should not be surprising since TOPS–20 does all disk input and output via PMAP. When a program does a BIN or a SIN JSYS, TOPS–20 reads the file by mapping it into pages that are considered to be part of the process that requested the input; TOPS–20 then copies one byte or a string of bytes from that page to the location specified by the JSYS.

Multiple–page PMAPs that specify *preloading* the mapped pages allow the rotational position optimization in the TOPS–20 disk service to work to fullest advantage. Normally, TOPS–20 will execute PMAP by simply setting the process map so that it points to the file page; the input operation is postponed until an actual reference is made to the process page corresponding to a mapped file page. When the PMAP call includes the PM%PLD flag bit, TOPS–20 does not wait for the program to touch the page; instead, it starts the input operation immediately, loading the pages into physical memory before they are actually needed. When consecutive file pages are allocated in consecutive disk pages, a considerable savings in elapsed time can result from preloading the file pages.

We offer a specific example of the advantage of preloading pages with a multiple–page PMAP. An RP06 disk holds five pages on each *track* of the disk, i.e., as the disk makes one revolution, five pages pass in sequence under the read head. Suppose those five pages are consecutive pages of one file. TOPS–20 can fulfill a single page PMAP request in approximately 11.66 milliseconds; this figure allows 8.33 milliseconds as the average rotational latency (half a rotation) and 3.33 milliseconds for

the page transfer time. A second PMAP that requests the next adjacent physical page on the disk will require 20 milliseconds: by the time the second request is made by the program, the disk will have revolved away from the second page; the program will be delayed by one full revolution of the disk plus one page transfer time. If five pages are read by single page requests, the total elapsed time will be $11.66 + 4 \times 20 = 91.66$ milliseconds.

In contrast, when a request is made to preload all five pages, the program will have to wait the same 11.66 milliseconds for the first page, but the second through fifth pages will be picked up on the same revolution of the disk, in the next 4 * 3.33 milliseconds, for a total elapsed time of only 25 milliseconds.

The optimum number of pages to preload in one PMAP is approximately 8 to 12. It should be noted that preloading pages will cause the working set size of the program to increase. In small or heavily loaded systems, the increased I/O throughput of the program must be balanced against the possibility of "thrashing" due to too large an aggregate of working set sizes.

25.7 Exercises

25.7.1 Use of Multiple–Page PMAP

Convert the file read and write subroutines given in this section to map five pages for each PMAP call.

25.7.2 Use the File Size to Detect End of File

Some programs write files in which the file size is shorter than the actual number of pages in the file. These programs "hide" extra information after the nominal end of file.

Change the given example to use the file byte count to determine the end of file.

You may wish to use the GTFDB JSYS to examine the file byte size (a field of the .FBBYV word) and the file byte count (the .FBSIZ word) to determine the number of seven–bit bytes in the file.

25.7.3 Reading Files with "Holes"

We have mentioned that by means of PMAP it is possible to create a file that has non–contiguous pages. A gap between extant pages is called a hole. Write a program to identify which pages of a file are extant. You may wish to consult the description of the FFUFP JSYS in [MCRM].

Chapter 26

Command Scanning

The TOPS–20 operating system implements a set of JSYS calls to perform command scanning. Although command scanning can be done by user–written modules, the use of COMND and related JSYS calls provides a uniform style of interface among many different programs. Users are more comfortable when they deal with only one style of command interface. The COMND JSYS provides a powerful, uniform mechanism that includes convenient command completion, help, and prompting.

The example program that we shall discuss demonstrates the use of the COMND JSYS. However, before we can get started on our explanation of command scanning, it is necessary to look again inside the the MACSYM macro package and use some of the macros contained there.

26.1 Field Definition Macros

Previously we have shown MACSYM macros that are useful for manipulating flags composed of single bits. In this section, we will examine some macros that can deal with fields of data. In this context, a *field* is a byte that when taken as an entity specifies a value or a function number. An example of a field is the byte size field in the OPENF JSYS, bits 0:5 of register 2.

Rather than remember the width and position of this field whenever we write a program, we can define a macro for the field, giving a name and the description of the field. Then, by mentioning the field name and the value we wish to have placed in that field, we can cause the right thing to be assembled. In the case of OPENF this field is called OF%BSZ; it is defined in MONSYM as 770000,,0, that is, one bits wherever the field exists.

When we want to set that field to a particular value, say 7, we can use the FLD macro. FLD(7,OF%BSZ) will adjust the given value to be right adjusted inside the specified field.

FLD is defined in the MACSYM package; it is interesting to look at how it works. In order to build the FLD macro, we need to know the bit number of the rightmost bit of the field. There is an interesting way to discover this value. A *field mask* is some number of consecutive one bits in a word, with zeros everywhere else:

```
MASK        0...001111100...0
```

The one's complement of this mask has zeros where the mask has ones, and vice versa. The two's complement of the mask (i.e., the arithmetic negative of the mask) is one more than the one's

complement:

```
              MASK       0...001111100...0
    One's Complement     1...110000011...1
    Two's Complement     1...110000100...0
```

When the bitwise AND of the mask and its two's complement is taken, the result is a single bit aligned with the rightmost bit of the mask.[1] This expression is written as <<MASK>&<-<MASK>>> in MACRO. We have mentioned before that the ampersand character, &, signifies the bitwise AND operation in MACRO. The pointed brackets group operands (as parentheses do in mathematical notation) to force MACRO to evaluate the expression in the order we require:

```
                       MASK       0...001111100...0
    One's Complement of MASK      1...110000011...1
    Two's Complement of MASK      1...110000100...0
          <<MASK>&<-<MASK>>>      0...000000100...0
```

We now have an expression that has only one bit. If we could compute the bit number of this bit, we could feed that result into a B shifting operator to adjust the value into the given field. Given a non–zero argument, the JFFO instruction will return the bit number of the leftmost one bit. MACRO invokes JFFO when we write the unary operator ^L before the argument expression.[2] (This is written as two characters, "^" and "L", it is *not* CTRL/L.) Thus the expression

```
        ^L<<MASK>&<-<MASK>>>
```

results in the bit number of the rightmost bit of the mask. We plug this expression into the POS, i.e., *POSition*, macro:

```
DEFINE POS(MASK)<<^L<<MASK>&<-<MASK>>>>>
```

To give a specific example, the value of POS(OF%BSZ) is <5>.

After the previous explanation, the definition of FLD must be considered charming in its simplicity:

```
DEFINE FLD(VAL,MSK)<<<<VAL>B<POS(MSK)>>&<MSK>>>
```

The first argument, the given value, is shifted appropriately by the B shifting operator. The amount to shift is specified by the result of the POS macro applied to the mask. The shifted value is bitwise ANDed with the given mask to prevent the data from spilling over into other fields of the result.

An example of the FLD macro appears below:

```
        MOVX    B,FLD(7,OF%BSZ)!OF%RD
```

This example assembles as

```
        MOVE    B,[070000200000]
```

[1] This technique will select the rightmost one bit of any non–zero value.

[2] By the way, the value of ^L0 is decimal 36.

26.2 SALL Pseudo–op

When we use macros extensively in our programs, there is a possibility that the text of the macro expansion will cause the listing or cross–reference file to become hopelessly cluttered. We can use the SALL pseudo–op in MACRO to suppress the listing of the macro expansions. We normally include SALL in our programs that make extensive use of these macro packages. Using SALL generally results in the tidiest listing files.

The XALL pseudo–op can be used to make MACRO revert to its normal state in which macro expansions are partially listed. If you must see more than the partial listing of macro expansions, you may use the LALL pseudo–op to force the listing of all macro expansions.

26.3 Programming Using the COMND JSYS

Now that we have some macros to help us, we can turn to an examination of the COMND JSYS. The prologue that we have just been through is representative of the complexity of the COMND JSYS itself; [MCRM] takes more than twenty pages to describe all the features of the COMND JSYS. The complexity of the COMND JSYS is due to its wide range of function. We strive to perfect the interface that a program presents to its users. Interactions between programs and people are quite complex: the general mechanism by which these interactions take place is implemented by a large and complex program that needs more than a superficial explanation.

What functions must we use when processing a simple command? Suppose we want to implement a command called COUNT that makes the program count up or count down. Some examples of the COUNT command follow:

```
COUNT UP (TO) 10              !Counts from 1 to 10
COUNT DOWN (FROM) 25          !Counts from 25 down to 1
COUNT 12                      !Counts from 1 to 12
COUNT                         !Counts from 1 to 20
```

These examples illustrate several ideas. The words COUNT, UP, and DOWN are called *keywords*. A keyword is one of a small number of words that are acceptable at various points in the command. The words in parentheses, FROM and TO, are called *guide words* or *noise words*. A noise word is a word or phrase that is present in the command to provide guidance to the user. Noise words never need to be typed, but they will appear when a field (such as UP or DOWN) is terminated by the user typing the escape key. When the user types escape, assuming the partially specified field is unique, the remainder of the keyword will be typed; any noise word that follows the keyword will also be typed.

From the example, it seems that the command COUNT 12 means the same as COUNT UP (TO) 12. This implies that when some fields are omitted they can *default* to a specified value or condition. In this case, the *default value* for the direction to count is upwards. Numeric arguments can be defaulted also. When the number is omitted from the COUNT command, the program seems to default to 20 as the upper limit.

The interesting thing is to see how to implement these various options by means of the COMND JSYS. Fundamentally, each call to COMND will parse one field of the command. By *parse* we mean the program will resolve the entire command into its component fields and attach some meaning to each field. The concept of parsing a command will become somewhat more definite as we go on.

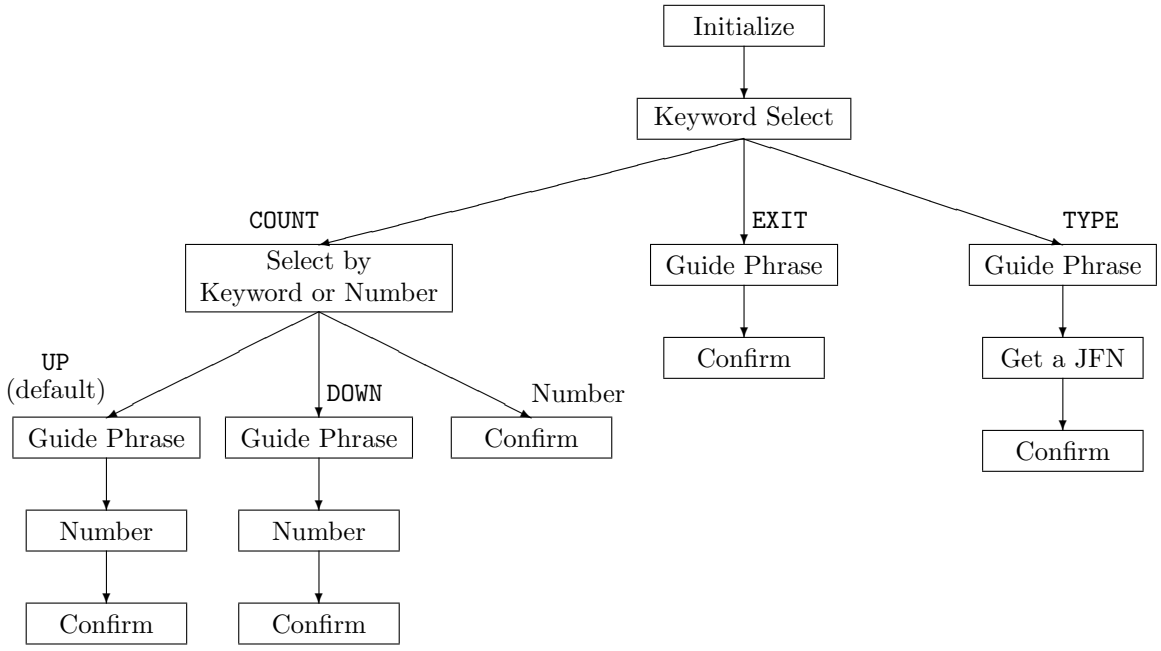


Figure 26.1: Parsing a Command

Because the COMND JSYS deals with only one field at a time, it is necessary to repeat this JSYS several times in order to obtain and understand a complete command line. Therefore, our program will necessarily call COMND several times. Each call will progress, field by field, toward the end of the command line. When the end is found, the program should execute the functions appropriate to implement the command that has been found. For reasons we shall explain below, it is necessary for the program to wait until the end of line has been found before taking irrevocable steps towards executing the command.

Another view of command processing is as a descent through a tree of possibilities. The command structure implemented by a program can be thought of as a tree; in parsing a command, COMND selects one branch of the tree at each call. The partial command tree for this program is depicted in Figure 26.1. Each call to COMND is represented by a box; the text within a box describes what kind of command element is being sought at that point. The components of this diagram will be more fully explained as we continue this discussion.

26.3.1 Initialization for COMND

As the user types a command, the characters are placed in a *command text buffer*. This buffer can also include the command line prompt, if any. Several byte pointers and counts reflect the current state of the parsing of the command. These pointers and counts are as follows:

.CMRTY This is a byte pointer to the beginning of the prompting text buffer. This pointer is also called the CTRL/R–buffer byte pointer since it indicates the initial part of the text that will be output in response to the user typing a CTRL/R. (The remainder of the text output in response to a CTRL/R is what the user had typed before he typed CTRL/R.) The buffer

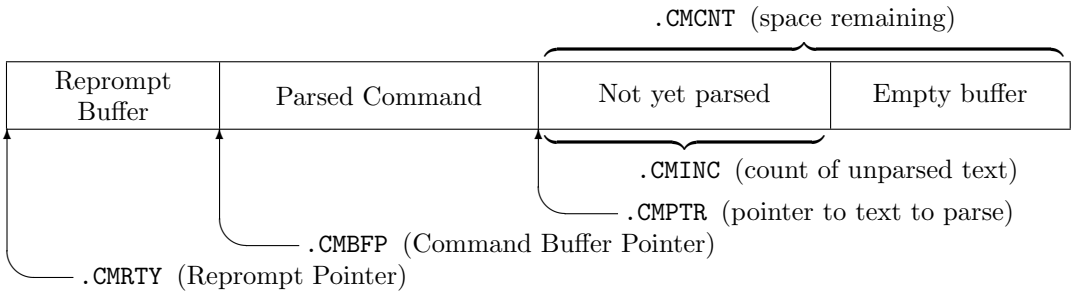


Figure 26.2: COMND Buffer Arrangement

containing the prompt need not be contiguous with the buffer containing the remainder of the command line. Typically this is a pointer to a string in the literal area.

`.CMBFP` This is a byte pointer to the beginning of the user's input text. This is the limit back to which the user can edit via `CTRL/W`, `RUBOUT`, and `CTRL/U`.

`.CMPTR` This is the byte pointer to the beginning of the next field to parse.

`.CMCNT` This is a count of the space remaining in the text input buffer following `.CMPTR`.

`.CMINC` This is the count of the number of characters in the buffer that have not yet been parsed.

Figure 26.2 depicts the logical arrangement of the byte pointers and counts. Conceptually, we think of the `CTRL/R` reprompt buffer as coming before the text buffer, but the reprompt buffer need not be adjacent to the rest.

These byte pointers, counts, and other information are contained in a *command state block*, whose address is one of the arguments to the `COMND` JSYS. The format of the command state block is shown in Figure 26.3.

The command state block includes the following items:

- The *reparse address* is the address within the program to which `COMND` will "return" when the user deletes over a field that has already been parsed. In such a case, state information that has been stored by the program is invalid. The state of the command scanner must be reinitialized and the command should be rescanned to parse the new value of any changed field.

  Extended addressing note: `COMND` was not updated for extended addressing. The address section in which to find the reparse address is implied from the address section in which `COMND` is executed. The command state block and other arguments must be in same address section as `COMND` itself. In general, this restriction is not burdensome.
- The flags in the left half of `.CMFLG` return information about the state of the parse.
- The word `.CMIOJ` contains a pair of JFNs, one for input and one for output.
- The most usual case is that `.PRIIN` and `.PRIOU` are used for input and output, respectively. It is possible to specify that input come from a disk file or other source of input.
- The text buffer was described above; it is where the user's text is accumulated before `COMND` parses it. The programmer tells `COMND` the location and size of the text buffer by supplying

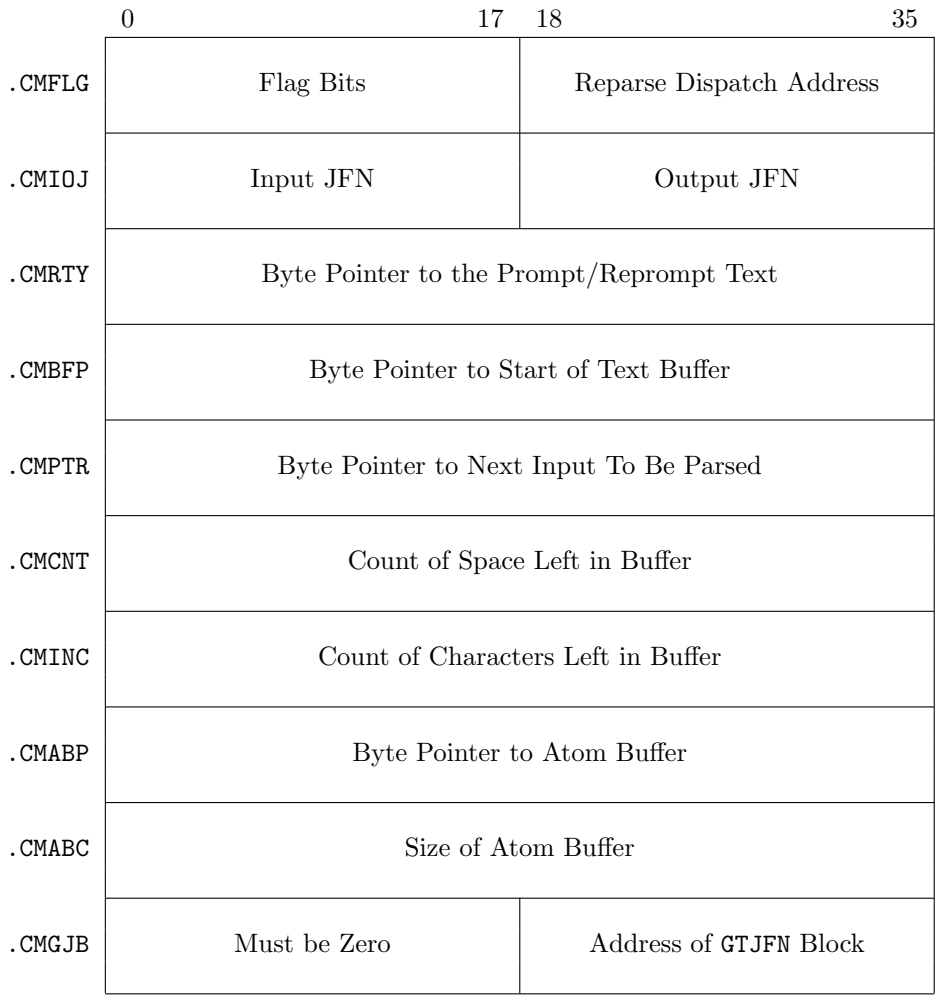


Figure 26.3: Command State Block

a byte pointer and a character count. (The words .CMPTR and .CMINC are for TOPS–20 to record intermediate states. The programmer does not need to initialize them; the programmer should avoid changing them while parsing a command.)

- The *atom buffer* is a temporary area used by COMND for collecting the contents of the current field. The atom buffer is identified by a byte pointer and a character count.
- The right half of the .CMGJB word supplies the in–section address of the argument block for a long–form GTJFN call. In Section 26.4.1 we will describe how this is used.

To accommodate these storage areas, our program will contain the following storage declarations:

```
CMDBUF: BLOCK   CMDBSZ                  ;Command buffer
ATMBUF: BLOCK   ATMBSZ                  ;Atom buffer
GTJBLK: BLOCK   .GJATR+1                ;GTJFN block
```

We will initialize the command state block as follows:

```
CMDBLK: 0,,CMRPRS                       ;Flags,,reparse routine addr
        .PRIIN,,.PRIOU                  ;JFNs for command I/O
        -1,,[ASCIZ/Small Executive>/]   ;CTRL/R bufr (prompt strng)
        -1,,CMDBUF                      ;Pntr: start of text buffer
        -1,,CMDBUF                      ;start of next input
        CMDBSZ*5-1                      ;command bufr byte count
        0                               ;Number of unparsed characters
        -1,,ATMBUF                      ;Pntr: start of atom buffer
        ATMBSZ*5-1                      ;Size of atom buffer
        0,,GTJBLK                       ;Pointer to GTJFN block
```

At every call to COMND, we will put the address of this command state block in register 1. When we call COMND, register 2 must contain the address of the first (or only) *command function descriptor block*. We use the macro FLDDB. to build command function blocks, as we shall explain below.

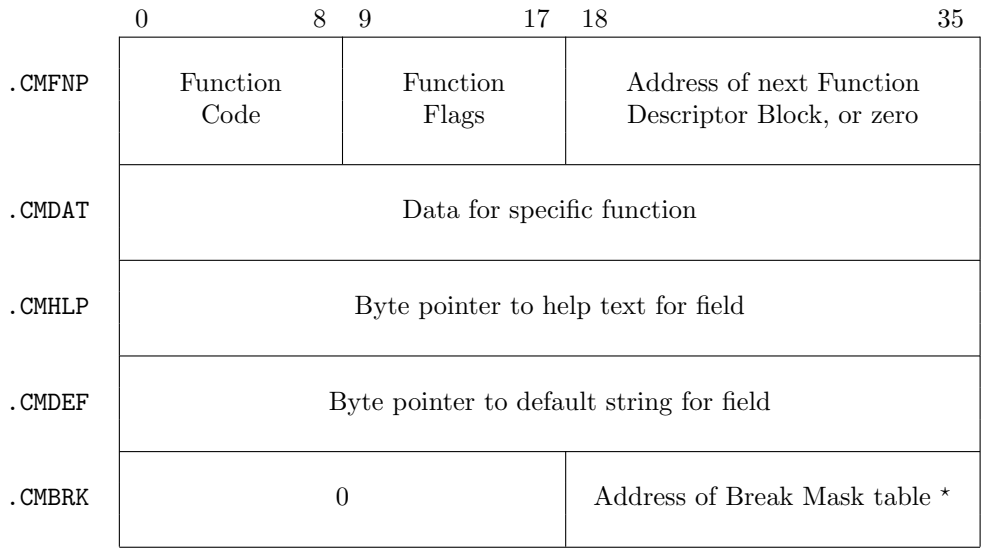
The first command function that we execute is a necessary initialization step. The .CMINI function of COMND is always the first function to call when starting to parse a command line. The .CMINI function initializes internal pointers and counters. The reprompt pointer supplies the initial prompt for a command; this function prints the prompt.

```
GETCMD: MOVEI   A,CMDBLK                ;A points to the state block
        MOVEI   B,[FLDDB. .CMINI]       ;Initialize state block, watch
        COMND                           ; for CTRL/H, do output prompt
         ERCAL  FATAL                   ;This should never happen.
        MOVEM   P,SAVPDL                ;Save stack pntr for reparse
CMRPRS: MOVE    P,SAVPDL                ;Restore stack if reparse
```

The execution of the .CMINI function causes the initial prompt to be sent to the terminal:

```
        Small Executive>
```

The program executing COMND should use the same command state block throughout the successful parse of one command. The program should not reset the byte pointers in the command state block.



* The .CMBRK word is present only when the CM%BRK flag is set.

Figure 26.4: Command Function Descriptor Block

We set up the state block once at the beginning of the program; then we use the .CMINI function when we begin to parse each command line. We avoid resetting the contents of the command state block because the .CMINI function implements the CTRL/H error recovery feature: when a CTRL/H is typed at the beginning of a line, the .CMINI function retrieves all the correctly parsed text of the immediately previous unconfirmed command. This text is made available to the user for further editing, so the entire command need not be retyped. The recovery of text from the last command is possible only when the program has left the command state block undisturbed following a command that was not confirmed (i.e., not successfully parsed to the end of the line).

26.3.2 Command Function Descriptor Block

A command line is parsed by repeated calls to COMND. Each call specifies the type of field that it expects by supplying an appropriate function code and any data needed for the function. This information is given in a *function descriptor block*, shown in Figure 26.4. On successful completion of each call, the current byte pointers and the counts are updated in the command state block, and any data obtained for the field is returned.

To assist us in building a function descriptor block, the MACSYM universal file includes the FLDDB. macro. The function descriptor begins with a 9–bit field that specifies which function to perform. The first instance of a function that we have seen is .CMINI, the initialization command. The FLDDB. macro takes as its arguments the command function code (called the command type), flags, the value of the data word, extra text for the help message, a default string that is used when the user requests recognition and has not supplied any string of his own, and the address of another function descriptor block.[3] The definition of FLDDB. appears below; an explanation follows.

[3] FLDDB. does not support the use of the break mask; use the FLDBK. macro if that feature is needed.

```
DEFINE FLDDB. (TYP,FLGS,DATA,HLPM,DEFM,LST)<
        ..XX==<FLD(TYP,CM%FNC)>+FLGS+<0,,LST>
   IFNB <HLPM>,<..XX=CM%HPP!..XX>
   IFNB <DEFM>,<..XX=CM%DPP!..XX>
        ..XX
   IFNB <DATA>,<DATA>
   IFB <DATA>,<0>
   IFNB <HLPM>,<POINT 7,[ASCIZ \HLPM\]>
   IFB <HLPM>,<IFNB <DEFM>,<0>>
   IFNB <DEFM>,<POINT 7,[ASCIZ \DEFM\]>
>;Define FLDDB.
```

The assembler conditionals `IFB` and `IFNB`, *IF Blank* and *IF Not Blank*, respectively, are used to establish reasonable defaults in case an argument is omitted. If either a help message or a default string is required, an appropriate bit is included in the flags. When fields are omitted, zero words are assembled in the command function block.

(There are two more conditionals that test textual arguments: `IFIDN` and `IFDIF`, meaning *IF IDeNtical* and *IF DIFferent*, respectively. Each is followed by two parameters and then by conditional text. If the two parameters are textually identical, `IFIDN` is satisfied. If the two paramerers are not identical, `IFDIF` is satisfied.)

The ability to link command functions together is important. When a field can be either a keyword or a number it is necessary to supply a list of function blocks. The first might call for a keyword, and the second ask to parse a number. When a number is input, `COMND` advances from the inappropriate keyword scan to the number scan in an attempt to find a correct parse. In Section 26.3.4 we shall give a specific illustration of linking command functions.

26.3.3 Parsing Keywords

We began this example with the idea that we had to perform some function in response to a command called `COUNT`. This implies that the program must first distinguish the keyword `COUNT` from among several other possible commands. The entire catalog of possible commands is held in a data structure called a *command table*. There are two things that any command table must include: the command name, and the address of a piece of program to execute in response to that command. The command table for the `COMND` JSYS can also be made to include command flags, but for the moment we need no flags. It is convenient to use a macro to help us build the proper data structure for the command table. The `CMD` macro (one that we write ourselves) generates an entry for a command table suitable for the `TBLUK` JSYS, which is called by `COMND` to perform this table lookup. The `CMD` macro assumes that by convention the subroutine that services a command has a name composed of a period followed by the same name as the command. This convention is used in the coding of TOPS–20 and in the EXEC.[4]

```
DEFINE CMD (COMMAND) <[ASCIZ/COMMAND/],,.'COMMAND>
```

In macro definitions, the character apostrophe (') is the *concatenation character*. In this macro we

[4]When command names are longer than 5 letters, labels longer than 6 letters are implied. If the newer MACRO that knows long names is used, you should be careful to provide long labels where the service routine is defined.

want to form the name of a label composed of a period and the macro argument. Simply writing `.COMMAND` does not work because that is recognized as an identifier name that is different from the name of the dummy argument, `COMMAND`. To solve this problem we use the apostrophe to concatenate, i.e., run together, the strings on either side of the apostrophe. Thus, when we write `.'COMMAND` in the macro definition, MACRO recognizes `COMMAND` as the name of the argument. The apostrophe disappears, its only purpose having been to alert MACRO to stop one identifier scan and to start another.

MACRO stores the definition of the `CMD` macro as text string that looks somewhat like this:

```
[ASCIZ/⊂1⊃/],,.⊂1⊃
```

The notation ⊂1⊃ is our way of representing that MACRO knows to copy the first actual argument into that point of the macro expansion. The net effect of the concatenation character is such that when the argument to `CMD` is `COUNT` this macro will expand as

```
[ASCIZ/COUNT/],,.COUNT
```

Now, with the help of the `CMD` macro, we can write our command table. We'll begin quite simply with just three commands: `COUNT`, `EXIT`, and `TYPE`. These will be served by program fragments at the labels `.COUNT`, `.EXIT`, and `.TYPE`:

```
CMDTAB: CMDTBL,,CMDTBL          ; actual number,,max number of entries
        CMD COUNT
        CMD EXIT
        CMD TYPE
CMDTBL==<.-CMDTAB>-1            ; number of entries in the table
```

Command names **must** appear in alphabetical order in the command table: many programs have malfunctioned because the programmer did not follow this rule. The left half of the first word in the command table contains the count of the actual number of entries present in the table; the right half is the maximum amount of room available. These counts do not include this header word. In our simple case, we will let both numbers be the same. In more complex circumstances, we could allow extra room in the table and use the `TBADD` and `TBDEL` JSYS calls to add or delete table entries.

Now, we shall call `COMND` to parse the command keyword:

```
CMRPRS: MOVE    P,SAVPDL                ;Restore stack if reparse
        MOVEI   B,[FLDDB. .CMKEY,,CMDTAB,<A command,>]
        COMND
         ERCAL  FATAL
        TXNE    A,CM%NOP                ;Did the parse fail?
         JRST   ERROR                   ;Yes - report error & return
        HRRZ    B,(B)                   ;Get addr of command server
        JRST    (B)                     ;Dispatch to it
```

In this call to `COMND` we have specified the keyword lookup function, `.CMKEY`. The data item for this function is `CMDTAB`, the address of the command table header word. We have also specified the string "A command," as an extra help message; because this help string includes a comma, we must wrap that macro argument inside pointed brackets.

When the user of this program types a question mark, COMND will print the following response:

```
Small Executive>? A command, one of the following:
 COUNT    EXIT    TYPE
Small Executive>
```

In general, each call to COMND causes it to gather the next field into the atom buffer. The .CMKEY function then performs a table lookup on that atom, using the command table address specified in the command function descriptor block. If the command name in the atom buffer cannot be matched in the table, the COMND JSYS returns with the *no parse* flag, CM%NOP, set in the left half of register 1. This program tests for that error flag; if present the program will jump to ERROR where an appropriate message will be typed. ERROR returns via RET to the caller of GETCMD, so the user can be given an opportunity to correct the error. We use the jump to ERROR consistently as a means to alert the user of his errors and to afford him an opportunity to retype the command correctly. This is in contrast to our use of the error routine FATAL which signifies that a programming error, rather than a user command error, has occurred.

Assuming that all goes well, register 2 will be set to the address of the command table entry that was matched. Since the right half of the table entry contains the command service routine address, we pick that up and dispatch to it. Assuming the COUNT command is recognized, we will dispatch to the label .COUNT.

26.3.4 Parsing with Defaults and Alternatives

Having arrived at the command server called .COUNT, we must now determine whether the user wants to count up or count down. In the examples we started with, we saw that either UP or DOWN would be permissible as the next field. Also, if neither UP nor DOWN is seen, we wish to default to counting up, and accept a number as an alternative to either of the keywords UP or DOWN.

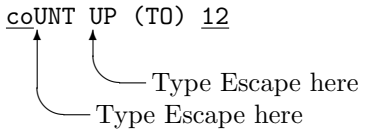
We must build a keyword table that contains the words UP and DOWN. Note that as with all command tables, these must be written in alphabetical order. This table is similar to the main command dispatch table that we saw earlier.

```
UDCTAB: UDCTLN,,UDCTLN
        CMD (DOWN)
        CMD (UP)
UDCTLN==<.-UDCTAB>-1
```

We can effect the recognition of a *default field* by building an appropriate command function descriptor block. The following is quite suitable:

```
FLDDB. .CMKEY,,UDCTAB,<a direction to count,>,UP
```

This command function descriptor block points to the new keyword table that we have made. If the user types either escape or return, the string UP will be taken as the default, and it will be parsed. Thus we can parse the command:



The problem of parsing an alternative field adds more complexity. If we want to be able to recognize the command `COUNT 10`, we must tell `COMND` that we want to parse either a keyword or a number. We do so by providing a list of command function descriptor blocks instead of just one block. The last argument to the `FLDDB.` macro is the address of an alternative function descriptor block. So, we will build a second function descriptor block, this one to parse a number, and then link these two.

We build the following function descriptor block for parsing a number. This block is also supplied with a help text.[5] To allow us to refer to this descriptor from the keyword command block, we give this block a label, `UPNUM`.

```
UPNUM:  FLDDB.  .CMNUM,CM%SDH,^D10,<a number to count up to>
```

The function code `.CMNUM` is used for parsing a number. The data supplied is the decimal number 10, signifying the desired input radix. The flag `CM%SDH` means *suppress default help message*. The default message for the `.CMNUM` function is something like "a number". The default message is suppressed and only our message is present as a help string. We put these two blocks together:

```
.COUNT: SETZM   UDFLAG          ;assume count up
        MOVEI   B,[FLDDB. .CMKEY,,UDCTAB,<a direction to count,>,UP,UPNUM]
        COMND
         ERCAL  FATAL
        TXNE    A,CM%NOP
         JRST   ERROR
```

We start by setting the up/down flag, `UDFLAG` to zero, signifying an upwards count. Register `B` is loaded with the address of function descriptor block that parses a keyword. Note that this block points to `UPNUM` as an alternative descriptor block. It is interesting to type a question mark to obtain help after getting past the `COUNT` keyword:

```
Small Executive>coUNT ? a direction to count, one of the following:
 DOWN    UP
or a number to count up to
Small Executive>coUNT
```

The `COMND` JSYS is now willing to accept either a keyword or a number. There is one more issue: how do we know whether `COMND` saw a number or a keyword? When `COMND` returns the right half of register `3` is set to the address of the particular command function descriptor block that was used to parse this field.

It should be noted that the various alternatives that are set up for `COMND` are attempted in the order they appear in the list. In cases where the parse might produce ambiguous results, the user should be careful to select the most useful order of these descriptor blocks. When a default value is supplied, it is applied only when no descriptor block has yielded a successful parse.

[5] There is no point in supplying a default field here; a default can't be used because this block is linked from the keyword block which already supplies a default.

```
.COUNT: SETZM   UDFLAG          ;assume count up
        MOVEI   B,[FLDDB. .CMKEY,,UDCTAB,<a direction to count,>,UP,UPNUM]
        COMND
         ERCAL  FATAL
        TXNE    A,CM%NOP
         JRST   ERROR
        HRRZ    C,C             ;get the addr of the funct descriptor used.
        CAIN    C,UPNUM         ;Did we parse a number?
         JRST   CNUM2           ;Yes.  The numeric result is in B.
        HRRZ    B,(B)           ;No, it was a keyword
        JRST    (B)             ;dispatch to the handler.
```

26.3.5 Noise Words

When the user types the escape character, the COMND JSYS will complete a partially typed field, if unambiguous, or apply a default where appropriate. In addition to typing the completion of a partially specified field name, the COMND JSYS will also type any *noise words* (also called "guide words") that the programmer has specified. Noise words are present as a guide to a user who is unfamiliar with the meaning of each field.

We define another macro called NOISE:

```
DEFINE NOISE (STRING) <
        MOVEI   B,[FLDDB. .CMNOI,,<-1,,[ASCIZ/STRING/]>]
        COMND
         ERCAL  FATAL
        TXNE    A,CM%NOP
         JRST   ERROR
>;Define NOISE
```

We can call this macro after any call in which COMND successfully parses a field. If recognition was used on that field, then the COMND JSYS will also type the noise words specified here. COMND will place the noise phrase in parentheses when that phrase is typed. Also, if the user so desires, the noise phrase can be typed as part of the input. If that is done, the noise phrase must be typed with parentheses.

Here are the servers for the UP and DOWN options of the COUNT command. Each server will supply or recognize an appropriate noise word and prepare for reading a decimal number as an argument.

```
.UP:    NOISE   (TO)
        MOVEI   B,[FLDDB. .CMNUM,CM%SDH,^D10,a number to count up to,20]
        JRST    CNUM

.DOWN:  NOISE   (FROM)
        SETOM   UDFLAG
        MOVEI   B,[FLDDB. .CMNUM,CM%SDH,^D10,a number to count down from,20]
CNUM:   COMND
         ERCAL  FATAL
        TXNE    A,CM%NOP
         JRST   ERROR
```

We supply the string 20 as the default for these numeric command blocks. Assuming the parse is successful, the argument will be returned in register 2 (our B). If it is negative or zero, we will jump to NONEG. When things go well, we will copy the number to register D and expand the CONFIRM macro to verify that the command line ends properly.

```
CNUM2:  JUMPLE  B,NONEG         ;a negative argument makes no sense.
        MOVE    D,B             ;save number over CONFIRM
        CONFIRM                 ;tie off the command.
```

The CONFIRM macro looks like this:

```
DEFINE CONFIRM <
        MOVEI   B,[FLDDB. .CMCFM]
        COMND
         ERCAL  FATAL
        TXNE    A,CM%NOP
         JRST   ERROR
>;Define CONFIRM
```

The .CMCFM command confirmation function is very important; every command must end its scanning process by invoking this function. .CMCFM waits until the user types a carriage return. If there is any text left over at the end of the command line parse, the confirmation function will catch the problem. If a line is successfully confirmed, the CTRL/H retry feature is disabled; however, if the confirm function is not successful or if it is not performed, then CTRL/H will retrieve all of the line up to the last error. This use of CTRL/H is implemented in the .CMINI function.

The command COUNT with no arguments works by taking two defaults. First, the default UP is taken because there is no text present. Then, on arrival at .UP we take the default 20 for the number.

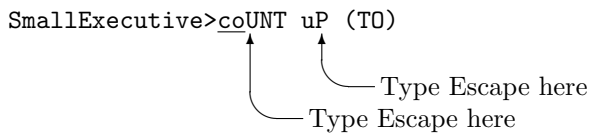
The remainder of the code for the COUNT command is only slightly devious:

```
        SKIPGE  UDFLAG          ;Counting up or down?  Skip if UP
         JRST   COUNT0          ;Counting down.  UDFLAG is -1, the max value
        MOVEM   D,UDFLAG        ;UDFLAG is now the upper bound
        SKIPA   D,[1]           ;And for up counting, D is initially 1
COUNT0:  MOVN   D,D             ;For down count, D is initially -n
                                ;Here D is the lower bound, UDFLAG is the
                                ;upper bound.  For counting down, we'll
                                ;actually count a negative number up to -1
        MOVEI   A,.PRIOU        ;set up to output to the terminal
COUNT1: MOVM    B,D             ;get magnitude of number to output
        MOVEI   C,^D10          ;decimal radix
        NOUT                    ;output it
         ERCAL  FATAL
        HRROI   B,CRLF          ;output delimiting CRLF
        SETZ    C,              ;terminate on null
        SOUT
         ERCAL  FATAL
        CAMGE   D,UDFLAG        ;(if counting down, UDFLAG is -1)
         AOJA    D,COUNT1       ;(if counting up, it is the upper bound)
        RET

NONEG:  HRROI   A,[ASCIZ/?I can't count to a negative limit.
/]                              ; defensive code against user giving a
        PSOUT
        RET                     ; zero or negative number
```

26.3.6 Reparsing a Command

The reparse address CMRPRS is the place to which COMND will return if the user deletes back into a field that has already been parsed. For example if a user types



The program has already parsed the keywords COUNT and UP because recognition was used. If the user now decides to backspace over the noise word and to eradicate the keyword UP, the COMND JSYS will detect that action and send the program back to the reparse address so the new command text can be scanned.

Due to the possibility that a reparse may occur, it is crucial that the parsing of a command should avoid any irrevocable alteration in the state of the data structures until the command is confirmed. For example, if the parsing of a command includes obtaining a JFN, if the parse is unsuccessful, the JFN must be released. (Otherwise, the program could collect all the JFNs allowed to your job.)

26.4 Example 17 — Small Executive Program

Now that we have accomplished our lengthy but essential introduction to the COMND JSYS, it seems that we are nearly ready to present the entire Small Executive program, example 17. The Small Executive is an extremely abbreviated model of the TOPS–20 EXEC program.

Although it performs only a few functions, it is still quite complex. Much of the structure is present to make it easy to add new commands. Even though our introduction to COMND has already been extensive, there are areas that we have not yet discussed. A further discussion follows the text of the program.

```
        TITLE   SMALL EXECUTIVE          ;Mark R. Crispin, 12/79
        SEARCH  MACSYM,MONSYM,QSRMAC     ;Get system macro and symbol definitions
        SALL                             ;Suppress macro expansions

;Accumulator definitions
A=1                                      ;JSYS arguments & temporary AC's
B=2
C=3
D=4
P=17                                     ;stack pointer

OPDEF   CALL    [PUSHJ P,]               ;(these are also defined in MACSYM)
OPDEF   RET     [POPJ P,]

;Standard version information

VWHO==2                                  ;who last edited program
VMAJOR==2                                ;major version
VMINOR==7                                ;minor version
VEDIT==14                                ;edit version


;Assembly Switches
;Note that CMDBSZ and ATMBSZ take on default values that can
;be overridden by the inclusion of a header file.

PDLEN==100                               ;length of the pushdown stack
IFNDEF CMDBSZ,CMDBSZ==^D50               ;length of the command text buffer
                                         ;   (250 characters)
IFNDEF ATMBSZ,ATMBSZ==^D20               ;length of the atom buffer (100 chars)
```

```
        SUBTTL  Useful Macro Definitions

;Parse a string of noise words
DEFINE NOISE (STRING) <
        MOVEI   B,[FLDDB. .CMNOI,,<-1,,[ASCIZ/STRING/]>]
        COMND
         ERCAL  FATAL
        TXNE    A,CM%NOP
         JRST   ERROR
>;Define NOISE

;Obtain confirmation, an end of line indication. Tie off command line.
DEFINE CONFIRM <
        MOVEI   B,[FLDDB. .CMCFM]
        COMND
         ERCAL  FATAL
        TXNE    A,CM%NOP
         JRST   ERROR
>;Define CONFIRM

;Call this macro to help build the command table.  This macro is more complex
;(and more useful) than the CMD macro described earlier.
;This macro is explained in Section 26.4.2, page 493.

DEFINE TBL (NAME,FLAGS,DISP) <
IFNB <DISP>,<..DISP==DISP>              ;;If a dispatch is given, use it
IFB  <DISP>,<..DISP==.'NAME>            ;;If none, default to .NAME
IFB  <FLAGS>,<[ASCIZ/NAME/],,..DISP>    ;;if no flags, assemble just the name
IFNB <FLAGS>,<[FLAGS!CM%FW              ;;if flags, use them and set CM%FW
                ASCIZ/NAME/],,..DISP>   ;;
        PURGE   ..DISP
>;Define TBL

        SUBTTL  Data Storage Areas
.PSECT  DATA,1001000

PDLIST: BLOCK   PDLEN                   ;pushdown list
SAVPDL: 0                               ;Save Pushdown Pointer
                                        ;in case of reparse

CORBEG==.                               ;This storage zeroed at START

                                        ;Storage used by COMND
CMDBUF: BLOCK   CMDBSZ                  ;command buffer
ATMBUF: BLOCK   ATMBSZ                  ;atom buffer
GTJBLK: BLOCK   .GJATR+1                ;GTJFN block

                                        ;Other storage
UDFLAG: 0                               ;up count/down count flag for COUNT
INPJFN: 0                               ;input JFN for TYPE
```

```
;Storage used by the PUSH command.  See Section 27, page 503.
EXCJFN: 0                               ;JFN for PUSH
FKHAN:  0                               ;fork handle for PUSH

;Storage used by the QUEUE command. The server for this command is
;explained in Section 28.1, page 515.
IPCBLK: BLOCK   .IPCFP+1                ;storage for IPCF JSYS calls
MYPID:  0                               ;PID for this program
QSRPID: 0                               ;PID for Quasar
FIRSTP: 0                               ;flag used in GETQRP

COREND==.-1                             ;end of area zeroed at START

;Command State Block

CMDBLK: 0,,CMRPRS                       ;Flags,,address of reparse routine
        .PRIIN,,.PRIOU                  ;JFNs for command I/O
        -1,,[ASCIZ/Small Executive>/]   ;CTRL/R buffer (i.e., prompt string)
        -1,,CMDBUF                      ;pointer to start of text buffer
        -1,,CMDBUF                      ;pointer to start of next input
        CMDBSZ*5-1                      ;size of command buffer in bytes
        0                               ;number of unparsed characters
        -1,,ATMBUF                      ;pointer to start of atom buffer
        ATMBSZ*5-1                      ;size of atom buffer
        0,,GTJBLK                       ;pointer to GTJFN block

.ENDPS                          ;end of the DATA psect.

        SUBTTL  Top Level, First Command Dispatch & Command Table
.PSECT  CODE/RONLY,1002000

START:  RESET                           ;initialize all I/O
        MOVE    P,[IOWD PDLEN,PDLIST]   ;initialize stack pointer
        SETZM   CORBEG                  ;initialize data area
        MOVE    A,[CORBEG,,CORBEG+1]
        BLT     A,COREND

TOPLEV: CALL    GETCMD                  ;get a command and run it
        SETO    A,                      ;here on return from command
        CLOSF                           ;clean up any stray JFN's left behind
         ERCAL  FATAL                   ;this shouldn't happen
        JRST    TOPLEV                  ;loop back to top level
```

```
GETCMD: MOVEI   A,CMDBLK                ;Register A points to the state block
        MOVEI   B,[FLDDB. .CMINI]       ;Initialize the state block, watch for
        COMND                           ; CTRL/H, do output prompt.
         ERCAL  FATAL                   ;This should never happen.
        MOVEM   P,SAVPDL                ;Save stack pointer for reparse
CMRPRS: MOVE    P,SAVPDL                ;Restore stack pointer for reparse
        MOVEI   B,[FLDDB. .CMKEY,,CMDTAB,<A command,>]
        COMND
         ERCAL  FATAL
        TXNE    A,CM%NOP                ;Did the parse fail?
         JRST   ERROR                   ;Yes - report the error and return
        HRRZ    B,(B)                   ;Get the address of command server
        JRST    (B)                     ;Dispatch to it

;This table format is discussed further in Section 26.4.2, page 493.

CMDTAB: CMDTBL,,CMDTBL                  ;actual,,maximum number of entries
        TBL (COUNT)
        TBL (EXIT)
        TBL (HELL,CM%NOR,0)             ;"HELL" is an illegal abbreviation
        TBL (HELLO)                     ;for "HELLO"
        TBL (HELP)
        TBL (PUSH)
        TBL (Q,CM%INV!CM%ABR,$QUEUE)    ;Q and QU are invisible
        TBL (QU,CM%INV!CM%ABR,$QUEUE)   ;abbreviations for QUEUE
$QUEUE: TBL (QUEUE)
        TBL (QUIT,CM%INV,.EXIT)         ;QUIT is an invisible alias for EXIT
        TBL (TYPE)
CMDTBL==<.-CMDTAB>-1                    ;number of entries in the table

        SUBTTL  Command Servers

;Server for the COUNT command.
.COUNT: SETZM   UDFLAG          ;assume count up
        MOVEI   B,[FLDDB. .CMKEY,,UDCTAB,<a direction to count,>,UP,UPNUM]
        COMND
         ERCAL  FATAL
        TXNE    A,CM%NOP
         JRST   ERROR
        HRRZ    C,C             ;get the addr of the funct descriptor used.
        CAIN    C,UPNUM         ;Did we parse a number?
         JRST   CNUM2           ;Yes.  The numeric result is in B.
        HRRZ    B,(B)           ;No, it was a keyword
        JRST    (B)             ;dispatch to the handler.

UPNUM:  FLDDB.  .CMNUM,CM%SDH,^D10,<a number to count up to>

UDCTAB: UDCTLN,,UDCTLN          ;secondary keyword table for the COUNT command
        TBL (DOWN)
        TBL (UP)
UDCTLN==<.-UDCTAB>-1
```

```
.UP:    NOISE   (to)
        MOVEI   B,[FLDDB. .CMNUM,CM%SDH,^D10,a number to count up to,20]
        JRST    CNUM

.DOWN:  NOISE   (from)
        SETOM   UDFLAG
        MOVEI   B,[FLDDB. .CMNUM,CM%SDH,^D10,a number to count down from,20]
CNUM:   COMND
         ERCAL  FATAL
        TXNE    A,CM%NOP
         JRST   ERROR
CNUM2:  JUMPLE  B,NONEG         ;a negative argument makes no sense.
        MOVE    D,B             ;save number over CONFIRM
        CONFIRM                 ;tie off the command.
                                ;Here to actually do the counting
        SKIPGE  UDFLAG          ;Counting up or down?  Skip if UP
         JRST   COUNT0          ;Counting down.  UDFLAG is -1, the max value
        MOVEM   D,UDFLAG        ;UDFLAG is now the upper bound
        SKIPA   D,[1]           ;And for up counting, D is initially 1
COUNT0:  MOVN   D,D             ;For down count, D is initially -n
                                ;Here D is the lower bound, UDFLAG is the
                                ;upper bound.  For counting down, we'll
                                ;actually count a negative number up to -1
        MOVEI   A,.PRIOU        ;set up to output to the terminal
COUNT1: MOVM    B,D             ;get magnitude of number to output
        MOVEI   C,^D10          ;decimal radix
        NOUT                    ;output it
         ERCAL  FATAL
        HRROI   B,CRLF          ;output delimiting CRLF
        SETZ    C,              ;terminate on null
        SOUT
         ERCAL  FATAL
        CAMGE   D,UDFLAG        ;(if counting down, UDFLAG is -1)
         AOJA   D,COUNT1        ;(if counting up, UDFLAG is high bound)
        RET

NONEG:  HRROI   A,[ASCIZ/?I can't count to a negative limit.
/]                              ;defensive code against user giving
        PSOUT                   ;zero or a negative number
        RET

;Server for the EXIT command, to have a graceful way of getting out!
.EXIT:  NOISE   (from Small Executive)
        CONFIRM
        HALTF                   ; return to EXEC
        RET                     ; continue puts us back in the parser
```

```
;Server for the HELLO command.
;format for printing a version number is MAJOR.MINOR(EDIT)-WHO
.HELLO: CONFIRM                 ;tie off the command
        HRROI   A,[ASCIZ/Hello, this is the Small Executive.
Version /]
        PSOUT
        MOVEI   A,.PRIOU                ;numeric output to terminal
        MOVEI   C,10                    ;in radix eight
        MOVE    B,VERSIO                ;get the version number
TXNE    B,VI%DEC                ;skip unless reporting in decimal
         MOVEI  C,12                    ;report in radix ten
        LDB     B,[POINTR(VERSIO,VI%MAJ)] ;major version
        NOUT
         ERJMP  .+1

        MOVEI   B,"."
        BOUT
        LDB     B,[POINTR(VERSIO,VI%MIN)] ;minor version
        NOUT
         ERJMP  .+1
        MOVEI   B,"("
        BOUT
        LDB     B,[POINTR(VERSIO,VI%EDN)] ;edit number, in parens
        NOUT
         ERJMP  .+1
        HRROI   A,[ASCIZ/)-/]
        PSOUT
        MOVEI   A,.PRIOU
        LDB     B,[POINTR(VERSIO,VI%WHO)] ;finally, who last edited.
        NOUT
         ERJMP  .+1
        HRROI   A,CRLF
        PSOUT
        RET

;Server for the HELP command
.HELP:  NOISE   (in using the Small Executive)
        CONFIRM
        HRROI   A,HLPMSG        ;output the built-in help message
        PSOUT
        RET

HLPMSG: ASCIZ/
The Small Executive is a simple command processor that demonstrates the
capabilities of the COMND JSYS.  Try typing ? to see what commands are
available.
/

;The actual server for the PUSH command will be presented in Section 27, page 503.
.PUSH:  NOISE   (command level)
        CONFIRM
        RET
```

```
;The real server for the QUEUE command will be presented in Section 28.1, page 515.
.QUEUE: NOISE   (status display)
        CONFIRM
        RET

.TYPE:  NOISE   (file on the terminal)
        SKIPE   A,INPJFN        ;Is there any JFN lying around?
         CLOSF                  ;Yes, well try to close it.
          ERJMP .+1             ;Ignore any failure
        SETZM   INPJFN          ;And don't try to close it again.
        MOVEI   A,CMDBLK        ;Reload A with pointer to state block
        MOVEI   B,[FLDDB. .CMIFI,CM%SDH,,name of the file you want to type]
        COMND                   ;Get an input file
         ERCAL  FATAL
        TXNE    A,CM%NOP        ;Maybe file not found or something?
         JRST   ERROR
        HRRZM   B,INPJFN        ;Save the JFN we got
        CONFIRM                 ;Tie off the command
        MOVE    A,INPJFN        ;Open the file, using the JFN from COMND
        MOVX    B,<FLD(7,OF%BSZ)+OF%RD> ; read access, 7 bit bytes
        OPENF
         ERJMP  ERROR           ;if OPENF fails for any reason
TYPE1:  MOVE    A,INPJFN
        BIN                     ;Simple byte-by-byte copy loop.
         ERJMP  TYPE2           ;Jump if Error, may be end of file.
        MOVEI   A,.PRIOU        ;This is written for brevity, not speed.
        BOUT
        JRST    TYPE1           ;Loop back for next character

TYPE2:  MOVE    A,INPJFN        ;Error here.  Is it end of file?
        GTSTS
        TXNN    B,GS%EOF
         CALL   ERROR           ;Some serious problem; report it.
        MOVE    A,INPJFN
        CLOSF                   ;Close file.
         ERCAL  .+1
        SETZM   INPJFN          ;Zero JFN storage.
        RET
```

```
        SUBTTL  Error Handlers & Miscelleny

; Fatal error routine, for "impossible errors" only
; Called by ERCAL FATAL after the failing JSYS

FATAL:  CALL    ERROR           ; first output the reason the JSYS died
        HRROI   A,[ASCIZ/, JSYS at PC=/]
        PSOUT
        MOVEI   A,.PRIOU        ; now output the PC
        POP     P,B             ; get the PC back from the stack
        SUBI    B,2             ; back up over the ERCAL to the JSYS address
        MOVX    C,NO%MAG!10     ; output free format unsigned octal
        NOUT
         ERJMP  .+1             ; this can't happen, but avoid recursion
        HRROI   A,CRLF
        PSOUT
        MOVEI   A,.PRIIN        ; flush the TTY input buffer
        CFIBF
FATAL0: HALTF                   ; return to EXEC since we're crashing
        HRROI   A,[ASCIZ/?Can't continue
/]
        PSOUT
        JRST    FATAL0          ; disallow CONTINUE command

; Ordinary JSYS error routine.  Just outputs the error string for the
; failing JSYS and returns.

ERROR:  HRROI   A,[ASCIZ/Error: /]
        ESOUT
        MOVEI   A,.PRIOU        ; error message to primary output
        HRLOI   B,.FHSLF        ; this fork,,last error
        SETZ    C,              ; no limit
        ERSTR
         ERJMP  .+1             ;Neither of these is supposed to happen
         ERJMP  .+1
        HRROI   A,CRLF
        PSOUT
        RET

CRLF:   BYTE(7)15,12

;Entry vector
EVEC:   JRST    START       ;START entry point
        JRST    START       ;REENTER entry point
VERSIO: FLD(VWHO,VI%WHO)!FLD(VMAJOR,VI%MAJ)!FLD(VMINOR,VI%MIN)!FLD(VEDIT,VI%EDN)
                            ;version #. Label for Hello.
EVECL==.-EVEC               ;length of the entry vector

        END     <EVECL,,EVEC>
```

We will now continue by presenting further discussion of those portions of this program that pertain especially to the COMND JSYS. (We shall defer discussion of the QUEUE and PUSH commands to later chapters.)

26.4.1 Reading JFNs via COMND

The server for the TYPE command exemplifies the use of COMND to obtain a JFN as part of the command scanning process. Before we attempt to get a new JFN, we make certain that there is no old JFN left from any prior abortive command. If INPJFN is non–zero, we attempt to close that JFN. We ignore any failure from CLOSF and we zero INPJFN to prevent any further attempts to close it.

```
.TYPE:  NOISE   (file on the terminal)
        SKIPE   A,INPJFN        ;Is there any JFN lying around?
         CLOSF                  ;Yes, well try to close it.
          ERJMP .+1             ;Ignore any failure
        SETZM   INPJFN          ;And don't try it again.
```

The .CMIFI function parses an input file specification. In our call, we suppress the normal help facility and supply an appropriate help text of our own:

```
        MOVEI   A,CMDBLK        ;Reload A with pointer to state block
        MOVEI   B,[FLDDB. .CMIFI,CM%SDH,,name of the file you want to TYPE]
        COMND                   ;Get an input file
         ERCAL   FATAL
        TXNE    A,CM%NOP        ;Maybe file not found or something?
         JRST   ERROR
        HRRZM   B,INPJFN        ;Save the JFN we got
        CONFIRM                 ;Tie off the command
```

The .CMIFI function makes COMND execute a long form GTJFN call to parse the specification for an existing file using no default fields.[6] The .CMGJB address in the command state block must be supplied. Any data previously in the GTJFN block will be overwritten by the .CMIFI function, which will also set GTJFN flags appropriately. On a successful return, register 2 contains the JFN assigned.

There are at least two ways that we could arrive at .TYPE with a JFN still in INPJFN. First, if the call to CONFIRM failed to find the end of the command after INPJFN was stored, then the user might be inclined to visit this routine again. There is an easy way to fix that source of missing JFNs: a CLOSF JSYS in the top level loop can clean up any stray JFNs. After each command or attempted command we perform a CLOSF with register 1 set to −1; this CLOSF will close and release all of the JFNs that belong to this process.

Using this CLOSF is not entirely satisfactory because a reparse does not go through that path. After a file name has been identified and a JFN has been stored in INPJFN, if the user decides to delete back into the file name field, a reparse will be necessary. When the reparse is made, the program will come back to .TYPE. Thus, it is at .TYPE that we release any JFN that had been obtained by the previous parse.

The JFN is stored in INPJFN and the command is confirmed. The remainder of the processing should be obvious enough. The JFN is opened for reading, and a simple read/write loop is called. This loop was written for simplicity rather than for efficiency. A SIN or, better yet, a PMAP would be a far more suitable way to read any significant file.

[6] The details of the long form of GTJFN can be found in [MCRM].

```
        MOVE    A,INPJFN        ;Open file, using the JFN from COMND
        MOVX    B,<FLD(7,OF%BSZ)+OF%RD> ; read access, 7 bit bytes
        OPENF
         ERJMP  ERROR           ;If OPENF fails for any reason
TYPE1:  MOVE    A,INPJFN
        BIN                     ;Simple byte-by-byte copy loop.
         ERJMP  TYPE2           ;Jump if error, may be end of file.
        MOVEI   A,.PRIOU        ;This code is brief, not fast
        BOUT
        JRST    TYPE1           ;Loop back for next character

TYPE2:  MOVE    A,INPJFN        ;Here if error, is it end of file?
        GTSTS
        TXNN    B,GS%EOF        ;Skip if it is end of file.
         CALL   ERROR           ;Serious: tell user; abort command
        MOVE    A,INPJFN
        CLOSF                   ;Close file.
         ERCAL  .+1
        SETZM   INPJFN          ;Zero JFN storage.
        RET
```

26.4.2 Command Tables and TBLUK

When we first discussed what we need to have in a command table we alluded to command flags. Now that we have a more complex menu of commands, it is desirable to explain the use of these flags and the flexibility they afford.

The `TBLUK` JSYS is used by `COMND` to search a keyword table. The table must be alphabetized because `TBLUK` uses a *binary search* to find items. In spite of disadvantages when adding or deleting from the command table, this format for the table is appropriate because it lets us identify matching entries fairly quickly; it allows us to display the various possibilities in alphabetical order; and, when only an ambiguous prefix is known, all possible completions can be located readily and shown to the user.

A binary search initially defines the boundary of the search space as the two ends of the command table. It knows where the ends are because the first word of the command table specifies the effective length of the table. For each probe into the table, the binary search program bisects the search space. The midpoint of the search space is calculated and the data item at that bisection point is examined. If the element matches the sought–for input, the binary search is finished. If we have not yet matched the input, we must change the boundary of the search space. If the bisection element is smaller than the sought–for item, the lower bound of the search space is moved up to the bisection point; otherwise, the upper bound is moved down to the midpoint. The search continues, halving the size of the search space at each unsuccessful probe. If an item is present, it can be found within $\log_2(n)$ probes, where n is the number of items in the table. An examination of the immediate neighbors of a matching item will reveal if the item is ambiguous.

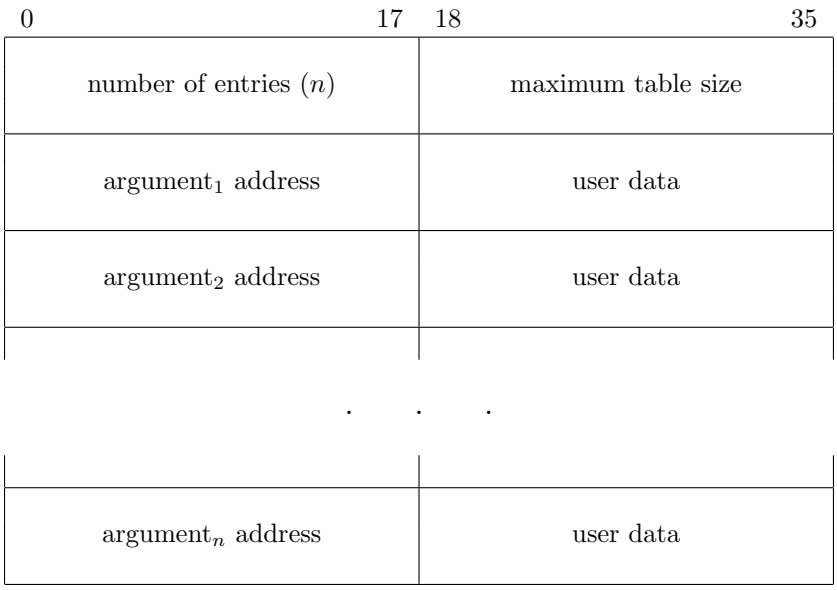


Figure 26.5: Command Table

26.4.2.1 Command Table Format

The command table used by COMND and TBLUK is shown in Figure 26.5. The first word of the table, index number zero, contains two counts. The left half of the word should contain the number of actual entries in the table. The right half is a count of the maximum number of entries. Neither count includes this header word. The left half is used to govern the binary search; the right half tells how much space was allocated so that TBADD won't overrun the allotted space.[7]

The remainder of the data words (as specified by the left–half count) contain an argument address in the left half. The argument itself contains optional bits that control TBLUK and the text of the ASCIZ string. The right half of each table entry is available to the user program (except, see CM%ABR below).

The argument pointed to by the left half of each table entry can have one of two formats. The format is selected by the value of bits 0:7 of the first word of the argument; see Figure 26.6. If bits 0:6 are non–zero, then the word is considered to be the start of the ASCIZ string for this table entry. If bits 0:7 are all zero, the string is the null string.

The second argument format permits flags to be associated with a table entry. When bits 0:6 are all zero and bit 7, CM%FW, is one then the string actually begins in the next word of the argument, and the remainder of the first word contains data bits relevant to the string.

When flags are needed, the programmer must remember to include CM%FW among the bits in the flag word. The other flag bits include

CM%ABR Consider that this entry is a valid abbreviation for another entry in the table. The right

[7] In the example program, we avoid changing the command tables so it is appropriate for them to be in the read only psect. A program that uses TBADD or TBDEL would put the command tables in a writeable psect.

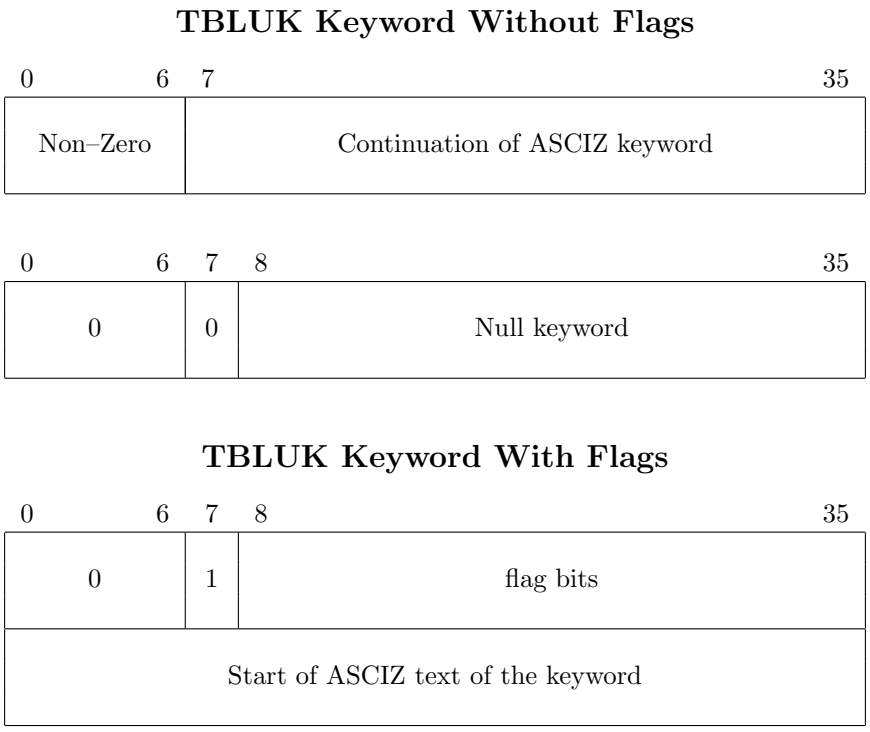


Figure 26.6: Alternate Keyword Formats for TBLUK

half of this table entry points to the entry for which this is an abbreviation. The program can set this bit to include entries in the table that are less than the minimum unique abbreviation. For example, this bit can be set to include the entry QU (for QUEUE) in the table. If TBLUK is performed for QU, it will be accepted as a valid abbreviation even though there may be other entries beginning with QU, e.g., QUIT.

CM%NOR Do not recognize this string. Even when a string is given that matches this keyword exactly, that match is considered to be ambiguous. A program can set this bit to include entries that are initial substrings of other entries in the table to enforce a minimum abbreviation of these other entries (e.g., to include D and DE in the table to enforce DEL as the minimum abbreviation of DELETE). When COMND notices this bit on for an entry it will treat it as invisible (see CM%INV below) in response to a question mark.

CM%INV Suppress this keyword in the list output in response to a "?". The program can set this bit to include entries in the table that are kept invisible because they are not preferred keywords. For example, this bit can be set to allow the keyword QUIT to be valid, even though the preferred keyword may be EXIT. The QUIT keyword would not be listed in the output given in response to a question mark. This bit is also used in conjunction with the CM%ABR bit to suppress abbreviations from the output given for a "?". This bit is not used by TBLUK, but rather it is used by COMND as part of the help process.

We define a macro, TBL, to help us build the command table. There are three arguments. The first is the command name, the second is the flags word, and the third is a dispatch address in cases where we want to override our default dispatch convention:

```
DEFINE TBL (NAME,FLAGS,DISP) <
IFNB <DISP>,<..DISP==DISP>              ;;If a dispatch is given, use it
IFB  <DISP>,<..DISP==.'NAME>            ;;If none, default to .NAME
IFB  <FLAGS>,<[ASCIZ/NAME/],,..DISP>    ;;if no flags, assemble just the name
IFNB <FLAGS>,<[FLAGS!CM%FW              ;;if flags, use them and set CM%FW
               ASCIZ/NAME/],,..DISP>    ;;
        PURGE   ..DISP
>; Define TBL
```

Now we can deal with the command table itself. For most commands we simply mention the name of the command. An entry such as we described earlier is made for the command. Remember that we must keep this table alphabetized.

```
CMDTAB: CMDTBL,,CMDTBL                  ;actual,,max num of entries
        TBL (COUNT)
        TBL (EXIT)
        TBL (HELL,CM%NOR,0)             ;"HELL" is an illegal abbrev
        TBL (HELLO)                     ;for "HELLO"
        TBL (HELP)
        TBL (PUSH)
        TBL (Q,CM%INV!CM%ABR,$QUEUE)    ;Q and QU are invisible
        TBL (QU,CM%INV!CM%ABR,$QUEUE)   ;abbreviations for QUEUE
$QUEUE: TBL (QUEUE)
        TBL (QUIT,CM%INV,.EXIT)         ;invisible alias for EXIT
        TBL (TYPE)
CMDTBL==<.-CMDTAB>-1                    ;number of entries in table
```

The entry for "HELL" demonstrates the use of CM%NOR. In case some user might be offended by a program that accepts "HELL" as an abbreviation of "HELLO", we insert an entry for HELL marked as "no recognition". This will prevent HELL from being accepted as an abbreviation for HELLO.

The entry for QUIT is marked as invisible. Although QUIT is accepted as a command to terminate this program, the preferred command is EXIT. Note that we provide an explict third argument to direct the QUIT command to .EXIT.

Since QUIT exists without visible documentation, we strive to make Q and QU legal abbreviations for the QUEUE command. This is done by adding explicit entries for Q and QU that are marked as invisible abbreviations that point to the entry for the QUEUE command. The reason it is necessary for this entry to point to $QUEUE, the table entry for the QUEUE command, rather than to .QUEUE, the dispatch address for QUEUE, is that if recognition is used on Q, the COMND JSYS needs the pointer to $QUEUE to type the balance of the recognized text.

26.4.2.2 TBLUK JSYS

We came to discuss the TBLUK JSYS because the .CMKEY function of the COMND JSYS uses it. We will briefly examine how to use TBLUK directly.

The TBLUK JSYS compares a string in the caller's address space with strings indicated by a standard–format command table. This call is used to implement a consistent style of user program command recognition and abbreviation. The TBLUK JSYS call performs the function of string lookup in the table. Related JSYSes, TBADD and TBDEL, perform the functions of adding to and deleting from the table.

The addresses in the command table must be sorted according to the alphabetical order of the strings. This order results in efficient searching of strings and determination of ambiguous strings. Letters are always considered as upper case: the strings ABC and abc act as equivalent strings.

To use TBLUK, load register 1 with the address of the header word of the command table. Load register 2 with a byte pointer to the string that you want to look up in the table.

When TBLUK returns, register 1 will contain the address of the entry that matches the string, or the address in the table where the entry would be if it were in the table. Register 2 contains flags to inform you of whether the search was successful.

TL%NOM The input string does not match any string in the table.

TL%AMB The input string is ambiguous; it matches more than one string in the table.

TL%ABR The input string is a valid abbreviation of a string in the table.

TL%EXM The input string is an exact match with a string in the table.

Register 3 is returned with a byte pointer to the remainder of the string in the table if the match was an abbreviation. This string can then be output to complete the command.

26.4.2.3 TBADD JSYS

The TBADD JSYS adds an entry to a command table. Put the address of the header word of the command table in register 1. Put the new table entry itself in register 2. When TBADD is done it will return the table address of the new entry in register 1. Errors occur if either the table fills up or the

given string already exists in the table. Clearly, the command table must be allocated in memory that is writeable by the program.

There are a few circumstances in which a programmer might want to construct a keyword table dynamically. One example is in a program that deals with a set of named resources, say system devices, the complete list of which is not known when the program is assembled. At runtime, the program can query the operating system to learn the appropriate names from which it can create a keyword table in which only those names appear.

26.4.2.4 TBDEL JSYS

The `TBDEL` JSYS deletes an entry from a command table. Put the address of the header word of the command table in register `1`. Put the address of the entry to be deleted in register `2`. This address is the same as the address returned by `TBADD` or by `TBLUK` in register `1`. An error occurs if the table is empty or if the address given is not present in the table. Clearly, the command table must be allocated in memory that is writeable by the program.

26.5 Programming Conventions

There are a number of unrelated facets of assembly language programming that we now want to discuss. These are items that we were able to do without in our earlier efforts, but that are relevant when building complex programs for general use.

26.5.1 Version Numbering

In order to keep track of which version of a program is being run, it is a good idea to record the version number in the executable image of the program. It should be possible to relate that copy of the version number back to a particular source file. Version numbering is of particular importance when a program is made available to several different sites: when a new version is ready, not everyone will install it at the same time. This means that when the program is reputed to fail, it can be determined which version is failing.

There are five components to the version number word:

`VI%DEC` This single bit field specifies how the other fields are to be interpreted. If this bit is zero, the fields are interpreted as octal numbers; if it is one, the other fields are interpreted as decimal numbers. (The name `VI%DEC`, and the names that follow, are among the symbols defined in the `MACSYM` universal file.)

`VI%MAJ` This field is the *major version number*. This number is supposed to count each time a major change or addition of functionality is made to the program.

The `MACSYM` file defines `VI%MAJ` as `777B11`, that is a nine bit field spanning bits 3–11 inclusive. When a field is defined this way, we have the opportunity to use another helpful macro from `MACSYM`. The `POINTR` macro generates a byte pointer from two arguments: the first is an address expression that identifies the word that contains the byte; the second is a *field* that identifies the bits within the word to fetch. `POINTR` uses the `FLD` and `POS` macros, as explained earlier in this chapter, to build an appropriate byte pointer.

`VI%MIN` This is the minor version number. This field should be changed each time a new function is installed. When the major version is changed, the minor version number should be reset to zero.

`VI%EDN` The edit number is changed for each bug fix or editing session; the edit number should never be reset to zero.

`VI%WHO` This 3–bit field identifies who last edited the program. By convention a value of `0` means that the edit was done by the program developers at Digital; `1` signifies a change by DEC software specialists. We use the value `2` that represents a customer site.

These numbers are changed manually.[8] Care should be taken to make sure that version numbers are faithfully updated to reflect the changes made to the program.

Our Small Executive program is now in major version `2`, minor version `7`, edit `14`.

```
VWHO==2                                 ;who last edited program
VMAJOR==2                               ;major version
VMINOR==7                               ;minor version
VEDIT==14                               ;edit version
```

These numbers are formed into a version number word. Typically, either a `BYTE` pseudo–op is used to form the version number word, or the `FLD` macro may be used:

```
BYTE (3)VWHO(9)VMAJOR(6)VMINOR(1)0(17)VEDIT      ;version number

FLD(VWHO,VI%WHO)!FLD(VMAJOR,VI%MAJ)!FLD(VMINOR,VI%MIN)!FLD(VEDIT,VI%EDN)
```

There are three standard places to put the version number word:

- In section 0 programs, put the version number in the location called `.JBVER` which is address `137` in the job data area, as kept by TOPS–10.
- In extended addressing programs, add the version to the program data vector (PDV). This is accomplished via a command switch to LINK: `/PVDATA:VERSION:` followed by either a constant value or the name of a symbol whose value is used to set the version number. This switch can be part of a command file to LINK, a batch script, or it can be an argument sent via the `.TEXT` pseudo–op.
- In any program, place the version word as the third item in the program *entry vector*, as explained below.

The EXEC command `INFORMATION VERSION` reports the contents of the version number word of the program that you are running. This program's `HELLO` command will report its version number.

26.5.2 Entry Vector

There are two ways in which we can describe where to start the execution of our program. For programs that use section 0, the starting address is conventionally placed by LINK in the right half

[8] Complex programs are assembled by scripts to the batch processing facility. Sometimes these scripts contain instructions that increment the edit number automatically.

of location 120, .JBSA, in the job data area. (Link sets the left half of this word to the first free location at and above which nothing has been loaded.)

For programs that do not use section zero, TOPS–20 implements an *entry vector*. The entry vector is a series of consecutive words that describe the starting address, the reentry address, and the version number. Additional words beyond these can be used for other purposes. The first word in the entry vector (i.e., the word at offset +0 from the given vector) is executed by the EXEC in response to a START command, or in any other circumstance that starts the program, e.g., "$g" in DDT. The second word of the vector (offset +1) is executed by the EXEC in response to the command REENTER. The REENTER command allows a program to be started at an alternate entry point as defined by the program. The word at offset +2 from the given entry vecrtor is the program version word.

For our small executive program, we define the following entry vector:

```
EVEC:   JRST    START       ;START entry point
        JRST    START       ;REENTER entry point
VERSIO: FLD(VWHO,VI%WHO)!FLD(VMAJOR,VI%MAJ)!FLD(VMINOR,VI%MIN)!FLD(VEDIT,VI%EDN)
                            ;Version number, Labeled for Hello
EVECL==.-EVEC               ;Length of vector
```

The address and length of the entry vector may be used instead of the program's starting address in the END statement. END requires a one–word argument that specifies the length of the entry vector in the left half, and the first word of the entry vector in the right half. The length of an entry vector is limited to 177 (octal) words.

```
        END     <EVECL,,EVEC>
```

When left half of the argument is omitted, END uses octal 254000 as the "length" of the entry vector. This is the code for the JRST instruction. As the length of an entry vector, 254000 signifies that the old form of starting address has been specified. Customarily the LINK program will copy the right half of this quantity to location 120, .JBSA, in the job data area. When the EXEC sees an entry vector with a length of 254000, it starts the program by using the value found in the right half of .JBSA. The EXEC does *not* use the value from the right half of the old–format entry vector.

There are two JSYS calls related to entry vectors: GEVEC, *Get Entry Vector*, and SEVEC, *Set Entry Vector*. We mention these for completeness; for details consult [MCRM].

26.6 Exercises

26.6.1 Resource Deallocation at Reparse

One problem with the parsing technique that we have discussed is that during the course of a parse it may be necessary to obtain resources (e.g., JFNs or stack space) that should be released on reparse or if an error occurs. For example, as mentioned in Section 26.4.1, page 492, our handling of the JFN for the TYPE command is less than ideal. For example, the sequence of commands depicted below will force the program to get a JFN and then halt without disposing of it:

```
Small Executive>Type file
                         ^ Type escape to complete the file name.
                           Then type CTRL/U  to force a reparse.
Small Executive>Exit
```

Design an alternate strategy. You should be able to handle a command such as

```
TYPE filename (PAGE) n CONFIRM
```

where the help message for "n" includes the actual number of the last page in the file.

26.6.2 SYSTAT Command

Add a routine to our small executive which parses a list of octal terminal numbers, separated by commas, and prints out who is using the terminal, e.g.,

```
Small Executive>SYSTAT 3,5,6,11
Terminal  User
 3        GORIN
 5        OPERATOR
 6        RMK
11        JQJOHNSON
```

You will need to use the GETJI and DIRST JSYSes to obtain the information to be printed. These are documented in [MCRM].

One problem that you must overcome is where to store the terminal numbers as you parse them. One place might be the stack, but such a use of the stack may complicate error handling and reparsing.

Chapter 27

Process Structure

In TOPS–20, a *process* or *fork* is the executable entity; every program is run within a process. Each process contains all of the resources that we have identified as being crucial to the execution of a program: a process has its own address space, its own accumulators, and a program counter. TOPS–20 schedules the running of individual processes rather than entire jobs; each process is scheduled independently of other processes.

TOPS–20 allows a job to have multiple processes that may run simultaneously. Every process exists within a job and bears some particular relationship to the other processes within the job. There are system calls by which a program can create a new process, load an executable file into the address space of that process, and start it. The originating process, called the *superior* process, can control the created (*inferior*) process in several ways. In the usual TOPS–20 environment, the EXEC program is superior to any process that you might run.

Each process has its own address space that may or may not be shared with other processes in the same job. When a new process is created, the superior process may share a portion of its address space with the newly created process.

The relationship among processes in a job is depicted in Figure 27.1. The processes in a job form a tree. Each process (excepting the top level process) has precisely one immediate superior process. A process can have zero or more inferior processes. Two processes are said to be parallel if they share the same immediate superior. A process can create and control inferior processes. A process cannot create a parallel or superior process. In most cases, the top level process is the TOPS–20 EXEC command language. The word *fork*, meaning one of these processes, comes from the forking structure of the branches of the process tree.

In this example, process A is the immediate superior of process D. Processes A, B, and C are parallel, as are processes F and G. Processes D and E are not parallel because they do not share a common immediate superior.

27.1 Process Identifiers

In order to communicate with another process or to exert control over another process, each process must be addressable by some identifier. This identifier is called a *fork handle*, since a handle is what you grasp when you want to manipulate a fork.

A fork handle is a number in the range from `400000` to `400777`. It is meaningful only to the process

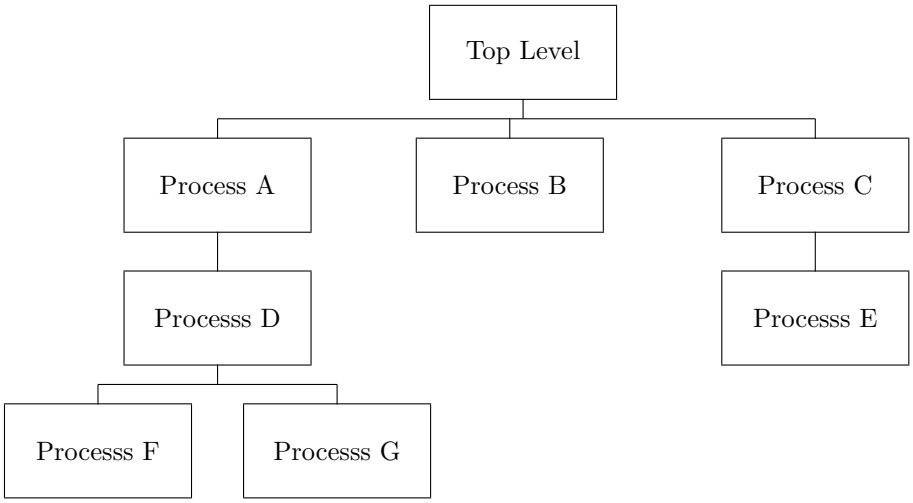


Figure 27.1: TOPS–20 Process Structure

that it was given to, i.e., the superior process. In the example above, the top level process may have had the handle `400003` returned when it created process C. It can use that handle when it executes monitor calls to manipulate process C. Meanwhile, process D might have had the same number assigned as its handle on process G.

There are several standard process handles that are never assigned by the system. These have specific meaning when used by any process within the job's process structure. These handles can be used when a process needs to communicate with a process other than an immediate inferior, or when it needs to address multiple processes at once. These special handles are

`.FHSLF` The current process, i.e., Fork Handle to SeLF.

`.FHSUP` The immediate SUPerior of the current process.

`.FHTOP` The top level process in the job structure.

`.FHSAI` The current process (Self) And all its Inferiors.

`.FHINF` All of the INFeriors of the current process.

`.FHJOB` All processes in the job structure.

In the job structure depicted above, from the viewpoint of process D, the following relations hold:

`.FHSLF` is process D.

`.FHSUP` is process A.

`.FHTOP` is the process called "top level".

`.FHSAI` includes processes D, F, and G.

`.FHINF` includes processes F and G

`.FHJOB` includes all processes A through G and "top level".

27.2 Applications for Multiple Processes

Multiple processes are appropriate in a variety of applications. As these tend to be fairly complex situations, we will only sketch the outlines of these examples.

27.2.1 Asynchronous Processes

While a programmer is at the terminal, possibly in the middle of debugging a program, it may become necessary to send a message via MAIL. By means of the EXEC command `PUSH` a new EXEC process is started in parallel to the process being debugged. That EXEC can in turn run MAIL beneath it. When the user is done sending the message, the `POP` command to the inferior EXEC will terminate it, and control of the terminal will be passed back to the EXEC from which the original `PUSH` occurred. The other process may then be continued.

27.2.2 Preservation of Editor Context

A process is relatively expensive to create and initialize. If there is an expectation that this process will be used several times during the course of a job, it might be preserved in a fork by itself. A common example of this is an editor program. The current file, position in the file, macros, modes, and contexts may be difficult to reestablish. The usual edit, compile, debug, and edit loop for program development frequently requires resumption of the editing function. For greater efficiency when it becomes necessary to return to the editor, some versions of EXEC keep the editor in a process parallel to the one in which the compiler and debugger are run. Then returning to the editor is accomplished by continuing the editor process.

27.2.3 Cooperating Processes

In some applications, two terminals must access a common data base. One very effective way to coordinate this data sharing and multiple access updates is by running the data base service as a process that has two inferior processes, each of which runs one terminal. Although there are other ways to address this problem, the multiprocess approach is among the most straightforward.

27.2.4 IDDT

In our discussion of DDT thus far, we have observed that the debugger is present in the very process that is being debugged. There are circumstances where having the debugger in the same address space as the buggy process is undesirable. For one thing, the process may need the space occupied by DDT; for another, the bugs in the process may be smashing portions of DDT. A special version of DDT, called IDDT (for *invisible* DDT), can be placed in a process superior to the one being debugged. Then the inferior can be monitored and changed by IDDT, without the target process being aware that DDT is present.[1]

[1]The use of IDDT seems to have lapsed with the increasing use of extended addressing. DDT now parks itself in a high section number where it is out of the way of most programs.

27.3 Process Communication

Processes can communicate with other processes in the same job in several ways. A superior process can create an inferior process and completely specify the contents of the new process. The superior can start, suspend, or resume the operation of its inferiors. When the inferior process finishes its assigned tasks, the superior can remove that process (and its inferiors) from the job structure.

The superior can make any of its own pages part of the inferior's address space. The two processes can then communicate through this shared memory space. The inferior's access to the superior's pages is limited by whatever the superior dictates when this mapping is established.

The pseudo–interrupt system and the interprocess communication facility can be used for communicating messages and changes in status between two processes. These topics will be discussed in later sections.

27.3.1 Processor Emulation

While hardware engineers were designing the Toad system, software engineers wanted to debug the Toad version of TOPS–20. Since the software engineers did not have a Toad to work with, they invented an emulation of the processor to run on the KL10.

The Toad, of course, was designed to execute most instructions in the same way as the KL10. The Toad differed from the KL10 in the privileged instruction set, the instructions for controlling devices, interrupts, and such, that are illegal for user programs to perform. The idea was for the KL10 to execute ordinary instructions at full speed and to fault whenever a privileged instruction was attempted.

In the emulation, the memory space of the emulated machine resided in a process inferior to the emulator. The emulator would load the target program into that memory space and start the program. Whenever the emulated program executed a privileged instruction, that process would stop and hand control to the emulator. The emulator would examine the failing instruction and change the inferior's memory or accumulators in a manner consistent with the emulated processor having executed the instruction. (This would include such things as reading a disk file into the emulated program's memory space.) Then it would continue the process at the instruction following the one that faulted.

It was tedious to build the emulator; it did not run fast. However, by the time the hardware was ready, the emulated TOPS–20 had run all of its initialization code and started timesharing.

27.4 Example 17–A: Server for the PUSH Command

The `PUSH` command that is implemented in the Small Executive program gives us an example of some of the system calls necessary for the creation and supervision of multiple processes.

The server for the `PUSH` command begins by supplying a guide phrase and obtaining confirmation of the command. Then we get a new JFN for the file `SYSTEM:EXEC.EXE`. The logical name `SYSTEM:` usually includes `PS:<SYSTEM>` where the executable image of the EXEC command language is found. We shall use this JFN later for initializing the address space of a newly created process.

```
        SUBTTL Example 17-A, the server for the PUSH command

.PUSH:  NOISE   (Command Level)
        CONFIRM                         ;tie off command
        MOVX    A,GJ%OLD!GJ%SHT         ;try to get an EXEC
        HRROI   B,[ASCIZ/SYSTEM:EXEC.EXE/]
        GTJFN
         ERJMP  ERROR
        MOVEM   A,EXCJFN                ;save JFN we got for the EXEC
```

27.4.1 Process Creation

We turn our attention to creating a new process and getting a fork handle for it. The CFORK JSYS creates a new process inferior to this one. It returns the handle to the new process in register 1. CFORK has several options; register 1 should be set up with flags prior to calling CFORK. The flags accepted by CFORK and their meanings are these:

CR%MAP — Set the inferior's page map to be the same as that of the current process, via indirect pointers. If this bit is on, the inferior will share the same pages as the creating process. If this bit is off, the inferior process will have no pages in its map; in such a case, the creating process would then use PMAP or GET to add pages to the inferior's map. In our example, we shall leave CR%MAP off because we intend to set the inferior's address space by means of a GET JSYS.

The CR%MAP flag allows the inferior to see the same address space as that of the superior. The inferior process will have read and write access to the superior's address space. The pages are shared, and changes made by one process will be seen by the other.

CR%CAP — Set the inferior's capabilities to be the same as those of the current process. If this bit is off, the inferior process has no special capabilities. We shall set this bit to pass all of our capabilities to the newly created inferior.

CR%ACS — Set the inferior's accumulators from the block whose address is in register 2. If this bit is off, the inferior's accumulators are set to zero; the contents of register 2 are ignored. We will leave this bit off.

CR%ST — If this flag is on, start the new process at the address specified in the right half of this accumulator. If this bit is off, the inferior process is not started; the right half this register is ignored. This option exists because in some cases the same program wants to be executing as two or more processes; starting the new process makes sense only after you have set up its map. In this example, since we do not initialize the inferior's map, we do not attempt to start it either.

For our program, we simply set CR%CAP in register 1 and perform the CFORK. The result will be a new process inferior to our own, and the process handle will be returned in register 1.

```
        MOVX    A,CR%CAP                ;pass on our capabilities
        CFORK                           ;as we create a new fork
         ERJMP  ERROR
        MOVEM   A,FKHAN                 ;save fork handle.
```

27.4.2 Setting the Map of an Inferior Process

We now have a *virgin process*, a new process that contains an empty address space.[2] Recall that the PMAP JSYS allows us to specify a process handle and a page number within that process as an argument. Thus, we could load the address space of the new process by means of PMAP.

Although it would be possible for us to use PMAP to deal with the file, the GET JSYS is far more useful. GET understands the format of an executable (EXE) file. The GET JSYS, when given a JFN and a fork handle, reads the first page of the file associated with that JFN. That page, which is (confusingly) called the *directory page* of the executable file, describes how to load the remainder of the file into the specified process. Usually, the GET JSYS loads the file into memory by means of PMAP operations.[3] GET figures out the file format and does the appropriate JSYS calls, saving us from having to interpret several different file formats. GET even knows not to use PMAP when the file resides on magnetic tape.

To use GET we load register 1 with a process handle in the left half and the JFN and flag bits in the right half. For our present purposes, we need no flag bits. Recall that register A already contains the appropriate fork handle:

```
                                ;form JFN,,Fork Handle:
HRL     A,EXCJFN                ;stuff fork with JFN for EXEC
MOVS    A,A                     ;Fork Handle,,JFN for GET
GET                             ;Copy EXEC.EXE into new fork
 ERCAL   FATAL
```

- The GET JSYS copies or maps an executable file into the memory space of the specified process. The entry vector of the process is updated from the file. GET can be executed for either sharable or nonsharable save files that were created with the SSAVE or SAVE JSYS calls, respectively. The file must not be open when GET is called.
- When GET is executed for a sharable save file, pages from the file are mapped into pages in the process; any previous contents of those process pages are overwritten. Pages of the process that are not specified by the file are left unchanged. For a sharable save file, GET changes the ownership of the JFN to the newly loaded process. Usually, that JFN will be released when the process is terminated. The superior process should make no further reference to that JFN.
- When GET is executed for a nonsharable save file, individual words of the file are copied into the process. The JFN of the nonsharable file will be released by GET.
- GET never loads the accumulators.
- An *execute–only file* is a file to which the user has execute access but not read access. When GET is attempted using a JFN that refers to an execute–only file, particular restrictions apply. It is necessary to map the entire execute–only file;[4] the file can be mapped only into a virgin process. The resulting process will be an *execute–only process*. Execute–only files and processes are designed to protect proprietary software from unauthorized copying. Some of the means by which a superior process can manipulate an inferior are blocked when the inferior is an execute–only process.

[2] A virgin process is one that has just been created by CFORK, but no operations to change its address space, PC, accumulators, interrupt system, traps, etc. have been performed.

[3] In a sharable EXE file (the usual case), the description of the data being loaded may occupy more than just the first page of the file. In a non–sharable file, the SIN JSYS is used to read the file, instead of PMAP.

[4] You have not been told how to map less than the entire process, but it is possible to tell GET to select only a particular range of pages.

27.4.3 Starting an Inferior Process

Now that SYSTEM:EXEC.EXE is loaded into an inferior fork, we should start that fork. Starting a fork is easily done. We load register 1 with a process handle. The right half of register 2 is the offset in the starting vector; we set the offset to zero, signifying the main starting address. (An offset of 1 specifies the reentry address.) The process is started at the specified position in the entry vector.

```
MOVE    A,FKHAN                 ;get the fork handle back
SETZ    B,                      ;offset 0 in entry vector
SFRKV                           ;Start the fork
 ERCAL  FATAL
```

Special considerations apply when the EXE file contains a program with a TOPS–10 style job data area:

- If the process has a TOPS–10 format entry vector (i.e., a JRST in the left half), then the left half of register 2 in the SFRKV call is the start address offset. The only legal offsets are 0 and 1, and they are legal only for entry vector position 0. For TOPS–10 entry vectors, the left half of register 2 will be added to the contents of the right half of .JBSA to calculate the starting address.
- For TOPS–10 entry vectors, SFRKV vector position 0 uses the contents of the right half of .JBSA as the start address. Vector position 1 uses the contents of .JBREN as the reentry address.

27.4.4 Process Wait and Termination

There's no requirement that a superior fork to wait for its inferiors to complete. Any other processing that it might want to perform can be done while the inferior is running. However, the small executive program has nothing else to do, so it waits until the inferior process finishes. The WFORK JSYS requires a fork handle in register 1. In our case, that fork handle is left over from the call to SFRKV. When the freshly started fork terminates via HALTF or by means of other errors, this fork will continue from the WFORK.

When the inferior process terminates, we have no further use for it. We shall use the KFORK monitor call to "kill" the inferior process. When a process is killed, all private memory acquired by that process is released. All JFNs that the process created are released.

```
WFORK                           ;wait for it to complete
 ERCAL  FATAL
KFORK                           ;now kill that fork
 ERCAL  FATAL
RET
```

It is expensive to create or kill a process. If there is a likelihood that a second inferior process will be needed after a first has been killed, save time by transforming the first process into the second. Simply unmap all the pages of the first process via one multiple–page PMAP, and map the new process via GET. This is much more efficient than killing the first process and creating a new one.[5]

[5]However, if either process is an execute–only process this technique does not work; a new process must be created. If the first process is execute–only, its pages can not be unmapped; if the second process is execute–only, it can be mapped only into a virgin process.

27.4.5 Complete Server for the PUSH Command

To summarize what we have said, we present the complete program for the PUSH command.

```
        SUBTTL Example 17-A, the server for the PUSH command

.PUSH:  NOISE   (Command Level)
        CONFIRM                         ;tie off command
        MOVX    A,GJ%OLD!GJ%SHT         ;try to get an EXEC
        HRROI   B,[ASCIZ/SYSTEM:EXEC.EXE/]
        GTJFN
         ERJMP  ERROR
        MOVEM   A,EXCJFN                ;save JFN we got for the EXEC

        MOVX    A,CR%CAP                ;give our capabilities
        CFORK                           ;to newly created fork
         ERJMP  ERROR
        MOVEM   A,FKHAN                 ;save fork handle.
                                        ;form JFN,,Fork Handle:
        HRL     A,EXCJFN                ;stuff fork with JFN for EXEC
        MOVS    A,A                     ;Fork Handle,,JFN for GET
        GET                             ;Copy EXEC.EXE into new fork
         ERCAL  FATAL

        MOVE    A,FKHAN                 ;get the fork handle back
        SETZ    B,                      ;offset 0 in entry vector
        SFRKV                           ;Start the fork
         ERCAL  FATAL
        WFORK                           ;wait for it to complete
         ERCAL  FATAL
        KFORK                           ;now kill that fork
         ERCAL  FATAL
        RET
```

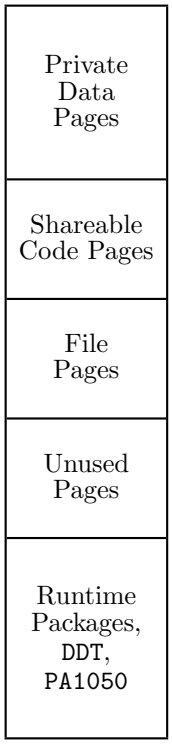
27.5 Efficiency Considerations

The process is the basic consumer of resources in TOPS–20; it is the unit to which TOPS–20 allocates memory, CPU, and I/O resources. All work done within the system is performed by individual processes. TOPS–20 schedules each process independently and asynchronously.

It is important to understand what resources are associated with each process so we can minimize the impact of a process on the system as a whole. In many systems the size of physical memory is a significant constraint. In such cases it is extremely desirable that each process be kept as small as practicable. In order to keep the process small, we must understand what information is contained within a process. The components of a process are shown in Figure 27.2.

The *virtual process size* is the sum of code pages, private pages, and file pages. Code pages consist of the code that you write, any runtime support packages that your program may need (such as FOROTS, LIBOL, or PA1050), plus the monitor code that your program invokes by executing the various JSYS instructions.

Private pages are the variables, arrays, buffers, and COMMON areas defined by your program, plus the



Each TOPS–20 process includes:

- Virtual Address Space
- Program Counter
- Accumulators
- Sharable Code
- File Pages
- Data Pages

In addition to the pages shown here, the TOPS–20 monitor code, work areas, and mapped files also contribute to the space used by each process. For memory management purposes, these pages are counted as part of the process.

`FOROTS` or `LIBOL`, the runtime support packages for Fortran and Cobol, will be present in the process address space when requested. `DDT` and `PA1050` (the TOPS–10 emulation package) may also be present.

Figure 27.2: A TOPS–20 Process

working areas for the runtime packages, plus the monitor work area associated with the process. File pages include the pages that your program maps via `PMAP`, plus the pages that are mapped by the monitor (or runtime packages) to perform sequential I/O.

The *effective process size*, or *working set size*, is determined dynamically. The effective process size is dependent on the program currently being executed. Every page that is touched by the process within a 2–second window of CPU time is counted in the working set.[6] The effective process size is the sum of code pages touched during execution, plus the work space used, plus file pages touched, plus any runtime or monitor code that is called. TOPS–20 allocates physical memory to accommodate the effective process size; the virtual size of a process is less important. It is the effective process size that we seek to minimize. Some techniques that can help reduce the effective process size are mentioned below.

The first technique to reduce the effective process size is to write *localized* code. Often the program is partitioned into distinct phases. These phases are often loops that are executed many times. In each of these phases, all the subroutines that are unique to the phase should be grouped together in the memory space of the program. Seek to minimize the number of different pages that code crosses through. Some of these ideas are depicted in Figure 27.3.

Perhaps more important than the reduction or localization of the code is the localization of work–space references. The choice of a data structure can have significant effects on the working set size of a program. This idea has been developed in some detail in the example of array addressing in Section 21.5, page 338. For many kinds of data access, a record structure is better localized than parallel arrays.

27.6 Exercises

27.6.1 Program Execution Monitor

Modify one of your previous programs so that it keeps track of the number of times each subroutine is called, and the total CPU time spent in each subroutine. Immediately before the program terminates, print a table of names, frequency counts, and time spent in each routine.

You will want to look at the description of the `HPTIM` JSYS in [MCRM]. `HPTIM` allows your program to read the high–precision clock.

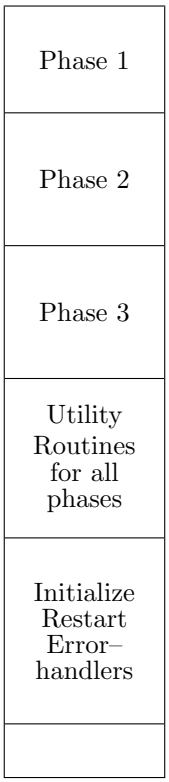
Try to think of an approach to this problem that minimizes the changes made to the existing program. Redefining `CALL` as a macro might be a good thing to try.

27.6.2 Simulate the GET JSYS

The format of an `EXE` file is documented in [MCRM]. Write a program to simulate the behavior of the `GET` JSYS, by creating, loading, and starting an inferior process from a non–sharable (i.e., `SAVE`–format) `EXE` file.

Note: most `EXE` files on a TOPS–20 system are in the more complicated sharable format as created by the `SSAVE` JSYS.

[6] The duration of this sampling window may vary depending on which processor is used.



To be worthwhile, each phase should run for more than two CPU seconds.

Avoid the use of overlays: unless special precautions are taken, an overlay will not be sharable. This means that multiple users of the same program will not benefit from the TOPS–20 page–sharing mechanism.

Long running `FOR`–loops (`DO`–loops) form natural program phases. Try to group together all routines that are called from phase 1 only. Make another group for phase 2, etc.

Utility routines are those that are called from several phases. Group these together also.

Segregate infrequently executed code to reduce the size of the main body of code. Segregate initialization code, error handlers, termination and restart handlers.

Figure 27.3: Localize the Code for Efficiency

Chapter 28

Interprocess Communication

Sometimes it is necessary to pass information between two jobs. There are two ways to do this. One is to use the *InterProcess Communication Facility* (IPCF). The other is by the use of `PMAP` to share a file for interjob communication. We will describe both of these in the sections that follow.

28.1 Interprocess Communication Facility

The Interprocess Communication Facility allows messages to be sent between cooperating processes. IPCF provides a useful means for processes to communicate requests and responses by means of exchanging message packets. Each sender and receiver is identified by a unique *Process IDentifier*, a PID, that the system assigns to each process that uses IPCF.

When a process performs the `MSEND` JSYS, a packet is sent to another process. The packet is placed in the input queue of the receiving process. The packet remains in the receiver's queue until the receiver performs the `MRCV` JSYS. The receiving process has the option of either periodically checking to see if there is an incoming packet, or enabling itself to receive a software interrupt when a packet is placed in the input queue.

A special system program, INFO, is the information center for IPCF. This process performs system functions by which names and process identifiers can be associated. Any process in the system can request these functions by sending an appropriate packet to INFO.

Information carried by IPCF is in the form of packets. Each packet is divided into a packet descriptor block and a data block. The packet descriptor block is four to nine words long. The length of the data block is specified in the packet descriptor. When we send a packet, we must supply the first four words of the packet descriptor to the `MSEND` JSYS. When we receive a packet, we must provide at least four words to accept the packet descriptor. When we make more words available, more information will be supplied by the `MRCV` JSYS. The format of a packet descriptor block is shown in Figure 28.1.

The packet itself is located in memory starting at the specified address. The length of the message, a positive number, is defined in the packet descriptor block.

The flag bits contained in the `.IPCFL` word of the packet header are defined in [MCRM].

The `QUEUE` command that is implemented in the Small Executive program gives us an example of sending and receiving IPCF packets. The object of the `QUEUE` command is to obtain the status of the

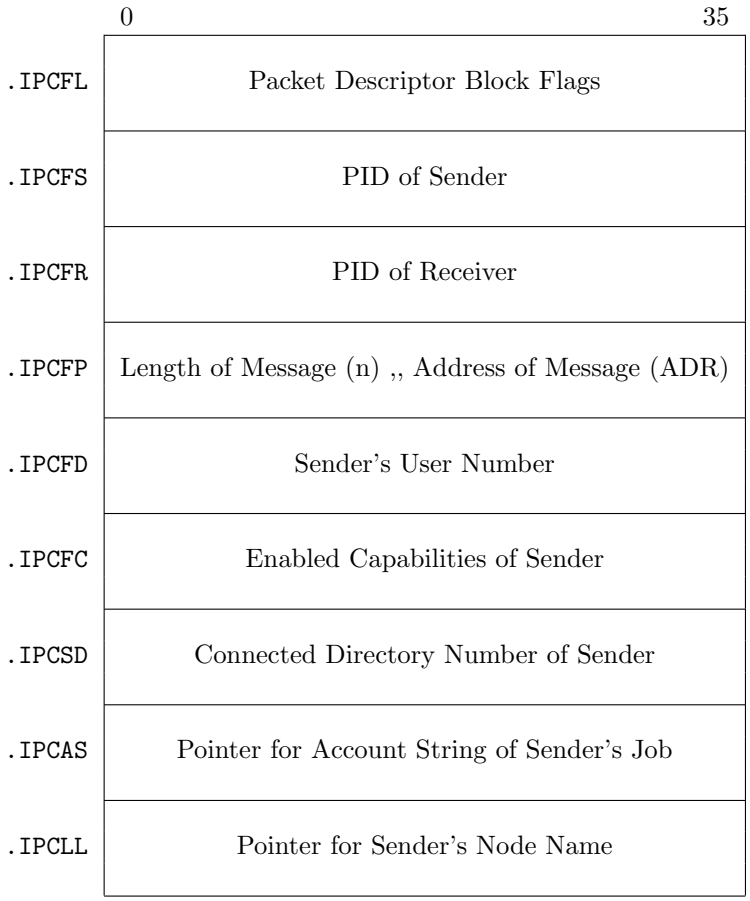


Figure 28.1: IPCF Packet Descriptor Block

several queues that are managed by QUASAR, the TOPS-20 spooling system; these queues include the batch job and printer queues. First, we must send an IPCF message to QUASAR in which we request the queue status. In response to our query, QUASAR replies with one or more messages. Each of these messages contains the status of several queued requests. We now present the server for the `QUEUE` command.[1]

28.2 Example 17–B: Using IPCF

28.2.1 Obtaining PIDs

This server, our example 17-B, begins by supplying a guide phrase and confirming the command. The program must then obtain two process identifiers. First we will get the PID that identifies QUASAR; then we will obtain a PID by which this process will be identified. We use the IPCF utility JSYS, `MUTIL` to obtain the PID for QUASAR. `MUTIL` requires that register `1` be set up with the length of the argument block; the address of the argument block is given in register `2`.

For this `MUTIL` call, the argument block is three words long. The first word specifies the function code, `.MURSP`, which requests that a PID be returned from the system PID table. The system PID table holds a small number of commonly used PIDs. Among these are the PIDs for INFO, QUASAR, and ORION, which all are software systems that make heavy use of IPCF. In the second word of the argument block we specify which of the system PIDs we want. The code `.SPQSR` tells `MUTIL` that we want the PID for QUASAR. If all goes well, `MUTIL` will return QUASAR's PID in the third word of the argument list. We store this as `QSRPID`.

```
.QUEUE: NOISE   (STATUS DISPLAY)
        CONFIRM
;First, we need to get PIDs for QUASAR and for this process.
        MOVEI   A,3                     ;length of arg block for MUTIL
        MOVEI   B,IPCBLK                ;address of block for MUTIL
        MOVEI   C,.MURSP                ;get PID from system PID table
        MOVEM   C,IPCBLK                ;Store as function or MUTIL
;Get QUASAR's PID
        MOVEI   C,.SPQSR                ;Code to request QUASAR's PID
        MOVEM   C,IPCBLK+1              ;from the system PID table
        MUTIL
        ERJMP   ERROR
        MOVE    C,IPCBLK+2              ;QUASAR's PID returns in
        MOVEM   C,QSRPID                ; arg blk. Save QUASAR's PID
```

Once we have QUASAR's PID, we must get one of our own. First, we check to see if we might have executed this code before. If we have done so, then there is no need to get an additional PID; we reuse the one that we obtained previously.[2]

Again, we use the `MUTIL` JSYS. This time we set `.MUCRE` as the function in the first word of the argument block. This requests the creation of a PID. The second word of the argument block is set

[1] This example is designed for the release 4 version of QUASAR. It may not work properly with other versions of QUASAR. Consult the file `QSRMAC.MAC`.

[2] This program may be too cautious. It always obtains a copy of QUASAR's PID to guard against the possibility that QUASAR may have crashed and had a new PID assigned since the last time this command was run.

to .FHSLF to identify ourselves as the process with which this PID will be associated. Flags may be set in the left half of this word, but we don't need any.

```
        SKIPE   MYPID                   ;Is there a PID for me yet?
        JRST    QUEUE3                  ;yes, ready to send a message
        MOVEI   C,.MUCRE                ;no, must make a PID for me
        MOVEM   C,IPCBLK                ;Create PID funct for MUTIL
        MOVEI   C,.FHSLF                ;PID for this fork, no flags
        MOVEM   C,IPCBLK+1              ;store argument for MUTIL
        MUTIL
        ERJMP   ERROR
        MOVE    C,IPCBLK+2              ;returned value from .MUCRE
        MOVEM   C,MYPID                 ;save as my PID
;We have the PIDs we need.  Tell QUASAR to send us the information.
QUEUE3:
```

The PID that is created is returned in the third word of the argument block. We copy this to MYPID for later use. This PID is valid until this process performs a RESET JSYS; note the initialization code that follows our RESET will zero MYPID as part of the data area.

28.2.2 Sending Messages

Now it's time to send a message to QUASAR. We must set up the data in the message itself, and we must prepare the proper message descriptor block. The message itself is shown below. This is assembled in, and remains constant:

```
;This is the message we send to QUASAR to make it divulge the queues.
QSRMSG: QSRLEN,,.QOLIS                  ;length of block,,list queues
        0,,'SYS'                        ;flags,,3 letter mnemonic
        0                               ;acknowledge word
        LS.ALL                          ;flags - show me everything
        1                               ;one argument following
        2,,.LSQUE                       ;2 wds this arg,,Queues I want
        LIQALL                          ;list all queues
QSRLEN==.-QSRMSG                        ;length of message
```

Some of the special symbols used in this block are defined in the universal file QSRMAC. The data contained in this message is suitable for the release 4 version of QUASAR.

The program that prepares the packet descriptor and sends the message is quite simple:

```
QUEUE3: SETZM   IPCBLK+.IPCFL           ;no flags
        MOVE    C,MYPID                 ;My PID is the
        MOVEM   C,IPCBLK+.IPCFS         ; PID of sender
        MOVE    C,QSRPID                ;QUASAR's PID is the
        MOVEM   C,IPCBLK+.IPCFR         ; PID of the receiver
        MOVE    C,[QSRLEN,,QSRMSG]      ;The length and addr of the
        MOVEM   C,IPCBLK+.IPCFP         ; packet for the descriptor
        MOVEI   A,.IPCFP+1              ;length of packet desc. block
        MSEND                           ;B still points to IPCBLK
        ERJMP   ERROR                   ;report error & return to user
```

The length of the packet header is placed in register `1`; the address of the packet descriptor block, `IPCBLK`, remains in register `2` from the calls to `MUTIL`. `MSEND` copies the packet descriptor and the packet data into the queue for QUASAR.

28.2.3 Receiving Messages

Eventually, QUASAR reads this message and responds by sending one or more packets. We initialize a flag called `FIRSTP` by which we signal the first time through the loop at `GETQRP`, *GET QUASAR RePly*. We prepare ourselves to read a packet by setting up a packet descriptor block and calling the `MRCV` JSYS.

```
        SETOM   FIRSTP                  ;Mark first time thru GETQRP
;Loop, reading the replies from QUASAR
GETQRP: MOVX    C,IP%CFV                ;flag requesting 1 data page
        MOVEM   C,IPCBLK+.IPCFL         ;in the packet descriptor flag
        SETZM   IPCBLK+.IPCFS           ;sender (system fills this in)
        MOVE    C,MYPID                 ;My PID is
        MOVEM   C,IPCBLK+.IPCFR         ;  the receiver
        MOVE    C,[1000,,MSGPAG]        ;put data on the message page
        MOVEM   C,IPCBLK+.IPCFD
        MOVEI   A,.IPCFP+1              ;length of packet descr. block
        MOVEI   B,IPCBLK                ;address of our block
        MRECV                           ;Now get the reply from QUASAR
        ERJMP   ERROR                   ;report error & return to user
```

We set the flag `IP%CFV` in the packet descriptor. This flag means that we expect an entire memory page in response from QUASAR. Basically, there are two kinds of messages, short messages and whole-page messages. The short messages are copied from the sender, through TOPS-20, to the receiver. The whole-page messages are handled by the general page mapping mechanism. The map pointer to the relevant page is removed from the sender's map and eventually added to the receiver's map. Of necessity, a whole-page message must be page aligned. It must start as a private page in the sender's address space, from which it is removed by `MSEND`. A paged IPCF packet must be received into a private or previously non-existent page of the receiver's address space. Any prior contents of the receiver's page are lost.

We don't fill in the sender's PID in the packet descriptor: we will get the next message that is queued to us regardless of who sent it. We do set our PID as the receiver's PID, and in the `.IPCFD` word

we set a message length of octal 1000, and an address that is page aligned. We are not interested in knowing the user number, capabilities, etc., of the sender, so our packet descriptor is kept to the minimum length, four.

This MRCV call will wait until a packet is present, copy that packet into the memory space that we assigned for it, and then return. Now that we have a packet, we must scrutinize it to see what we got. The first thing we do is to look at the .IPCFS word in the packet header. This word contains the sender's PID. We check to see if that PID matches our copy of QUASAR's PID. If it fails to match, someone has sent us junk mail; after briefly reporting the problem, we discard the packet.

```
        MOVE    C,IPCBLK+.IPCFS         ;get PID of sender of message
        CAME    C,QSRPID                ;was it QUASAR?
        JRST    [HRROI A,[ASCIZ/% Ignoring an irrelevant IPCF message
/]
                PSOUT                   ;someone else sent to us
                JRST GETQRP]            ;try to get QUASAR's reply
```

The remainder of what we do is specialized for this application. Normally, QUASAR's response is an ASCIZ string that commences at .OHDRS+1 in the message packet. However, if this is the first message from QUASAR, then there is a header string that we skip. The length of the header is given in the left half of the .OHDRS word. This length is added to the address .OHDRS+1 to compute the address of the actual message string. That string is sent to the terminal via PSOUT. Finally, QUASAR returns flags in the .OFLAG word. One of these is called WT.MOR, which if a one signifies that there are more packets to be processed. If this flag is set, the program loops to GETQRP; otherwise it returns to the top level.

```
        HRROI   A,MSGLOC+.OHDRS+1       ;get pointer to text block
        HLRZ    B,MSGLOC+.OHDRS         ;get block's size
        AOSN    FIRSTP                  ;is this the first message?
        ADD     A,B                     ;Yes. Point past header msg
        PSOUT                           ;output the message
        MOVE    B,MSGLOC+.OFLAG         ;get the flags from QUASAR
        TXNE    B,WT.MOR                ;Are there any more messages?
        JRST    GETQRP                  ;Yes, handle next message
        RET                             ;None left.  All done
```

28.2.4 Server for the QUEUE Command

We summarize the server for the QUEUE command, example 17-B, below:

```
        SUBTTL Example 17-B, the server for the QUEUE command.

;Select a page for IPCF replies from QUASAR
IFNDEF MSGPAG,MSGPAG==670               ;put IPCF replies on page 670
MSGLOC=MSGPAG_ ^ D9                       ;first location on MSGPAG
```

```
;This is the message we send to QUASAR to make it divulge the queues.
QSRMSG: QSRLEN,,.QOLIS                  ;length of block,,list queues
        0,,'SYS'                        ;flags,,3 letter mnemonic
        0                               ;acknowledge word
        LS.ALL                          ;flags - I want to see everything
        1                               ;one argument following
        2,,.LSQUE                       ;2 words this argument,,queues I want
        LIQALL                          ;list all queues
QSRLEN==.-QSRMSG                        ;length of message

.QUEUE: NOISE   (STATUS DISPLAY)
        CONFIRM
;First, we need to get PIDs for QUASAR and for this process.
        MOVEI   A,3                     ;length of argument block for MUTIL
        MOVEI   B,IPCBLK                ;address of block for MUTIL
        MOVEI   C,.MURSP                ;Read a PID from system PID table
        MOVEM   C,IPCBLK                ;Store as function or MUTIL

;Get QUASAR's PID
        MOVEI   C,.SPQSR                ;Code to request QUASAR's PID
        MOVEM   C,IPCBLK+1              ;from the system PID table
        MUTIL
        ERJMP   ERROR
        MOVE    C,IPCBLK+2              ;QUASAR's PID is returned in arg blk
        MOVEM   C,QSRPID                ;Save QUASAR's PID

;Get a PID for this process
        SKIPE   MYPID                   ;Is there a PID for me yet?
        JRST    QUEUE3                  ;yes, ready to send off a message
        MOVEI   C,.MUCRE                ;no, must create a PID for myself
        MOVEM   C,IPCBLK
        MOVEI   C,.FHSLF                ;I want the PID for myself with no flags
        MOVEM   C,IPCBLK+1
        MUTIL
        ERJMP   ERROR
        MOVE    C,IPCBLK+2              ;returned value from .MUCRE
        MOVEM   C,MYPID                 ;save as my PID

; Here we have the PIDs we need.  Now tell QUASAR to send us the information.
QUEUE3: SETZM   IPCBLK                  ;no flags
        MOVE    C,MYPID
        MOVEM   C,IPCBLK+1              ;my PID
        MOVE    C,QSRPID
        MOVEM   C,IPCBLK+2              ;QUASAR's PID
        MOVE    C,[QSRLEN,,QSRMSG]
        MOVEM   C,IPCBLK+3
        MOVEI   A,.IPCFP+1              ;length of packet descriptor block
        MSEND
        ERJMP   ERROR                   ;report an error & return to the user
        SETOM   FIRSTP                  ;Set this is first time through GETQRP
```

```
;Loop, reading the replies from QUASAR
GETQRP: MOVX    C,IP% CFV                ;flag to request one page of data
        MOVEM   C,IPCBLK+.IPCFL          ;in the packet descriptor flag
        SETZM   IPCBLK+.IPCFS            ;sender (system fills this in)
        MOVE    C,MYPID                  ;My PID is
        MOVEM   C,IPCBLK+.IPCFR          ;  the receiver
        MOVE    C,[1000,,MSGPAG]         ;put data on the message page
        MOVEM   C,IPCBLK+.IPCFD
        MOVEI   A,.IPCFP+1               ;length of packet descriptor block
        MOVEI   B,IPCBLK                 ;address of our block
        MRECV                            ;Now get the reply from QUASAR
        ERJMP   ERROR                    ;report error & return to user

        MOVE    C,IPCBLK+.IPCFS          ;get PID of the sender of this message
        CAME    C,QSRPID                 ;was it QUASAR?
        JRST    [HRROI A,[ASCIZ/% Ignoring an irrelevant IPCF message
/]
                PSOUT                    ;someone other than QUASAR sent to us
                JRST GETQRP]             ;try again to get QUASAR's reply
        HRROI   A,MSGLOC+.OHDRS+1        ;get pointer to text block
        HLRZ    B,MSGLOC+.OHDRS          ;get block's size
        AOSN    FIRSTP                   ;is this the first message?
        ADD     A,B                      ;Yes. Point past header msg
        PSOUT                            ;output the message
        MOVE    B,MSGLOC+.OFLAG          ;get the flags from QUASAR
        TXNE    B,WT.MOR                 ;Are there any more messages?
        JRST    GETQRP                   ;Yes, handle next message
        RET                              ;None left.  All done
```

28.3 Simultaneous Shared Access to a File

The Interprocess Communication Facility implements an arm's length communication protocol. The processes communicate with some degree of mutual suspicion. TOPS-20 provides absolute identification of the sender to the receiver by including in the received packet descriptor the logged in directory of the sender and his present capabilities. These cannot be counterfeited.

For situations where processes in independent jobs view each other in a cooperative light and with mutual trust, there is a more efficient way to pass messages. If two jobs both have thawed write access to the same file, then if they both PMAP the same page or pages of that file into their address space, then the file and the two processes are shared in common. The two processes have obtained windows into each other.

Figure 28.2 depicts two independent processes, A and B, mapping a file to which they share write access. Page `70` of process A, page `0` of the file, and page `251` of process B are all the same page. Any change made to page `70` by process A will be seen in the file and in page `251` of process B. Any change made by B to page `251` will likewise affect the file and be seen in process A. Similarly, page `71` of process A, page `1` of the file, and page `252` of process B are the same page.

It is relatively easy to effect page sharing in this way. Both process A and process B must agree on the name of the file that is being shared in this way. Both processes must be able to have read and write access to the file in the normal way. When each process does an `OPENF`, it must specify

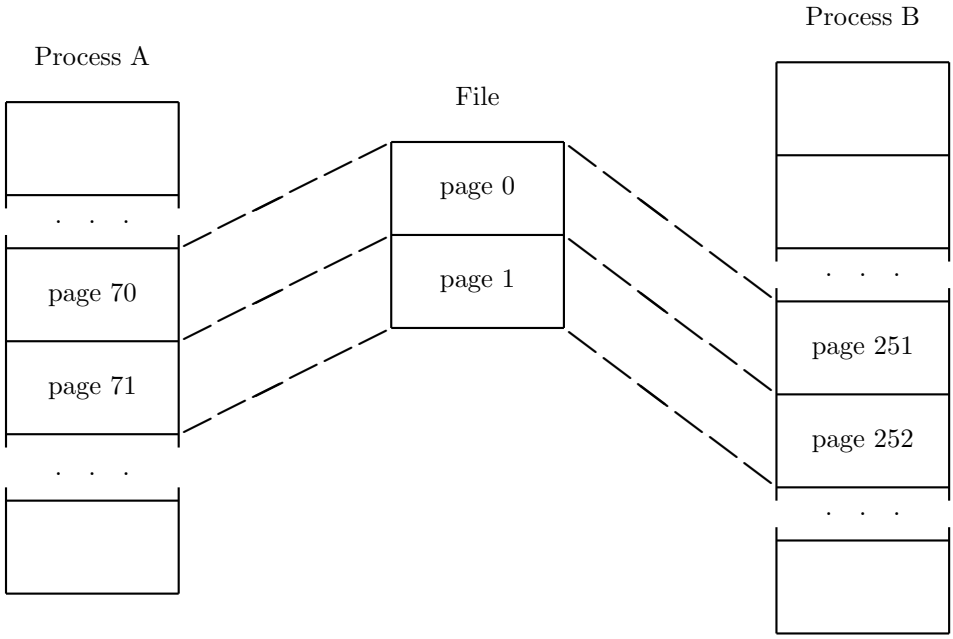


Figure 28.2: Inter-Process Page Sharing via Shared File Pages

`OF%RD`, `OF%WR`, and a flag that is new to us, `OF%THW`. The flag `OF%THW` signifies that the process seeks to obtain *thawed access* to the file.

Usually, write access is said to be *frozen*: only one process may have the file open for writing at any time. By requesting thawed access, a process tells TOPS-20 to expect and to permit several simultaneous writers. Each such simultaneous writer must seek thawed access for itself. While the file is open for thawed writes, a request for frozen write access will be refused. While the file is open for frozen write, all further requests for write access, whether thawed or frozen, will be refused. When both processes have obtained read and thawed write access to the file they can map file pages into their address space and begin sharing data.

Chapter 29

Traps and Interrupts

In concept, *traps* and *interrupts* have some similarities. Both are mechanisms by which specific events cause one instruction stream to be suspended and another instruction stream to be started. In the hardware, a trap is a mechanism by which a specific *trap routine* is run in response to a processor exception (such as arithmetic overflow or stack overflow). An interrupt is a hardware mechanism by which an *interrupt service routine* is run in response to an external event (such as the completion of an input or output operation).

The TOPS–20 software blurs the distinction between traps and interrupts. Originally all asynchronous notifications were handled through the software interrupt system. Recently, for higher efficiency, the TOPS–20 operating system has been adapted to support traps for handling arithmetic exceptions. All other events, regardless of the method by which they affect the hardware, are handled through the software interrupt system in TOPS–20. The software interrupt system, also called the *pseudo interrupt* (PSI) system allows a program to respond in different ways to a variety of events. The PSI system selects a particular interrupt service routine, as specified by the programmer, to respond to each of the various different kinds of events.

In TOPS–20 the only events that can be trapped are arithmetic exceptions.[1] A trap is a very efficient way to deal with arithmetic exceptions. In contrast, the interrupt system is less efficient but much more flexible. The PSI system can be sensitized to a variety of software–detected conditions. For example, software interrupts can be taken in response to specific characters being typed, e.g., `CTRL/C`; to conditions such as end of file or a data error in an Input/Output operation; or to events external to this process, e.g., an IPCF packet being queued as input. The software interrupt system allows the programmer to prioritize the interrupts; the most urgent can be dealt with first. While engaged in processing one interrupt, the program may be interrupted again with a request of higher priority.

Although it is possible to get quite far in programming without having to use either traps or interrupts, in many cases traps or interrupts provide the most suitable mechanism for dealing with events that are not synchronized with the behavior of the program. For example, using the `JFCL` instruction it is possible to test for arithmetic overflow after every arithmetic operation. However, it is decidedly inconvenient to do so. By means of the trap system, the program can request to be run at a specific place when an overflow does occur. Then, when an overflow occurs, the *trap routine* will run; it will be provided with the program counter that addresses the instruction that caused the trap. By examining the instruction and its result the program can determine how to proceed:

[1] Although some events such as pushdown overflow and illegal references to memory cause hardware traps, TOPS–20 reports these events via the PSI system.

a message might be printed that warns of potentially incorrect results, or the accumulator in which the incorrect result appears might be changed before returning to the instruction stream in which the trap occurred.

We will give an example of arithmetic traps now; after this example, we will explain the interrupt system.

29.1 Traps

The `SWTRP%` JSYS allows the program to tell TOPS–20 how to respond to an arithmetic trap.[2] This JSYS requires a process handle in accumulator 1. Register 2 should be loaded with the symbolic constant `.SWART` to select the function by which TOPS–20 sets the arithmetic trap location for the program. Register 3 should be set to the address of the trap block, a 4–word block as shown in Figure 29.1. A zero address in register 3 can be used to disable further arithmetic traps.

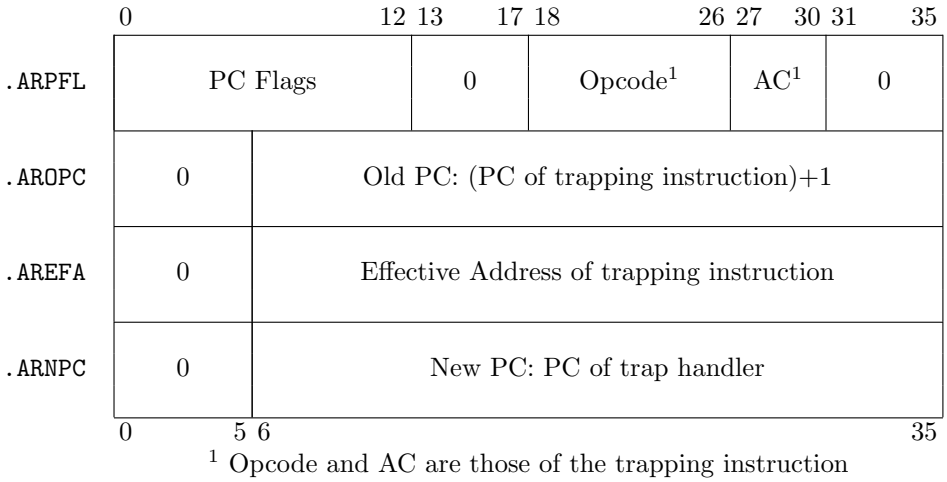


Figure 29.1: Trap Block

An arithmetic exception can be caused only by the execution of an operation specified by an instruction. The instruction whose execution results in an exception is called the *trapping instruction*. When an arithmetic exception occurs, the processor *traps*: instead of executing the next instruction in sequence, the processor suspends the execution of the present instruction stream and starts another stream.

In a trap, the processor must store the program counter and flags pertaining to the instruction from which the trap occurred. For our convenience, the processor also stores an image of the trapping instruction: the opcode and accumulator fields, and the effective address. Finally, after these items of state information have been saved, the processor commences the execution of an instruction stream that presumably will deal with the trap. All the data relevant to the trap appears in the trap block.

[2]All new JSYS names in release 4 and later versions of TOPS–20 end with a percent sign "`%`" to avoid confusion with user–defined symbols from earlier programs. Users should not define symbolic names that end with percent. Actually, all JSYS mnemonics are available with the percent, e.g., `SIN%`, for applications where the JSYS name might be ambiguous.

In greater detail, when a trap occurs the processor stores the current program counter and flags in order to allow the eventual continuation of the instruction stream in which this trap occurred. The program counter is stored in two pieces. The PC flags are stored in the left half of the .ARPFL word; the program counter itself is stored in the .AROPC, old PC, word. The flags will reflect the specific error condition that caused the trap; the program counter addresses the next instruction following the trapping instruction.

The instruction image that is stored in the trap block reflects the result of the processor's effective address computation. The processor resolves any indexed or indirect addressing to an effective address. The effective address computed for the trapping instruction is stored in the .AREFA word. The opcode and AC field of the trapping instruction are stored in the right half of the .ARPFL word; the index and indirect fields of the trapping instruction are stored as zero in this word [3].

After storing all this information, the processor commences executing a different instruction stream by jumping to the address specified in the new PC word, .ARNPC.

Instead of the 18–bit addresses that we are familiar with, we find there are thirty bits reserved in the trap block for the old and new program counters and for the effective address that was computed by the trapping instruction. We find long addresses here because this trap block is constructed to accommodate the 30–bit extended addresses that we shall talk about in Section 30.

When an arithmetic exception occurs, the processor finishes the current instruction, stores the (erroneous) results as directed by the instruction, and traps. The accumulators are not changed by the trap itself; they contain the information resulting from the execution of the instruction that caused the trap. The trap routine should save the accumulators before doing any other processing, and restore them prior to returning to the interrupted program sequence.

In order to return from the trap, the program should clear the arithmetic exception flags from the .ARPFL word, and execute an XJRSTF instruction to restore the flags and PC from the flags and PC doubleword at .ARPFL and .AROPC. The XJRSTF instruction will be further explained in the discussion of the example that follows.

29.1.1 Example 18 — Traps

The following program, example 18, is presented to demonstrate a trap handler. The explanation will follow.

```
        TITLE   TRAP, Arithmetic Traps - Example 18
        SEARCH  MONSYM,MACSYM

A=1                                     ;usual accumulator definitions
B=2
C=3
D=4
ACP=16                                  ;AC Pointer Register
P=17
```

[3] In the case of an instruction executed under XCT, the image of the arithmetic instruction will be stored, with the PC of the XCT. In the case of an instruction executed under EXTEND, the image of EXTEND and its accumulator field will be stored; the effective address will point to the the aritmethic instruction itself.

```
OPDEF   GSNGL   [021B8]                ;works only under Extend
OPDEF   GDBLE   [022B8]
OPDEF   DGFIX   [023B8]
OPDEF   GFIX    [024B8]
OPDEF   DGFIXR  [025B8]
OPDEF   GFIXR   [026B8]
OPDEF   DGFLTR  [027B8]
OPDEF   GFLTR   [030B8]
OPDEF   GFSC    [031B8]
OPDEF   GFAD    [102B8]                ;these execute directly
OPDEF   GFSB    [103B8]
OPDEF   GFMP    [106B8]
OPDEF   GFDV    [107B8]

;the following right-halfword flags are used in FXUTAB
ACF==1                                 ;result in AC
DDF==2                                 ;result in double AC, AC+1
MMF==4                                 ;result in memory

FLNDIV==1B17                           ;local flag for floating div check
                                       ;this is NOT a processor flag

PDLEN==100

OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ P,]

PDLIST: BLOCK   PDLEN

;Arithmetic trap block
TRAPB:  0                              ;Flags,,Instruction image
        0                              ;Trap PC
        0                              ;Instruction effective address
        0,,TRAPIT                      ;new PC

;Message table for various PC flags
TRPMSG: PC%OVF![ASCIZ/Integer Overflow /]
        PC%FOV![ASCIZ/Floating Overflow /]
        PC%NDV![ASCIZ/Integer Division by Zero /]
        PC%FUF![ASCIZ/Floating Exponent Underflow /]
        FLNDIV![ASCIZ/Floating Divide by Zero (Divide Check) /]
TRPMLN==.-TRPMSG
```

```
;Macro to help generate the table of floating-point operations
DEFINE FOPXX (OP) <                     ;;make table of floating operations
        F'OP    ACF                     ;;e.g., FAD     result in AC
        F'OP'M  MMF                     ;;      FADM    result in MEM
        F'OP'B  ACF!MMF                 ;;      FADB    result in Both
        F'OP'R  ACF                     ;;      FADR    result in AC
        F'OP'RI ACF                     ;;      FADRI   result in AC
        F'OP'RM MMF                     ;;      FADRM   result in MEM
        F'OP'RB ACF!MMF                 ;;      FADRB   result in Both
        DF'OP   ACF!DDF                 ;;      DFAD    result in AC and AC+1
        GF'OP   ACF!DDF                 ;;      GFAD    result in AC and AC+1
>                                       ;end of FOPXX

;Table of instructions that might cause floating underflow.
FXUTAB: FOPXX   (AD)
        FOPXX   (SB)
        FOPXX   (MP)
        FOPXX   (DV)
        FSC     ACF
        EXTEND                  ;no flags - special case
FXUTLN==.-FXUTAB

EXTBL:  GSNGL   ACF             ;result in AC (G to F)
;       GDBL    ACF!DDF         ;result in AC AC+1  (F to G) - can't overflow
;       GFIXR   ACF             ;result in AC.  G to I  - can't underflow
;       GFIX    ACF             ;result in AC.  G to I - can't underflow
;       DGFIX   ACF!DDF         ;result in AC AC+1  G to DI - can't underflow
;       DGFIXR  ACF!DDF         ;result in AC AC+1  G to DI - can't underflow
;       GFLTR   ACF!DDF         ;F to G can't overflow
;       DGFLTR  ACF!DDF         ;DF to G: can't overflow.
        GFSC    ACF             ;like FSC.  can over/underflow
EXTBLN==.-EXTBL

;Co-routine to save accumulators 0:16 on the stack.  Call via PUSHJ
;The caller of this co-routine must return with either 0 or 1 skips.

SAVACS: ADJSP   P,17                            ;add locations for 0-16
        MOVEM   16,0(P)                         ;Save AC 16 on stack
        MOVEI   16,-16(P)                       ;place on stack for some acs
        BLT     16,-1(P)                        ;store ACs 0 to 15 on stack
        MOVEI   ACP,-16(P)                      ;Address of the saved ACs
        CALL    @-17(P)                         ;return to "caller"
        SKIPA                                   ;non-skip return. Avoid AOS
        AOS     -20(P)                          ;skip return. pass skip upwards
        MOVSI   16,-16(P)                       ;restore saved acs to ACS
        BLT     16,16
        ADJSP   P,-20                           ;discard saved acs & first return
        RET

TRAPIT: CALL    TRAPNT                          ;call trap processing routine
        XJRSTF  TRAPB+.ARPFL                    ;return to trapped-from sequence
```

```
TRAPNT: CALL    SAVACS                          ;save accumulators
        MOVE    B,TRAPB+.ARPFL                  ;get the flags
        AND     B,[PC%OVF!PC%FOV!PC%NDV!PC%FUF] ;keep only the important bits
        TXNE    B,PC%FUF                        ;was it a floating underflow?
        TXZ     B,PC%OVF!PC%FOV                 ;yes, report only the underflow
        TXNE    B,PC%FOV!PC%NDV                 ;floating ovflw or no divide?
        TXZ     B,PC%OVF                        ;yes, don't report integer ovf
        TXNE    B,PC%NDV                        ;was it divide by zero?
        TXZN    B,PC%FOV                        ;yes, and floating?
        SKIPA                                   ;not floating divide check
        TXC     B,PC%NDV!FLNDIV                 ;set flag for floating no divide

        MOVSI   D,-TRPMLN                       ;-length of message tbl
TRPNT1: TDNN    B,TRPMSG(D)                     ;is flag on for this message?
        JRST    TRPNT2                          ;no flag means no print
        HRRO    A,TRPMSG(D)                     ;set LH of A to -1
        PSOUT                                   ;print msg for this flag
TRPNT2: AOBJN   D,TRPNT1                        ;loop thru msg table
;fall through

        HRROI   A,[ASCIZ/at PC = /]             ;Print the adjusted PC
        PSOUT                                   ;which is one less than the
        MOVEI   A,.PRIOU                        ;trap PC, because the trap
        MOVE    B,TRAPB+.AROPC                  ;PC points to the next
        SUBI    B,1                             ;instruction.
        MOVEI   C,10
        NOUT
        ERJMP   .+1
```

```
        HRROI   A,[ASCIZ/
Failing instruction: /]
        PSOUT                                   ;display octal image of
        MOVEI   A,.PRIOU                        ;the failing instruction
        HRRZ    B,TRAPB+.ARPFL                  ;opcode & AC of failing
        MOVEI   C,10                            ;instruction
        NOUT
        ERJMP   .+1
        HRROI   A,[ASCIZ/,,/]
        PSOUT
        MOVEI   A,.PRIOU
        MOVE    B,TRAPB+.AREFA                  ;this might be as many
        MOVEI   C,10                            ;as 30 bits of address
        NOUT
        ERJMP   .+1
        HRROI   A,[ASCIZ/
/]
        PSOUT
                                                ;special processing for underflow
        MOVE    B,TRAPB+.ARPFL                  ;get trap flags again
        TXNE    B,PC%FUF                        ;check for exponent underflow
        CALL    DOFXU                           ;process exponent underflow
        MOVX    B,PC%OVF!PC%FOV!PC%NDV!PC%FUF   ;clear selected PC flags
        ANDCAM  B,TRAPB+.ARPFL                  ;before exiting and restoring
        RET                                     ;them

;For floating underflow, we change the erroneous result to zero.  An underflow
;means that a result too small to be representable has been computed.
;the result of this instruction will be too large by a factor of approximately
;1.0E77.  It's usually much better to store a zero result instead.
DOFXU:  SKIPA   D,[-FXUTLN,,FXUTAB]             ;get length and loc of table
DOFXU9: SKIPA   D,[-EXTBLN,,EXTBL]              ;use alternate tbl. A is set up
        LDB     A,[POINT 9,TRAPB+.ARPFL,26]     ;opcode of failed instr
DOFXU1: LDB     B,[POINT 9,(D),8]               ;get opcode from table
        CAMN    A,B                             ;do we match?
        JRST    DOFXU2                          ;yes. D=index to table
        AOBJN   D,DOFXU1                        ;advance to next instr
        HRROI   A,[ASCIZ/The instruction that caused this trap is
not known to this program as one of the floating-point instructions.
I give up.
/]
        PSOUT
        HALTF
        RET

;here, register D contains index to the floating instructions table:
;the right half of FXUTAB tells where the result will be: AC, MEM, BOTH, etc.
DOFXU2: HRRZ    A,(D)                           ;where is the result?
        JUMPN   A,DOFX2A                        ;jump if any flags are set
;here for EXTEND instruction.  Go look further
        MOVE    A,TRAPB+.AREFA                  ;get 30-bit EffAdr of EXTEND
        LDB     A,[POINT 9,0(A),8]              ;get opcode field of instr
        JRST    DOFXU9                          ;look in alternate table
```

```
DOFX2A: LDB     B,[POINT 4,TRAPB+.ARPFL,30]     ;AC field of instru.
        CAIL    B,17                            ;mustn't reference stack AC
        JRST    DOFXU0                          ;We think this can't happen.
        ADD     B,ACP                           ;offset to saved acs.
        TRNN    A,ACF                           ;Is result in AC?
        JRST    DOFXU3                          ;no.
        MOVE    C,(B)                           ;computed result to C
        SETZM   (B)                             ;clear saved copy of AC
DOFXU3: TRNN    A,MMF                           ;is result in memory?
        JRST    DOFXU4                          ;no.
        HRRZ    B,TRAPB+.AREFA                  ;yes, get in-section result addr
        HLRZ    D,TRAPB+.AREFA                  ;get the section # of result
        CAIG    B,16                            ;skip if this is not an AC addr
        CAILE   D,1                             ;skip if definite AC address
        JRST    DOFXU6                          ;not an AC address
        ADD     B,ACP                           ;offset to stored acs
        MOVE    C,(B)                           ;copy result to C
        SETZM   (B)                             ;and clear result location
        JRST    DOFXU4

DOFXU6: MOVE    C,@TRAPB+.AREFA                 ;computed result to C
        SETZM   @TRAPB+.AREFA                   ;clear memory result
DOFXU4: TRNN    A,DDF                           ;is result in AC+1?
        JRST    DOFXU5                          ;no.
        LDB     B,[POINT 4,TRAPB+.ARPFL,30]     ;AC field of instr again
        ADDI    B,1                             ;advance to AC+1
        CAIL    B,17                            ;mustn't reference stack AC
        JRST    DOFXU0                          ;We think this can't happen
        ADD     B,ACP                           ;offset to saved ACs
        SETZM   (B)                             ;clear saved AC+1
DOFXU5: HRROI   A,[ASCIZ/Changing the computed result, /]
        PSOUT
        MOVEI   A,.PRIOU                        ;print result as floating
        MOVE    B,C                             ;the data item to print
        MOVEI   C,0                             ;format control flags
        FLOUT                                   ;convert floating to characters
        ERJMP   .+1                             ;(wrong if argument is G-Format)
        HRROI   A,[ASCIZ/, to zero
/]
        PSOUT
        RET

DOFXU0: HRROI   A,[ASCIZ/Instruction seems to reference the stack
accumulator.  This is very confusing.
/]
        PSOUT
        HALTF
        RET
```

```
START:  RESET
        MOVE    P,[IOWD PDLEN,PDLIST]
        MOVEI   A,.FHSLF                        ;affect this fork
        MOVEI   B,.SWART                        ;set trap block address
        MOVEI   C,TRAPB                         ;the trap block address
        SWTRP%

                                                ;now, let's make some traps:
        MOVEI   A,100                           ;Begin with zero divide,
        IDIVI   A,0                             ;which should produce an error
        CALL    PNTA                            ;now show what's in A
                                                ;fl underflow
        MOVE    A,[032400000000]                ;approximately 0.98E-31
        FSBR    A,[031777777777]                ;also, approximately 0.98E-31
        CALL    PNTA                            ;Show contents of A
                                                ;fl underflow via XCT
        MOVE    A,[032400000000]                ;approximately 0.98E-31
        XCT     [FSBR A,[031777777777]]         ;also, approximately 0.98E-31
        CALL    PNTA                            ;Show contents of A

                                                ;fl overflow from G-format
        MOVSI   B,220040                        ;a little too big to fit
        SETZ    C,
        EXTEND  A,[GSNGL B]
        CALL    PNTA
                                                ;fl underflow from G-format
        MOVSI   B,157740                        ;a little too small to fit
        SETZ    C,
        EXTEND  A,[GSNGL B]
        CALL    PNTA

        MOVSI   A,000140                         ;a small G-format number
        SETZ    B,
        EXTEND  A,[GFSC -1]                      ;should work
        CALL    PNTA                             ;expect 000040,,0

        EXTEND  A,[GFSC -1]                      ;now, a smaller number
        CALL    PNTA                             ;should underflow

        HRROI   A,[ASCIZ/done
/]
        PSOUT
        HALTF                                   ;it isn't much of a program
        JRST    START
```

```
;Print A
PNTA:   ADJSP   P,3                             ;allocate space on stack
        DMOVEM  A,-2(P)
        MOVEM   C,0(P)
        HRROI   A,[ASCIZ/The current result is /]
        PSOUT
        MOVEI   A,.PRIOU
        MOVE    B,-2(P)                         ;the saved copy of A
        MOVEI   C,10
        NOUT
        ERJMP   .+1
        HRROI   A,[ASCIZ/
/]
        PSOUT
        MOVE    C,0(P)                          ;restore accumulators
        DMOVE   A,-2(P)
        ADJSP   P,-3                            ;restore stack
        RET                                     ;return

        END     START
```

Execution of this program produces the following results:

```
@execute ex18.mac
MACRO:  TRAP
LINK:   Loading
[LNKXCT TRAP execution]
Integer Division by Zero at PC = 520
Failing instruction: 231040,,0
The current result is 100
Floating Exponent Underflow at PC = 523
Failing instruction: 154040,,734
Changing the computed result, 8.507059E37, to zero
The current result is 0
Floating Exponent Underflow at PC = 526
Failing instruction: 154040,,734
Changing the computed result, 8.507059E37, to zero
The current result is 0
Floating Overflow at PC = 532
Failing instruction: 123040,,736
The current result is 0
Floating Exponent Underflow at PC = 536
Failing instruction: 123040,,736
Changing the computed result, 8.507059E37, to zero
The current result is 0
The current result is 40000000
Floating Exponent Underflow at PC = 544
Failing instruction: 123040,,737
Changing the computed result, 1.595074E38, to zero
The current result is 0
done
@
```

29.1.2 Discussion of Traps

This program displays some interesting points. An important programming concept, the co–routine, is illustrated in a very useful application. The main program is quite short; it initializes for arithmetic traps and then causes some. The trap handler is set up by means of the call to SWTRP%:

```
MOVEI   A,.FHSLF                ;affect this fork
MOVEI   B,.SWART                ;function: set trap block addr
MOVEI   C,TRAPB                 ;argument: the trap block addr
SWTRP%
```

The trap block itself is simply initialized to point to the trap handler routine, TRAPIT.

```
;arithmetic trap block
TRAPB:  0                               ;Flags,,Instruction image
        0                               ;Trap PC
        0                               ;Instruction effective address
        0,,TRAPIT                       ;new PC
```

With this initialization complete, when the program executes any instruction that causes an arithmetic trap, control will be passed to the trap handler at `TRAPIT`. To test our trap handler, we will cause some traps. First, we will try an unimaginative division by zero; register `A` is loaded with octal 100, and a division by zero is attempted. From our discussion of the division instructions, we know that division by zero sets the arithmetic overflow and no divide flags; it does not change the dividend.

```
        MOVEI   A,100
        IDIVI   A,0                     ;divide by zero makes an error
        CALL    PNTA                    ;now show what's in A
```

The result from this sequence is the following typeout:

```
Integer Division by Zero at PC = 500
Failing instruction: 231040,,0
The current result is 100
```

Our trap handler has identified the location of the failing instruction, the specific nature of the error, and the instruction itself. As we can see, the result in `A`, as we predicted, is the same octal 100 that we started with.

Next we perform a floating–point subtraction that produces an underflow.

```
        MOVE    A,[032400000000]        ;approximately 0.98E-31
        FSBR    A,[031777777777]        ;also, approximately 0.98E-31
        CALL    PNTA                    ;show contents of A
```

The following printout results:

```
Floating Exponent Underflow at PC = 503
Failing instruction: 154040,,661
Changing the computed result, 8.507059E37, to zero
The current result is 0
```

Notice that the incorrect result has been set to zero. Setting register `A` to zero is an effect of our trap routine.

29.1.3 Trap Handler

The identification of the error and the corrective response are functions of the trap handler. The trap handler is organized rather differently from anything that we have seen before. The trap handler itself is deceptively simple:

```
TRAPIT: CALL    TRAPNT                  ;call trap processing routine
        XJRSTF  TRAPB+.ARPFL            ;return to trapped-from code
```

We begin by calling TRAPNT, which really does all the work. We shall see why a subroutine is needed here when we examine TRAPNT and SAVACS.

29.1.3.1 Returning from the Trap

The trap routine ends with a form of the JRST instruction that is new to us. XJRSTF is a jump that is especially suited for the program counter and flags format that is used in the extended machine. Although we will not delve deeply into extended addressing until Section 30, at this time we will say that to accommodate thirty bits of program counter, a new format for storing the flags and PC has been implemented. A flags and program counter doubleword is shown in Figure 13.2

The XJRSTF instruction will restore the flags and program counter from the flags and PC doubleword found at the instruction's effective address. Note that the format needed by XJRSTF matches the data found in the first two words of the trap block. Thus, assuming that the TRAPNT routine deals with the cause of the trap, this XJRSTF instruction will make the program resume at the instruction following the one that trapped.

29.1.3.2 Co–Routine to Save Accumulators

The TRAPNT subroutine actually does whatever work is necessary in response to each trap condition. The first thing that TRAPNT does is to call the SAVACS co–routine to save the accumulators.

```
TRAPNT: CALL    SAVACS          ;save the accumulators
```

A *co–routine* differs from a subroutine in that a co–routine stores local state information so that each call to the co–routine makes it resume from the state it had when it last relinquished control. Another way of looking at co–routines is that they violate the usual hierarchy of subroutines. In this case, although TRAPNT views SAVACS as a subroutine, it is also valid to say that TRAPNT is a subroutine that is called from SAVACS.

We say that SAVACS is a co–routine because when it is called from TRAPNT, it calls TRAPNT in turn as a subroutine. When TRAPNT finishes, instead of returning to its caller, TRAPNT returns to SAVACS. Finally, SAVACS returns to TRAPNT's caller. Does this sound confusing? Well, perhaps it is somewhat too ornate, but there are some very useful effects that are obtained by using this structure. Let us examine the details of how SAVACS works.

The major function of SAVACS is to save the state of the accumulators as they appeared when the trap occurred. A second function of SAVACS is to facilitate the restoration of these accumulator values after the trap routine performs its work. We must identify an area of memory in which to save the accumulators. In this case, we simply use the stack space that is provided on the pushdown list.

SAVACS starts by using an ADJSP instruction to allocate 17 locations on the stack. We will store accumulators 0 through 16 in these locations; register 17 is assumed to be the stack pointer; we will not preserve it here.[4]

[4]We assume that no well–behaved program ever makes reference above the stack top, e.g., to 1(P), etc. If such a

```
SAVACS: ADJSP   P,17                    ;add locations for 0-16
```

The stack environment following the execution of this ADJSP instruction is depicted in Figure 29.2.

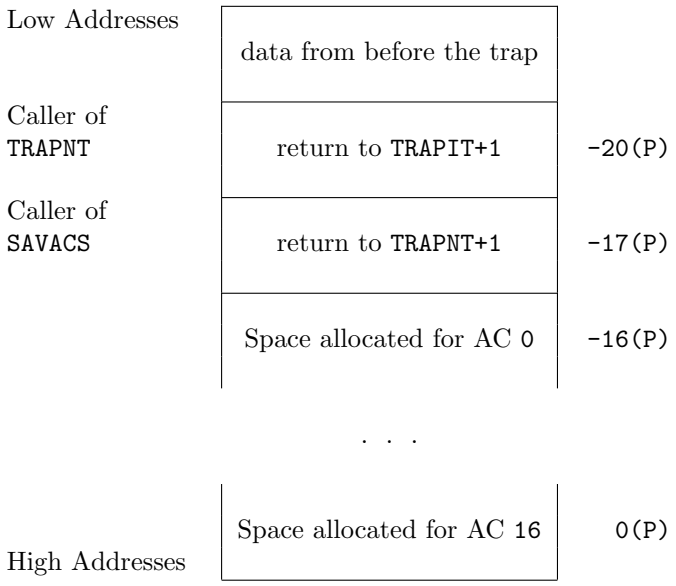


Figure 29.2: Stack Environment During the SAVACS Co–Routine

Now that we have space allocated on the stack for the accumulators, we proceed to copy them there by means of a BLT instruction. We will use register 16 for the pointer for the BLT instruction. Before loading the BLT pointer, we must save register 16 on the stack. We store 16 in 0(P); as indicated in the diagram, 0(P) is the word reserved for register 16.

Next, we load a source and destination address into 16. The source is zero; the diagram indicates that the destination is -16(P). Then, the BLT instruction is performed; the final destination address for the BLT is -1(P), where register 15 will be stored.

```
        MOVEM   16,0(P)                 ;Save AC 16 on the stack
        MOVEI   16,-16(P)               ;BLT pntr. Src=0; Dst=Stack
        BLT     16,-1(P)                ;store ACs 0 thru 15 on stack.
```

Next, we load a register that we call ACP with the address -16(P). This address is the word in which we stored register 0. ACP is the pointer to the saved accumulators. When we find it necessary to reference one of the saved accumulators, an address expression such as 6(ACP) would reference the saved register 6. We will make use of this pointer when we correct the result after an underflow.

```
        MOVEI   ACP,-16(P)              ;Address of the saved ACs
```

reference is a possibility, or in cases where it is not known what accumulator usage conventions prevail in the trapping program, we can provide a local stack for the exclusive use of the trap routine.

Now all the accumulators have been saved. The next instruction is a little peculiar. The diagram indicates that the address -17(P) contains the return to the caller of SAVACS. By executing the instruction

```
CALL    @-17(P)                 ;return to "caller"
```

we cause the execution of the calling program to resume at TRAPNT+1, because the effective address of this CALL instruction is the address that was pushed on the stack by the call to SAVACS. The strange thing is that we have put a new return address on the stack. After the execution of this CALL, the stack top contains a return address that points to SAVACS+6, the address that follows this CALL.

The TRAPNT routine has been resumed at TRAPNT+1. Now, TRAPNT can push and pop data items and return addresses as any normal program would do. The top of the stack contains a return to SAVACS+6; thus, when TRAPNT finishes executing and performs a RET, it will return there instead of to its original caller. The following code at SAVACS+6 will then be executed:

```
SKIPA                           ;non-skip return. Avoid AOS
AOS     -20(P)                  ;skip return, pass skip upward
MOVSI   16,-16(P)               ;restore saved acs to real ACs
BLT     16,16
ADJSP   P,-20                   ;discard saved ACs & 1st return
RET
```

The SKIPA instruction, whose purpose will be clarified later, will skip to the MOVSI. The MOVSI instruction simply loads register 16 with a BLT pointer that specifies the stack as the data source and register 0 as the first destination.

The BLT restores registers 0 through 16 from the data that was saved on the stack. We now perform an ADJSP to discard 20 (octal) locations from the stack. These locations contain the 17 saved accumulators plus the return address that points to TRAPNT+1. Since we've already used that address once, there is no reason to keep it any longer. The accumulators have been restored and the stack environment is as shown in Figure 29.3.

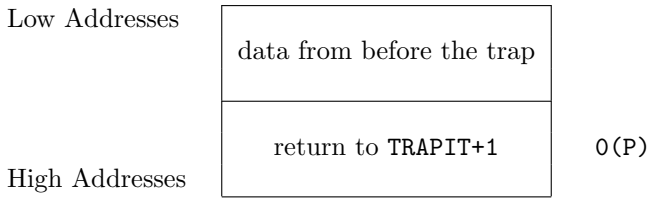


Figure 29.3: Stack Environment of SAVACS before Return

Now, when the program executes the RET, SAVACS returns to TRAPIT+1 which is the original caller of TRAPNT.

The last fragment of mystery is dispelled by the explanation of the SKIPA and AOS that are present in SAVACS. If TRAPNT increments its return address in order to effect a skip return, then the actual return will be to SAVACS+7 where that skip will be passed to the actual caller of TRAPNT by incrementing the real return address that is located in -20(P). This mechanism is provided to generalize SAVACS

so it could be used elsewhere; TRAPNT does not use a skip return.

Why isn't it better to write the simple BLTs at the entrance and exit of the TRAPNT routine? The answer is that this complex mechanism saves us from various errors that are very likely to occur in large programs. There are three plausible ways to effect the accumulator saves and restores in a subroutine. These are depicted below:

Method 1, Simple Coding

```
SUBR:   save the accumulators
        do the work
        restore the accumulators
        RET
```

This method is most natural. However, every path through this routine must be coded so that it exits via the instructions that restore the accumulators.

Method 2, Use of Another Subroutine

```
SUBR:   save the accumulators
        call another subroutine
        to do the work
        restore the accumulators
        RET
```

This is better than the first method because the other subroutine can't "forget" to leave via the path by which the ACs are restored.

Method 3, Use of Co–Routines

```
SUBR:   CALL SAVACS
        do the work
        RET
```

This is better than the second method because the AC save and restore code is in a subroutine so it can be used in many places without repeating the save and restore code from the second method.

The first method should be avoided because it is susceptible to a variety of errors. The most common error is to have some path that branches out of the subroutine without restoring the accumulators. Even experienced programmers can be tripped up by this one. It is not difficult to make such code work once, but this style of programming is not likely to survive the effects of later modifications.

The second method is better. By adding another level of subroutine call, the programmer makes certain that no return can occur that does not exit through the accumulator restore code. This method has the disadvantage that the save and restore code must be written explicitly for each routine that employs it; this can be somewhat alleviated by making the save and restore code into subroutines of their own.

The use of a co–routine such as SAVACS has the nice effect of being compact and foolproof. Although a somewhat higher initial investment of mental energy is needed to understand this method, the payoff comes when this complex mechanism is used in a very simple manner. The SAVACS routine can be called from any subroutine that wants to save all the accumulators. The calling subroutine need not know where the registers are saved, or by what mechanism. The person coding a routine that employs SAVACS need not worry about how to restore the registers on exit. By simple and reasonable behavior on the part of the caller, the registers will be restored automatically.

By now it should be clear why we employ an extra level of subroutines in which TRAPIT calls TRAPNT to perform all the work. The identification of TRAPNT as a separate routine is necessary for the proper use of SAVACS.

29.1.3.3 TRAPNT Routine

The main body of the trap handler is the TRAPNT subroutine. TRAPNT contains code to print the various trap parameters that are present in the trap block. We shall discuss those parts of TRAPNT that have some interest for us.

After saving the accumulators, TRAPNT tries to clean up the flags in order to simplify the report that it makes to the user. We start by loading the flags from the trap block into register B. An AND instruction is used to mask out all but those flags that we want to examine. We select the Arithmetic Overflow, Floating Overflow, Divide Check, and Floating Exponent Underflow flags. The PC flag name definitions that we use here come from the MACSYM file.

Since we know that certain flags imply other ones, we try to reduce these flags to the simplest set possible. For example, floating underflow sets the floating overflow and arithmetic overflow flags also. So, when we see floating underflow set, we will clear the other two flags in order to simplify the message we print. Similarly, divide check will set arithmetic overflow; when a divide check occurs in a floating–point operation, floating overflow will be set too.

The following code cleans up the flags to produce a minimal set. Also, we distinguish between integer division by zero and floating division by zero so our description of the problem will be better understood by the user. Ideally, only one of five flags (the four PC flags, plus our internal flag signaling floating divide check) will be set after this fragment executes.

```
TRAPNT: CALL    SAVACS                          ;save accumulators
        MOVE    B,TRAPB+.ARPFL                  ;get the flags
        AND     B,[PC%OVF!PC%FOV!PC%NDV!PC%FUF] ;keep important bits
        TXNE    B,PC%FUF                        ;floating underflow?
        TXZ     B,PC%OVF!PC%FOV                 ; yes - no "overflow"
        TXNE    B,PC%NDV!PC%FOV                ;divide by zero or F.Ov?
        TXZ     B,PC%OVF                        ; yes - no "integer overflow"
        TXNE    B,PC%NDV                        ;divide by zero?
        TXZN    B,PC%FOV                        ;yes, and floating?
        SKIPA                                   ;not float div check
        TXC     B,PC%NDV!FLNDIV                 ;set float no divide
```

Next, with flags set in register B, we search for appropriate messages to print. We start with a simple data structure and a search loop. The table contains flags and messages; each message whose flag corresponds to a flag set in B will be printed. The data table is

```
TRPMSG: PC%OVF![ASCIZ/Integer Overflow /]
        PC%FOV![ASCIZ/Floating Overflow /]
        PC%NDV![ASCIZ/Integer Division by Zero /]
        PC%FUF![ASCIZ/Floating Exponent Underflow /]
        FLNDIV![ASCIZ/Floating Divide by Zero (Divide Check) /]
TRPMLN==.-TRPMSG
```

The search code should be fairly obvious:

```
        MOVSI   D,-TRPMLN               ;-length of message tbl
TRPNT1: TDNN    B,TRPMSG(D)             ;is flag on for this message?
        JRST    TRPNT2                  ;no flag means no print
        HRRO    A,TRPMSG(D)             ;set LH of A to -1
        PSOUT                           ;print msg for this flag
TRPNT2: AOBJN   D,TRPNT1                ;loop thru msg table
```

After printing the reason for the trap, the program continues by printing the program counter that addresses the trapping instruction. Then the octal value of the image of the trapping instruction is printed, including the resolution of the effective address. All this information is simply present in the trap block.

After this standard information is printed, a special handler is called if the trap was caused by floating underflow. This handler will be discussed below. Finally, the `TRAPNT` routine exits, but first it clears the four interesting PC flags from the flags stored in the trap block. This is done because these flags would otherwise remain set and they would confuse the processing of the next trap. The flags are cleared by an `ANDCAM` instruction, which is highly recommended for this purpose; the bits that are set to one in register `B` are set to zero in the word at `TRAPB+.ARPFL`.

```
        MOVX    B,PC%OVF!PC%FOV!PC%NDV!PC%FUF
        ANDCAM  B,TRAPB+.ARPFL          ;clear selected flags
        RET
```

When, finally, the `RET` is executed, this routine will return to `SAVACS` where the accumulators will be restored. Then `SAVACS` will return to `TRAPIT` where the instruction stream that we trapped from will be resumed.

29.1.3.4 Handling Floating Underflow

We should expect underflows when we deal with small floating–point numbers. Sometimes these small numbers can be relatively large and still cause underflows. In a floating multiply, if both operands are smaller than about 0.38×10^{-19} then underflow will result.

This example demonstrates an underflow caused by subtraction. The two operands are nearly identical in magnitude, being close to 0.98×10^{-31}. Their actual difference is about 0.73×10^{-39}, but that difference is smaller than any representable number. The actual computed result is 0.85×10^{38}. Although in some cases of underflow it is possible to use the result in further calculations and eventually arrive at an arithmetically correct result, often it is appropriate to replace the computed result with zero.[5]

In order to replace the computed result, we must know where the instruction has put the result. In the usual case, the result will be in the accumulator specified in the image of the trapping instruction. However, in the case of an instruction such as `FMPRM`, the result will be in the memory location specified by the effective address. Also, some instructions store their result in both the effective address and in the accumulator. Still others store results in `AC` and `AC+1`.

[5] In some calculations floating underflow is just as detrimental as floating overflow. The algorithm in question should be studied carefully to determine the most correct response to an underflow. Most computer scientists specializing in numerical analysis agree that the user should be notified that arithmetic exceptions have occurred. Yet another approach is to implement *gradual underflow* in which unnormalized floating–point numbers are allowed as results when the exponent field has reached its minimum value; the PDP–10 does not provide for such gradual underflow, but with the help of further software it could be implemented.

With help from a macro that we call FOPXX, we will generate the table which, for each different instruction, tells us the location(s) of the result. We define three flags, ACF, MMF, and DDF, which we associate with each of the floating–point instructions. ACF indicates that the result appears in the accumulator; MMF appears with those instructions where the result is stored at the effective address; and DDF signifies a double–word result in AC and AC+1.

The table of instructions and where they put their results is generated by the code that appears below.

```
;Macro to help generate the table of floating-point operations
DEFINE FOPXX (OP) <       ;;generate table for floating operations
        F'OP    ACF       ;;e.g., FAD     result in AC
        F'OP'M  MMF       ;;      FADM    result in MEM
        F'OP'B  ACF!MMF   ;;      FADB    result in AC and in MEM
        F'OP'R  ACF       ;;      FADR    result in AC
        F'OP'RI ACF       ;;      FADRI   result in AC
        F'OP'RM MMF       ;;      FADRM   result in MEM
        F'OP'RB ACF!MMF   ;;      FADRB   result in AC and in MEM
        DF'OP   ACF!DDF   ;;      DFAD    result in AC and AC+1
>                         ; FOPXX

;Floating underflow table
FXUTAB: FOPXX   (AD)
        FOPXX   (SB)
        FOPXX   (MP)
        FOPXX   (DV)
        FSC     ACF
FXUTLN==.-FXUTAB               ;length of the table
```

The subroutine DOFXU is called from TRAPNT to clean up after an underflow. First, we search FXUTAB, the floating–point instruction table, for an entry that matches the opcode of the trapping instruction. If no match is found, the program terminates abruptly. When a match is found the program jumps to DOFXU2 with register D containing the index into the instruction table.

```
DOFXU:  MOVSI   D,-FXUTLN               ;Length of the table
        LDB     A,[POINT 9,TRAPB+.ARPFL,26] ;opcode of failed instr
DOFXU1: LDB     B,[POINT 9,FXUTAB(D),8] ;get opcode from table
        CAMN    A,B                     ;do we match?
        JRST    DOFXU2                  ;yes. D=index to table
        AOBJN   D,DOFXU1                ;advance to next instr
        ;print error here
```

At DOFXU2, we load A with the result flags associated with the trapping instruction. If the trapping instruction has placed a result in the accumulator, we load register B with the specific accumulator address. We bounds check register B to avoid the stack pointer in register 17. By adding the contents of ACP to the accumulator number in register B we compute the address where the contents of this accumulator were stored by SAVACS. We copy the saved accumulator to register C and zero the saved copy. When this saved copy is restored on the way out of TRAPNT and SAVACS, the real accumulator will be set to zero.

```
DOFXU2: HRRZ    A,FXUTAB(D)             ;where is the result?
        TRNN    A,ACF                   ;Is result in AC?
        JRST    DOFXU3                  ;no. skip this part
        LDB     B,[POINT 4,TRAPB+.ARPFL,30] ;AC field of instru.
        CAIL    B,17                    ;mustn't reference stack AC
        JRST    DOFXU0                  ;We think this can't happen.
        ADD     B,ACP                   ;offset to saved acs.
        MOVE    C,(B)                   ;computed result to C
        SETZM   (B)                     ;clear saved copy of AC
DOFXU3:
```

At DOFXU3 we test to see if the trapping instruction has stored a result in memory. If MMF is set then we must zero the word at the trapping instruction's effective address. The code at DOFXU6 copies the memory result to register C and zeroes it. The code before DOFXU6 tests for the special case of the effective address addressing one of the accumulators. If the effective address contained in the .AREFA word of the trap block is in the range from 0 to 16 then we zero the saved copy of the accumulator. Due to the possibility of extended addressing being used, we consider the addresses 1,,0 through 1,,16 to be accumulators also.[6]

```
DOFXU3: TRNN    A,MMF                   ;is result in memory?
        JRST    DOFXU4                  ;no.
        HRRZ    B,TRAPB+.AREFA          ;yes, get in-sect result addr
        HLRZ    D,TRAPB+.AREFA          ;get the section # of result
        CAIG    B,16                    ;skip if not an AC addr
        CAILE   D,1                     ;skip if definite AC address
        JRST    DOFXU6                  ;not an AC address
        ADD     B,ACP                   ;offset to stored acs
        MOVE    C,(B)                   ;copy result to C
        SETZM   (B)                     ;and clear result location
        JRST    DOFXU4

DOFXU6: MOVE    C,@TRAPB+.AREFA         ;computed result to C
        SETZM   @TRAPB+.AREFA           ;clear memory result
DOFXU4:
```

The case of a double–length result in an accumulator is dealt with at DOFXU4. The address of the saved copy of AC+1 is computed; that location is set to zero.

```
DOFXU4: TRNN    A,DDF                   ;is result in AC+1?
        JRST    DOFXU5                  ;no.
        LDB     B,[POINT 4,TRAPB+.ARPFL,30] ;AC field of instr again
        ADDI    B,1                     ;advance to AC+1
        CAIL    B,17                    ;mustn't reference stack AC
        JRST    DOFXU0                  ;We think this can't happen
        ADD     B,ACP                   ;offset to saved ACs
        SETZM   (B)                     ;clear saved AC+1
DOFXU5:
```

Finally, at DOFXU5, we print a message. Register C contains the result that we are about to discard. This result is copied to B and printed via the FLOUT JSYS, an operating system call that converts

[6]This code is written with the expectation that register 17 will not be involved in any arithmetic overflow.

floating–point numbers to strings. (Note: this program is incomplete in that it fails to print correctly if the data is a G–format item.)

```
DOFXU5: HRROI   A,[ASCIZ/Changing the computed result, /]
        PSOUT
        MOVEI   A,.PRIOU                ;print result as floating
        MOVE    B,C                     ;the data item to print
        MOVEI   C,0                     ;format control flags
        FLOUT                           ;convert floating to chars
        ERJMP   .+1
        HRROI   A,[ASCIZ/, to zero
/]
        PSOUT
        RET
```

We have omitted any explanation of the remainder of this program; there should be nothing present to cause the reader any special difficulty.

29.2 Interrupts

The following example displays the use of interrupts to determine the status of a program. As a user of TOPS–20, the reader should be aware of the EXEC's response to CTRL/T: when the user types CTRL/T the EXEC types out the status of the current program. Among the useful items of status information are what condition the program is in (running, waiting for input or output, etc.) and where the program is executing. The example that we give responds to CTRL/T by printing the status of the program, the octal value of the program counter, and the symbolic value of the program counter.[7]

29.2.1 Example 19 — Interrupts

Example 19 is a program module, a collection of data structures and subroutines, that can be assembled by itself and then linked with another program. The modification necessary to the main program is simply the inclusion of the declaration of INTINI as an external symbol, and a call to INTINI shortly after the stack pointer (which must be in register 17) has been initialized.

To test this module, the dictionary and sort program, our example 16, was modified by two very simple changes. First, the line containing the EXTERN pseudo–op was modified to include a new symbol, INTINI. Then, a call to the INTINI subroutine was added just following the start of the program:

```
        EXTERN  .JBSA,INTINI

        ...

START:  RESET
        MOVE    P,[IOWD PDLEN,PDLIST]   ;Initialize stack
        CALL    INTINI                  ;Initialize interrupt system
        ...
```

[7] Events have overtaken this example: symbolic typeout has been added to the EXEC's repertoire.

The pair of programs are assembled, linked together, and started by an appropriate EXECUTE command. A sample session in which the program is tested appears below. In this session, the program's response to the author's impatient use of CTRL/T is displayed.

```
@execute ex16,ex19
MACRO:  Dictio
MACRO:  INTSYM
LINK:   Loading
[LNKXCT DICTIO execution]
File name for input: IO Wait at 1143 DICTIO  GTINPF+6
old-MAIL.TXT.1 !Old generation!
File name for output: flotsam
Running at 1275 DICTIO  GETBUF+6
Running at 1255 DICTIO  GETCHR
Running at 1351 DICTIO  HASH1+3
Running at 1370 DICTIO  BLDBL2
Running at 1275 DICTIO  GETBUF+6
Running at 1130 DICTIO  ISLET
Running at 1224 DICTIO  PUTCHR+4
Running at 1341 DICTIO  NAMCMP+2
Running at 1110 DICTIO  GETWD1+3
Running at 1340 DICTIO  NAMCMP+1
Running at 1275 DICTIO  GETBUF+6
Running at 1354 DICTIO  HASH2+1
Running at 1275 DICTIO  GETBUF+6
Running at 1102 DICTIO  GETWRD+3
Running at 1506 DICTIO  NSCOMP+1
Running at 1224 DICTIO  PUTCHR+4
Running at 1250 DICTIO  FINISH+3
@
```

The text of the program module appears below; the explanation will follow.

```
        TITLE   INTSYM  CTRL/T Interrupts will print symbolic PC
        SEARCH  MONSYM,MACSYM
        EXTERN  .JBSYM
        INTERN  INTINI

A=1                                     ;usual accumulator definitions
B=2
C=3
D=4
P=17

PDLEN==100
```

```
PRCACS: BLOCK   20                      ;ACs of main process
MAINFK: 0                               ;fork handle of main process
PCLEV3: 0                               ;place for level 3 interrupt PC
LV3SAC: BLOCK   20                      ;place for level 3 interrupt ACs
LV3PDL: BLOCK   PDLEN                   ;stack for level 3 interrupt processing
INTBUF: BLOCK   20                      ;line buffer for interrupt processing
LINPTR: 0                               ;pointer to line buffer
BSYM:   0                               ;address in SYMTAB of best sym so far
BPROG:  0                               ;addr of program name for BSYM
XVAL:   0                               ;target value for SYMLOK
FKSTAT: 0                               ;fork status

CHNTAB: 3,,CHN0SV                       ;priority 3 service for channel # 0
        BLOCK   CHNTAB+^D36-.           ;fill remainder of 36 words

LEVTAB: 0                               ;level 1 PC save address
        0                               ;level 2 PC save address
        0,,PCLEV3                       ;level 3 PC save address

;Call this co-routine, via JSR, to enter level 3.
;Save the ACs, setup the private stack & co-return
ENTLV3: 0                               ;this location gets PC of caller
        MOVEM   17,LV3SAC+17            ;save ACs of interrupted process
        MOVEI   17,LV3SAC               ;in the private area for level 3
        BLT     17,LV3SAC+16
        MOVE    P,[IOWD PDLEN,LV3PDL]   ;private stack for level 3
        CALL    @ENTLV3                 ;co-return: let caller handle interrupt
        MOVSI   17,LV3SAC               ;restore ACs of the interrupted
        BLT     17,17                   ;process
        DEBRK                           ;return to interrupted process
```

```
INTINI: MOVEM   17,PRCACS+17            ;save acs of caller
        MOVEI   17,PRCACS
        BLT     17,PRCACS+16

        MOVEI   A,.FHSLF                ;This fork
        SKPIR                           ;Skip if interrupts are enabled
        JRST    INTIN1                  ;no.  We are ok. Proceed
        HRROI   A,[ASCIZ/Interrupts are already enabled at INTINI.
INTINI won't change anything.
/]
        PSOUT
        MOVE    A,PRCACS+A
        MOVE    17,PRCACS+17
        RET

INTIN1: MOVX    A,CR%MAP!CR%CAP!CR%ACS  ;create another fork for caller
        MOVEI   B,PRCACS                ;setup return of ACS
        CFORK
        ERJMP   ERRCOM
        MOVEM   A,MAINFK                ;save fork handle to new process

;tell system the location of the channel (dispatch) and level (PC save) tables
        MOVEI   A,.FHSLF                ;setup this fork for interrupts
        MOVE    B,[LEVTAB,,CHNTAB]
        SIR                             ;set interrupt locations
        MOVEI   A,.FHSLF
        EIR                             ;enable interrupts
        MOVEI   A,.FHSLF
        MOVX    B,1B0                   ;activate channel 0
        AIC
        MOVE    A,[.TICCT,,0]           ;assign CTRL/T to channel 0
        ATI

;now, start the new process.  This looks to it like a return from the INTINI subr
        HRRZ    A,MAINFK                ;Start new fork
        MOVEI   B,CPOPJ                 ;at CPOPJ
CONTIN: SFORK                           ;(here to continue a halted fork)
        ERJMP   ERRCOM

;now, all we do is wait for the new process to terminate, at which point, we
;terminate also.  Meanwhile, this process is interruptable.
        MOVE    A,MAINFK                ;wait for process to terminate
        WFORK
        HALTF
;In case of a CONTINUE command, we will continue the inferior and wait
;again for it to terminate.  Note this particular use of SFORK does NOT
;work prior to TOPS-20 Release 4.  See text.
        HRRZ    A,MAINFK                ;in case of CONTINUE command,
        TXO     A,SF%CON                ;Continue the inferior
        JRST    CONTIN                  ;set CONTINUE flag in A, and do SFORK
```

```
ERRCOM: HRROI   A,[ASCIZ/Error: /]
        ESOUT
        MOVEI   A,.PRIOU
        MOVX    B,<.FHSLF,,-1>
        MOVEI   C,0
        ERSTR
        ERJMP   .+1
        ERJMP   .+1
        HRROI   A,[ASCIZ/
/]
        PSOUT
        HALTF
        JRST    .-1

        [ASCIZ/Unknown Status/]
FKSTM:  [ASCIZ/Running/]
        [ASCIZ/IO Wait/]
        [ASCIZ/Halted/]
        [ASCIZ/Forced Termination/]
        [ASCIZ/Fork Wait/]
        [ASCIZ/Sleep/]
        [ASCIZ/Trap Wait/]
        [ASCIZ/Address Break/]
FKSTML==.-FKSTM
```

```
CHNOSV: JSR     ENTLV3                  ;Call Enter Level 3
        HRRZ    A,MAINFK                ;fork handle of inferior
        RFSTS                           ;get the status of this fork
        MOVEM   A,FKSTAT                ;save status of the fork
        HRRZM   B,XVAL                  ;and PC of the fork (arg to SYMLOK)
        MOVE    A,[POINT 7,INTBUF]      ;byte pointer to the line buffer
        LDB     B,[POINT 17,FKSTAT,17]  ;get the fork status indication
        CAIL    B,FKSTML                ;is this a known status?
        MOVEI   B,-1                    ;no. give the unknown status message
        HRRO    B,FKSTM(B)              ;message pointer to B
        MOVEI   C,0                     ;stop on null
        SOUT                            ;copy message to line buffer
        ERJMP   .+1
        HRROI   B,[ASCIZ/ at /]
        SOUT                            ;copy more to line buffer
        ERJMP   .+1
        MOVE    B,XVAL                  ;get the PC value
        MOVEI   C,10                    ;print in Octal
        NOUT                            ;copy octal of PC to line buffer
        ERJMP   .+1
        MOVEI   B,11                    ;add a tab character
        IDPB    B,A
        MOVEM   A,LINPTR                ;save pointer to line buffer
        CALL    SYMLOK                  ;print the symbolic for the PC
        MOVEI   A,15                    ;add CR, LF, null to line buffer
        IDPB    A,LINPTR
        MOVEI   A,12
        IDPB    A,LINPTR
        MOVEI   A,0
        IDPB    A,LINPTR
        HRROI   A,INTBUF                ;print the buffer
        PSOUT
        RET
```

```
;Call with XVAL = Value to seek
SYMLOK: SETZB   C,BPROG                 ;no best program name yet
        SETZM   BSYM                    ;no best symbol
        MOVE    D,.JBSYM
        HLRO    A,D
        SUB     D,A                     ;-count,,ending address +1
SYMLK1: LDB     A,[POINT 4,-2(D),3]     ;symbol type
        CAILE   A,2                     ;0=prog name.  1=global, 2=local
        JRST    SYMLK2                  ;none of the kind we want
        JUMPE   A,SYML1Z                ;this is a program name
        MOVE    A,-1(D)                 ;this is the value of the symbol
        CAMN    A,XVAL                  ;is this an exact match?
        JRST    SYML2A                  ;yes.  select it. Escape from loop.
        CAML    A,XVAL                  ;is this smaller than value sought?
        JRST    SYMLK2                  ;no. too large
        SKIPN   B,BSYM                  ;get best one so far.
        JRST    SYML1A                  ;no previous best. remember this one.
        CAMG    A,-1(B)                 ;compare to previous best
        JRST    SYMLK2                  ;previous best was better
SYML1A: MOVEM   D,BSYM                  ;current symbol is best match so far.
        MOVEM   C,BPROG                 ;save it and look for a better one
        JRST    SYMLK2

SYML1Z: MOVE    C,D                     ;save pointer to current program name
SYMLK2: ADD     D,[2000000-2]           ;add 2 in the left, sub 2 in the right
        JUMPL   D,SYMLK1                ;loop unless control count is exhausted
        JRST    SYMLK3
```

```
SYML2A: MOVEM   D,BSYM                  ;Here for an exact match.
        MOVEM   C,BPROG                 ;Save values of C and D
SYMLK3: SKIPN   D,BSYM                  ;did we find anything helpful?
        RET                             ;no
        MOVE    A,XVAL                  ;desired value
        SUB     A,-1(D)                 ;less symbol's value = offset
        CAIL    A,100                   ;is offset small enough?
        RET                             ;no. not a good enough match
        MOVE    D,BPROG                 ;pointer to the program name
        MOVE    A,-2(D)                 ;get the program name
        CALL    R50DOP                  ;copy program name to output line
        MOVEI   A,11                    ;add a tab to the line
        IDPB    A,LINPTR
        MOVE    D,BSYM                  ;get the symbol's address
        MOVE    A,-2(D)                 ;symbol name
        CALL    R50DOP                  ;print symbol name
        MOVE    B,XVAL                  ;get desired value
        SUB     B,-1(D)                 ;less this symbol's value
        JUMPE   B,CPOPJ                 ;if no offset, don't print "+0"
        MOVEI   A,"+"                   ;add + to the output line
        IDPB    A,LINPTR
        MOVE    A,LINPTR                ;and copy numeric offset to output
        MOVEI   C,10
        NOUT
        ERJMP   .+1
        MOVEM   A,LINPTR                ;store pointer to current line.
CPOPJ:  RET

;Convert a 32-bit quantity in A from RADIX50 to ASCII
R50DOP: TLZ     A,740000                ;clear any symbol flags
R50DP1: IDIVI   A,50                    ;divide by 50
        PUSH    P,B                     ;save remainder, a character
        SKIPE   A                       ;if A is now zero, unwind the stack
        CALL    R50DP1                  ;call self again, reduce A
        POP     P,A                     ;pop one Radix50-coded character
        ADJBP   A,[POINT 7,R50TAB,6]    ;convert Radix50 code to byte pointer
        LDB     A,A                     ;get ASCII decode of Radix50 character
        IDPB    A,LINPTR                ;Store ASCII in the line buffer
        RET

R50TAB: ASCII   / 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ.$%/

        END                             ;a subroutine file has no start address
```

With relative ease we could write a subroutine that readies itself for interrupts from CTRL/T and reports the value of the program counter at the time of the interrupt. However, we can not report on the status of the program itself, because in order to execute the interrupt code, the program necessarily is running. In order to get a more useful status report, we place the "real" program in one process, and take the CTRL/T interrupts in a second process. At each interrupt the second process will report the program counter and status of the first process.

29.2.2 Interrupt Initialization

The INTINI routine must accomplish three things. First, it verifies that the interrupt system is not active before changing it. Second, it must create an inferior process in which to continue running the program that called it. Finally, it must enable itself to receive interrupts. The latter two functions are somewhat intermingled in the code of INTINI; in the explanation we will try to disentangle these functions and describe why they are intertwined.

In order to place the calling program in an inferior fork, we use the CFORK JSYS to create another fork that has the same pages and capabilities as this fork. When the INTINI subroutine is entered, the caller's accumulators are saved in the block called PRCACS; later, we will tell CFORK to initialize the accumulators of the new process from PRCACS.

```
INTINI: MOVEM   17,PRCACS+17            ;save acs of caller
        MOVEI   17,PRCACS
        BLT     17,PRCACS+16
```

Next, the program must verify that the interrupt system is inactive. Although it would be possible to add our single–purpose function to an already active interrupt system, the code to do so is too complex for this example. Instead, we will simply test the state of the interrupt system to determine that it is presently idle. The SKPIR JSYS will skip if the interrupt system is enabled for the specified process. If it skips, the program prints an apology, restores the accumulators and returns to the caller without performing its normal function.

```
        MOVEI   A,.FHSLF                ;This fork
        SKPIR                           ;Skip if interrupts enabled
        JRST    INTIN1                  ;no.  We are ok. Proceed
        HRROI   A,[ASCIZ/Interrupts are already enabled at INTINI.
INTINI won't change anything.
/]
        PSOUT
        MOVE    A,PRCACS+A
        MOVE    17,PRCACS+17
        RET
```

The program arrives at INTIN1 knowing that the interrupt system is idle. It is now ready to create an inferior fork and give it the accumulators that were stored in PRCACS:

```
INTIN1: MOVX    A,CR%MAP!CR%CAP!CR%ACS ;create another fork for caller
        MOVEI   B,PRCACS               ;setup return of ACS
        CFORK
        ERJMP   ERRCOM
        MOVEM   A,MAINFK               ;save handle to new process
```

We do not let CFORK start this fork because we are not yet ready to handle interrupts to report its status. Also, we should not enable interrupts until the new fork has been created, because the fork handle of the new fork is needed by the interrupt service routine.

After creating the new fork, we enable the interrupt system, as we shall further describe below. Once the interrupt system is ready, we can start the new fork by the SFORK JSYS. We start the new fork at the label CPOPJ; the RET at CPOPJ will have the effect of returning from the INTINI subroutine, but the program that called INTINI will now be running in the newly created process.

```
        HRRZ    A,MAINFK                ;Start new fork
        MOVEI   B,CPOPJ                 ;at CPOPJ
CONTIN: SFORK                           ;(continue a halted fork here)
        ERJMP   ERRCOM
```

Now, INTINI is still running in the original process. It can do whatever it wants. In this case, we will demonstrate commendable restraint by not doing anything more than waiting for the newly created process to terminate.

```
        MOVE    A,MAINFK                ;wait for process to terminate
        WFORK
```

When the newly created process terminates, this process terminates also. As befits an intelligent superior process, this process is receptive to the EXEC's CONTINUE command. If this program is continued, it will pass that continuation down to its inferior process, by executing the SFORK JSYS at the label CONTIN. For continuation of the inferior, the argument to SFORK is the fork handle in the right half of register 1 and a flag called SF%CON in the left half of 1. Note that this use of SFORK is not supported prior to release 4 of TOPS–20.[8]

```
        HALTF
        HRRZ    A,MAINFK                ;in case of CONTINUE command,
        TXO     A,SF%CON                ;Continue the inferior
        JRST    CONTIN                  ;set CONTINUE flag in A,
                                        ;  and do SFORK
```

Pending the termination of the inferior process, this process remains quiescent. However, after creating the new process by using CFORK, this program enables the interrupt system to provide the original process with interrupts whenever a CTRL/T is typed. We will now examine the code that enables the interrupt system.

Each *interrupt channel* is a software entity that is associated with an event. Some channels are permanently associated with particular events such as stack overflow, file data errors, reference to non–existent pages, etc. The other channels can be assigned by the programmer to various events such as terminal interrupts, IPCF interrupts, and program initiated interrupts. There are thirty–six interrupt channels; each channel corresponds to a bit in one computer word.

The user program provides two data structures that the interrupt system refers to. These are called the *channel table* and the *level table*. By tradition, these are called CHNTAB and LEVTAB, respectively.

The channel table is thirty–six words long; it specifies the priority level and interrupt handler address for each of the interrupt channels. The priority for each channel is specified in the left half of the channel table entry; the right half is the address of the interrupt handler. Four levels of priorities are recognized. Priority level 1 has the highest priority; level 3 has the least priority; level 2 is intermediate. Level 0 signifies that the given interrupt channel is not accepting interrupts. When an event occurs that is assigned to a channel of higher priority than the priority of the routine currently in progress, the routine in progress will be interrupted while the service routine for the higher priority event is run.

In this example, only interrupt channel zero is used. A priority level three handler at CHNOSV is specified. Space for the other channel table entries is reserved and set to zero.

[8] In earlier releases of TOPS–20, an inferior is continued by starting it using the value of the program counter returned by the RFSTS JSYS, as we shall describe shortly.

```
CHNTAB: 3,,CHNOSV                       ;priority 3 for channel # 0
        BLOCK   CHNTAB+^D36-.           ;fill remainder of 36 words
```

The level table has three words. Each word specifies the address at which the program counter of the interrupted process will be stored when an interrupt of a given priority level is started. The first word in the table corresponds to level 1, etc. Different addresses for saving the PC are needed for each different priority level, because level 1 may interrupt level 2 which had interrupted level 3. Each level needs a private PC storage location and a private pushdown list. Since our example uses only one level, only one entry in LEVTAB is needed.

```
LEVTAB: 0                               ;level 1 PC save address
        0                               ;level 2 PC save address
        0,,PCLEV3                       ;level 3 PC save address
```

We bring the addresses of the channel table and the level table to the attention of the operating system by the SIR JSYS.[9] This system call takes a process handle in register 1 and the two addresses in register 2 as shown:

```
        MOVEI   A,.FHSLF                ;setup fork for interrupts
        MOVE    B,[LEVTAB,,CHNTAB]
        SIR                             ;set interrupt locations
```

Now that the basic data structures are in place, we can enable interrupts. This is done quite simply by the EIR call. Until the interrupt system is enabled and the table addresses established, there are no interrupts.

```
        MOVEI   A,.FHSLF
        EIR                             ;enable interrupts
```

Next, the specific channel that we want to use must be activated. The interrupt system allows channels to be selectively activated and deactivated. While a channel is deactivated, no interrupts for that channel will be taken. We specify channel 0 by setting bit 0 in register 2:

```
        MOVEI   A,.FHSLF
        MOVX    B,1B0                   ;activate channel 0
        AIC
```

Finally, we must assign the terminal interrupt for CTRL/T to our interrupt channel 0. This is done by the ATI JSYS. The specific terminal interrupt code for CTRL/T, .TICCT, is placed in the left half of register 1 and the channel number is placed in the right half:

```
        MOVE    A,[.TICCT,,0]           ;assign CTRL/T to channel 0
        ATI
```

[9] As an aside, the RIR JSYS can be used to ask the operating system to report where the channel and level tables were defined.

The effect of these calls is to prepare the process to receive an interrupt each time a CTRL/T character is typed on the terminal.

You might wonder what happens to the EXEC's enabling of CTRL/T interrupts. The answer is that when multiple processes have enabled the same terminal interrupt code, the lowest inferior process receives the interrupt. This process intercepts the interrupt before it becomes visible to the EXEC. This behavior is broadly consistent throughout TOPS–20: exception conditions are handled at the lowest level that is enabled to do so; unhandled exceptions are passed upward in the hierarchy of processes until a process is found that will handle the exception. (The EXEC is supposed to be prepared to handle "anything". If the EXEC fails to handle an exception, the exception is referred to an unusual part of TOPS–20 called the mini–exec. If the mini–exec is bothered by an unhandled exceptiom, it logs out the job in which the exception occurs.)

As an aside, to obtain interrupts on other kinds of events, such as the arrival of an IPCF packet, look for JSYS calls associated with the specific facility. For IPCF, a function of the MUTIL JSYS can be used to associate an interrupt channel with a PID.

29.2.3 Interrupt Service

Now, when the user types CTRL/T, the program will get an interrupt on channel 0. Since channel 0 is active, and the interrupt system is enabled, the system will look in CHNTAB for the priority and dispatch address for channel 0. The level number in CHNTAB+0 is non–zero and legal; the current process PC is stored in the word specified by LEVTAB+2. Then the operating system starts running this process at the address given in the right half of CHNTAB+0, CHNOSV, the service routine for channel 0.

Note that a certain amount of care is needed when setting up the interrupt system. For example, an interrupt may occur immediately following the ATI JSYS, before starting the inferior fork at CONTIN. The reader should verify that even in this peculiar case, the interrupt service routine will behave reasonably.

CHNOSV starts by calling the ENTLV3 co–routine that saves the accumulators and sets up a stack. ENTLV3 is actually a little simpler than SAVACS in the previous example.

```
CHNOSV: JSR     ENTLV3                  ;Enter Level 3
```

This is our first example of the JSR instruction. We tend to avoid using JSR because it stores the return PC in the instruction stream, effectively modifying the code. Alas, here there is no other reasonable choice. There is no local stack set up, so we can not use PUSHJ. We must preserve the accumulators, so JSP can not be used. So we settle for one *impure* (i.e., self–modified) routine.[10]

The JSR instruction stores a return PC at ENTLV3 and jumps to ENTLV3+1. The first three instructions are a typical accumulator save sequence. Since the three priority levels of interrupts can run independently and be active simultaneously, each different level needs it own local PC save area, AC save area, and local stack. This routine is aimed at the level three AC save area and stack.

After saving the accumulators, a stack pointer is set up, and the caller of ENTLV3 is resumed by a co–routine return (could we call this a *co–return*?). When eventually a RET instruction is executed, control returns to ENTLV3 where the original accumulators are restored and the interrupt is dismissed via the DEBRK JSYS.

[10]Because routines called by JSR are impure, programs often segregate the impure return PC word by placing it and a JRST in the program's data area. The JRST jumps back to pure code.

```
ENTLV3: 0                               ;this loc gets PC of caller
        MOVEM   17,LV3SAC+17            ;save ACs of interrupted process
        MOVEI   17,LV3SAC               ;in the private area for level 3
        BLT     17,LV3SAC+16
        MOVE    P,[IOWD PDLEN,LV3PDL]   ;private stack for level 3
        CALL    @ENTLV3                 ;co-return: let caller handle interrupt
        MOVSI   17,LV3SAC               ;restore ACs of the interrupted
        BLT     17,17                   ;process
        DEBRK                           ;return to interrupted process
```

While an interrupt service routine is executing, the interrupt process is said to be *in progress* at the level specified in the channel table, e.g., "in progress at level 3". While in progress at a given level, no further interrupts at that level or at any level of lower priority can be started. Execution of the DEBRK JSYS *dismisses* the interrupt in progress by restoring the program counter and interrupt priority level of the process that was interrupted. Dismissing the interrupt makes the PSI system receptive to further interrupts at the level from which the DEBRK was executed. While in progress at one level, if another interrupt is requested at a level of equal or lower priority, that request will be delayed until the level in progress is dismissed from.

After the accumulators have been saved, CHNOSV executes the RFSTS JSYS to read the status of the inferior fork in which the main program is running. The short form of RFSTS requires a fork handle in the right half of register 1, with a zero in the left half. RFSTS returns the fork status word in register 1 and the value of the fork's program counter in register 2. The fork status and program counter are stored in memory locations.

```
CHNOSV: JSR     ENTLV3                  ;Call Enter Level 3
        HRRZ    A,MAINFK                ;fork handle of inferior
        RFSTS                           ;get the status of this fork
        MOVEM   A,FKSTAT                ;save status of the fork
        HRRZM   B,XVAL                  ;and PC of the fork (arg to SYMLOK)
```

Bits 1:17 of the fork status are the state of the process. These codes, which are further explained along with the RFSTS JSYS in [MCRM], are small numbers. Our table, FKSTM, has messages corresponding to each of the currently defined status codes. The appropriate message is copied from the table to the line buffer. Various other messages are added to the line buffer, including the octal value of the inferior fork's program counter.

The SYMLOK subroutine is called to translate the numeric program counter to a symbolic expression, as we will discuss shortly. After SYMLOK, a carriage return, line feed, and null are added to the line buffer, and the line is sent to the terminal via PSOUT. When the RET is executed, CHNOSV returns to the ENTLV3 co–routine, which restores the accumulators and dismisses the interrupt via the DEBRK JSYS.

29.2.4 Symbol Table Format

By now, the reader should be aware from personal experience with DDT that symbols defined in the assembly language program are available when the program is running. DDT uses this information to display addresses in symbolic form. We will use the same information to convert the numeric value of the program counter into a symbolic name or expression. Before we can describe how to convert numeric values to their symbolic form, we must describe the format of the symbol table.

Along with the binary values of the assembled code, the REL file output by MACRO contains the

names and values of symbols. The symbols are collected by LINK and placed into a *symbol table* in the memory space of the program. When LINK finishes loading the various programs into memory, it stores a pointer to the complete symbol table in the job data area location known as `.JBSYM`.

`.JBSYM` contains a pointer in `AOBJN` format: a negative word count in the left halfword, and the lowest address in the right halfword. As we shall see, the structure of the symbol table dictates that it be scanned from its highest address to its lowest. Thus, we must use some instruction other than `AOBJN` to control our scanning loop; we will think of some clever way to do it.

Each entry in the symbol table consists of a pair of words. The first word contains the name of the symbol in Radix50 notation, as we will describe below, and four flag bits to identify what type of symbol it is. The second word contains the value of the symbol. A typical entry is shown in Figure 29.4.

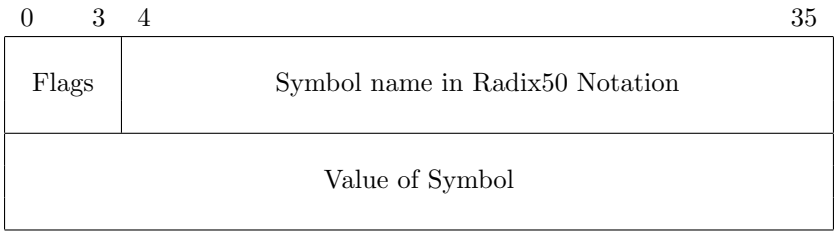


Figure 29.4: Symbol Table Entry

In the symbol table, symbols are grouped by program module. Each separately assembled module has its own region in the symbol table. This grouping is required because the same local symbol name may appear in different modules with different meanings.

Figure 29.5 shows the symbol table corresponding to our present example. The dictionary program and this subroutine have been loaded into memory. The loader builds the symbol table starting at the highest address and working down. Therefore, the first program module loaded will appear at the high end of the symbol table. In this example, we have loaded two programs, but four modules appear. The two modules `JOBDAT` and `PAT..` are provided automatically by LINK. `JOBDAT` contains the global definitions of the TOPS–10 job data area locations. Although many of these locations are unused by TOPS–20, these definitions are handy for symbols such as `.JBSA` and `.JBSYM` that we sometimes have occasion to use.

The `PAT..` module is the last module that is loaded. This module contains a block of `100` (octal) words labeled with the global name `PAT...` Usually, `PAT..` is nestled in a gap between the end of all the loaded programs and the beginning of the symbol table. There are two reasons why this space is provided. First, it provides a patch space. While debugging via DDT, programmers may wish to add more code and test it. The patch space provided here is some room in which such fragments can be added. The second reason for providing a buffer zone between the program and the symbol table is that when a new symbol is defined in DDT the symbol table is made larger. The symbol table grows towards lower addresses, and the patch space provides room for some modest expansion.

Our program from example 16, titled "Dictionary and Sort", is the first module loaded after the job data area definitions. This program module is identified by the name `DICTIO`, the first six letters from the program title. Following that, LINK loads our subroutine module, `INTSYM`.

Figure 29.5 shows the detailed structure of the `INTSYM` program module in the symbol table. The entry for the program name contains zero for the four bits of symbol flags. A zero in this field

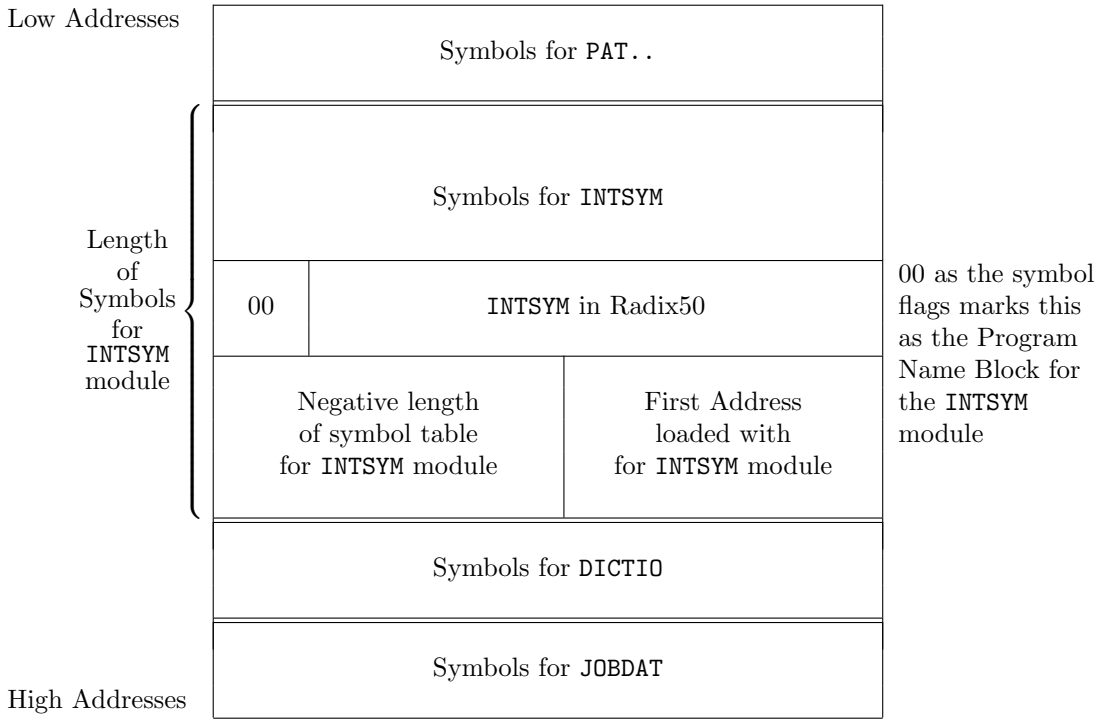


Figure 29.5: Symbol Table Structure

signifies that this entry is a program name block. The program name block is the first symbol table entry made for a program module, and so it goes into the highest symbol table address available to that program module. The second word associated with a program name contains two halfwords. The left half is the negative of the number of words used by this module in the symbol table. The right half contains the address of the first memory location that was loaded for this module; this value is also known as the *relocation constant* for this module. The relocation constant is added to each relocatable quantity in this module as it is loaded into memory. In the case of the module `INTSYM`, this address specifies where `PRCACS` is loaded, as `PRCACS` labels the first memory space used by this module.

29.2.4.1 Radix50 Notation for Symbol Names

Symbol names are encoded using a special notation called *Radix50* (the actual radix used is decimal 40 which is octal 50). Radix50 notation was invented when it was noticed that thirty–two bits can represent $2^{32} = 4294967296$ different values and that there are $40^6 = 4096000000$ different six–letter symbol names possible in an alphabet containing forty characters. Therefore, if a suitable representation is used, thirty–two bits are sufficient for all six–letter names chosen from an alphabet of forty characters. Radix50 is a representation in which we store a symbol name in just thirty–two bits, and where we use the remaining four bits for flags.

In Radix50, the forty characters are the twenty–six letters, ten digits, and four special characters: percent sign, dollar sign, period, and blank. These characters, except for blank, are precisely the characters that can be used in symbol names. Encoding a symbol name into Radix50 is very similar to converting a digit string to a number. Given the symbol composed of characters $C_1C_2C_3C_4C_5C_6$ the translation to Radix50 is given by the expression $((((T_1 \times 40 + T_2) \times 40 + T_3) \times 40 + T_4) \times 40 + T_5) \times 40 + T_6$, where T_n represents the Radix50 translation of the character C_n, and 40 is decimal.

29.2.4.2 Meaning of Symbol Flags

Although sixteen combinations of the four flag bits are possible, only five or six are commonly used in the symbol table. By convention, these flag bits are always written as two octal digits, left–adjusted in a six–bit field. Thus, they are written as multiples of four.

`00` Program Name definition. The left half of the second word contains the negative number of symbol table words used by this definition; the right half is the value of the relocation constant that was used when loading this program module.

`04` Global Symbol definition. The second word is the value of the symbol. Global definitions are accessible to all the modules that are loaded together. The name of each global symbol must be unique within the set of modules loaded; if two globals have the same name, they must have the same value.

`10` Local Symbol definition. The second word is the value of the symbol. Different modules may use the same local names to have different meanings; there is no conflict because local names in one module are not visible to other modules.

`14` Block Name. This symbol type is used by block–structured languages such as Algol, SAIL, and FAIL. In a block–structured language, symbols in a block are visible inside that block, and inside all sub–blocks contained within that block. Outside a block, that block's symbols are not visible.

`44` Suppressed Global Definition. This is similar to symbol type `04` but the global symbol is marked so that DDT will not print this symbolic name.

`50` Suppressed Local Definition. This is similar to symbol type `10` but the symbol is marked so that DDT will not print this symbolic name.

29.2.4.3 RADIX50 Pseudo–Operator

Although we have no need for it in this program, we wish to mention a pseudo–op that is related to the Radix50 notation and symbol table format. MACRO provides the `RADIX50` pseudo–op to convert a name to a Radix50 value. The `RADIX50` pseudo–op has two arguments. The first argument is the flag value, a multiple of four in the range from 0 to 74 (octal); MACRO divides this value by four and places the quotient in bits 0:3 of the result word. The second argument is a name containing legal Radix50 characters. That name is converted to Radix50 and placed in bits 4:35 of the result word.

29.2.5 Symbol Table Lookup

Our conversion of a numeric value of the program counter to a symbolic expression is accomplished by the `SYMLOK` subroutine. When `SYMLOK` is entered, the location `XVAL` contains the number that we want to convert. In the conversion process we want to locate the best match for the number given. The best match is the symbol with the highest value that is less than or equal to the specified value. Then, we will print the best symbol name and an additive offset, if necessary, to reflect the desired value. Besides printing the symbol name, we will print the name of the module in which the symbol was found.

We begin by zeroing three quantities, `C`, `BPROG`, and `BSYM`. In general, register `C` will contain the symbol table address of the most recently seen program name entry. `BSYM` is the address of the symbol that is closest to the desired value, so far. When a new value is stored in `BSYM`, the current value of `C` is stored in `BPROG`. The pair `BPROG` and `BSYM` contain the address of the program name and the address of the symbol name that are the closest match seen thus far in the symbol search.

```
SYMLOK: SETZB   C,BPROG                 ;no best program name yet.
        SETZM   BSYM                    ;no best symbol
```

Next, we take the symbol table pointer from `.JBSYM` and convert it from

```
-count,,lowest address in the symbol table
```

to

```
-count,,highest address in the symbol table + 1
```

This conversion is accomplished simply by subtracting the negative count from the right side of the original pointer. The resulting right half, equivalent to the sum of the symbol table origin plus the positive count, represents the first address beyond the end of the symbol table. This pointer format resides in register `D`. Observe that when the right half of `D` points to the word just beyond the high end of the symbol table, the address expression `-1(D)` points to the last value–word in the symbol

table, and the address expression -2(D) points to the last name in the table. Register D will remain askew throughout this routine, and we shall customarily use the offsets -1 and -2 in conjunction with indexing by D.

```
        MOVE    D,.JBSYM                ;-count,,first address
        HLRO    A,D                     ;whole word copy of -count.
        SUB     D,A                     ;-count,,ending address +1
```

The main search loop is the region between SYMLK1 and SYMLK2. At SYMLK1 we commence by extracting the flag field from a symbol table entry. We are interested only in types 00, 04, and 10, which correspond to flag values 0, 1, and 2. If the flag field is larger than 2, the program jumps to SYMLK2 where we advance to the next symbol. If the symbol type is 0, the symbol is a program name. The program jumps to SYML1Z.

```
SYMLK1: LDB     A,[POINT 4,-2(D),3]     ;symbol type
        CAILE   A,2                     ;0=prog name 1=global 2=local
        JRST    SYMLK2                  ;none of the kind we want
        JUMPE   A,SYML1Z                ;this is a program name
```

When a program name entry is seen, we copy the current symbol table pointer, the contents of D, to C. Then we advance to the next symbol by executing the code at SYMLK2.

In this search, we want to "advance" through the symbol table by subtracting 2 from the address in the right half of D. Also, we have a control count in the left half of D, to which we would like to add 2. We can accomplish the addition of two in the left half of D and the subtraction of two from the right half by adding the quantity 2000000-2 to D. If the control count stays negative, we have not yet finished the examination of all the symbols.

```
SYML1Z: MOVE    C,D                     ;save pointer to current program name
SYMLK2: ADD     D,[2000000-2]           ;add 2 in the left, sub 2 in the right
        JUMPL   D,SYMLK1
```

When we find a symbol that is either a global or a local, we compare the symbol's value with the target value. If the current symbol is the same as the target, we have an exact match. The program jumps to SYML2A which takes us out of the loop and saves the current values of D and C.

If the given symbol's value is larger than the target, the program goes to SYMLK2, to get the next symbol.

When the first symbol is found that is smaller than the target value, BSYM will be zero; the program jumps to SYML1A where this symbol will be remembered as the best match so far. When a subsequent symbol is found that is smaller than the target value, the non–zero value of BSYM addresses the previous best match symbol entry. That symbol's value is compared to the current symbol's value. If the current symbol's value is smaller, we discard it by jumping to SYMLK2; otherwise, we keep it instead of the previous value, by executing the code at SYML1A.

At SYML1A the program copies C to BPROG and D to BSYM, the program name and symbol address of the best match so far.

```
        MOVE    A,-1(D)             ;this is the value of the symbol
        CAMN    A,XVAL              ;is this an exact match?
        JRST    SYML2A              ;yes.  select it.  Leave loop.
        CAML    A,XVAL              ;is this smaller than value sought?
        JRST    SYMLK2              ;no. too large
        SKIPN   B,BSYM              ;get best one so far.
        JRST    SYML1A              ;no previous best. remember this one.
        CAMG    A,-1(B)             ;compare to previous best
        JRST    SYMLK2              ;previous best was better
SYML1A: MOVEM   D,BSYM              ;current symbol is best match so far.
        MOVEM   C,BPROG             ;save it and look for a better one
        JRST    SYMLK2
```

Eventually, either an exact match is found, by which the program escapes this loop by jumping to `SYML2A`, or the count of symbols runs out. In either case the program will wind up at `SYMLK3`. Presumably `BSYM` contains the address of the best symbol match that was found. In case none were found, this subroutine exits. The best match is tested to make sure that the symbol found is within a reasonable distance of the desired value. In this case, "reasonable" is defined as octal 100. If the best–match symbol isn't close enough, this routine exits.

```
SYMLK3: SKIPN   D,BSYM              ;did we find anything helpful?
        RET                         ;no
        MOVE    A,XVAL              ;desired value
        SUB     A,-1(D)             ;less symbol's value = offset
        CAIL    A,100               ;is offset small enough?
        RET                         ;no. not a good enough match
```

Now, `BPROG` contains the address of the program name corresponding to the symbol addressed by `BSYM`. We get the program name into register `A` and call `R50DOP` to convert the Radix50 program name to an ASCII character string. After copying the program name to the output line, this program adds a tab character and then copies the symbol name to the output line. The symbol name is converted by the same routine.

```
        MOVE    D,BPROG             ;pointer to the program name
        MOVE    A,-2(D)             ;get the program name
        CALL    R50DOP              ;copy prog name to output line
        MOVEI   A,11                ;add a tab to the line
        IDPB    A,LINPTR
        MOVE    D,BSYM              ;get the symbol's address
        MOVE    A,-2(D)             ;symbol name
        CALL    R50DOP              ;print symbol name
```

Finally, the additive offset is calculated. If the offset is zero, nothing more is sent to the output line. For a non–zero offset, the character `+` is placed in the output and the value of the offset is converted to text and placed in that line.

29.2.6 Printing Radix50 Values

Decoding a symbol name from Radix50 is accomplished by the recursive routine R50DOP. This routine is structurally similar to the decimal print routine; it successively divides the given encoding by octal 50, stacking the remainders. After all the remainders have been computed, they are popped and converted to ASCII characters. The conversion technique employed in R50DOP uses the ADJBP instruction; the Radix50 value is used to adjust a byte pointer to make a pointer to the appropriate ASCII character. When R50DOP is first entered, it clears bits 0:3 of register A. These bits are the symbol type flags; they are not wanted in this conversion procedure.

```
R50DOP: TLZ     A,740000                ;clear any symbol flags
R50DP1: IDIVI   A,50                    ;divide by 50
        PUSH    P,B                     ;save remainder, a character
        SKIPE   A                       ;if A is zero, unwind stack
        CALL    R50DP1                  ;call self again, reduce A
        POP     P,A                     ;pop one Radix50-coded character
        ADJBP   A,[POINT 7,R50TAB,6]    ;convert Radix50 code to byte pointer
        LDB     A,A                     ;get ASCII decode of Radix50 character
        IDPB    A,LINPTR                ;Store ASCII in the line buffer
        RET

R50TAB: ASCII   / 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ.$%/
```

Chapter 30

Extended Addressing

Thus far, we have considered only the traditional PDP–10 architecture in which programs are limited to 18–bit addresses in the range from 0 to 777777. We now examine a recent extension of the PDP–10 architecture to provide a 30–bit virtual address. The new architecture defines techniques for generating 30–bit addresses. The use of this larger address space is generally called *extended addressing*. In extended addressing, the address space is divided into *sections*. Each section contains 512 512–word pages. Sections are numbered from 0 to 7777.

The program counter has been extended to thirty bits. The program counter is considered to be right–adjusted in a 36–bit word in which bits 0:5 are always zero. Bits 6:17 of the program counter define the section in which the program is executing. A program is said to be executing or running in the section whose number is specified by bits 6:17 of the program counter. Thus, we say that a program is executing (or running) in section 5 when bits 6:17 of its PC contain 5.

Section 0 is reserved as the *compatibility section*, where programs operate exactly as they do in an unextended system. Existing programs that make no use of extended addressing can be run in section 0 without modification. All the examples that we have written thus far have been run in section 0. When the processor is running in section 0, all effective addresses and all memory operands are in section 0.

A limited implementation of extended addressing exists in the DECSYSTEM–2060, where twenty–three bits of virtual address have been provided.[1] Only sections 0 to 37 are implemented in the 2060. Although only twenty–three bits are currently provided, we shall talk of the 30–bit addresses specified by the new architecture. When writing a program that uses extended addressing, the programmer should be aware that future machines may implement all thirty bits.

30.1 Differences in Non–Zero Sections

While running in a non–zero section, the central processor performs effective address calculations differently than it does in section 0. Particular instructions behave differently when executed in a non–zero section. New data structures for indirect addressing and for the program counter and flags are used in non–zero sections. New formats may be used for stack pointers and for byte pointers.

[1] Extended addressing for user programs was not available prior to release 4 of TOPS–20. Extended addressing is not available in either the 2020 or in any KL10–based systems other than the 2060.

30.1.1 Effective Address Calculation

In a non–zero section the effective address computation may develop either a local 18–bit address, called an *in–section* address, or a 30–bit *global* address. (It is more correct to think of the effective address as 31 bits: one bit that specifies either *global* or *local* and 30 bits of address.)

An *address word* is a computer word that is being used in the effective address computation. Initially, the instruction itself is the address word. When indirect addressing is specified, each word fetched to resolve the indirect reference is an address word. There are two formats for address words, local and global. These are depicted in Figure 30.1.

A *local address word* contains `I`, `X`, and `Y` fields in the same format as in an instruction. In fact, an instruction word is a local address word. A local address word that is fetched in the process of resolving an indirect address is called a *local indirect word*. A local indirect word must have a one in bit 0 and zero in bits 1:12.

A *global address word* can be brought into the effective address computation only by means of indirect addressing. This format is also called a *global indirect word*. A global indirect word contains a zero in bit 0; bit 1 is the indirect bit; bits 2:5 form the X field; and bits 6:35 contain the 30–bit extended `Y` field.

Note that the processor distinguishes global address words only when they appear in a non–zero section. In section zero, all address words are interpreted as local address words (and the configuration of bits 0:12 is irrelevant).

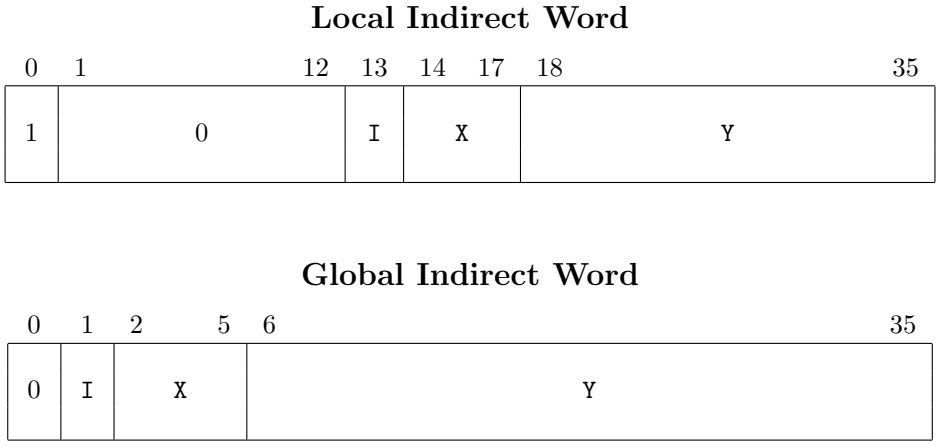


Figure 30.1: Extended Format Address Words

30.1.1.1 Local Addresses

When the effective address calculation is performed using an instruction word the result may be an 18–bit *local address*. A local address results from the most frequent addressing situation in which the `I` and `X` fields of an instruction are zero. Without indexed or indirect addressing, the local address is given by the `Y` field, bits 18:35 of the instruction.

When the effective address calculation yields a local address, a section number must be attached to that local address to extend it to a complete 30–bit address. The section number of the address

word is used to form the section number of the resulting address.

In the case of an instruction that does not specify indirect addressing, the address word is the instruction itself. The section number of the instruction word is the current PC section, bits `6:17` of the PC. Thus, when indirect addressing is not used, a local address will reference the current PC section.

A local address can also be derived from indirect addressing through a local indirect word. In such a case, the section number of the local indirect word forms the section number of the resulting address.

30.1.1.2 Indexed Addressing

When the computer is running in a non–zero section, the interpretation of an index register depends on which format address word the index register appears in.

When an index register appears in a local address word (such as an instruction), the effects of indexing depend on the contents of the index register. If the left half of the index register contains either zero or a negative number, *local indexing* is performed. In local indexing, the sum of the `Y` field plus the right half of the specified index register is truncated to an 18–bit local address. The section number of the effective address is supplied by the section number of the address word.

When an index register appears in a local address word and the register itself contains a zero in bit 0 and a non–zero number in bits 6:17, a form of *global indexing* takes place. In this form, the `Y` field is considered to be a signed displacement of the global address given in the index register. Bit 18 of the `Y` field is extended through the left halfword and this quantity is added to bits 6:35 of the index register to form the 30–bit global effective address.

The distinction based on the sign of the contents of the index register allows local indexing to work in a non–zero section in much the same way as indexing works in section zero. Moreover, those instructions that use a negative control count in the left half of an index register, e.g., `AOBJN` and the stack instructions, can continue to be used in non–zero sections.

Finally, when an index register appears in a global indirect word, the sum of the 30-bit `Y` field plus bits 6:35 of the specified index register form the 30–bit global effective address.

30.1.1.3 Indirect Addressing

When indirect addressing is specified, the results of the computations involving `X` and `Y` generate a 30–bit address from which a further address word is fetched. That address word may be either a local or a global indirect word, as described above. The effective address computation begins again using the new address word.

If an address word is fetched from section `0`, the remainder of the effective address computation will be performed following the rules of section `0`; the resulting effective address will be in section `0`.

30.1.1.4 Immediate Instructions

At the end of the effective address computation if the address is intended as a memory address, then either it is a global address or it is interpreted as referencing the section from which the last address word was fetched. If the effective address is an immediate operand then generally it will be truncated to eighteen bits (or fewer as in the case of the shift instructions). However, there are two instructions, `XMOVEI` and `XHLLI` that explicitly handle an extended address as an immediate operand.

30.1.1.5 Extended Effective Address Computation

To summarize the computation of effective addresses in non–zero sections, we offer the flow chart shown in Figure 5.6, page 54. In that figure, the flag "G" being 1 signifies that the resulting address is global; otherwise, the address is local.

30.1.2 Accumulator Addresses

An address references an accumulator when its in–section part is in the range from 0 to 17, and either the address is local, or the address is supplied by the PC, or the section number is 0 or 1.

The virtual addresses `1000000` through `1000017` are the global addresses of the accumulators. A global address in a section other than 0 or 1 refers to memory, i.e, `37000005` refers to word `5` in section `37`.

Instruction fetch is defined to use local addressing. Thus, when bits `18:31` of the program counter contain zero, instructions are fetched from the accumulators. For example, when the PC contains `16000006` an instruction will be fetched from accumulator `6`. If the program stores into accumulator `6`, and then jumps to `6`, it will fetch the instruction just stored there. To a first approximation, this behavior is desirable. However, the reader is cautioned that a subroutine may not be able to find any *in–line* arguments when it is called from the accumulators.[2]

30.1.3 Program Counter and Flags

The program counter is a 30–bit quantity. Although data structures are allowed to cross section boundaries, the program cannot change to another section except by an explicit jump. When the program counter is incremented as part of the normal instruction sequence, the carry out of bit 18 is suppressed to prevent changes to the PC section. Thus, after the program executes an instruction at `16777777` the program counter will be advanced to `16000000`; the instruction will be fetched from accumulator `0`.

The 30–bit program counter will not fit in the same word as the twelve flag bits. Most subroutine calling instructions store the program counter and flags in one word when executed in section zero. However, when these are executed in a non–zero section they will store only the 30–bit PC. The 30–bit PC will be stored in the format of a global indirect word with zero in bits 0:5. Among these instructions are `PUSHJ`, `JSR` and `JSP`.

Most of the common subroutine returns (especially those that do not attempt to restore the PC flags) will continue to work properly. Some examples are

```
        PUSHJ   P,SUBR1         ;Call SUBR1.  Return PC is on the stack
        ...

SUBR1:  ...
        POPJ    P,              ;POPJ will restore a 30-bit PC when
                                ;executed in a non-zero section
;------------------------------
```

[2]The subroutine `XIO1` in Section 32.4, page 616 is an example of a subroutine that has an in–line argument.

```
        JSR     SUBR2           ;Call subroutine SUBR2
        ...

SUBR2:  0                       ;30-bit PC is stored here
        ...
        JRST    @SUBR2          ;By indirect addressing, the
                                ;30-bit PC is restored.
;--------------------------------

        JSP     AC,SUBR3        ;Call subroutine SUBR3.  Return PC in AC
        ...

SUBR3:  ...
        JRST    (AC)            ;By global indexing, the
                                ;30-bit PC is restored.
```

If flags are wanted, they can be saved explicitly by the `SFM` instruction. `SFM` is one of the forms of `JRST`; it stores the flags into bits 0:12 of the word that it addresses, clearing the remainder of the word. `SFM` is legal only in a non–zero section.

The `XJRSTF` instruction restores the PC and flags from a double–word as shown in Figure 30.2.

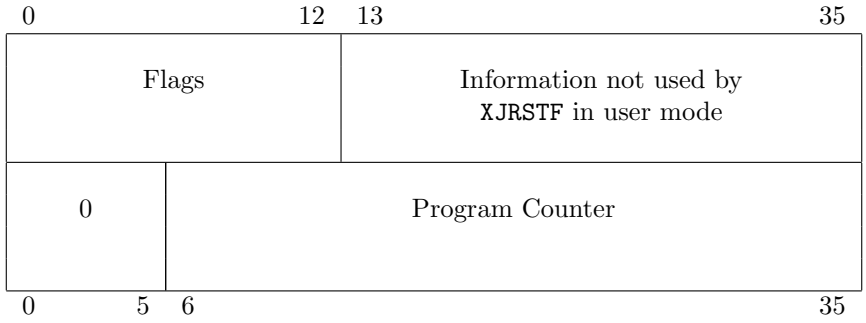


Figure 30.2: Flags and Program Counter Double–Word

30.1.4 Operand Addressing Differences

The `DMOVE` instruction (among others) refers to `C(E)` and to `C(E+1)`. When `E` addresses the last word in a section, i.e., when the in–section component of `E` is 777777, there is some ambiguity about what is meant by `E+1`.

This ambiguity is resolved by examining whether the computation of `E` resulted in a global address or a local address. If `E` is a local address then `E+1` is at in–section 0 in the same section as `E`; that is, there is no carry into the section number. If `E` is a global address then `E+1` is at in–section 0 of the next section past the section specified by `E`; that is, the sum `E+1` carries into the section number.

30.1.5 Instruction Differences

No control count is kept with a global stack pointer. Neither pushdown overflow nor underflow can be detected by the instructions. One method that may be employed to detect these conditions is to allocate some number of whole pages for the stack. Then by making the two pages adjacent to the stack area inaccessible to the program, the stack overflow or underflow condition will be transformed to an illegal reference to memory that can trigger an interrupt or program termination.[3]

30.2 Operating System Support

Release 4 of TOPS–20 is the first operating system to provide any support for user mode extended addressing. We have already seen one example of this support: the trap block in the `SWTRP%` JSYS has space for 30–bit addresses. Other support for extended addressing exists, but the user should be aware that not all system calls are equipped to handle global addresses.

30.2.1 Example 20 – Mapping to Section 1

The facilities in MACRO and LINK for multi–section programs are presently quite primitive. Until they are more fully developed, the usual way to make a program execute using extended addressing is to have the program copy itself from section `0` to a non–zero section. The `MAPIT` subroutine, described below, accomplishes this purpose. Please be aware that the developers of TOPS–20 are working to make this subroutine unnecessary; however, it demonstrates some useful techniques.

The `MAPIT` subroutine is called shortly after starting a program; it copies all the pages from section `0` to section `1`. If DDT is present, we want it to run in a non–zero section; as we will discuss shortly, the `MAPIT` routine accomplishes this also. `MAPIT` is called from section `0`; it runs in section `0` until all the mapping has been done. Finally, it jumps to section `1` and arranges to return in section `1`.

The original code in section `0` remains there. This is somewhat unfortunate because one of the most frequent programming errors is to jump into section `0`. When the program jumps into section `0` it loses the ability to perform extended addressing; the program may not immediately notice this inability.

The entire `MAPIT` subroutine, our example 20, appears below. Further explanation follows.

[3] One way to make a page inaccessible is to `PMAP` a non–existent page of a file into a memory page with only read access requested. Neither read, write, nor execute access to such a page is allowed.

```
        SUBTTL  Copy Program from Section 0 to Section 1.   Example 20

MAPIT:  MOVEI   A,0                     ;make a private section
        MOVE    B,[.FHSLF,,1]           ;section 1
        MOVE    C,[PM%CNT!PM%RD!PM%WR!PM%EX!1] ;1 sect. all access
        SMAP%
        ERJMP   SMPFAI                  ;shouldn't fail
        MOVE    D,[.FHSLF,,0]           ;initial fork page pointer
MAPIT1: MOVE    A,D                     ;fork,,page #
        RMAP                            ;read fork's map.
        CAMN    A,[-1]                  ;-1 means no access
        JRST    MAPIT2                  ;no work for no page
        MOVE    C,B                     ;access bits from RMAP
        MOVE    B,D                     ;fork,,source page#
        IORI    B,1000                  ;fork,,destination page #
        PMAP                            ;copy map from sect 0 to 1
        ERJMP   INITPE                  ;unexpected failure
MAPIT2: ADDI    D,1                     ;increment fork page pointer
        TRNN    D,1000                  ;done yet?
        JRST    MAPIT1

;Now, unmap UDDT from section 0
        SETO    A,                      ;unmap process page
        MOVE    B,[.FHSLF,,771]         ;unmap UDDT from section 0
        MOVE    C,[PM%CNT!7]            ;remove pages 771 through 777
        PMAP
        ERJMP   .+1                     ;ignore errors

        MOVE    A,[.FHSLF,,770]         ;is UDDT present?
        RMAP
        CAMN    A,[-1]
        JRST    MAPIT5                  ;no DDT at all.
```

```
;Make Section 0 UDDT jump to section 1.
;We can't write this page 770, so we discard it.

        PUSH    P,770000                ;save 770000,1,2
        PUSH    P,770001
        PUSH    P,770002
        SETO    A,                      ;unmap page 770
        MOVE    B,[.FHSLF,,770]
        MOVEI   C,0
        PMAP
        ERJMP   .+1
        POP     P,770002                ;Put back three words
        POP     P,770001
        POP     P,770000
        MOVE    A,[XJRSTF 770004]       ;the starting instruction
        MOVEM   A,770003                ;for UDDT goes into 770003
        SETZM   770004                  ;new PC flags for XJRSTF
        MOVE    A,[1,,770000]           ;new PC for XJRSTF
        MOVEM   A,770005
MAPIT5: XJRSTF  [0                      ;Now, jump to section 1
                1,,.+1]
        MOVSI   A,1                     ;Set section 1 in return addr
        HLLM    A,(P)                   ;set left half to section 1
        RET                             ;return in section 1
```

MAPIT begins by creating an empty section 1 for its use. The section is created by means of the SMAP% JSYS. We already examined this fragment for calling SMAP%.

The copying of pages from section 0 to section 1 is accomplished in the loop at MAPIT1. We could perform a word–by–word copy using a BLT instruction, for example, but it is much faster to simply make the map for section 1 the same as the map for section 0. This map changing can be done in several ways. We choose to do the following:

The RMAP JSYS allows us to read the map for each page. The argument to RMAP is simply a fork handle and page number in register 1. RMAP returns a *page handle* in register 1, and access flags in register 2. By a *page handle* we mean a description of a page; it may be a JFN and file page number, or a fork handle and a process page number. The page handle and access bits returned by RMAP can be used by PMAP. RMAP returns -1 in register 1 to signify that the page we asked about does not exist.

The loop at MAPIT1 examines every page in section 0, as selected by register D. If the page does not exist, the program jumps to MAPIT2. For each extant page, the page handle returned by RMAP is used as a source (register 1 of PMAP); the access bits are copied to register 3, and the destination page is set to 1000 more than the source page. Thus, the PMAP makes page 1000 a copy of page 0, page 1001 a copy of page 1, etc.

At MAPIT2 the source page in register D is incremented. The loop continues until the source page is counted to 1000.

```
        MOVE    D,[.FHSLF,,0]           ;initial fork page pointer
MAPIT1: MOVE    A,D                     ;fork,,page #
        RMAP                            ;read fork's map.
        CAMN    A,[-1]                  ;-1 means no access
        JRST    MAPIT2                  ;no work for no page
        MOVE    C,B                     ;access bits from RMAP
        MOVE    B,D                     ;fork,,source page#
        IORI    B,1000                  ;fork,,destination page #
        PMAP                            ;copy map from sect 0 to 1
        ERJMP   INITPE                  ;unexpected failure
MAPIT2: ADDI    D,1                     ;increment fork page pointer
        TRNN    D,1000                  ;done yet?
        JRST    MAPIT1
```

At this point, we should mention one of the alternatives to this loop that might be used. The `PMAP` JSYS can be used to make section `1` be the same as section `0` by placing *indirect pointers* in the map for section `1`. The fragment to accomplish this mapping is

```
        MOVE    A,[.FHSLF,,0]           ;Source: this fork page zero
        MOVE    B,[.FHSLF,,1000]        ;Destination: fork page 1000
        MOVE    C,[PM%RD!RM%WR!PM%EX!PM%CNT!1000] ;All access.
        PMAP                                       ;  1000 pages
        ERJMP   ...
```

This fragment produces `1000` (octal) page pointers to describe section `1`. Each is an indirect pointer; when the program references page `6` in section `1`, the system finds a pointer that specifies the use of whatever pointer is found in page `6` of section `0`.

`MAPIT` avoids making indirect pointers because there is no need to use them in this situation. The subroutine shown here produces direct pointers to the pages mapped into section `1`. One disadvantage of an indirect pointer is that it becomes useless when the underlying page disappears, thus, `MAPIT` would not be able to remove the section `0` copy of DDT (as it will do below) if DDT had been mapped into section `1` by indirect pointers.

If DDT is present, we want it to run in a non–zero section; when DDT runs in section `0`, it can display only un–extended addresses. Since `MAPIT` maps everything into section `1`, if DDT is present when `MAPIT` runs, most of the work will be done automatically. However, we must change DDT to execute in section `1`. The entry vector for DDT (the file `SYS:UDDT.EXE`) is located in `770000`; the EXEC `DDT` command starts DDT at `770000`. Traditionally, the word at `770000` is a jump to `770003`. We will change the instruction in `770003` to be an `XJRSTF` that jumps to `1770000`. Thus, DDT will start again, but this time it will be executing in section `1`.

When present, DDT uses pages `766` through `777`. Pages `766` and `767` are a data area for DDT; we will leave these alone because they contain any breakpoints that may have been set prior to starting the program. Page `770` includes the starting vector for DDT; this page must be changed. The remaining pages, `771` through `777` can be discarded.

We begin our manipulation of DDT by discarding pages `771` through `777`. These are code and constants that have already been copied to in section `1`.

```
SETO    A,                      ;unmap process page
MOVE    B,[.FHSLF,,771]         ;unmap UDDT from section 0
MOVE    C,[PM%CNT!7]            ;remove pages 771 through 777
PMAP
ERJMP   .+1                     ;ignore errors
```

Next, we test to see if DDT is present at all. If page 770 does not exist, DDT is not present and the program will avoid executing the rest of this fragment. Again, RMAP is used to test to see if a page exists:

```
MOVE    A,[.FHSLF,,770]         ;is UDDT present?
RMAP
CAMN    A,[-1]
JRST    MAPIT5                  ;no DDT at all.
```

The program jumps to MAPIT5 if DDT is absent. Assuming that page 770 is present, it will be write–protected. The code and constants for DDT are write–protected in an attempt to prevent any malfunction of the program that is being debugged from affecting DDT. Since our intention is to change the contents of page 770, we must circumvent the protection that has been applied.

There are two ways to deal with this write–protected page. First, we could use the SPACS JSYS to give this program write access to the page. The second way is to get rid of the page entirely. Since there are only three words on the page that we want to keep, getting rid of the page seems like good choice. The program saves the three words that we want preserved; these are saved by pushing them on the stack. Then the program discards the write–protected page 770 by unmapping it.

```
PUSH    P,770000                ;save 770000,1,2
PUSH    P,770001
PUSH    P,770002
SETO    A,                      ;unmap page 770
MOVE    B,[.FHSLF,,770]
MOVEI   C,0
PMAP
ERJMP   .+1
```

Now, when the program pops the stacked values into locations on page 770, TOPS–20 creates a new writable page 770 for these values. Finally, the program creates the instruction XJRSTF 770004 in location 770003, and stores a flag and PC double–word into locations 770004 and 770005; this PC addresses 1770000. The EXEC's DDT command will start DDT at location 770000. The instruction at 770000 jumps to 770003 where the XJRSTF sends it to 1770000.

```
        POP     P,770002                ;Put back three words
        POP     P,770001
        POP     P,770000
        MOVE    A,[XJRSTF 770004]       ;the starting instruction
        MOVEM   A,770003                ; for UDDT goes into 770003
        SETZM   770004                  ;new PC flags for XJRSTF
        MOVE    A,[1,,770000]           ;new PC for XJRSTF
        MOVEM   A,770005
```

As a result of this code, when the EXEC starts DDT at `770000`, DDT will jump to the section `1` copy.

At `MAPIT5` our work is nearly complete. The program performs an `XJRSTF` to enter section `1`. Then the return PC on the stack is changed to point to section `1`. This subroutine then returns to its caller, now running in section `1`.

```
MAPIT5: XJRSTF  [0                      ;Now, jump to section 1
                1,,.+1]
        MOVSI   A,1                     ;Set sect 1 in return address
        HLLM    A,(P)                   ;set left half to section 1
        RET                             ;return in section 1
```

Note that `MAPIT` returns with the program running in section `1`. This characteristic means that `MAPIT` must be called from the top level of the startup code. If `MAPIT` were called from within any other subroutine, the eventual return would not work properly; observe the contrast:

Proper use of MAPIT

```
START:  RESET
        MOVE    P,[IOWD PDLEN,PDLIST]
        CALL    MAPIT
        ...                         ;return here running in section 1
```

Incorrect use of MAPIT

```
START:  RESET
        MOVE    P,[IOWD PDLEN,PDLIST]
        CALL    INIT
        ...                         ;the program may never return here.

INIT:   ...
        CALL    MAPIT
        ...                         ;return here running in section 1
        RET                         ;return to the wrong section
```

In the incorrect usage, the `RET` to return from the `INIT` routine will restore an incorrect section number because the corresponding `CALL` was executed in section `0`. The actual result may vary depending on the state of the PC flags stored in the call to `INIT`, and on the particular hardware and microcode on which the program is executing. In no event will the result approximate any desirable behavior.

30.2.2 Interrupts

The `SIR` JSYS provides only 18–bit addresses for the channel table and level table. These short addresses are insufficient when extended addressing is in use. Therefore, the `XSIR%` JSYS has been implemented to extend the functionality of the `SIR` JSYS to long addresses.

The `XSIR%` JSYS requires a fork handle in register `1`. Register `2` should be loaded with the 30–bit address of the three–word argument block. The first word of the argument block is the block length which should be the constant `3`. The next word should be the 30–bit address of the extended level table in bits `6:35`, and zero in bits `0:5`. The third word contains the address of the extended channel table, in same format as the level table address. The `XRIR%` JSYS reads the addresses of the channel table and level table into an block in this format.

The *extended level table* contains three 30–bit addresses; each specifies the address of a double–word where the PC and flags will be stored. The *extended channel table* contains 30–bit addresses also; the level of each channel is specified in bits `0:5` of the channel's table entry.

30.3 Other Extended Addressing Examples

To better understand the style of extended addressing, we offer two brief examples.

30.3.1 Global Subroutines

In the discussion of the `SIN` library subroutine in Section 21.2.3, page 315 we saw an example of the standard subroutine calling convention. In that call, an *argument pointer* register is set up to contain the address of a block of arguments to the subroutine. The argument block generally contains the address of the actual arguments.

This format can be adapted to extended addressing by the inclusion of global address words in the argument block, and by adapting the calling sequence slightly:

```
MOVE    AP,[GIW ARGLST] ;global addr of the argument blk in AP
PUSHJ   P,@[GIW SUBR]   ;Call SUBR in another section
```

Here the symbol `GIW` signifies a user–defined macro that generates a global address word with the 30–bit address of the specified argument.

In the case where the argument list is in the same section as the caller, the argument pointer can be loaded by the `XMOVEI` instruction.

30.3.2 Large Arrays

Suppose we need to reference a million–element array, `Z[0..99,0..99,0..99]`. The array accessing polynomial can be used. The element `Z[i,j,k]` is accessed by this fragment:

```
MOVE    A,I                     ;first index
IMULI   A,^D100                 ;size of first dimension
ADD     A,J                     ;add second index
IMULI   A,^D100                 ;times size of second dim
ADD     A,K                     ;plus third index
MOVE    B,@[BYTE(2)0(4)A(30)Z000] ;access via global indirect
```

Here, the symbol `Z000` represents the 30–bit address of `Z[0,0,0]`.

30.4 Recommendations and Cautions

Building a multi–section program requires careful attention to details at the time the program is designed. As a general principle, programs are small and data structures are large. Therefore, one approach to building programs is to put all the code in one section and all the data (or at least the large data structures) in another section or sections.

Programs can never cross section boundaries "unconsciously". Although the program counter increments as a 30–bit quantity, the instruction following the one at 3777777 is fetched from accumulator 0, not from virtual memory location 4000000. If code is too large for one section, you must divide it into multiple one–section modules and make explicit jumps from one section to another. Large modules of code can be written for in–section use, with just a few places needing to know that cross–section jumps must be made explicitly. Subroutines that are commonly used by several modules may be jumped to via global addresses, or they can be mapped locally into (the same in–section pages of) several sections. Returning via `POPJ` to the caller in another section is automatic.

DDT presents a special problem. In order for its execute and single–step logic to work correctly, it must be able to execute (single–step) instructions from the current PC section. To accomplish this, DDT reserves a set of 100 (octal) locations in *every* section for its own use. (Of course, at any instant, DDT will use only those locations in the current PC section.) By default, DDT uses the last 100 words in the section: 777700 through 777777. The consequence of this is that you should not put any part of a multi–section array in a section that contains code. (DDT's use of those locations can be prevented by zeroing location `$4M` in DDT. Doing so disables single–stepping.)

The extended addressing version of DDT will locate itself at the high–end of section 37. (You may

presume that a DDT built for a machine that implements more than 23 bits of virtual address will locate itself at the extreme reaches of addressing range.)

Chapter 31

The TOPS–20 File System

The TOPS–20 operating system stores files on large capacity, high speed magnetic disk storage units. These disks are sometimes called *mass storage* devices because the amount of data they hold is typically many times larger than the main memory (core or MOS) that is present in the computer system. In this section we shall discuss the methods by which TOPS–20 imposes a useful structure, a *file system*, on the collection of disks that it uses. We have already examined the file system capabilities as they are seen by a user program; in this discussion our viewpoint is that of a systems programmer. This discussion is somewhat complex; we will have to talk about several areas which at first appear unrelated. When we understand each of these areas, the interconnection will be more apparent. As a road map or overview, our further discussion will include the following areas:

- **Physical Characteristics of Disks**. Disk drives have particular characteristics relevant to how data is stored and retrieved. The terms *sector*, *track*, and *cylinder* will be defined; these are the terms by which we specify the location of a data region on the disk.
- **Addressing the Disk**. A formula, much like the array addressing polynomial, is used to relate the *logical record number* to a specific disk unit, cylinder, track and sector. The logical record number is the form of *disk address* used by TOPS–20 to specify the location of data on the disk.
- **Home Block**. The home block is a special sector on each disk pack that identifies the contents of the disk. The home block defines the name of the structure of which this disk forms a part. The home block contains the pointer to the *root directory* for the structure.
- **Structure of a File**. Files are built from pages that may be scattered throughout the disk. Each data page is pointed to by an *index page* that defines the location of all the data pages. The file descriptor block points to the index page.
- **Directory Files**. A directory file contains file descriptor blocks. A file descriptor block describes the name, index page location, and attributes of a file in the directory. Directory files contain other information such as the password, disk quota, and date of most recent login.
- The **root directory** is the root of the tree of files; the root directory points to other directories that point to to yet further directories or to user files.

We will also discuss three data structures that are kept as files on the disk. In general, these are updated as part of the bookkeeping activities that TOPS–20 performs for the maintenance of the file

system. TOPS–20 goes through considerable effort to make file system operations happen quickly and accurately. These data structures contribute to that effort.

- **Index Table**. The index table is a data structure that provides a shortcut in directory access operations.
- **Bit Table**. The bit table is a data structure that defines the location of free and used space on the disk. The bit table is examined to locate free disk space when files are being allocated. When a file is deallocated, bits in the bit table that correspond to the data pages of the file are marked as free.
- **BAT Blocks**. Our discussion will conclude with a brief description of error recovery and how TOPS–20 avoids reusing bad spots on the disk.

31.1 Physical Characteristics of Disks

A typical disk drive contains a disk assembly, a positioner mechanism, and some number of read/write heads. The disk assembly, shown in Figure 31.1, contains one or more rigid platters, each resembling a phonograph record. The platters are attached to and supported by a hub assembly. As the hub turns, all the platters turn. Generally, both surfaces of a platter are coated with a thin layer of magnetizable material, e.g., iron oxide. Information is stored by changing the magnetization of small regions on the surface of the disk; this is similar to the way that sound is stored on recording tape.

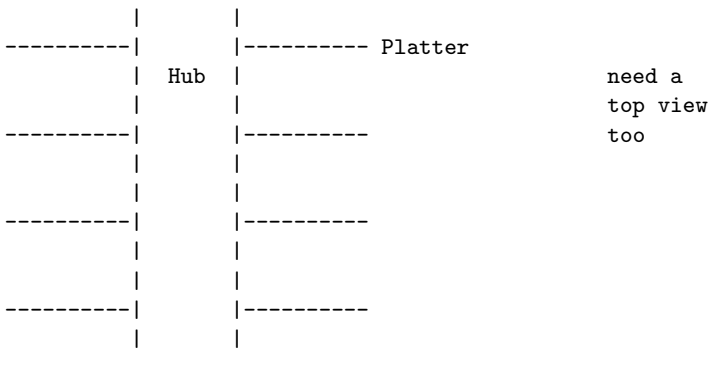


Figure 31.1: Disk Assembly

In some disk drives, the platter assembly is a removable unit, called a *disk pack*. In other disk drives, the disk assembly is an integral part of the mechanism and cannot be removed. Disks are thus characterized as having either *fixed* or *removable* media. Typically, a fixed media disk can store information more densely than is practicable on a removable disk pack. However, some applications require removable media, known as *mountable structures* in TOPS–20.

Information is stored on and retrieved from the disk by means of the read/write heads. Each of these heads rides just above the surface of a platter. It is common to see disks in which the *flying*

height of the read/write heads is only several microns (millionths of a meter). Near the read/write heads, the disk surface is moving faster than 100 miles per hour.

As the head is held in a fixed position above a spinning platter, the region of the platter that passes under the head is a narrow ring called a *track*. Information is written along this ring as the disk spins. Later, this information can be read in the same sequence as it was written.

In very high performance disks, there are many heads for each platter, and the heads do not move. These are called *fixed–head* or *head per track* disks. These disks permit higher performance than the *moving–head* disks (see below) because there are no delays while a head is moved to the desired track. Head per track disks are expensive because they require many precision read/write heads for each recording surface.

An economical alternative to the head per track disk is the moving–head disk. In a moving–head disk, generally there is only one read/write head for each recording surface; the heads for all the surfaces are attached to one positioner mechanism. This positioner moves all the heads along a radial line. As the positioner moves, the heads pass over different tracks. For each discrete position of the heads, the accessible data lies on a collection of tracks occupying a cylindrical region; unsurprisingly, this region is called a *cylinder*. To retrieve information from a moving–head disk, it is necessary to know which discrete cylinder number the positioner was at when the information was written. The positioner is then returned to that same location, and the data can be retrieved. Although the positioner mechanism is complex and expensive, a moving–head disk is less expensive than its fixed–head counterpart because the moving head disk needs vastly fewer read/write heads. Because of the delay inherent in moving the positioner, moving–head disks generally have lower performance than fixed–head disks.

In Figure 31.2 we show an arrangement of read/write heads attached to a single positioner mechanism.

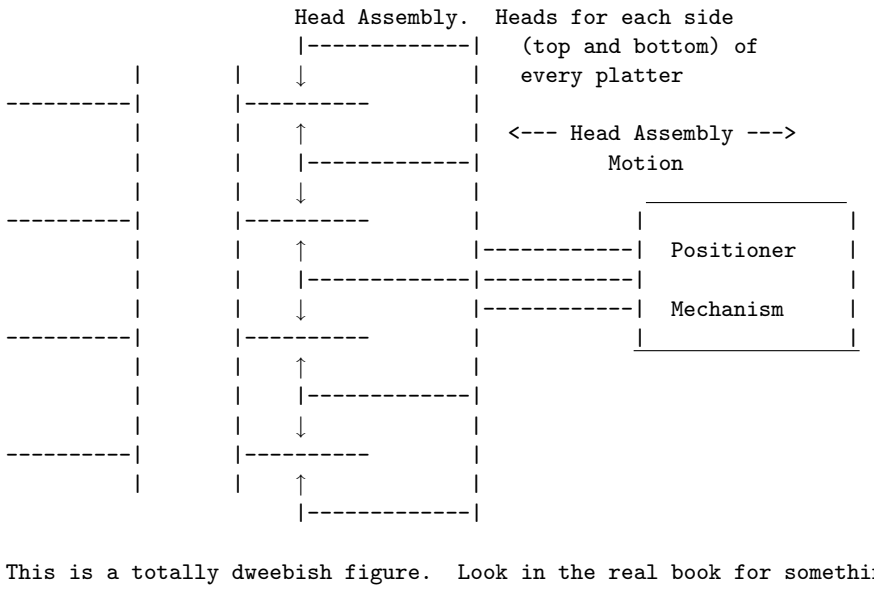


Figure 31.2: Disk Heads and Positioner Mechanism

We will continue this discussion with a specific example. The Digital Equipment Corporation RP06 disk drive is a typical removable media moving head disk. The disk pack for the RP06 contains 19 data surfaces. Each surface has 815 tracks of which 800 are used by TOPS–20. Tracks are further subdivided into *records* (or *sectors*), which are the smallest unit of information that can be stored on the disk. When used with the DECSYSTEM–20, an RP06 track holds 20 (decimal) records of 128 words each. The total capacity of each disk is 38.9 million words ($800 \times 19 \times 20 \times 128$).

31.2 Addressing the Disk

In order to retrieve information from a disk it is necessary to know where on the disk that information was written. The disk must be told the cylinder number, head number, and sector number to specify where to read or write. Since carrying these three numbers around is somewhat cumbersome, there is a simple formula to translate these numbers into a uniform linear address space. This formula allows us to deal with disk addresses as just one number. For the RP06 the *Logical Record Number* (`LRN`) is given by this expression:

```
LRN = (Cylinder * 19 + Head) * 20 + Sector

0 ≤ Cylinder < 800

0 ≤ Head  < 19

0 ≤ Sector < 20
```

This expression is essentially identical to the polynomial addressing formula for accessing arrays.

In TOPS–20 a *structure* is one or more physically identical disk drives that are logically one collection of information. TOPS–20 implements a complete file system, as described in detail below, on each structure. Because a structure may span more than one disk drive, the addressing formula is expanded as follows:

```
LRN = ((LUN * 800 + Cylinder) * 19 + Head) * 20 + Sector

0 ≤  LUN < number of units in the structure

0 ≤  Cylinder < 800

0 ≤  Head  < 19

0 ≤  Sector < 20
```

The variable `LUN` represents the *Logical Unit Number* that selects one of the several disk units comprising the structure. A 3–disk structure would have logical units 0, 1, and 2.

The following program fragment decomposes a logical record number, `LRN`, into sector, head, cylinder, and logical unit number.

```
MOVE    1,LRN           ;start with the logical record number
IDIVI   1,^D20          ;sector number is the remainder
MOVEM   2,SECTOR        ;save sector number
IDIVI   1,^D19
MOVEM   2,HEAD          ;head or track number is the remainder
IDIVI   1,^D800
MOVEM   2,CYLIND        ;cylinder number is the remainder.
MOVEM   1,LUN           ;and the logical unit is the quotient
```

The logical unit number is not sufficient to describe the specific physical disk drive that contains the data. Each physical disk is identified to the system by its *physical unit number* and by the *physical channel number* of the data channel through which the disk is connected to the system. It is necessary for TOPS–20 to translate the logical unit number into a physical channel and unit number. This translation is done by a table lookup. When a structure is first mounted, TOPS–20 builds a table that identifies the physical channel and unit for each logical unit of a structure.

The actual instructions by which the computer reads and writes on the disk will be discussed in Section 32, page 593.

31.3 Home Blocks

Each disk pack contains an area called the *Home Block* that identifies the structure that this disk belongs to, and which logical unit this disk pack represents within that structure. If the structure has space allocated for swapping or for a front–end file system, the location and extent of that space is defined in the home block. The home block is one record (128 words) in length; some of the record is currently unused. A typical home block for `PS:`, the public structure, is shown in Figure 31.3.

The home block is stored in record number 1 on every disk unit that is part of a structure built by TOPS–20. (To satisfy your curiosity, record 0 contains a bootstrap loader for the front-end processor.) Record 1 is located at cylinder 0, head 0, sector 1. When TOPS–20 first starts it reads the home block from every disk drive. The TOPS–20 initialization routines build the internal tables necessary to locate the one or more logical units comprising each structure.

Quite importantly, the home block contains the disk address of the *index page* that describes the *root directory* file. As we shall describe in greater detail below, a file's index page is a collection of disk addresses that describe the data area of a file. The root directory is the file from which all other files can be located. In the event that the root directory becomes damaged, a backup copy of the root directory is kept; the address of the backup copy's index page is also kept in the home block.

If the home block becomes damaged, there is a backup copy of the home block kept in record 12. When a structure is first mounted, TOPS–20 reads both versions of the home block; if each looks good then they must match. A good home block will contain the constant `SIXBIT/HOM/` in the first word and word 176 will contain 707070. If only one of the home blocks looks good then the good one is copied to replace the bad one. If both blocks are bad, or if they differ, then manual intervention is needed to reconstitute a consistent structure; in such a case, you might wish that this discussion had been more specific.

```
;Word   Contents                ;"RN" means Record Number

  0     'HOM   '                ;Sixbit HOM to identify this as a home block
  1     0
  2     0
  3     'PS    '                ;The name of this structure in Sixbit
  4     2,,0                    ;This pack is logical unit 0 of
                                ; 2 units in this structure
  5     1,,12                   ;RN of this HB ,, RN of backup HB
                                ;(in the backup HB, this word is 12,,1)
  6     11653                   ;Count of swapping pages present on this unit
  7     565                     ;Number of first cylinder for swapping space
 10     10005740                ;RN of Root Directory Index page
                                ; 10000000 is a flag signifying a disk address
 11     10013700                ;RN of backup Root Directory Index Page
 12     0
 13     1121600                 ;Number of records in this unit
 14     3100                    ;Number of cylinders in this structure
 15     310027,,270000          ;"media ID"

 61     100000,,14474           ;RN of Front End File system
 62     7330                    ;Record count of Front End File system

101     100000,,574             ;RN of Bootstrap Area
102     400                     ;Record Count of Bootstrap Area

173     BYTE(2)0(8)" "," "(2)0(8)"0","T"        ;"  TO" in PDP-11 ASCII
174     BYTE(2)0(8)"S","P"(2)0(8)"2","-"        ;"PS-2"
175     BYTE(2)0(8)" ","0"(2)0(8)" "," "        ;"0   "
176     707070                  ;constant to verify this is a home block
177     1                       ;RN of this record (this is 12 in backup HB)
```

Figure 31.3: Typical Home Block

31.4 Structure of a File

In our discussion of file mapping we said that the data portion of a disk file is composed of pages that correspond to pages of main memory space. Each page of a file is allocated on four consecutive disk records that form a block of 512 contiguous words. Successive pages of a file, however, may be placed at widely separated disk addresses. Among the functions of the TOPS–20 operating system are the allocation of disk pages to files, and the retrieval of data based on the file name and page number within the file. We will now discuss how TOPS–20 manages to retrieve data given a file's *index page* and the page number of the desired data page. Later, we will describe how TOPS–20 locates the index page from the file name. After describing the directories, we will explain how TOPS–20 keeps track of free space on the disk.

When all of a file's data pages are in the range between page 0 and page 777 (octal) then one *index page* is used to store the *disk address* of each data page. By looking in word 6 of the index page, TOPS–20 discovers where the file's data page 6 was stored on disk. When a particular file data page does not exist, the corresponding index page entry is zero. On the disk a file appears as an index page and a collection of data pages pointed to by the index page, as depicted in Figure 31.4.

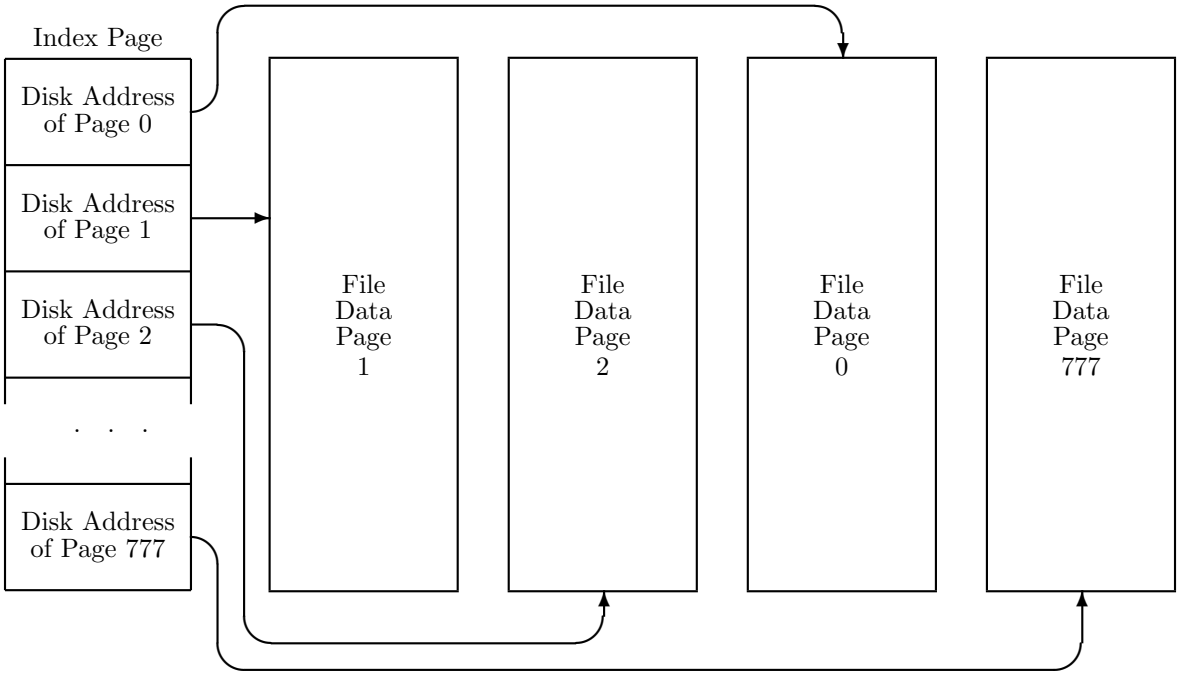


Figure 31.4: Structure of a File in TOPS–20

The index page is also stored on disk. Once TOPS–20 discovers the disk address of the index page, it can access all the data pages. TOPS–20 will copy the index page into main memory while actively accessing the file.[1]

When a file contains more than 512 (decimal) pages, or when it contains pages scattered at or above page number 1000 (octal), one index page is too small to describe the data pages of the file. Such a file is called a *long file*. In a long file, each group of 512 pages is described by an index page; the

[1]The file data pages are accessed by TOPS–20's virtual memory system, using the index page as a map to locate the data pages.

collection of index pages, up to 512 index pages, is described by a *super index* page. The structure of a long file shown in Figure 31.5.

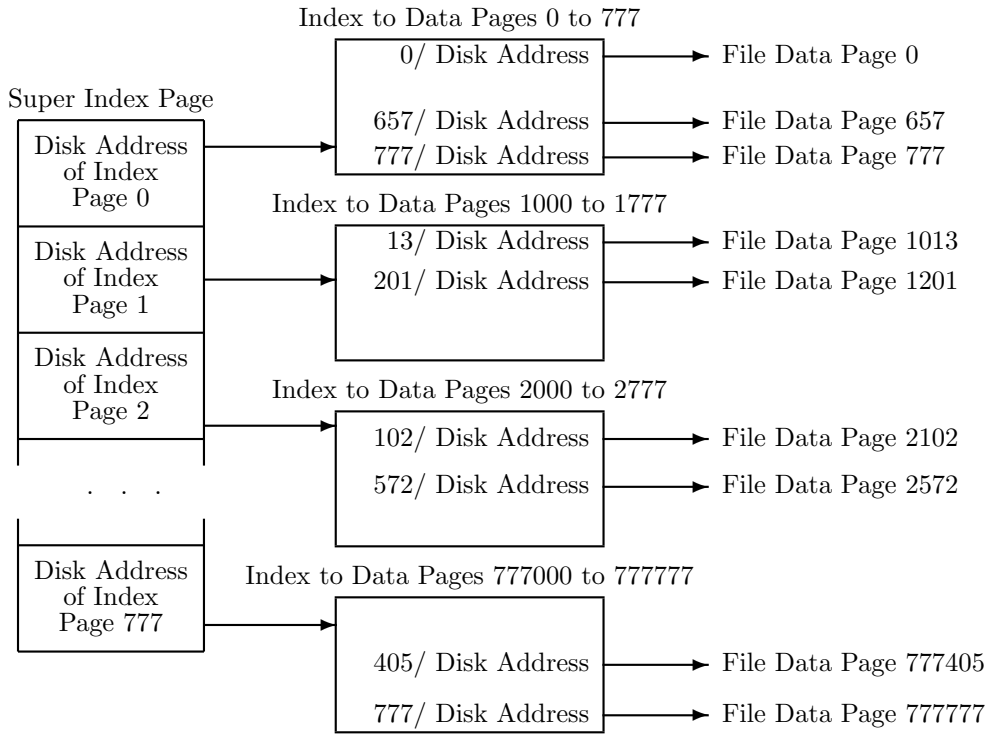


Figure 31.5: Structure of a Long File in TOPS–20

To locate a specific data page in a long file, TOPS–20 divides the given page number by 512. The quotient of this division determines the offset within the super index; this quotient identifies the disk address of the index page that describes the group of 512 pages in which the desired page appears. The remainder is used as an offset within that index page to locate the desired data page. When accessing the data pages of a long file, TOPS–20 will keep a copy of the super index page and some of the index pages in main memory.

At any point in the file if no data page exists, the pointer in the index page will be zero, i.e., empty. If a block of pages do not exist, then, generally, the corresponding index page will not exist. For example, if pages 13000 through 13777 are non–existent, then slot `13` in the super index normally will be zero.

We now know how to access a file if we can locate its index (or super index) page. The file system must provide a way to find the address of the index page. The disk address of a file's index page (or super index page in the case of a long file) is kept in the file descriptor block (FDB) for that file. The FDB is part of the directory file to which the selected file belongs.

31.5 Directory Files

The *root directory* is the root of a tree of directory files. The root directory points to user directories and to other directories that eventually point to each user directory file.

The root directory is a file; the disk address of the index block of the root–directory is found in the home block. The root directory is also a *directory file*, a data format known to the operating system. Directory files contain pointers to other files. These other files may be either ordinary data files or other directory files.

Among the ordinary files present in the root directory are the bit table file, the index table file, a bootstrap file to assist the PDP–11 front–end processor in initially loading itself, and a file containing the front–end processor's file system.

When TOPS–20 is told to look for a particular file, e.g., `PS:<ADMIN.GORIN.BOOK>B.MSS`, it starts searching in the root directory of the selected structure. In this example, the root directory should contain the file `PS:<ROOT-DIRECTORY>ADMIN.DIRECTORY`. `ADMIN.DIRECTORY` is another directory file; it is the directory corresponding to the user known as `ADMIN`. A search of this directory will yield an entry for the file `GORIN.DIRECTORY` within the directory for `<ADMIN>`. The full name of this file is `PS:<ADMIN>GORIN.DIRECTORY`. This is the directory belonging to the user `ADMIN.GORIN`. That file is searched to locate yet another directory file `PS:<ADMIN.GORIN>BOOK.DIRECTORY`. Finally, the `BOOK` directory is searched to find an entry for the file `B.MSS`. The extent of this search is quite remarkable: see Figure 31.6.

in principle, TOPS–20 has to follow this search structure whenever any file is sought. In practice, this search would take too long; the *index table* provides a shortcut that we shall describe later.

Each directory file contains the name, disk address, and other attributes of the files contained in the directory. Directory files have a special format that is understood and maintained by the operating system. A directory contains a symbol table (of file names), a file descriptor block for every file, and information about the user of the directory, e.g., the directory password, default account strings, disk quota, etc.

A directory file is divided into two regions. Most of the directory is made from a variety of *directory data blocks*; the remainder of the directory is called the *symbol table*. The directory header block is the data block that defines the extent of the various spaces. An overview of a directory is shown in Figure 31.7.

The symbol table includes an alphabetized listing of the first five letters of the name of every file in the directory. Associated with each of these name fragments is an address pointing to the data block that contains the FDB of the first file whose name starts with these five letters. When a specific file is sought, a binary search of the symbol table is performed. This search will locate the pointer to a list of FDBs that match the first five letters of the name that is sought; this list of FDBs will all be for files that have the same file name. Different file types are chained from the first FDB; pointers to different generations are likewise linked. In those cases where two or more different names begin with the same five letters, each of the different names will have an appearance in the symbol table.

Each directory data block has the format shown in Figure 31.8. A block is identified with an 18–bit block type in the left half of the first word of the block. The length of the block is held in bits 24:35 of that word. Another field in the first word is used to distinguish different versions of the same block type.

Included among the various different block types are the blocks for directory page headers, file descriptors, file names, types, account names, free space, etc.

Each page of the directory begins with a directory header block. The directory header includes the

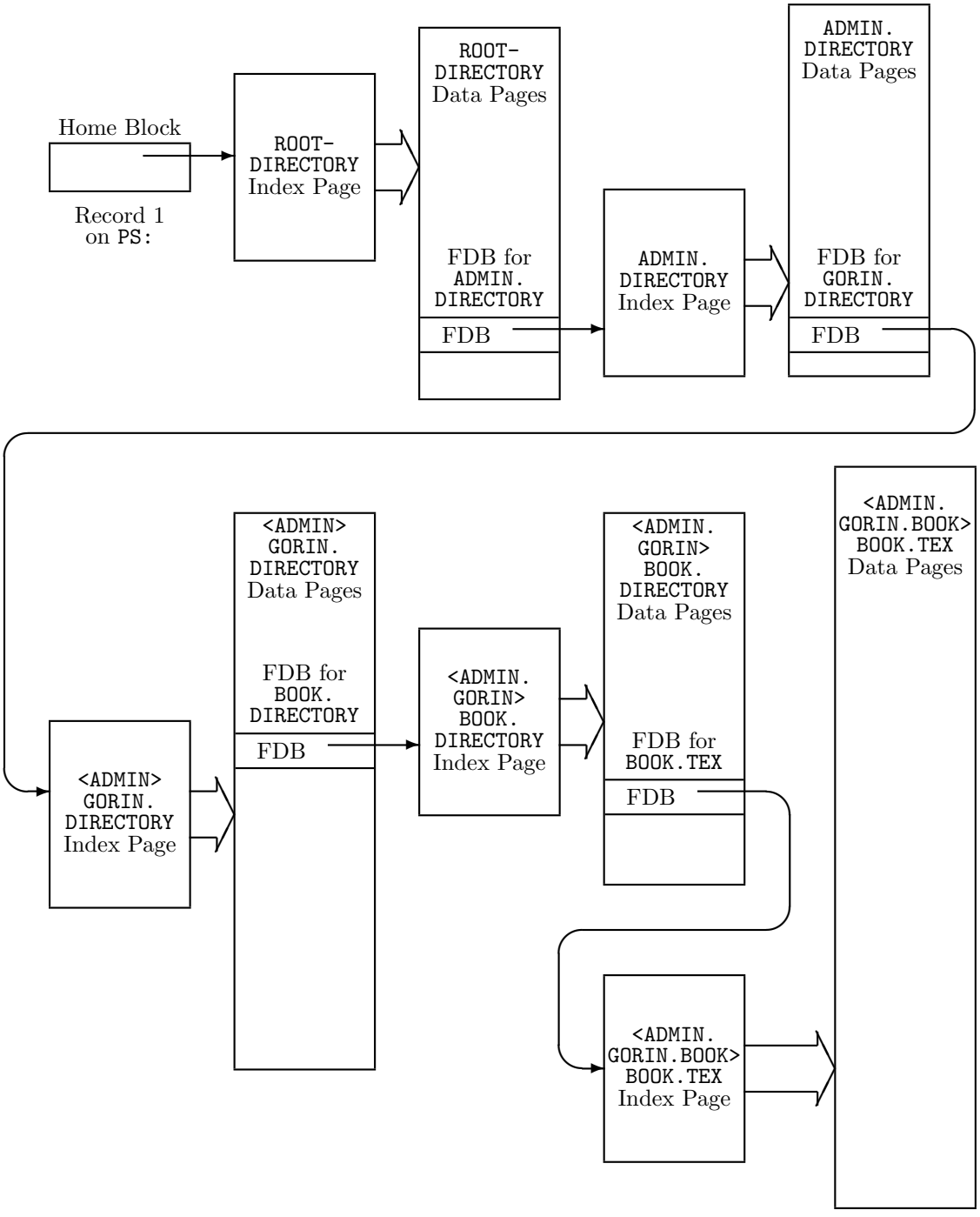


Figure 31.6: Retrieval Path for PS:<ADMIN.GORIN.BOOK>BOOK.TEX

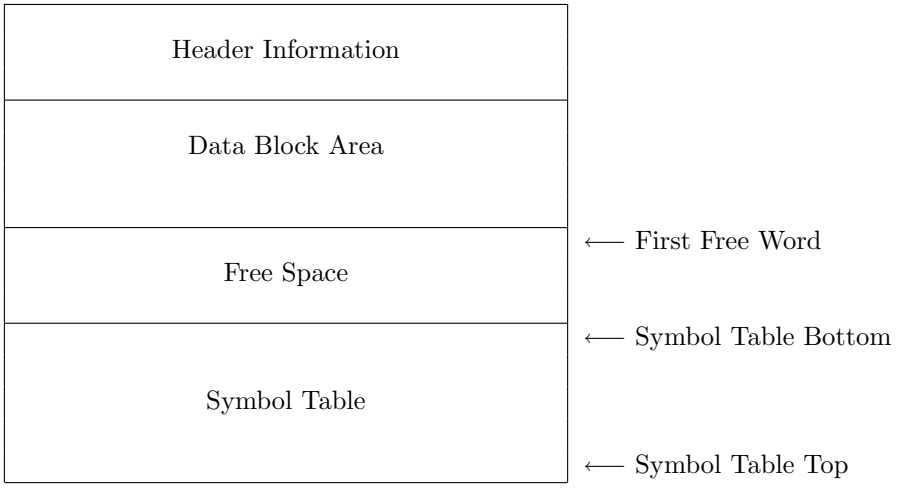


Figure 31.7: Directory Format

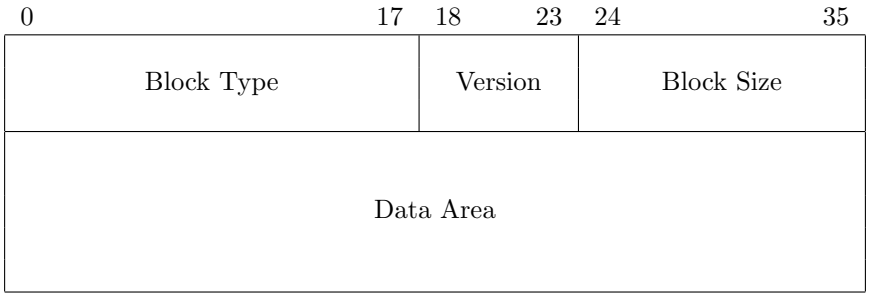


Figure 31.8: Directory Data Block

page number of this page in the directory, a copy of the directory number, and a pointer to the first block of free space, if any, on this page.

On the first directory page, the directory header block is expanded to include a multitude of other information about the directory. Among the information included here is

- Symbol Table Boundaries,
- The address of the Free Space List and the address of the Free–pool Bit Table,
- The Default File Protection and Default Directory Protection,
- The Working and Permanent Quotas, and
- The address of the Directory Name string and the address of the Password string.

The addresses in the directory are simply word numbers relative to the first word of the directory.

An important component of each file directory is the collection of File Descriptor Blocks. Each FDB describes the name, type, generation, author, account, creation date, and disk address of a file. The FDB also contains indications that describe the file as a long file, a directory file, an archived file, etc. As we have mentioned above, the symbol table provides a map from a file name to a list of likely file descriptor blocks. For directory scanning operations, as implemented in the `GNJFN` JSYS, the file descriptor blocks are linked in a fairly complex way. The file descriptor blocks, as we have already mentioned, are readable by using the `GTFDB` JSYS. The precise format of an FDB is described in [MCRM].

Text data, such as file names and file types are represented quite simply. A file name pointer in an FDB is simply the address of the multi–word block that contains a head word and the ASCIZ text of the name.

Other strings are represented in a more complex way. A user name string may represent either the author or last writer of a file. To save space in the directory, instead of having a separate block for each different use of one name, TOPS–20 uses only one user name string block for all occurrences of the same name. The same user name string or account string block may be referenced by more than one FDB. TOPS–20 must cope with two additional complications in order to use only one copy of each of these string blocks. First, when a particular string is needed in a new FDB, we must be able to find if that string exists already. This is accomplished by having a special section of the directory symbol table that points to user name and account strings. Second, when an FDB is deleted because its corresponding file is being expunged, we must know whether the strings it refers to can be deleted too. This can be decided by keeping a *reference count* in each string block; the reference count is incremented each time a pointer to the string is placed in an FDB; it is decremented each time an FDB that references this block is deleted. If the reference count becomes zero when it is decremented, the block can be deleted too.

31.6 Index Table

A data structure called the *index table* is used to reduce the number of operations needed to locate a file in the hierarchy of directories. The index table is contained in the file `INDEX-TABLE.BIN` in the root directory. TOPS–20 maps this file into its address space, starting at `IDXTAB`, when it needs access to the data.

Each directory file has a unique *directory number* that identifies it to the system. When a user logs in to the system, say, as ADMIN.GORIN, the file <ADMIN>GORIN.DIRECTORY must be located to check the password. Once the directory is found and the password is verified, TOPS–20 copies the directory number into a data area associated with the new job. This data item is called the job's *logged–in directory number*.

When a job is logged in, its logged–in directory number is copied to another data item called the job's *connected directory number*. This cell is changed when an EXEC CONNECT command is successfully performed.

When the user references one of the files in his or her connected directory, TOPS–20 uses the job's connected directory number as an index into IDXTAB. Given a directory number, IDXTAB contains the disk address of the index page of the specified directory. IDXTAB also contains information such as the directory number of the parent directory.

31.7 Bit Table

The bit table defines the location of free and used space on the structure. The bit table is contained in the file <ROOT-DIRECTORY>DSKBTTBL. The bit table has two sections; the first one is a count of how many free pages are present on each cylinder. The second section contains the actual bits: for each cylinder, one bit per page on the cylinder. The bit table is read when a structure is mounted; the two sections are checked for consistency.

There is a general intention in TOPS–20 to start allocating a new file on a cylinder that has a lot of free pages. Thereafter, TOPS–20 tries to minimize the number of different cylinders used to store each file.

It is extremely important to perform disk allocation and deallocation operations in the correct order. If allocation is done correctly then although the system may crash from time to time, the disk will contain an intact and consistent structure. Only files that were open at the time of the crash will be incomplete. As an example of the necessity of correct ordering of allocation operations, consider the problem of adding one page to an existing file. For this operation the system must allocate a free page from the bit table and add the corresponding address to the file's index page. If the operations are done correctly, the revised bit table will be written on the disk before the system writes the updated index page. If the index page were written first, and if the system were to crash before writing the bit table, a dangerous inconsistency would be present. That is, the bit table would say that the page was free, but some file's index page would claim that page as part of the file. If left uncorrected, that page would subsequently be reassigned to another file, spreading confusion and damaging data on the disk.

When the operations are correctly ordered, the bit table reflects that the page is in use. If the system crashes before writing the index page the resulting inconsistency is more benign: the page is marked as being used, although no file can be found that claims it. We can deal with that inconsistency by occasionally running the CHECKD program that rebuilds the bit table from the file structure. If disk operations were not performed in the correct order, it would be necessary to rebuild the bit table after every system crash.

It is also wise to be cautious when releasing space. When a deleted file is being expunged, the proper order of operations is to remove the deleted file's FDB from the directory and then rewrite the directory. Only after the directory is written is it safe to modify the bit table to release the space occupied by the file's index page and data pages.

31.8 Error Recovery and BAT Blocks

Occasionally, bad spots are found on a disk. Sometimes these are manufacturing defects, such as voids in the magnetizable coating. In other cases, bad spots develop after the disk has been in use for some time. In any event, if you value your data, you won't write it on a bad sector.

Some disk data errors are correctable. TOPS–20 will retry a failing operation several times, using different techniques, before giving up. There are basically three methods of error recover: retries, track offset, and error correcting codes. The first of these is quite simple: retry the operation. Just asking the disk to read a record for a second time often makes some transient errors disappear.

The second technique is to move the head away from the nominal centerline of the track; this is called *track offset*. Sometimes the relative alignment of disk heads will vary slightly from drive to drive. Track offset allows the program to adjust the disk for this misalignment. Track offset moves the head away from the nominal track centerline in either direction. The movements are made in tiny increments ranging from 25 to 400 micro inches.

The third technique is more complex. Whenever data is written on the disk, extra information is added at the end of each sector. This information is called an *Error Correcting Code* or ECC. The ECC is a mathematical function of the data bits; the ECC bits are computed by the disk hardware as it is writing the data portion of a sector. When it reaches the end of the data portion, the disk drive writes the computed ECC bits. When the sector data and ECC bits are read, the same function is applied again. If the result is some particular constant (in the case of the RP06 the constant is zero) then the data is correct. Otherwise, the resulting *error correction syndrome* can be manipulated to attempt to identify the location of the error and which bits to change at that location; error correction will be successful if the error involves only bits within a small region of consecutive bits.

All forms of error recovery are costly in terms of disk throughput; while the disk is retrying a failed operation, or while its busy doing computations on the error syndrome, it can't be bothered with doing other transfers. For safety and throughput, it is best to avoid bad spots.

When TOPS–20 applies either track offset or error correcting codes to repair an erroneous sector, it places the offending disk address into a list of bad addresses; also, TOPS–20 marks the file as having a "possible data error." These bad disk addresses are held in a special record called the BAT Block. The primary BAT Block is located at record 2 on each disk, with a backup copy at record 13.

When a file that is flagged as having a possible data error is expunged, TOPS–20 consults the BAT Block before releasing the file pages. For each file page that is mentioned in the BAT Block, TOPS–20 does not release the page from the bit table. Thus, bad blocks are not re–used. When the CHECKD program is run to rebuild the bit table it makes sure that all bad blocks are marked as used space.

Chapter 32

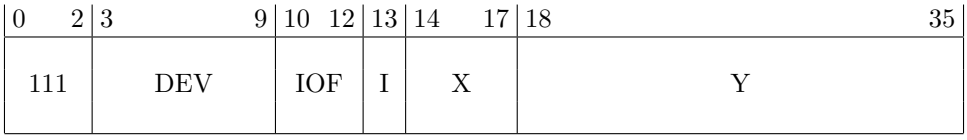
Hardware I/O Facilities

No discussion of assembly language programming would be complete without a description of how to direct the operation of peripheral units. The opportunities to practice what we discuss here are limited, unless the reader embarks upon writing parts of the operating system. The operation of peripheral devices is usually an operating system function; we will limit the depth of this discussion to avoid entanglement in a detailed description of how the operating system works.

This material should be viewed as an introduction to the concepts of I/O programming; it is the style and scope of these operations that we wish to emphasize, not the details of the instruction formats. The reader should note that the KS10 processor, used in the DECSYSTEM–2020, does not use the I/O instructions that are described here; nor is it likely that successors to the KL10 processor will use these I/O instructions either. Again, the concepts are important, not the details.

The DECSYSTEM–20 deals with input and output devices by means of a small set of privileged instructions. Only the operating system is allowed to execute these instructions; normally, these cannot appear in a user's program.[1]

In the KL10 processor, the input/output instructions have a format that differs from the regular instructions. The format for an I/O instruction in the KL10 is shown below:



111	Signifies an Input/Output Instruction	Bits 0:2 of I/O instructions
DEV	Input/Output device number	Bits 3:9 of I/O instructions
IOF	Input/Output function code	Bits 10:12 of I/O instructions
I	Indirect Bit	Bit 13
X	Index Field	Bits 14:17
Y	Address Field	Bits 18:35

[1]There are two exceptions in which I/O instructions are legal in user programs. First, privileged users can enter *User IOT mode* in which the I/O instructions are legal; usually this is done for diagnostic purposes. Second, ordinary users can execute I/O requests to specific devices which are identified by having *device code* numbers that are within a specific range. However, few systems are equipped with such devices.

PDP–10 Input/Output Instruction Format

The pattern `111` in bits 0:2 signals that this is an I/O instruction. The particular operation is encoded in the three bits `10:12` that are labeled `IOF`. The device number of the affected device is written in the `DEV` field, bits `3:9`.

There are eight I/O instructions. The four basic I/O instructions are

`DATAI`	Data In	Read Data from Device to Memory
`DATAO`	Data Out	Send Data from Memory to Device
`CONI`	Conditions In	Read Status from Device to Memory
`CONO`	Conditions Out	Send Control Information to Device

These four instructions transmit data and commands from the computer to the device, and transmit data and status from the device to the computer.

Two of the remaining I/O instructions are specializations of the `CONI` instruction:

`CONSZ`	Conditions In, Skip if Zero
`CONSO`	Conditions In, Skip if One

These instructions test the device conditions (i.e., status) against an immediate mask specified by the effective address field. In the case of `CONSZ`, the instruction skips if all of the bits selected by the mask correspond to zero bits in the device status. `CONSO` will skip if any condition bit matching a mask bit is set to one.

The two remaining I/O instructions are specialized versions of `DATAI` and `DATAO`:

`BLKI`	Block Data In
`BLKO`	Block Data Out

These two instructions languish in obscurity; they are not commonly used with modern devices. However, the operation codes for `BLKO` and `BLKI` are sometimes used to make an *internal device* perform special operations (see below).

The `DEV` field allows `128` different I/O devices to be selected. When an I/O instruction is written, the `DEV` field appears instead of an accumulator specification. In MACRO, device codes are always written as multiples of `4`; they take on values `0`, `4`, `10`, `14`, `20`, ... `774`. When MACRO sees a device code field, it right shifts the value by two bits and places the results in bits `3:9` of the instruction.

The `I`, `X`, and `Y` parts of an I/O instruction are interpreted and written in the same way as for other instructions.

When a I/O instruction is executed, the computer will transmit the device selection field and commands along a collection of signal wires called an *Input/Output bus*. (In the KL10, the internal version of this bus is called the E bus). The Input/Output bus carries device selection, commands, data, and other information; the bus connects the computer to all of its peripheral devices. When a connected device recognizes its own selection code, it responds to the command signals. For example, a device would respond to a `DATAO` instruction by accepting the data sent from the processor; a device would respond to the `DATAI` instruction by placing a data word on the Input/Output bus for the processor to read.

Some of the I/O device numbers are used to control parts of the processor that are not really peripheral devices; these are called *internal devices*. Among these parts are the processor itself (mnemonic `APR`, device number `0`), and the priority interrupt system (`PI`, device `4`). The use of

I/O device numbers has been extended to allow the operating system to control other parts of the complex processor; the KL10 includes as internal devices the cache sweeper (CCA), the pager (PAG), the performance and accounting meter (MTR), and the timer (TIM).

In general, an I/O device consists of some electronic and mechanical components that are the device itself, and additional electronic components that serve as the *interface* between the device and the computer's Input/Output bus. The distinction between the device and its interface is somewhat murky; from the programmer's point of view, the device and its interface are inseparable. When we write a program to affect a device, we are really affecting its interface. The interface is the medium through which our commands are delivered to the actual device.

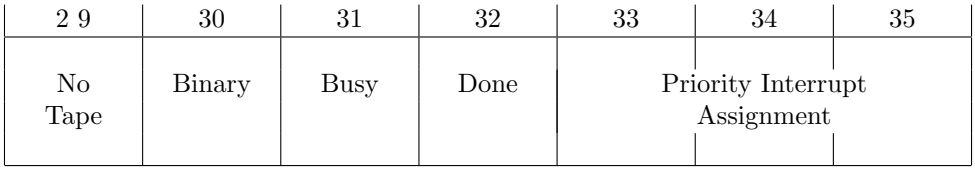
32.1 A Simple Example

The paper tape punch (PTP, device number 100) is one of the simplest examples of an I/O device. In the DECSYSTEM–20 the paper tape punch and reader is an option; the paper tape punch has been chosen for this example because of its inherent simplicity.

The punch itself has a motor that turns a shaft to which a cam is attached. A lever rides on the cam; the lever transmits up and down motion from the cam via eight independent pawls to the eight punches. Each punch is activated independently by energizing a magnetic coil that releases its pawl. The pawl catches on the lever and transmits the upwards stroke of the lever to the punch. As the lever moves down again, the punch is returned by a spring. When the punches are withdrawn from the paper tape, another mechanism driven from the same motor advances the tape to the next frame.

The punch interface is designed to conceal the detailed operation of the punch, leaving just enough visible so that a program can make the punch perform the desired operations. The punch interface contains knowledge of the mechanics of the punch, so that we don't have to worry about details such as how long to wait after starting the motor to be sure that it is going fast enough to punch the tape.

The punch interface has an 8–bit *data register* into which we put a binary pattern corresponding to the pattern that we want punched on one frame of tape. The PTP interface also has a *status register* that describes the condition of the device. By setting or clearing bits in the status register, the program controls the device.



Paper Tape Punch (PTP) Status Register

CONI and the other two condition testing instructions, CONSZ and CONSO, read the status register. Some bits in the status register can be set and cleared via the CONO operation. CONO is an immediate mode instruction; it transmits the 18–bit effective address to the status register of the device selected in the DEV field of the instruction. Not all status bits can be affected by CONO; for example, the No Tape bit, which signifies that the tape supply is exhausted, cannot be changed by CONO.

In order to program the paper tape punch conveniently, we make some definitions so we can refer to the device and its status bits by mnemonic names. Not all of the PTP status bits are germane to our present discussion.

```
PTP==100                ;PTP device number definition
PTPDON==10              ;PTP DONE status
PTPBSY==20              ;PTP BUSY status
```

When the paper tape punch is completely idle, both the flags BUSY and DONE will be zero. When the punch is not BUSY, i.e., when the PTPBSY status bit is zero, we can send data to the punch by performing the DATAO instruction:

```
        DATAO   PTP,DATA
```

In general, DATAO fetches the 36–bit word addressed by the effective address of the instruction. This data word is sent to the device selected by the device code field of the instruction. In this case, the contents of the word at DATA are sent to the data register in the paper tape punch. This data register holds only eight bits, bits 28:35 of the word that is sent; all other bits are discarded. The eight bits that are kept in the data register will be punched onto one frame of paper tape.

The DATAO instruction loads the data register in the paper tape punch interface. When data is loaded into the data register, the interface will start the punch motor, activate the punch magnet coils corresponding to the bits in the data register, punch one frame, and advance the tape.

Although the DATAO instruction is performed nearly as quickly as any other instruction, the function that DATAO initiates in the device may take a long time to complete. We must be certain that the punch has finished its activities before we can send another character to it. We can ascertain the readiness of the punch by examining bits in its status register.

When the tape punch interface receives the data sent by the DATAO instruction, it sets BUSY (and clears DONE) in the status register; then it starts the punch. When the device finishes punching one frame of tape, the interface clears BUSY and sets DONE. We shall test the status register to determine that the device is not BUSY before sending it more data. The CONI instruction reads the PTP status register. The instruction

```
        CONI    PTP,ADDR
```

reads the contents of the PTP status register into the word at ADDR. We can then examine the bits in ADDR to see if the BUSY flag is zero.

Rather than go through several instructions to read the status into a word and then test that word, we can use the CONSZ instruction to read and test the device status in one instruction. The instruction

```
        CONSZ   PTP,PTPBSY
```

reads the PTP status register and skips if bit 31 (BUSY, corresponding to the mask PTPBSY) is zero.[2]

To avoid sending characters too soon, we can use the following subroutine. PUNCH waits for the punch to be ready to receive another character and then punches the character found in the word

[2]In some devices there are more than eighteen status bits; bits in the left half of a device status register can be read by CONI but they cannot be tested by CONSO or CONSZ.

at DATA.

```
PUNCH:  CONSZ   PTP,PTPBSY      ;test punch status.  Skip if BUSY = 0
        JRST    PUNCH           ;loop until BUSY flag is zero.
        DATAO   PTP,DATA        ;not busy now.  Send a character.
        RET
```

The loop at PUNCH and PUNCH+1 is extremely wasteful of computer time; it may loop more than 10,000 times before punching each character. If you are writing a diagnostic or some other program in which you only want to do one thing at a time, it might not matter what you do while waiting for the punch to finish each frame of tape. However, if you are writing a program in an operating system environment where there is other work to be done, it is very important not to waste the 20,000 instruction times between each frame of tape. To avoid wasting time in such a loop, we can make the punch tell us when it is ready to receive another character. This can be accomplished by means of a hardware interrupt, via the *priority interrupt system*.

32.2 Hardware Priority Interrupt System

Hardware interrupts are similar to the software interrupts that we described in Section 29.2, page 545. The similarity is not coincidental; the software interrupt system is patterned after the way the hardware interrupts work. A hardware interrupt is an external event that causes the computer to put aside its other business and respond to the condition that caused the interrupt.

Each event has an assigned *priority level*. The priority determines the urgency of response; while processing an interrupt of a low priority, the computer may be interrupted again by an event of higher priority. When the computer is finished responding to an interrupt, it returns to the process that was interrupted.[3] The hardware interrupt system is often called the priority interrupt or PI system.

The PDP–10 provides seven levels of priority for I/O devices. These levels are numbered from one to seven, with level 7 having the least priority.[4] While serving an interrupt of some particular priority, e.g., 5, the processor is said to be executing at that priority level, or *interrupt in progress* at that level. User mode programs and much of TOPS–20 run in the unnamed level beyond level 7 where no interrupt is in progress.

In order to use the priority interrupt system with an I/O device we must decide several things. We must select a priority level for the device to use. We must turn on and enable the priority interrupt system for interrupts at that level. We must provide an appropriate *interrupt instruction* that jumps to the *interrupt service* routine. Lastly, we must tell the device which level to use and then induce it to interrupt.

There are two kinds of interrupts, *vectored* and *unvectored*. For a device that is capable of making a vectored interrupt, the program's initialization of the device will include the execution of an appropriate I/O instruction to assign an *interrupt address* to the device. Thereafter, when the device seeks to interrupt, it will supply that address to the processor; the processor will start the interrupt by executing the *interrupt instruction* that it finds at the given address. The interrupt instruction should store the PC and flags and jump to the interrupt service routine for this device.

[3] While processing one interrupt, an interrupt of lower priority may be requested; the lower priority interrupt will be postponed until the computer is finished responding to the interrupt of higher priority. Then, on its way back to the originally interrupted process, the computer will honor the pending low–priority interrupt.

[4] In the KL10 there is a priority level 0 that is unlike the other levels that are described here.

Traditionally, the interrupt instruction is a `JSR` that jumps to the interrupt service routine. When extended addressing is used, the interrupt instruction is `XPCW`, *Exchange PC Words*.

Less sophisticated devices use unvectored interrupts. When an unvectored interrupt occurs, the processor executes an interrupt instruction that it finds at a location that is specified as a function of the level number of the interrupt. In the KL10, for priority level `n`, the word at location `40 + 2 * n` in a special page called the *Executive Process Table* (EPT) is executed as the interrupt instruction. As in a vectored interrupt, the interrupt instruction must store the PC and flags of the interrupted process. In an unvectored interrupt, however, before a device specific interrupt service routine can be called, the program must determine which of several possible devices caused this interrupt. All unvectored devices share the same group of interrupt instructions. Specifically, all unvectored devices assigned to level `6` will use location `54` (`54 = 40 + 2 * 6`) in the executive process table as their interrupt instruction. Potentially, several devices may interrupt simultaneously; it is necessary to determine which device requires service. The device specific interrupt service routine is selected by finding which device is interrupting, as we shall describe below.

The purpose of the interrupt service routine is to determine the specific cause of the interrupt and make an appropriate response. The response must at least cause the interrupting device to stop requesting an interrupt. In the case we will examine, sending more data to the device will make it stop interrupting. The interrupt service routine may not assume that any accumulators are available or set up; it must not change any accumulators. The interrupt routine should use a local stack (private to this interrupt level), and save the accumulators in a block used only by this interrupt level. The interrupt service routine will service the source of the interrupt, performing whatever functions are needed by the device that caused the interrupt. When finished, the service routine should restore the accumulators, PC, and flags of the interrupted process, and exit from this interrupt level.

The following are code fragments that pertain to running the paper tape punch by means of the priority interrupt system.[5] We will select interrupt priority level `6` for the paper tape punch.

The first fragment is part of the general initialization of the priority interrupt system. This fragment stores an interrupt instruction in location `54` of the EPT. This is the instruction that will be executed when an unvectored device interrupts on level `6`.

```
MOVE    1,[JSR LEVEL6]  ;branch to level 6 interrogation routine
MOVEM   1,EPTORG+54     ;Store interrupt instruction in
                        ;location 54 of the EPT.
```

The initialization of the priority interrupt system includes turning the interrupt system on, and enabling particular levels to receive interrupts. The processor's priority interrupt system is controlled by commands issued to I/O device `4`, mnemonic `PI`. Some of the definitions pertaining to the `PI` system are given below.

[5] As these concepts are difficult enough without delving into every detail, this example shows the instructions appropriate to an unextended processor. Various simplifying assumptions have been made throughout; this example is not intended to be complete.

```
PI==4                   ;PI system device code

;PI CONO bits:

PICLR==10000            ;Clear PI system
PILVON==2000            ;Turn on selected levels
PILVOF==1000            ;Turn off selected levels
PIOFF==400              ;deactivate PI system
PION==200               ;activate PI system
PILV1==100              ;select level 1
PILV2==40               ;select level 2
PILV3==20               ;select level 3
PILV4==10               ;select level 4
PILV5==4                ;select level 5
PILV6==2                ;select level 6
PILV7==1                ;select level 7
```

As part of the initialization, we should execute the instruction

```
        CONO  PI,PION!PILVON!PILV6  ;turn on PI system, enable level 6
```

This instruction turns on the PI system and enables interrupts on level 6. Probably, in a real operating system environment more than just one priority level will be enabled.

The next fragment shows the LEVEL6 routine. This routine is called when the processor executes the JSR in location 54 of the EPT in response to an unvectored interrupt for priority level 6. This fragment is called a *CONSZ chain*, taking this name from its structure. For each device that may make an unvectored interrupt on level 6, a pair of instructions appears. The first of these is a CONSZ instruction to interrogate the device to see if this is the device that is requesting an interrupt. The CONSZ is followed by a jump to the device–specific interrupt service routine. If a device is not interrupting, its corresponding CONSZ skips to the CONSZ for the next device.

```
LEVEL6: 0                       ;Store PC and flags here.
        CONSZ   PTR,PTRDON      ;is paper tape reader interrupting?
        JRST    PTRINT          ;service the PTR
        CONSZ   PTP,PTPDON      ;is paper tape punch interrupting?
        JRST    PTPINT          ;service the PTP
        ...                     ;tests for other devices
```

When the paper tape punch finishes punching each frame of tape, DONE comes on and BUSY goes off. When DONE is on, and the device has been assigned a priority interrupt level, the device interface requests an interrupt on its assigned level. Assuming that we had the good sense to tell the punch to use level 6, when the processor honors the interrupt, the fragment at LEVEL6 will run. Since the DONE flag is on, the CONSZ PTP,PTPDON instruction will fail to skip, so the processor will jump to the PTPINT routine.

Thus, PTPINT is called when the paper tape punch is ready to accept data for another frame of tape. Fundamentally, the goal of PTPINT is to clear the interrupt condition by sending another character to the punch.

PTPINT begins by calling the LV6SAV co–routine. We shall examine LV6SAV in further detail shortly, but for now you may think of it as similar to the ENTLV3 co–routine that was explained in Section 29.2, page 545. LV6SAV saves the accumulators of the interrupted process, establishes a stack, and returns to its caller. PTPINT continues by decrementing a byte count, PTPCNT. If the result is positive or zero, the routine assumes that the byte pointer PTPBYP can be incremented to fetch another byte to punch on the tape. That byte is fetched into register 1 and then transmitted via DATAO to the device. Sending data to the punch makes it drop the DONE flag and set BUSY. Because the device no longer asserts DONE, it is no longer requesting an interrupt. The program returns through LV6SAV to dismiss the interrupt. (We have not yet addressed the question of who set up the byte pointer and count needed for this routine.)

When the byte count is exhausted the program jumps to PTPSTP. At PTPSTP the program stops the punch because there is no more data for it. We stop the punch by executing a CONO in which the PTPDON flag and the priority level assignment, bits 33:35 in the status register, are cleared to zero. Clearing DONE makes the device drop its interrupt request; setting the priority interrupt assignment to zero assures that the device will not interrupt again.

After stopping the device, the interrupt routine should reactivate any process that may be waiting for the punch to finish. We decline to discuss the nature of this reactivation in any detail, as it would take us deeply into operating system areas that are not germane to the discussion of I/O devices. Finally, PTPSTP returns through LV6SAV to dismiss the interrupt.

```
;Service an interrupt from the paper tape punch:

PTPINT: JSR     LV6SAV          ;Save ACs, establish stack.
        SOSGE   PTPCNT          ;Decrement the byte count
        JRST    PTPSTP          ;Finished with transmission
        ILDB    1,PTPBYP        ;Get a byte for the punch.
        DATAO   PTP,1           ;Send data to punch. Clr DONE set BUSY
        RET                     ;Exit to LV6SAV. Restore ACs & Dismiss

PTPSTP: CONO    PTP,0           ;Deassign priority level; clear DONE
        ....                    ;Reactivate any waiting process
        RET                     ;Restore ACs.  Dismiss interrupt.
```

Next, we show the co–routine LV6SAV that saves the accumulators and sets up a stack. This is quite similar to the ENTLV3 routine that we examined in the discussion of the software interrupt system. LV6SAV differs from ENTLV3 only in the final instruction. The JEN instruction is a form of JRST. As with JRSTF, JEN restores the PC and flags from the specified address; LEVEL6 contains the PC and flags of the interrupted process. In addition, JEN dismisses the interrupt. By dismissing the interrupt JEN restores the processor to the priority level that was in progress prior to starting this interrupt. JEN also permits further interrupts to occur at the priority level from which it dismisses.[6]

[6]In the extended processor, the appropriate instruction is XJEN.

```
LV6SAV: 0                       ;Co-routine: save ACs & make stack
        MOVEM   17,LV6SAC+17    ;Save level 6 ACs.
        MOVEI   17,LV6SAC
        BLT     17,LV6SAC+16    ;Save the ACs
        MOVE    P,[IOWD 100,LV6PDL]
        CALL    @LV6SAV         ;Return to the caller of LV6SAV
        MOVSI   17,LV6SAC       ;Restore the accumulators belonging
        BLT     17,17           ; to the interrupted process.
        JEN     @LEVEL6         ;Dismiss interrupt.  Restore the PC &
                                ; flags of the interrupted process.
                                ; Restore PI system for another
                                ; level 6 interrupt.
```

Finally, we can discuss the structure of the routine that is the interface between a user program and the paper tape punch. Our description will be as simple as possible. Assume that the system has an internal buffer region associated with the paper tape punch. When the user program performs an operation such as BOUT or SOUT data is moved from the user program into that buffer. As long as space remains in the buffer, the user program continues to run.

When the internal buffer becomes completely full, TOPS–20 will start the punch. The punch service routine will eventually empty the buffer. Meanwhile, as the punch is working, the user process that fills the buffer is made to wait; presumably the system has other processes that can be run at this time. After the device finishes punching the buffer, the operating system will allow the user process to continue. TOPS–20 also starts the punch (and makes the user process wait) when a CLOSF JSYS is performed to signal the end of the output data.

The steps are indicated below. The one most relevant to I/O programming is the CONO. That instruction sets the priority level assignment (the PIA bits in the device status register) to 6 and sets the DONE flag in the punch interface. Because DONE is set and the PI level assignment is non–zero, the punch requests an interrupt. This interrupt will eventually cause the PTPINT routine to transmit the first byte from the buffer to the device.

```
PTPGO:  ...                     ;Set up the buffer byte count in PTPCNT
        ...                     ;Set the buffer byte pointer in PTPBYP
        ...                     ;Start the device:
        CONO    PTP,PTPDON!6    ;set DONE and PI level assignment
        ...                     ;Wait here until reactivated by PTPSTP
        RET
```

In a slightly more realistic example, the system would provide two buffer areas for the punch. When the user process fills the first, the system starts the punch. When the user process finishes filling the second buffer it will then be forced to wait. As soon as the punch finishes emptying the first buffer, the user program is allowed to continue until it fills the first buffer again. This additional complexity allows the program to overlap some of its computations while the punch is active.

32.3 Channel I/O

For fast I/O devices such as disks and tapes, it would be too wasteful for the processor to handle every word that is transferred. For such devices, an interface called a *channel* is provided. The channel is a separate processor that establishes a direct path between a device and memory. Instead of having the CPU execute instructions to handle each word of the transfer, the channel does most of the work independently.

In general terms, the program builds a *channel program* that describes the location of the data in memory, and sends commands to the channel and its connected device that describe the direction of the transfer (in or out) and the location of the data on the device. The channel does all of the work necessary to effect the transfer. When the transfer is completed, the channel interrupts the processor to tell it that the transfer is complete and that the channel is available for other work. Also, the channel can be programmed to continue performing further activities without the immediate intervention of the CPU.

A channel interfaces to peripheral devices by means of a control and data bus. In the DECSYSTEM–20, the bus that connects devices to channels is called the *Massbus*. The Massbus is a system–independent and device–independent data and control path. It provides a control protocol to enable data and commands to move from a computer system to a mass storage device, e.g., disk or tape, and for data and status to move from the device to the computer system. The Massbus is divided into two sections. The synchronous data bus is used for all data transfer operations between the device and the computer. An asynchronous control bus allows the Massbus controller to issue device commands and to read device conditions. The bus sections can be used independently: while a data transfer operation is in progress, a control operation can be initiated.

Up to eight complete channels may be present in the DECSYSTEM–2060. In the 2060, each channel consists of two parts: shared logic in the central processor's memory control unit (M box) and a *Massbus Controller* called an RH20. The set of RH20 Massbus controllers share the channel logic in the processor's memory control. Due to terminological inexactitude, the Massbus controller is sometimes called the channel, even though part of the channel logic resides in the M box portion of the KL10 processor.

The RH20 Massbus controller is the specific adaptor by which the DECSYSTEM–2060 is connected to Massbus devices. The program controls the RH20 by executing I/O instructions that pass commands to the RH20 via the E bus (the internal I/O bus). The RH20 attaches to the memory system via the C bus (channel bus). Of course, the RH20 connects to the device by the Massbus. The program distinguishes between the eight possible RH20 controllers by their device numbers. The first Massbus controller, number `0`, corresponding to channel `0`, is identified by the device number `540`. RH20 number `1` is device number `544`, etc.

A Massbus controller is the system–specific interface between the Massbus device, the processor, and the memory. By building Massbus controllers for different computer systems, DEC can attach identical Massbus devices to their various systems. Massbus controllers have been designed for such other systems as the PDP–11, the VAX–11 family, and the 2020.

The attachment of the channels and devices in the DECSYSTEM–2060 is depicted in Figure 32.1.

In order to understand how data transfers to or from the disk are performed we must learn something about five distinct areas that are germane to channel programming. Briefly, these are

Channel Command Words — The memory control portion of the channel is governed by a list of *Channel Command Words*. Basically, a channel command word (CCW) defines the location and extent of

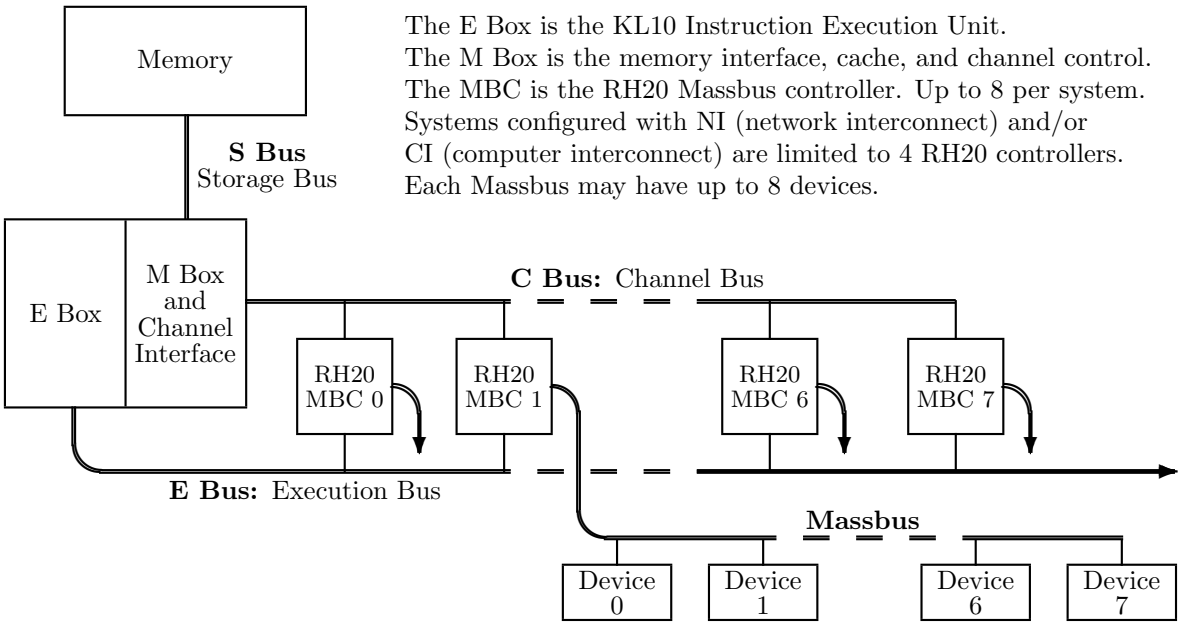


Figure 32.1: DECSYSTEM–2060 Channel and Disk Attachment

a data region in memory.

Channel Reset and Status Logout Area — Each channel has a 4–word area in the executive process table (EPT). This area contains an initial channel command word and two words of channel status; the fourth word is not presently used. When the channel is initialized, it reads its first command word from this area. When the channel detects an error, or on command from the program, two words of channel status are stored in this area.

Device Registers (External Registers) — The Massbus device contains several registers that are used to control the device and to interrogate its status. Non–data transfer commands are issued to the device to prepare it for data transfer operations. In the example that follows, we will discuss the RP06 disk drive. Most Massbus disk drives are operationally similar to the RP06, except in such details as the number of cylinders, number of tracks, and number of sectors.

Massbus Controller Internal Registers — The Massbus controller contains two registers that are of particular interest to us. One register helps define the location of the data on the device. The second register specifies the command to the Massbus device. These registers control a data transfer operation. After the program has prepared the device by issuing appropriate non–data transfer commands, the program will load these two registers to initiate a data transfer operation.

Massbus Controller Status and Control — It will be necessary for us to know the meaning of some of the flags present in the RH20 status register that we read via `CONI`. Also, we must know how to change the state of the RH20 by the bits we can set with `CONO`.

Before going into further details, it is appropriate to give an overview of channel operation. In order to perform a read operation from the disk, the following steps are necessary:

1. Determine the physical channel and unit where the data exists. Examine the status of that channel and unit to avoid interfering with any operation that might already be in progress.
2. Select the physical address where the data will be placed in memory. Build an appropriate list of channel command words that describes the size and location of the data buffer. Put a channel command word jump to this CCW list in the first word of the logout area corresponding to the channel being used.
3. Compute the cylinder number of the logical record where the data is located. Load the desired cylinder number into the selected drive's cylinder address register. Issue a seek command; wait for it to complete.
4. Write the track and sector number in the RH20 block address register. Send a read command to the RH20's transfer control register.
5. Wait for the transfer to finish or for a timeout. Examine the ending status of the Massbus controller and the disk to be certain that no errors have occurred.

We will now examine each of these topics in greater detail.

32.3.1 Channel Command Words

A list of *channel command words* or CCWs supply the memory addresses and word counts associated with any data transfer operation. Channel commands are fetched from the address contained in a register called the *Command List Pointer*. When starting a data transfer operation, the program can direct the channel to reset the command list pointer to the first word (word 0) of that channel's logout area (see below). This first word is typically a *channel jump* command that instructs the channel to continue fetching command words from another address.

As shown in Figure 32.2, there are three basic types of channel command words. They are data transfer, jump, and halt. A data transfer command initiates the M box side of a data transfer operation; a data transfer command supplies the data buffer address and word count that describe the size and position of the data area in memory. Each time a data transfer command is fetched by the channel, the channel will increment its command list pointer.

A channel jump command loads a new address into the command list pointer. This allows an unconditional branch during the execution of a command list. For example, the processor normally stores a jump in word 0 of the channel's logout area to cause the channel to continue fetching commands from a new address. Addresses contained in the command list pointer are interpreted as physical (not virtual) addresses; the only exception to this is that the initial command word is fetched from the executive process table.

The halt command causes channel operations to stop. The next sequential command word will not be fetched. (A data transfer command can also be made to halt channel activity at its conclusion.)

The halt command loads a new address into the command list pointer; that address will be the starting address of the next channel command list, unless the RH20 is told to reset the command list pointer at the start of the next transfer.

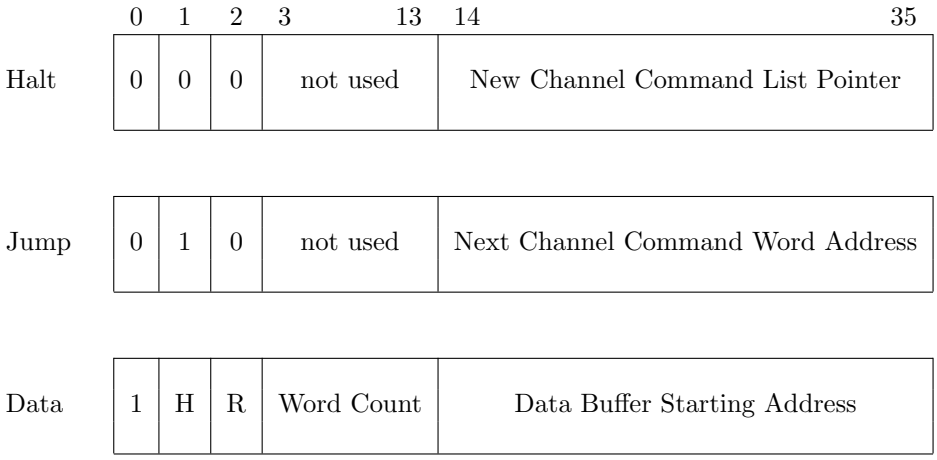


Figure 32.2: Channel Command Word Format

When bit 1 in a data transfer command (the "H" in Figure 32.2) is set to one, the channel will halt after completing this command. When bit 2 in a data transfer command ("R" in the figure) is set to one, the channel will transfer data in the reverse direction; during a reverse data operation (appropriate only to tapes) the data buffer address is decremented as the transfer proceeds. In a data transfer command, a zero as the data buffer starting address signifies a data skip operation. In a read operation, a data skip will cause the channel to discard the specified number of data words from the device. In a write operation, a data skip causes the channel to supply full words of block fill data from EPT locations `60` through `63` to write the remainder of a drive's data block.

32.3.2 Channel Reset and Status Logout Area

Each channel is assigned a 4–word area in the executive process table (EPT). This region is called the *channel reset and status logout area*, or, more briefly, the *logout area*. For channel number n, $0 \leq n \leq 7$, the logout area starts at word $4 \times n$ in the executive process table. The arrangement of the channel logout areas in the EPT is shown in Figure 32.3.

For each channel, the first word of the logout area contains the initial CCW. After the channel performs its *reset command list pointer* function, it will fetch the next CCW from this area.

Two words of status, show in Figure 32.4, are stored when the channel encounters an error condition or when it completes a transfer in which it was told to store status. The first status word reflects some channel conditions and gives the address from which the next CCW will be fetched. The second status word is simply the current (i.e., the most recently executed) CCW, with the word count and data buffer adjusted to reflect the number of words actually transferred.

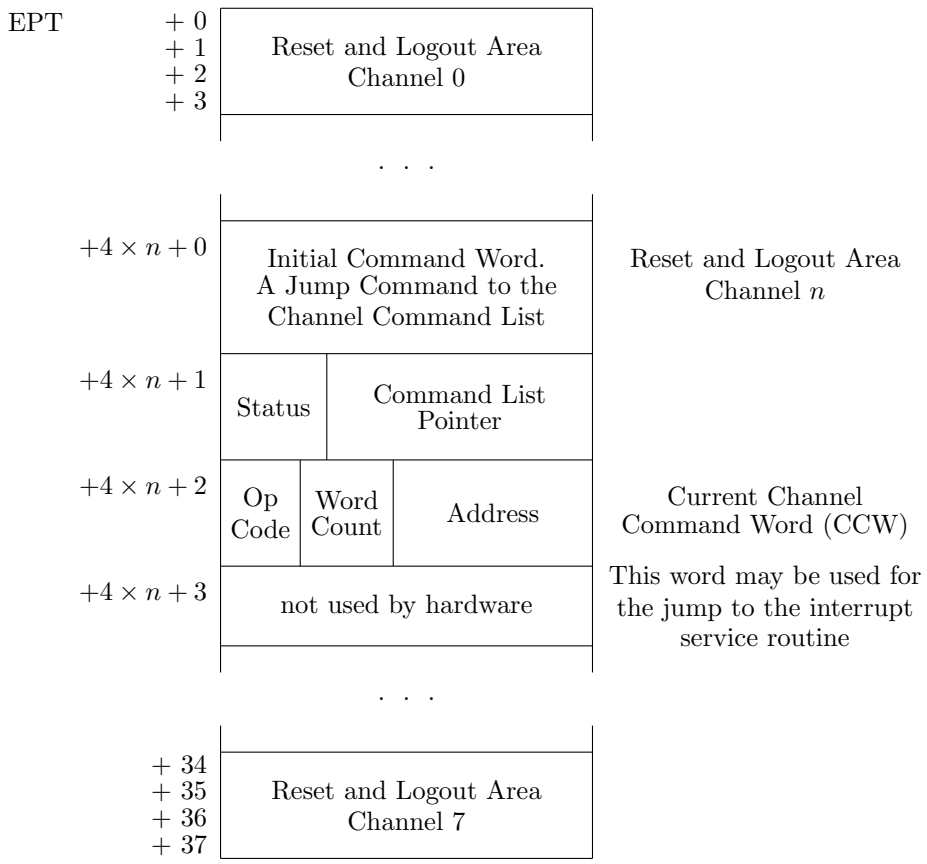
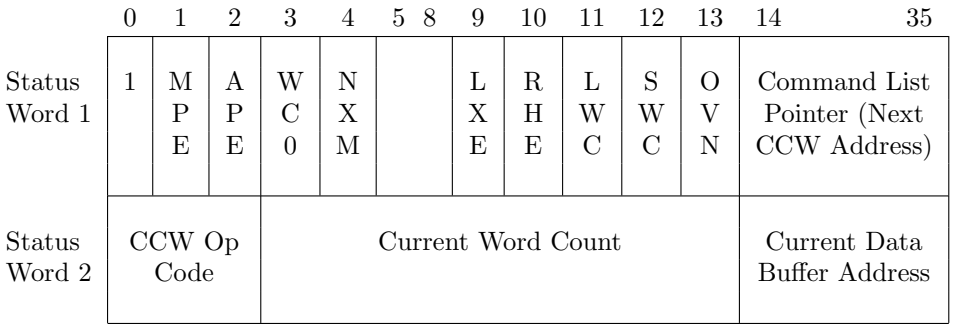


Figure 32.3: Channel Reset and Logout Area



`MPE` Memory Parity Error detected by channel.

`APE` A *zero* signifies that an Address bus Parity Error was detected.

`WCO` Word Count was not zero when status was stored.

`NXM` Non–eXistent Memory. The channel referenced a location that did not respond.

`LXE` Last Transfer Error. An error occurred after the RH terminated a transfer.

`RHE` RH20 Error. A hardware failure.

`LWC` Long Word Count. The RH20 finished, but the CCW word count was not exhausted.

`SWC` Short Word Count. The word count ran out before the RH20 finished.

`OVN` Overrun. The device sent or demanded data faster than the channel could accept or supply it.

Figure 32.4: Channel Status Words and Flags

32.3.3 Massbus Controller and Device Registers

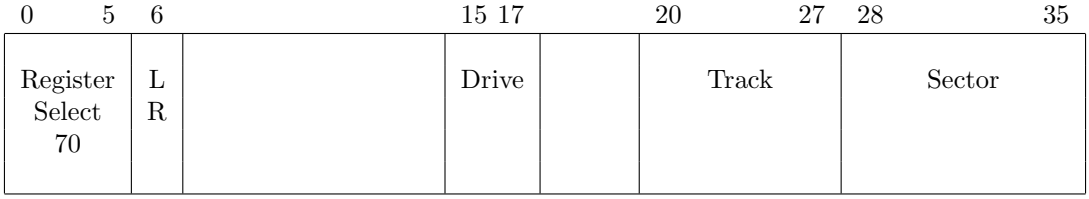
The RH20 Massbus controller and the Massbus devices that are attached to it appear to the programmer as a collection of device registers. The basic programmed operations that are supported by the RH20 are the `DATAO` and `DATAI` instructions. The Massbus controller contains eight *internal* registers that control its functions. Each of the attached Massbus devices contains its own collection of registers; from the point of view of the RH20, the device registers are called *external* registers.

Before reading a register we must first select it. The selection is accomplished by a `DATAO` instruction directed to the RH20; the data that is sent contains a register selection field. The register number alone is sufficient to select an internal register; an external register is selected by supplying both a register selection field and a drive selection field that specifies which unit is being addressed. The next `DATAI` directed to this RH20 will read the selected register. To write a register, a `DATAO` is performed in which the data sent to the RH20 selects a particular register and contains the *Load Register* (`LR`) bit; a subsequent `DATAI` will read the new contents of the selected register.

32.3.3.1 Massbus Controller Internal Registers

The RH20 contains eight internal registers; but only two of them are relevant to the example that we will present. The registers that we will need are the *Secondary Block Address Register* (`SBAR`) and the *Secondary Transfer Control Register* (`STCR`). This register pair is used in any data transfer operation. The program should load the `SBAR` first. When the program loads the `STCR`, the RH20 will start the data transfer, provided that any previous data transfer operation has been finished. The RH20 starts this transfer by sending the data in this register pair to a drive. The `SBAR` and `STCR` registers are depicted in Figure 32.5.

Secondary Block Address Register



Secondary Transfer Control Register

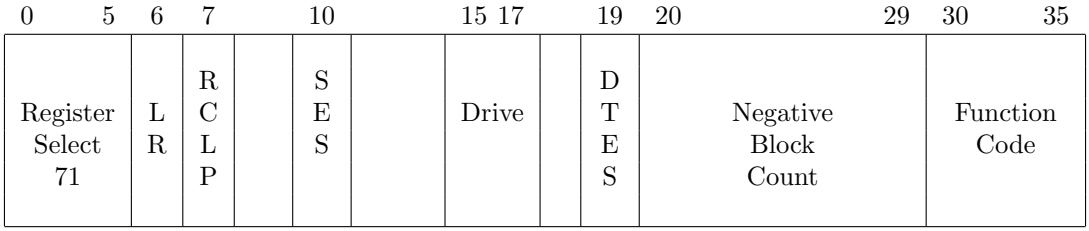


Figure 32.5: SBAR and STCR

When the load register bit (LR) is one, the register will be loaded from the other data fields.

The block address field in the SBAR specifies the starting track (head) and sector number for the transfer; when the transfer is started, the RH20 will send bits 20:35 of the block address register to the selected drive's block address register.

The function code in the STCR is a 6–bit field that will be sent to the selected drive's command register. When the drive receives this command, it commences a data transfer operation.

The other bits in the STCR are not passed to the drive. Instead, these direct special initialization, termination, and error handling in the Massbus controller and the M box.

RCLP means *Reset Command List Pointer*. When this bit is set to one, the M box channel will completely reset itself before starting the data transfer. The first channel command will be fetched from word 0 of this channel's command and logout area in the EPT.

When set to one, the *Store Ending Status* (SES) bit causes the M box to store the channel status at the end of a data transfer. Status is stored in the channel's logout area in the EPT. When a transfer error occurs, ending status is stored regardless of the state of this flag.

The DTES bit forces the channel to complete a transfer despite data bus parity error or drive exceptions. This bit would be asserted during an attempt to retry a failing record; it would allow as much data as possible to be transferred despite a drive exception.

In order to synchronize the M–box channel, the RH20, and the drive, the programmer must send the desired block address to the SBAR and the transfer command to the STCR in the RH20. After the program loads the STCR, as soon as there is no data transfer in progress, the RH20 will perform its *command file transfer* function. The RH20 renames the SBAR and STCR to be the *primary* register set; then it transfers the contents of the new primary block address register to the drive, followed by the drive command from the new primary transfer control register. Meanwhile, the RH20 has told the M box to commence executing the list of channel command words. Presumably the M box will be ready to accept (or supply) data when the the drive begins to supply (or demand) data.

The command file transfer operation starts a data transfer. Also it provides a new set of secondary registers that can be loaded by the program. These registers provide a *backup command* that will commence a second transfer as soon as the one now in progress is finished. Thus, the channel can start on another transfer without immediate intervention by the processor; this ability allows greater disk throughput than would otherwise be possible.

The program should not load a non–data transfer command into the STCR; nor should it load a data transfer command directly into a drive's command register.

32.3.3.2 Massbus Controller Status Register

The CONI instruction (and CONSZ and CONSO) can interrogate the status of the channel. As befits a complex device, there are many status bits. The readable status bits are depicted in Figure 32.6. The CONO bits are shown in Figure 32.7; the CONO instruction allows certain conditions to be set.

32.3.3.3 Device Registers

The RP06 contains sixteen registers. These external registers are accessed by DATAO and DATAI instructions that are directed at the Massbus controller to which the disk drive is connected. Each of the RP06 registers contains only sixteen bits, for compatibility with smaller computers. To write a drive register, perform a DATAO function selecting the RH20 to which the drive is connected. The

18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33 35
DPE	DEE	LWC	SWC	CE	DRE	RAE	CNR	OVN	MBE	ATN	SCRF	ATE	PCRF	CMD	Priority Interrupt Assignment

DPE Data Bus Parity Error. Bad parity was detected on the Massbus data lines.

DEE Drive Exception Error. An error was detected by the drive during a data transfer operation.

LWC Long Word Count. During a write transfer the RH20 stopped transmitting data before the M box channel reached its final word.

SWC Short Word Count. During a write transfer the RH20 asked for more words than the M box channel was told to supply.

CE Channel Error. M box detected an error during a data transfer operation.

DRE Drive Response Error. No drive responded when the RH20 tried to load a data transfer command into its command register.

RAE Register Access Error. An error, either no response or control bus parity error, was detected while the RH20 was attempting to access an external register. RAE requests an interrupt. Until this bit is cleared, all further `DATAO` operations will be refused.

CNR Channel Ready. The channel is ready to start a data transfer.

OVN Data Overrun Error. The device supplied (read) or demanded (write) data faster than the M box channel was able to accept or supply it.

MBE Massbus Enabled. The Massbus transmitters in the RH20 are enabled.

ATN Attention. A Massbus device will set Attention when it completes some non–data operations, and when drive errors are detected.

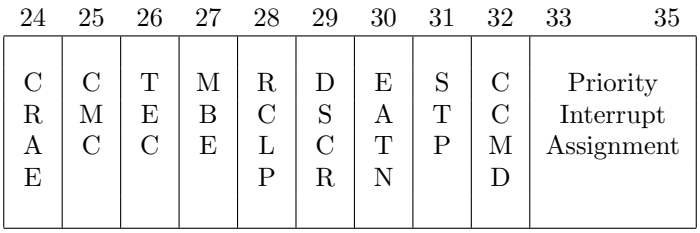
SCRF Secondary Command Register Full. Indicates that the `STCR` is full, i.e., a transfer command is pending.

ATE Attention Interrupt Enabled

PCRF Primary Command Register Full

CMD Command Done. The previously requested data transfer command has finished. When this condition is true an interrupt is requested.

Figure 32.6: RH20 Status Register (CONI)



`CRAE` Clear Register Access Error. Resets `CONI` bit 24.

`CMC` Clear RH and Devices. Perform power–on reset to the RH20 and all attached devices.

`TEC` Transfer Error Clear. Clear `CONI` bits 18:23 and 26.

`MBE` Enable Massbus. Enable the RH20 to transmit commands and data on the Massbus. Normally this is turned on by the program; it would be off during some diagnostic procedures that test the RH20 without affecting Massbus devices.

`RCLP` Reset Channel and Command List Pointer. Initializes the M–box portion of the channel. The next CCW will be fetched from the channel's Logout area in the EPT.

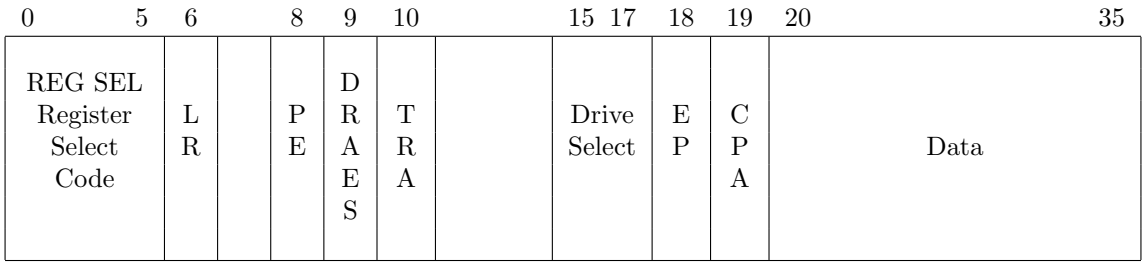
`DSCR` Delete Secondary Command Register. When a transfer error occurs, the channel will not proceed to any backup command until the error status is cleared. If the program wants to abort the pending backup command, it may assert this bit.

`EATN` Enable Attention Interrupt. Allows the attention signal on the Massbus to generate an interrupt.

`STP` Stop Transfer. Terminate the current data transfer. This should be used only when a transfer appears to be hung.

`CCMD` Clear Command Done. Resets `CONI` bit 32.

Figure 32.7: RH20 Command Flags (CONO)



`REG SEL` This field selects a particular external register in the addressed device. The meaning of the various register selections will be summarized in the table of RP06 drive registers.

`LR` Load Register. Write new data into the selected drive register.

`PE` Control bus Parity Error while accessing this register (Read Only)

`DRAES` Disable Register Access Error indication for this access; if RAE is already set, this bit being set forces the register access attempt anyway.

`TRA` TRAnsfer received: a drive responded when the RH20 accessed it. (Read Only)

`EP` Write Even Parity on the control bus. Used in diagnostics to test the drive's parity error detection circuitry. (Write Only)

`CPA` Control bus Parity bit: the state of the parity bit received on the Massbus (Read Only).

Figure 32.8: External Register DATAO / DATAI Word Format

36–bit data word should contain register select and drive select codes according to the format shown in Figure 32.8.

The RP06 has sixteen registers that can be accessed via the `DATAI` and `DATAO` instructions. These registers are numbered in octal from 0 to 17. The function of each register is discussed below.

00 Control. Bits 30:35 are the drive command field, in which the program may write any non–data transfer command. When the RH20 initiates a transfer operation, it will write a data transfer command into this register from its transfer control register. The various function codes and their meanings are listed in the table of RP06 drive commands. On a `DATAI` operation, a one in bit 24 means that this drive is available to this channel; a dual ported drive would be unavailable while it is in use by another channel.

01 Status. A read–only register that reports the status of the drive. See Figure 32.9.

02 Error Register 1. This is the first of three error registers. This register generally reports error conditions in and detected by the drive interface, as distinct from the disk drive itself.

03 Maintenance. This is a special command and status register used to force the interface and drive to perform diagnostic functions.

04 Attention Summary. This register is common to all the drives that are connected to the same Massbus. One bit is present in the register for each of the eight physical units that might be

connected. A bit is on in this register for each unit on this channel that is requesting attention. The drive select field is ignored when this register is accessed: all drives on the Massbus respond, but each responds with a unique bit. Drive 0 responds on bit 35; drive 1 on bit 34, etc.

05 Block Address. When the RH20 initiates a data transfer, before loading the command function into the control register, it first writes its block address register into this drive register. The disk expects to find the desired sector in bits 31:35 and the track number in bits 23:27.

06 Drive Type. This read–only register reports which type of Massbus device this unit is. The program reads this register to determine whether this is a tape, a disk, a fixed–head disk, etc. By table lookup of the nine–bit drive type field, the program can identify other characteristics of the drive.

07 Current Block Address. A read–only register that reports the current sector number. This register is included to allow the software to optimize disk access time by minimizing rotational delays.

10 Error Register 2. The second error register. In contrast to error register 1, this register and error register 3 report conditions in the disk drive itself, rather than conditions in the drive interface.

11 Offset. "Offset" means moving the heads away from the centerline of the track to adjust for misaligned heads. Offset is sometimes used as an error recovery technique. The program writes the amount and direction of offset in this register and executes an Offset command, and then retries the read. One unit of offset is 25 micro–inches. Various other control flags associated with error recovery are present in this register.

12 Cylinder Address. Loaded by the program prior to giving a seek command. The desired cylinder number is placed in bits 26:35.

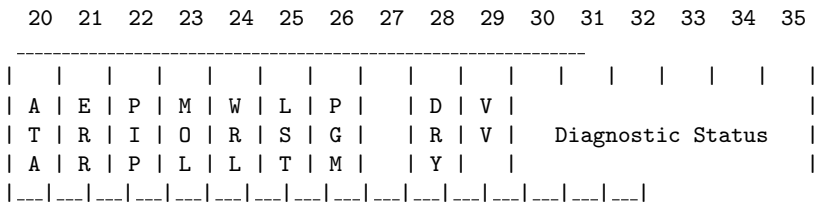
13 Current Cylinder Address. The drive reports the current position of the head assembly. (Read Only)

14 Serial Number. This read–only register reports the drive serial number. It is distinguished from the Drive Type register in that the serial number reports which specific unit of a given type this is. For error logging purposes, it is wise to identify a failing drive by some means other than channel and unit, since unit identification can be switched quite easily by an operator. Thus, reporting by drive serial number uniquely identifies the device, despite changes in system configuration.

15 Error Register 3. This is the third error register. Complicated devices potentially can have many different things go wrong.

16 ECC Position. This read–only register contains the bit position at which to apply the error correction pattern. The contents of this register are valid only when the drive error register indicates a correctable data error.

17 ECC Pattern. When a correctable error occurs, this pattern specifies which bits in memory are wrong. The ECC position specifies the location where the error pattern should be applied. The program should repair the data in memory by computing the exclusive OR of this pattern and the memory data. This result should be stored in memory to replace the data that was read incorrectly.



ATA ATtention Active. Set when the drive has completed a positioning command and is asserting attention.

ERR The composite ERRor indication. Some error has occurred; the three drive error registers should be examined to determine what caused the problem.

PIP Positioning In Progress. The drive is currently performing a positioning command.

MOL Medium On–Line. The drive is powered up; the disk is spinning and the heads are loaded.

WRL The pack is WRite–Locked.

LST Last Sector Transferred. This is set by the drive when the last addressable sector on the disk has been transferred.

PGM ProGraMmable. Set when a dual–port drive has its port select switch in the neutral position.

DRY Drive ReadY. At the completion of every command, the drive sets this bit.

VV Volume Valid. When a disk first spins up, this bit is zero. The program should perform a *Pack Acknowledge* or *Read–in Preset* command to indicate that it knows the pack may have been changed.

Figure 32.9: RP06 Drive Status Register

32.3.3.4 RP06 Commands

Commands are issued to the RP06 by sending a 6–bit code to the drive command register. The least significant bit of each of these codes is one, called the Go bit. Note that only non–data transfer commands can be sent directly to the drive. Data transfer commands must be sent to the internal transfer control register of the RH20, as we have already described.

`01` No Operation

`03` Unload. The read/write heads are retracted and the disk is stopped. The drive is set to standby status; it will require operator intervention to make it ready again.

`05` Seek. Move the read/write heads to the cylinder specified in the cylinder address register. When the heads reach their new position, the desired cylinder address register is copied to the current cylinder address register, as they are now both the same.

`07` Recalibrate. Seek to cylinder zero. Set the desired cylinder address to zero.

`11` Drive Clear. Reset various drive internal states. Make the drive receptive to further commands. Clears all errors and attention.

`13` Release. Perform a drive clear command and allow the drive to be used by the other port.

`15` Offset. Perform the offset seek in the amount specified by the offset register.

`17` Return to Centerline. Return to nominal track centerline to conclude the use of offset for error recovery.

`21` Readin Preset. Set volume valid; clear any offset or error–recovery modes; seek to cylinder 0; prepare to read track 0 sector 0. This command is used to bootstrap from the device.

`23` Pack Acknowledge. This command sets the volume valid bit. This command must be issued after a drive has come on–line, before any data transfer or positioning commands can be performed. This interlock is primarily intended to prevent the software from missing any pack changes.

`31` Search. Wait until the disk spins to the desired sector and then interrupt the program. This command is used to optimize channel utilization when several disks are active on one channel.

`61` Write Data. Accept data from the Massbus and write it at the next available sector.

`63` Write Header & Data. Similar to Write Data, but in addition to the data, formatting information is written too. This operation is performed when a new disk pack is being initialized for use on the system.

`71` Read Data. Starting at the beginning of the next sector, read data from the disk and transmit it through the Massbus.

`73` Read Header & Data. Similar to Read Data, but also reads the formatting information as written by Write Header & Data.

Non–data commands (e.g., seek, recalibrate) can be loaded directly into the drive's command register by executing a `DATAO` in which the data word specifies the `LR` bit and whose register and drive fields select the command register of an extant drive. Data transfer commands must be sent through the `STCR` as we have described.

32.4 Example 21 —Bare Machine Channel Input

With these preliminaries in mind, we now present a code fragment that has been rewritten from the BOOT program. BOOT runs on what we call the *bare machine*, i.e., on the computer itself, without the benefit of an operating system. BOOT is used in the DECSYSTEM–20 to read an executable core image file into memory; usually that file is the TOPS–20 operating system itself. BOOT is a realistic example of what a program has to do in order to read data from a file–structured disk. However, it is not nearly so complicated as an operating system: BOOT runs without using the priority interrupt system; it simply waits for its I/O operations to complete.

The purpose of the following fragment is quite simple. In our discussion of the file system, we mentioned that a table is needed to translate from a logical unit number within a structure to a physical channel and unit. Given a structure name in the word `STRNAM`, this fragment builds such a table in `DSKTAB`.

The accumulator conventions in this fragment are those that have been adopted for operating system programming. We hope the departure from our usual naming conventions does not cause the reader undue distress. We will present the text of this lengthy example now; the explanation will follow.

Example 21 Channel I/O

```
        SUBTTL  FDSK - Locate the physical units for a given structure.

;General assumptions
; Register Use
;       P1      Holds the physical channel number 0 <= channel < MAXCHN (8)
;       P2      Holds the physical unit number  0 <= unit < MAXDRV (8)
;
; Cells:
;       DIORG   Returned with the disk address of the Root-dir index page
;       STRNAM  Call with the sixbit name of the sought-for structure
;       ICCW    Set to a Channel Command Word for a 1000 word transfer
;               to the page starting at A%FPO.  ICCW+1 will be set to
;               zero, a Halt CCW.
;       MAXUNI  Maximum unit number of the structure that was found.
;               (This is one less than the number of units in structure)
;       DSKTYP  Initially zero, the Drive Type of the units in this str.
;       NUMCYL  Number of cylinders/unit for this drive type
```

```
; Arrays
;       A%FP0   Virtual address of one page to be used as a data buffer
;               The home block is read into the second record (+200) of
;               this page. The following definitions of home block words
;               are needed:
                HOMNAM==A%FP0+200       ;SIXBIT/HOM/
                HOMSNM==HOMNAM+3        ;Structure name
                HOMLUN==HOMNAM+4        ;Logical Unit Number
                HOMRXB==HOMNAM+10       ;Root Directory Index Page
                HOMCOD==HOMNAM+176      ;Home block constant = 707070
                CODHOM==707070          ;Home block constant
;       A%EPT   Virtual address of the Executive Process Table
;       DSKTAB  Table of words indexed by the logical unit number,
;               each word contains:
;               400000!<Physical Channel>,,<Physical Unit>

; Constants
        ICA=0           ;Offset from EPT origin to channel 0's Logout area
        ENTFLG=400000   ;Flag in left half of DSKTAB table used to
                        ;distinguish channel 0 unit 0 from a missing entry.
        MAXCHN==8       ;maximum number of channels
        MAXDRV==8       ;maximum number of drives per channel

;
;Device Definitions
;       RH20
                RH0==540                ;device number
;       RH20 CONO/CONI
                .RHRAE==1B24            ;Clear RAE (In CONI, Register Access Err)
                .RHMBR==1B25            ;Clear RH20 & attached Devices
                .RHCTE==1B26            ;Clear transfer error
                .RHMBE==1B27            ;Enable Massbus Transmitters (also CONI)
                .RHSTP==1B31            ;Stop a hung transfer
                .RHDON==1B32            ;Clear Done (In CONI, Transfer Done)
                RHERR==775120           ;All the RH20 error bits (in CONI)
                                        ;Data Bus Parity; Drive Exception
                                        ;Long WC; Short Wc; Channel Error
                                        ;Drive Response Error; RAE; Overrun
                                        ;and Either Command Buffer Full
;       RH20 DATAO/DATAI
                LR==1B6                 ;Load Register Bit
```

```
;       RP06 Registers
                R4%CSR==0B5             ;Drive Command Register
;               RP06 Drive Commands
                        R4%CRC==07      ;Recalibrate
                        R4%RIP==21      ;Read-in Preset
                        R4%CPA==23      ;Pack Acknowledge
                        R4%CRD==71      ;Read Data
                R4%DSR==1B5             ;Drive Status Register
                        .RPERR==1B21    ;Composite error
                        .RPMOL==1B23    ;Medium On-Line
                        .RPDRY==1B28    ;Drive Ready
                R4%ATN==4B5             ;Attention Summary Register
                R4%DST==5B5             ;Drive Sector & Track
                R4%DTR==6B5             ;Drive Type Register
                        R4%TYP=777      ;Mask for drive type bits only
                .RHSBR==70B5            ;RH20 Secondary Block Address Reg
                .RHSTR==71B5            ;RH20 Secondary Transfer Control Reg
                        RCLP==1B7       ;Reset Command List Pointer
                        STLW==1B10      ;Store Ending Status


FDSK:   SETZB   P1,DIORG                ;Channel 0 to P1
                                        ;Root-Dir XB addr unknown
        SETOM   MAXUNI                  ;No highest unit in structure
        MAP     T1,A%FP0                ;Convert addr of FP0 to physical
        TLZ     T1,777760               ;Keep the 22 physical addr bits
        TLO     T1,(<1B0+1B1+<1000B13>>);CCW: Data, Halt, Word Count
        MOVEM   T1,ICCW                 ;Store first data CCW

;At FDSK1, examine the channel whose number is in P1
FDSK1:  MOVEI   T1,.RHMBR               ;RH20 Reset function
        CALL    RHCONO                  ;Reset the RH20
        MOVEI   T1,.RHMBE               ;RH20 Enable function
        CALL    RHCONO                  ;Enable Massbus transmitters
        CALL    RHCONI                  ;Now read RH CONI, result to T1
        TRNN    T1,.RHMBE               ;Did the enable bit come back?
        JRST    FDSK4                   ;No. No bit means no channel

;This channel exists.
        MAP     T1,ICCW                 ;Get physical address of ICCW
        TLZ     T1,777760               ;Keep 22 phys addr bits
        TXO     T1,1B1                  ;Make a JUMP CCW
        MOVE    T2,P1                   ;Get the current channel number
        LSH     T2,2                    ;Shift it to Channel*4
        MOVEM   T1,A%EPT+ICA(T2)        ;Store initial CCW for channel
```

```
;Scan to see what physical units are present.
        MOVEI   P2,0                    ;Start at unit 0
;Start on a Unit. Unit number in P2
FDSK2:  MOVX    T1,R4%DTR               ;Read the Drive Type register
        CALL    RHREAD
        MOVE    T2,T1                   ;Copy Drive Type to T2
        CALL    RHCONI                  ;Check for Register Access Error
        TRNN    T1,.RHRAE               ;Test CONI bits for RAE
        JRST    FDSK2A                  ;All is well: a drive responded
        MOVEI   T1,.RHRAE!.RHMBE        ;Must clear RAE. Set ENABLE.
        CALL    RHCONO                  ;CONO to clear RAE & set ENABLE
        JRST    FDSK3                   ;Drive doesn't exist. Try next.

;A unit is present.  See what kind of drive it is.
; GOTDRV does all the real work
FDSK2A: CALL    GOTDRV                  ;See what we have here
FDSK3:  CAIGE   P2,MAXDRV-1             ;Done with all units yet?
        AOJA    P2,FDSK2                ;No, go do next unit now.
FDSK4:  CAIGE   P1,MAXCHN-1             ;Checked all channels yet?
        AOJA    P1,FDSK1                ;No, advance to next channel

;Now, see if we found a consistent structure.
        SKIPGE  T1,MAXUNI               ;Now, check for full structure
        JRST    MISSTR                  ;Structure not found
CHKSTR: SKIPN   DSKTAB(T1)              ;Any physical addr for this LUN?
        JRST    MISUNI                  ;Bad structure: missing unit
        SOJGE   T1,CHKSTR               ;Loop thru all units
        MOVE    T2,DSKTYP               ;Get the type of this disk
        MOVEI   T1,N.CLP4               ;Get # of cylinders/unit for 04
        CAIE    T2,.R4TYP               ;RP04?
        MOVEI   T1,N.CLP6               ;No.  Assume RP06. Get cyl/unit
        CAIN    T2,.R3TYP               ;RM03?
        MOVEI   T1,N.CLP3               ;Yes
        CAIN    T2,.R7TY1               ;RP07?
        MOVEI   T1,N.CLP7               ;Yes
        MOVEM   T1,NUMCYL               ;Save result
        RET
```

```
        SUBTTL  GOTDRV Examine the Drive and Home Block to see if we want it

;Initially DSKTYP is zero.  When we find the first disk, we set DSKTYP
;to the negative of the drive type number.  If we like the home blocks,
;we set DSKTYP to positive and thereafter, we look at only physical
;units that are similar.
GOTDRV: ANDI    T2,R4%TYP               ;Keep only the device type code
        CAIN    T2,.R5TYP               ;Is this an RP05
        MOVEI   T2,.R4TYP               ;Yes, make it look like an RP04
        CAIN    T2,.R7TY2               ;Alternate kind of RP07?
        MOVEI   T2,.R7TY1               ;Make it the usual kind of RP07
        SKIPG   T1,DSKTYP               ;Are we looking for a specific disk?
        JRST    GOTDV3                  ;Not yet, check if this is allowable
        CAMN    T2,T1                   ;Yes.  Is this the same as we want?
        JRST    GOTDV0                  ;Same as we want. Read its home block
        RET                             ;No.  Don't use this one.


;The desired Drive Type has not been selected yet.
;Check type against all known disk types:

GOTDV3: CAIE    T2,.R7TY1               ;RP07?
        CAIN    T2,.R3TYP               ;or RM03?
        JRST    GOTDV4                  ;Yes.
        CAIE    T2,.R6TYP               ;RP06?
        CAIN    T2,.R4TYP               ;or RP04?
        JRST    GOTDV4                  ;Yes. Save type (negated)
        RET                             ;This is no kind of disk for us

GOTDV4: MOVNM   T2,DSKTYP               ;Save negative drive type
                                        ;in case the home blocks are bad
GOTDV0: MOVE    T1,[R4%CSR!LR!R4%CPA]   ;Send Pack Acknowledge.
        CALL    RHWRIT                  ;Do DATAO.  Sets Volume Valid
        MOVX    T1,R4%DSR               ;Read device status register
        CALL    RHREAD                  ;Via DATAO/DATAI
        TRNN    T1,.RPMOL               ;Is Disk Pack On-Line?
        RET                             ;No. Forget it.
        SETZM   A%FP0                   ;Clear data page
        MOVE    T1,[A%FP0,,A%FP0+1]     ;Set up for the BLT
        BLT     T1,A%FP0+777            ;Zero data page before the Read
        CALL    RD0                     ;Read disk page 0 (records 0,1,2,3)
        CALL    CHKHOM                  ;Check the home block
        JRST    GOTDV2      ;no skip    ;Bad.  Try the Backup home block
        RET                 ;one skip   ;Good block, but wrong Structure
        MOVMS   DSKTYP      ;two skips  ;Set DSKTYP Positive.
        RET                             ;Check next unit.
```

```
;Primary Home Block was bad.  Try reading the Backup home block.
;CHKHOM expects to see the home block in record 1 (words 200:377) of
;the page so we position the backup home block in record 1 of the page,
;by starting the transfer at record 11.  Then record 12 will wind up at
;words 200:377
GOTDV2: CALL    RD11                    ;Read records 11,12,13,14
        CALL    CHKHOM                  ;Check the backup home block
        RET                ;no skip     ;Error
        RET                ;one skip    ;Wrong Structure
        MOVMS   DSKTYP     ;two skips   ;Set DSKTYP Positive.
        RET                             ;Check next unit.

;RH20 Register Read and Write, CONO and CONI.   Call with P1 = RH #

;An RH20 register is read by DATAO'ing the specified register number to
;the RH and then DATAI'ing the RH.  An RH20 register is written by
;DATAO'ing the specified register number, the LR (load register) bit,
;and the remainder of the data to the RH.  After writing the register,
;a DATAI is performed to read the new contents.

RHREAD: TXZ     T1,LR                   ;Clear LR bit
RHWRIT: TSO     T1,P2                   ;Insert Drive Number in left half
        CALL    XIO1                    ;Set IO Device Field & Do DATAO
        DATAO   .-.,T1                  ;Send register select & data to device
        CALL    XIO1                    ;Set IO Device Field & Do DATAI
        DATAI   .-.,T1                  ;Read selected register
        RET

RHCONI: CALL    XIO1                    ;Set IO device and do CONI
        CONI    .-.,T1                  ;Read device status
        RET

RHCONO: CALL    XIO1                    ;Set IO device and do CONO
        CONO    .-.,(T1)                ;Set device conditions
        RET
```

```
;IOXCT Subroutine
;   Call with F = device code (7-bit I/O device number)
;   next instruction should be:   IOOP 0,Address
;
;   Effect of routine is to execute the specified I/O instruction as
;   though its device code field were set to the contents of F.
;   Note that F is NOT the assembler's device code! Just the 7-bit number
;
;XIO1:  Call with P1 = channel number, otherwise, as IOXCT

XIO1:   MOVEI   F,<RH0/4>(P1)           ;Produce I/O device code
IOXCT:  PUSH    P,@(P)                  ;Fetch argument, to top of stack
        AOS     -1(P)                   ;Skip once to skip the argument,
        DPB     F,[POINT 7,(P),9]       ;Store IO Device Number in instruction
        XCT     (P)                     ;Execute the target instruction
        CAIA                            ;Target didn't skip.
        AOS     -1(P)                   ;Target skipped.  Pass the Skip upwards
        ADJSP   P,-1                    ;Discard the stacked instruction
        RET

;RD0:  Routine to read page 0 (Records 0,1*,2,3) *= home block
;RD11: Routine to read backup home block (Records 11,12*,13,14)
;
; Accepts in P1/ Channel number
;            P2/ Unit number
;                CALL RD0

RD11:   SKIPA   P4,[.RHSBR!LR!0B27!11B35] ;Desired Sector and track
RD0:    MOVE    P4,[.RHSBR!LR!0B27!0B35]  ;For RH20 SBAR
        MOVEI   T3,NRETRY                 ;Initialize Retry counter
RD00:   MOVE    T1,[R4%CSR!LR!R4%CRC]     ;Recalibrate Function
        CALL    RHWRIT                    ;Execute Recalibrate
RD0A:   MOVX    T1,R4%DSR                 ;Select unit status register
        CALL    RHREAD                    ;Get drive status
        TXNN    T1,.RPDRY                 ;Is drive ready?
        JRST    RD0A                      ;No, Wait till recalibrate done
        MOVE    T1,[R4%CSR!LR!R4%RIP]     ;Read-in Preset Function
        CALL    RHWRIT                    ;Execute read-in preset

        SETZM   ICCW+1                    ;End command list with a HALT
        MOVE    T4,[.RHSTR!LR!RCLP!STLW!R4%CRD] ;Write a disk READ Command
                                          ;into the STCR
        MOVNI   T1,N.BKPG*1               ;One Page
        DPB     T1,[POINT 10,T4,29]       ;Put in negative block count
```

```
;Do a data transfer operation with disk
; Accepts in T4/        DATAO Data        ;Data Xfer command for STCR
;            P1/        Channel number
;            P2/        Unit number
;            P4/        Position select word ;For the SBAR

        MOVE    T1,P4                     ;Get Block Address Word
        CALL    RHWRIT                    ;Write it to SBAR
        MOVEI   T2,TIMOUT                 ;Get timeout count
        MOVE    T1,T4                     ;Get the TCR word
        CALL    RHWRIT                    ;Write STCR. Start Xfer
DOOP1:  CALL    RHCONI                    ;Get RH status
        TXNN    T1,.RHDON                 ;RH done? Is operation done?
        SOJG    T2,DOOP1                  ;No.  Loop till done or timeout

        MOVE    T4,T1                     ;Copy final CONI
        MOVEI   T1,.RHRAE!.RHCTE!.RHMBE!.RHDON  ;Clear Done and errors
        CALL    RHCONO                    ;Via CONO to RH
        JUMPLE  T2,DOOP2                  ;Did operation ever finish?
        MOVX    T1,R4%DSR                 ;Yes. Read disk status
        CALL    RHREAD                    ;Get device status
        TRNE    T1,.RPERR                 ;Check composite error flag
        JRST    DOOP2                     ;Some drive-detected error
        CALL    RHCONI                    ;Get RH status bits
        TRNN    T1,RHERR                  ;Any errors there?
        RET                               ;No errors.  Return.
DOOP2:  MOVEI   T1,.RHMBR+.RHMBE+.RHCTE+.RHSTP  ;Here if errors. Setup CONO:
        CALL    RHCONO                    ;Massbus Reset and Enable bits
        MOVEI   T1,.RHMBE                 ;Get Massbus Enable
        CALL    RHCONO                    ;Enable Channel
        MOVE    T1,[R4%ATN!LR!377]        ;Select Attention Register
        CALL    RHWRIT                    ;Clear Attention in all drives
        SOJG    T3,RD00                   ;Repeat failing operation.
        RET                               ;Catch errors elsewhere
```

```
;Check for correct home blocks
;       no skip:        bad home block
;       one skip:       wrong structure
;       two skips:      Ok.

CHKHOM: MOVS    T1,HOMNAM               ;Get block name
        CAIE    T1,'HOM'                ;Does it say SIXBIT/HOM/ ?
        RET                             ;Bad.
        MOVE    T1,HOMCOD               ;Get block type code
        CAIE    T1,CODHOM               ;Check it (707070)
        RET                             ;Bad home block
        AOS     (P)                     ;At least one skip now.
        MOVE    T1,HOMSNM               ;Get Structure name
        CAME    T1,STRNAM               ;Does it match what we want?
        RET                             ;Wrong Structure (one skip)
        HRRZ    T1,HOMLUN               ;Get logical unit number
        SKIPE   DSKTAB(T1)              ;Is this unit number defined?
        JRST    DPLUNI                  ;Yes. Bad: Duplicate units
        HRRZM   P2,DSKTAB(T1)           ;Store Physical Unit number
        MOVEI   T2,ENTFLG(P1)           ;Phys Channel + entry-used flag
        HRLM    T2,DSKTAB(T1)           ;Store phys channel + flag
        HLRZ    T1,HOMLUN               ;Get number of units in str
        SOS     T1                      ;Convert to maximum unit #
        SKIPGE  MAXUNI                  ;Is MAXUNI already set?
        MOVEM   T1,MAXUNI               ;No: set it now.
        CAME    T1,MAXUNI               ;Is this unit the same as others?
        JRST    BADSTR                  ;No: home blocks inconsistent
        MOVE    T2,HOMRXB               ;Get root dir XB address
        SKIPN   DIORG                   ;Is root XB already set?
        MOVEM   T2,DIORG                ;No.  Set it.
        CAME    T2,DIORG                ;Must be same as any previous
        JRST    BADSTR                  ;No: home blocks are inconsistent
CPOPJ1: AOS     (P)                     ;Another skip. A good pack
CPOPJ:  RET
```

The FDSK subroutine begins with some necessary initialization code. Register P1 is set to zero. Throughout this subroutine P1 will contain the channel number of the channel of current interest. The word DIORG is initialized to zero; eventually this word will hold the disk address of the index page for this structure's root–directory. The word MAXUNI is initialized to negative one; eventually it will be set to the unit number of the highest unit in the structure.

```
FDSK:   SETZB   P1,DIORG                ;Channel 0 to P1
                                        ;Root-Dir XB addr unknown
        SETOM   MAXUNI                  ;No highest unit in structure
```

A channel command word describing a one–page data transfer is built and stored in the word at ICCW. This CCW is built by taking the virtual address of a page, A%FPO and converting it to a 22–bit physical address via the MAP instruction. The MAP instruction is illegal in user mode programs; most programs have no need to know physical addresses. However, channel command words use physical addresses, so the MAP instruction converts the virtual address A%FPO into a physical address in register T1. Some extra bits are placed in the left half of T1; these signify what modes of access to this address are legal. The program uses the TLZ instruction to clear out the bits that it has no use

for; the TLZ leaves the 22–bit physical address. Next, using TLO, the program creates a data transfer CCW by setting bits 0 and 1 (Data Transfer and Halt at the end of this transfer) and by setting the transfer word count to 1000 right adjusted in the field composed of bits 3:13. The resulting CCW is stored in the word ICCW.

```
        MAP     T1,A%FP0                ;Convert addr of FP0 to physical
        TLZ     T1,777760               ;Keep the 22 physical addr bits
        TLO     T1,(<1B0+1B1+<1000B13>>);CCW: Data, Halt, Word Count
        MOVEM   T1,ICCW                 ;Store first data CCW
```

The FDSK subroutine provides loop control by which all the disks attached to a system are scrutinized. A pair of nested loops is implemented. The outer loop steps the channel number from 0 to 7. If the Massbus controller for a channel is present, it will respond to a CONO instruction that enables its Massbus transmitters. That response can be verified by a CONI instruction which will return the enable bit if the RH20 is present. If the RH20 is absent, the program proceeds to the next channel; otherwise, the program enters an inner loop in which all devices attached to the RH20 will be examined.

The label FDSK1 is the top of the outer loop. The program does a CONO to reset the Massbus controller and a second to enable the RH20's Massbus transmitters. Finally, a CONI is executed to read the RH20's status register. If the Massbus Enabled bit comes back, the RH20 exists; the program will enter its inner loop. If the RH20 is absent, the TRNN will not skip; the program will jump to FDSK4 where it advances to the next channel.

```
FDSK1:  MOVEI   T1,.RHMBR               ;RH20 Reset function
        CALL    RHCONO                  ;Reset the RH20
        MOVEI   T1,.RHMBE               ;RH20 Enable function
        CALL    RHCONO                  ;Enable Massbus transmitters
        CALL    RHCONI                  ;Now read RH CONI, result to T1
        TRNN    T1,.RHMBE               ;Did the enable bit come back?
        JRST    FDSK4                   ;No. No bit means no channel
;Here if this RH20 exists.
```

Before we continue the discussion of the FDSK subroutine, it is useful to examine the subroutines that perform the CONO and CONI instructions. The RHCONO subroutine delivers the command flags in T1 to the RH20. If we were dealing with only one RH20, say the one corresponding to channel 0, we could simply write

```
RH0==540                                ;device code for first RH20

        CONO    RH0,(T1)                ;Transmit command flags to RH # 0
```

However, the RHCONO subroutine must deal with any of the eight RH20s, each of which is identified by its own device number. We resort to a traditional trick to generalize the device field of the CONO instruction. We write the following:

```
RHCONO: CALL    XIO1                    ;Set IO Device and do CONO
        CONO    .-.,(T1)                ;Set device conditions
        RET
```

The strange configuration of the I/O device field of the CONO, .-., is designed to alert the reader that something strange is happening. The assembler treats .-. as zero (i.e., the current value of

the location counter minus itself), so there really isn't anything mystifying about the notation so far. The strangeness resides in the XIO1 subroutine.

XIO1 will make a copy of the instruction that follows its call. In this case, it makes a copy of the CONO. Into that copy, XIO1 deposits a value for the device field. Then that copy is executed. In detail, XIO1 looks like this:

```
XIO1:   MOVEI   F,<RH0/4>(P1)           ;Produce I/O device code
IOXCT:  PUSH    P,@(P)                  ;Fetch argument, to top of stack
        AOS     -1(P)                   ;Skip once to skip the argument,
        DPB     F,[POINT 7,(P),9]       ;Store IO Device Number in instruction
        XCT     (P)                     ;Execute the target instruction
        CAIA                            ;Target didn't skip.
        AOS     -1(P)                   ;Target skipped.  Pass the Skip upwards
        ADJSP   P,-1                    ;Discard the stacked instruction
        RET
```

XIO1 begins by loading accumulator F with the 7–bit device number. The number RH0/4 is the actual 7–bit number that goes into the device field of an I/O instruction. Although we write device codes as multiples of four, the device numbers themselves are just seven bits; the division by four reduces the device code to the actual number to write into the device field. Register P1 contains the channel number which is added to RH0/4 to form in register F the device number of the specific RH20.

The instruction following the call to XIO1 is copied onto the stack. At IOXCT, the stack top, (P) contains a return PC. Indirecting through that return PC addresses the instruction at the return address. That instruction is the one that follows the CALL; it is pushed onto the stack. When returning, this routine must skip over that instruction. The return address is now at -1(P); the return address is incremented to provide a skip.

The device number in F is stored into the device field of the I/O instruction that was copied to the stack top. That instruction is then executed. If the executed instruction skips, another increment to the return address is made, thus passing the skip upwards to the caller. The copy of the instruction is removed from the stack and this routine returns.

The RHCONI routine is similar to RHCONO except it returns the device status bits in register T1:

```
RHCONI: CALL    XIO1                    ;Set IO device and do CONI
        CONI    .-.,T1                  ;Read device status
        RET
```

Now that we've finished the digression into RHCONO, we return to the loop at FDSK1. Recall that we have just discovered an RH20 is present for the channel whose number is in P1. The program builds a jump CCW to store as the initial command word for this channel. The jump CCW contains a one in bit 1 and the physical address of the word ICCW. The physical address is obtained by using the MAP instruction. Superfluous bits are zeroed; the one in bit one is added by the TXO. A copy of the channel number is multiplied by four (by the LSH instruction) for use as an index into the EPT; the jump CCW is stored in the channel's reset and status logout area in the executive process table.

```
        MAP     T1,ICCW                 ;Get physical address of ICCW
        TLZ     T1,777760               ;Keep 22 phys addr bits
        TXO     T1,1B1                  ;Make a JUMP CCW
        MOVE    T2,P1                   ;Get the current channel number
        LSH     T2,2                    ;Shift it to Channel*4
        MOVEM   T1,A%EPT+ICA(T2)        ;Store initial CCW for channel
```

We embark now on an inner loop in which the program examines the devices that are attached to this RH20's Massbus. We begin by initializing the device unit number in P2 to zero. The label FDSK2 represents the top of the inner loop in which the program scrutinizes the devices attached to one RH20's Massbus. At FDSK2 the program calls the RHREAD subroutine to read the drive type register of the selected unit.

```
        MOVEI   P2,0                    ;Start at unit 0
;Start on a Unit. Unit number in P2
FDSK2:  MOVX    T1,R4%DTR               ;Read the Drive Type register
        CALL    RHREAD
```

The result, returned in register T1 is copied to T2. The program next inspects the status of the RH20. The error flag called *Register Access Error* will be present if no device responded to the drive number in P2. If the RAE flag is present, the program clears the flag, re–enables the RH20's Massbus transmitters, and jumps to FDSK3 where the program will advance to the next drive number. If a drive responded, the RAE flag will not be set; the program jumps to FDSK2A.

```
        MOVE    T2,T1                   ;Copy Drive Type to T2
        CALL    RHCONI                  ;Check for Register Access Error
        TRNN    T1,.RHRAE               ;Test CONI bits for RAE
        JRST    FDSK2A                  ;All is well: a drive responded
        MOVEI   T1,.RHRAE!.RHMBE        ;Must clear RAE.  Set ENABLE.
        CALL    RHCONO                  ;CONO to clear RAE & set ENABLE
        JRST    FDSK3                   ;Drive doesn't exist. Try next.
```

At FDSK2A, a drive of some kind is present at this channel and unit address. The subroutine GOTDRV is called to determine what kind of device this is and to make all further examination of it. GOTDRV returns to FDSK3 where the unit number will be advanced. After all units have been examined, the program falls into FDSK4 (to which it jumps if an RH20 is absent) to advance to the next channel. After all channels have been scrutinized, the program checks to see if a consistent structure has been found. A consistent structure will have non–zero entries in DSKTAB for each logical unit. GOTDRV is responsible for building the DSKTAB array and for setting up the contents of MAXUNI.

```
FDSK2A: CALL    GOTDRV                  ;See what we have here
FDSK3:  CAIGE   P2,MAXDRV-1             ;Done with all units yet?
        AOJA    P2,FDSK2                ;No, go do next unit now.
FDSK4:  CAIGE   P1,MAXCHN-1             ;Checked all channels yet?
        AOJA    P1,FDSK1                ;No, advance to next channel
;Now, see if we found a consistent structure.  (GOTDRV does all the real work.)
        SKIPGE  T1,MAXUNI               ;Now, check for full structure
        JRST    MISSTR                  ;Structure not found
CHKSTR: SKIPN   DSKTAB(T1)              ;Any physical addr for this LUN?
        JRST    MISUNI                  ;Bad structure: missing unit
        SOJGE   T1,CHKSTR               ;Loop thru all units
```

After finding that all the units are present, the word NUMCYL, the number of cylinders in each unit is computed before returning. NUMCYL is set according to the type of disk unit, as found in DSKTYP. DSKTYP is another word that is set up by the GOTDRV subroutine.

```
        MOVE    T2,DSKTYP               ;Get the type of this disk
        MOVEI   T1,N.CLP4               ;Get # of cylinders/unit for 04
        CAIE    T2,.R4TYP               ;RP04?
        MOVEI   T1,N.CLP6               ;No.  Assume RP06. Get cyl/unit
        CAIN    T2,.R3TYP               ;RM03?
        MOVEI   T1,N.CLP3               ;Yes
        CAIN    T2,.R7TY1               ;RP07?
        MOVEI   T1,N.CLP7               ;Yes
        MOVEM   T1,NUMCYL               ;Save result
        RET
```

Before we discuss GOTDRV, we should examine the subroutine that reads and writes device registers. At FDSK2 the program calls the RHREAD routine. That routine looks like this:

```
RHREAD: TXZ     T1,LR                   ;Clear LR bit
RHWRIT: TSO     T1,P2                   ;Insert Drive Number in left half
        CALL    XIO1                    ;Set IO Device Field & do DATAO
        DATAO   .-.,T1                  ;Send register select & data to device
        CALL    XIO1                    ;Set IO Device Field & do DATAI
        DATAI   .-.,T1                  ;Read selected register
        RET
```

The RHREAD routine is called with a drive register selection code in T1, a drive number in P2, and the channel number in P1. In RHREAD, the program clears the LR bit from T1; if LR were set, the program would write the device register. Then, the drive number is copied into the left half of T1 by the TSO instruction. XIO1 sets the device number for the appropriate RH20 in the DATAO instruction and executes the DATAO. The DATAO data in T1 is sent to the RH20; the data includes the register and drive select fields. The RH20 remembers the register and drive select fields. When XIO1 executes the DATAI instruction, the RH20 will interrogate the previously specified drive and register; the resulting data is returned to the processor. The processor stores the result in T1.

We turn now to GOTDRV. In overview, GOTDRV will determine if this unit is a known type of disk. If not, it will return quickly; perhaps the device is a tape unit. If the device is a disk, and if it has a pack mounted and spinning (i.e., media on–line), then GOTDRV will attempt to read the home block. If GOTDRV successfully reads the home block and recognizes this unit as part of the structure that it is seeking, GOTDRV will store information in the DSKTAB table to establish the mapping from logical unit to physical channel and unit.

After recognizing one unit of the sought–for structure, GOTDRV becomes more selective. It will ignore disk units that are not of the same type as the first unit of the structure.

The program enters GOTDRV with the drive type word in register T2. The nine drive type bits are kept; all others are discarded by the ANDI instruction. At this point various inconsistencies in drive types are eliminated. For example, the RP04 and RP05 disks are functionally equivalent, so the program relabels any RP05 it finds as an RP04.

```
GOTDRV: ANDI    T2,R4%TYP               ;Keep only the device type code
        CAIN    T2,.R5TYP               ;Is this an RP05
        MOVEI   T2,.R4TYP               ;Yes, make it look like an RP04
        CAIN    T2,.R7TY2               ;Alternate kind of RP07?
        MOVEI   T2,.R7TY1               ;Make it the usual kind of RP07
```

Initially the word DSKTYP is zero. Until we find the first unit of the sought–for structure, DSKTYP remains non–positive, and the program examines every unit found. When one of the structure's units is found, the program will set DSKTYP to be the (positive) type of that unit. When DSKTYP is positive, the program limits its attention to just those units that match the unit already found.

```
        SKIPG   T1,DSKTYP               ;Are we looking for a specific disk?
        JRST    GOTDV3                  ;Not yet, check if this is allowable
        CAMN    T2,T1                   ;yes.  is this the same as we want?
        JRST    GOTDV0                  ;Same as we want. Read its home block
        RET                             ;No, don't use this one.
```

If no unit of the structure has been found, the program continues at GOTDV3. If a unit has been found, and this unit is the same type, the program scrutinizes this unit at GOTDV0. Otherwise, this unit is ignored, because it doesn't match the rest of the structure.

At GOTDV3 the program is receptive to looking for the structure on any kind of disk unit known to it. The drive type is tested to see if it matches any of the disk types that this program is prepared to accept. If the drive type corresponds to one of the recognized disk units, the program jumps to GOTDV4. If the drive type is not recognized this subroutine exits.

```
GOTDV3: CAIE    T2,.R7TY1               ;RP07?
        CAIN    T2,.R3TYP               ;or RM03?
        JRST    GOTDV4                  ;Yes.
        CAIE    T2,.R6TYP               ;RP06?
        CAIN    T2,.R4TYP               ;or RP04?
        JRST    GOTDV4                  ;Yes. Save type (negated)
        RET                             ;This is no kind of disk for us
```

At GOTDV4 the program tentatively saves the drive type negated in DSKTYP. If this unit is discovered to be part of the sought–for structure, DSKTYP will be made positive. The program arrives at GOTDV0 with a unit whose home blocks must be read. A pack acknowledge command is sent to the drive; if the drive is freshly powered–up, the pack acknowledge is necessary before attempting to access the device. The drive's status register is then examined. This routine exits unless the drive reports the medium on–line bit; when the disk isn't spinning it's hard to read the home blocks.

```
GOTDV4: MOVNM   T2,DSKTYP               ;Save negative drive type
                                        ;in case the home blocks are bad
GOTDV0: MOVE    T1,[R4%CSR!LR!R4%CPA]   ;Send Pack Acknowledge.
        CALL    RHWRIT                  ;Do DATAO.  Sets Volume Valid
        MOVX    T1,R4%DSR               ;Read device status register
        CALL    RHREAD                  ;Via DATAO/DATAI
        TRNN    T1,.RPMOL               ;Is Disk Pack On-Line?
        RET                             ;No. Forget it.
```

This program has found an on–line disk drive. The program zeroes the page of memory at A%FP0; this page will be used as a data buffer. The program calls the RD0 subroutine to read page zero,

records 0, 1, 2, and 3. Record 1 contains the home block.

```
        SETZM   A%FP0                   ;Clear data page
        MOVE    T1,[A%FP0,,A%FP0+1]     ;Set up for the BLT
        BLT     T1,A%FP0+777            ;Zero data page before the Read
        CALL    RD0                     ;Read disk page 0 (records 0,1,2,3)
```

When RD0 returns, GOTDRV calls CHKHOM to check whether record 1 is a home block that matches the structure we seek. CHKHOM has three returns. The non–skip return indicates that the record was not a proper home block; the program will jump to GOTDV2 to read and check the backup home block. The single–skip return signifies that the wrong structure was found; the GOTDRV subroutine returns. Finally, the double–skip return signals that a proper home block of the sought–for structure has been found; the program sets DSKTYP to be a positive number and returns.

```
        CALL    CHKHOM                  ;Check the home block
        JRST    GOTDV2                  ;Bad.  Try the backup home block
        RET                             ;Good block, but wrong structure
        MOVMS   DSKTYP                  ;Set DSKTYP Positive.
        RET
```

If the primary home block is bad, the program jumps to GOTDV2 where it calls RD11 to read records 11, 12, 13, and 14. Record 12 is the backup home block. Again, CHKHOM is called to see whether a good home block is present.

```
GOTDV2: CALL    RD11                            ;Read records 11,12,13,14
        CALL    CHKHOM                          ;Check the backup home block
        RET                                     ;Error.
        RET                                     ;Wrong structure
        MOVMS   DSKTYP                          ;Set DSKTYP positive.
        RET
```

The data transfer operation is accomplished by the RD0 (and RD11) subroutines. These routines differ only in the data that they send to the block address register in the RH20. The respective subroutine entry points load commands to the disk that start data transfers on track 0, record 0, or on track 0, record 11:

```
RD11:   SKIPA   P4,[.RHSBR!LR!0B27!11B35]       ;Desired sector and track
RD0:    MOVE    P4,[.RHSBR!LR!0B27!0B35]        ;For RH20 SBAR
```

Next, a retry counter is established in register T3. If an operation fails, it will be repeated (by jumping back to RD00) as many times as permitted by this retry counter.

At RD00 a recalibrate operation is performed. The heads are retracted from the data portion of disk onto a region near the edge of the platter called the guard zone. Then the positioner seeks inward until it comes to the first data track; this is cylinder zero. It is sometimes possible for the disk and disk interface to become confused about the positioner location; the recalibrate operation restores the positioner to a known condition.

```
        MOVEI   T3,NRETRY                       ;Initialize retry counter
RD00:   MOVE    T1,[R4%CSR!LR!R4%CRC]           ;Recalibrate function
        CALL    RHWRIT                          ;Execute recalibrate
```

Because the positioner may take a long time to respond, the program loops, sampling the drive

status register, looking for the drive ready indication to come on, indicating the completion of the recalibration.

```
RDOA:   MOVX    T1,R4%DSR                       ;Select unit status register
        CALL    RHREAD                          ;Get drive status
        TXNN    T1,.RPDRY                       ;Is drive ready?
        JRST    RDOA                            ;No. Wait till recalibrate done
```

When the recalibrate finishes, the program sends a read–in preset command to the drive. This command clears any cylinder offset that may have been left over, and it clears the desired cylinder address register to zero.

```
        MOVE    T1,[R4%CSR!LR!R4%RIP]           ;Read-in preset function
        CALL    RHWRIT                          ;Execute read-in preset
```

Next, the program stores a zero in the word at `ICCW+1` to halt the channel's command list. A disk read command is loaded into `T4`. Remember that data transfer commands must be sent to the transfer control register in the RH20; thus the secondary transfer control register appears in the register select field of this command. The command bits to cause the channel to reset its command list pointer and to store the ending status are set in the data word. To finish specifying the data word for the transfer control register, the program must put the negative block count in bits `20:29`. There are four disk blocks (records) for each page; a `-4` is deposited in the appropriate field of `T4`. When this word is sent to the RH20, the data transfer will start.

```
        SETZM   ICCW+1                          ;End command list with a HALT
        MOVE    T4,[.RHSTR!LR!RCLP!STLW!R4%CRD] ;Write a disk READ Command
                                                ;into the STCR
        MOVNI   T1,N.BKPG*1                     ;One page
        DPB     T1,[POINT 10,T4,29]             ;Put in negative block count
```

The program is now ready to initiate a data transfer operation. The word in `P4`, containing the track and sector number is sent to the RH20's block address register. Then the read command in `T4` is sent to the transfer control register. When the RH20 receives this command word, it passes the read command to the disk drive and it tells the M box channel to prepare to receive data from the disk.

```
        MOVE    T1,P4                           ;Get Block Address Word
        CALL    RHWRIT                          ;Write it to SBAR
        MOVE    T1,T4                           ;Get the TCR word
        CALL    RHWRIT                          ;Write STCR. Start Xfer
```

When the disk positions itself to the right sector, it begins transferring words to the RH20. The RH20 passes these data words along the C bus to the M box channel. The M box deposits these words in memory addresses as directed by the command list.

Meanwhile, the program loads register `T2` with a large count and waits for the RH20 status to indicate that the transfer is done. Alternatively, if the transfer becomes stuck for any reason, the count in `T2` will expire.

```
        MOVEI   T2,TIMOUT                       ;Get timeout count
DOOP1:  CALL    RHCONI                          ;Get RH status
        TXNN    T1,.RHDON                       ;RH done? Is operation done?
        SOJG    T2,DOOP1                        ;No, Loop till done or timeout
```

The status bits in T1 are copied to T4. A CONO is performed to clear the transfer error flags, register access error, and the command done flag. If the timeout count expired without the transfer finishing, T2 will be zero; the program will jump to the general error handler at DOOP2.

```
        MOVE    T4,T1                           ;Copy CONI status to T4
        MOVEI   T1,.RHRAE!.RHCTE!.RHMBE!.RHDON  ;Clear DONE and errors
        CALL    RHCONO                          ;Via CONO to RH
        JUMPLE  T2,DOOP2                        ;Did operation ever finish?
```

Unless there was a timeout, the program will read the drive status to test the composite error indication. If any error is present, the program jumps to the error retry code at DOOP2. If the drive reports no errors then the program tests the RH20's status bits that are held in T4. If there are no errors and no unusual status bits in T4, this subroutine returns; otherwise it falls into the retry code at DOOP2.

```
        MOVX    T1,R4%DSR                       ;Yes. Read disk status
        CALL    RHREAD                          ;Get device status
        TRNE    T1,.RPERR                       ;Check composite error flag
        JRST    DOOP2                           ;Some drive-detected error
        TRNN    T4,RHERR                        ;Any errors in RH CONI?
        RET                                     ;No errors.  Return
;Fall into DOOP2
```

At DOOP2 the program clears out the channel and the device. It resets all connected Massbus devices; in case of a hung transfer, it stops the transfer by setting the .RHSTP bit in the CONO. All attention interrupts that may be present in the attached devices are cleared by writing into all drives' attention registers. The retry count in T3 is decremented. If the retry count remains positive, the program loops to RD00 where it will restart the failing transfer. After all retries have been exhausted, this subroutine gives up and returns to its caller.

```
DOOP2:  MOVEI   T1,.RHMBR+.RHMBE+.RHCTE+.RHSTP  ;Here if errors. Setup CONO:
        CALL    RHCONO                          ;Massbus Reset and Enable bits
        MOVEI   T1,.RHMBE                       ;Get Massbus Enable
        CALL    RHCONO                          ;Enable Channel
        MOVE    T1,[R4% ATN!LR!377]              ;Select Attention Register
        CALL    RHWRIT                          ;Clear Attention in all drives
        SOJG    T3,RD00                         ;Repeat failing operation.
        RET                                     ;Catch errors elsewhere
```

We come at last to the CHKHOM subroutine. The data reading routines, RD0 and RD11 have left the home block in the second record of the data page. The word HOMNAM is the first word of that record, located at A%FP0+200. If this record is really a home block then it will begin with a word containing the constant SIXBIT /HOM/; also, it will contain another constant, octal 707070 near the end of the record. CHKHOM looks for both of these data items. If either is absent, this routine returns without skipping, indicating a bad home block.

```
CHKHOM: MOVS    T1,HOMNAM               ;Get block name
        CAIE    T1,'HOM'                ;Does it say SIXBIT/HOM/ ?
        RET                             ;Bad.
        MOVE    T1,HOMCOD               ;Get block type code
        CAIE    T1,CODHOM               ;Check it (707070)
        RET                             ;Bad home block
```

If both words match, this record is plausibly a home block. The return PC is incremented now. Next, if the structure name in the home block fails to match the name of the sought–for structure, CHKHOM returns. A good home block was found, but it did not match the structure that this program is looking for.

```
AOS     (P)                     ;At least one skip now
MOVE    T1,HOMSNM               ;Get structure name
CAME    T1,STRNAM               ;Does it match what we want?
RET                             ;Wrong structure (one skip)
```

At this point the program has found the home block of one unit of the structure that it has been looking for. Some checks are performed to avoid being tricked into using an inconsistent structure. First, the program checks the right half of HOMLUN. This should be the logical unit number of this unit. If DSKTAB indicates that we have seen this logical unit number before, then the program exits to DPLUNI, where it will complain of duplicate units.

```
HRRZ    T1,HOMLUN               ;Get logical unit number
SKIPE   DSKTAB(T1)              ;Is this unit number defined?
JRST    DPLUNI                  ;Yes. Bad: duplicate units
```

If this is the first time we have seen this logical unit number, the program will store the physical unit number in the right half of DSKTAB indexed by the logical unit number. Then it stores the physical channel number and the flag called ENTFLG into the left half of DSKTAB. The flag is included because without it, the entry for physical channel 0 unit 0 would look like an empty entry.

```
HRRZM   P2,DSKTAB(T1)           ;Store physical unit number
MOVEI   T2,ENTFLG(P1)           ;Phys channel + entry-used flag
HRLM    T2,DSKTAB(T1)           ;Store phys channel + flag
```

The number of units in the structure is contained in the left half of HOMLUN. That number is copied to T1 and decremented. The result is the maximum unit number in the structure. If the word MAXUNI has not been changed, this result is stored there. If the word MAXUNI has been set, its value is compared to the one that has just been computed. If the values fail to match then the previous home block said that there are a different number of units in this structure than this home block claims; if this mismatch occurs, the program exits to BADSTR where it complains of inconsistent structures.

```
HLRZ    T1,HOMLUN               ;Get number of units in str
SOS     T1                      ;Convert to maximum unit #
SKIPGE  MAXUNI                  ;Is MAXUNI already set?
MOVEM   T1,MAXUNI               ;No: set it now.
CAME    T1,MAXUNI               ;Is this unit the same as others?
JRST    BADSTR                  ;No: home blocks inconsistent
```

The last consistency test is simple. The word HOMRXB contains the disk address of the index page for this structure's root–directory. If the word DIORG has not yet been initialized, then HOMRXB is copied to it. If DIORG has been initialized from some previous home block, the new value of HOMRXB must agree with the old; otherwise, the program complains.

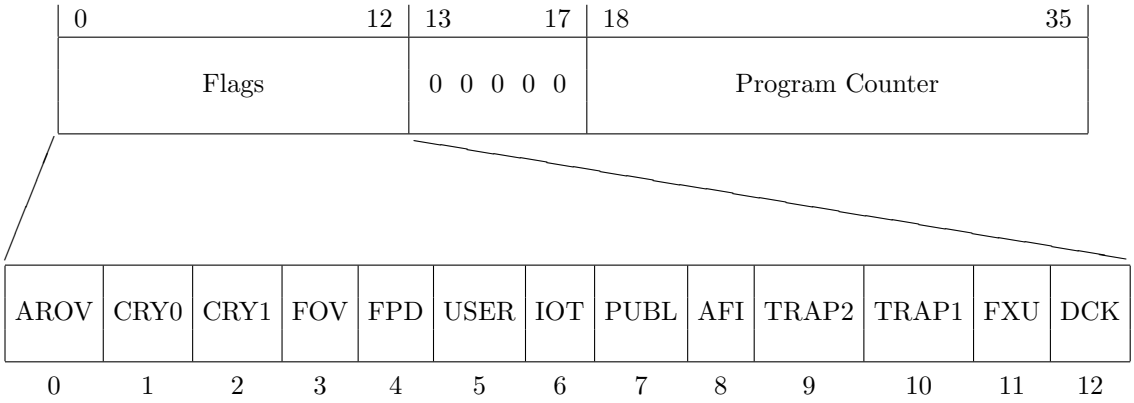
```
        MOVE    T2,HOMRXB               ;Get root dir XB address
        SKIPN   DIORG                   ;Is root XB already set?
        MOVEM   T2,DIORG                ;No.  Set it.
        CAME    T2,DIORG                ;Must be same as any previous
        JRST    BADSTR                  ;No: home blocks are inconsistent
CPOPJ1: AOS     (P)                     ;Another skip. A good pack
CPOPJ:  RET
```

In cases where the program detects an inconsistent structure, it has no choice but to request manual intervention by the computer operator to correct the difficulty.

Appendix A

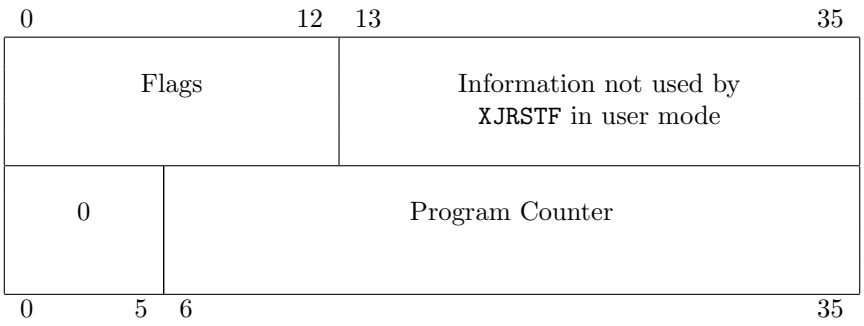
Program Counter (PC) & Flags

The state of the CPU is indicated by a small number of single–bit entities called flags. Most subroutine call instructions, when executed in section 0 (or on an unextended machine), save the program counter and flags in the single–word format shown below.



Flags and Program Counter Single–Word

Subroutine calls that are executed in non–zero sections usually save only the 30–bit extended PC; the flags are often ignored. However, the flags can be saved using the `SFM` or `XPCW` instructions. A doubleword containing both flags and program counter is used in the `XJRSTF` and `XJEN` instructions. The `XPCW` instruction references a 4–word block in which it stores the old flags and PC doubleword, and from which it fetches a new flags and PC doubleword. See also Section 30, page 565.



Flags and Program Counter Double–Word

A.1 Flags

AROV — The AROV flag signifies that an arithmetic overflow condition has occurred. Once set, this flag remains set until cleared by JFCL or until the flags are restored by JRSTF or another appropriate form of JRST. AROV is set by any of the conditions listed below.

- A single instruction has set one of CRY0 or CRY1 without setting them both.
- An ASH or ASHC has left–shifted a significant bit out of bit 1 of the specified accumulator.
- A MULx instruction has multiplied -2^{35} by itself.
- A DMUL instruction has multiplied -2^{70} by itself.
- An IMULx instruction has produced a product less than -2^{35} or greater than $2^{35}-1$.
- A FIX or FIXR has fetched an operand with exponent greater than decimal 35.
- The FOV flag, signifying floating–point overflow, has been set.
- The DCK (divide check) flag has been set.

CRY0 — The CRY0 flag when set indicates that an instruction has caused a one bit to be carried out of bit 0 (and discarded). Once this flag is set, it stays set until cleared by the JFCL instruction, or by some instruction that restores the PC and flags.

If CRY0 is set by an instruction that doesn't set CRY1 also, an overflow condition is signaled. AROV will be set. This indicates any of the following conditions:

- An ADDx has added two negative numbers with sum less than -2^{35}.
- A SUBx has subtracted a positive number from a negative number and produced a result less than -2^{35}.
- A SOSx or SOJx has decremented -2^{35}.

If CRY0 and CRY1 are both set, this indicates that one of the following non–overflow events has occurred:

- In ADDx both summands were negative, or their signs differed and the positive one was greater than or equal to the magnitude of the negative summand.

- In SUBx the sign of both operands was the same and the accumulator operand was greater than or equal to the memory operand, or the accumulator operand was negative and the memory operand was positive.
- An AOJx or AOSx has incremented −1.
- A SOJx or SOS has decremented a non–zero number other than -2^{35}.
- A MOVNx has negated zero.

CRY1 The CRY1 flag is set when a one is carried out of bit 1 into bit 0. If CRY1 is set by an instruction that does not also set CRY0, an overflow condition is indicated; AROV will be set. This indicates that any of the following conditions has occurred:

- An ADDx has added two positive number with a sum greater than $2^{35} - 1$.
- A SUBx has subtracted a negative number from a positive number to form a difference greater than $2^{35} - 1$.
- An AOSx or AOJx instruction has incremented $2^{35} - 1$.
- A MOVNx or MOVMx has negated -2^{35}.
- A DMOVNx has negated -2^{70}.

FOV The FOV flag indicates that a floating–point overflow condition has occurred. FOV is set by any of the following conditions:

- In a floating-point instruction other than FLTR, DMOVNx, or DFN, the exponent of the result exceeds decimal 127.
- Some instruction has set the FXU (floating exponent underflow) flag.
- The DCK (divide check) flag was set by FDVx, FDVRx, or DFDV.

FPD The FPD (first part done) flag is set when the processor responds to a priority interrupt after having completed the first part of a two–part instruction (e.g., ILDB). When the instruction resumes, this flag being set will inhibit a repetition of the first part. This flag is not usually of interest to the programmer.

USER This flag is set while the processor is in user mode. In user mode, various instruction and addressing restrictions are in effect.

IOT User In/Out mode, also called IOT User mode, is a special mode in which some of the user mode instruction restrictions are removed. In this mode a user program may perform the hardware I/O instructions, but may not violate the addressing restrictions.

PUBL Public mode signifies that the processor is in user public mode or in exec supervisor mode.

AFI If the Address Failure Inhibit flag is set, address break is inhibited during the execution of the next instruction. Address Failure Inhibit is used after an *address break trap* to proceed past the instruction that trapped. After one instruction (presumably the one that trapped) is executed, this flag is cleared so a subsequent reference to the same address would cause another trap. This flag isn't of particular use to the user–mode programmer, as these traps are processed by the operating system. However, it should be noted that in TOPS–20, the address break system may be used to help debug user–mode programs. The TOPS–20 EXEC has a command (SET ADDRESS-BREAK) for manipulating the status of the address–break system. The address break system is not implemented in the 2020 systems.

`TRAP2` If `TRAP1` is not also set, `TRAP2` signifies that a pushdown overflow has occurred. If traps are enabled, setting this flag immediately causes a trap. At present no hardware condition sets both `TRAP1` and `TRAP2` simultaneously.

`TRAP1` If `TRAP2` is not also set, `TRAP1` signifies that an arithmetic overflow has occurred. If traps are enabled, setting this flag immediately causes a trap. At present no hardware condition sets both `TRAP1` and `TRAP2` simultaneously.

`FXU` Floating exponent underflow is set to signify that in a floating–point instruction other than `DMOVNx`, `FLTR`, or `DFN`, the exponent of the result was less than decimal −128. The flags `AROV` and `FOV` will also be set.

`DCK` The divide check flag signifies that one of the following conditions has set `AROV`:

- In a `DIVx` instruction the high–order word of the dividend was greater than or equal to the divisor.
- In an `IDIVx` instruction the divisor was zero.
- In an `FDVx`, `FDVRx`, or `DFDV`, the divisor was zero, or the magnitude of the dividend fraction was greater than or equal to twice the magnitude of the divisor fraction (this condition can not occur if the divisor is properly normalized). In either case, `FOV` is also set.

A.2 The Program Counter

In the single word containing the PC and flags, bits 13 through 17 of the PC word are always zero. This facilitates the use of indirect addressing to return from a subroutine.

A 30–bit PC is stored with bits 0 through 5 set to zero. This is the format of a global indirect pointer. This allows the use of indirect or indexed addressing when returning from a subroutine.

When the program counter is stored by one of the subroutine calling instructions, the PC will already have been incremented to point to the instruction immediately following the subroutine call; this is the normal return address. This return address is stored in the PC word. Thus, the PC word points at the address to which the subroutine should return.

Appendix B

Instruction Nomenclature and Instruction Set

`E`	The effective address resulting from the `I`, `X`, and `Y` parts of the instruction. In the description of instructions executed under `EXTEND`, E refers to the effective address of the instruction found at the effective address of the `EXTEND` instruction.
`L,,R`	The fullword composed of `L` in the left half and `R` in the right half.
`C(E)`	The contents of the word addressed by `E`.
`C0(E)`	The value of bit 0 in the word addressed by `E`.
`C18(E)`	The value of bit 18 in the word addressed by `E`.
`CR(E)`	The contents of the right half of the word addressed by `E`.
`CL(E)`	The contents of the left half of the word addressed by `E`.
`CS(E)`	The fullword composed of the swapped contents of `E: CR(E),,CL(E)`
`AC`	The value of the accumulator field of the instruction.
`C(AC)`	The contents of the accumulator selected by `AC`.
`C(AC AC+1)`	A double–word accumulator in which `C(AC)` is most significant.
`C(E E+1)`	A double–word memory operand in which `C(E)` is most significant.
`PC`	The 18–bit contents of the program counter.
$\wedge$	Boolean AND
$\vee$	Boolean Inclusive OR
$\neg$	Boolean Negation (One's Complement)
$\oplus$	Boolean Exclusive OR
$\equiv$	Boolean Equivalence

Notation for Instruction Descriptions

Arithmetic and Boolean Operators in MACRO

Operation	Operator Code	Example		
AND	&	5&11	=	1
OR (inclusive)	!	5!11	=	15
XOR	^!	5^!11	=	14
NOT (ones complement)	^-	^-173	=	777777,,777604
Addition	+	5+11	=	16
Subtraction	-	5-11	=	777777,,777774
Multiplication	*	5*11	=	55
Division	/	11/5	=	1
Logical Shift	_	5_11	=	5000
Logical Shift	B	5B11†	=	500,,0
Force Octal Radix	^O	^O123	=	123
Force Decimal Radix	^D	^D123	=	173
Bit Position (JFFO)	^L	^L123	=	35

† The second parameter is taken as a (decimal) bit number

PDP–10 Instruction Set

	0	1	2	3	4	5	6	7
000	MUUO	LUUO CMSL†	LUUO CMSE†	LUUO CMSLE†	LUUO EDIT†	LUUO CMSGE†	LUUO CMSN†	LUUO CMSG†
010	LUUO CVTDBO†	LUUO CVTDBT†	LUUO CVTBDO†	LUUO CVTBDT†	LUUO MOVSO†	LUUO MOVST†	LUUO MOVSLJ†	LUUO MOVSRJ†
020	LUUO XBLT†	LUUO GSNGL†	LUUO GDBLE†	LUUO GDFIX†	LUUO GFIX†	LUUO GDFIXR†	LUUO GFIXR†	LUUO DGFLTR†
030	LUUO GFLTR†	LUUO GFSC†	LUUO	LUUO	LUUO	LUUO	LUUO	LUUO
040	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO
050	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO
060	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO
070	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO	MUUO
100	MUUO	MUUO	GFAD	GFSB	JSYS	ADJSP	GFMP	GFDV
110	DFAD	DFSB	DFMP	DFDV	DADD	DSUB	DMUL	DDIV
120	DMOVE	DMOVN	FIX	EXTEND	DMOVEM	DMOVNM	FIXR	FLTR
130	UFA§	DFN§	FSC	IBP	ILDB	LDB	IDPB	DPB
140	FAD	FADL§	FADM	FADB	FADR	FADRI	FADRM	FADRB
150	FSB	FSBL§	FSBM	FSBB	FSBR	FSBRI	FSBRM	FSBRB
160	FMP	FMPL§	FMPM	FMPB	FMPR	FMPRI	FMPRM	FMPRB
170	FDV	FDVL§	FDVM	FDVB	FDVR	FDVRI	FDVRM	FDVRB
200	MOVE	MOVEI	MOVEM	MOVES	MOVS	MOVSI	MOVSM	MOVSS
210	MOVN	MOVNI	MOVNM	MOVNS	MOVM	MOVMI	MOVMM	MOVMS
220	IMUL	IMULI	IMULM	IMULB	MUL	MULI	MULM	MULB
230	IDIV	IDIVI	IDIVM	IDIVB	DIV	DIVI	DIVM	DIVB
240	ASH	ROT	LSH	JFFO	ASHC	ROTC	LSHC	MUUO
250	EXCH	BLT	AOBJP	AOBJN	JRST	JFCL	XCT	MAP
260	PUSHJ	PUSH	POP	POPJ	JSR	JSP	JSA§	JRA§
270	ADD	ADDI	ADDM	ADDB	SUB	SUBI	SUBM	SUBB
300	CAI	CAIL	CAIE	CAILE	CAIA	CAIGE	CAIN	CAIG
310	CAM	CAML	CAME	CAMLE	CAMA	CAMGE	CAMN	CAMG
320	JUMP	JUMPL	JUMPE	JUMPLE	JUMPA	JUMPGE	JUMPN	JUMPG
330	SKIP	SKIPL	SKIPE	SKIPLE	SKIPA	SKIPGE	SKIPN	SKIPG

† This instruction is available only under EXTEND. § This instruction is obsolete.

PDP–10 Instruction Set

	0	1	2	3	4	5	6	7
340	AOJ	AOJL	AOJE	AOJLE	AOJA	AOJGE	AOJN	AOJG
350	AOS	AOSL	AOSE	AOSLE	AOSA	AOSGE	AOSN	AOSG
360	SOJ	SOJL	SOJE	SOJLE	SOJA	SOJGE	SOJN	SOJG
370	SOS	SOSL	SOSE	SOSLE	SOSA	SOSGE	SOSN	SOSG
400	SETZ	SETZI	SETZM	SETZB	AND	ANDI	ANDM	ANDB
410	ANDCA	ANDCAI	ANDCAM	ANDCAB	SETM	SETMI XMOVEI‡	SETMM	SETMB
420	ANDCM	ANDCMI	ANDCMM	ANDCMB	SETA	SETAI	SETAM	SETAB
430	XOR	XORI	XORM	XORB	IOR	IORI	IORM	IORB
440	ANDCB	ANDCBI	ANDCBM	ANDCBB	EQV	EQVI	EQVM	EQVB
450	SETCA	SETCAI	SETCAM	SETCAB	ORCA	ORCAI	ORCAM	ORCAB
460	SETCM	SETCMI	SETCMM	SETCMB	ORCM	ORCMI	ORCMM	ORCMB
470	ORCB	ORCBI	ORCBM	ORCBB	SETO	SETOI	SETOM	SETOB
500	HLL	HLLI XHLLI‡	HLLM	HLLS	HRL	HRLI	HRLM	HRLS
510	HLLZ	HLLZI	HLLZM	HLLZS	HRLZ	HRLZI	HRLZM	HRLZS
520	HLLO	HLLOI	HLLOM	HLLOS	HRLO	HRLOI	HRLOM	HRLOS
530	HLLE	HLLEI	HLLEM	HLLES	HRLE	HRLEI	HRLEM	HRLES
540	HRR	HRRI	HRRM	HRRS	HLR	HLRI	HLRM	HLRS
550	HRRZ	HRRZI	HRRZM	HRRZS	HLRZ	HLRZI	HLRZM	HLRZS
560	HRRO	HRROI	HRROM	HRROS	HLRO	HLROI	HLROM	HLROS
570	HRRE	HRREI	HRREM	HRRES	HLRE	HLREI	HLREM	HLRES
600	TRN	TLN	TRNE	TLNE	TRNA	TLNA	TRNN	TLNN
610	TDN	TSN	TDNE	TSNE	TDNA	TSNA	TDNN	TSNN
620	TRZ	TLZ	TRZE	TLZE	TRZA	TLZA	TRZN	TLZN
630	TDZ	TSZ	TDZE	TSZE	TDZA	TSZA	TDZN	TSZN
640	TRC	TLC	TRCE	TLCE	TRCA	TLCA	TRCN	TLCN
650	TDC	TSC	TDCE	TSCE	TDCA	TSCA	TDCN	TSCN
660	TRO	TLO	TROE	TLOE	TROA	TLOA	TRON	TLON
670	TDO	TSO	TDOE	TSOE	TDOA	TSOA	TDON	TSON

‡ This operation is available only in non-zero sections.

Appendix C

DDT

DDT is a large program that changes as new ideas and helpful features are added. It is not possible to create a description that will be both complete and totally accurate as the program continues to develop. We believe the following is a correct (but not entirely complete) description of DDT in release 7 of TOPS–20. Some of this is a repetition of the material found in Section 9. The intention is to provide a nearly complete description of all the features of DDT.

In the description of DDT commands, the following rules of nomenclature apply:

- The dollar sign character (`$`) signifies places where the escape key must be typed. This key is labeled as `ESC` or `ALT-MODE` on most terminals. When you type the escape key, DDT will display a dollar sign.
- Numbers are represented by n. Numbers are interpreted as octal, except that digits followed by a decimal point are base ten; if digits follow the decimal point, a floating–point number is assumed.
- A number that follows an escape, written as `$`n, may be interpreted as either decimal or as octal, depending on the command in which it appears.

C.1 Examines and Deposits

You may specify the location that you wish to examine by any numeric or symbolic address expression. Follow the address expression with one of the command characters that opens a location.

When a location is examined, the contents of that location are displayed. Initially, the *mode* in which locations are displayed is *symbolic*, that is, the contents of locations will be interpreted as instructions; the addresses of locations will be interpreted as labels where possible. The radix for displaying numbers initially is octal. See Appendix C.2, page 646 for the commands by which you can change the display mode or the radix.

To *open* a location means to read and (usually) display the contents of the location. You may deposit new data into an open location by typing a new value followed by a command that performs a deposit; DDT will store the new value, obliterating the previous contents. Data, instructions, or the contents of the accumulators may all be changed in this way.

Some special symbols exist in DDT. Among the most important of these are the *current location*, referred to by the character period (`.`), and `$Q`, the *current quantity*. Additionally, there are special symbols called *masks*. Each mask controls some function within DDT; for example, the *search mask* (`$M`) affects the DDT word searches.

C.1.1 Current Location

The character period (`.`) is the symbolic name of the *current location*. Most commands that open locations set the current location to the address that has just been opened.

C.1.2 Current Quantity

The symbol `$Q` is the name of the *current quantity*. The current quantity is either the last value typed by DDT (i.e., the value of the location most recently displayed), or any new address or value that has been typed by the user. Some DDT commands use the right half of the current quantity as an address when no address argument is specified.

The value of the current quantity with right and left halves swapped is accessible by the symbolic name `$$Q`.

C.1.3 Examine Commands

`addr/` Opens the location specified by the address expression "`addr`".

The contents of that location are displayed in the current mode. The *current location*, ".", is set to this address.

If no address expression is mentioned, DDT will open and display the location addressed by the right half of the current quantity; when no address expression is mentioned, DDT will avoid changing the value of ".".

`addr[` Opens the specified location; displays its contents as a number in the current radix; DDT will change "`.`" to this address. If no address expression appears, "`.`" is not changed; the address to open is taken from the right half of the current quantity.

`addr]` Opens the location; displays its contents as symbolic, i.e., as an instruction, if possible. Changes ".". Again, if no address expression appears before the `]`, the address to open is taken from the right half of the current quantity, the current location is unchanged.

`addr!` Opens the specified location without displaying the contents. Changes ".".

C.1.4 Deposit Commands

Before we describe the commands to store (deposit) new values in memory or accumulators, we must first mention that DDT normally respects the restrictions that you establish for your program. TOPS-20 allows you to write–protect instructions and constants; e.g., by placing them in a read–only psect. DDT will refuse to store into a location in a read–only region of memory.

However, if you want to change a word within a read–only region of memory, you can tell DDT to override the protections that you have previously set. The following commands direct how DDT handles memory protection:

`$W` directs DDT to ignore the memory protections that you have set and perform the deposits that you request. This command does not change the memory protection as seen by the running program: it affects only DDT's behavior. When DDT needs to write into a read–only page, it changes the protection of that page, performs the write, and restores the protection.

`$$W` makes DDT respect memory protection.

`$1W` directs DDT to create a page when you tell it to deposit into a non–existant memory page; this is the default in TOPS–20.

`$$1W` tells DDT not to create a page when you attempt to deposit into a non–existant page.

The deposit commands generally work as follows. After you have opened a location (generally, by examining its contents), you may deposit a value to replace the contents by typing the new value and giving any one of the commands that follow.

To avoid an unintended deposit, you should close an open location before you type a new value or before you type another DDT command. Carriage return closes an open location. Don't type a new value before you type carriage return: carriage return is also a deposit command.

Section C.4 describes how to form the values to deposit.

`CR` If a location is open and new value has been typed, deposit that value and close the location. (If there is no new value, just close the location.) Any temporary display modes that are in effect are cancelled; further displays will default to the prevailing permanent display mode.

`LF` If a location is open and a new value has been typed, deposit the value, close the location, and open location `.+1`, i.e., the next consecutive location. Display the contents of that location in the current (temporary) mode. (If there is no new value, open `.+1`.) Change "`.`".

`^` If a location is open and a new value has been typed, deposit that value, close the location, and open `.-1`, i.e., the previous consecutive location. (If there is no new value, open `.-1`.) Display the contents of the new location in the current mode; change "`.`".

`Back Space` The Back Space command is the same as the `^` command.

`TAB` If a location is open and a new value has been typed, deposit that value, close the location, open the location (in the current section) whose in–section address is given by bits 18–35 of the value deposited. (If there is no new value, close the location and open the location, in the current section, whose in–section address is given by bits 18–35 of the current quantity; usually the current quantity will the contents of the open location.) This command does not clear temporary modes; it changes "`.`".

`addr\` Deposits any new value (i.e., the address argument to this command) and opens the specified location. Displays the contents in the current mode. Does not change "`.`". If no address is specified, the location addressed by the right half of `$Q` is opened.

`first<last>val$Z` Store the expression `val` into all words in the locations in the address range from the expression "`first`" to the expression "`last`". If `val` is omitted, zero is stored.

C.2 DDT Output Modes

In the commands that follow, use the escape key once to set the mode temporarily. Type the escape key twice in succession to set the mode "permanently." The temporary mode is cleared by the `CR` command. The permanent mode may be changed by a subsequent command that sets a new "permanent" mode.

`;` The semicolon character tells DDT to retype the current value in the current display mode. This command usually follows a command that changes the display mode.

`=` The equal sign tells DDT to retype the current value as numerals in the current radix. (See the discussion of `$`*n*`R` below.)

`_` The underscore character causes DDT to retype the current value in symbolic mode.

`$A` Set the display mode for addresses to absolute. Location addresses will be typed as numbers, rather than as symbolic. This mode can be undone by the `$R` command.

`$C` Display as a fullword constant in the current radix. If the radix is octal, this is the same as `$H`, otherwise, it is the same as `$nR`.

`$F` Display quantities as either floating–point or decimal integer. DDT scrutinizes the quantity that is being displayed; if it looks as though it might be a normalized floating–point number, DDT will display it as a floating–point number. Otherwise the quantity will be displayed as a radix `10` integer.

`$2F` Interpret the words at `.` and `.+1` as a double precision floating–point quantity and display the value.

`$3F` Interpret the words at `.` and `.+1` as a double precision integer and display the value.

`$4F` Interpret the words at `.` through `.+3` as a quadruple precision integer and display the value.

`$H` Display quantities in halfword format.

`$`*n*`O` Display quantity as left–justified *n*–bit bytes. The number *n* is interpreted as decimal. If *n* does not evenly divide 36, then one extra byte will be output, but that byte represents some smaller number of bits. The extra byte will be displayed with extra zeros added at the *right*.

If *n* is zero, the contents of the byte typeout mask word, `$3M`, defines the locations of the byte boundaries. Each 1 bit in the byte typeout mask defines the right end of a byte. For example the command `1061,,1$3M` will set the byte typeout mask to display the 9–bit opcode, the 4–bit accumulator, the indirect bit, the 4–bit index register, and 18–bit address fields of a word.

If *n* is omitted, the value of *n* set by the previous `$`*n*`O` or `$$`*n*`O` command will be used. The `$$`*n*`O` command makes the setting of the byte typeout mask "permanent."

`$R` Display location addresses in relative (i.e., symbolic) mode.

`$`*n*`R` Set the display radix for numbers to the decimal value *n*; *n* must be larger than 1. This affects values when they are typed as numbers, e.g., following "`=`", or when no satisfactory symbolic value is known for a word or a field.

When n is 8, values that have zero in the left half are typed as themselves; other values are typed as `left,,right`. For example `-1=777777,,777777`.

When n is 10, values are typed as signed integers and they are followed by a decimal point, e.g., `-12=-10.`

For other values of n, the quantity is typed as a unsigned number in the requested radix. When n is in the range from 11 to 36, letters of the alphabet are used for digit values larger than 9: "A" means 10, etc. This is useful for hexadecimal values. When n is greater than 36, other ASCII characters are used for the larger digits.

`$S` Display the contents of locations in symbolic mode, i.e., as instructions.

`$1S` display locations in symbolic mode, but prefer the machine instruction names over user–defined opcode OPDEF names.

`$T` Display quantities as seven–bit ASCII text. DDT tries to decide if the quantity is left–justified text or right justified text, and displays the quantity accordingly. Sometimes, DDT guesses wrong, in which case the command `$7O` is helpful.

`$0T` Display the contents of `.` and the locations following it as seven–bit ASCII text. Stop when a null byte is found. `.` is not changed.

`$6T` Display quantities in the "sixbit" subset of ASCII. The "sixbit" format is more prevalent among TOPS–10 systems; there it is used for file names. The sixbit character set maps the ASCII character codes in the range from octal 40 through 137 into the range 0 to 77. This makes the letters, digits, and some special characters fit into six bits, or six characters per word. Sixbit notation is a useful format for identifier names that are limited to six characters (as in the assembler, for example).

`$8T` and `$9T` Display values as four characters of left–justified eight– or nine–bit ASCII.

`$5T` Display values in "radix 50" notation. Radix 50 is a format used to pack a 6–character identifier into thirty–two bits. Radix 50 is used in the symbol table that DDT interprets. The four remaining bits are used as flags that are associated with each symbol name.

C.3 DDT Program Control

DDT allows you to stop the execution of your program at specific instructions by the installation of breakpoints. This section describes breakpoints, single–stepping, and other ways by which you can use DDT to control the execution of the program.

`CTRL/Z` Exit from DDT. If you intend to resume debugging the current program, but need to get to the EXEC, this is the command. For example, if you want to save a program that includes breakpoints, you must exit from DDT by `CTRL/Z`. (If you were to leave DDT by typing `CTRL/C`, then DDT would not have the opportunity to put the breakpoints where they belong.)

`adr$G` Start execution at the location specified by the address expression. If the address expression is omitted, DDT will start the program at the same address that the EXEC `START` command would use.

`$`*n*`G` Start execution at location *n* (an octal number) in the program's entry vector.

`adr2<adr1$`*n*`B` Install a breakpoint.

The address expression `adr1` specifies the location of the instruction at which to place the breakpoint. (DDT cannot place a breakpoint at location zero.)

The optional decimal number *n* is the breakpoint number. If you omit *n*, the first available breakpoint number will be assigned to this new breakpoint. You may specify *n* to recycle an old breakpoint to a new location. If you exhaust DDT's supply of breakpoints (usually limited to eight, numbered from 1 to 8), you will have to select one to overwrite.

The optional address expression `adr2<` specifies the address of the location to automatically open and display whenever this breakpoint is hit. (DDT cannot automatically display location zero.) If you omit the expression `adr2<`, DDT will automatically open `adr1`, the breakpoint location.

(This form of command to install a breakpoint is new in the version of DDT that became available in release 4 of TOPS–20. In release 3A and prior versions of TOPS–20, breakpoints are installed by the command `adr2,,adr1$nB`. Again, `adr1` is the location of the instruction where breakpoint "n" will be placed. `Adr2` is the address of the location to automatically examine when the breakpoint is hit.)

DDT creates a breakpoint by replacing the instruction at `adr1` with an instruction that jumps to DDT. When the computer executes the breakpoint instruction, DDT gains control. Two cautions:

- If you want a breakpoint in a read–only psect, you must tell DDT to override memory protections. Do so with the the command `$W`.
- Do not place a breakpoint on a location that may be considered data. For example, `ERJMP` and `ERCAL` have no effect when executed as instruction by the computer. The entire effect of these "instructions" is created by TOPS–20 when it finds one of these following an instruction that caused an error. If you put a breakpoint on an `ERJMP`, the breakpoint will hide the `ERJMP` from TOPS–20.

When a breakpoint is installed by a command with two consecutive escape characters, e.g., `adr1$$B`, DDT will proceed from the breakpoint automatically. Automatic proceed continues until DDT detects that characters are available from the terminal when the breakpoint is executed. See also the `$$P` description below.

The symbolic name `$`*n*`B` can be used to address the breakpoint block associated with breakpoint number *n*. The breakpoint block contains several interesting data items associated with each breakpoint. For example, `$`*n*`B` contains the address of the instruction breakpointed by breakpoint *n*. The locations following `$`*n*`B` contain such information as the proceed count, the address of the cell to display automatically, and the conditional instructions that can be executed before deciding whether to stop on any particular occurrence of the break.

`$B`	Remove all breakpoints.
`0$`*n*`B`	Remove breakpoint number *n*.
`$P`	Proceed from the present breakpoint, or following the previous single–stepped instruction. Execution of the program resumes at full speed until another (or the same) breakpoint is executed.
n`$P`	Proceed *n* times from this breakpoint.
`$$P`	Proceed automatically until the breakpoint is executed while input characters from the terminal are available. That is, this breakpoint will be passed automatically until you type something.
`$X`	Single–step the next instruction. You must be at a breakpoint or you must have previously used `$X` to single–step an instruction. Repetitions of `$X` cause subsequent instructions to be single–stepped. After single–stepping, a `$P` command will resume the normal full–speed execution of the program. When an instruction is single–stepped, the argument and results of the instruction are displayed. Although single–stepping is a very slow way to find out what a program is doing, it may be worthwhile for novice users who are uncertain of the effects of particular instructions.[1] Single–stepping is a very fragile part of DDT. When a program uses the software interrupt system, results of single stepping may be unpredictable. (When a program uses the hardware interrupt system, as when debugging the timesharing monitor, the results of single–stepping are notably unpredictable.) The command *n*`$X` will single–step the next *n* instructions.
`$$X`	Single–step automatically and without typeout until the program counter reaches one or two locations beyond the address of the instruction that you are `$$X`–ing. This command is useful for "single–stepping" instructions that call subroutines. Single–stepping is a very slow process. If the subroutine that is being `$$X`–ed is complex, it would be better to insert a breakpoint after the call to the subroutine, and then execute the subroutine at full speed.
`instr$X`	If the expression `instr` contains a non–zero value in bits 0–8 (the opcode), then DDT will execute `instr` as an instruction. (If the expression is zero in bits 0–8, it is interpreted as a repeat count, as in the command *n*`$X`.

C.4 DDT Assembly Operations and Input Modes

As described above, DDT allows you to deposit new values into memory locations. First, open a location, then type a description of the new value. Finally, type one of the commands that deposits a value.

[1] Prior to Release 4 of TOPS–20, single–stepping does not always give the same result as full–speed execution. If a JSYS call is single–stepped, an `ERJMP` instruction that follows the JSYS in the normal instruction stream will not be visible to TOPS–20, because DDT copies the instruction being single–stepped elsewhere before executing it. If you suspect that your program is acting differently while you single–step it, use breakpoints instead.

To help you form the new values, several assembler functions are built into DDT. The general instruction format

```
OP AC,@Y(X)
```

is recognized and assembled as MACRO would assemble it. Specifically the names of the machine instructions are recognized by DDT. JSYS names that are used by the program are recognized. Symbols defined by the program are available for use by DDT's instruction assembler.

Neither literals, pseudo–ops, nor macros are available in DDT. However, DDT does provide commands for entering text and numbers.

DDT evaluates expressions using integer arithmetic. The operators "`+`", "`--`", and "`*`" work as you might expect for addition, subtraction, and multiplication, respectively. Because the slash character is used to open locations, division is signified by the apostrophe character (`'`).

A single comma in an instruction signifies the end of the accumulator field.[2] A pair of commas separates left–half and right–half quantities.

Parentheses may be used to signify the index register field. Technically, the expression that appears within parentheses is swapped (as in the `MOVS` instruction) and added to the word being assembled.

The at–sign character, "`@`", sets bit 13, the indirect bit, in the expression being assembled

The blank character is a separator and adding operator in the instruction assembler.

Numeric input is octal except that digits followed by a decimal point are radix 10. If further digits following the decimal point are typed, input is floating–point. A floating–point number may be followed by `E`, an optional plus or minus, and an exponent.

In addition to the input formatting functions described above, special commands exist by which various text formats can be entered:

`"/text/` — Left–justified ASCII text, in 7–bit bytes. Instead of the slash (`/`) you may use any character that doesn't appear in the text itself. Repeat that character to end the input string. A long input string will be stored in consecutive memory locations.

To enter the sequence CRLF you must type only CR. To get a LF alone, you may type LF. CR alone can't be had.

For example: `"\this is a sample\`

`"X$` — One right–justified ASCII character. Example: `"A$`

`$"/text/` — Left–justified sixbit text. Lower–case input is converted to upper–case.

`$"X$` — One right–justified sixbit character, e.g., `$"U$`

`text$5"` — The value of "`text`" in Radix 50.

`$`*n*`"/text/` — ASCII characters in left–justified *n*–bit bytes. In this command, *n* is a decimal number in the range 8 to 36.

[2] If an input/output instruction is being assembled, the comma signifies the end of the device number field.

C.5 DDT Symbol Manipulations

DDT can change the symbol table that MACRO supplies. The changes include adding new symbols, removing symbols, and suppressing symbols.

To say that a symbol is *suppressed* means that the symbolic disassembler in DDT will not consider this symbol name as a possible name to output. However, the definition of a suppressed symbol is available when you use the symbol name as input. Suppressed symbols are sometimes called *half–killed* symbols.

`sym$:` Open the symbol table of the program named "`sym`". (The program name is set from the first six letters of the first word that follows the `TITLE` statement in a MACRO program.)

When a program's symbols are opened, those symbols are available preferentially for symbolic input and output. In an environment where several separately assembled modules have been loaded together there may be repetitions of symbol names among the several modules. Opening a program's symbols serves to eliminate the ambiguity arising in such cases.

`$1:` Display the name of the currently open symbol table, if any.

`$$:/symname/` Use the program data vector named "`symname`" and its symbols. The "`/`" character may be any character that does not appear in the text "`symname`". (Set `$5M`.)

`$$1:` Display the name of the current program data vector, if any.

`$D` Suppress the last symbol typed. Retype the same value.

`sym$K` Suppress (half–kill) the specified symbol.

`sym$$K` Kill this symbol. This symbol is removed from the symbol table and will no longer be available for either input or output.

`sym?` Type out the name of each program module in which the symbol named `sym` is defined.

`sym$`n`?` Display the name and value of every symbol whose name begins with the letters of "`sym`".

Each radix 50 symbol in the symbol table has a four–bit flags field associated with it; by convention these flags are written as multiples of 4 in the range 0 to 74. If a value, n, is supplied, DDT will print a symbol only if its flags contain all the 1 bits contained in n. That is, `xyz$44?` will print `44,XYZZY` or `74,XYZ3A` but not `40,XYZA`.

The value 1 may be added to the flags in n. If 1 is present, only symbols from the currently open symbol table are printed.

`sym:` Define the symbolic name `sym` to have the value of the current location (i.e., "`.`"). If `sym` is already defined, this changes the old definition; otherwise, a new definition is added to the symbol table.

`val<sym:` Define (or redefine) the symbolic name `sym` to have the value specified by the expression `val`.

`sym#` While using DDT's instruction assembler (see Section C.4, page 649), the sharp sign (`#`) following the name of a symbol, `sym`, declares that symbol to be undefined; the symbolic name `sym` will be added to DDT's table of undefined symbols. Subsequently, when `sym` is defined (by one of the commands explained above) all places where `sym` was used will be fixed to reflect the value of the new definition. DDT is notably less powerful than MACRO; you would be well advised to use a text editor and MACRO to construct anything complicated.

`?` Displays the names of all the currently undefined symbols.

C.6 DDT Searches

DDT can be used to find particular data patterns that appear in single words. The commands used to locate such patterns are called searches. Every search is governed by a search range and by a mask.

The search range is specified by telling DDT the lowest and highest addresses of the locations in which to search.

The search mask allows you to specify which bits in every word are to be considered significant to the search. Place a one in the mask wherever a bit is considered significant. Place a zero bit in the mask at every bit position that you want to ignore.

`$M` The symbolic name `$M` addresses the location that contains the search mask. Put 1 bits in the mask to signify bit positions that are significant to the search. Initially `$M` is set to −1. You can change the mask by the command `val$M`, where `val` is an expression that describes the new search mask. The state of the mask can be examined by the `$M/` command.

`first<last>val$W` Search for words that match the expression `val` in the locations within the address range from `first` to `last`. A word matches if the Boolean `AND` of the word and the search mask is the same as the value of `val`.

For every match found, DDT types the address and contents in the current display mode; "`.`" is set to the last address in which a match is found.

`first<last>val$N` Search for words that do *not* match `val` in the range between `first` and `last`.

`first<last>val$E` Search within the specified range for words in which the effective address matches the expression `val`.

C.7 Patch Insertion Facility

DDT offers a few commands that are useful for adding small fixes, called *patches*, to a program.

`$<` Begin a patch. The patch area is identified by the symbol `PAT..`, which is automatically generated by the loader program (LINK).

When this command is given, DDT remembers the address of the currently open location. DDT opens the patch area. You may write the patch by depositing instructions in the

patch area. When you have finished adding the new instructions, use the `$>` command to finish the patch.

`$>` End a patch. After you have finished typing the patch, use this command. DDT will copy the instruction from the location you are patching (that it remembers from the `$<` command) to the next location in the patch area. Then it inserts the sequence `JUMPA 1,LOC+1` and `JUMPA 2,LOC+2` where `LOC` represents the location being patched. The symbol `PAT..` is next redefined to address the next available location in the patch area. Finally, `LOC` is patched with a `JUMPA` to the first location occupied by the patch.

This order of events makes it certain that the patch is not inserted until it has been entirely composed. The use of the `JUMPA` instruction rather than `JRST` is used to signal the reader that a patch is present. The use of two `JUMPA` instructions to end the patch ensures the correct operation of the patch, should the final instruction be some kind of skip. The redefinition of `PAT..` allows for subsequent use of the patch facility without clobbering previous patches.

An optional argument, n, given as `$`n`>`, tells DDT to end the patch using n `JUMPA` instructions to allow for the possibility of the final instruction skipping up to $n-1$ times.

`$$<` Begin a patch. In most respects this is similar to the `$<` command. The difference is that the patched instruction (at `LOC`) is copied by DDT to the start of the patch area, instead of being copied at the end of the area as in `$<`.

An address argument can appear before either the `$<` or the `$$<` commands. If present, the address specifies the location of the patch area. No symbol name will be updated when the patch is ended.

C.8 Location Sequence

A *sequence* is started by any command other than `LF` or "`^`" that changes "`.`". Generally, a sequence is begun each time you mention an address to examine. The last location in the current sequence is remembered in a stack whenever a new sequence is started. The most recent old sequence can be resumed by the commands shown below.

The stack is actually a circular list of old sequence locations. In most cases it acts like a stack, but the depth of the stack is limited; after too many "pops", it will reverts to an old sequence.

`$CR` Return to the previous sequence of locations. Display the last location in the previous sequence.

`$LF` Return to the previous sequence of locations. Display the last location plus one in the previous sequence.

`^` Return to the previous sequence of locations. Display the last location minus one in the previous sequence.

Example:

```
100/   ..... LF             (Start a sequence at 100. Examine 101.)
101/   ..... 523/  ..... LF (Start a sequence at 523. Old seq. is 101)
524    ..... $LF            (Return to previous sequence, at 101+1)
102/   .....
```

C.9 Miscellaneous Features

`$M` specifies the search mask, used in `$N` and `$W` commands as explained above.

`$1M` specifies the terminal control mask.

`$2M` specifies the symbol offset quantity.

`$3M` specifies the byte typeout mask. See the discussion of the `$nO` command, above.

`$4M` the right half, if non-zero, specifies the local–address in *every* section of a 100–word table that DDT uses for such things as $X processing. By default, this address is set to 777700, so DDT may use the last portion of every one of your sections. To disable this feature, set $4M to zero.

`$5M` specifies the address of the current PDV, *Program Data Vector*, if present. The selected PDV defines which symbol table is currently open. Normally, this value is loaded by the `$$:/symname/` command, above.

`$6M` specifies the "permanent" default section number, i.e., the section number to be used when only 18 bits of address are specified.

`$.` refers to the current `$x` PC, that is the location at which to resume program execution or from which to fetch the next instruction to be single–stepped.

`$$.` refers to the previous `$x` PC. It is the PC from which the current instruction is being single–stepped.

C.10 FILDDT

FILDDT is a version of DDT adapted to looking at files.

FILDDT can examine (and patch) files, including `EXE` files.

(If you patch the `EXE` file of a program while it is being executed, the memory image of the executing program will be modified if it includes the file page that you patch.)

C.11 EDDT and KDDT

EDDT is a version of DDT designed to run on the bare computer without benefit of an operating system. One use of EDDT is as a component of TDBOOT, the TOAD bootstrap program.

KDDT is TOPS–20's kernal mode DDT. KDDT is part of the operating system itself; it knows TOPS–20's symbol table. KDDT interacts via the terminal that the computer knows as its console. When KDDT is running, timesharing is suspended. KDDT allows a programmer to add breakpoints to the monitor and to debug or troubleshoot the monitor.

We shall refer to KDDT and EDDT as executive DDTs.

A small number of special commands are available in the executive DDTs. Among these are the family `arg$`*n*`U`, which invokes various features broadly lumped under the name "mapping". The value *n* is a decimal number by which the user selects a particular subfunction.

`$$U` Set physical address mode. Subsequent to this command, the user supplies physical addresses and an executive DDT reads or writes the selected locations. (The format of physical addresses differ from system to system. On the KL10, a physical address is just a number. On the TOAD system, a physical address is a 36–bit bus address word, composed of the device bit, a slot number, and an in–module address.)

`$U` Set virtual address mode: undo `$$U`. Addresses are viewed through the Executive map, as defined by the EPT and the contents of the map pointers found from there.

k`$4U` Select accumulator block *k*. Various computers have multiple register blocks. This command tells an executive DDT which block to use when examining or depositing the registers.

Executive DDTs have special locations in which they store hardware values. Among these is `$I`, which contains the state of the hadrware interrupt system as of the moment when the executive DDT was started. (This data is obtained by the `CONI PI,` or `RDPI` instructions. By examining the contents of the location called `$I`, the programmer can determine the interrupt system state when this DDT was activated. DDT uses this value (and others related to it) to restore the hardware state when proceeding from a breakpoint, etc.

There are many other executive DDT functions hiding under the "`U`" and "`I`" command characters. If you need to know them, you'll have to read the program.

C.12 MDDT

Monitor DDT is invoked by JSYS number 777. It runs the MDDT module of TOPS–20. MDDT has access to TOPS–20 symbols, code, and the present job and process variables.

MDDT can be used to examine the running monitor's data structures. If you're brave, or foolhardy, you can use MDDT to patch of TOPS–20 while timesharing continues. (Any attempt to use breakpoints is ill–advised.) Caution is warranted at all times.

Exit from MDDT via `Ctrl/Z` or by `MRETN$G`.

Appendix D

Obsolete Instructions

Programming style and tastes change by the passage of time. Some of these instructions fell from favor as newer techniques and implementations became available. In any case, these are included here for the sake of completeness.

D.1 JSA – Jump and Save AC

The JSA instruction combines some of the features of JSR and JSP. Like JSR, JSA stores into the instruction stream. Like JSP, JSA stores the return address in an accumulator where you can do something with it. Better than JSP it preserves the accumulator; better than JSR it supports multiple–entry points. Unfortunately, it doesn't bother to save the PC flags.

The instruction JSA AC,E first stores the selected accumulator in E. Then it stores a word consisting of E in the left half and the return PC in the right half in the selected accumulator. Finally, it jumps to E+1.

```
JSA     C(E) := C(AC); C(AC) := <E,,PC>; PC := E+1;
```

The advantages of the JSA instruction are that it preserves the accumulator, it supports multiple–entry points, and it is easy to find (and skip over) arguments that follow the JSA instruction. The disadvantage of JSA is that it is impure (i.e., it stores into the instruction stream), and it is hopelessly linked to the old 18–bit addressing mode.

The JRA instruction unwinds this call.

D.2 JRA – Jump and Restore AC

JRA is the return from JSA. This instruction loads AC from the address contained in the left half of AC. It then jumps to the effective address.

```
JRA     C(AC) := C(CL(AC));   PC := E;
```

Programming example:

```
        JSA 16,SUB3
        0,,ARG1         ;pointer to (i.e., address of) first argument
        0,,ARG2         ;pointer to second argument


SUB3:   0               ;Save AC number 16 here
        MOVE 0,@0(16)   ;load first argument
        MOVE 1,@1(16)   ;load second argument
        ...
        JRA 16,2(16)    ;restore 16. Return to 2 past caller's PC
                        ;(i.e., skip over argument list)
```

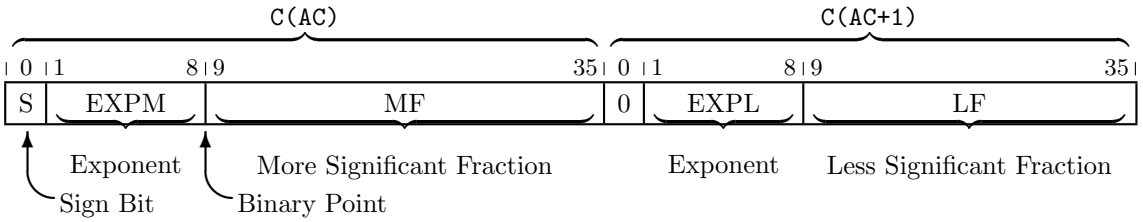
On the balance, JSA and JRA are almost never seen in current coding. PUSHJ and POPJ, even though they might need an additional register to facilitate argument passing, are most commonly used.

D.3 Long Floating–Point

The KA10 lacks double–precision floating–point hardware. However, there are several instructions by which software may implement double–precision. These instructions are DFN, UFA, FADL, FSBL, FMPL, and FDVL. Users of the DECSYSTEM–20 are strongly advised to avoid using these six instructions. These instructions are inferior in precision and speed compared to the DF class that are available in the newer processors.

FADL	C(AC AC+1)	⇐	C(AC) + C(E); floating–point arithmetic.
FSBL	C(AC AC+1)	⇐	C(AC) − C(E); floating–point arithmetic.
FMPL	C(AC AC+1)	⇐	C(AC) × C(E); floating–point arithmetic.
FDVL	Temp1	⇐	Quotient of C(AC AC+1) ÷ C(E); floating–point arithmetic.
	Temp2	⇐	Remainder of C(AC AC+1) ÷ C(E); floating–point arithmetic.
	C(AC)	⇐	Temp1; C(AC+1) ⇐ Temp2

The long results of FADL, FSBL, and FMPL, and the long operand of FDVL have this format:



In this diagram, S represents the sign bit. EXPM is the exponent of the most significant word; EXPL is the exponent of the least significant word. EXPL is smaller than EXPM by decimal 27 (octal 33). MF is the most significant fraction part; LF is the fraction of the least significant word. There are a total of fifty–four bits of fraction.

In a negative number, the most significant word together with the LF fraction bits are represented

in two's complement. Bit 0 of the least significant word is always zero. The exponent in the least significant is always represented in positive form, even in a negative number.

This format is known as KA10 or *software* format for double–precision floating–point. This format is not used in the DECSYSTEM–20.

D.4 DFN – Double Floating Negate

The `DFN` instruction is used to negate the KA10's software–format double–precision floating–point numbers.

```
DFN     C(AC,E) := - C(AC,E);   interpret C(AC,E) as a KA10 double
                                precision floating--point number.
```

`DFN` treats `AC` and `E` as a KA10 double–precision floating–point number which it negates and stores back. `AC` is the high–order word. Usually the low–order word is in `AC+1`, so the instruction most often appears as `DFN  AC,AC+1`.

D.5 UFA – Unnormalized Floating Add

In the KA10, the `UFA` instruction is used to assist in software format double precision arithmetic. `UFA` will add `C(AC)` to `C(E)` and place the result in `AC+1`. The result of `UFA` will not be postnormalized unless in the original operands the exponents and signs were the same and a fraction with magnitude greater than or equal to `1` was produced. Only in this case will a one–step normalization (right shift) occur. `UFA` will overflow in the same circumstances as `FAD`. Underflow is not possible.

```
UFA     C(AC+1) := C(AC) + C(E);        floating addition, without
                                        postnormalization
```

Appendix E

Common Pitfalls

There are a number of errors that are commonly seen among novice users of assembly language. This list is by no means exhaustive, but at least some of the most common errors can be avoided or repaired by reference to this list.

- The assembler input file should consist of entire lines. Although the EDIT program won't allow you to make this mistake, some of the Teco–based display editors, EMACS and VTED among them, allow you to end the file without a carriage return and line feed following the `END` statement. Unless the carriage return and line feed is present in the file, the `END` statement cannot be seen. As there are many other reasons why the `END` may become invisible, you should take pains to avoid this one.
- The assembler error *Missing End* means that either there was no `END` or that MACRO couldn't see the `END`. Apart from the missing carriage return and line feed on the `END` statement, there are a number of causes of *Missing End*. The most common error is to omit the matching delimiter in `ASCIZ`, `COMMENT`, or any of the related pseudo–ops that accept multiple lines. Another similar way to obtain this problem is to omit the closing pointed bracket in a macro definition.
- If you are using conditional assembly, count the pointed brackets carefully. An unterminated conditional can cause the remainder of the file to be skipped. Macro arguments also can be a problem. Complex arguments are enclosed in pointed brackets; if you omit the closing bracket, a large part of the program can be "eaten." Beware of pointed brackets in comments! The macro processor does not understand the difference between a comment and regular text as it skips over un–assembled text. An extra `<` will cause the closing bracket to become invisible. An extra `>` will cause the conditional material to end too soon.
- When you request a cross–reference listing be sure to process the resulting file through the CREF program. Otherwise, you'll be very surprised when you see the results.
- When assigning a value to a symbol name, be sure the equal sign immediately follows the symbol name:

```
Right           Wrong

A= Z + 32       A = Z + 32
```

- Values are in octal, but MACRO happily swallows numbers that have the digits "8" and "9" in them. If you refer to accumulators 8 and 9 you may discover that registers 10 and 11 seem to change mysteriously.
- Some obscure errors can result when you select a variable name that is the same as either a pseudo–op or a macro name. Among the favorite manifestations of these problems are P (*Phase*) and Q (*Questionable*) error indications. This error is especially likely when large macro packages are used, e.g., MACSYM.

Generally speaking, assembly errors are relatively easy to fix. The errors in the logic and coding of the program itself are harder. A logical, thoughtful approach to the problem using DDT to obtain information and insight is usually called for.

Some of the more common types of runtime errors are listed below.

- Improper loop indices. Loops that don't terminate. Loops that write into the wrong memory locations via incorrect use of index registers.
- Failure to initialize the stack pointer register.
- Failure to explicitly zero registers, variables and arrays at program initialization. Failure to perform a RESET JSYS at initialization.
- Erroneous arguments to JSYSes, subroutines.
- Always write both a carriage return and a line feed in any output stream to signify end of line. Although some devices may need only one of these, the majority of terminals and printers will work correctly when the sequence carriage return, line feed is sent to them.
- Did you use indirect addressing through a byte pointer? It works fine until you increment the pointer. Try indexed addressing instead.

Appendix F

References

[BELL] Bell, C.G., J.C. Mudge, and J.E. McNamara: *Computer Engineering: A DEC View of Hardware Systems Design*. Maynard, Mass., Digital Press, 1978.

[KNUTH 1] Knuth, D.E.: *The Art of Computer Programming: Fundamental Algorithms*, Vol 1, Second Edition. Reading, Mass., Addison Wesley, 1973.

[KNUTH 2] Knuth, D.E.: *The Art of Computer Programming: Seminumerical Algorithms*, Vol 2, Second Edition. Reading, Mass., Addison Wesley, 1981.

[KNUTH 3] Knuth, D.E.: *The Art of Computer Programming: Sorting and Searching*, Vol 3. Reading, Mass., Addison Wesley, 1973.

[LINK] *LINK Reference Manual*. Maynard, Mass., Digital Equipment Corporation, 1978. AA–4183B–TM.

[MACRO] *MACRO Assembler Reference Manual*. Maynard, Mass., Digital Equipment Corporation, 1978. AA–4159C–TM.

[MCRM] *TOPS–20 Monitor Calls Reference Manual*. Maynard, Mass., Digital Equipment Corporation, 1980. AA–4466D–TM.

[MCUG] *TOPS–20 Monitor Calls Users Guide*. Maynard, Mass., Digital Equipment Corporation, 1976. DEC–20–OMUGA–A–D.

[SYSREF] *DECsystem–10/DECSYSTEM–20 Processor Reference Manual*. Maynard, Mass., Digital Equipment Corporation, 1980. AA–H391A–TK.

Glossary

AC	An abbreviation for *accumulator, qv.*
Accumulator	In the PDP–10, any one of the first sixteen locations, addresses `0` through octal `17`. The accumulators are used to hold intermediate results of arithmetic operations, flags, stack pointers, index values, and other data of frequent interest. Often the word *register* is used to refer to an accumulator.
Address	An address is a number that expresses the location of an item in memory. Such items may be either instructions or data. When used as a verb, "address" means "refer to." See also *effective address.*
Address Section	in extended addressing is a 2^{18}–word region of address space aligned at an address that is a multiple of 2^{18}.
Algorithm	An algorithm is a completely specified computational procedure that terminates. By completely specified we mean that at no point in the process is there any doubt about what to do next. The procedure should not perform arithmetic operations that are undefined, e.g., division by zero, etc. The procedure should terminate, i.e., come to an end, eventually.
Array	An array is an organized collection of data items, in which each item is uniquely identified by one or more index numbers. Arrays are usually stored so that some arithmetic function of the given indices specifies the address of the desired data item.
ASCII	The American Standard Code for Information Interchange. This code is the convention that is followed in the DECSYSTEM–20 by which we specify the internal representation of characters. The table of ASCII characters is given in Section 4.8, page 38.
Assembler Program	A program that translates mnemonic instruction names into the binary patterns that the central processor can execute. Usually an assembler understands user–defined symbolic names, has the ability to create binary patterns for data as well as for instructions, and provides a great many bookkeeping facilities to make programming at this level more productive. The name *assembler* comes from the observation that this translation process usually involves reading several fields of the input program and placing values for those fields into the output; the output is effectively assembled from various fields of an input line.
Binary	Radix `2` notation for the representation of numbers. Most computers are composed of electronic, magnetic, and mechanical storage elements, each of which

is fundamentally a two–state device. Because of the two–state nature of the storage devices, the computer relies on binary representations for all the numbers, characters, and instructions stored in the computer. A full discussion of representations appears in Section 4, page 29.

Bit — A binary digit. A bit is the smallest unit of storage in a computer. A bit is a two–state storage item whose possible values are typically called `0` and `1`. Other interpretations such as True/False are also possible.

Breakpoint — A debugging tool. The debugging program, DDT, allows the programmer to place breakpoints at selected locations in the program. When the normal course of execution reaches one of these breakpoint instructions, the program will jump to the debugger. The programmer can then examine the state of the computation, with the intention of determining the cause of any errors that have occurred.

Bus — A collection of wires that carry information between distinct computing subsystems. A bus generally has two sections: control and data. The control section of the bus allows one of the parties on the bus to address a specific other party and to *request* a specific *response*. Data is passed between the requestor and the selected unit by means of the data section of the bus.

In the DECSYSTEM–20 buses are used to connect the processor to the memory; the processor to I/O devices; controllers to the memory; and controllers to devices. The various connections are shown in Figure 32.1.

Byte — In the PDP–10, a byte is any contiguous group of bits within a computer word. Bytes can be used for storing characters, or other data where using an entire word for each data item would be too wasteful. In other computers, *byte* has come to mean an 8–bit memory location at a fixed position with respect to word boundaries.

C Bus — In the KL10–based DECSYSTEM–20, the C bus connects the processor's memory control unit (M box) to the Massbus controllers. The C bus passes all data that is transferred between memory and the Massbus disk and tape devices.

Central Processor — This is the part of the computer where arithmetic, logic, and control functions are performed. The PDP–10 CPU includes the sixteen accumulators, the program counter, and the necessary data paths and logical testing operations to effect data movement, arithmetic, and decision making based on computed results.

Channel — An *Interrupt Channel* is a software entity that is associated with an event. Some channels are permanently associated with particular events such as stack overflow, file data errors, reference to non–existent pages, etc. The other channels can be assigned by the programmer to various events such as terminal interrupts, IPCF interrupts, and program initiated interrupts. There are thirty–six interrupt channels; each channel corresponds to a bit in one computer word.

In hardware terminology, a *channel* is a device that serves as a conduit for data moving between memory and a peripheral. Usually, a channel is capable of fetching or storing sequential data locations without the intervention of the CPU; this leaves the CPU uninvolved in the details of implementing input and output operations.

Character — Unit of storage for text data. Each character is encoded using some convention, e.g., the ASCII code. Characters are often stored in bytes.

Code — From the observation that instructions and data are encoded into binary patterns and stored in the computer: any data in memory, especially sequences of instructions. The detailed work of writing the correct instruction sequences, in contrast to the designing or organizing the problem solution, is called *coding*.

Complement — Unless modified, this usually means the Boolean negation, called the *one's complement* of a quantity. The complement of a bit pattern is obtained by changing all zero bits to one, and all one bits to zero.

When truth values, `TRUE` and `FALSE` are used, usually `TRUE` is represented by one or a series of bits that all have value one; then `FALSE` is the complement of `TRUE` and is represented by one or more zero bits.

Conditional — In the assembler, a conditional operator controls what text gets assembled. At run time, a conditional instruction directs the flow of program control.

CPU — See Central Processor.

Cross–Reference — A program listing that is augmented by the inclusion of a table that for each symbol displays the line number on which the references to and defining occurrence(s) of the symbol appear. Also, the CREF program that produces these augmented listings.

DDT — Dynamic Debugging Technique. This is the name of the interactive debugging program; for details, consult Appendix C, page 643.

Decrement — Any decrease in value. Most usually a decrease in value by one.

DEC — Digital Equipment Corporation, the manufacturer of the DECSYSTEM–20, and of its predecessors, the DECsystem–10 and PDP–6. Digital also makes 16–bit computer systems, the PDP–11 family, and 32–bit systems, the VAX family.

DECsystem–10 — A computer system that includes a PDP–10 processor and which uses the TOPS–10 operating system. Historically, the DECsystem–10 is the predecessor of the DECSYSTEM–20, and as such has been influential in many ways on the development and organization of the DECSYSTEM–20 hardware, operating system, and utility software. Many programs that are run on the DECSYSTEM–20 were originally written for the DECsystem–10. Included among these are the MACRO–10 assembler and LINK–10 loader.

DECSYSTEM–20 — A timesharing computer system. Programming the DECSYSTEM–20 is the subject of this book. The DECSYSTEM–20 includes a PDP–10 central processor (at present, a KL10 or KS10 processor), and the TOPS–20 operating system and its related software.

Default — The action taken by a program in the absence of action or information to the contrary. As in a default value for a field in a command, or for a switch or assembly parameter in a program.

Directory — On the disk, a file that contains the names and pointers to other files. Also, the listing of the contents of a directory file.

Disk — A mechanical memory made from rotating magnetizable surfaces. A disk is much slower than registers or main memory, but it allows long–term storage of data in files whose names are kept in directories. Systems are typically configured with disks that contain 10 to 1000 times as much data as the main memory.

E Box — In the KL10 processor, the E box is that portion of the processor that actually executes instructions. The E box also exerts control over and responds to peripheral devices by means of the E bus.

E Bus — In the KL10–based DECSYSTEMs–20, the E bus connects the processor's execution unit (E box) to the various peripherals units. Among the attached peripherals are the Massbus controllers and the DTE20 interface to the PDP–11 front–end processor. More elaborate systems may have additional communications processors or the adaptor that creates an external Input/Output bus to which further devices can be attached.

Effective Address — The actual address that a PDP–10 instruction refers to. The effective address takes into account the effects of index registers and indirect addressing. In the traditional PDP–10 architecture, effective addresses are limited to eighteen bits; see Section 5.3, page 45. In the extended PDP–10, addresses can be thirty bits, as explained in Section 30, page 565.

Entry Vector — In TOPS–20 the entry vector defines the starting address of the program. The entry vector is read by the EXEC and by DDT to decide where to start a program. The EXEC command `REENTER` also inspects the entry vector. See also Section 26.5.2, page 499.

EXEC — A component of the TOPS–20 environment. The EXEC is the program by which users communicate their desires to the monitor, and through which the monitor communicates to the user.

Exec Mode — A processor mode in which the user mode restrictions do not apply. The TOPS–20 monitor normally operates in Exec mode. Just to confuse terms, the TOPS–20 EXEC, operates in user mode, with all user mode restrictions applying.

Execute — To fetch an instruction from memory and perform the function that it specifies. By extension, to perform the sequence of instructions that comprise a subroutine or an entire program.

Fetch — To read a data item or instruction from memory.

Field — A portion of a command, a record, or a word.

File — A named collection of information, often resident on the disk. A file is identified by its name; files have *attributes*, such as length and time of creation. Files also have contents, the actual information stored in the file.

Flag — A two–state quantity. Conventionally, flag quantities take on either the value `TRUE` or the value `FALSE`. Flags are most compactly represented as one bit, but in some cases, it convenient to use an entire PDP–10 word for a flag. Also, the flags associated with the program counter; see Appendix A, page 635. See also *switch*.

Fork — See *process*.

Global — A symbol that is defined in one program and referenced in another. Global symbols are created or referenced in three pseudo–operators. An `INTERNAL` symbol is a global that is defined in the present program. An `EXTERNAL` symbol is a global that is defined elsewhere than the present program. When a symbol in the present program is the name of a subroutine in a *library*, it must be declared as an `ENTRY` symbol: an entry symbol is an internal global symbol that can be found by the LINK program in a library search.

The presence of an `EXTERNAL` symbol in a MACRO program results in a *global request* for that symbol being placed in the `REL` file. Among LINK's functions is the resolution of such requests by loading additional modules and by searching libraries. In addition to the `EXTERNAL` pseudo–operator, MACRO allows the programmer to declare a symbol to be external by placing the characters `##` after any mention of the symbol's name.

Hash Table — A data structure to implement a partitioning of a search space for the purpose of making searches more efficient.

Heap — A data structure used in the Heapsort algorithm. A heap is a binary tree in which the father of each node bears a specific arithmetic relationship (e.g., larger) to its two sons.

Increment — Any increase to a quantity, but especially an increase by one.

Index — A quantity used to select a particular array element; an index register, where an index quantity can be placed to effect that selection; an addressing mode that uses an index register; any distinguishing mark, as the index mark on a disk; a table, as an index page, that points to data pages; a sorted table of keys that points to the data related to those keys; to step from one value to the next, as in advancing an indexable file handle.

Indexed Address — A mode of address in the PDP–10 in which an index register is used to modify the effective address. See also *effective address*.

Indirect Address — A mode of address in the PDP–10 in which the direct or indexed address serves to select a word whose contents are then used in the further calculation of an effective address. See *effective address*.

Input — The operation of conveying information from the external world into the main memory of the computer. Also, the particular information that is thus conveyed.

Instruction — One of the set of fundamental operations that the computer hardware can perform. Instructions specify the nature of the particular operation that is desired, and the data on which this operation is to act. Typical operations include arithmetic (add, subtract, etc.), transfer of control, comparison of numbers, etc. Assembly language programs consist of a sequence of instructions. When the program is being executed, the instructions are kept in main memory.

Interrupt — An asynchronous event that changes the normal instruction sequence. In TOPS–20, software interrupts can be enabled to alert the program to specific keyboard characters, incoming IPCF messages, and other events. The processor hardware also implements an interrupt mechanism so that events external to the

processor (or asynchronous to the program's operation) can change the normal instruction sequence.

IPCF — Interprocess Communications Facility. A collection of monitor calls and systems programs that implement message passing between programs. See Section 28.1, page 515.

JFN — Job File Number: the handle assigned by TOPS–20 by which a program can access a file; see also Section 19.1, page 277.

Job — In TOPS–20, a job is a software entity that consumes computer resources. A job is composed of one or more processes; each process can run a program.

JSYS — Jump to System. This is the instruction by which a user program transfers control to the TOPS–20 monitor for the purpose of obtaining a particular service. Monitor calls, i.e., calls to subroutines in the monitor for performing input/output and other necessary functions, are made via the JSYS instruction. The various different monitor functions are selected by modifying the particular JSYS instruction. Most often, these functions are simply called by their names, e.g., `PSOUT`, `GTJFN`, etc.

Jump — A change to the program counter, for the purpose of effecting a change from the normal sequence of instructions that would be obtained by simply incrementing the program counter.

KL10 — The manufacturing designation for the central processor that appears in the DECSYSTEM–2040, 2050, and 2060 configurations. The KL10 processor is also present in DECsystem–10 configurations including the 1080, 1090, and 1091 systems.

KS10 — The manufacturing designation for the processor in the DECSYSTEM–2020 configuration. The KS10 is also capable of running a version of the TOPS–10 operating system.

Label — In MACRO, a symbolic name for a specific location in the instruction sequence or in the data area. Labels are defined when a symbolic name and a colon appear at the beginning of a line. In contrast to other symbols, labels are assigned a value implicitly, from the current location counter. Moreover, labels can not be redefined.

Library — A collection of useful subroutines, most often in `REL` file (relocatable) form. The LINK program has the facility to search a library file for definitions to resolve as yet undefined external requests. In dealing with a library file, LINK loads only those modules that are demanded by unsatisfied global requests.

List — A data structure characterized by pointers connecting one element to the next. Often *list* is used to mean a one–dimensional structure, but lists can be generalized to include quite complex data structures.

Load — Generally to copy data from a general memory location to an accumulator. Also, to fill memory with a program or data; this is the function of the LINK program.

Location Counter — In the MACRO assembler, the location counter holds the address of the memory location into which the next instruction or data word will be assembled. In the normal course of operation, each time that MACRO assembles a word, it increments the location counter to point to the next address. In MACRO, the current value of the location counter can be referenced by the symbolic name "." (period).

LUUO — Local Unimplemented User Operation: these are thirty–one operation codes that are not assigned to explicit instructions. When one of these is executed the processor traps in a specified way. These can be used to implement subroutines in those cases where the more usual calling sequences are inappropriate.

M Box — The M box is the portion of the KL10 processor that controls system–wide access to the main memory. To make instructions execute more rapidly, the M box contains a high–speed buffer memory called a *cache*.[1] The cache allows the E box to use a memory system that is effectively twice as fast as the actual main memory. The M box also implements the data channels and the C bus that connects to the Massbus controllers; every data word transferred between a Massbus peripheral device and the memory passes through the M box.

Macro — The assembler for the DECSYSTEM–20. Alternatively, a macro is a text subroutine implemented by the assembler.

Map — A data structure that describes the location of other objects, especially the *page map* that describes the virtual memory seen by one process. As a verb, *map* means to change the contents of such a data structure.

Massbus — The Massbus is a collection of signal wires that connects between a *Massbus Controller* and one or more Massbus devices such as disks and tapes. The Massbus is defined to be independent from specific systems and implementations so that the same design of a peripheral unit can be used on several different computer systems. The Massbus is characterized by its ability to perform some commands and to read device conditions while simultaneously transferring data. The Massbus is especially suited for storage devices where blocks of data are regularly transferred between the device and consecutive memory locations.

Massbus Controller — The massbus controller is the system–specific interface between a particular processor/memory system and the massbus. In the KL10–based DECSYSTEM–20, the massbus controller is an RH20; in the 2020 system it is an RH11.

Memory — Main memory, as distinct from *registers*, *disk memory*, and other forms of storage, is that portion of the computer system in which the CPU can store or retrieve a limited amount of data very quickly; it is not permanent storage. Often, main memory is called *core memory* because for many years it has been made from magnetic cores. Presently, there is an increasing reliance on semiconductor memory rather than core. DECSYSTEMs–20 may have either core or MOS semiconductor memories, or both.

Monitor — The TOPS–20 operating system itself, as distinct from the EXEC and other components of the Tops–20 environment. See *operating system*.

[1] The cache is not present in the 2040 configuration.

MUUO — Monitor Unimplemented User Operation: these are operation codes that are not assigned to explicit instructions. In contrast to the *LUUO* codes, when one of the MUUO instructions is executed the processor traps to the operating system. The JSYS instruction is implemented as an MUUO.

The TOPS–10 operating system makes extensive use of MUUOs as monitor calls. It might be noted that when a TOPS–10 MUUO is executed on a DECSYSTEM–20, TOPS–20 obtains control. When TOPS–20 discovers that the MUUO is one of those implemented by TOPS–10, it passes the MUUO to the TOPS–10 compatibility package, where the effect of the MUUO is simulated.

NIL — See *null pointer*

Null Character — The ASCII character, octal zero, that often signifies the end of a sequence of characters.

Null Pointer — In list or record structures, a particular value that signifies that there are no further elements in this list or structure. The PDP–10 instruction set makes it convenient to use zero as the value of the null pointer.

Octal — Radix `8` representation of numbers. Octal is used because of its close relation to and easy interconvertibility with binary. See *binary*.

One's Complement — A value obtained by changing all zero bits in a quantity to one and all one bits to zero. One's complement also describes a system of representation of negative numbers, in which the negative of a number is the one's complement of the number itself.

Operating System — The program that generally controls the operation of the computer. In a time-sharing system, the operating system schedules user programs, allocates resources, controls devices, and performs other services for users that they could not practicably do for themselves. The TOPS–20 operating system schedules hardware resources and provides the services requested via JSYS operations.

Output — The operation of conveying information from the main memory of the computer to an external storage, network, or display. Also, the information that is conveyed outwards.

Page — In TOPS–20, a page is `512` computer words. Programs and files are composed of pages. As a verb, *page* means to move a page from disk into memory or vice versa.

Page Map — In TOPS–20, a page map describes the actual pages that compose a process or file. The page map of a process contains pointers to the specific pages in the virtual address space of that process. Among these pages may be *private pages* that belong to this process exclusively; *shared pages* which are common to several processes; and *file pages* which are appearances of actual disk file pages in the memory space of the process.

The page map of a file describes the actual disk addresses where the file pages are stored.

PC — An abbreviation for *Program Counter, qv.*

PDP–10 — This is the generic name we use for any central processor that implements Digital Equipment Corporation's 36–bit instruction set. In this book, the term *PDP–10* includes DEC's 166 central processor (found in the PDP–6 system), the KA10, and KI10 processors (DECsystem–10), and the KL10, and KS10 processors (DECsystem–10 and DECSYSTEM–20), and any future processors of compatible architecture.

PID — Process Identification, as used in the IPCF calls; see Section 28.1, page 515.

Process — In TOPS–20 a process is a software entity composed of memory space, accumulators, program counter, and related information. A process is the unit in which a program is executed.

Program Counter — A register contained within the central processor that generally contains the address of the next instruction to execute. In the PDP–10, the program counter normally increments during the execution of each instruction. This causes consecutive instructions to be fetched and executed from consecutive memory locations. Special instructions, called *jumps* and *skips* can be used to change the program counter to effect program loops and other forms of program control.

Program Section — A subdivision of a program, also called a "psect", that is loaded in a particular range of addresses.

Pseudo–Operator — An intrinsic function of the assembler program. These are used to generate data and to control the function of the assembler.

Pushdown List — See *stack*.

Read — To bring the contents of a memory location into the central processor. To bring data from an external source or device into main memory.

Record — A data structure, in memory or in a file.

Recursion — A technique of programming in which a subroutine performs its function by calling itself. In order for a recursive subroutine to perform its task properly two conditions must be met. First, the subroutine must somehow simplify the problem before invoking itself. Second, the subroutine must recognize some simplest case that it can handle without making a recursive call. In addition to these two criteria, such a subroutine must also obey certain rules about allocating temporary storage. Generally, a stack (pushdown list) is necessary in any implementation of recursion.

Register — Any collection of one–bit storage elements in which related information can be saved, and from which information can be retrieved. Often the word *register* is used interchangeably with *accumulator*.

An *index register* is an accumulator that is used to modify the effective address calculation of an instruction.

A *device register* is an array of storage elements that exists in some input or output device. Device registers contain the data that is transmitted between the device and main memory, or they are used to control the device and to report its status.

Relocatable — A characteristic of assembler output. A relocatable program is one which can be placed anywhere in the user's virtual address space, at the convenience of the loader program. Relocatable programs are useful because often a program is not executed by itself; sometimes there are are library subroutines and other information that must be loaded into the same memory space. By writing a relocatable programs, we remain uncommitted to specific addresses until the program is actually loaded. We can assemble various modules independently; if they are all relocatable we can load them together in a compatible way. The loader has to do extra work to cope with relocatable programs. Since the assembler output is most often in relocatable form, it is conventional to call the binary output a relocatable file, or `REL` file.

Representation — A convention by which we express numbers, real–world objects, and concepts, as binary patterns inside the computer. See Section 4, page 29.

S Bus — In KL10–based systems, the S bus connects the processor's memory control unit (M box) to the memory. To write in memory, the M box transmits an address, a write command, and the data along the S bus. A selected memory unit responds by accepting the data and storing it. To read from memory, the M box transmits an address and a read command; the memory responds by supplying the data and notifying the M box when the data is ready. The M box is actually capable of reading or writing up to four words for each address and command supplied. Up to four memory units are activated simultaneously; they accept (or supply) data from (to) the S bus in rapid succession. A modified version of the S bus, called the X bus, is present in those DECSYSTEM–20 configurations that have MF20 semiconductor (MOS) memory.

Skip — An instruction that changes the program counter by incrementing it one extra time. Most instructions increment the program counter once, to advance to the next instruction. A skip instruction would increment the program counter twice, effectively causing the computer to avoid the execution of the instruction that immediately follows the skip. In most cases, the skip is *conditional*, that is, the instruction will skip or not, depending on the status of the computation.

Stack — A data structure in which the last item added to the stack is the first item accessible for removal from the structure. Specific terminology applies to stacks. By analogy with cafeteria stacks of dishes, adding an item to the stack is accomplished by a *push* operation; removal of an object is done by a *pop*.

Store — Move (write) data from the CPU into memory for later use. Transmit information from memory to an external device. *Store* implies that the information can and will be retrieved at a later time by the computer; information can be stored (or *written*) on disk, but information can only be *written* to (never *stored* on) a printer.

Structure — A *Data Structure* is an organized collection of information. Among the forms of data structures described in this book are arrays, stacks, lists, records, hash tables, and heaps.

A *Control Structure* is a means of directing the flow of execution of a program. Among the common forms of control structure are top–test loops, bottom–test loops, recursion, and flags.

A *File Structure* in TOPS–20 is one or more disk units that together represent one file system. A file structure contains one root directory, various other file directories, one bit table, and user files.

A *Mountable Structure* is a file structure that can be mounted on (or dismounted from) physical disk units at will. Generally, in a TOPS–20 system, one structure is considered to be permanently mounted; conventionally the permanent structure is called `PS`, *Public Structure*. Any other structure in the system is mountable.

Switch — A means of controlling the state or function of a program. See *flag*. Sometimes a switch is distinguished from a flag by the switch being accessible to the user of the program as a means of choosing program options. For example, we speak of the `/C` switch in the command line to MACRO as selecting cross–reference output.

A *context switch* is any event in which the program switches from one mode of operation to another. For example, when a program executes a JSYS instruction, the program execution is suspended while TOPS–20 executes to grant the program the service that it requested. Sometimes, TOPS–20 decides that one program has run long enough; TOPS–20 then switches context by suspending one program and continuing another.

Symbol — In the assembler and debugger a symbol consists of a name and a value. The assembler is adept at substituting the value of the symbol for occurrences of its name. The assembler also recognizes *defining occurrences* of the name, at which point it assigns a particular value to the symbol. See also *label*.

Timesharing — One mode in which a computer may be operated, as distinct from *batch* or *real–time*. Timesharing allows many people to use the computer simultaneously. People use terminals to converse with the various programs on the computer system. The timesharing operation is controlled by a program called an operating system. In the DECSYSTEM–20, the operating system is TOPS–20.

TOPS–10 — The operating system, utility programs, and programming environment found in the DECsystem–10 computer systems.

TOPS–20 — The operating system, command scanner, utilities, and environment found in the DECSYSTEM–20. TOPS–20 is the result of extensive development and refinement of the TENEX operating system that was written for the Defense Advanced Research Projects Agency by a group of research programmers working at Bolt, Beranek and Newman.

Trap — A change in the normal flow of execution of a program, often caused by some unpredictable event such as arithmetic overflow. As a result of a trap, the program finds itself running in a trap service routine that must respond to the trap and eventually restore the state of the program so that it may continue.

Two's Complement — A convention for the representation of negative numbers. The two's complement of a number is formed by adding one to the one's complement of that number. Two's complement representation is used in the PDP-10 to represent the negatives of integer and floating–point numbers.

Universal File — A file containing symbol and macro definitions that may be copied into another assembly by means of the `SEARCH` pseudo operator. A universal file is created by the assembler in response to a `UNIVERSAL` pseudo operator appearing in the source instead of a `TITLE` statement.

User Mode — A restricted operating mode in which most programs are run. The PDP–10 limits user mode programs to the memory assigned to them by the operating system. User mode programs are normally refused the ability to perform input/output operations directly. Essentially, the only program that is free from these restrictions is the operating system itself. In some cases, the operating system may grant privileges to some programs or users; these privileges include the ability to perform input and output operations directly.

Programs that are run in user mode under the supervision of the operating system are said to be *user programs*, despite the fact that some have been supplied by the manufacturer.

UUO — See *LUUO* and *MUUO*.

Virtual Memory — A memory system is constructed in part from hardware and in part from operating system software. The virtual memory is the space in which a process executes. It differs from real memory in that a program's virtual memory is implemented by a combination of main memory and disk memory: when a page of virtual memory is referenced, the software brings that page into main memory. When the operating system is pressed for space in main memory, it moves infrequently used pages onto the disk. These manipulations happen without the explicit knowledge of the programmer or user.

Word — A collection of bits of a useful size. The word size of computer system is typically the size of an instruction, and the size of ordinary arithmetic operations. In the DECSYSTEM–20, each word contains thirty–six bits and has its own address. In the usual, unextended, DECSYSTEM–20 programs that we discuss, up to 262,144 words of memory can be addressed.

Write — Transmit data from memory to an external device, or from an accumulator to memory.

Index of Instructions

Index