Introduction to DECsystem-10 Assembly Language Programming

Ralph E. Gorin
Computer Science Department
Stanford University
20 July 1985

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* and were copyrighted © 1976, 1977, 1978, 1979, and 1980 by The New Yorker Magazine, Inc.

This manuscript was prepared using editing and text formatting facilities on DECsystem-10, DECSYSTEM-20, and Xerox Alto computer systems. The final version was prepared using the SCRIBE text formatting program, a product of UNILOGIC, Ltd. It was printed on the Xerox Dover printing system.

Introduction to DECsystem-10 Assembly Language Programming

Ralph E. Gorin
Computer Science Department
Stanford University
20 July 1985

ABOUT THE AUTHOR

Ralph E. Gorin is Director of the Low Overhead Timesharing (LOTS) Computer Facility at Stanford University and Director of the Computer Science Department Computer Facilities. He received his B.S. and M.E. degrees from Rensselaer Polytechnic Institute in 1970. Since 1970, he has been associated with Stanford University. He has taught courses in computer science and electrical engineering. He has been actively involved in assembly language programming of the DECsystem-10 since 1969. He was responsible for operating systems development 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 has written many programs for the DECsystem-10 and DECSYSTEM-20, including SPELL, the spelling correction program, portions of FAIL, the fast one-pass assembler, and the internal data structures and sorting routines for the CREF program. His current activities include the development of a local area computer network to connect the many disparate computer systems at Stanford University, and the development of a networked system of multiple microprocessors for instructional computing.

*

Preface

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-10. The DECsystem-10 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-10 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-10 an outstanding pedagogical vehicle.

The aim of this text is to provide a thorough treatment of assembly language programming for the DECsystem-10, emphasizing the analysis of programs and various methods of program synthesis. This text presents the detailed structure of the DECsystem-10 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-10 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-10 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 Monitor calls [MCRM], and the MACRO assembler [MACRO].[1] Rather, those references should be called upon as a supplement when necessary.

This text is divided into two major areas:

- Sections 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.
- The balance of the text presents 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, and file name parsing are among the topics covered.

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.

Section 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-10 are mentioned.

A view of the computer system as a central processor and memory is introduced in section 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 section 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.

Section 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 section 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 section 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 375 for references.

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

Pushdown stacks are presented in section 8. The relevant instructions and pseudo-ops are shown. A simple application of a stack for reversing an input stream is demonstrated.

The output of the assembler, the binary and listing files, are explained in section 9. A brief discussion of the effects of the linking loader is included here.

Section 10 demonstrates the symbolic debugger, DDT, showing how it can be used to locate problems inside a program.

Section 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 section displays several examples of their use.

The halfword instructions are explained in section 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 section 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 section 14. The same process performed by the example in section 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 section 15.

Section 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 section 17. An example is presented that demonstrates these topics and integer arithmetic operations.

Local UUOs are presented in section 18. An example of LUUOs and of floating-point arithmetic is given. Section 18 generally concludes the presentation of the instruction set; beyond this point, the text deals with applications and additional system features.

An overview of the operating system functions that are so important in real programs appears in section 19. These topics include input and output processing; interprocess communication; and interrupts. The sections that follow expand on these topics and apply the instruction set to more complex data structures and examples.

A very simple first example in section 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.

Section 21 is a discussion of 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 section 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 are discussed in section 23. The example in this section 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.

Section 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 sections following section 24 haven't been written for the DECSystem-10 yet. When written, these will cover the following topics

Random access input and output are discussed in section 25.

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.

R. E. G.
Stanford, California
July, 1985

*

Table of Contents

1. Introduction 1

1.1. Algorithms 2
1.2. Machine Instructions 2
1.3. Operating System - The Software Instruction Set 3
1.4. The Assembler 3
1.5. Debugging Aids 3

2. DECsystem-10 Hardware Overview 5

2.1. The Memory 5
2.1.1. Data in Memory 5
2.1.2. Addresses in Memory 8
2.2. The Central Processing Unit 9
2.2.1. Computer Instructions 10
2.2.1.1. Operation Code 10
2.2.1.2. Operand Addressing 10
2.2.1.3. Instruction Sequences 11
2.2.1.4. Instructions in Memory 12
2.2.2. Historical Notes 14

3. First Example 15

3.1. Review of Example 1 18
3.1.1. Pseudo-Operators 19
3.1.1.1. TITLE 19
3.1.1.2. COMMENT 19
3.1.1.3. ASCIZ 20
3.1.1.4. END 20
3.1.2. MUUOs 20
3.1.2.1. RESET 21
3.1.2.2. EXIT 21
3.1.2.3. OUTSTR 21
3.2. Programs and Memory 22
3.3. Exercise - Self-Identification 22

4. Representation of Data 25

4.1. Representations 25
4.2. Binary Integers 25
4.3. Arithmetic in the Binary System 27
4.4. Representing Negative Numbers 27
4.4.1. Odometer Arithmetic and Ten's Complement Notation 27
4.4.2. Two's Complement Arithmetic 29
4.4.3. Overflow in Two's Complement 30
4.5. Octal Notation 31
4.6. Converting Between Number Systems 31
4.7. Octal Numbers in the PDP-10 33
4.8. The ASCII Code 33
4.8.1. The ASCII and ASCIZ Pseudo-Operators 33
4.9. Exercises 35
4.9.1. Decimal to Binary Conversion 35
4.9.2. Decimal to Two's Complement Conversion 35
4.9.3. Binary to Octal Conversion 36
4.9.4. ASCII Text Assembly 36

5. PDP-10 Instructions 37

5.1. Instruction Format in Memory 37
5.2. How the Assembler Translates Instructions 38
5.3. Effective Address Computation 41
5.3.1. Examples of Effective Address Calculation 42
5.3.1.1. Direct Addressing 42
5.3.1.2. Indexed Addressing 42
5.3.1.3. Indirect Addressing 45
5.3.2. Summary 45
5.4. Instruction Classes 46
5.5. Exercises 46
5.5.1. Instruction Components and Addressing 46

6. Data Movement and Loops 47

6.1. Full-Word Data Movement 47
6.1.1. MOVE Class 47
6.1.2. EXCH Instruction 49
6.2. Jump and Skip Instructions 49
6.2.1. JRST 50
6.2.2. Conditional Jumps and Skips 50
6.2.2.1. JUMP Class 50
6.2.2.2. SKIP Class 52
6.2.2.3. AOS Class 53
6.2.2.4. SOS Class 54
6.2.2.5. AOJ Class 54
6.2.2.6. SOJ Class 54
6.2.2.7. CAM Class 55
6.2.2.8. CAI Class 55

6.2.3. AOBJP and AOBJN 56
6.3. Constructing Program Loops 57
6.3.1. Forward Loops 57
6.3.2. Applying AOBJN 58
6.3.3. Backwards Loops 59
6.3.4. Nested Loops 59
6.4. Exercise 63

7. Terminal Input **65**

7.1. The INCHWL MUUO 65
7.2. The Echo Program 65
7.2.1. Program Outline 65
7.2.2. Supplying Details 66
7.2.3. Literals 66
7.2.4. Character Processing 67
7.2.5. Testing for the End of the Line 68
7.2.6. Testing for an Empty Line 69

8. Stack Instructions **71**

8.1. PUSH Instruction 71
8.2. Defining the Pushdown List 72
8.2.1. BLOCK to Reserve Space 72
8.3. Initializing the Stack Pointer 73
8.3.1. IOWD Pseudo-Operator 73
8.3.2. Defining Symbolic Names 73
8.3.3. Symbolic Names for Accumulators 74
8.4. POP Instruction 74
8.5. ADJSP - Adjust Stack Pointer 75
8.6. Examples of PUSH and POP 75
8.7. Example 4-A 79
8.7.1. Summary of Example 4-A 83

9. The Assembler and Loader **85**

9.1. Overview of Assembly and Loading 85
9.2. Assembler Output 85
9.2.1. Page Headings 88
9.2.2. Listing the Source Lines 89
9.2.3. Listing the Symbol Table 90
9.2.4. Symbol Cross-Reference 91
9.2.5. Operator Cross-Reference 91
9.3. The Loader and Relocatable Code 91

10. Debugging with DDT **93**

10.1. DDT Functions 93
10.2. Loading and Starting DDT 94
10.3. A Sample Session with DDT 95
10.4. Methodical Debugging 99
10.5. DDT Command Descriptions 100

10.5.1. Examines and Deposits 101
10.5.1.1. Current Location 101
10.5.1.2. Current Quantity 101
10.5.1.3. Examine Commands 101
10.5.1.4. Deposit Commands 102
10.5.2. DDT Output Modes 102
10.5.3. DDT Program Control 103
10.5.4. DDT Assembly Operations and Input Modes 104
10.5.5. DDT Symbol Manipulations 105

11. Byte Instructions 107

11.1. LDB - Load Byte 108
11.2. DPB - Deposit Byte 108
11.3. IBP - Increment Byte Pointer 108
11.4. ILDB - Increment Pointer and Load Byte 109
11.5. IDPB - Increment Pointer and Deposit Byte 109
11.6. POINT Pseudo-Operator 110
11.7. Programming Example 110
11.8. ADJBP - Adjust Byte Pointer 111
11.9. Example 4-B 112
11.10. Character Processing; Example 5 115
11.11. Alternative Techniques 119
11.11.1. Flags for Control 119
11.11.2. Control Without Flags 119
11.12. Exercises 120
11.12.1. Test for an Empty Line 120
11.12.2. Interleave Program 120

12. Halfword Instructions 121

12.1. Using Halfword Instructions 123

13. Subroutines and Program Control 125

13.1. Program Counter Format 125
13.2. Subroutine Call Instructions 126
13.2.1. PUSHJ - Push Return PC and Jump 127
13.2.2. POPJ - Pop Return PC and Jump 127
13.2.3. Applications of PUSHJ and POPJ 127
13.2.3.1. Nesting Subroutines 128
13.2.3.2. Restoring Flags 128
13.2.3.3. Skip Returns 128
13.2.3.4. Recursive Subroutines 129
13.2.4. JRST Family 129
13.2.4.1. JRSTF Jump and Restore Flags 130
13.2.4.2. Other JRSTs 130
13.2.5. JSR - Jump to Subroutine 130
13.2.6. JSP - Jump and Save PC 131
13.3. Program Control Instructions 132

13.3.1. JFCL - Jump on Flag and Clear 132
13.3.2. JFFO - Jump if Find First One 133
13.3.3. XCT - Execute Instruction 133
13.4. Example 6-A 134
13.5. Exercises 141
13.5.1. Change INDONE 141

14. Tests and Booleans **143**

14.1. Logical Testing and Modification 143
14.2. Boolean Logic 146
14.3. Example 6-B - Extract Vowels 148
14.3.1. Analysis of Program 6-B 151
14.3.1.1. Two-Pass Structure 151
14.3.1.2. Inner-Loop Instructions 151
14.3.1.3. PROC1 and PROC2 Subroutines 153
14.3.1.4. ISVOW Subroutine 154
14.3.1.5. The BYTE Pseudo-op 154
14.4. Exercises 155
14.4.1. Pig Latin 155

15. Block Transfer and Shift Instructions **157**

15.1. BLT Instruction 157
15.1.1. Warnings about BLT 157
15.1.2. BLT Programming Examples 159
15.2. Shift Instructions 160
15.2.1. LSH - Logical Shift 160
15.2.2. LSHC - Logical Shift Combined 160
15.2.3. ASH - Arithmetic Shift 161
15.2.4. ASHC - Arithmetic Shift Combined 161
15.2.5. ROT - Rotate 161
15.2.6. ROTC - Rotate Combined 161

16. Arithmetic **163**

16.1. Fixed-Point Arithmetic 163
16.1.1. ADD Class 163
16.1.2. SUB Class 163
16.1.3. IMUL Class 164
16.1.4. IDIV Class 164
16.1.5. MUL Class 165
16.1.6. DIV Class 165
16.2. Double-Word Moves 165
16.3. Double-Precision Fixed-Point Arithmetic 166
16.4. Floating-Point Operations 166
16.4.1. Floating-Point Representations 166
16.4.1.1. Single-Precision Floating-Point 166
16.4.1.2. Double-Precision 168
16.4.2. Floating-Point Arithmetic Operations 168

16.4.2.1. Special Cautions 170
16.4.2.2. Floating-Point Exceptions 170
16.4.3. Floating-Point Instruction Set 170
16.4.3.1. FIX -- Convert Floating-Point to Fixed-Point 171
16.4.3.2. FIXR -- Fix and Round 171
16.4.3.3. FLTR -- Float and Round 171
16.4.3.4. FSC - Floating Scale 172
16.5. Exercises 172
16.5.1. Date and Time Conversion 172

17. Macros and Conditionals 173

17.1. Macros 173
17.1.1. Arguments to Macros 174
17.2. Conditional Assembly 175
17.2.1. The IF Construct 175
17.2.2. The IFNDEF Conditional 175
17.2.3. Macros to Control Conditional Assembly 176
17.3. Example 7 - Numeric Evaluator 176
17.3.1. Synthesis of the Main Program 177
17.3.2. Terminal Input and Output 178
17.3.3. Decimal Output and Recursive Subroutines 179
17.3.3.1. Recursion 182
17.3.4. Expression Evaluation 182
17.3.5. Macros for Data Structures 184
17.3.6. Decimal Input Routine 184
17.3.7. Complete Program for Example 7 186
17.4. Exercises 189
17.4.1. Recursive Computation of the Sine Function 189
17.4.2. Russian Multiplication 190
17.4.3. Efficient Exponentiation 191

18. Local UUOs 193

18.1. Example 8 - Floating-Point Input and Output 194
18.1.1. SUBTTL Pseudo-Operator 200
18.1.2. Local UUO Processing 201
18.1.2.1. External Symbols 201
18.1.2.2. Definitions of Local UUOs 201
18.1.2.3. Initialization of the LUUO Handler 203
18.1.2.4. The LUUO Handler 203
18.1.3. FLINP0 - Floating-Point Input Scan 205
18.1.3.1. Processing the Decimal Point 205
18.1.3.2. Processing the Exponent 205
18.1.4. FLOUTP - Floating-Point Output Processing 206
18.1.4.1. FLOUTN 206
18.1.4.2. FLOUTS and FLOUTL 206
18.1.4.3. DECFIL - Decimal Output with Leading Fill 207
18.2. Exercises 207

18.2.1. Simulate the ADJBP Instruction 207
18.2.2. Create the Inverse of ADJBP, SUBBP 208

19. Operating System Facilities 209

19.1. Input/Output 209
19.2. Other Operating System Features 210
19.2.1. Memory Usage Control 210
19.2.2. Information about the Environment 210
19.2.3. Interrupts and Traps 210
19.2.4. Interprocess Communication 211

20. File Output 213

20.1. Example 9 - File Output 213
20.1.1. The OPEN MUUO 214
20.1.2. ENTER MUUO 215
20.1.3. Buffer Rings and the OUTPUT MUUO 216
20.1.4. The CLOSE and RELEAS MUUOs 218
20.1.5. Where are the Buffers? 218
20.2. Example 10 - Long-Precision Fixed-Point Output 219
20.2.1. Mathematical Basis of the DECPBG Routine 222
20.2.2. File Output 224

21. Arrays 225

21.1. One-Dimensional Arrays 225
21.1.1. Example 11 - Factorials to 100! 227
21.1.2. Exercises 235
21.1.2.1. Compute Pascal's Triangle 235
21.1.2.2. Compute *e*, the Base of Natural Logarithms 235
21.2. Two-Dimensional Arrays 237
21.2.1. Array Addressing via Side-Tables 237
21.2.2. Address Polynomials 240
21.2.3. Plot Program, Example 12 241
21.2.3.1. Defining the Array 241
21.2.3.2. Accessing the Array 242
21.2.3.3. Plotting Figures 243
21.2.3.4. Constructing SINTAB and COSTAB 244
21.2.3.5. Writing the Array to a File 246
21.2.3.6. The Completed Plot Program 247
21.2.4. Fortran Library SIN Function 251
21.3. Multi-Dimensional Arrays 254
21.4. Efficiency Considerations 255
21.5. Array Exercises 256
21.5.1. Magic Square 257
21.5.2. Tic Tac Toe 258
21.5.3. Triangular Matrices 259

22. File Input 261

22.1. OPEN for Input 261
22.2. LOOKUP MUUO 261
22.3. Simple Disk Input, Example 13a 262
22.4. Search Program, Example 13b 265
22.4.1. Structured Programming 266
22.4.2. GTINPF - Get Input File 274
22.4.3. File Name Scan 275
22.4.4. GTSTRG - Get Search String 275
22.4.5. HEADER 275
22.4.6. FIND 276
22.4.7. GETLIN and EOFLIN 277
22.4.8. LOOK and GTINCH 277
22.4.8.1. LOOK 278
22.4.8.2. An Optimization of LOOK 280
22.4.9. FIN - Finish Routine 280
22.5. Exercises 281
22.5.1. Maze 281
22.5.2. Saddle Points in an Array 282
22.5.3. Crossword Puzzle 282

23. File Directory and Sort 285

23.1. Directory Processing 285
23.2. Dynamic Space Allocation 286
23.3. Bubble Sort 287
23.4. Directory I/O and Sort Program 287
23.5. Discussion of this Program 293
23.5.1. RDUFD 294
23.5.2. CMPRES 296
23.5.3. SORT 297
23.5.4. PRINT 298
23.6. Heapsort 300
23.6.1. Machine Representation of a Heap 301
23.6.2. Building a Heap 301
23.6.3. Removing Sorted Data from the Heap 303
23.6.4. Intermediate Storage for Heapsort 304
23.6.5. High-Level Representation of Heapsort 304
23.6.6. Heapsort Subroutine 305
23.6.7. Discussion of the Heapsort Subroutines 307
23.6.8. Timing Analysis of Heapsort 309
23.7. Exercises 309
23.7.1. Cryptogram Program 309
23.7.2. Directory Cleaner 312
23.7.3. Fixed-Field Sorting Program 312

24. Lists and Records 313

24.1. Dictionary Program - Example 16 314
24.2. Dictionary Records 316
24.2.1. Suppressed Labels 316
24.2.2. PHASE and DEPHASE Pseudo-Operators 316
24.2.3. .ORG Pseudo-Operator 317
24.3. Searching by Hash Code 317
24.3.1. PROCWD 319
24.3.1.1. HSHFUN 321
24.3.1.2. NAMCMP 321
24.3.1.3. BLDBLK 321
24.3.1.4. Efficiency Improvements 324
24.3.2. GETWRD 324
24.4. Dictionary and Sort Program 325
24.4.1. NSSORT 336
24.4.2. PRDICT 341
24.5. Exercises 342
24.5.1. Token Scanning 342
24.5.2. Cross-Reference Program 344
24.5.3. KWIC Index Program 345
24.5.4. Set Operations 346

25. Random Access I/O 347

25.1. Example Program 347
25.2. Update-in-Place 350
25.3. Random Access in Buffered Modes 351

Appendices

A. PC & Flags 353

A.1. Flags 353
A.2. The Program Counter 356

B. Instruction Nomenclature 357

C. DDT Commands 359

C.1. Examines and Deposits 359
C.1.1. Current Location 360
C.1.2. Current Quantity 360
C.1.3. Examine Commands 360
C.1.4. Deposit Commands 360
C.2. DDT Output Modes 361
C.3. DDT Program Control 362
C.4. DDT Assembly Operations and Input Modes 363
C.5. DDT Symbol Manipulations 364
C.6. DDT Searches 365
C.7. Patch Insertion Facility 366
C.8. Location Sequence 366

C.9. Miscellaneous Features 367

D. Obsolete Instructions 369

D.1. JSA - Jump and Save AC 369
D.2. JRA - Jump and Restore AC 369
D.3. Long Floating-Point 370
D.4. DFN -- Double Floating Negate 371
D.5. UFA -- Unnormalized Floating Add 371

E. Common Pitfalls 373

F. References 375

Glossary 377

Index of Instructions 387

Index 389

*

List of Figures

Figure 2-1: KL10-based DECsystem-10 Configuration 6
Figure 2-2: Processor and Memory Configuration 7
Figure 2-3: DECsystem-10 Virtual Memory 9
Figure 3-1: Machine Representation of the Program 22
Figure 3-2: Sample Homework 1 23
Figure 5-1: PDP-10 Instruction Formats 37
Figure 5-2: Instruction Loop & Effective Address Calculation 41
Figure 5-3: Comparison of Array Access and Record Access 44
Figure 9-1: Overview of the Assembler and Loader 86
Figure 9-2: Assembler Listing of the Source Program 87
Figure 9-3: Assembler Listing of the Symbol Table 88
Figure 9-4: Assembler Listing of the Source Program 88
Figure 9-5: Selected Lines from the CREF Listing 90
Figure 12-1: Binary Tree with Halfword Links 124
Figure 15-1: BLT Example 158
Figure 15-2: LSH Data Movement 162
Figure 15-3: LSHC Data Movement 162
Figure 15-4: ASH Data Movement 162
Figure 15-5: ASHC Data Movement 162
Figure 15-6: ROT Data Movement 162
Figure 15-7: ROTC Data Movement 162
Figure 20-1: A Two-Buffer Buffer Ring 217
Figure 21-1: Sample Output from the LISAJ Subroutine 251

*

List of Tables

Table 4-1: Decimal, Octal and Binary Equivalents 32
Table 4-2: The ASCII Character Set 34
Table 6-1: Notation for Instruction Descriptions 48
Table 6-2: The MOVE Instructions 48
Table 6-3: Modifiers for Jumps, Skips and Compares 50
Table 14-1: Boolean Functions 146
Table 16-1: Floating-Point Instruction Set 170

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 are 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 mechanized. The *assembler program* (together with a loader) handles the problems of allocating space in memory for the program and variables, and generally performs useful bookkeeping chores. Note, however, that we continue to deal with the primitive functions, the 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 perhaps 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 *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-10 is called TOPS-10.

When a computer such as the DECsystem-10 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-10 operating system places various restrictions on the programs that it runs. These restrictions are implemented by running all programs (except for TOPS-10 itself) in *user mode.* In user mode, programs are restricted to memory assigned to them by TOPS-10; 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-10 provides various subroutines (accessed via the MUUO operations) 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 209.

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. In the DECsystem-10 the assembler is called MACRO.

1.5. DEBUGGING AIDS

The DECsystem-10 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 10, page 93.

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.

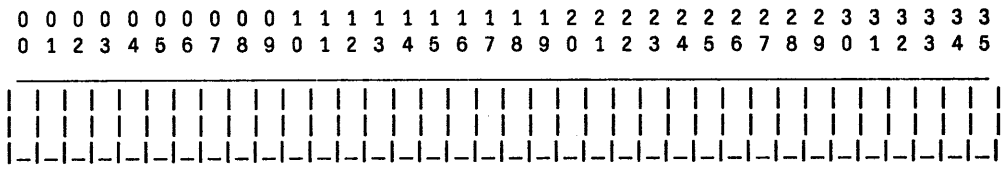
Chapter 2
DECsystem-10 Hardware Overview

A computer system is a whole composed of many parts. The DECsystem-10 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.

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. The bits in every 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 25.

The figure from page 1-6 of the DECsystem-10/20
Hardware Reference Manual, would be appropriate.

Figure 2-1: KL10-based DECsystem-10 Configuration

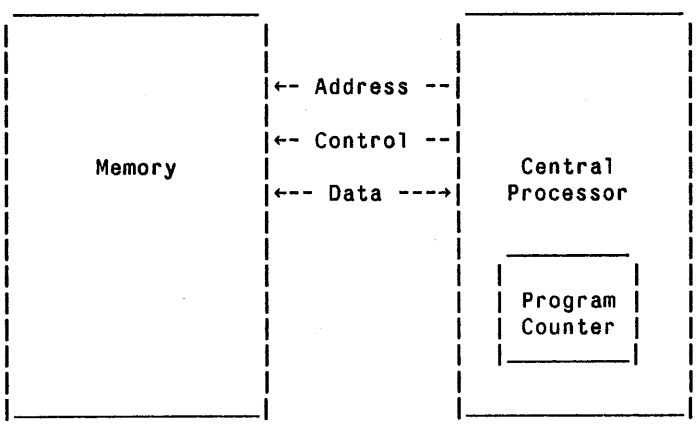


Figure 2-2: Processor and Memory Configuration

The following pattern represents a quite common number, 0:

```
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
 _______________________________________________________________________
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
|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|0|0|0|0|0|0|0|0|
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
```

Another familiar number is the integer 1:

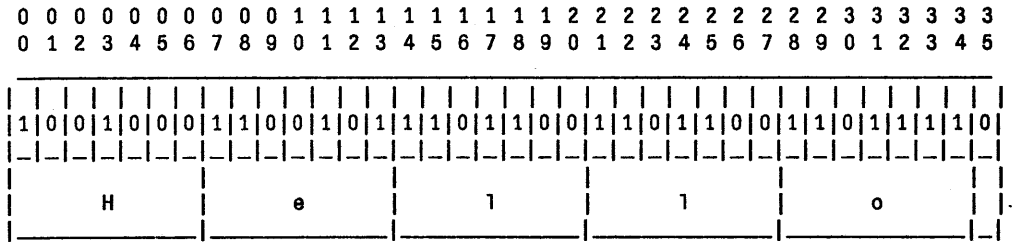
```
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
 _______________________________________________________________________
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
```

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:

```
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
 _______________________________________________________________________
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
|0|1|0|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|
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|
```

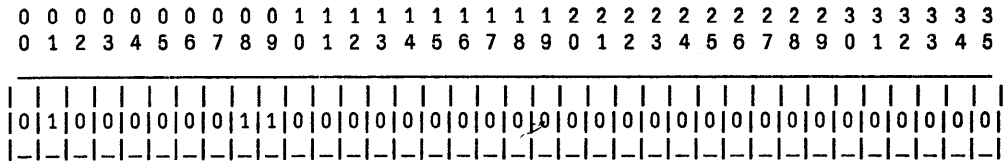
Even if you understand binary notation, this pattern isn't expected to make much sense until we discuss the details of floating-point representations in section 16.4, page 166.

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 33, 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:



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).

Again, the idea is this: the memory does not distinguish among the varieties of data stored within it. It is a program that implicitly supplies an 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 DECsystem-10 sees a memory space that contains addresses in the range from `0` to `262,143`.[1] We will often use *octal* (i.e., radix 8) notation when we talk about addresses and the contents of memory words (see section 4.2, page 25). In octal, addresses range from `0` to `777777`, as shown in figure 2-3.

In the DECsystem-10 the memory space seen by a program is called a *virtual memory,* as distinct from

[1]The TOPS-10 operating system further restricts the program to use only those memory addresses that the program has asked for.

the actual *physical memory*, because the operating system creates the appearance of this memory space from fragments of real memory.

Sixteen memory locations, addresses 0 through 17 (octal), are distinguished from all other locations. 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.

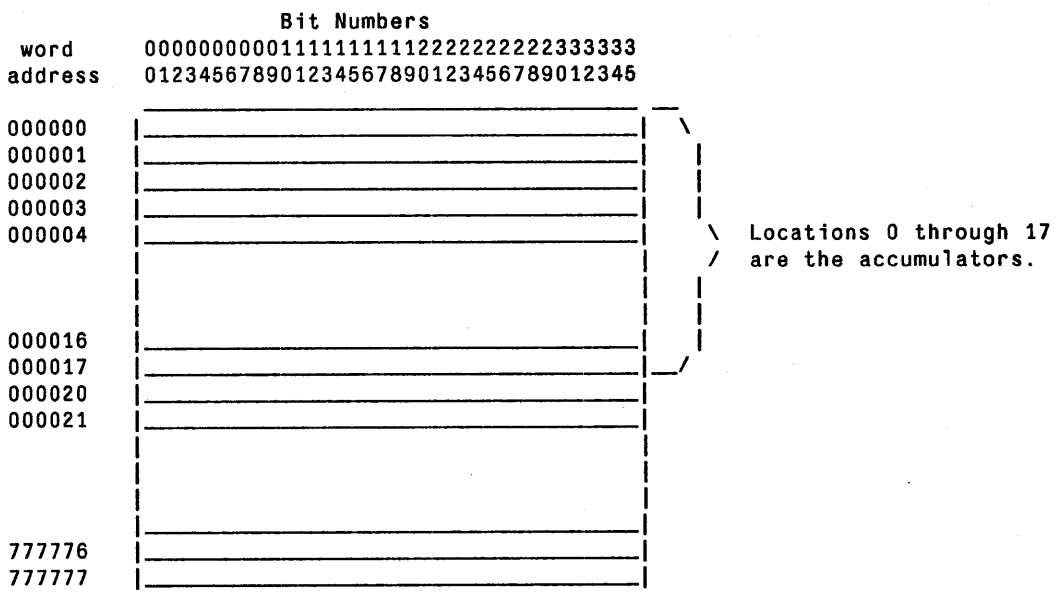


Figure 2-3: DECsystem-10 Virtual Memory

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. 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.

---

[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.

- 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 called *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 `1234` is executed, the CPU normally executes the instruction at `1235`, 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.

2.2.1. 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 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.

Although the "half" address is restricted, it is very important. It names one of the first 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 addresses 1 to 17 can be used as *index registers* to modify any instruction's effective address calculation (see section 5.3, page 41).
- 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.

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.

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.[4] 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 the programmer. The two examples approximately correspond to the Pascal 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.

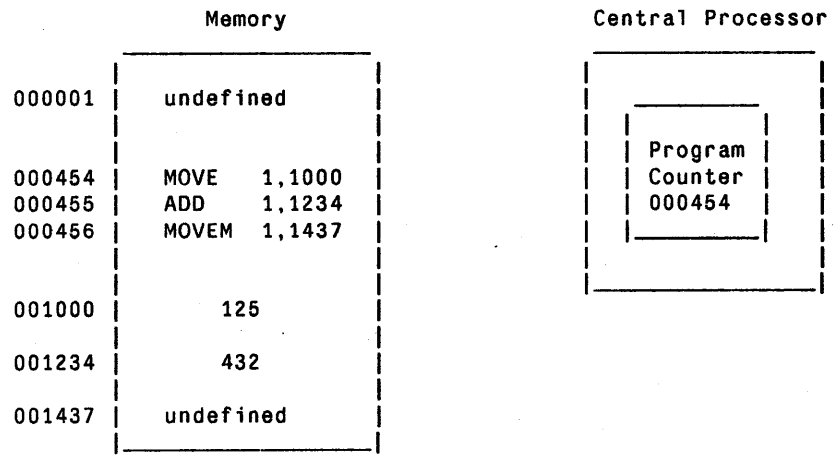
When it is necessary to copy data from one accumulator to another, usually one instruction is sufficient. The accumulator that is being copied from can appear as the memory operand in a MOVE instruction. For example, to copy the data from register 1 to register 16 we could write:

```
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:

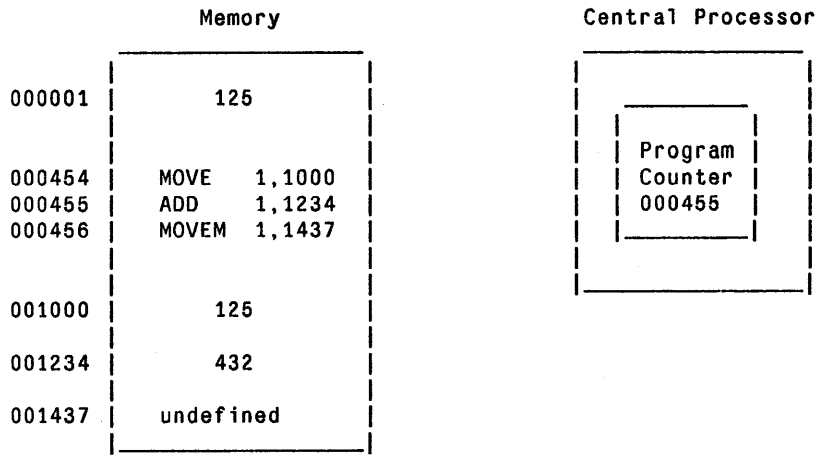


We have arbitrarily selected locations 454, 455, and 456 as the three consecutive locations to hold this program fragment. Generally speaking, this program fragment could be anywhere in memory. The only restriction is that, as we shall see, some memory locations are changed by the program; it would be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

---

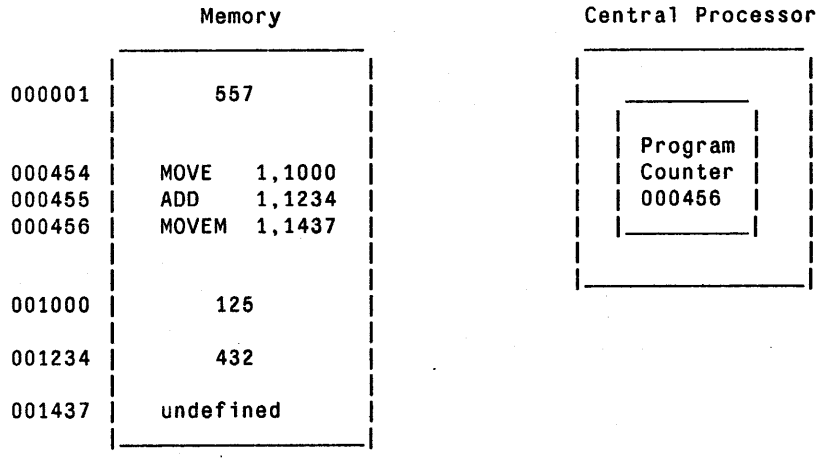
[4] This ADD instruction works for numbers stored in the PDP-10's fixed-point format.

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

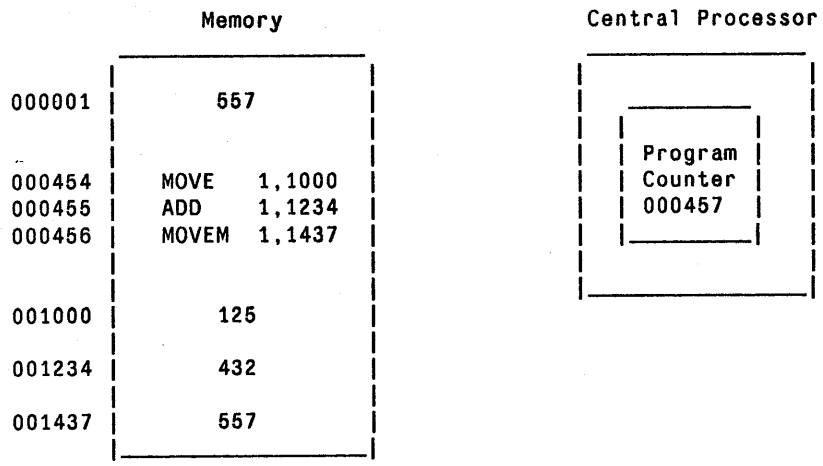


Note that the contents of location 1 have been changed. Also the program counter now points to 455.

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 further instructions.[5]

2.2.2. Historical Notes

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 1080, 1090, 1091, 2040, 2050 and 2060 configurations. We shall generally discuss the KL10 as it appears in the 1090 system.[6] The evolution of the DECsystem-10 and DECSYSTEM-20 is discussed in [BELL].

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. The DECSYSTEM-20 represents some advances beyond the techniques practiced in the DECsystem-10; consequently most DECSYSTEM-20 programs will not run on the DECsystem-10.

[5] We shall discuss how to stop the computer at the end of a program in section 3, page 15.

[6] The 2060 provides an extension of earlier designs to support a larger address space. It is difficult to deal with these extensions in the body of an introductory work.

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 only of 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-10 the instruction that types a string on the terminal is called `OUTSTR`, meaning *OUTput STRing.* `OUTSTR` is not really an instruction in the sense that `MOVE` and `ADD` are instructions. Instead, `OUTSTR` is actually a subroutine call to TOPS-10 that requests an output operation. These subroutine calls to TOPS-10 are called MUUO instructions, meaning *Monitor Unimplemented User Operation.*

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

```
.
.
.
OUTSTR
.
.
.
```

OUTSTR 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 OUTSTR subroutine also requires an argument that describes the string to output.

Before we go on to further describe the argument that OUTSTR 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, MUUO 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:

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

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

Well, now we have the string. But how do we connect this string to the OUTSTR MUUO? First, we need to introduce the concept of a *label.* A label is simply a name we give to an address in memory. Since in general we don't know exactly the address in memory where each object has been placed, a label is a symbolic name by which we can make reference to the various things in memory.

We will tell MACRO to create a label that describes where in memory this string can be found. We do this by changing the line on which the string is defined to include the name of the label, MESAGE, and a colon

---

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

(:) at the left margin. The colon tells the MACRO assembler to define a label called MESAGE. MESAGE identifies the computer word that contains the start of the string.[2] A label can be up to six letters long; we selected the name MESAGE as an an obvious corruption of "message".

```
        .
        .
        .
        OUTSTR
        .
        .
        .
MESAGE: ASCIZ   /Hi
/
        .
        .
        .
```

Next we will change the OUTSTR MUUO to make reference to the label MESAGE. You may say that the resulting instruction *addresses* or *refers to* the label MESAGE.

```
        .
        .
        .
        OUTSTR  MESAGE
        .
        .
        .
MESAGE: ASCIZ   /Hi
/
        .
        .
        .
```

Instructions are executed by the computer one after another. After the OUTSTR the computer will want something to do next. Since the OUTSTR concludes the work we intended, we tell the computer to stop running our program by including the EXIT operation. EXIT is another subroutine call to the system; it tells the system to stop executing this program. If we don't include an EXIT, the computer will fetch the next word following the OUTSTR 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 OUTSTR we add an EXIT.

```
        .
        .
        .
        OUTSTR  MESAGE
        EXIT
        .
        .
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

---

[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.

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:  .
        OUTSTR  MESAGE
        EXIT
        .
        .
MESAGE: ASCIZ   /Hi
/
        .
        .
        END     START
```

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

We have nearly built 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.

$

START:  RESET                   ;RESET the state of I/O devices
        OUTSTR  MESAGE          ;Output the message at MESAGE
        EXIT                    ;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 will 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 four *pseudo-operators*: TITLE, COMMENT, ASCIZ and END.
- The program contains two user-defined *labels*: START and MESAGE.
- The program contains three *MUUO operations*, which are requests made to the operating system: RESET, OUTSTR, and EXIT.
- The remainder of the program is mostly comments, except for *arguments* to the pseudo-operators TITLE, 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 four pseudo-ops in example 1 are TITLE, COMMENT, 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 and 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 10, page 93.

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.[3]

3.1.1.3. 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-10. 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 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 33.

3.1.1.4. END

The END pseudo-op signifies the end of 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 EXIT MUUO 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. MUUOs

As we have mentioned, our programs can request specific functions from the TOPS-10 operating system by means of a special group of instructions called MUUOs. This program contains three MUUO operations, RESET, OUTSTR and EXIT. Each of these can be thought of as a subroutine call to the TOPS-10 operating system. The particular MUUO name specifies which one of the many available functions to perform.

---

[3] See the discussion of macros and conditional assembly, section 17, page 173.

3.1.2.1. RESET

The RESET MUUO 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. EXIT

The EXIT MUUO tells the operating system to stop executing the program. This is the normal way to signify that a program has reached its end. When TOPS-10 starts running a program, the terminal is made available to that program for its input and output activity. When the program executes the EXIT function the operating system stops running it. Then terminal input is directed to the TOPS-10 command processor, which prompts for further commands.

3.1.2.3. OUTSTR

The OUTSTR MUUO is used to send a string of characters to the terminal. When OUTSTR is executed, TOPS-10 assumes that the word referred to by the address field of the OUTSTR is the beginning of a string of ASCII characters in the format that the ASCIZ pseudo-operator builds. TOPS-10 proceeds then to copy the string from the program and send it to the terminal.

In this program, the address field of the OUTSTR 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 OUTSTR MESAGE, it looks up MESAGE in the dictionary and substitutes the value of symbol. In this example, the symbol MESAGE has the value 143.[4] When the assembler sees the text OUTSTR MESAGE, it substitutes the value 143 for MESAGE resulting in something equivalent to OUTSTR 143. Of course, MUUO names such as OUTSTR also have numeric values. The entire line OUTSTR MESAGE is translated to the octal number 051140000143.

Even though a reference to the label MESAGE appears before the definition of the label, MACRO can make the proper substitution (of 143 for MESAGE) because MACRO really 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 OUTSTR to the number 051140000000). 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 151, the assembler will change every reference

---

[4] Actually, the value 143 results from the action of the loader in addition to the action of the assembler.

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.

```
Legal Symbols                          Illegal Symbols

MESAGE                                 1MORE   (starts with a digit)
.EXAM1                                 MARK!TWAIN   (has an illegal character)
mesages   (this is equivalent to MESAGE)
```

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. Locations 40 and 41 are used by the hardware as the LUUO trap locations (discussed in section 18, page 193).

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. 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.

The assembler together with the loader (remember the loader is biased towards loading programs starting at 140) 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.

```
Address         Contents              Meaning
140             047000000000          RESET
141             051140000143          OUTSTR  MESAGE  (MESAGE is 143)
142             047000000012          EXIT
143             443221505000          "H", "i", carriage return
                                              line-feed, and null.
```

Figure 3-1: Machine Representation of the 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-10 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-10 commands to produce a cross-reference listing:

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

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 9.2, page 85, it will still be useful for you to study it.

```
        TITLE   HW1 Self-Identification

START:  RESET
        OUTSTR  MESAGE
        EXIT

MESAGE: ASCIZ   /
My name is Ralph Gorin
I work at Stanford University
If I were a student in this course,
I would mention something memorable about myself.
I am studying assembly language because it is fascinating.
/
        END     START
```

Figure 3-2: Sample Homework 1

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 leave off the carriage return and line feed that terminate the line where END appears, MACRO will also complain that the END is missing.
- For more advice about what to avoid see appendix E, page 373.

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.

The computer represents numbers using the binary or 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:

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

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

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

These 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
```

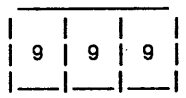
If our machine were limited to five bits, we could not compute the sum `10000+10000` because the correct result, `100000`, does not fit into five bits.

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.

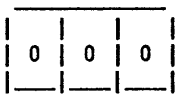
4.4.1. Odometer Arithmetic and Ten's Complement Notation

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 `999`



and we add one to it (by driving one more mile) the result is

[1] Federal law forbids actual experimentation.



More compactly we could write `999 + 1 = 000`. 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 arithmetic" the pattern of characters "999" 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":

```
Number      ----|----|----|----|----|----|----|----|----|----
Line           -4   -3   -2   -1    0    1    2    3    4

Odometer    ----|----|----|----|----|----|----|----|----|----
Line          996  997  998  999  000  001  002  003  004
```

This may seem like a strange way to write negative numbers, but it makes some sense: suppose we add six to the representation of negative four. Negative four is represented by `996`; adding six (`006`) results in `002`, representing two. For another example, let us add negative three to two. Adding `997` and `002` results in `999` which represents negative one. These examples suggest that there is some interpretation of this representation 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 different numbers in each group.

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

```
Number     ... -|----|----|- ... -|----|----|----|----|- ... -|----|----|- ...
Line          -501 -500 -499     -2   -1    0    1    2      498  499  500

Odometer            |----|- ... -|----|----|----|----|- ... -|----|
Line               500  501     998  999  000  001  002     498  499
```

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.

It might be noted that the figures 500 represent negative five hundred, a number for which there is no positive counterpart. This must be true because there are an equal number of non-negative and negative numbers. Since the non-negative numbers include zero, there must be one fewer positive numbers than there are negative numbers. This leads to the inescapable fact that 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:

```
original          one's          two's          one's          two's
 number        complement     complement     complement     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).

---

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

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:

Column Weights	-32	16	8	4	2	1	
	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 (including 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 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,

---

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

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 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 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.

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

```
  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
 -12    number         -12  quotient        -0  quotient
  53                    2 <-second           1 <-third
 -50                      remainder            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 ASCII code (American Standard Code for Information Interchange) is used in the DECsystem-10 for communicating 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. About ninety-five of these characters actually print things. The remainder are *control characters*, some of which have special functions when sent to terminals or other devices. The line feed character, for example, has the function of advancing the paper (or terminal screen) to make a fresh line available for printing.

In table 4-2 we present the 7-bit ASCII code used in the DECsystem-10. 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.

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.

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

	0	1	2	3	4	5	6	7	
000	NUL							BEL	NUL Null Character BEL Bell
010	BS	HT	LF	VT	FF	CR			BS Backspace HT Horizontal Tab LF Line Feed VT Vertical Tab FF Form Feed CR Carriage Return
020									
030				ESC					ESC Escape
040	SP	!	"	#	$	%	&	'	SP Space
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	DEL Delete or Rubout

Table 4-2: The ASCII Character Set

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 Characters	36-Bit Octal
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 `OUTSTR` 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 `OUTSTR` might require the length of the string as an explicit argument. We find that it is easy to use `OUTSTR` since it figures out the length by itself. But there is one disadvantage: a routine such as `OUTSTR` 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 107 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.

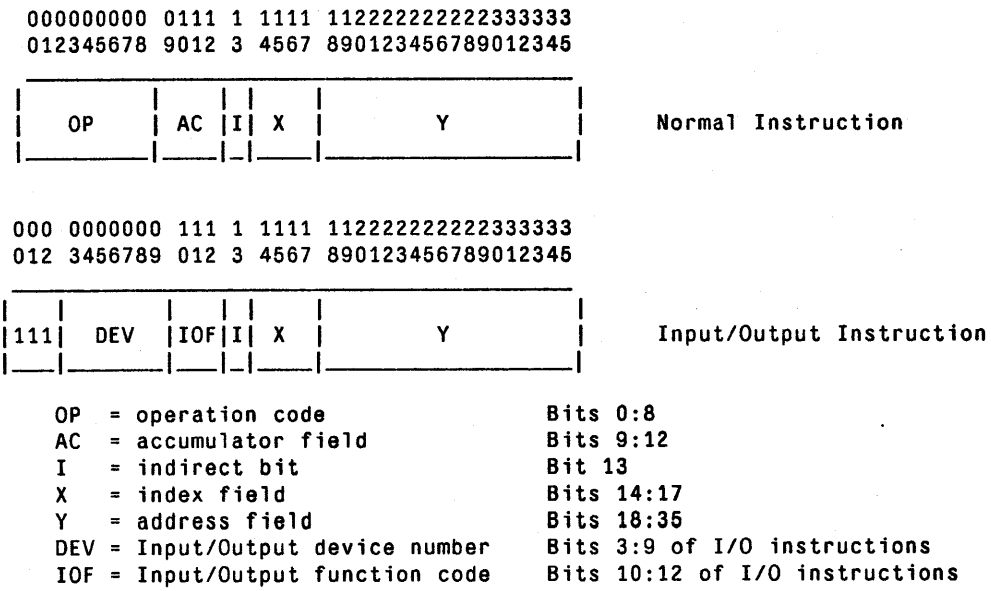


Figure 5-1: PDP-10 Instruction Formats

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.

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 41.

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 41) 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:

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

- 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

```
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
```

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

```
000000000 0111 1 1111 112222222222333333
012345678 9012 3 4567 890123456789012345
_____________________________________________
|         |    | |    |                     |       Octal          Source Text
|010101101|0000|0|0000|00000000000000000|     255000 000000    JFCL
|_________|____|_|____|_____________________|
```

Consider another example, one that we have seen before:

```
MOVE    1,1000
```

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:

```
000000000 0111 1 1111 112222222222333333
012345678 9012 3 4567 890123456789012345
_____________________________________________
|         |    | |    |                     |       Octal          Source Text
|010000000|0001|0|0000|00000001000000000|     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:

```
000000000 0111 1 1111 112222222222333333
012345678 9012 3 4567 890123456789012345
```

					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:

```
000000000 0111 1 1111 112222222222333333
012345678 9012 3 4567 890123456789012345
```

					Octal	Source Text
011111000	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

```
000000000 0111 1 1111 112222222222333333
012345678 9012 3 4567 890123456789012345
```

					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 might well expect that these two instructions don't do the same thing. A comma makes a big difference!

```
000000000 0111 1 1111 112222222222333333
012345678 9012 3 4567 890123456789012345
```

					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 an 18-bit quantity;[2] 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.

The program fragment in figure 5-2 depicts the entire instruction execution cycle, including the 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. This program introduces some further CPU internals. Among these are

- The Memory Address register (MA) is an internal register by which the CPU specifies the address of the word in memory that it wants to read or write. The result of the effective address calculation will appear in MA.
- 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 `EXIT` MUUO.
- 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`).

```
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:     *)
  MA := PC;                           (* Get the instruction word addressed by the PC *)
  IR := MEM[MA][0:12]                 (* Instruction Register is set from bits 0:12    *)
                                      (*    of the instruction word.                   *)
  REPEAT                              (* Compute the effective address:                *)
    Y := MEM[MA][18:35];              (* Initialize I, X, and Y fields from the        *)
    X := MEM[MA][14:17];              (*    memory word addressed by MA.  Initially,   *)
    I := MEM[MA][13:13];              (*    this word is the instruction word.         *)
    IF X = 0 THEN MA := Y             (* In Direct Addressing, the effective address,  *)
                                      (* in MA, is the Y field of the instruction.     *)
             ELSE MA := 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.   *)
  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                       *)
  END;                                (*     and loop to the next instruction.         *)
```

Figure 5-2: Instruction Loop & Effective Address Calculation

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

---

[2] In the 2060, a wider effective address is possible.

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.

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. Examples of Effective Address Calculation

The effective address calculation process is very important to our further progress with assembly language. As mentioned above, every instruction calculates an effective address; all instructions calculate their effective addresses in precisely the same 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 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 examples that you may find helpful.

5.3.1.1. 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 the instruction register and into the I, X, and Y variables. The instruction register holds bits 0:12 of the instruction; these bits are the operation code and accumulator fields.

The X field of this instruction word contains zero, so the MA 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, the UNTIL clause is satisfied, and no repetition of the addressing cycle takes place. The effective address computation is complete; the result, 1000, is in the MA register.

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

5.3.1.2. 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 that involves indexing.

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 word 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, MA 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 MA, 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, MA 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 the MA. After this truncation, the result in MA 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 225). 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 5-3.

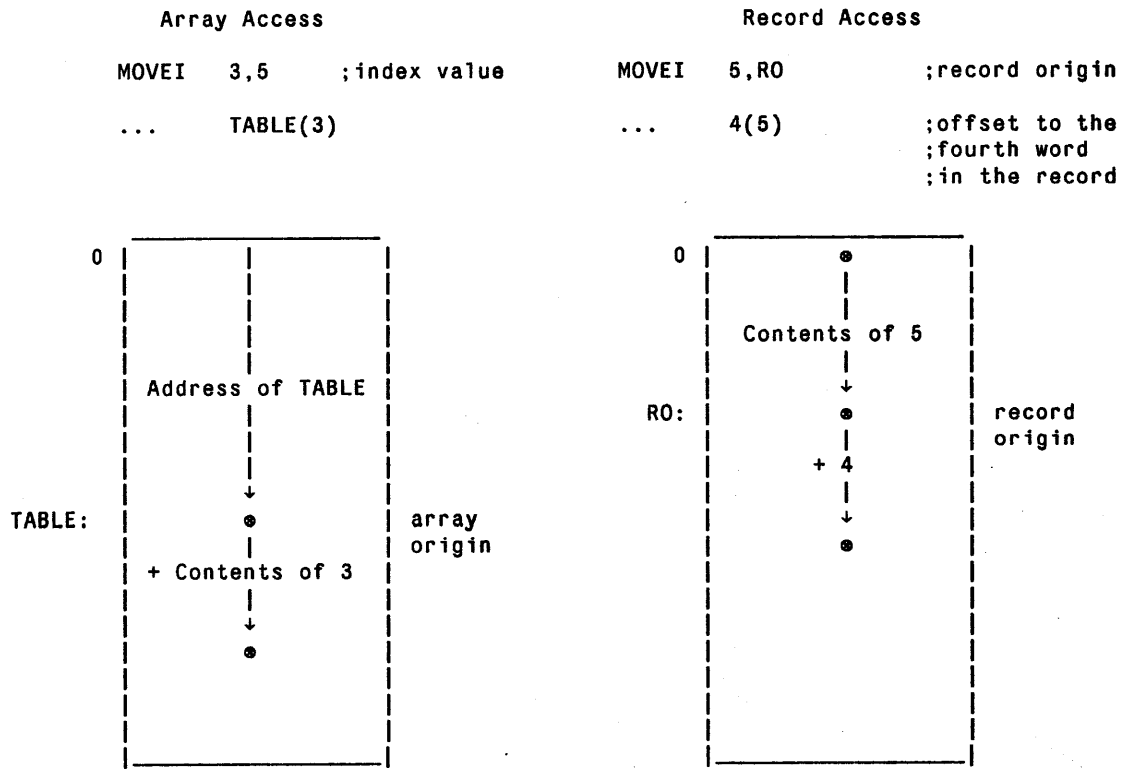


Figure 5-3: Comparison of Array Access and Record Access

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 more difficult to debug. For this reason, and to increase system efficiency, programs should avoid changing themselves.

---

[3] Records will be discussed in section 24, page 313.

5.3.1.3. 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.

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, MA is set to 17240, as X is zero. However, since I is one, 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 read from the word addressed by MA. Note that MA now addresses the word at 17240. We have specified that location 17240 contains 167, this means that the new Y field is 167 and I and X are both zero. The MA 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 MA.

For our final example of this section, 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 MA is set from the Y field of the instruction, 1767. Since indirect addressing is specified, the UNTIL clause is unsatisfied and the REPEAT loop is executed again. The word addressed by the MA, location 1767, is fetched to supply new values for the I, X, and Y fields. The word at 1767 supplies a new Y field of 2; the new X field is 17, and I is zero.

Now, the MA 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 MA. 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. Summary

We hope that you will come to regard the effective address calculation as a simple process, but we must say that 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.

5.4. INSTRUCTION CLASSES

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 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 are similar to the ones in the MOVE class.

5.5. EXERCISES

5.5.1. Instruction Components and Addressing

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
HRROI   1,5004
XCT     2025(6)
JRST    @672
SETOM   -5(7)
FMPRI   3,204500
MOVE    2,@561(1)
```

Now, compute the effective address for each of the lines of code that is listed above. The values of the accumulators and memory locations are as indicated below. 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:         10427
accumulator 7:          3612
location  575:       1000673
location  672:  301000001772
```

Chapter 6
Data Movement and Loops

Eighty-four different instructions are presented in this section. 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 57.

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 accumulators and memory, occasionally with some minor 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 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 in this table:

```
MOV
     |E no modification |  from memory to AC
     |N negate source   |I Immediate. Source is 0,,E to AC
     |M magnitude       |M from AC to memory
     |S swap source     |S to self.  If AC>0 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 as presented in table 6-2.

E	The effective address of the I, X, and Y parts of the instruction.
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 doubleword accumulator in which C(AC) is most significant.
PC	The 18-bit contents of the program counter.

Table 6-1: Notation for Instruction Descriptions

```
MOVE            C(AC) := C(E)
MOVEI           C(AC) := 0,,E
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) := -E
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,,E
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) := E,,0
MOVSM           C(E)  := CS(AC)
MOVSS           Temp  := CS(E); C(E) := Temp; if AC>0 then C(AC) := Temp
```

Table 6-2: The MOVE Instructions

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).

```
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 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).

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.

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 should be avoided 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.[1] 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;
```

6.2. 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.

---

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

6.2.1. JRST

The most frequently executed instruction in the PDP-10 is JRST. The JRST instruction really has several different functions; the particular function is selected by the accumulator field.

When the accumulator field is zero, a JRST instruction is simply 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 12345.

```
JRST  0,    PC := E;    Unconditional jump.  The AC field must be zero.
```

We shall discuss the other functions of JRST in section 13.2.4, page 129.

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 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 are displayed in table 6-3.

blank	Never
L	Less than
LE	Less than or Equal
E	Equal
N	Not Equal
GE	Greater than or Equal
G	Greater than
A	Always

Table 6-3: Modifiers for Jumps, Skips and Compares

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)  ≤ 0 then PC := E;
JUMPE     If C(AC)  = 0 then PC := E;
JUMPN     If C(AC)  ≠ 0 then PC := E;
JUMPGE    If C(AC)  ≥ 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, in that it possesses an instruction named JUMP that means *never jump.*

The instruction JUMPA is an unconditional jump. However, the JRST instruction is preferred because JRST is faster than JUMPA on all CPU models.[2]

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.
```

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 this:

```
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

---

[2] 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 the KL10 made a special effort to make JRST faster.

```
        . . .                   ;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.

```
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) ≠ 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 fails to jump, so too the SKIP instruction fails to skip.

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 146) 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
        . . .
        JRST    WLOOP           ;back to the top of the loop.

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

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 143.

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 also 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 ≤ 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 ≠ 0 then skip.
AOSGE   Temp := C(E)+1; C(E) := Temp; If AC > 0 then C(AC) := Temp;
                If Temp ≥ 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 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)  ≤ 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)  ≠ 0 then PC := E;
AOJGE   C(AC) := C(AC)+1;  If C(AC)  ≥ 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)  ≤ 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)  ≠ 0 then PC := E;
SOJGE   C(AC) := C(AC)-1;  If C(AC)  ≥ 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

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 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 (references memory)
CAML            If C(AC)  < C(E) then skip;
CAMLE           If C(AC)  ≤ C(E) then skip;
CAME            If C(AC)  = C(E) then skip;
CAMN            If C(AC)  ≠ C(E) then skip;
CAMGE           If C(AC)  ≥ 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 left half and the effective address in the right half. If the indicated condition is true, the instruction will skip. Note that the immediate operand is always considered to be a positive number.

```
CAI             no op
CAIL            If C(AC)  < E then skip;
CAILE           If C(AC)  ≤ E then skip;
CAIE            If C(AC)  = E then skip;
CAIN            If C(AC)  ≠ E then skip;
CAIGE           If C(AC)  ≥ E then skip;
CAIG            If C(AC)  > E 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 is a lower-case letter:

```
CAIGE   1,141   ;skip if it is larger than or equal to lower-case A
JRST    NOTLOW  ;not a lower-case letter
CAILE   1,172   ;skip if smaller than or equal 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 107), 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.
```

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 ≤ I) AND (I ≤ 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 less than or equal to decimal 100
JRST    ARYERR  ;jump to Array Subscript Error routine
```

By the way, this is equivalent to the Pascal statement:[3]

```
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) ≥ 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

[3] But, in Pascal there are no symbolic labels.

desired 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            ;less than or equal to 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.[4]

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

---

[4] 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.

```
        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 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:

```
        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.

Since effective address calculation only considers the right half of the index register, we have accomplished 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 126.)

```
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.
```

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.

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. Nested Loops

Nested loops are quite simple to do. For example, let us write a program to produce the triangular pattern depicted below:

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

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).

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

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:  OUTSTR  ASTER           ;Print one asterisk
        SOJG    11,STARS        ;decrement star count, loop if more
        OUTSTR  NEWLIN          ;print carriage return and line feed. end line.
        SOJG    10,LINE         ;decrement the line number held in 10.
                                ; If the result is positive, do another line
        EXIT                    ;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.

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

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
        OUTSTR  BLANK                   ;print a blank
        JRST    ELIN                    ;test for end of line

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

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

An alternative approach to controlling the execution of this program is presented below. Also, a new MUUO, OUTCHR, *OUTput CHaRacter* is presented. In this approach, we assume that we will have to print a blank character. At the label CHAR the program loads a blank character into register 1. Then, the program determines whether this assumption was right: the CAML instruction will skip if we are close to the left side of the line. However, if the program has already placed enough blanks on the line, the CAML does not skip and an asterisk character is loaded into register 1, obliterating the blank that was there. Then, the OUTCHR MUUO is used to print the one character that it finds in register 1.

```
CHAR:   MOVEI   1," "           ;Assume we must send a blank
        CAML    13,12           ;Skip if near front of line
        MOVEI   1,"*"           ;Past the blanks, print a star
        OUTCHR  1               ;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
```

We have already mentioned that the notation `"*"` causes MACRO to assemble the number corresponding to the asterisk character. The `OUTCHR` MUUO sends the one character it finds at the right-end (bits `29:35`) of the word at its effective address. Note too that although address `1` is an accumulator, it appears in the address field of the `OUTCHR` instruction.

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 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 character is either a carriage return or a line feed. When an empty line is typed in, the program will stop itself.

7.1. THE INCHWL MUUO

We will use the INCHWL MUUO to wait for a complete line of terminal input and then read the terminal characters one by one. When the program executes INCHWL, TOPS-10 makes the program wait until the user types a complete line on the terminal. Then, TOPS-10 reactivates the program and returns to it the first character from the line. INCHWL returns the character right-adjusted in the word at its effective address. On a subsequent call to INCHWL, the program will not be made to wait; the next available character on the line will be returned immediately.

7.2. THE ECHO PROGRAM

We begin the process of writing the Echo program, as described at the beginning of this chapter, by writing an outline.

7.2.1. Program Outline

We write an outline to reveal the structure of the proposed program. Not every detail needs to be filled in. From the program outline we can tell if the structure is adequate to serve the need. We will check the outline to determine if it addresses all the concerns stated in the problem description.

For the Echo program we propose the following outline:

```
START:  initialize the program
GETLIN: initialize for another input line
CHLOOP: obtain a character and process it
        if not end of line, jump to CHLOOP
        if not empty line, jump to GETLIN
        EXIT
```

The outline leaves some questions unanswered. For example, the outline does not specify the method by which the program identifies an empty line. These details are important, and they will be supplied in due course.

7.2.2. Supplying Details

Once a satisfactory outline is made, we can turn our attention to any portion of it and work on the details. We will begin at START and supply some of the "boilerplate" needed with any program:

```
        TITLE   Echo a Line of Input.   Example 3

START:  RESET                   ;initialize the program
        OUTSTR  GREET           ;send a friendly message
GETLIN: initialize for another input line
CHLOOP: obtain a character and process it
        if not end of line, jump to CHLOOP
        if not empty line, jump to GETLIN
        EXIT

GREET:  ASCIZ/Welcome to the Echo Program
/
        END     START
```

7.2.3. Literals

Before we contine supplying details of how this program is constructed, we digress to the entralling subject of literals in MACRO-10. Before we define what we mean by a *literal*, we offer an example.

Every time that we have used OUTSTR we have had to invent a label for the message that we wanted to send. We have just written an OUTSTR in which we've repeated this practice:

```
        . . .
        OUTSTR  GREET
        . . .
GREET:  ASCIZ/Welcome to the Echo Program
/
```

A literal saves us from having to write the label and make a reference to it. We will rewrite this sequence making use of a literal, thus:

```
        . . .
        OUTSTR  [ASCIZ/Welcome to the Echo Program
/]
        . . .
```

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 to

- Make a group of words that contain the binary representation of the material found within the square brackets.
- Put those words somewhere, in consecutive locations. The first location used is the address of the literal.
- Store the address of the literal in the place where the literal text appears. That is, MACRO supplies the address of where it put the material that was present between the square brackets.

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.

7.2.4. Character Processing

It is now time to focus our attention on the processing done by this program. As we add more detail, we may discover a need to revise the outline that we made. In this program, we have added detail to the outline, and discovered a few changes in detail. The outline has been revised in the vicinity of CHLOOP as follows:

```
CHLOOP: obtain a character
        if this character is a line feed, jump to EOLN
        print the character
        jump to CHLOOP

EOLN:   perform any end-of-line cleanup
        if not empty line, jump to GETLIN
        EXIT
```

The revision of the outline occurs not because the original outline was particularly faulty, but because as we come to examine the outline at this level of detail, it is necessary to unfold the meaning of vague expressions such as "read and process a character." Although lack of detail somewhat obscures this revision, we are much closer to the level of detail necessary for writing the program.

As we examine the outline, some of the program becomes apparent. We will use the INCHWL MUUO to obtain a character from the input line. INCHWL will place the character in the word that the INCHWL addresses. This can be any memory word. However, looking ahead to the point in the program where we must determine if the character is a line feed, we can expect to need one of the compare instructions. Because the compare instructions require that one of the operands be present in an accumulator, we will write the INCHWL to bring the character into an accumulator. Our decision about which accumulator to use appears to be quite unconstrained; we'll use register number 1.

We have made considerable progress thus far. Since the character will appear in register 1, we know what instruction will be used to print the character. We will now fill in the outline where we know how:

```
        TITLE   Echo a Line of Input.   Example 3

START:  RESET                   ;initialize the program
        OUTSTR  [ASCIZ/Welcome to the Echo Program
/]                              ;send a friendly message
GETLIN: initialize for another input line
CHLOOP: INCHWL  1               ;obtain a character
        if this character is a line feed, jump to EOLN
        OUTCHR  1               ;print the character
        JRST    CHLOOP          ;loop to process another character

EOLN:   perform any end-of-line cleanup
        if not empty line, jump to GETLIN
        EXIT

        END     START
```

This is beginning to look quite a lot like a program. Notice that we have carefully recycled our outline by turning it into the comments that go with the code.

7.2.5. Testing for the End of the Line

In the DECsystem-10 the end of a line of terminal input is signalled by the presence of a line feed character. Referring to the table of ASCII characters (section 4.8) we see that line feed is octal 12. The character that appears in register 1 is right-justified; that is, it is held in bits 29:35. The CAIE or CAIN instructions are appropriate for making character comparisons. We can replace the portion of the outline that says "if this character is a line feed, jump to EOLN" with the following fragment:

```
        CAIN    1,12            ;skip unless this is a line feed
        JRST    EOLN            ;this is a line feed.  Jump to EOLN
```

As a user of the DECsystem-10 you are aware that usually you end a line by typing the carriage return key. When TOPS-10 sees the return key, it adds a line feed after the return. Thus, usually, a program will see both a carriage return and a line feed when it reads the terminal. However, please note that TOPS-10 considers the line feed to be end of the line, so a program that is searching for the end of a line should not be satisfied until it finds the 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.

We are left with the question of what to do when we see a carriage return. The traditional answer, and one which has been proven efficatious in many applications, is to discard the carriage return.

We will adjust the program to discard the carriage return, by augmenting our fragment thus:

```
        CAIN    1,15            ;skip unless this is a carriage return
        JRST    CHLOOP          ;discard CR: go get the next character
        CAIN    1,12            ;skip unless this is a line feed
        JRST    EOLN            ;this is a line feed.  Jump to EOLN
```

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, some prompting is within our capabilities. We will add input prompting and other refinements to our program. The program outline, which is rapidly being transformed into the program itself, now appears as this:

```
        TITLE   Echo a Line of Input.   Example 3

START:  RESET                   ;initialize the program
        OUTSTR  [ASCIZ/Welcome to the Echo Program
/]                              ;send a friendly message
GETLIN: OUTSTR  [ASCIZ/Please type a line:    /]
        initialize for another input line
CHLOOP: INCHWL  1               ;obtain a character
        CAIN    1,15            ;skip unless this is a carriage return
        JRST    CHLOOP          ;discard CR: go get the next character
        CAIN    1,12            ;skip unless this is a line feed
        JRST    EOLN            ;this is a line feed.  Jump to EOLN
        OUTCHR  1               ;print the character
        JRST    CHLOOP          ;loop to process another character

EOLN:   OUTSTR  [ASCIZ/
/]                              ;add carriage return and line feed
        if not empty line, jump to GETLIN
        EXIT

        END     START
```

7.2.6. Testing for an Empty Line

The one area of the program that still needs work is the test for an empty line. An empty line is a line that contains either line feed alone, or carriage return and line feed. We will effect this test in the following way: notice that if any character other than return or line feed appears, then that character will be echoed by the OUTCHR MUUO. If we simply count the number of times that OUTCHR is executed, if that count is non-zero then the line was not empty.

We will count the number of times that OUTCHR is executed in register 2. Before we start processing the line, we should set register 2 to zero. We will replace the JRST that follows the OUTCHR with an AOJA instruction that unconditionally jumps and which also increments register 2. Thus, register 2 will contain the count of how many times the OUTCHR has been executed. In the case of an empty line, the OUTCHR is never executed, and register 2 will be zero.

Also, we embellish the program somewhat by adding a caption to the echoed line. Note how a SKIPG instruction is used to avoid printing the caption more than once.

```
        TITLE   Echo a Line of Input.   Example 3

START:  RESET                   ;initialize the program
        OUTSTR  [ASCIZ/Welcome to the Echo Program
/]                              ;send a friendly message
;initialize for another input line
GETLIN: OUTSTR  [ASCIZ/Please type a line:    /]
        MOVEI   2,0             ;Initialize character Counter
CHLOOP: INCHWL  1               ;obtain a character
        CAIN    1,15            ;skip unless this is a carriage return
        JRST    CHLOOP          ;discard CR: go get the next character
        CAIN    1,12            ;skip unless this is a line feed
        JRST    EOLN            ;this is a line feed.  Jump to EOLN
        SKIPG   2               ;Skip unless this is the first time here
        OUTSTR  [ASCIZ/The line you typed is: /]   ;first time: write message
        OUTCHR  1               ;print the character
        AOJA    2,CHLOOP        ;Increase the character count
                                ; and loop to process another character

EOLN:   OUTSTR  [ASCIZ/
/]                              ;add carriage return and line feed
        JUMPG   2,GETLIN        ;If line wasn't empty, get another line
        OUTSTR  [ASCIZ/All done.
/]
        EXIT

        END     START
```

Chapter 8
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. The right half of the stack pointer addresses the current stack top; the left half of the stack pointer is usually a control count that describes how many unused locations are available in the stack.

8.1. 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 accumulator in the PUSH instruction initially 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.

If, as a result of the addition, the left half of the stack pointer becomes positive, a pushdown overflow condition results (but the instruction proceeds to completion). Usually, the left half is initialized to make this warning mechanism effective.

```
PUSH    C(AC) := C(AC) + <1,,1>;    C(CR(AC)) := C(E)
```

[1] In the KI10 and later processors, any carry from bit 18 to bit 17 is suppressed.

8.2. DEFINING THE PUSHDOWN LIST

In the PDP-10, a pushdown list is simply an array of consecutive locations in memory. When we use the stack instructions we must reserve consecutive locations in memory 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 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 sections of a program are possible, multiple stacks may become necessary. For instance, the TOPS-10 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.

8.2.1. BLOCK to Reserve Space

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. For example, the following fragment reserves 200 octal (i.e., 128 decimal) locations. We label the first location with the symbolic name PDLIST:

```
PDLIST: BLOCK   200             ;Reserve space for the stack
```

The symbol PDLIST labels the first location reserved by the BLOCK pseudo-op. Symbolically, the locations in this region may be referred to as PDLIST+0, PDLIST+1, PDLIST+2, etc., through PDLIST+177:

```
PDLIST+0
PDLIST+1
PDLIST+2

 . . .

PDLIST+176
PDLIST+177
```

It is important to note that when we write something like PDLIST+25 we mean the *value of the symbol* PDLIST plus octal 25; in the case of a label such as PDLIST, the value of the symbol is the location or address of PDLIST at runtime. In most high-level languages such an expression would mean the *contents of the location called* PDLIST plus decimal 25. Apart from the difference between octal and decimal numbers, the interpretation of symbolic names is quite 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 possibly 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 PDLIST plus octal 25, you must write something like

```
MOVE    5,PDLIST
ADDI    5,25
```

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.

8.3. 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 right half of the stack pointer should be initialized to the address PDLIST-1.

The left half of the 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. In short, we can initialize the accumulator that will be used to hold the stack pointer by means of one instruction:

```
MOVE    17,[-200,,PDLIST-1]
```

8.3.1. 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 200,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]

8.3.2. Defining Symbolic Names

One of the things about programming that is certain is that programs change. Note that we have used the length of the stack in two places. We arbitrarily selected 200 as the length of the pushdown list in the BLOCK pseudo-op. Having made this choice, we find we must use 200 when we describe the initial stack pointer. Fundamentally it is a bad practice to sprinkle constants throughout a program. The decision to use the number 200 as the stack size was purely arbitrary, and it was made without any information about what the stack would be used for. It would not be surprising then to discover that 200 was an inappropriate size.

If we ever wanted to change the stack size in this program we would have to go back, reread the program carefully and find all the places where the number 200 appears as the pushdown list length and change them. It might be a very difficult task.

To avoid the use of constants throughout the program, MACRO allows us to define new symbolic names and assign to these symbols the values that we choose. We will define a symbol whose value is the desired length of the stack. Then, instead of writing 200 everywhere, we write the name of 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., PDLEN=200. This form of definition is called an *assignment* (in contrast to labels that are defined with a colon). We will ask MACRO to make such a definition. However, by writing two equal signs we instruct MACRO to *suppress* this symbol so the debugger will not type it out.[3] To our program fragment we add the definition of the symbolic name PDLEN, signifying the length of the pushdown stack:

---

[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. Moreover, some of the TOPS-10 input and output operations are characterized by descriptors of this format.

[3]See the discussion of DDT, section 10, page 93.

```
PDLEN==200                      ;define the size of the pushdown list

 . . .

PDLIST: BLOCK   PDLEN           ;allocate space for the pushdown list

 . . .
        MOVE    17,[IOWD PDLEN,PDLIST]
```

There is one important restriction on the use of symbolic names such as PDLEN. The assembler insists that the argument to the BLOCK pseudo-op be completely defined by the time BLOCK is seen. This dictates that we must place the definition of PDLEN before the BLOCK pseudo-op.

8.3.3. 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.[4] Just as we wrote PDLEN==40 to define a symbolic name for the length of the stack, 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, for instance

```
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 use only one equal sign. This makes the symbolic name available to DDT for typeout during symbolic disassembly.

8.4. POP INSTRUCTION

The POP instruction undoes the effect of PUSH as follows: 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 becomes 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.

```
POP     C(E) := C(CR(AC));   C(AC) := C(AC) - <1,,1>
```

---

[4]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.

[5]In the KI10 and later processors, any carry from bit 18 to bit 17 is suppressed.

8.5. ADJSP - ADJUST STACK POINTER

While we are on the subject of stacks, it seems appropriate to mention the ADJSP instruction. This instruction exists in the KL10 and later processors. The ADJSP instruction adds E (the effective address) to each half of the specified accumulator. There is no carry between halves of the accumulator.

If E is positive, this instruction effectively allocates space on the stack. If E is negative, this instruction deallocates space on the stack. If a negative adjustment changes CL(AC) from positive to negative, or if a positive adjustment changes CL(AC) from negative to positive, then a stack overflow condition is reported.

```
ADJSP   C(AC) := C(AC) + <E,,E>;  Stack Overflow is possible
```

8.6. EXAMPLES OF PUSH AND POP

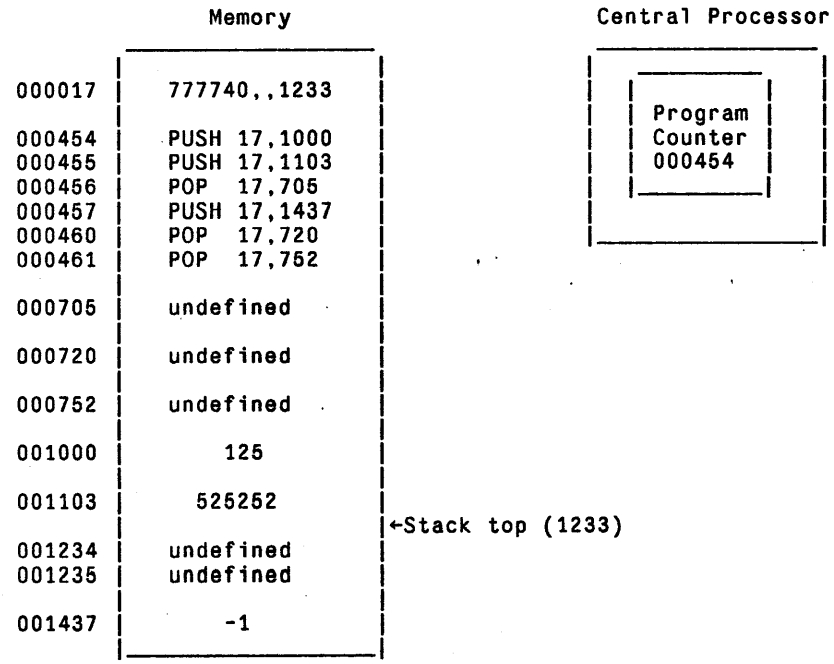
Stacks are useful in recursive subroutines and for temporary storage. Since we are not quite ready to talk about recursive subroutines, for the present we shall discuss only the use of stacks for temporary storage.

Let us examine some pushes and pops. Suppose we have executed the instruction

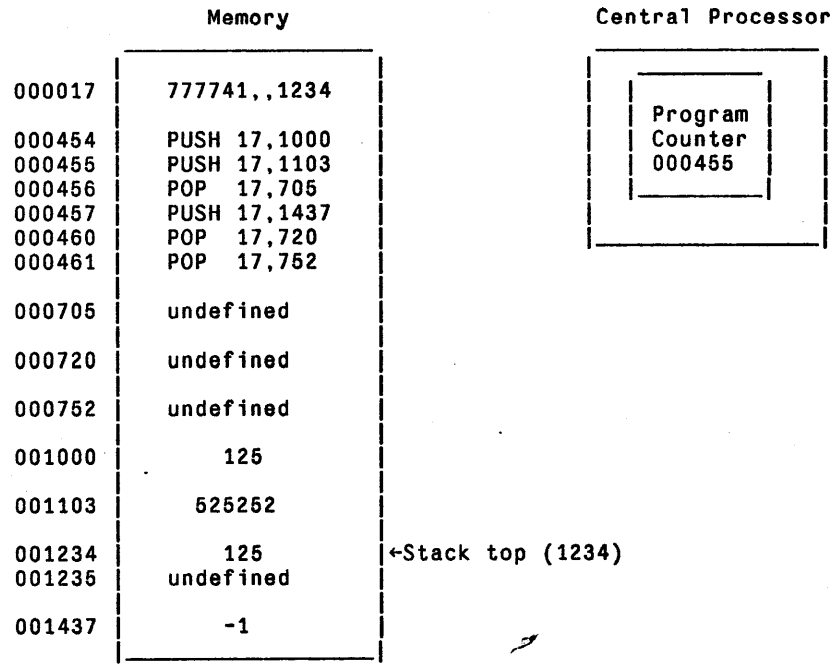
```
MOVE    17,[IOWD 40,1234]
```

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.

We set up memory to contain some data and a short program:

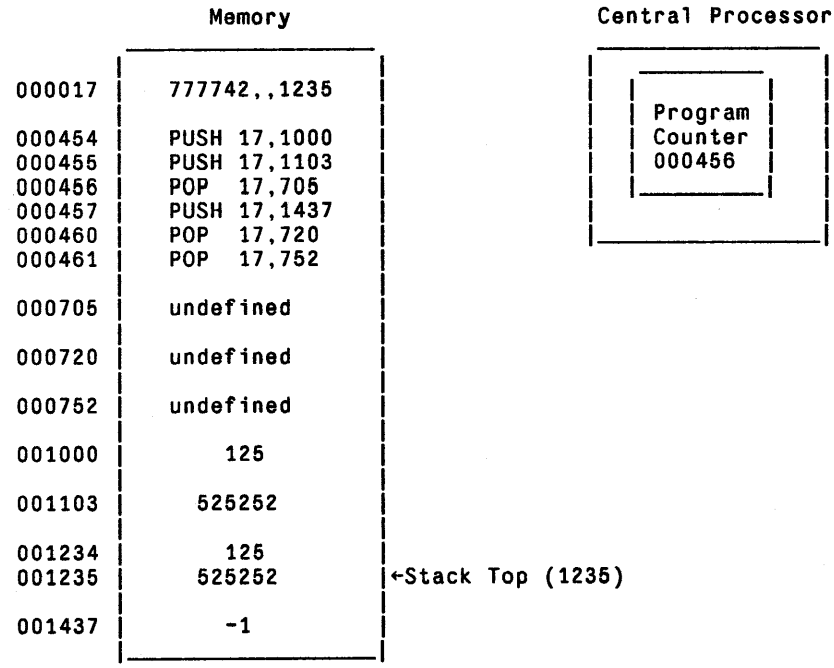


When this program is executed, the PUSH instruction in location 454 will advance the stack pointer in register 17; register 17 now addresses location 1234, the first location in the stack. The PUSH instruction goes on to read the word at location 1000 and copy it to the word addressed by the stack pointer. As a result of executing the PUSH in location 454, the CPU and memory now have the following appearance:

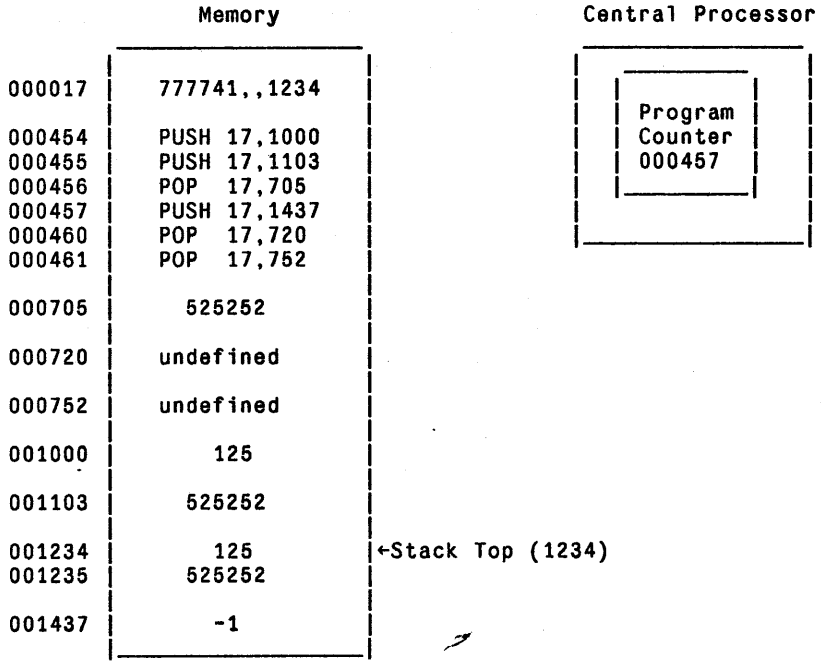


Note that the stack pointer in register 17 has been changed. Also, location 1234, the current stack top, has been changed to be a copy of the data that is in location 1000. The stack pointer generally addresses the current stack top.

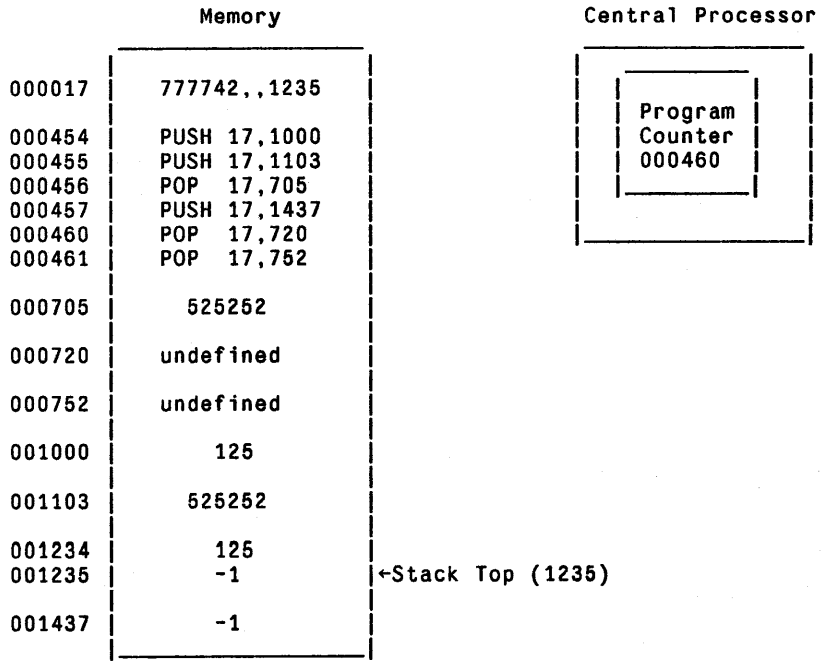
Next, the PUSH in location 455 is executed. The stack pointer is advanced and the data at location 1103 is copied to the stack. The program counter is incremented to address the instruction at location 456:



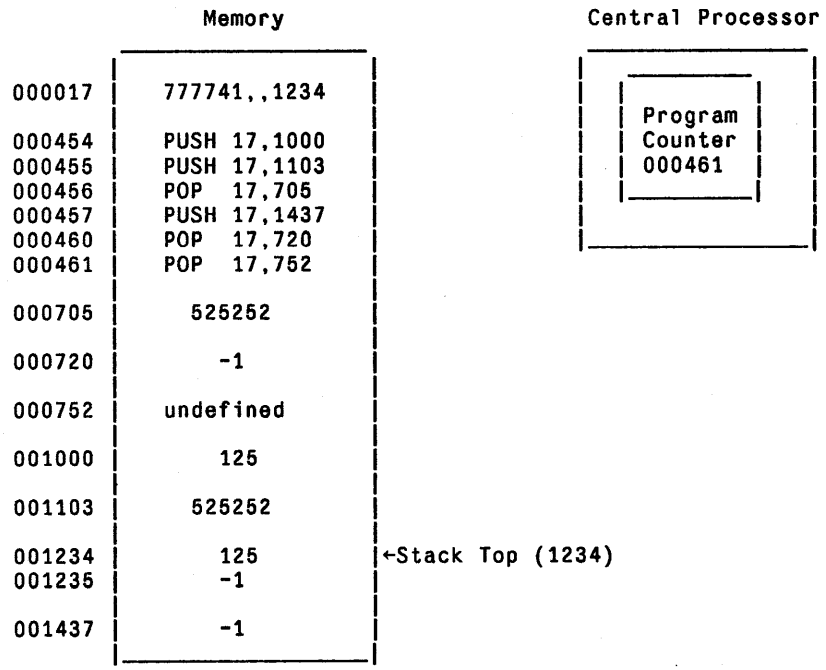
The POP instruction at location 456 undoes the previous PUSH by copying data from the stack to location 705. The stack pointer is backed up to reflect the removal of an item from the stack. Note that the popped item is still present in the memory allocated to the stack; it is considered to be removed from the stack because the stack pointer no longer includes that item.



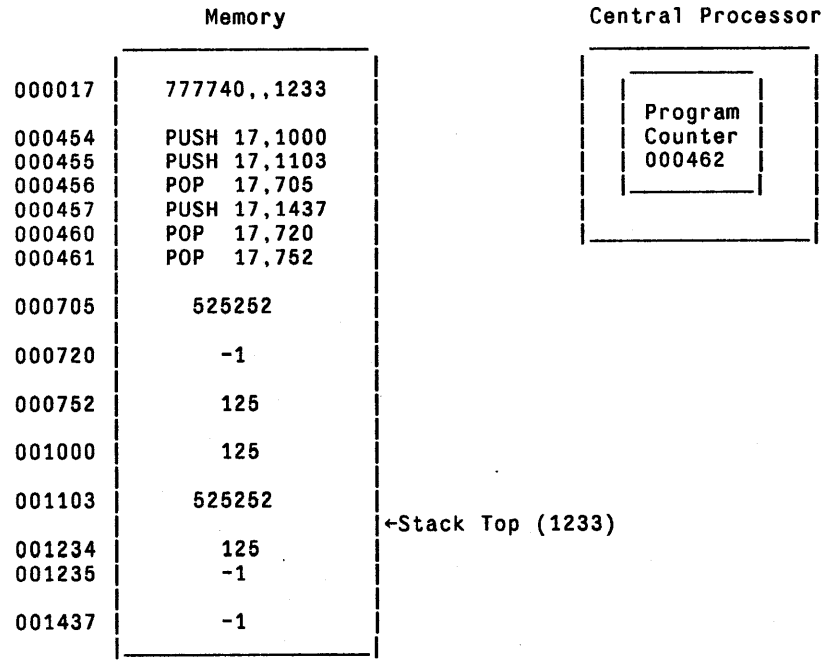
The next PUSH obliterates the stale copy of the item that we just removed from the stack:



Now, we execute the POP at location 460. 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.

8.7. EXAMPLE 4-A

In this example we read a line and reverse it. A pushdown list is used to reverse the order of characters in the line.[6]

Let us start with an outline, similar to the one we made for the previous example:

```
START:  initialize the program
GETLIN: initialize for another input line
CHLOOP: obtain a character
        if end of line, jump to EOLN
        add character to the stack
        jump to CHLOOP

EOLN:   if the line is empty, EXIT
POPIT:  remove a character from the stack
        print the charater
        if the stack isn't empty, jump to POPIT
        finish output line
        jump to GETLIN
```

We have elected to learn from our experience with example 3: This outline reflects some of what we have learned from doing that example. We start with an outline that is somewhat more detailed than we used at the start of example 3.

We will fill in this example by copying as much as we think appropriate from example 3. Much of the outline is replaced with actual code:

---

[6]Perhaps this is a silly and contrived example. However, we'll have good reason to employ a stack for this purpose when we come to the DECOUT subroutine in section 17.3.

```
        TITLE   Reverse a Line of Input.   Example 4A
A=1                             ;symbolic names for accumulators

START:  RESET                   ;initialize the program
        OUTSTR  [ASCIZ/Welcome to the Reverse Program
/]                              ;send a friendly message
        initialize stack
;initialize for another input line
GETLIN: OUTSTR  [ASCIZ/Please type a line:    /]
CHLOOP: INCHWL  A               ;obtain a character
        CAIN    A,15            ;skip unless this is a carriage return
        JRST    CHLOOP          ;discard CR: go get the next character
        CAIN    A,12            ;skip unless this is a line feed
        JRST    EOLN            ;this is a line feed.  Jump to EOLN
        add character to the stack
        JRST    CHLOOP          ;loop to obtain another character

EOLN:   if the line is empty, jump to STOP
        OUTSTR  [ASCIZ/The reversed line is:  /]
POPIT:  remove a character from the stack
        OUTCHR  A               ;print the charater
        if the stack isn't empty, jump to POPIT
        finish output line
        JRST    GETLIN          ;go get another input line

STOP:   OUTSTR  [ASCIZ/All done.
/]
        EXIT

        END     START
```

One change from example 3 is that we have adopted a symbolic name for register 1. 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 listing. Hereafter, all our accumulators will have symbolic names.

We have already explained that we are 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 CHLOOP. All of this is accomplished by the following augmentation of the outline:

```
        TITLE   Reverse a Line of Input.   Example 4A
A=1                             ;symbolic names for accumulators
P=17                            ;register for the stack pointer

PDLEN==200                      ;room for 128 characters

PDLIST: BLOCK   PDLEN           ;reserve space for the stack

START:  RESET                   ;initialize the program
        OUTSTR  [ASCIZ/Welcome to the Reverse Program
/]                              ;send a friendly message
        MOVE    P,[IOWD PDLEN,PDLIST]  ;initialize stack pointer
;initialize for another input line
GETLIN: OUTSTR  [ASCIZ/Please type a line:    /]
CHLOOP: INCHWL  A               ;obtain a character
        CAIN    A,15            ;skip unless this is a carriage return
        JRST    CHLOOP          ;discard CR: go get the next character
        CAIN    A,12            ;skip unless this is a line feed
        JRST    EOLN            ;this is a line feed.  Jump to EOLN
        add character to the stack
        JRST    CHLOOP          ;loop to obtain another character

EOLN:   if the line is empty, jump to STOP
        OUTSTR  [ASCIZ/The reversed line is:  /]
POPIT:  remove a character from the stack
        OUTCHR  A               ;print the charater
        if the stack isn't empty, jump to POPIT
        finish output line
        JRST    GETLIN          ;go get another input line

STOP:   OUTSTR  [ASCIZ/All done.
/]
        EXIT

        END     START
```

We can now specify the nature of the input processing done in the loop at CHLOOP. 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:

```
CHLOOP: INCHWL  A               ;obtain a character
        CAIN    A,15            ;skip unless this is a carriage return
        JRST    CHLOOP          ;discard CR: go get the next character
        CAIN    A,12            ;skip unless this is a line feed
        JRST    EOLN            ;this is a line feed.  Jump to EOLN
        PUSH    P,A             ;save character on the stack
        JRST    CHLOOP          ;loop to obtain another character
```

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 OUTCHR MUUO will send a character to the terminal. We start our fragment of output loop with an instruction to pop a character into register A, followed by an OUTCHR MUUO:

```
POPIT:  . . .
        POP     P,A                     ;get one character from stack
        OUTCHR  A                       ;send it to the terminal
        JRST    POPIT                   ;loop.
```

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

POPIT:  CAMN    P,[IOWD PDLEN,PDLIST]
        JRST    OUTFIN
        POP     P,A
        OUTCHR  A
        JRST    POPIT

OUTFIN: ;here when done
```

```
        Bottom Test

POPIT:  POP     P,A
        OUTCHR  A
        CAME    P,[IOWD PDLEN,PDLIST]
        JRST    POPIT
        ;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 POPIT 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.[7] 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, since we said that the program should stop when an empty line in input, we can add some code in front of the output loop to ensure proper operation of the program.

```
        CAMN    P,[IOWD PDLEN,PDLIST]   ;is the stack empty?
        JRST    STOP                    ;empty line, stop running.
POPIT:  POP     P,A                     ;get one character from stack
        OUTCHR  A                       ;send it to the terminal
        CAME    P,[IOWD PDLEN,PDLIST]   ;is the stack empty now?
        JRST    POPIT                   ;not yet.  Loop again.
        . . .                           ;here when done
```

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:

```
        CAMN    P,[IOWD PDLEN,PDLIST]   ;is the stack empty?
        JRST    STOP                    ;empty line, stop running.
POPIT:  POP     P,A                     ;get one character from stack
        OUTCHR  A                       ;send it to the terminal
        CAME    P,[IOWD PDLEN,PDLIST]   ;is the stack empty now?
        JRST    POPIT                   ;not yet.  Loop again.
        OUTSTR  [ASCIZ/
/]                                      ;add CR LF to end of the output line
```

When this fragment is added to our program outline, the program is complete:

[7] After some 262,144 characters have been typed, the program will escape from this loop, unless some other error supervenes.

```
        TITLE   Reverse a Line of Input.   Example 4A

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.
$

A=1                             ;symbolic names for accumulators
P=17                            ;register for the stack pointer

PDLEN==200                      ;room for 128 characters

PDLIST: BLOCK   PDLEN           ;reserve space for the stack

START:  RESET                   ;initialize the program
        OUTSTR  [ASCIZ/Welcome to the Reverse Program
/]                              ;send a friendly message
;initialize for another input line
GETLIN: OUTSTR  [ASCIZ/Please type a line:    /]
        MOVE    P,[IOWD PDLEN,PDLIST]  ;initialize stack pointer
CHLOOP: INCHWL  A               ;obtain a character
        CAIN    A,15            ;skip unless this is a carriage return
        JRST    CHLOOP          ;discard CR: go get the next character
        CAIN    A,12            ;skip unless this is a line feed
        JRST    EOLN            ;this is a line feed.  Jump to EOLN
        PUSH    P,A             ;add character to the stack
        JRST    CHLOOP          ;loop to obtain another character

EOLN:   CAMN    P,[IOWD PDLEN,PDLIST]   ;is the stack empty?
        JRST    STOP                    ;empty line, stop running.
        OUTSTR  [ASCIZ/The reversed line is:  /]
POPIT:  POP     P,A                     ;get one character from stack
        OUTCHR  A                       ;send it to the terminal
        CAME    P,[IOWD PDLEN,PDLIST]   ;is the stack empty now?
        JRST    POPIT                   ;not yet.  Loop again.
        OUTSTR  [ASCIZ/
/]                                      ;add CR LF to end of the output line
        JRST    GETLIN                  ;go get another input line

STOP:   OUTSTR  [ASCIZ/All done.
/]
        EXIT

        END     START
```

8.7.1. Summary of Example 4-A

The starting code executes a RESET, sends a greeting message 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. 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 CHLOOP, 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 CHLOOP reads the input line character by character. The INCHWL reads one character from the terminal; the character appears in register 1, which we call A. CHLOOP 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    EOLN
```

will branch to EOLN to leave the input loop.

At EOLN 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 EOLN jumps to STOP; the program will halt.

At POPIT, 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 OUTCHR. 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 POPIT will be executed to take us around the output loop once more. The program loops through the code sequence at POPIT 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 POPIT, 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.

There is one further infelicity of this program. The repeated calls to OUTCHR are inherently wasteful. Every system call is relatively expensive, regardless of how little useful work is performed. OUTSTR moves an entire string from the program to the system in just one system call. OUTCHR, in contrast, must be called once for every character that is printed: 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 MUUO call.

Part of the overhead in each monitor 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-10 to run a program more efficiently.

We have used OUTCHR in this example because we are not ready to deal with a string on a character-by-character basis. Where practicable, the overhead of repetitive monitor calls can be avoided by building an entire output string and sending it to the terminal with OUTSTR. In example 4-B, after we have demonstrated the byte instructions, we shall show a better way to perform this function.

Chapter 9
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-10 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.

9.1. OVERVIEW OF ASSEMBLY AND LOADING

Figure 9-1 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:UUOSYM.UNV`. A universal file is a file containing definitions; when we speak of the more complicated system calls, we will make use of such definitions.

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 *core-image.*

9.2. ASSEMBLER OUTPUT

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 (a slightly modified version of) our example 4-A, the Reverse program. The listing that follows was obtained by entering the text of the program into the file `EX4A.MAC` and then issuing the commands:

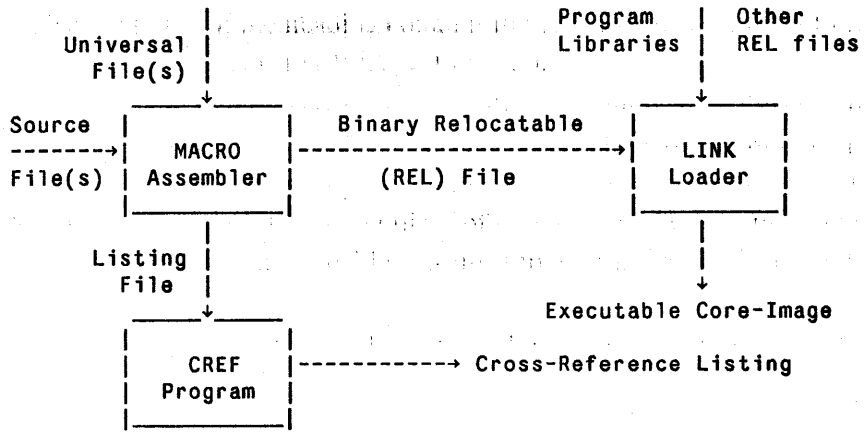


Figure 9-1: Overview of the Assembler and Loader

```
.COMPILE/COMPILE/CREF EX4A.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 9-2, 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 9-3. 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 9-4. 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

- 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

```
Reverse a Line of Input.   Example 4A   MACRO %53A(1152) 15:17 28-Aug-81 Page 1
EX4A    MAC     28-Aug-81 15:15

    1                                   TITLE   Reverse a Line of Input.   Example 4A
    2
    3                                   Comment $
    4                                           Program to reverse the characters on each line of input.
    5                                           Program will stop when an empty line is input.
    6
    7                                           This program demonstrates the last-in, first-out property
    8                                           of push-down stacks.
    9                                   $
   10
   11                   000001          A=1                             ;symbolic names for accumulators
   12                   000017          P=17                            ;register for the stack pointer
   13
   14                   000200          PDLEN==200                      ;room for 128 characters
   15
   16  000000' 127 145 154 143 157     WELCOM: ASCIZ   /Welcome to the Reverse Program
   17  000001' 155 145 040 164 157
   18  000002' 040 164 150 145 040
   19  000003' 122 145 166 145 162
   20  000004' 163 145 040 120 162
   21  000005' 157 147 162 141 155
   22  000006' 015 012 000 000 000     /
   23
   24  000007'                         PDLIST: BLOCK   PDLEN            ;reserve space for the stack
   25
   26  000207' 047 00 0 00 000000      START:  RESET                    ;initialize the program
   27  000210' 051 03 0 00 000000'             OUTSTR  WELCOM           ;send a friendly message
   28                                  ;initialize for another input line
   29  000211' 051 03 0 00 000236'     GETLIN: OUTSTR  [ASCIZ/Please type a line:    /]
   30  000212' 200 17 0 00 000242'             MOVE    P,[IOWD PDLEN,PDLIST]  ;initialize stack pointer
   31  000213' 051 04 0 00 000001      CHLOOP: INCHWL  A                ;obtain a character
   32  000214' 306 01 0 00 000015              CAIN    A,15             ;skip unless this is a carriage return
   33  000215' 254 00 0 00 000213'             JRST    CHLOOP           ;discard CR: go get the next character
   34  000216' 306 01 0 00 000012              CAIN    A,12             ;skip unless this is a line feed
   35  000217' 254 00 0 00 000222'             JRST    EOLN             ;this is a line feed.  Jump to EOLN
   36  000220' 261 17 0 00 000001              PUSH    P,A              ;add character to the stack
   37  000221' 254 00 0 00 000213'             JRST    CHLOOP           ;loop to obtain another character
   38
   39  000222' 316 17 0 00 000242'     EOLN:   CAMN    P,[IOWD PDLEN,PDLIST]   ;is the stack empty?
   40  000223' 254 00 0 00 000233'             JRST    STOP                    ;empty line, stop running.
   41  000224' 051 03 0 00 000243'             OUTSTR  [ASCIZ/The reversed line is:  /]
   42  000225' 262 17 0 00 000001      POPIT:  POP     P,A                     ;get one character from stack
   43  000226' 051 01 0 00 000001              OUTCHR  A                       ;send it to the terminal
   44  000227' 312 17 0 00 000242'             CAME    P,[IOWD PDLEN,PDLIST]   ;is the stack empty now?
   45  000230' 254 00 0 00 000225'             JRST    POPIT                   ;not yet.  Loop again.
   46                                          OUTSTR  [ASCIZ/
   47  000231' 051 03 0 00 000250'     /]                                      ;add CR LF to end of the output line
   48  000232' 254 00 0 00 000211'             JRST    GETLIN                  ;go get another input line
   49
   50                                  STOP:   OUTSTR  [ASCIZ/All done.
   51  000233' 051 03 0 00 000251'     /]
   52  000234' 047 00 0 00 000012              EXIT
   53
   54                   000207'                END     START

NO ERRORS DETECTED

PROGRAM BREAK IS 000254
CPU TIME USED 00:00.233

17P CORE USED
```

Figure 9-2: Assembler Listing of the Source Program

```
Reverse a Line of Input.   Example 4A   MACRO %53A(1152) 15:17 28-Aug-81 Page S-1
EX4A    MAC     28-Aug-81 15:15         SYMBOL TABLE

A               000001
CHLOOP          000213'
EOLN            000222'
EXIT    047000  000012
GETLIN          000211'
INCHWL  051200  000000
OUTCHR  051040  000000
OUTSTR  051140  000000
P               000017
PDLEN           000200  spd
PDLIST          000007'
POPIT           000225'
RESET   047000  000000
START           000207'
STOP            000233'
WELCOM          000000'
```

Figure 9-3: Assembler Listing of the Symbol Table

```
                                        Symbol Cross-Reference
A       11#     31      32      34      36      42      43
CHLOOP  31#     33      37
EOLN    35      39#
GETLIN  29#     48
P       12#     30      36      39      42      44
PDLEN   14#     24      30      39      44
PDLIST  24#     30      39      44
POPIT   42#     45
START   26#     54
STOP    40      50#
WELCOM  16#     27

                                        Operator Cross-Reference
EXIT    52
INCHWL  31
OUTCHR  43
OUTSTR  27      29      41      46      50
RESET   26
```

Figure 9-4: Assembler Listing of the Source Program

symbols; the second includes all the special operators and macros used in the program.[1] 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.

9.2.1. Page Headings

MACRO produces a page heading on each page of the listing file. A sample heading appears below:

```
Reverse a Line of Input.   Example 4A   MACRO %53A(1152) 15:17 28-Aug-81 Page 1
EX4A    MAC     28-Aug-81 15:15
```

[1] See the discussion of `OPDEF`, section 13.4, page 134, and the discussion of macros in section 17, page 173.

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 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 SOS editor (and to form-feed characters in a TECO file). 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.

9.2.2. 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 more than one; see lines 16 and 24.
- The location counter may be flagged with an apostrophe character (') signifying that the value of the counter is *relocatable*. The section that follows this discusses relocatable symbols and the loader.
- 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 16 through 22 each show the five 7-bit fields that MACRO assembled for the ASCIZ pseudo-op.
- Finally, the source text of the line is printed. If the SOS editor is used for the preparation of the source program, the SOS line numbers will appear with the text of each line.

We will now examine the features found on several lines of this output listing. Please refer to figure 9-5.

On line 33 we see that the assembler is planning to store a word in location 215; the apostrophe after the 215 signals that this value is *relocatable*. 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 215 words after the first location that is loaded. We will discuss relocation further in the next section; meanwhile, it is important to note that locations such as relocatable 6 are not included among the accumulators, even though it looks like a small enough address.

Continuing on line 33, the operation code has been assembled as 254; this corresponds to the JRST instruction. The accumulator field is 0. The Y field is relocatable 213; this must be the assembler's translation of the symbol CHLOOP. We can verify this assumption by examining line 31 where the symbol CHLOOP is defined: note that the location counter there has the value relocatable 213. Thus we see the effect of some of the assembler's bookkeeping functions. The symbol CHLOOP is defined as 000213' (i.e., relocatable 213); at places where CHLOOP is referred to, that definition has been substituted for the symbolic name. On line 37, for example, the reference to CHLOOP has also been translated to 213'.

```
16  000000' 127 145 154 143 157     WELCOM: ASCIZ   /Welcome to the Reverse Program
22  000006' 015 012 000 000 000     /
23
24  000007'                         PDLIST: BLOCK   PDLEN           ;reserve space for the stack
25
26  000207' 047 00 0 00 000000      START:  RESET                   ;initialize the program
27  000210' 051 03 0 00 000000'             OUTSTR  WELCOM          ;send a friendly message
28                                  ;initialize for another input line
29  000211' 051 03 0 00 000235'     GETLIN: OUTSTR  [ASCIZ/Please type a line:    /]
30  000212' 200 17 0 00 000242'             MOVE    P,[IOWD PDLEN,PDLIST]  ;initialize stack pointer
31  000213' 051 04 0 00 000001      CHLOOP: INCHWL  A               ;obtain a character
32  000214' 306 01 0 00 000015              CAIN    A,15            ;skip unless this is a carriage return
33  000215' 254 00 0 00 000213'             JRST    CHLOOP          ;discard CR: go get the next character
 ↑      ↑   ↑  ↑ ↑  ↑      ↑      ↑
 |      |   |  | |  |      |      |
 |      |   |  | |  |      |      *--- the source text begins here
 |      |   |  | |  |      *---------- Y portion
 |      |   |  | |  *----------------- X portion
 |      |   |  | *-------------------- I part
 |      |   |  *---------------------- AC field
 |      |   *------------------------- Opcode
 |      *----------------------------- Location counter
 *------------------------------------ Cref Line number
```

Figure 9-5: Selected Lines from the CREF Listing

On line 32, the expression 15 has been assembled to 15. Note that no apostrophe follows the 15 in the code. This means that 15 is not relocatable; symbols and expressions that are not relocatable are said to be *absolute.*

Line 29 displays the first literal in the program. Note that MACRO indicates 235' as the address part of this instruction. Thus, MACRO must have decided to put the assembled code corresponding to this literal at location 235'. To verify this, note that line 54 in figure 9-2, the last line that was assembled, uses location 234'. This literal has been placed in the first location following the code listed with the text of the program. Literals are normally placed after all other material has been assembled.[2]

Another literal is displayed in lines 30, 39 and 44. Note that in every case, the same address is assembled. When MACRO sees two or more copies of the same literal, it assembles only one copy. Each reference to that literal uses the same copy.

Lines 16 through 22 show the assembler's translation of the ASCIZ pseudo-op. Line 16 shows the location counter, 0', and the octal for the first five characters of the prompting message. The other characters appear on subsequent lines.

9.2.3. Listing the Symbol Table

Examination of figure 9-3 shows the entire symbol table, with symbol names arranged alphabetically. On this page too, the symbol CHLOOP appears with the value 000213'.

Most labels have relocatable values. These include START, GETLIN, and STOP. The symbols A, P, and PDLEN are not relocatable.

Some symbols have 36-bit values. EXIT, for example, is defined as 047000 000012. You might notice that all the MUUO values start with operation codes in the range from 40 to 77. The codes for RESET

---

[2]See also the discussion of the LIT Pseudo-op in [MACRO].

and EXIT start with opcode 047; this is the CALLI MUUO, which decodes its effective address as the function number. The right half value, 12 in the case of EXIT, specifies which system function is wanted. The terminal input and output calls all begin with opcode 051; this is the TTCALL MUUO, which decodes its accumulator field as the function number. INCHWL, for instance, contains an accumulator field of 4 (appearing as 200). In the bad old days we would have to write TTCALL 4,A instead of INCHWL A; fortunately, MACRO has been augmented by the inclusion of many useful definitions.

Some symbol values are followed by a three-letter code. The code "spd" means suppressed, and applies to our one suppressed symbol, PDLEN. The code "int" means *internal* and does not occur in this example. Internal symbols are those that are available or visible to other programs that you load. The code "ext" means *external*; again, we have none in this example. External symbols are those that are are referred to in this program module, but which are not defined here; it is assumed that they are internal to some other program module.

9.2.4. Symbol Cross-Reference

The section 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 appears on lines 40 and 50. The sharp sign (#) after 50 means that STOP is defined there. These numbers refer to the line numbers at the extreme left of the listing.

9.2.5. Operator Cross-Reference

The operator cross-reference has the same format as the symbol cross-reference. Operators (including the MUUO names) 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 134; such operators would be listed here also.

9.3. 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 programs starting at location 140. TOPS-10 uses the locations in the range from 20 through 137 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.

Normally, MACRO will assemble what is called *relocatable code.* The loader 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 of the job.

In example 1, when we said that MACRO assigned the value 143 to the symbol MESAGE we did not tell the entire truth. Actually MACRO assigned the value 3' to the symbol MESAGE. Again, the apostrophe after

the 3 means that the value is relocatable. The relocation mark (or its absence) is associated with every symbol definition and with every word that MACRO assembles.[3] 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 `OUTSTR MESAGE` it assembled a word containing `051140000003';3'` is the value of `MESAGE`.

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 program. In this way, MACRO defers until load-time the decision about where to put the program. In the case of this `OUTSTR` instruction, LINK will add the relocation constant, `140`, to produce `051140000143`. This value is what LINK actually places into memory. 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 `EXIT`, value `047000000012`, is absolute. Absolute symbols and expressions are not relocated by LINK.

If you would examine line `32` in the assembler listing, the value of the expression `15` is simply `15`; note that the `15` does not have an apostrophe. Because the `15` 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].

---

[3] Actually, there are two relocation marks (or absences) for each word - one for the left half and one for the right.

Chapter 10
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 373. When a problem arises, you might want to review that list before attacking the problem with DDT.

10.1. 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.

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.

10.2. LOADING AND STARTING DDT

In TOPS-10, DDT must be added to the program at the time it is loaded into memory by LINK. DDT can be included with a program by using the 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 TOPS-10 command processor,
- 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, 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 DDT command. When you do so, TOPS-10 stores the old value of the program counter in the job data area location called .JBOPC, *JoB Old PC*, It will be necessary to use this value of the program counter if you decide to continue the execution of the program after your session in DDT.

When DDT is loaded with a program, the starting address of DDT can be found in the right half of the word called .JBDDT.

10.3. 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 4A, Reverse Line.

We begin by using the DEBUG command to assemble and load the file EX4A.MAC:

```
.debug ex4a.mac
MACRO:  Revers
LINK:   Loading
[LNKDEB DDT execution]
DDT
```

The DEBUG command causes LINK to load DDT with the program and to start DDT. When DDT is started, it types the message "DDT" and awaits our commands.

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 REVERS. We tell DDT to open the symbols for the program named REVERS by mentioning the program name and typing the command characters escape and colon.

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. (If your terminal does not have a key labeled escape or ESC, you might try Alt-Mode or CTRL/[.)

We type precisely the eight characters "R", "E", "V", "E", "R", "S", escape, and colon:

```
.debug ex4a.mac
MACRO:  Revers
LINK:   Loading
[LNKDEB DDT execution]
DDT
revers$:
```

DDT types a tab character after our colon to signify that it has accepted our command. The symbols for the REVERS 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.

```
[LNKDEB DDT execution]
DDT
revers$:    START/    RESET
```

The contents of the location called START have been displayed. This is the RESET MUUO. To continue examining locations, type the line feed key. 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 once:

```
[LNKDEB DDT execution]
DDT
revers$:    START/    RESET     type line feed
START+1/    OUTSTR WELCOM
```

The instruction that follows the RESET is displayed. 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.

It will be interesting to examine the string at WELCOM. We could type the name WELCOM and a slash. However, DDT provides the tab command (either the tab key or CTRL/I) to examine the location that the present instruction addresses. We type the tab character next:

```
[LNKDEB DDT execution]
DDT
revers$:   START/   RESET    type line feed
START+1/   OUTSTR WELCOM     type tab
WELCOM/    HRLES 2,@461736(15)
```

Now, at WELCOM 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 WELCOM DDT has stumbled; it has wrongly interpreted the contents of the word as an instruction.

This is yet one more reminder of a point that we have been trying to make: everything inside the computer is a number. The correct interpretation of the number is the responsibility of the programmer. Simply stated, the source program cannot be recovered by DDT. As the user of DDT, you must be sufficiently familiar with the program (or clever) to avoid going astray here.

Because we recognize this output as an anomaly of DDT, we can ask DDT for an alternative interpretation. By the command characters escape, "T", and semicolon, we instruct DDT to change the output mode to text and to retype the current quantity in the new mode:

```
[LNKDEB DDT execution]
DDT
revers$:   START/   RESET    type line feed
START+1/   OUTSTR WELCOM     type tab
WELCOM/    HRLES 2,@461736(15)    $t;Welco
```

In response to these commands, DDT retypes the contents of the word at WELCOM as text. In this case, we discover the word holds the five characters "Welco". We can go on to examine additional words by typing the line feed key:

```
[LNKDEB DDT execution]
DDT
revers$:   START/   RESET    type line feed
START+1/   OUTSTR WELCOM     type tab
WELCOM/    HRLES 2,@461736(15)    $t;Welco    type line feed
WELCOM+1/   me to    type line feed
WELCOM+2/    the     type line feed
WELCOM+3/   Rever    type line feed
WELCOM+4/   se Pr    type line feed
WELCOM+5/   ogram    type line feed
WELCOM+6/
```

The command characters escape and "t" set the *temporary output mode* to text. The line feed command does not clear the temporary mode, so while we type line feed characters, the output mode remains set to text mode.

The contents of the word at WELCOM+6 are not obvious from this display. We can change the mode again, this time by typing the command characters escape, 7, the letter O, and semicolon. This command changes the output mode to display left-adjusted seven-bit bytes and requests that the current quantity be re-typed:

```
WELCOM+5/    ogram    type line feed
WELCOM+6/
   $7o;15,12,0,0,0,0
```

The command $70 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 15 corresponds to the carriage return, 12 is the line feed. Then three zero bytes fill the word. The sixth field is present because five 7-bit fields do not entirely exhaust the 36-bit word. 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 $70. 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.

```
WELCOM+5/    ogram     type line feed
WELCOM+6/
   $7o;15,12,0,0,0,0     type carriage return
```

After typing carriage return the output mode reverts to symbolic. We now resume our progress through the program by typing START+1 and a slash, followed by some line feed characters:

```
WELCOM+6/
   $7o;15,12,0,0,0,0     type carriage return
start+1/     OUTSTR WELCOM          type line feed
GETLIN/      OUTSTR STOP+2          type line feed
GETLIN+1/    MOVE P,STOP+7          type line feed
CHLOOP/      INCHWL A               type line feed
CHLOOP+1/    CAIN A,15              type line feed
CHLOOP+2/    JRST CHLOOP            type line feed
CHLOOP+3/    CAIN A,12              type line feed
CHLOOP+4/    JRST EOLN              type line feed
CHLOOP+5/    PUSH P,A               type line feed
CHLOOP+6/    JRST CHLOOP            type line feed
EOLN/        CAMN P,STOP+7
```

DDT does a credible job of disassembly. Occasionally there are minor problems, such as the literals being turned into symbolic names that are not especially meaningful (e.g., STOP+7).

To demonstrate the concept of breakpoints, we will place a breakpoint at EOLN. 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, EOLN 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 EOLN$B, but .$B is a convenient shorthand.)

```
EOLN/    CAMN P,STOP+7      .$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:

```
EOLN/    CAMN P,STOP+7      .$b        $g
Welcome to the Reverse Program
Please type a line:
```

The program prompts for input, via the OUTSTR MUUO. Now, the INCHWL is being executed; TOPS-10 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-10 adds a line feed character and gives the entire line, character by character, to the program via the repeated executions of the INCHWL MUUO. The program runs until the INCHWL read the line feed character and jumps to EOLN. Instead of executing the instruction at EOLN, the

breakpoint there causes DDT to be called. DDT displays the location and contents of the current breakpoint:

```
EOLN/   CAMN P,STOP+7      .$b        $g
Welcome to the Reverse Program
Please type a line:    this is a test
$1B>>EOLN/   CAMN P,STOP+7
```

We are once again in DDT. Further examines are now permitted. We type the symbolic name of an accumulator, P, and a slash, to examine what the accumulator contains:

```
EOLN/   CAMN P,STOP+7      .$b        $g
Welcome to the Reverse Program
Please type a line:    this is a test
$1B>>EOLN/   CAMN P,STOP+7   P/   -162,,PDLIST+15    type return
```

The contents of register P, the stack pointer, are displayed. We can tell that 16 (octal) data items have been pushed on the stack, since the right half of register P has been advanced from PDLIST-1 to PDLIST+15. We shall now proceed to examine the contents of the stack. First, we type carriage return to close the current register. Then we type the name PDLIST and a slash:

```
$1B>>EOLN/   CAMN 17,STOP+7   17/   -162,,PDLIST+15    type return
pdlist/   PDLIST+15
```

Once again DDT has failed us. The contents of the word at PDLIST have been interpreted by DDT as PDLIST+15. Let us not go astray. We think there are characters to be seen there. First, let us type the command character equal-sign. When we do so, DDT responds by reporting the octal value of the current quantity:

```
$1B>>EOLN/   CAMN 17,STOP+7   17/   -162,,PDLIST+15    type return
pdlist/   PDLIST+15   =164
```

DDT reveals that the word at PDLIST contains octal 164. This is still insufficently responsive. We will once again try telling DDT to use text mode. We type escape, t, and semicolon:

```
$1B>>EOLN/   CAMN 17,STOP+7   17/   -162,,PDLIST+15    type return
pdlist/   PDLIST+15   =164    $t;t
```

At last, a resonable response. The first location in the stack contains the letter "t". Let us now examine the rest of the stack. We will type a series of line feed characters to examine the successive locations:

```
$1B>>EOLN/   CAMN 17,STOP+7   17/   -162,,PDLIST+15    type return
pdlist/   PDLIST+15   =164    $t;t  type line feed
PDLIST+1/   h       type line feed
PDLIST+2/   i       type line feed
PDLIST+3/   s       type line feed
PDLIST+4/           type line feed
PDLIST+5/   i       type line feed
PDLIST+6/   s       type line feed
PDLIST+7/           type line feed
PDLIST+10/  a       type line feed
PDLIST+11/          type line feed
PDLIST+12/  t       type line feed
PDLIST+13/  e       type line feed
PDLIST+14/  s       type line feed
PDLIST+15/  t
```

We see exactly what we expected: the first data item in the stack corresponds to the first character we typed; the last data item is the final "t" in "test." The stack top addresses the last character that was input.

We have now reached a point where we should be comfortable about allowing the program to proceed. Type the command characters escape and P. The instruction at the breakpoint will be executed, and execution resumes:

```
PDLIST+15/  t       $p
The reversed line is:  tset a si siht
Please type a line:
```

As the program continues, it types a heading and the reversal of the line that we typed. It loops to prompt again for input. We will now supply an empty line. After we type carriage return, the program runs until it hits the breakpoint:

```
Please type a line:      type return
$1B>>EOLN/    CAMN P,STOP+7
```

We will now experiment with single-stepping. We will type escape and "X". DDT will execute the current instruction:

```
Please type a line:
$1B>>EOLN/    CAMN P,STOP+7    $x
   P/    -200,,WELCOM+6    STOP+7/    -200,,WELCOM+6
EOLN+1/    JRST STOP
```

DDT types the contents of register P and the contents of the memory location STOP+7. We can observe that the contents are identical. DDT then executes the CAMN instruction; the CAMN does not skip. DDT then displays the next instruction that will be executed. Again we type escape and "X" to single step this instruction:

```
Please type a line:
$1B>>EOLN/    CAMN P,STOP+7    $x
   P/    -200,,WELCOM+6    STOP+7/    -200,,WELCOM+6
EOLN+1/    JRST STOP    $x
   STOP
<JUMP>
STOP/    OUTSTR STOP+16
```

Observe that DDT has executed (or rather simulated the execution of) the JRST to the label STOP. We can continue single-stepping, or we can allow the program to proceed at full speed. We will type escape and "P" for full-speed execution:

```
STOP/    OUTSTR STOP+16    $p
All done.
EXIT
^C
.
```

10.4. 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

- 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 suspicious 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.
- 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` in the startup sequence.
- Erroneous arguments to system calls and subroutines can cause much confusion.
- DDT and your program compete for the use of your terminal and keyboard. In particular, you cannot place a breakpoint in a loop that has an `INCHWL` MUUO. This is because DDT can not distinguish the keystrokes that you send to the program from the keystrokes that serve to control DDT. When it's necessary to debug a program that uses terminal input, the wisest thing to do is to read the entire terminal line at once and store the characters in a buffer. In this way, the terminal input portion will be in a simple and very localized region. Once that part works, DDT won't interfere with the use of the terminal. In the next chapter we'll discuss how to build such a buffer for terminal input.

When your program doesn't work, it is wise to adopt a skeptical attitude towards the code that you have written. The most difficult bugs to find are those located in the place where you know the program works "perfectly."

10.5. 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 359.

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 a decimal number.

10.5.1. 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 10.5.2, page 102 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*, 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.

10.5.1.1. 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.

10.5.1.2. 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`.

10.5.1.3. 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.

10.5.1.4. Deposit Commands

CR	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.
LF	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.
Back Space	The back space (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 ".".

10.5.2. 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 CR 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.
$nO	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 omitted, the value of n set by the previous $nO command will be used.
$nR	Set the display radix for numbers to n. The number n is decimal. ~~It is not meaningful for n to be larger than decimal 10.~~
$S	Display the contents of locations in symbolic mode, i.e., as instructions.
$T	Display quantities as 7-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 $70 is helpful.

10.5.3. 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 right 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 accumulators and 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.

`adr2,<adr1$nB` This is the command by which a breakpoint is installed. In this command, the address expression `adr1` specifies the location of the instruction at which to place the breakpoint. You cannot place a breakpoint at location zero.

The 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 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`, no location will be automatically displayed.

If a breakpoint is installed by a command with two consecutive escape characters, e.g., `$$nB`, then DDT will proceed from the breakpoint automatically. Automatic proceed continues until DDT detects that characters are available from the terminal when the breakpoint is executed. See the description of `$$P` below.

`$B` Remove all breakpoints.

`0$nB` Remove breakpoint `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 system call 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.

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$X` If the expression `instr` is a full-word quantity, then DDT will execute it as an instruction. If the left half of the given expression is zero, then the expression is interpreted as a repeat count as in the command n$X.

10.5.4. 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 CR, LF, ^, 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. MUUO 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.[1] 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 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.

[1] If an input/output instruction is being assembled, the comma signifies the end of the device number field.

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: "w$

10.5.5. 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.

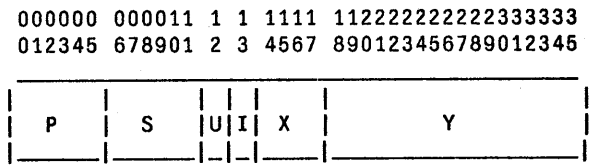
sym: Define the symbolic name sym to have the value of the current location (i.e., "."). If sym is already defined, this changes the old definition; otherwise, a new definition is added to the symbol table.

val<sym: Define the symbolic name sym to have the value specified by the expression val.

Chapter 11
Byte Instructions

In the PDP-10 a *byte* is some number of contiguous bits within one word. A *byte pointer* is a word 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. We will discuss six instructions that manipulate byte pointers and bytes.[1]

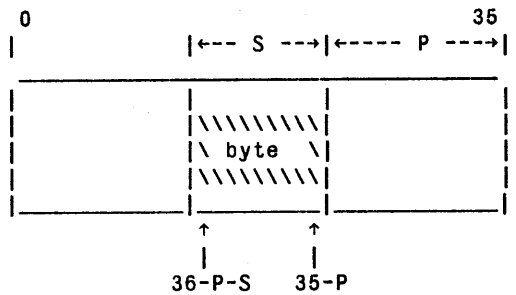
A byte pointer has the following format:



Several fields are present in a byte pointer word:

- 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, decimal 35 minus the bit number of the rightmost bit in the byte).
- The S field specifies the byte size in bits.
- The U field is ignored by older processors. In the extended KL10, this field should be zero.
- The I, X, and Y fields are the same as in an instruction.

Pictorially, a byte within a word looks like this:



[1]There are also instructions in the KL10 extended instruction set that manipulate strings; we will not discuss these, but you may consult [SYSREF] for details.

The byte includes consecutive bits starting at bit number 36-P-S through bit number 35-P. The byte includes S bits and is located P bits to the left of the right end of the word.

11.1. 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.

```
LDB     C(AC) := the byte described by C(E)
```

Example:

```
        LDB     5,[270400,,40]
```

This instruction 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.

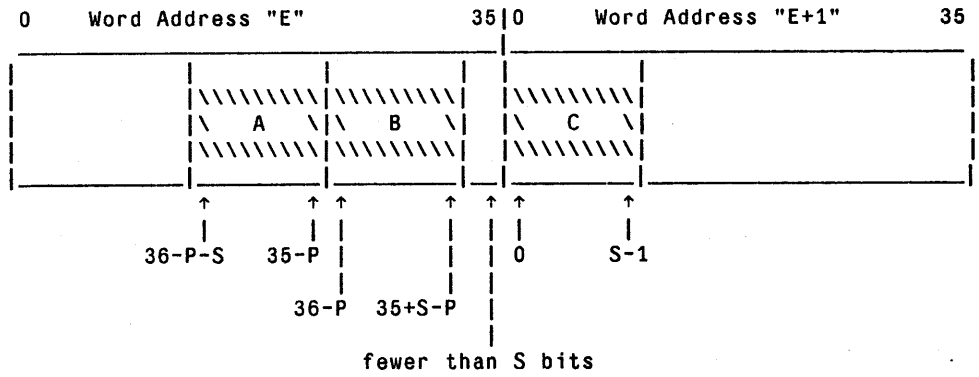
11.2. DPB - DEPOSIT BYTE

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-S:35 of C(AC)
```

11.3. 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 and 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 next word.



This diagram 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 left in 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 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 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-S (where 36 is decimal). 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 byte pointer is incremented.[2] Because the Y field of a pointer may be changed in this way, the programmer should avoid indirect addressing in any byte pointer that is 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 next two instructions, 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 is interpreted as a byte pointer; it is incremented. The updated byte pointer is stored at the effective address, replacing the original byte pointer.

```
IBP     C(E) := Incremented byte pointer in C(E).
                The AC field of the IBP instruction must be zero.
```

11.4. 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 in C(E).
        C(AC) := Byte described by the new C(E).
```

11.5. 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 in C(E).
        Byte described by the new C(E) := Bits 36-S:35 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 difficulty when initializing for string processing. We shall see that the POINT pseudo-op neatly allows us to handle this problem.

---

[2] 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.

11.6. POINT PSEUDO-OPERATOR

For convenience, the assembler has a pseudo-op for creating 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:

```
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
```

Once again, the size and position are interpreted as *decimal* numbers.

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. Let us try an example. Suppose that the word at 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 (or 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, 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 avoid trying 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.
```

11.7. PROGRAMMING EXAMPLE

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 in loop, process all characters

LOOPX:  . . .                   ;finished processing.
```

Usually this loop structure will be implemented in a subroutine.

11.8. 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.[3] 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, increment or decrement it by the 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.

```
ADJBP   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 position of the left-most byte of a word, as defined by the P and S fields. Numerically, the byte alignment is:

```
36-P modulo S           (36 is decimal)
```

Ordinarily, strings are packed with zero alignment because the IBP, ILDB and IDPB instructions all force the alignment to zero when a word boundary is crossed. ADJBP, however, preserves byte alignment when crossing a word boundary.

When given a reasonable byte pointer, ADJBP will returns 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 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-P) DIV S) + (P DIV S)      DIV means divide and truncate the quotient
                                to an integer.  The 36 is decimal.
```

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.[4]

If the number of bytes per word is zero, the divide check flag (see section 13, page 125) is set and the instruction avoids changing the accumulator or memory. 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.)

---

[3] ADJBP does not exist in the KI10 and earlier processors.

[4] If S is zero, the ADJBP instruction simply copies C(E) to the accumulator.

11.9. EXAMPLE 4-B

In this version of the reverse program we will build the output line in an *output buffer* before sending the line to the terminal. An output buffer is simply a place in memory where we collect data prior to sending it to an output device. Sending the entire output buffer to the terminal via OUTSTR is much more efficient than the repetitions of OUTCHR that we resorted to in example 4-A.

The input processing remains unaffected by this change. Let us reexamine the output loop from example 4-A:

```
EOLN:   CAMN    P,[IOWD PDLEN,PDLIST]   ;is the stack empty?
        JRST    STOP                    ;empty line, stop running.
        OUTSTR  [ASCIZ/The reversed line is:  /]
POPIT:  POP     P,A                     ;get one character from stack
        OUTCHR  A                       ;send it to the terminal
        CAME    P,[IOWD PDLEN,PDLIST]   ;is the stack empty now?
        JRST    POPIT                   ;not yet.  Loop again.
        OUTSTR  [ASCIZ/
/]                                      ;add CR LF to end of the output line
        JRST    GETLIN                  ;go get another input line
```

We focus on the OUTCHR at POPIT+1. It is this instruction that we should change. Our plan is to define an output buffer, establish a byte pointer to that buffer, and then to deposit characters into that buffer using the given byte pointer. When the POPIT loop terminates, we will have to add the characters carriage return, line feed and null to the output buffer. Then one OUTSTR operation will transmit the buffer to the terminal.

Let us begin by defining the output buffer. The buffer, again, is simply a group of consecutive memory locations that we will use for temporary storage. We define a symbolic name for the length of the buffer BUFLEN, and then use a BLOCK pseudo-op to actually reserve space:

```
BUFLEN==40
OBUFR:  BLOCK   BUFLEN

        . . .
```

Now, given that we have a buffer, we must have a byte poiunter that we can use to deposit characters into the buffer. Here we have to plan ahead a little ways. We are intending to use OUTSTR to send the data to the terminal. Thus far, we have always used OUTSTR in conjunction with text strings that have been created by our use of the ASCIZ pseudo-op. We should be certain that the buffer we build exactly matches the format of the strings made by that pseudo-op. The relevant characteristics of the ASCIZ pseudo-op are these:

- Strings are composed of 7-bit bytes, left-adjusted in consecutive words.
- The first character of a string begins at the left-end of a word.
- The string ends with a null (zero) byte.

This doesn't seem to be too difficult. Since the string is composed of 7-bit bytes, the byte size is dictated. Since the IDPB and ILDB functions already work with left-adjusted strings, it will come as no surprise that we'll use IDPB to store into the buffer. We will have to initialize the byte pointer so that the first IDPB stores into the character at the left-end of the word at OBUFR.

We know how to make such a byte pointer. It is simply POINT 7,OBUFR. We must initialize some location to contain this byte pointer prior to the first time through the output loop.

We shall hold the byte pointer in an accumulator that we will call B. It is not necessary that the byte pointer be held in an accumulator; it may be in any memory location. It is just convenient to hold it in an accumulator.

```
B=2
        . . .
        MOVE    B,[POINT 7,OBUFR]
POPIT:  . . .
```

Recall the explanation of the POINT pseudo-op: the byte pointer that is formed from the expression POINT 7,OBUFR points to the non-existent byte to left of the first real byte in the word at OBUFR. 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 store a byte. The instruction to replace OUTCHR at POPIT+1 should now be obvious. The byte pointer is in B; the byte is in A. An IDPB instruction does the trick:

```
B=2
        . . .
BUFLEN==40
OBUFR:  BLOCK   BUFLEN
        . . .
EOLN:   CAMN    P,[IOWD PDLEN,PDLIST]   ;is the stack empty?
        JRST    STOP                    ;empty line, stop running.
        OUTSTR  [ASCIZ/The reversed line is:  /]
        MOVE    B,[POINT 7,OBUFR]
POPIT:  POP     P,A                     ;get one character from stack
        IDPB    A,B                     ;send it to the output buffer
        CAME    P,[IOWD PDLEN,PDLIST]   ;is the stack empty now?
        JRST    POPIT                   ;not yet.  Loop again.
        . . .                           ;add CR LF and NULL to end the
                                        ;output line
        JRST    GETLIN                  ;go get another input line
```

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:

```
EOLN:   CAMN    P,[IOWD PDLEN,PDLIST]   ;is the stack empty?
        JRST    STOP                    ;empty line, stop running.
        OUTSTR  [ASCIZ/The reversed line is:  /]
        MOVE    B,[POINT 7,OBUFR]
POPIT:  POP     P,A                     ;get one character from stack
        IDPB    A,B                     ;send it to the output buffer
        CAME    P,[IOWD PDLEN,PDLIST]   ;is the stack empty now?
        JRST    POPIT                   ;not yet.  Loop again.
        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 OUTSTR happy
        OUTSTR  OBUFR                   ;Send the buffer to the terminal.
        JRST    GETLIN                  ;Go get another line
```

The instruction sequence at POPIT, 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 OUTSTR MUUO.

The entire program that we have constructed appears below:

```
        TITLE   REVERSE - Example 4-B

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
P=17                                    ;Symbolic for push down pointer

BUFLEN==40
PDLEN==200

OBUFR:  BLOCK   BUFLEN+1                ;buffer for output line
PDLIST: BLOCK   PDLEN

START:  RESET                           ;initialize IO
        OUTSTR  [ASCIZ/Welcome to the Reverse Program
/]                                      ;send a friendly message
;initialize for another input line
GETLIN: OUTSTR  [ASCIZ/Please type a line:    /]
        MOVE    P,[IOWD PDLEN,PDLIST]   ;initialize stack pointer
CHLOOP: INCHWL  A                       ;obtain a character
        CAIN    A,15                    ;skip unless this is a carriage return
        JRST    CHLOOP                  ;discard CR: go get the next character
        CAIN    A,12                    ;skip unless this is a line feed
        JRST    EOLN                    ;this is a line feed.  Jump to EOLN
        PUSH    P,A                     ;add character to the stack
        JRST    CHLOOP                  ;loop to obtain another character

EOLN:   CAMN    P,[IOWD PDLEN,PDLIST]   ;is the stack empty?
        JRST    STOP                    ;empty line, stop running.
        OUTSTR  [ASCIZ/The reversed line is:  /]
        MOVE    B,[POINT 7,OBUFR]
POPIT:  POP     P,A                     ;get one character from stack
        IDPB    A,B                     ;send it to the output buffer
        CAME    P,[IOWD PDLEN,PDLIST]   ;is the stack empty now?
        JRST    POPIT                   ;not yet.  Loop again.
        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 OUTSTR happy
        OUTSTR  OBUFR                   ;Send the buffer to the terminal.
        JRST    GETLIN                  ;Go get another line

STOP:   OUTSTR  [ASCIZ/All done.
/]
        EXIT

        END     START
```

11.10. CHARACTER PROCESSING; EXAMPLE 5

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 an outline plus the fragment that reads a line of input from the terminal. We will define a buffer space, called BUFFER, and the length of the buffer area, BUFLEN. The symbolic accumulator names that we have introduced will again 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.

```
BUFLEN==40
BUFFER:  BLOCK   BUFLEN                  ;space for an input line
         . . .                           ;initialize

GETLIN: OUTSTR   PROMPT                  ;prompt for input
        MOVE     B,[POINT 7,BUFFER]      ;read input to buffer area
GETCHR: INCHWL   A                       ;read a character
        CAIN     A,15                    ;is it a carriage return?
        JRST     GETCHR                  ;yes, discard CR
        CAIN     A,12                    ;is it a line feed?
        MOVEI    A,0                     ;yes, convert it to null to end line
        IDPB     A,B                     ;store the character in the buffer
        JUMPN    A,GETCHR                ;loop, unless end of line.

        . . .                            ;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 a null: we discarded the carriage return, and we have converted the line feed to a null before storing it. 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
        JUMPE    A,STOP                  ;if 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 character in odd buffer
        . . .                           ;store space in even buffer
        . . .                           ;get an even character
        . . .                           ;exit loop if end of line
        . . .                           ;store even character 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 quite simple: when we read the input line, we placed a null character in the buffer to mark the end. Each time we read a character from the buffer, we check to see if it's zero. Note that the existence of instructions such as JUMPE and JUMPN make it easy to test for a null character.

```
LOOP:   ILDB    A,INP                   ;get an odd character
        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
        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
        . . .                           ;store even character 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
        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
```

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 OUTSTR happy.
```

Now that the lines have been created, they must be printed:

```
        OUTSTR  HEAD                    ;print leading spaces
        OUTSTR  OLINE                   ;print the odd line
        OUTSTR  HEAD                    ;print more spaces
        OUTSTR  ELINE                   ;print the even characters
```

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

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
```

```
START:  RESET                           ;a good way to start
        OUTSTR  [ASCIZ/Welcome to Even-Odd
/]                                      ;send greetings
GETLIN: OUTSTR  PROMPT                  ;prompt for input
        MOVE    B,[POINT 7,BUFFER]      ;read input to buffer area
;obtain an input line
GETCHR: INCHWL  A                       ;read a character
        CAIN    A,15                    ;is it a carriage return?
        JRST    GETCHR                  ;yes, discard CR
        CAIN    A,12                    ;is it a line feed?
        MOVEI   A,0                     ;yes, convert it to null to end line
        IDPB    A,B                     ;store the character in the buffer
        JUMPN   A,GETCHR                ;loop, unless end of line.
;determine if the line is empty
        LDB     A,[POINT 7,BUFFER,6]    ;read the first character
        JUMPE   A,STOP                  ;if 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
        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
        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 OUTSTR happy.
;print the odd line, then the even line
        OUTSTR  HEAD                    ;print leading spaces
        OUTSTR  OLINE                   ;print the odd line
        OUTSTR  HEAD                    ;print more spaces
        OUTSTR  ELINE                   ;print the even characters
;done with one line.
        JRST    GETLIN                  ;do another line

STOP:   EXIT

BUFLEN==40
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
```

11.11. 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.

11.11.1. 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
        JUMPE   A,LOOPX                 ;leave loop if null is seen
        JUMPE   ODDC,ELOOP              ;jump when doing an even character
        IDPB    A,ODDP                  ;deposit an odd character
        IDPB    B,EVENP                 ;deposit a blank in even line
        SOJA    ODDC,LOOP               ;set to 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               ;set to odd for next time; loop
```

11.11.2. 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
        JUMPE   A,LOOPX                 ;leave loop if null is seen
        IDPB    A,ODDP                  ;deposit the character in one place
        IDPB    B,EVENP                 ;and a blank in the other.
        EXCH    ODDP,EVENP              ;exchange odd and even pointers!
        JRST    LOOP
```

11.12. EXERCISES

11.12.1. 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
        JUMPE   A,STOP                  ;line is empty.  Stop now.
```

11.12.2. 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.

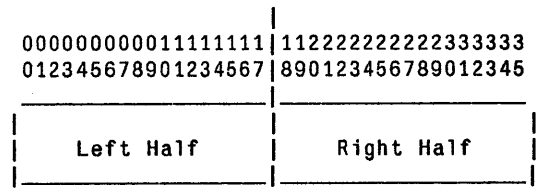
If the first line contains nothing more than a carriage return and line feed, make the program stop.

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 modification to perform 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.




```
H    halfword from the

        |R right  half of source
        |L left   half of source

to the      |R right half of destination
            |L left half of destination

with            |  no modification of the other half of the
                       destination
                |Z zero the other half of the destination
                |O set the other half of the destination to ones
                |E sign extend the source halfword into the other
                       half of the destination

where           |  source is memory and destination is AC
                |I source is the immediate quantity 0,,E and the
                       destination is AC
                |M destination is memory and source is AC
                |S source and destination are the same memory
                       location; if the selected accumulator is
                       other than AC 0 a copy of the entire
                       resulting destination word is stored in
                       the selected accumulator.
```

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.

```
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 usually 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 usually 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)   := 0,,CR(AC)
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) := CL(E),,0

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
```

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.

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.

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:

```
        SKIPN   A,ROOT          ;Get the address of the root node
        JRST    EMPTY           ;there is no tree at all
LOOP:   MOVE    B,A             ;save the address of the current node in B
        HLRZ    A,0(A)          ;change A to point to left subtree
        JUMPN   A,LOOP          ;Loop while the left subtree isn't empty
        MOVE    A,1(B)          ;Load A with datum from the leftmost leaf
        . . .                   ;B contains the address of the leftmost leaf
```

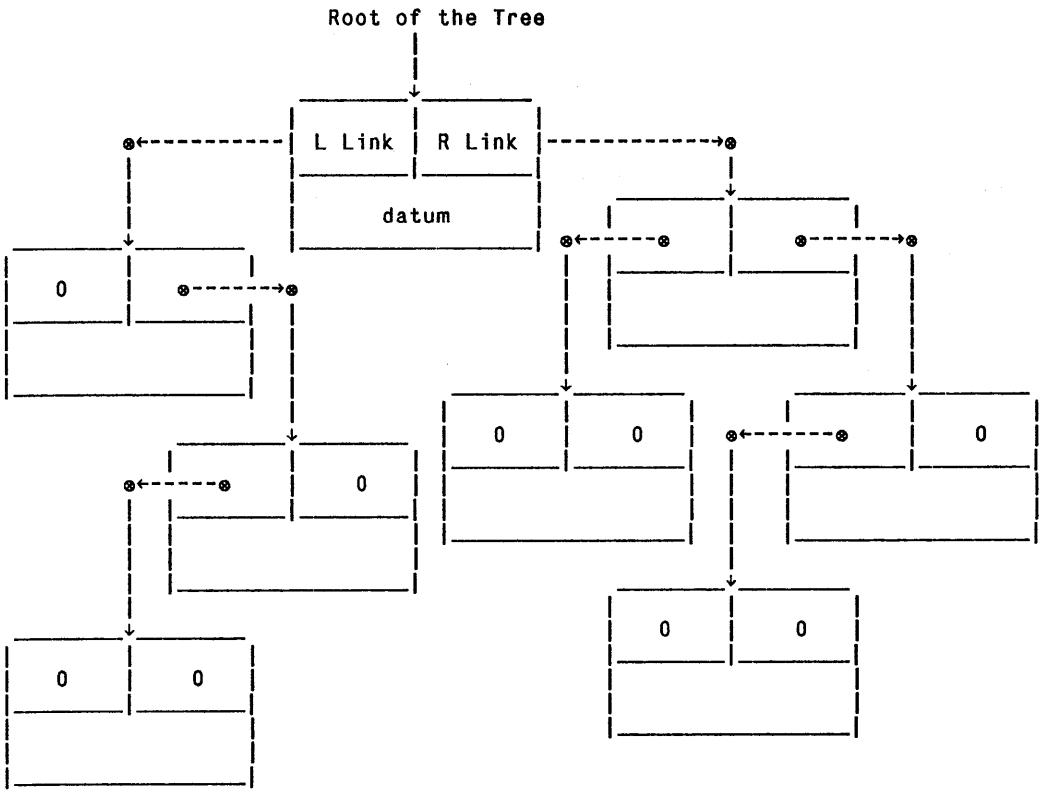


Figure 12-1: Binary Tree with Halfword Links

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.

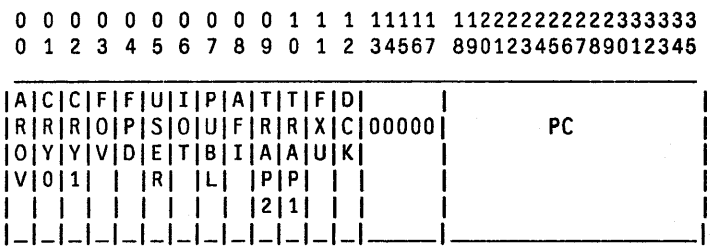
As we have already said, the program counter usually advances through consecutive memory locations as instructions are executed. The skip instructions change the program counter by incrementing it an extra time. The jump instructions supply an entire new value for the program counter. The subroutine 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 FORMAT

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.

The subroutine call instructions (PUSHJ, JSR, and JSP) store a full word that contains the program counter and the central processor flags. The flags reflect some of the state of the computation. If a subroutine means to be completely transparent to the calling program, it will make an effort to restore these flags. 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.

The format of the program counter and flags word is depicted below:[1]



[1] In the 2060 a double word may be needed in some circumstances.

For many of our programs, a detailed understanding of the PC flags is not important. The explanation of the more obscure flags appears in appendix A, page 353.

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.
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.[2]
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.[3]
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 -128. 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.

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.

13.2. 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 not all 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

---

[2]The TRAP2 flag does not exist in the KA10 and earlier machines.

[3]The TRAP1 flag does not exist in the KA10 and earlier machines.

associated return JRA are now so out of favor that they are discussed in the appendix of obsolete instructions (appendix D, page 369). Also, several additional forms of the JRST instruction will be described.

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.

```
PUSHJ   C(AC) := C(AC)+<1,,1>;  C(CR(AC)) := <flags,,PC>;  PC := E;
```

As in the PUSH instruction, a pushdown overflow condition occurs if the 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. 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.[4] The POPJ instruction unwinds the stack by decrementing both halves of the accumulator containing the stack pointer by 1. The effective address of the POPJ instruction is ignored.

```
POPJ    PC := CR(CR(AC)); C(AC) := C(AC)-<1,,1>
```

A pushdown overflow (actually an underflow) condition results if the 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.

---

[4]Except, while the extended machine is operating in a non-zero section, the left half word is assumed to contain the return PC section number.

```
        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!

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

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 130):

```
        POP     P,TEMP          ;pop the return pc and flags
        JRSTF   @TEMP           ;restore flags and return.
```

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 to work,
        POPJ    P,              ;P should not be accumulator number 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 right-half

contents of P. The result, since Y is zero, will be identical to the address contained in the right half 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 123.

```
        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*.[5] Then if you have a situation such as exists in the COUNT subroutine depicted above, where some conditional jump to a POPJ is needed, you'll have one that is already labeled.

Note that 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.

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 this table:

---

[5] Another convention is to use the label R, *return*, for this purpose.

```
JRST    0,      PC := E;
JRST    1,      PC := E; Portal Instruction
JRST    2,      PC := E; Restore Flags
JRST    4,      PC := E; Halt the Processor
JRST   10,      PC := E; Dismiss Current Interrupt
JRST   12,      PC := E; Restore Flags and Dismiss Interrupt
```

When the accumulator field is zero, a simple unconditional jump occurs; we have already mentioned that JRST is the favorite unconditional jump instruction.

13.2.4.1. JRSTF Jump and Restore Flags

JRST 2, (i.e., a JRST instruction in which the accumulator field has been set to 2) signifies the jump and restore flags operation; MACRO recognizes the mnemonic JRSTF for JRST 2,. If indirection is used in JRSTF, then the flags are restored from the last word fetched in the address calculation. If indexing is used with no 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 is usually 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. Other JRSTs

JRST 1, The PORTAL instruction 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 where it is willing to be entered.

The following modes of JRST are all illegal in user mode and are trapped as unimplemented instructions (MUUOs).

JRST 4, The HALT instruction sets the PC from E and stops the processor.

JRST 10, This instruction dismisses the current priority interrupt. Usually JRST 12, is used for this purpose since JRST 10, fails to restore flags.

JRST 12, The JEN instruction will dismiss the current priority interrupt and restore the PC and flags of the interrupted process. JEN combines the functions of JRST 10, and JRST 2,.

Additional subfunctions of JRST exist in the extended KL10 processor. Some of these will be discussed when we explain extended addressing.

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.

```
JSR     C(E) := <flags,,PC>; PC := E+1;
```

Programming example:

```
        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 if flags are unimportant.
```

Note that the line where the label SUB1 appears contains just 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.

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.

```
JSP     C(AC) := <flags,,PC>;   PC := E;
```

Programming example:

```
        JSP     AC,SUB2         ;call subroutine SUB2
        ...

SUB2:   ...                     ;first instruction of SUB2
        ...
        JRSTF   @AC             ;return, restoring flags.  JRST @AC
                                ; is permitted if flags are unimportant
                                ;or
        JRSTF   0(AC)           ;restore flags and return
                                ;or
        JRST    1(AC)           ;skip one instruction immediately following
                                ;the normal return address caller
```

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 point
         ...
        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 three additional control instructions to discuss. These are JFCL, JFFO, and XCT. 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.

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 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] AND IR[9:12]
```

0 then PC := E; PC[0:3] := PC[0:3] AND (NOT 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 instruction
NEXT:   inst                        ;instruction 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.

13.3.2. JFFO - Jump if Find First One

The JFFO instruction tests the selected accumulator. If the accumulator contains zero then AC+1 is set to zero and no jump occurs.[6] 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
           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 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.

13.3.3. XCT - Execute Instruction

The XCT instruction fetches the word specified by the effective address and executes that word as an instruction. If an instruction that stores the PC is executed from an XCT, then the return PC that is stored points to the instruction following the XCT. If an instruction executed via XCT should skip, then that skip is relative to the location of the XCT. The accumulator field of the XCT should be zero.[7]

```
XCT     Execute the instruction found in C(E)
```

The XCT instruction is quite helpful in two 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. An alternative, now considered disreputable, is to store the computed instruction into the normal stream of instructions. 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:

---

[6] When we speak of AC+1 we mean (AC+1 modulo octal 20). That is, 17+1 is 0.

[7] 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.

```
                                        ;here with the effective address in 6
        HLL     6,[MOVE 1,]             ;set the appropriate left half
        XCT     6                       ;half in register 6 and Execute it
```

The following sequence is quite disreputable. For compatibility with future processors, we strongly recommend *against* storing into the instruction stream:

```
        ;A bad example:  don't change the instruction stream!
                                        ;here with the effective address in 6
        HRRM    6,NEXT                  ;store address into 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:

```
        MOVE    7,CASNUM                ;copy case number to 7
        XCT     CASTAB(7)               ;execute appropriate case

        ...

CASTAB: PUSHJ   17,CASE0                ;The case table simply contains
        SOS     6,J                     ;instructions appropriate for
        MOVE    6,J                     ;each case.
        AOS     6,J
```

13.4. EXAMPLE 6-A

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 shall take this opportunity to introduce subroutines. We will write a subroutine that reads an entire line into a buffer region and stores a null byte to mark the end of it. The subroutine will be called GETLIN, *GET LINe.*

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,GETLIN to call this subroutine. The GETLIN 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.[8]

We will use nearly the same code as appears in example 5. The main difference is that GETLIN has been transformed into a subroutine. The GETLIN subroutine will use registers A and B; it will store the input line in the region called BUFFER; the input line will end with a null character.

We begin by writing a fragment that includes the initialization code and the call to the GETLIN subroutine. The usual definitions and buffer space declarations are made:

```
        TITLE   EXTRACT - Example 6-A

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
P=17                                    ;symbolic for push-down pointer

OPDEF   CALL    [PUSHJ P,]              ;Call a subroutine
OPDEF   RET     [POPJ P,]               ;Return from a subroutine.

BUFLEN==40                              ;buffer size
PDLEN==100                              ;stack size

START:  RESET                           ;reset i/o
        MOVE    P,[IOWD PDLEN,PDLIST]   ;initialize stack
NXTLIN: OUTSTR  PROMPT                  ;ask for input
        CALL    GETLIN
        . . .
```

[8]There is an obsolete TOPS-10 system call named CALL. The use of the CALL UUO 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 UUO. You must not add the OPDEF CALL to an old program unless you first check to make sure that there are no CALL UUOs. 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 UUO instead.

```
GETLIN: MOVE    B,[POINT 7,BUFFER]      ;read input to buffer area
GETCHR: INCHWL  A                       ;read a character
        CAIN    A,15                    ;is it a carriage return?
        JRST    GETCHR                  ;yes, discard CR
        CAIN    A,12                    ;is it a line feed?
        MOVEI   A,0                     ;yes, convert it to null to end line
        IDPB    A,B                     ;store the character in the buffer
        JUMPN   A,GETCHR                ;loop, unless end of line.
        RET
        . . .

STOP:   EXIT

BUFFER: BLOCK   BUFLEN
PDLIST: BLOCK   PDLEN
PROMPT: ASCIZ   /Type a line: /
        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 order to read the characters from BUFFER we will need a byte pointer. We will define the name INPTR to be one of the accumulators and initialize that accumulator to contain a byte pointer to BUFFER.

```
INPTR=5
        . . .                           ;initialize
        MOVE    INPTR,[POINT 7,BUFFER]  ;byte pointer to the input line
INLOOP: ILDB    A,INPTR                 ;get an odd character
        JUMPE   A,INDONE                ;exit from loop if end of line
        . . .                           ;store an odd character in BUFR2
        ILDB    A,INPTR                 ;get an even character
        JUMPE   A,INDONE                ;exit from loop if end of line
        . . .                           ;store an 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. 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.

```
INPTR=5                                 ;input byte pointer
ODDPTR=6                                ;odd buffer pointer
OUTPTR=7                                ;output buffer pointer

        . . .                           ;initialize
        MOVE    INPTR,[POINT 7,BUFFER]  ;byte pointer to the input line
        MOVE    OUTPTR,[POINT 7,OBUFR]  ;pointer to output buffer
        MOVE    ODDPTR,[POINT 7,BUFR2]  ;pointer for storing the odd letters
INLOOP: ILDB    A,INPTR                 ;get an odd character
        JUMPE   A,INDONE                ;exit from loop if end of line
        IDPB    A,ODDPTR                ;store an odd character in BUFR2
        ILDB    A,INPTR                 ;get an even character
        JUMPE   A,INDONE                ;exit from loop if end of line
        IDPB    A,OUTPTR                ;store an even character in OBUFR
        JRST    INLOOP                  ;continue until end of line

;here at the end of the first scan.
INDONE: . . .
```

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 the 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
        . . .                   ;deposit 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 buffer
        JRST    NXTLIN          ;get the next 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. go halt this 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. go halt this 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 a character from odd buffer
        JUMPE   A,ODONE                 ;jump to ODONE if odd buffer is empty
        . . .                           ;Is it a vowel?  If not, go to OLOOP1
        IDPB    A,ODDPTR                ;deposit a vowel in the vowel buffer
        JRST    OLOOP                   ;process another character

OLOOP1: IDPB    A,OUTPTR                ;deposit a non-vowel in output buffer
        JRST    OLOOP                   ;process another character

ODONE:  IDPB    A,OUTPTR                ;add null to the output buffer
        MOVEI   B,15                    ;add cr lf null to vowel buffer
        IDPB    B,ODDPTR
        MOVEI   B,12
        IDPB    B,ODDPTR
        IDPB    A,ODDPTR
        OUTSTR  OBUFR                   ;print output buffer
        OUTSTR  BUFR2                   ;print vowel buffer
        JRST    NXTLIN                  ;go get the next 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 NXTLIN 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:

- The input character will be in register A.
- Register A will not be changed by this subroutine.

- If the given character is not a vowel, the subroutine will return to the instruction immediately following the subroutine call; if the character is a vowel, the subroutine will skip past one instruction immediately following the subroutine 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 buffer
        JUMPE   A,ODONE                 ;jump to ODONE if odd buffer is empty
        CALL    ISVOW                   ;Is it a vowel?
        JRST    OLOOP1                  ; Not a vowel.  Go to OLOOP1
        IDPB    A,ODDPTR                ;deposit a vowel in the vowel buffer
        JRST    OLOOP                   ;process another character
        . . .

;Test character in A; skip if it is a vowel. No skip if it is not a vowel.
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 the RET if this is no 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 128.

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

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                                    ;symbolic for push-down pointer

BUFLEN==40                              ;buffer space
PDLEN==100                              ;stack size

OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ P,]

START:  RESET                           ;reset i/o
        MOVE    P,[IOWD PDLEN,PDLIST]   ;initialize stack
NXTLIN: OUTSTR  PROMPT                  ;ask for input
        CALL    GETLIN                  ;read the input line

;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]  ;pointer for storing the odd letters
INLOOP: ILDB    A,INPTR                 ;get an odd character
        JUMPE   A,INDONE                ;jump if end of line
        IDPB    A,ODDPTR                ;store odd character in buffer
        ILDB    A,INPTR                 ;get an even character
        JUMPE   A,INDONE                ;jump if end of line
        IDPB    A,OUTPTR                ;store even character for output
        JRST    INLOOP                  ;go on

;here at the end of the first scan.
INDONE: CAMN    ODDPTR,[POINT 7,BUFR2]  ;were there any odd characters?
        JRST    STOP                    ;no.  empty line.  stop now.
        IDPB    A,ODDPTR                ;Store null to end the odd buffer
;prepare for second scan
        MOVE    INPTR,[POINT 7,BUFR2]   ;fetch odd characters 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                ;not a vowel, store in output buffer
        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 OUTSTR
        IDPB    A,OUTPTR                ;end output buffer
        OUTSTR  OBUFR                   ;list of even chrs and odd consonants
        OUTSTR  BUFR2                   ;address of odd vowel string
        JRST    NEXT                    ;time for another input line

;Obtain an input line from the terminal.  Put it in BUFFER.  End with a Null
GETLIN: MOVE    B,[POINT 7,BUFFER]      ;Initial byte pointer to buffer area
GETCHR: INCHWL  A                       ;read a character from the input
        CAIN    A,15                    ;is this a carriage return?
        JRST    GETCHR                  ;yes, discard CR.
        CAIN    A,12                    ;Is it a line feed?
        MOVEI   A,0                     ;yes,  change to a null
        IDPB    A,B                     ;store character in the buffer
        JUMPN   A,GETCHR                ;loop for more, unless end of line
        RET

;Test character in A; skip if it is a vowel. No skip if it is not a vowel.
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 the RET if this is no vowel
CPOPJ1: AOS     (P)                     ;perform a skip-return
        RET

STOP:   EXIT

BUFFER: BLOCK   BUFLEN                  ;input buffer area
BUFR2:  BLOCK   BUFLEN                  ;odd characters buffer/vowel buffer
OBUFR:  BLOCK   BUFLEN                  ;output buffer
PDLIST: BLOCK   PDLEN                   ;stack space
PROMPT: ASCIZ   /Type a line: /
        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.

14.1. LOGICAL TESTING AND MODIFICATION

The test instructions are used for testing and modifying bits in an accumulator. There are sixty-four instructions. Each mnemonic begins with the letter T and is followed by three modifiers.

```
T Test accumulator

 |R Immediate right side mask
 |L Immediate left side mask
 |D Direct mask
 |S Swapped mask

       |N No modification of AC
       |Z Zero the bits in AC selected by the mask
       |O Set the bits in AC selected by the mask to One
       |C Complement the bits in AC selected by the mask

              |  Do not skip
              |N Skip if Not all the selected bits are zero
              |E Skip if all the selected bits Equal zero
              |A Always skip
```

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,,E>; for L it is <E,,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.

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.

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) AND (NOT mask).

When the modification code O appears, bits that are one in mask are set to one in the accumulator. C(AC) := C(AC) OR mask.

When the modification code C appears, bits that are one in mask are complemented in the accumulator. C(AC) := C(AC) XOR 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 below. In these descriptions, several new or unfamiliar operators appear:

```
∧       Boolean AND
∨       Boolean Inclusive OR
¬       Boolean Negation (One's Complement)
XOR     Boolean Exclusive OR
≡       Boolean Equivalence
```

```
TRN             No-op
TRNE            If CR(AC) ∧ E = 0 then skip
TRNN            If CR(AC) ∧ E ≠ 0 then skip
TRNA            Skip

TRZ             CR(AC) := CR(AC) ∧ ¬E
TRZE            If CR(AC) ∧ E = 0 then skip; CR(AC) := CR(AC) ∧ ¬E
TRZN            If CR(AC) ∧ E ≠ 0 then skip; CR(AC) := CR(AC) ∧ ¬E
TRZA            Skip; CR(AC) := CR(AC) ∧ ¬E

TRO             CR(AC) := CR(AC) ∨ E
TROE            If CR(AC) ∧ E = 0 then skip; CR(AC) := CR(AC) ∨ E
TRON            If CR(AC) ∧ E ≠ 0 then skip; CR(AC) := CR(AC) ∨ E
TROA            Skip; CR(AC) := CR(AC) ∨ E

TRC             CR(AC) := CR(AC) XOR E
TRCE            If CR(AC) ∧ E = 0 then skip; CR(AC) := CR(AC) XOR E
TRCN            If CR(AC) ∧ E ≠ 0 then skip; CR(AC) := CR(AC) XOR E
TRCA            Skip; CR(AC) := CR(AC) XOR E
```

```
TLN             No-op
TLNE            If CL(AC) ∧ E = 0 then skip
TLNN            If CL(AC) ∧ E ≠ 0 then skip
TLNA            Skip

TLZ             CL(AC) := CL(AC) ∧ ¬E
TLZE            If CL(AC) ∧ E = 0 then skip; CL(AC) := CL(AC) ∧ ¬E
TLZN            If CL(AC) ∧ E ≠ 0 then skip; CL(AC) := CL(AC) ∧ ¬E
TLZA            Skip; CL(AC) := CL(AC) ∧ ¬E

TLO             CL(AC) := CL(AC) ∨ E
TLOE            If CL(AC) ∧ E = 0 then skip; CL(AC) := CL(AC) ∨ E
TLON            If CL(AC) ∧ E ≠ 0 then skip; CL(AC) := CL(AC) ∨ E
TLOA            Skip; CL(AC) := CL(AC) ∨ E

TLC             CL(AC) := CL(AC) XOR E
TLCE            If CL(AC) ∧ E = 0 then skip; CL(AC) := CL(AC) XOR E
TLCN            If CL(AC) ∧ E ≠ 0 then skip; CL(AC) := CL(AC) XOR E
TLCA            Skip; CL(AC) := CL(AC) XOR E

TDN             No-op
TDNE            If C(AC) ∧ C(E) = 0 then skip
TDNN            If C(AC) ∧ C(E) ≠ 0 then skip
TDNA            Skip

TDZ             C(AC) := C(AC) ∧ ¬C(E)
TDZE            If C(AC) ∧ C(E) = 0 then skip; C(AC) := C(AC) ∧ ¬C(E)
TDZN            If C(AC) ∧ C(E) ≠ 0 then skip; C(AC) := C(AC) ∧ ¬C(E)
TDZA            Skip; C(AC) := C(AC) ∧ ¬C(E)

TDO             C(AC) := C(AC) ∨ C(E)
TDOE            If C(AC) ∧ C(E) = 0 then skip; C(AC) := C(AC) ∨ C(E)
TDON            If C(AC) ∧ C(E) ≠ 0 then skip; C(AC) := C(AC) ∨ C(E)
TDOA            Skip; C(AC) := C(AC) ∨ C(E)

TDC             C(AC) := C(AC) XOR C(E)
TDCE            If C(AC) ∧ C(E) = 0 then skip; C(AC) := C(AC) XOR C(E)
TDCN            If C(AC) ∧ C(E) ≠ 0 then skip; C(AC) := C(AC) XOR C(E)
TDCA            Skip; C(AC) := C(AC) XOR C(E)

TSN             No-op
TSNE            If C(AC) ∧ CS(E) = 0 then skip
TSNN            If C(AC) ∧ CS(E) ≠ 0 then skip
TSNA            Skip

TSZ             C(AC) := C(AC) ∧ ¬CS(E)
TSZE            If C(AC) ∧ CS(E) = 0 then skip; C(AC) := C(AC) ∧ ¬CS(E)
TSZN            If C(AC) ∧ CS(E) ≠ 0 then skip; C(AC) := C(AC) ∧ ¬CS(E)
TSZA            Skip; C(AC) := C(AC) ∧ ¬CS(E)

TSO             C(AC) := C(AC) ∨ CS(E)
TSOE            If C(AC) ∧ CS(E) = 0 then skip; C(AC) := C(AC) ∨ CS(E)
TSON            If C(AC) ∧ CS(E) ≠ 0 then skip; C(AC) := C(AC) ∨ CS(E)
TSOA            Skip; C(AC) := C(AC) ∨ CS(E)

TSC             C(AC) := C(AC) XOR CS(E)
TSCE            If C(AC) ∧ CS(E) = 0 then skip; C(AC) := C(AC) XOR CS(E)
TSCN            If C(AC) ∧ CS(E) ≠ 0 then skip; C(AC) := C(AC) XOR CS(E)
TSCA            Skip; C(AC) := C(AC) XOR 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.

```
C(AC)   0   1   0   1
Mem     0   0   1   1

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

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,,E>`. The result is stored in the accumulator.

`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 location 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,,E
ANDM            C(E)  := C(AC) ∧ C(E)
ANDB            Temp  := C(AC) ∧ C(E); C(AC) := Temp; C(E) := Temp
```

Besides the usual arithmetic operations that MACRO performs on symbols and values, the Boolean AND of two values can be obtained by the ampersand operator. Thus, in MACRO, the expression 5&11 has the value 1.

```
ANDCA           C(AC) := ¬C(AC) ∧ C(E)
ANDCAI          C(AC) := ¬C(AC) ∧ 0,,E
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) := 0,,E[1]
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,,E>
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) XOR C(E)
XORI            C(AC) := C(AC) XOR 0,,E
XORM            C(E)  := C(AC) XOR C(E)
XORB            Temp  := C(AC) XOR C(E); C(AC) := Temp; C(E) := Temp
```

In the MACRO assembler, the exclusive OR of two values can be obtained by the operator ∧!. For example, if the symbol SNARK has the value 577 then the assignment BOOJUM==SNARK∧!173 would define the symbol BOOJUM to have the value 404. Note that this operator is composed of two characters.

```
IOR             C(AC) := C(AC) ∨ C(E)
IORI            C(AC) := C(AC) ∨ 0,,E
IORM            C(E)  := C(AC) ∨ C(E)
IORB            Temp  := C(AC) ∨ C(E); C(AC) := Temp; C(E) := Temp
```

In MACRO the Boolean OR of two values can be obtained by the exclamation point operator. Thus, the expression 5!11 has the value 15.

MACRO recognizes OR, ORI, etc. as alternative mnemonics for the IOR instructions.

```
ANDCB           C(AC) := ¬C(AC) ∧ ¬C(E)
ANDCBI          C(AC) := ¬C(AC) ∧ ¬<0,,E>
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,,E
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
```

---

[1] When an extended KL10 (the DECSYSTEM-2060) is operating in a non-zero PC section, the instruction SETMI will load an extended address into the selected accumulator. When used for this purpose, the approved mnemonic is XMOVEI, *eXtended MOVE Immediate.*

```
ORCA            C(AC) := ¬C(AC) v C(E)
ORCAI           C(AC) := ¬C(AC) v 0,,E
ORCAM           C(E)  := ¬C(AC) v C(E)
ORCAB           Temp  := ¬C(AC) v C(E);  C(AC) := Temp; C(E) := Temp

SETCM           C(AC) := ¬C(E)
SETCMI          C(AC) := ¬<0,,E>
SETCMM          C(E)  := ¬C(E)
SETCMB          C(AC) := ¬C(E); C(E) := ¬C(E)
```

In the MACRO assembler, the Boolean negation (i.e., the NOT or one's complement) of a value can be obtained by the operator ∧-. For example, if the symbol XYZZY has the value 173, then the expression ∧-XYZZY would have the value 777777777604. Note this operator is composed of two characters.

```
ORCM            C(AC) := C(AC) v ¬C(E)
ORCMI           C(AC) := C(AC) v ¬<0,,E>
ORCMM           C(E)  := C(AC) v ¬C(E)
ORCMB           Temp  := C(AC) v ¬C(E);  C(AC) := Temp; C(E) := Temp

ORCB            C(AC) := ¬C(AC) v ¬C(E)
ORCBI           C(AC) := ¬C(AC) v ¬<0,,E>
ORCBM           C(E)  := ¬C(AC) v ¬C(E)
ORCBB           Temp  := ¬C(AC) v ¬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
```

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
```

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

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,]

START:  RESET                           ;Begin execution here
        MOVE    P,[IOWD PDLEN,PDLIST]   ;initialize a stack pointer
        OUTSTR  [ASCIZ/Welcome to Extract
/]
NXTLIN: OUTSTR  PROMPT                  ;request an input line
        CALL    GETLIN                  ;get the input line

        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:   ILDB    A,INPTR                 ;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)
        OUTSTR  OBUFR                   ;send even + odd non-vowels
        OUTSTR  BUFFER                  ;send buffer (odd vowels)
        OUTSTR  CRLF                    ;send crlf
        JRST    NXTLIN                  ;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

;Obtain an input line from the terminal.  Put it in BUFFER.  End with a Null
GETLIN: MOVE    B,[POINT 7,BUFFER]      ;Initial byte pointer to buffer area
GETCHR: INCHWL  A                       ;read a character from the input
        CAIN    A,15                    ;is this a carriage return?
        JRST    GETCHR                  ;yes, discard CR.
        CAIN    A,12                    ;Is it a line feed?
        MOVEI   A,0                     ;yes,  change to a null
        IDPB    A,B                     ;store character in the buffer
        JUMPN   A,GETCHR                ;loop for more, unless end of line
        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:  PUSH    P,[SETD]                ;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"
        JRST    .+2                     ;C is not lower case.
        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.
        AOBJN   D,ISVOW1                ;not yet. Loop unless at end of table
CPOPJ:  RET                             ;return

VOWTAB: "A"
        "E"
        "I"
        "O"
        "U"
        "Y"
VOWTLN==.-VOWTAB
```

```
STOP:   EXIT

BUFLEN==40
BUFFER: BLOCK   BUFLEN
OBUFR:  BLOCK   BUFLEN
PDLEN==40
PDLIST: BLOCK   PDLEN
CRLF:   BYTE    (7)15,12
PROMPT: ASCIZ   /Type a line: /
        END     START
```

14.3.1. Analysis of Program 6-B

The portions of the program at START and NXTLIN 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:   ILDB    A,INPTR                 ;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, ILDB A,INPTR is performed to obtain a character from the input buffer. When no further characters are available, a zero appears in register A. When the input line is exhausted, the program will jump to PEND to terminate this pass.

Once a character has been fetched by this ILDB, 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 describing the code at LOOP. Assuming that PROC1 works as described, the next instruction is XCT DISPOSE(D). Note that the result in D, as returned by PROC1, is used to modify this instruction.[2] 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; 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

---

[2]The seven-letter name, DISPOSE, is longer than allowable symbol names; MACRO will shorten the name to DISPOS.

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 label SETD. This label appears on the TDZA, SETO sequence that follows PROC1. Thus, we can 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.

Finally, we eliminate the JRST ISVOW in PROC2 by the trivial expedient of moving PROC2 to immediately precede ISVOW:

```
PROC2:  PUSH    P,[SETD]                ;save return address
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.[3] 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.

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.

14.3.1.5. 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:

---

[3]Only the right half of an index register is significant in address calculations (except in a system that is using extended addressing).

```
CRLF:   BYTE(7)15,12
```

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, BLT, instruction, and the various shift instructions.

15.1. BLT INSTRUCTION

The BLT (BLock Transfer) instruction copies a block of consecutive words from one place in 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.

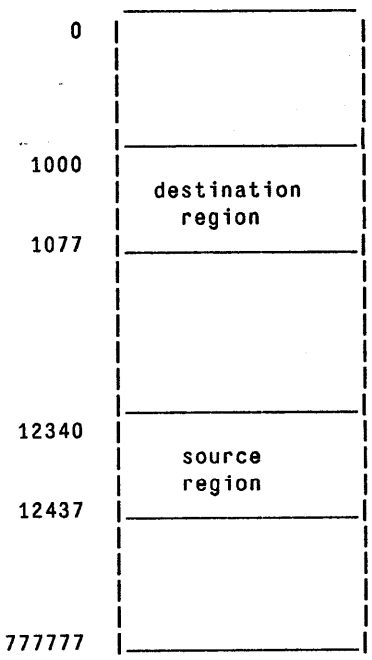
More specifically, the left half of the BLT accumulator specifies the first source address. The right half of the accumulator is the first destination address. The effective address of the BLT is the last destination address. Words are copied, one by one, from the source to the destination, until a word 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.

15.1.1. Warnings about BLT

Because there are so many different things going on inside BLT, there are some things that must be avoided when using a BLT. Despite the following list of warnings, BLT is quite useful, but it must be used carefully.

- BLT modifies its accumulator. On the KL10 and KS10 processors, when BLT is finished, the



Example: 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, is the address of the last word to store into.

Figure 15-1: BLT Example

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 any 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 it updates this pointer, it can resume its execution without repeating any work that it did previously. Note that 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.

15.1.2. BLT Programming Examples

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.

```
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
```

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                   ;Start by setting the first to zero
        MOVE    AC,[TABLE,,TABLE+1]     ;Source and
        BLT     AC,TABLE+77             ; destination overlap
```

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. 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.

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. 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.

15.2.1. 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 15-2.

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. In addition to the usual arithmetic operators, MACRO has a 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.

15.2.2. LSHC - Logical Shift Combined

Combined mode shifts involve the double-word accumulator pair formed from AC and AC+1 as shown in figure 15-3. The doubleword accumulator, denoted by C(AC AC+1), is 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 file names and device names in TOPS-10. 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
B=2                             ;For LSHC, B must be A+1

;Enter here with B containing a SIXBIT word.  Convert to ASCII and print
SIXOUT: JUMPE   B,[RET]         ;Return when nothing but 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 OUTCHR A          ;Print the ASCII character
        JRST    SIXOUT          ;Loop until B is empty.
```

15.2.3. 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. In a right-shift, bit 0 is copied into bit 1; this provides non-significant bits at the left end of the accumulator. See figure 15-4.

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 always 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.

15.2.4. 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 15-5. 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 is extended to AC bit 1.

15.2.5. ROT - Rotate

The 36-bit contents of the selected accumulator are rotated as shown in figure 15-6. 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.

15.2.6. ROTC - Rotate Combined

The data movement for the ROTC instruction is shown in figure 15-7: 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

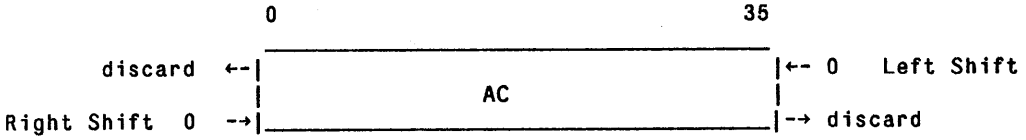


Figure 15-2: LSH Data Movement

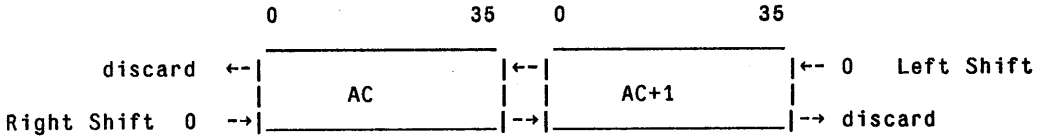


Figure 15-3: LSHC Data Movement

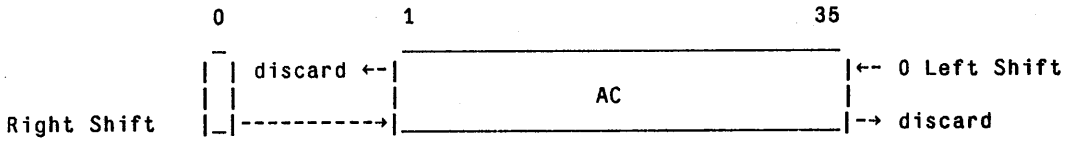


Figure 15-4: ASH Data Movement

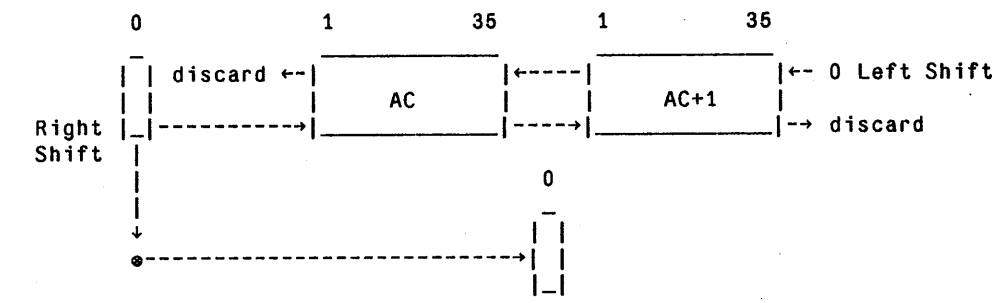


Figure 15-5: ASHC Data Movement

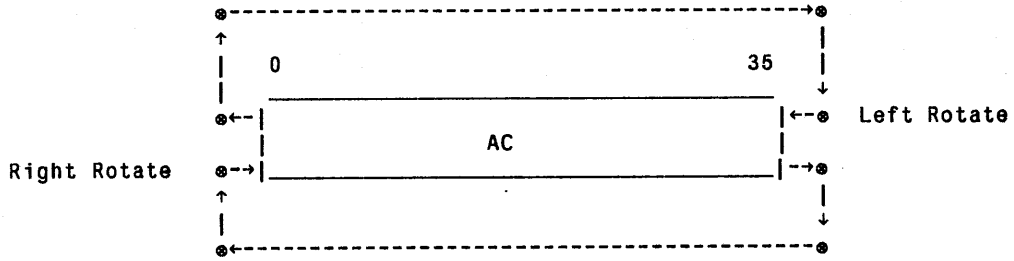


Figure 15-6: ROT Data Movement

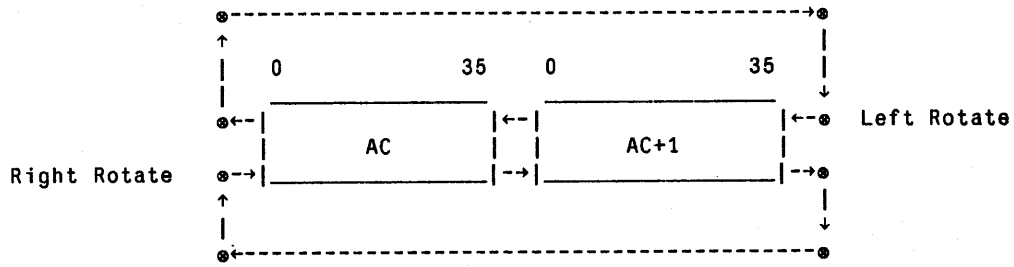


Figure 15-7: ROTC Data Movement

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 25.

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, in TOPS-10, 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 is greater than or equal to 2^{35} or less than -2^{35}. In case of an overflow, the result that is stored will be arithmetically correct, except the sign bit will be wrong.

16.1.3. IMUL Class

The IMUL instructions are for multiplying numbers where 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}.

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, AC+1 modulo 20 octal), except the IDIVM instruction does not store a remainder.

If the divisor is zero, the AROV and DCK (arithmetic overflow and 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 are

Dividend	Divisor	Quotient	Remainder
5	2	2	1
-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 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 dividend 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. If either condition obtains, AROV 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 MOVES

There are four double-word move instructions. These are suitable 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)
DMOVNM  C(E E+1)   := -C(AC AC+1)
```

In the KL10, DMOVN and DMOVNM can be used to manipulate double-precision integers. One word of caution however: 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. The DMOVN and DMOVNM instructions always set bit 0 of the second word to zero (double-precision floating-point format). In arithmetic operations, bit 0 of the second word is ignored. In comparisons, however, bit 0 of the second word would be looked at. For safety, if you use DMOVN to negate a double-precision integer, you should probably use a sequence like:

```
DMOVN   AC,E
SKIPGE  AC              ;skip if result is positive
TLO     AC+1,400000     ;set sign bit of second word.
```

---

[1] The DMOVN and DMOVNM instructions must not be used with any KA10-format double-precision floating-point numbers; see also appendix D, page 369.

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 70-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 140 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. FLOATING-POINT OPERATIONS

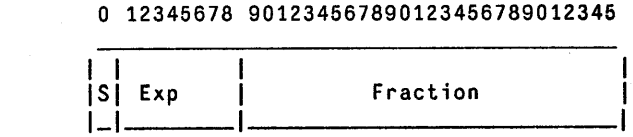
When it performs floating-point arithmetic operations, the computer hardware takes upon itself the burden of scaling numbers properly to make sensible results. Since the hardware helps in this way, floating-point is more suitable than fixed-point arithmetic for many purposes.

16.4.1. Floating-Point Representations

The PDP-10 offers two formats of floating-point numbers: single- and double-precision. 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 format.

16.4.1.1. Single-Precision Floating-Point

Single-precision floating-point numbers are represented in one 36-bit word as follows:



The field S is the sign bit. 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 represents a power of 2; it 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.[2] Conventionally, the fraction takes on some value greater than or equal to one-half and less than one.

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 16.4.2, page 169) bit 9 is different

[2] By a mistaken analogy to logarithms, the fraction is sometimes called the *mantissa*.

from bit 0.[3]

Floating-point numbers can represent numbers with magnitudes within the range from $0.5*2^{-128}$ to $(1-2^{-27})*2^{127}$, and 0. In more familiar notation, the magnitude range is approximately from $1.4*10^{-39}$ to $1.7*10^{38}$ and 0.

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. So we divide the 1.0 by 2, and increase the exponent: $0.5*2^1$. Now, 0.5 is easy to write as a binary fraction. It is 0.1. So we build a word that contains the binary pattern:

```
 0|10 000 001|100 000 000 000 000 000 000 000 000
 ↑  exponent ↑   fraction
sign     binary point
```

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:

```
 0|10 000 010|100 000 000 000 000 000 000 000 000
 ↑  exponent ↑   fraction
sign     binary point
```

Notice that this is the same as the pattern for 1.0, except the exponent is different. Adding 1 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:

```
 0|10 000 100|101 000 000 000 000 000 000 000 000
 ↑  exponent ↑   fraction
sign     binary point
```

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 2^2 or 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 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 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, 10.0/4 is 2.5.

---

[3]An exception to this is that 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.

As a final example, we shall try a hard one. We will convert 4/3 to a floating-point number. We begin by converting the numerator and denominator to binary, and doing long division:

```
      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

```
0|10 000 001|101 010 101 010 101 010 101 010 101
```

16.4.1.2. Double-Precision

In many applications, the single-precision format does not produce sufficiently accurate results. A second format of floating-point numbers is available for circumstances where extra precision is necessary. 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. The additional fraction bits do not materially affect the range of representable numbers, rather they extend the precision. Double-precision floating-point is available in the KI10 and later processors.

```
0 00000000 01111111111222222222233333 3 0 00000000011111111112222222222333333
0 12345678 90123456789012345678901234 5 0 12345678901234567890123456789012345
 ____________________________________________________________________________
|  |        |                                |  |                              |
|S| Exp     |          Fraction              |0|       Extended Fraction      |
|__|________|________________________________|__|______________________________|
```

16.4.2. 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 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
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
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 1 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 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 so 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 three bits of the result are now zero. 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].

16.4.2.1. 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.

16.4.2.2. 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.

If a number that is larger in magnitude than 2^{127} is computed, FOV will be set.

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.

16.4.3. 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.

```
                                          |  Result to AC
                          |R Rounded      |I Immediate.  Result to AC
                |AD ADd   |               |M Result to Memory
F Floating      |SB SuBtract |          /  |B Result to Both memory and AC
                |MP MultiPly |
                |DV Divide   |
                          |  no rounding  |  Result to AC
                                          |M Result to Memory
                                          |B Result to Both memory and AC

                      |AD ADd        |
DF Double Floating    |SB SuBtract   |  Memory Operand is C(E, E+1)
                      |MP MultiPly   |  Result to C(AC, AC+1)
                      |DV DiVide     |

FIX  Convert Floating-Point to Fixed-Point     |  No Rounding
                                               |R With Rounding

FLTR Convert Fixed-Point to Floating-Point with Rounding

FSC  Floating SCale
```

Table 16-1: Floating-Point Instruction Set

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.

The instructions that follow are used to convert between fixed and floating formats, and to scale floating-point numbers.

16.4.3.1. 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 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 0:8 of accumulator. Then perform an arithmetic shift 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.

```
FIX     C(AC) := fixed-point version of the floating number C(E)
```

16.4.3.2. FIXR -- Fix and Round

The FIXR instruction will convert a floating-point number to an integer by rounding. If the exponent field of the floating-point number in C(E) is greater than octal 243 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 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.

Rounding is always in the positive direction: 1.4 becomes 1, 1.5 becomes 2, -1.5 becomes -1, and -1.6 becomes -2.

```
FIXR    C(AC) := fixed, rounded version of the floating number C(E)
```

16.4.3.3. FLTR -- Float and Round

This instruction will convert an integer in C(E) to a floating-point number in AC. The data from C(E) is copied to AC where it is shifted right 8 places, extending the sign and retaining the bits that were shifted off the right end. The exponent 243 is inserted in bits 1:8; if the number is negative, the exponent field is set to the one's complement of 243,534.

The resulting number is normalized until bit 9 becomes significant; normalization may result in some or all of the bits that were right-shifted being brought back into the result. Finally, if any of the bits that were right-shifted still remain outside the accumulator then the result is rounded by looking at the bit to the right of the accumulator.

```
FLTR    C(AC) := floating, rounded version of the fixed number C(E)
```

16.4.3.4. FSC - Floating Scale

This 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 and FOV if the resulting exponent exceeds decimal 127. FXU (and AROV and FOV) 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     C(AC)[1:8] := C(AC)[1:8]+E; normalize result
```

16.5. EXERCISES

16.5.1. 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 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.

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:

```
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

[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.

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.

You should 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 used. 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 *dummy arguments* in parentheses. The assembler treats the dummy arguments, ACC and BYP, as placeholders for the text that will be supplied as *actual arguments* when the macro is called.

```
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 called, 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 call may be in parentheses, or not, at your option. The macro could be called 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 dummy argument called ACC; similarly, the actual argument string PTR-1(Y) has replaced each instance of the dummy argument BYP. This example shows how an arbitrary string can be used as an actual argument. Dummy arguments, however, must be identifiers.

17.2. CONDITIONAL ASSEMBLY

The assembler provides *conditional assembly* features to allow one source file to be turned into several different programs, for the situations where two programs differ only in minor characteristics. A general example is the difference between the TOPS-10 and TOPS-20 versions of some programs. TOPS-10 and TOPS-20 programs execute 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 IF Construct

The basic conditional assembly construction in MACRO is the `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 a conditional is

```
IFx exp,<
        text to be assembled if the expression "exp" bears
        relation "x" to zero
>
```

In this example, `IFx` should be replaced by one of the six `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 ">".

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 thrown out, you should be careful not to have any superfluous angle brackets lying around. In an unsatisfied conditional, the only processing that goes on is the counting of 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 IFNDEF Conditional

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 called `IFDEF`, meaning *IF DEFined.*)

In the program that follows, an `IFNDEF` conditional will be used to assign a default value to the symbol

called TOPS10. The symbol TOPS10, which defaults to 1, 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 called 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==0
```

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-20 portion of the following program appears without explanation. If you find it necessary to program for TOPS-20, you should refer to the DECSYSTEM-20 System Calls Manual for an explanation of the JSYS instructions 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-20 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. We take this opportunity to remind the reader that TOPS-10 and TOPS-20 run on the same (or quite similar) hardware; only the operating system calls and facilities distinguish between these two systems.

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.

```
        TITLE   NUM - NUMERIC EVALUATOR - Example 7

IFNDEF TOPS10,<TOPS10==1>       ;Default to TOPS-10 Assembly
DEFINE  T10     <IFN TOPS10>
DEFINE  T20     <IFE TOPS10>

T20,<   SEARCH  MONSYM          ;For TOPS-20, add JSYS definitions >;T20
```

The line that contains the IFNDEF conditional ensures that the symbol TOPS10 will have some value. By default, the value will be one, selecting the TOPS-10 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.

Essentially the same assembler program is used for TOPS-10 and for TOPS-20. MACRO contains the names and definitions of all the instructions and TOPS-10 system calls. When MACRO is used with a DECSYSTEM-20, it must be told to load the definitions for all the TOPS-20 system calls. This is accomplished in the TOPS-20 version of this program by means of the SEARCH pseudo-op.

The SEARCH pseudo-op, which we shall have occasion to use in our more complex examples, takes as its argument the name of a file. In this example, the name MONSYM really means the file SYS:MONSYM.UNV, a *universal file* containing definitions of the TOPS-20 monitor symbols. An analogous file, SYS:UUOSYM.UNV exists for TOPS-10, however, our example is still simple enough so that for the TOPS-10 program we can avoid using this file.

When MACRO sees the SEARCH pseudo-op, it reads the specified file. MACRO adds the definitions that it finds in the file to its internal symbol table. Thereafter, when a symbol is referenced, MACRO can find the appropriate definition. For example, the TOPS-20 version of this program makes use of the system call named HALTF. The definition of HALTF is added to MACRO by means of the SEARCH pseudo-op.

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-10 case, STOP labels an EXIT MUUO. For TOPS-20, the code at STOP is a HALTF JSYS that suspends 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 execution of the RDTTY JSYS. RDTTY obtains an entire line of input from the terminal; the line appears in the region beginning at BUFFER. The TOPS-10 version uses a version of the GETLIN subroutine that we have seen before.

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; the -1, which was placed in the left half of A by the HRROI, 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 31. 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; 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.

```
PDLIST: flags,,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:

---

[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.

```
PDLIST: flags,,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: flags,,PRINT+1
        3
        flags,,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: flags,,PRINT+1
        3
        flags,,DECOT1+4
        2
        flags,,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.

```
PDLIST: flags,,PRINT+1
        3
        flags,,DECOT1+4
        2
        flags,,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: flags,,PRINT+1
        3
        flags,,DECOT1+4
        2
        flags,,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: flags,,PRINT+1
        3
        flags,,DECOT1+4
        2                       <----- P (after the first RET)
        flags,,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: flags,,PRINT+1
        3                       <---- P (after second RET)
        flags,,DECOT1+4         (this has already been popped)
        2                       (This was popped and printed second)
        flags,,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. 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

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.

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 -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==1>       ;Default to TOPS-10 Assembly
DEFINE  T10     <IFN TOPS10>
DEFINE  T20     <IFE TOPS10>

T20,<   SEARCH  MONSYM          ;For TOPS-20, add 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

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

OP1:    0
X1:     0
BUFLEN==40
BUFFER: BLOCK   BUFLEN
PDLEN==40
PDLIST: BLOCK   PDLEN
CRLF:   BYTE    (7)15,12
PROMPT: ASCIZ   /Type an arithmetic expression: /
        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

$$\sin(X) = 3*\sin(X/3) - 4*[\sin(X/3)]^3$$

to compute values of the sine function.[3] Discuss this program. Give convincing evidence that this procedure is effective, i.e., that it actually works properly.

---

[3]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)
        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.[4]

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 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.

---

[4]Claims to the contrary notwithstanding, the Russians did not invent this method. It was used by Egyptian mathematicians in 1800 B.C.

- 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.

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.

When one of these LUUO instructions is executed, the CPU performs the following operations:

1. The effective address specified in the instruction is calculated.
2. A copy of the instruction and its effective address is stored in location 40 of the program.
3. The CPU executes the instruction in location 41.

The instruction image that is stored in location 40 has the same bits 0:12 as the LUUO instruction. Bits 13:17 are zero, and bits 18:35 reflect the results of the effective address calculation.

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.

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 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 he sees fit.

In the example that follows, a Local UUO handler is provided to make the calling sequences for the floating point input and output subroutines more palatable.

18.1. EXAMPLE 8 - 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 as though it were a complete program, it really should be thought of as two useful subroutines, plus a typical LUUO handler.

As we have done before, we shall present the entire program first and then subject it to analysis.

```
        TITLE   FLOATIO    Floating-point Input and Output
        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
;

        SUBTTL  Definitions of Operators
OPDEF   CALL    [PUSHJ  P,]
OPDEF   RET     [POPJ   P,]

DEFINE  UUONAM <
        UU      (ERROR,ERRUUO)          ;Error.  Print String and Stop
        UU      (FLIN,FLINP)            ;Floating Input
        UU      (FLOUT,FLOUTP)          ;Floating Output
>

DEFINE  UU(NAME,DISP)<
OPDEF   NAME    [UUONUM*1B8]
UUONUM==UUONUM+1
>

UUONUM==1                               ;Number of the first legal LUUO
        UUONAM                          ;Expand the operator definitions


;

        SUBTTL  Buffers, Stack and storage areas
PDLEN==200
PDLIST: BLOCK   PDLEN
BUFLEN==100
BUFFER: BLOCK   BUFLEN
OUTBUF: BLOCK   BUFLEN
UUACAD: 0
;
```

```
        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
        OUTSTR  [ASCIZ/
Floating-point I-O Example
/]
;

NEXT:   OUTSTR  [ASCIZ/Type a number, any number: /]
        CALL    GETLIN                  ;Get a line into the buffer area
        LDB     A,[POINT 7,BUFFER,6]    ;Read the first character
        JUMPE   A,DONE                  ;Exit if a blank line was typed.
        MOVE    B,[POINT 7,BUFFER]      ;a byte pointer to buffer
        FLIN    A,B                     ;Read a number.  Result to A.
        MOVE    B,[POINT 7,OUTBUF]      ;Pointer to the output buffer
        FLOUT   A,B                     ;convert internal to external
        MOVEI   A,15                    ;add CR LF NULL
        IDPB    A,B
        MOVEI   A,12
        IDPB    A,B
        MOVEI   A,0
        IDPB    A,B
        OUTSTR  [ASCIZ/You typed: /]    ;and print results.
        OUTSTR  OUTBUF
        JRST    NEXT

DONE:   OUTSTR  [ASCIZ/Done!
/]
        EXIT
;

        SUBTTL  Read a Line from the terminal

GETLIN: MOVE    B,[POINT 7,BUFFER]      ;read input to buffer area
GETCHR: INCHWL  A                       ;read a character
        CAIN    A,15                    ;is it a carriage return?
        JRST    GETCHR                  ;yes, discard CR
        CAIN    A,12                    ;is it a line feed?
        MOVEI   A,0                     ;yes, convert it to null to end line
        IDPB    A,B                     ;store the character in the buffer
        JUMPN   A,GETCHR                ;loop, unless end of line.
        RET
;
```

```
        SUBTTL  Handle Local UUOS

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 ac's on stack
        MOVE    0,-16(P)                ;just in case you wanted 0 preserved
        CALL    DOUUO                   ;perform the UUO function
        JRST    .+2                     ;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.  Set to zero: illegal.
        MOVE    D,UUOTAB(A)             ;Get the dispatch address
        JRST    (D)                     ;Return via POPJ or CPOPJ1
;


;Construct the dispatch table for the various UUOs.
;First, redefine the UU macro:
DEFINE  UU(NAME,DISP)<
0,,DISP
>

UUOTAB: 0,,ILLUUO                       ;0 is illegal.  See also DOUUO
        UUONAM                          ;Expand the dispatch table
UUOTLN==.-UUOTAB                        ;Define the length of UUOTAB

;

        SUBTTL  ERROR LUUO and Illegal UUOs

ILLUUO: ERROR   [ASCIZ/Illegal Local UUO detected
/]

ERRUUO: CLRBFI                          ;The Error UUO.  Clear typeahead.
        OUTSTR  [ASCIZ/?Error: /]       ;
        OUTSTR  (C)                     ;text of the message
        OUTSTR  [ASCIZ/
/]
        EXIT
;
```

```
        SUBTTL  FLINP - Floating-point Input Scan LUUO

;       FLIN    AC,BYP
;
;Convert characters to internal format floating-point numbers.  Call
;with a byte pointer to the input text string in the effective address.
;Returns the number in the specified accumulator.  the trailing
;delimiter can be found by a LDB that refers to the byte pointer.
;
;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.
;
;

;This is the handler for the FLIN LUUO.  Call with
;       B       Number of the Result AC
;       C       Address of the Byte Pointer
FLINP:  MOVE    W,C                     ;Get address of the byte pointer
        CAIG    W,17                    ;Was BYP in one of the ACs?
        ADD     W,UUACAD                ;Yes, Relocate to stacked copy of AC
        ADD     B,UUACAD                ;Relocate address of result AC
        PUSH    P,B                     ;Save address of Result AC
        CALL    FLINP0                  ;Obtain Result into B
        POP     P,A                     ;Address of Result AC
        MOVEM   B,(A)                   ;Put result in AC save Area
        RET

FLINP0: 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

;       FLOUT   AC,BYP
;
;Convert an internal format floating-point number to a character string.
;Call with a byte pointer to the output text string in the effective address
;and the floating-point number in the specified accumulator.
;
;      if the number, X, is zero, or if   0.1 <= Abs(X) < 10000.0 then
;      the number will be printed in the form of    sD.dddd
;      where s is the sign, either "-" or absent,
;            D is the minimum number of digits needed to print the
;              integer part of the number X, and
;            dddd is a four-digit fraction.
;
;      if the number X lies outside the range set forth above, it
;      will be printed as  sd.ddddEsD
;      where s is the sign of either the number or the exponent,
;              either "-" or absent;
;            d.dddd are five digits of the result;
;            E is an uppercase letter "e"; and
;            D is the minimum number of digits needed to print
;              the exponent.
;This is the handler for the FLOUT LUUO.  Call with
;       B       Number of the Result AC
;       C       Address of the Byte Pointer
FLOUTP: MOVE    W,C                     ;Get address of the byte pointer
        CAIG    W,17                    ;Was BYP in one of the ACs?
        ADD     W,UUACAD                ;Yes, Relocate to stacked copy of AC
        ADD     B,UUACAD                ;Relocate address of the data AC
        MOVE    A,(B)                   ;Get the data item.
        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
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.

        END     START
```

This program has three areas of special interest: the LUUO handler, the floating-point input scan, and the floating-point output conversion. In addition to these main points of interest, we have introduced another pseudo-op, SUBTTL and another MUUO, CLRBFI. These will all 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.

18.1.2. Local UUO Processing

This program shows how Local UUOs can be used. 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.

18.1.2.1. 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). 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.

18.1.2.2. Definitions of Local UUOs

When we define an LUUO, we need to provide three things:

1. An operation code number, in the range from 1 to 37 (octal), for each LUUO that we define.
2. A symbolic name for the function that is defined.
3. A code fragment (subroutine) that implements the new operation.

Some of these requirements can be met easily: we can define a symbolic name as a new operator via the OPDEF pseudo-op; we will let the first new operator be number 1, and we will assign sequential values as new operators are defined. The challenging parts are to write the necessary code to implement the operation, and to connect these three items together.

Let us attend to the problem of interconnecting the various parts. We find it convenient to use a macro that is somewhat analogous to the data structure definition macro that was used in the previous example. The relevant definitions have been extracted from the text of the program and are presented below.

```
DEFINE  UUONAM <
        UU      (ERROR,ERRUUO)          ;Error:  Print String and Stop
        UU      (FLIN,FLINP)            ;Floating Input
        UU      (FLOUT,FLOUTP)          ;Floating Output
>

DEFINE  UU(NAME,DISP)<
OPDEF   NAME    [UUONUM*1B8]
UUONUM==UUONUM+1
>

UUONUM==1                               ;Number of the first legal LUUO
        UUONAM                          ;Expand the operator definitions


;Construct the dispatch table for the various LUUOs.
;First, redefine the UU macro:

DEFINE  UU(NAME,DISP)<
0,,DISP
>

UUOTAB: 0,,ILLUUO                       ;0 is illegal.  See also DOUUO
        UUONAM                          ;Expand to make the dispatch table
UUOTLN==.-UUOTAB                        ;Define the length of the dispatch
```

The macro called UUONAM expands the UU macro for each LUUO that is defined. The call to the UU macro contains two arguments. The first is the symbolic name of the LUUO; the second is the symbolic name of the subroutine that will implement the LUUO.

Prior to the first expansion of UUONAM, the UU macro is defined to be an OPDEF that defines the symbolic name of the LUUO. The symbol UUONUM is set to 1, the operation code number of the first LUUO. The first expansion of UUONAM produces the following:

```
UUONUM==1                               ;Number of the first legal LUUO
        UU      (ERROR,ERRUUO)          ;Error:  Print String and Stop
        UU      (FLIN,FLINP)            ;Floating Input
        UU      (FLOUT,FLOUTP)          ;Floating Output
```

By virtue of the current definition of the UU macro, this in turn becomes

```
UUONUM==1                               ;Number of the first legal LUUO
OPDEF   ERROR   [UUONUM*1B8]
UUONUM==UUONUM+1
OPDEF   FLIN    [UUONUM*1B8]
UUONUM==UUONUM+1
OPDEF   FLOUT   [UUONUM*1B8]
UUONUM==UUONUM+1
```

UUONUM is initialized to 1. The name ERROR is defined to be an operator having the value 1*1B8. The notation 1B8 means the first value, 1, shifted so that the least significant bit occupies the bit specified by the second value. The second value, a bit number, is interpreted as decimal. The value 1B8 places the quantity 1 in bit 8 of a word; this is equivalent to the octal quantity 001000,,000000. The name ERROR is defined to be 1 times this quantity, or precisely 1 in the operation field of an instruction word.

After this definition, the symbol UUONUM is redefined by the assignment UUONUM==UUONUM+1. As in many other languages, this assignment changes the value of the symbol UUONUM by adding one to it. The resulting value, 2 is used in the next OPDEF where the name FLIN is defined to be 2*1B8, i.e., operation code number 2.

The binary patterns assigned to the names ERROR, FLIN and FLOUT do not correspond to hardware instructions. Instead, these patterns are executed as LUUOs.

The LUUO dispatch table is used inside the LUUO handler, as we shall discuss below. The dispatch table is created by a second expansion of the UUONAM macro. Prior to this expansion, the UU macro is redefined to expand only its second argument. Altogether, the result produced is

```
UUOTAB: 0,,ILLUUO                       ;0 is illegal.  See also DOUUO
        UU      (ERROR,ERRUUO)          ;Error:  Print String and Stop
        UU      (FLIN,FLINP)            ;Floating Input
        UU      (FLOUT,FLOUTP)          ;Floating Output
UUOTLN==.-UUOTAB                        ;Define the length of the dispatch
```

Under the influence of the new definition of UU, this becomes

```
UUOTAB: 0,,ILLUUO                       ;0 is illegal.  See also DOUUO
        0,,ERRUUO
        0,,FLINP
        0,,FLOUTP
UUOTLN==.-UUOTAB                        ;Define the length of the dispatch
```

18.1.2.3. 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.

18.1.2.4. 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 LUUO 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 instruction that follows, HRRZM P,UUACAD, 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 LUUO 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 ERRUUO. The execution of an ERROR LUUO causes the code at ERRUUO to be run. The ERROR LUUO is intended to print a message and stop the program when an error occurs.

```
ERRUUO: CLRBFI                          ;The Error UUO.  Clear typeahead.
        OUTSTR  [ASCIZ/?Error: /]       ;
        OUTSTR  (C)                     ;text of the message
        OUTSTR  [ASCIZ/
/]
        EXIT
```

This routine starts by executing a CLRBFI MUUO; this will clear the terminal input buffer. If the user been typing ahead of the program, that typing is now discarded: when an error occurs, it is a good idea to force the user to re-enter further commands.

The ERRUUO routine continues by prefixing the string "?ERROR:" to the program-supplied message. The question mark at the start of the string will alert such programs as the batch monitor to the occurrence of an error. The code at DOUUO set up register C to contain the effective address of the ERROR LUUO. This addresses a string that is supplied by the program. By means of indexed addressing, the address in C is used in the second OUTSTR MUUO to point to the program-supplied string. Finally, a carriage return and line feed are sent to the terminal and the programs stops itself by issuing an EXIT LUUO.

It might be argued that to rumble through all the code in UUOHAN and DOUUO for the purpose of performing only a few OUTSTRs 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. OUTSTR is expensive; in any event, it is relatively infrequent (in relation to computer speeds), since ERROR is not going to happen more than once during the execution of the program.

Examples of the use of UUACAD occur in FLINP and FLOUTP. We will study the code at FLINP that interfaces to the LUUO handler. Later, we will thoroughly investigate how this scanner works.

```
FLINP:  MOVE    W,C             ;Get address of the byte pointer
        CAIG    W,17            ;Was BYP in one of the ACs?
        ADD     W,UUACAD        ;Yes, Relocate to stacked copy of AC
        ADD     B,UUACAD        ;Relocate address of result AC
        PUSH    P,B             ;Save address of Result AC
        CALL    FLINP0          ;Obtain Result into B
        POP     P,A             ;Address of Result AC
        MOVEM   B,(A)           ;Put result in AC save Area
        RET
```

FLINP is called from DOUUO. When it is called, register C contains a copy of the effective address of the instruction. That address is copied to register W. The word at the effective address is supposed to be a byte pointer that can be used for reading characters. If the effective address is greater than octal 17, then the byte pointer resides in a memory cell. Register W will address that byte pointer. An instruction such as ILDB A,(W) would refer to that byte pointer.

Consider, however, the case where the byte pointer is in one of the accumulators. This routine must use the specified byte pointer, and it must update that byte pointer to indicate which characters have been processed. Two problems must be solved: first, since the user may select any accumulator, irrespective of what accumulators this routine uses, the byte pointer must reside somewhere other than in one of the ACs

while this LUUO is executing. Second, the updated byte pointer must be copied back to the same AC that it came from. After this explanation of the difficulties, the solution may appear to be quite simple. The reason it is simple, is that we have anticipated the problem and we have already done most of the work necessary to solve it.

If the byte pointer argument is in an accumulator, register W will contain the number of that AC; it will be smaller than octal 20. All of the ACs have been copied onto the stack. The address where register 0 is stored is held in the word called UUACAD. By adding the contents of UUACAD to the accumulator number in W, we obtain the address of the stacked copy of the specified AC. When an instruction such as ILDB A,(W) is executed, the stacked copy of the accumulator will be referred to; the copy will be updated. The updated byte pointer is copied back into the specified accumulator automatically when the UUO handler restores the stacked copy of the ACs to the real accumulators.

The FLIN LUUO is specified as returning the result in the accumulator specified in its AC field. This is accomplished as follows: when FLINP is called, B contains a copy of the AC field of the FLIN LUUO. To this accumulator number we add the address contained within UUACAD. This sum represents the address of the stacked copy of the specified accumulator. This address is saved. The routine FLINPO is called to perform the floating-point input scan. FLINPO returns its results in register B. The address of the stacked copy of the target accumulator is popped into A; the instruction MOVE B,(A) stores the result in the stacked copy of the accumulator. When the accumulators are restored from the stack, this result will appear in the real accumulator.

18.1.3. FLINPO - Floating-Point Input Scan

The FLINPO routine is an extension of the DECIN routine that we examined in example 7. The essential difference is that FLINPO 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 FLINPO. Besides the change to floating-point instructions, FLINPO has several additional problems to worry about. When scanning a real number, decimal points and exponents have to be dealt with.

18.1.3.1. 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.

18.1.3.2. Processing the Exponent

The FLINPO 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.

18.1.4. FLOUTP - Floating-Point Output Processing

FLOUTP is the handler for the FLOUT LUUO. The FLOUT LUUO converts the floating point number in the specified accumulator into a character string which is stored using the byte pointer specified in the effective address of the instruction.

Much of the set-up needed for FLINP is repeated here. One difference is that the argument is present in an accumulator, instead of a result being returned in one. The accumulator argument is copied from the stacked copy of the accumulators to register A. Next, 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 ≤ C(A) < 10000.0 and zero are handled at FLOUTN.

18.1.4.1. 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]

18.1.4.2. 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.

---

[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.

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.

18.1.4.3. 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.

18.2. EXERCISES

18.2.1. 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 special 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 decimal 36?
- 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?

Can you make your program 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.

18.2.2. 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.

Chapter 19
Operating System Facilities

At this point our discussion of the hardware instruction set is essentially complete. There are 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].

19.1. INPUT/OUTPUT

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-10 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-10 a file name such as `DSKA:MONITR.EXE[1,4]` 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. 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 *Software Input/Output Channel*, or more simply just *channel* is such a shorthand notation.

TOPS-10 implements sixteen input/output channels. These are numbered from `0` to `17` (octal). In many of the input/output MUUOs, the channel number will appear instead of an accumulator field. A channel is a software entity that corresponds to data structures inside of TOPS-10. These data structures allow TOPS-10 to remember the status and condition of your program's input and output activities.

In this section, we intend to present a brief (and incomplete) overview of input and output programming. We'll get to the details soon enough.

The `RESET` MUUO that we have used in every program forces all channels to an idle state. In order to

begin to use a channel the OPEN MUUO (or the INIT MUUO) can be exected. OPEN will establish the correspondence between a channel and a *device*. A device is an input or output unit, e.g., a line printer, magnetic tape, or disk drive.

When the selected device is a *directory device*, i.e., one which contains named files, such as the disk or dectapes, then the program must specify the name of the file to read or write. For input, the LOOKUP MUUO is used to select a file. For output, the ENTER MUUO specifies the name of the file being written.

The IN or INPUT MUUOs can be used to read data from the device into memory. The OUT or OUTPUT MUUOs write data from memory to the device.

When the program is finished with the file, it breaks the correspondence between the channel and the file by means of the CLOSE MUUO. When the program is finished with the device itself, the channel is returned to idleness by the RELEAS MUUO.

The *mode* is another attribute of an input/output channel. There are two major divisions of mode: *buffered* and *unbuffered*; the term *dump mode* is often used to refer to an unbuffered mode. In a dump or unbuffered input/output mode, the program supplies explicit commands to TOPS-10 that describe where in memory the data is to be put (or where it is to come from). In a buffered mode, TOPS-10 reserves a portion of your program's memory space for buffers. Each call to TOPS-10 for more input will result in the program being told the location and size of another buffer.

19.2. OTHER OPERATING SYSTEM FEATURES

A variety of other facilities exist in the TOPS-10 operating system. We will briefly mention some of these now, and defer our detailed discussion of these to later sections.

19.2.1. Memory Usage Control

TOPS-10 allows the program to determine its memory requirements and memory usage strategy dynamically. Also, TOPS-10 permits a program to divide itself into two segments, one of which can be shared by several jobs.

19.2.2. Information about the Environment

TOPS-10 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.

19.2.3. Interrupts and Traps

The software interrupt system provides a means for a program 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 interrupt system. However, the trap system can only detect and respond to arithmetic 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.

19.2.4. Interprocess Communication

The *InterProcess Communication Facility* (IPCF) allows messages to be sent between cooperating processes. In connection with the interrupt system, IPCF provides a useful means for processes to communicate requests and responses.

Chapter 20
File Output

We offer two programs to exhibit file output. The first is kept as simple as possible. In the second, we exhibit some further techniques.

20.1. 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

A=1                                     ;Define symbolic names for ACs
B=2
C=3
D=4
P=17

PDLEN==100

OUT==7                                  ;Channel number for output

OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ  P,]

PDLIST: BLOCK   PDLEN
OBUFF:  BLOCK   3

START:  RESET                           ;Reset any i/o activity
        MOVE    P,[IOWD PDLEN,PDLIST]   ;Establish a stack
;Obtain an IO channel.  Connect it to device DSK.  Output in ASCII Mode
        OPEN    OUT,[0                  ;Open a channel for DSK, ASCII mode
                SIXBIT/DSK/             ;Device is logical disk
                OBUFF,,0]               ;Output Buffer header.  No input BH
        JRST    NODISK                  ;Jump if we can't open this channel
;
```

```
;Select an Output File Name
        MOVE    A,['SAMPLE']            ;Make a 4-word block for ENTER.
        MOVSI   B,'OUT'                 ;First the name, then the extension
        SETZB   C,D                     ;let protection and PPN default
        ENTER   OUT,A                   ;Select SAMPLE.OUT as the output file
        JRST    NOENTR                  ;In case the ENTER fails.

;now send some output to the file
        MOVE    B,[POINT 7,[ASCIZ/This is sample data to be written
to the file
/]]
        CALL    PUTSTR                  ;Send data, stop at null byte
;

;Close the file and release the channel
        CLOSE   OUT,                    ;Close the channel
        STATZ   OUT,740000              ;test for errors
        JRST    IOERR                   ;jump if an error occurs
        RELEAS  OUT,
        OUTSTR  [ASCIZ/Done
/]
        EXIT

PUTSTR: ILDB    A,B                     ;Get a byte from the string
        JUMPE   A,CPOPJ                 ;A null byte signals the end
        CALL    PUTCHR                  ;send this character
        JRST    PUTSTR                  ;loop to process the string

PUTCHR: SOSLE   OBUFF+2                 ;decrement buffer byte count
        JRST    PUTCH1                  ;room remains in the buffer
        OUTPUT  OUT,                    ;Send the buffer.
        STATZ   OUT,740000              ;test for errors
        JRST    IOERR                   ;report error
PUTCH1: IDPB    A,OBUFF+1               ;add character to buffer
CPOPJ:  RET                             ;return

;

;here in case of an error
IOERR:  OUTSTR  [ASCIZ/File output error
/]                                      ;send error typeout
        EXIT

NOENTR: OUTSTR  [ASCIZ/Could not ENTER the output file
/]                                      ;send error typeout
        EXIT

NODISK:  OUTSTR  [ASCIZ/Could not open the output channel
/]                                      ;send error typeout
        EXIT

        END     START
```

Most of this program is concerned with putting the output into a file, in contrast to printing it on the terminal.

20.1.1. The OPEN MUUO

As we briefly mentioned in the previous chapter, the OPEN MUUO serves to establish a connection between a software entity called an *input/output channel* and an input/output device. OPEN actually performs additional functions as well. Among the additional functions are the initialization of the data mode

and the device status, and the attachment of *buffer headers* to the channel. We will detail the meaning of these later in this discussion.

The following program fragment performs the OPEN MUUO:

```
;Obtain an IO channel.  Connect it to device DSK.  Output in ASCII Mode
        OPEN    OUT,[0                  ;Open a channel for DSK, ASCII mode
                SIXBIT/DSK/             ;Device is logical disk
                OBUFF,,0]               ;Output Buffer header.  No input BH
        JRST    NODISK                  ;Jump if we can't open this channel
```

In the OPEN MUUO, the accumulator field represents an input/output channel number. In this example, we use the symbolic name OUT to represent channel number 7. The effective address of the OPEN MUUO points to a three word block that should contain the following items:

- The first word contains special flags in the left halfword and the input/output mode in the right half. In this example, the flags are zero; the mode is zero also. Mode zero is a buffered mode for 7-bit ASCII characters.
- The second word contains the device name. In TOPS-10, device names and file names are represented using the six-bit subset of the ASCII code. We described this notation when we discussed the shift instructions. There are two notations in MACRO that help us deal with this character set. The first is the SIXBIT pseudo-operator, which is analogous to the ASCII and ASCIZ pseudo-ops. In this example, we call for the device name DSK which is the generic name for any disk storage unit. The second notation by which MACRO allows us to represent characters in the six-bit ASCII subset is by the use of single quote (apostrophe) marks. A string of letters inside single quotes is interpreted as right-justified sixbit. Thus, 'DSK ' (the spaces are important) is equivalent to what we have used in this example.
- The third word contains the address of the output *buffer header* in the left half word, and the address of the input buffer header in the right half word. Since this program is not doing input, the right half of this word is set to zero. The buffer header is a three- or four-word block that describes the condition, size, and location of the currently active buffer. We'll have more to say about the buffer header somewhat later.

When OPEN is executed without error, it skips the instruction that immediately follows the OPEN MUUO. When an error occurs, OPEN does not skip. The most typical error from OPEN is that the requested device does not exist or it cannot be opened by this program because it is already in use elsewhere. The device DSK is a sharable resource; we do not consider that it's likely for this OPEN to fail. Nevertheless, we provide a small routine to handle this error if it should occur.

20.1.2. ENTER MUUO

The ENTER MUUO is used to specify the name of an output file. An ENTER is mandatory whenever output is done to a *directory device*, i.e., a to a device where names are attached to each file. The disk is the most commonly used device. As a user of TOPS-10, you are aware that files on the disk have names by which we refer to them.

The following fragment performs the ENTER that selects the name SAMPLE.OUT for the output file:

```
;Select an Output File Name
        MOVE    A,['SAMPLE']            ;Make a 4-word block for ENTER.
        MOVSI   B,'OUT'                 ;First the name, then the extension
        SETZB   C,D                     ;let protection and PPN default
        ENTER   OUT,A                   ;Select SAMPLE.OUT as the output file
        JRST    NOENTR                  ;In case the ENTER fails.
```

The ENTER MUUO selects an output file to be associated with a given output channel. As is usual with input/output MUUOs, the channel number appears in the accumulator field of the ENTER. The effective address of the ENTER points to a four-word block that contains data representing the desired file name. In the most usual case, the four contain the following items:

- The first word contains the file name. File names are represented in the six-bit subset of ASCII. File names typically are left-justified in this field. Anything but all zeros (six blanks in the six-bit notation) is legal here; but most programs are unable to handle anything but left-justfied alpha-numerics in file names.
- The left half of the second word contains the file extension (also called the file type). This is an additional zero- to three-letter name that is associated with the file. In this example, we will put zero in the right half of the second word: the system will supply one date and part of a second date as the create date and write date of the file.
- In the third word we are allowed to set the file protection, mode, part of a date and the file write time. We will store a zero in this word to allow TOPS-10 to supply appropriate default values.
- The fourth word contains the project number in the left half, and the programmer number in the right half. When we supply a zero in this word, we refer to ourselves, i.e., to the project and programmer number associated with the currently running job.

TOPS-10 modifies the argument block that we supply: it fills in some of the fields that we omitted. Because the argument block changes, we cannot use a literal in the effective address of an ENTER. Instead, we copy data into a scratch area, and tell the ENTER to read its arguments from that temporary copy. In this program, the temporary area is composed of accumulators 1 through 4. We use accumulators because they are easy to set up.

20.1.3. Buffer Rings and the OUTPUT MUUO

When the OPEN MUUO is performed, TOPS-10 is told to use the three word block at OBUFF as the output buffer header. In the process of doing OPEN, TOPS-10 stores a zero in OBUFF+2 to signify that there is no room for any more bytes in the output buffer. Because we specified data mode zero, buffered 7-bit bytes, TOPS-10 stores the byte size, 000700,,0, in OBUFF+1.

The PUTCHR routine is our single low-level mechanism for output to the file. PUTCHR is supposed to send the byte that is in register A to the file:

```
PUTCHR: SOSLE   OBUFF+2                 ;decrement buffer byte count
        JRST    PUTCH1                  ;room remains in the buffer
        OUTPUT  OUT,                    ;Send the buffer.
        STATZ   OUT,740000              ;test for errors
        JRST    IOERR                   ;report error
PUTCH1: IDPB    A,OBUFF+1               ;add character to buffer
CPOPJ:  RET                             ;return
```

Let us now examine how PUTCHR works. The very first time we try to output a byte at PUTCHR the byte count in OBUFF+2 is zero, as set by the OPEN MUUO. The SOSLE instruction decrements this zero and

skips. The next instruction executed is the OUTPUT MUUO at PUTCHR+2. This is the very first time that an OUTPUT has been executed for this channel.

When this OUTPUT is executed, TOPS-10 examines the buffer header words and determines that we have not yet provided any buffer areas for it to use. Therefore, TOPS-10 graciously supplies buffers. The structure of a buffer ring with two buffers is depicted in figure 20-1.

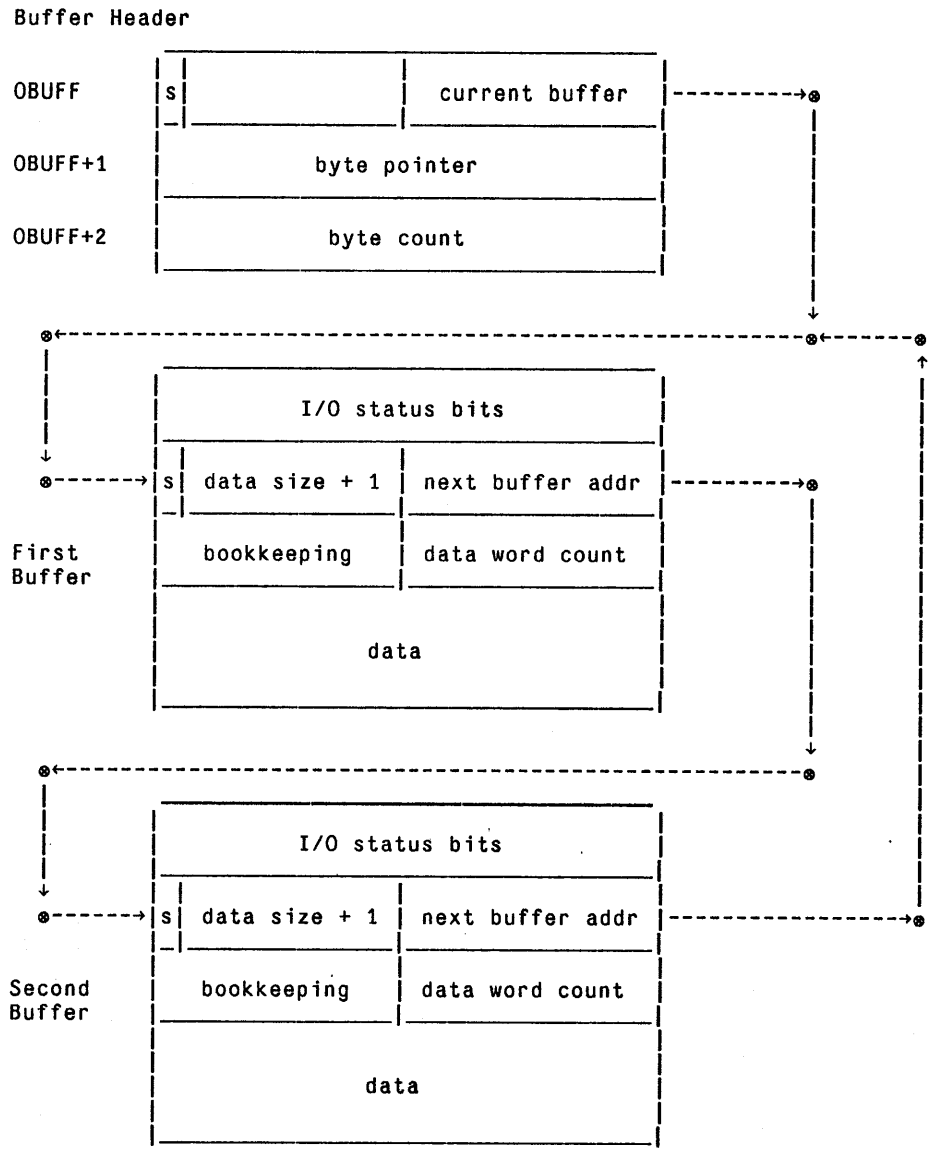


Figure 20-1: A Two-Buffer Buffer Ring

Again, the very first time the PUTCHR routine is called, it will perform an OUTPUT MUUO. This first OUTPUT will cause TOPS-10 to create a buffer ring for this input/output channel. As a result of the OUTPUT MUUO, TOPS-10 will store the address of the first buffer in OBUFF; it will store a byte pointer in OBUFF+1, which will point to the non-existent byte immediately before the buffer's data area; finally, TOPS-10 will store the byte count in OBUFF+2. The byte count is precisely the number of bytes (of the given byte size) that will fit in the buffer.

The code at PUTCHR continues: the byte in register A is stored in the output buffer by means of the byte pointer in OBUFF+1; then PUTCHR returns to its caller.

Note that when PUTCHR returns, it has stored one byte in buffer but it has not yet decreased the byte count. Thus, outside this routine, the byte count in OBUFF+2 always appears to be one greater than the number of bytes that will fit in the buffer. The PUTCHR routine gets around this problem by decrementing the count and testing for zero (by SOSLE instead of the SOSL that you might otherwise expect) as the first instruction that it performs. As long as you use this PUTCHR routine exactly as written, and have no OUTPUT MUUOs in your program except within a routine of this kind, you can ignore this peculiar behavior. If, however, you need to do fancy things, such as change the byte size or force output at certain places, you should be aware of this off-by-one characteristic, and guard against the problems it raises.

This program is a very simple one. Essentially, it does nothing but one call to PUTSTR, which in turn calls PUTCHR for each character in the string that it processes. When enough characters accumulate in the buffer, the PUTCHR routine will perform another OUTPUT MUUO. Each OUTPUT after the very first one, will transmit data from the program's memory space to the selected output device. When TOPS-10 returns control to your program after an OUTPUT, the data items in the buffer header will be changed to point to the next buffer in the ring. If your program manages to fill the second buffer before the output device is finished emptying the first buffer, TOPS-10 makes your program wait until the device has finished with the first buffer.

The status of the output device can be monitored by means of the STATZ MUUO. This system call takes a channel number in the accumulator field, and status bits in the address field, as immediate operands. The STATZ MUUO will skip if all the status bits in the effective address correspond to input/output status bits that are zero in the status word corresponding to this channel. In this case, we are testing for the four most common error indicators. If none of them are one, this program proceeds. If any of the error indicators is a one, then an error has occured; the program attempts to issue some error message before stopping.

There are two MUUOs related to STATZ: STATO which skips if any selected status bit is a one; and GETSTS which reads the status word into the memory word specified by its effective address.

20.1.4. The CLOSE and RELEAS MUUOs

The alert reader will already have noticed a problem with the mechanism for buffered output. In general, a buffer is sent to the device only when more characters need to be added to an already full buffer. So, the problem is how do we send the last, partially-full buffer? The answer is easy: TOPS-10 provides the CLOSE MUUO which includes this function. Our program must execute the CLOSE MUUO to finish doing output. CLOSE cause all the buffers to be transmitted to the device and an End-of-File indication will be written.

The RELEAS MUUO breaks the connection between the device and the channel. If you were using a non-sharable device, such as the magnetic tape, it would be kept unavailable to other jobs until you execute a RELEAS MUUO.

20.1.5. Where are the Buffers?

We have explained that the buffers appear as a result of the first OUTPUT MUUO that the program performs. We haven't yet said where the buffers are placed. One of the words in the job data area is .JBFF, the job's *first free* location.

When the loader loads your program, DDT, symbols, and library subroutines into memory, it sets the right half of location .JBSA to the starting address that you designate in the END pseudo-op. The loader also stores the address of the next word available to to your program in the left half of this word. When the loader computes this quantity, it takes into account all of your code, literals, symbols, subroutines, etc. that may be loaded with your program. The value in the left side of .JBSA is the first location available after everything has been loaded.

The RESET MUUO copies the value from the left half of .JBSA to the right half of .JBFF. When TOPS-10 builds buffers, it locates them at the location specified in .JBFF and it updates .JBFF to indicate the amount of space that it consumed.

In some circumstances, either the default number of buffers or the default location of buffers is not acceptable to the programmer. To force extra buffers to be allocated, use the OUTBUF MUUO. (For input buffers use INBUF.) The OUTBUF MUUO requires a channel number in the accumulator field and the number of buffers is the immediate value given by the effective address. The specified number of buffers will be built immediately.

If the default placement of buffers is not acceptable, the following fragment changes .JBFF temporarily to tell TOPS-10 where to put the buffers:

```
MOVEI   A,BUFADR            ;This is where we want the buffers
EXCH    A,.JBFF             ;Put this address in .JBFF,
OUTBUF  OCHAN,10            ;Build 8 buffers
MOVEM   A,.JBFF             ;restore former value of .JBFF
```

20.2. 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.

```
        TITLE   Long-precision Decimal Output & File Output.  Example 10

A=1
B=2
C=3
D=4
P=17

OUT==0                              ;Symbolic name for a channel

OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ P,]

PDLEN==100
```

```
START:  RESET
        MOVE    P,[IOWD PDLEN,PDLIST]
        CALL    OPNOUT                  ;Open the Output File
        MOVE    B,[POINT 7,[ASCIZ/Sample Results file

/]]
        CALL    PUTSTR                  ;copy string to output.

;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    PUTSTR                  ;Add spaces to output file
        MOVEI   B,1                     ;Compute next power of two
        MOVEI   A,0
        ASHC    A,@COUNT                ;double shift: 70-bit result
        CALL    DECPBG                  ;Print BIG number
        MOVE    B,[POINT 7,CRLF]
        CALL    PUTSTR                  ;copy CRLF to output
        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    PUTSTR                  ;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    PUTSTR

;Finished with computations.  Close the file
DONE:   CLOSE   OUT,                    ;Close the file, force buffers out.
        STATZ   OUT,740000              ;Test for errors
        JRST    IOERR
        RELEAS  OUT,                    ;release the output channel
        EXIT                            ;all finished


;Subroutine to obtain an IO channel, connect it to device DSK in ASCII Mode
OPNOUT: OPEN    OUT,[0                  ;Open a channel for DSK, ASCII mode
                SIXBIT/DSK/             ;Device is logical disk
                OBUFF,,0]               ;Output Buffer header.  No input BH
        JRST    NODISK                  ;Jump if we can't open this channel
;

;Select an Output File Name
        MOVE    A,['SAMPLE']            ;Make a 4-word block for ENTER.
        MOVSI   B,'OUT'                 ;First the name, then the extension
        SETZB   C,D                     ;let protection and PPN default
        ENTER   OUT,A                   ;Select SAMPLE.OUT as the output file
        JRST    NOENTR                  ;In case the ENTER fails.
        RET

;Copy a string pointed to by B to the output buffer
PUTSTR: ILDB    A,B                     ;copy string from B to
        JUMPE   A,CPOPJ                 ;the output buffer.  End at null
        CALL    PUTCHR                  ;send character to output
        JRST    PUTSTR
```

```
;Transmit the output buffer to the file
PUTCHR: SOSLE   OBUFF+2                 ;decrement buffer byte count
        JRST    PUTCH1                  ;room remains in the buffer
        OUTPUT  OUT,                    ;Send the buffer.
        STATZ   OUT,740000              ;test for errors
        JRST    IOERR                   ;report error
PUTCH1: IDPB    A,OBUFF+1               ;add character to buffer
        RET


;Decimal output with fill
DECZFL: SKIPA   B,["0"]                 ;fill character is zero
DECFIL: MOVEI   B," "                   ;fill character is space
        MOVEM   B,FILLCH
DECFL1: IDIVI   A,12                    ;this is fairly standard
        PUSH    P,B                     ;save a remainder
        SUBI    C,1
        JUMPE   A,DECFL3
        CALL    DECFL1
DECFL2: POP     P,A                     ;retrieve a saved remainder
        ADDI    A,"0"
        IDPB    A,OUTPNT
CPOPJ:  RET

DECFL3: JUMPLE  C,DECFL2                ;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.
        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"
        IDIV    A,[^D10000000000]       ;break up the high part "s" "t"
        DIV     B,[^D10000000000]       ;break up the low parts "u" "v"
        PUSH    P,C                     ;Save lowest part "v"
        JUMPE   A,DECPB2                ;less work for no very big stuff
        DIV     A,[^D10000000000]       ;handle the big parts "s" "u"
        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                ;If high part is zero, nothing
        MOVEI   C,16                    ;14 spaces
        CALL    DECFIL                  ;decimal print.  fill space.
        JRST    DECPB1                  ;write low part ("v") in 10 columns.

DECPB3: POP     P,A                     ;write the low part in 24 columns
        MOVEI   C,30                    ;blank fill.
        JRST    DECFIL
```

```
;here in case of an error
IOERR:  OUTSTR  [ASCIZ/File output error
/]                                      ;send error typeout
        EXIT

NOENTR: OUTSTR  [ASCIZ/Could not ENTER the output file
/]                                      ;send error typeout
        EXIT

NODISK:  OUTSTR  [ASCIZ/Could not open the output channel
/]                                      ;send error typeout
        EXIT

CRLF:   BYTE(7)15,12
PDLIST: BLOCK   PDLEN
COUNT:  0                               ;index variable, 0 to 69
FILLCH: 0                               ;fill character for DECFIL
OBUFF:  BLOCK   3

        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:
       118059162071741130342 3
```

This program has two items of interest to us. One is a double-precision integer printout routine. The second is another application 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.

20.2.1. Mathematical Basis of the DECPBG Routine

Suppose we have a very large 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 $a*2^{35}+b$.

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.[1] Since the a part can range up to 2^{35}, we must choose a very large divisor. Suppose we choose 10^{10} as the divisor; 10^{10} is the largest power of

[1] Yes, we could have used the DDIV instruction to avoid this problem. But then, suppose you wanted to print 140-bit numbers.

10 less than 2^{35}. If we divide a*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*2^{35}+b by 10^{10}.

To solve this problem, we resort to some tricks by which we rewrite the quantity a*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*10^{10}+t. Then, the original number can be written as

$$a*2^{35}+b = (s*10^{10}+t)*2^{35}+b = s*2^{35}*10^{10}+t*2^{35}+b$$

The number t*2^{35}+b can be rewritten as u*10^{10}+v; there is no problem using DIV to divide t*2^{35}+b by 10^{10}; we know that t is less than 10^{10} because it is the remainder resulting from having divided a by 10^{10}.

So the entire number, a*2^{35}+b can be written as

$$s*2^{35}*10^{10}+u*10^{10}+v = (s*2^{35}+u)*10^{10}+v$$

where

s = the quotient of (a/10^{10});	s < 10, since a < 2^{35}
t = remainder of (a/10^{10});	t < 10^{10}
u = quotient of ((t*2^{35}+b)/10^{10});	u < 2^{35}
v = remainder of ((t*2^{35}+b)/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*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*2^{35}+u is rewritten as w*10^{10}+x. Therefore the original number has been decomposed into

$$a*2^{35}+b = w*10^{20} + x*10^{10} + v,$$

where, in conjunction with the definitions of s, t, u and v above, we define

w = the quotient of ((s*2^{35}+u)/10^{10})
x = the remainder of ((s*2^{35}+u)/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.

20.2.2. File Output

The entire mechanism for file output that was present in example 9 has been transplanted to this program. It should be noted that an optimization of the PUTSTR subroutine is possible. It removes one instruction from the loop:

```
PUTST1: CALL    PUTCHR                  ;send a byte to the file
PUTSTR: ILDB    A,B                     ;get a byte
        JUMPN   A,PUTST1                ;loop unless null
        RET                             ;return
```

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 section we will discuss two-dimensional and multi-dimensional arrays.

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 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 23.6, page 300) 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

```
length of the array  =  highest index - 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*(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. This 1 is the value of 0!. FACT and NDIG will both be set to 0.

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 1. 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 157. 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 (a 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 0 held in ACCUM+0 to make it 1; this makes the ACCUM array contain 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. 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.[1]

---

[1] Another approach is to define symbols such as ZBEG and ZEND which are the boundaries of the space that is zeroed at startup.

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 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 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	0	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	0	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.

In the preceding explanation we have attributed too much judgment to the computer. At times we said that it noticed that results either would or would not fit into a word of ACCUM. The program is not smart enough to "notice" anything. As with the other programs we write, this one performs its chores by rote. In every case it 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.

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.

```
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 now 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 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

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


A=1
B=2
C=3
D=4
W=5
X=6
Y=7
Z=10
P=17

OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ P,]

PDLEN==100

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

START:  RESET                           ;initialize I/O
        MOVE    P,[IOWD PDLEN,PDLIST]   ;initial stack
        OUTSTR  [ASCIZ/Factorials up to 100
/]                                      ;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:   OUTSTR  [ASCIZ/

Done!
/]
        EXIT                            ;Stop here.
```

```
        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
        OUTSTR  OBUFR                   ;send output line
        RET

        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  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. Exercises

These exercises are intended to provide the student with an opportunity to experiment with one-dimensional arrays.

21.1.2.1. 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.

21.1.2.2. Compute *e*, the Base of Natural Logarithms

To compute an approximation to *e*, the base of natural logarithms, evaluate the first 101 terms of the series:

$$\sum_{k=0}^{\infty} \frac{1}{k!}$$

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.

The best way to compute 1/k! is to divide 1/(k-1)! by k. Assuming that at some point you have

computed the fraction 1/(k-1)!, then to compute 1/k! divide this previously obtained value by k. Use long division, as you were taught in grammar school.

Also, remember that 0! is defined to be 1.

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! as follows:

```
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 the example 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 x 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 x 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].

```
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.[2]

We shall see what difference these changes accomplish below.

---

[2]Some minor changes are needed if extended addressing is used.

```
QX:     Q1-1(Y)         ;Y,,address of Q[1,0].  At QX+0
        Q2-1(Y)         ;Y,,address of Q[2,0].  At QX+1
        Q3-1(Y)         ;Y,,address of Q[3,0].  At QX+2
        Q4-1(Y)         ;Y,,address of Q[4,0].  At QX+3
        Q5-1(Y)         ;Y,,address of Q[5,0].  At QX+4
        Q6-1(Y)         ;Y,,address of Q[6,0].  At QX+5
        Q7-1(Y)         ;Y,,address of Q[7,0].  At QX+6
        Q8-1(Y)         ;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]. Q[5,3] is transformed 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 word at QX+4 is fetched, and the effective address calculation is continued, using the address fields of that word.

The 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.

Before leaving the subject of side-tables, we should mention that we can make the MACRO assembler 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 rows of 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
        Q+IIII-1(Y)             ;analog of 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 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

```
Q0 + (i-1)*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:

```
K ≤ i ≤ L   and   M ≤ j ≤ N
```

Let S0 be the address of S[K,M]. Then the address of S[i,j] is given by the expression

```
S0 + (i-K)*(N-M+1) + (j-M)
```

The expressions (i-K) and (j-M) represent *normalized* or *effective* subscripts. By subtracting the lower row bound from i we have made a number (i-K) that ranges from 0 to (L-K); the normalized row number is multiplied by the number of words per column (the expression N-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-K)*(N-M+1) + (j-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 254.

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. Both of these expressions should be computed when the array is allocated. Then the polynomial to address S[i,j] becomes

$$S_{00} + i*L_C + j$$

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
IMUL    1,LC            ;row index times the number of columns
MOVE    2,J             ;column index
CAML    2,M             ;bounds check
CAMLE   2,N
CALL    AIDXER          ;an indexing error
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. Plot Program, Example 12

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 normalized (-1.0 to +1.0)
                        ; 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`.[3] 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

---

[3] 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.

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.

```
;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. 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.

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 * 3.14159265 * I / STABLN) for values of I in the range $0 \le I < STABLN$. 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 thru 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 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 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-10 all Fortran functions and subroutines have a standardized way to pass arguments and results.[4] 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 argument.

The argument list for this program appears below:

```
        -1,,0
ARGLST: 4,SINARG
```

The word containing -1,,0 has the negative of the number of arguments in the left half.

---

[4]For further information, consult the appendices to the DECsystem-10 Fortran Reference Manual.

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+π/2)
```

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(π/2)        cos(0)          COSTAB + 0
                          ....
SINTAB + <STABLN/2>       sin(π)          cos(π/2)
                          ....
SINTAB + 3*<STABLN/2>     sin(3π/2)       cos(π)
                          ....
SINTAB + STABLN-1         sin(2π-ε)       cos(3π/2-ε)
```

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(π/2)        cos(0)          COSTAB + 0
                          ....
SINTAB + <STABLN/2>       sin(π)          cos(π/2)
                          ....
SINTAB + 3*<STABLN/2>     sin(3π/2)       cos(π)
                          ....
SINTAB + STABLN           sin(0)          cos(3π/2)
                          ....
SINTAB + 5*STABLN/4 -1    sin(π/2-ε)      cos(2π-ε)
```

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 PUTCHR subroutine that we have used in the previous instances of buffered output.

```
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
```

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

        .REQUEST SYS:FORLIB             ;Ask LINK to look in SYS:FORLIB.REL
        EXTERN SIN                      ;for the Fortran SIN routine

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


OUT==0                                  ;Channel for disk output

OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ P,]

PDLEN==100
;

CRLF:   BYTE(7)15,12
PDLIST: BLOCK   PDLEN
OBUFF:  BLOCK   3                       ;buffer header for output buffer

;argument list for the Fortran SIN routine
        -1,,0                           ;-number of arguments,,0
ARGLST: 4,SINARG                        ;Real (4 as the AC), address of argument

SINARG: 0                               ;one word to hold the argument to SIN
```

```
;Define the Sine/Cosine Table
LOGSTL==11                              ;LOG (base 2) of the Sin Table's Length
STABLN==1_LOGSTL                        ;Sine TABle LeNgth.
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


;Main program
START:  RESET                           ;Always start this way
        MOVE    P,[IOWD PDLEN,PDLIST]   ;Initialize a stack
        CALL    INIT                    ;Build the sine/cosine tables
        CALL    SETFILE                 ;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    CLSFILE                 ;close the output file.
        OUTSTR  [ASCIZ/Done
/]
        EXIT                            ;done.
;
```

```
;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
        MOVEI   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.
;Obtain an IO channel.  Connect it to device DSK.  Output in ASCII Mode

SETFILE:OPEN    OUT,[0                  ;Open a channel for DSK, ASCII mode
                SIXBIT/DSK/             ;Device is logical disk
                OBUFF,,0]               ;Output Buffer header.  No input BH
        JRST    NODISK                  ;Jump if we can't open this channel
;

;Select an Output File Name
        MOVE    A,['PLOTS ']            ;Make a 4-word block for ENTER.
        MOVSI   B,'OUT'                 ;First the name, then the extension
        SETZB   C,D                     ;let protection and PPN default
        ENTER   OUT,A                   ;Select SAMPLE.OUT as the output file
        JRST    NOENTR                  ;In case the ENTER fails.
        RET
;

;here in case of an error
IOERR:  OUTSTR  [ASCIZ/File output error
/]                                      ;send error typeout
        EXIT

NOENTR: OUTSTR  [ASCIZ/Could not ENTER the output file
/]                                      ;send error typeout
        EXIT

NODISK:  OUTSTR  [ASCIZ/Could not open the output channel
/]                                      ;send error typeout
        EXIT
;

;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: SOSLE   OBUFF+2                 ;decrement buffer byte count
        JRST    PUTCH1                  ;room remains in the buffer
        OUTPUT  OUT,                    ;Send the buffer.
        STATZ   OUT,740000              ;test for errors
        JRST    IOERR                   ;report error
PUTCH1: IDPB    A,OBUFF+1               ;add character to buffer
        RET

;Close the file and release the channel
CLSFILE:CLOSE   OUT,                    ;Close the channel
        STATZ   OUT,740000              ;test for errors
        JRST    IOERR                   ;jump if an error occurs
        RELEAS  OUT,
        RET

        END     START
```

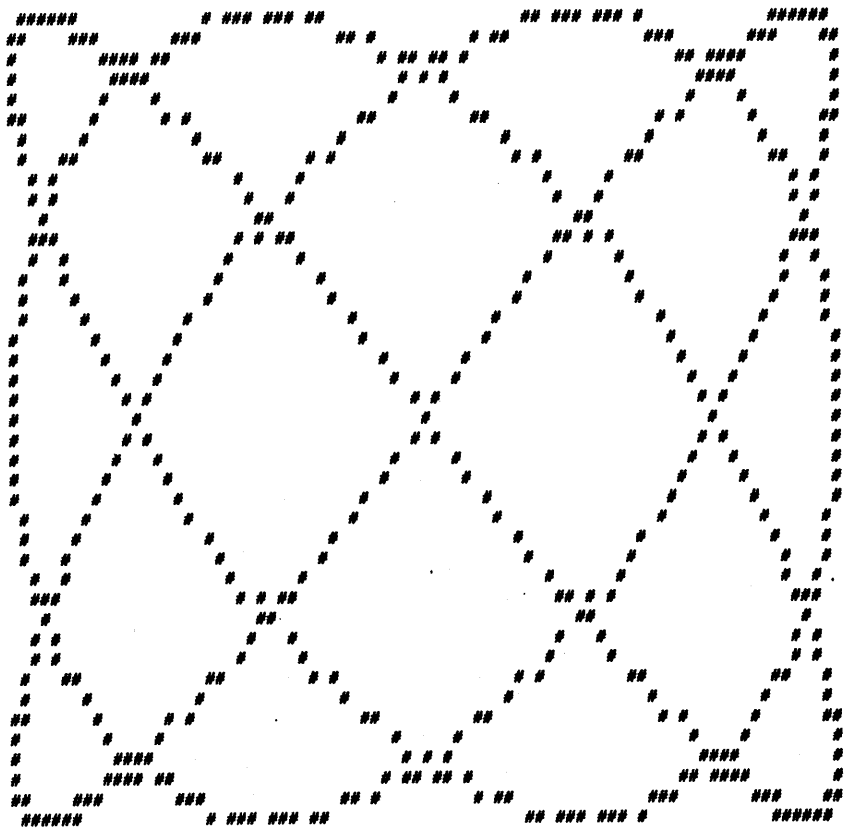


Figure 21-1: Sample Output from the LISAJ Subroutine

21.2.4. Fortran Library SIN Function

In case you find yourself curious, the following is a listing of the SIN routine that is found in version 5A of the DECsystem-10 Fortran Library. The code comes from SYS:FORLIB.REL; the comments are by this author. This is not the best computation of sine(x), but it does illustrate some of the influences of mathematics on programming. This example is presented without any explanation beyond the comments; 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 254.

```
        TITLE SIN Sine, Cosine and related function for FORLIB
        ENTRY SIN,COS,SIND,COSD          ;mark subroutine entry points

Comment $

The Maclaurin series expansion for Sin(X) is:

                  3      5      7      9
                 X      X      X      X
Sin(X) =  X -   ---  + ---  - ---  + ---  - ...
                 3!     5!     7!     9!

This subroutine transforms the given angle by dividing by pi/2.
The transformed angle is normalized into the first or fourth
quadrants (between -1.0 and +1.0) before expanding this series.
Expanding this series with abs(X) < 1.0 leads to more rapid
convergence.

---------

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    0
        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    0
        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,
        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.
;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, mantissa 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 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 represents [abs(X)/(pi/2)] modulo 1.0.  "How far into the quadrant"
;Register 2 contains the quadrant number 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
        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.
        TLCE    2,3000          ;skip if quad 2
        TLNN    2,3000          ;Quad 3 or 4.  Skip if Quad 4
        MOVNS   1               ;Quad 2 or 3, negate argument
```

```
;we arrive at S2 with register 1 containing the quantity x
;       let x0 = abs(X)/(pi/2) modulo 1.0
;       let q  = (abs(X)/(pi/2) - x0) modulo 4
;       then x is defined by the following table (note that in
;            this transformed system, 1.0 represents pi/2):
;       q = 0, quadrant 1, x = x0
;       q = 1, quadrant 2, x = 1-x0     Sin(x + pi/2) = Sin(pi/2 - x)
;       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)

S2:     SKIPGE  SX              ;test sign of original argument
        MOVNS   1               ;if negative, negate adjusted arg again.
        MOVEM   1,SX            ;save x, = X/(pi/2)
        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
SX:     0                       ;room to store the (adjusted) argument
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
SC:     0                       ;save accumulator 2 here
ONE:    1.0                     ;constant 1.0

;Special exit to restore things if any of these routines is called by the
;old JSA instruction.
CEXIT.: CEXIT.+1                ;address of special exit if called by JSA
        HLRM    16,CEXIT1       ;store the entry address in the EXCH instr
        HRLI    16,304000       ;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

$$S: \text{ARRAY } [L_1..U_1, L_2..U_2, \ \ldots\ , L_k..U_k]$$

then the address of a specific element, $S[i_1, i_2, \ \ldots\ , i_k]$, is given by the formula

$$S0 + (\ldots(E_1 * D_2 + E_2) * D_3 + E_3 \ \ldots) * D_k + E_k$$

where S0 is the address of $S[L_1, L_2, \ \ldots\ , L_k]$, 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. 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-10 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.[5] 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-10 virtual memory. When your program becomes too large to fit in physical memory, TOPS-10 allows your program to "go virtual" and use the virtual memory capabilities of the operating system. 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-10 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-10 suspends the program until that page can be brought into memory. Generally, except for these page faults which delay the process, the memory used by the program behaves as if it were really there, i.e., the memory seen by the program posesses all of the virtues of real memory (without the drawback of having had to buy it).

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

---

[5] The cache that is described here is the one present in the 1080 and 1090 configurations; other configurations may act somewhat differently.

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).

```
(* 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.[6]

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 a characteristic of an operating system, not of particular user programs. However, a program that uses memory in a 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 situations are usually less clear-cut than this example, the programmer should consider his or her program's effect on system performance.

21.5. ARRAY EXERCISES

These exercises are an opportunity to practice using two-dimensional arrays.

---

[6] This is an oversimplification, but the principle is correct.

21.5.1. 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 x N square, so that the sum of each column, of each row, and of each of the two main diagonals is identical.

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 $(N^3+N)/2$.

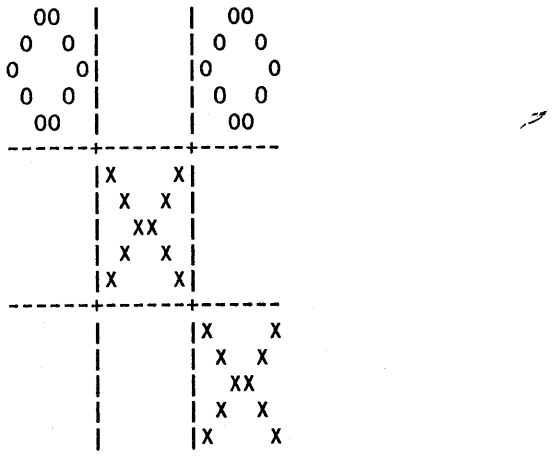
Extra credit: this exercise was designed to give you practice with arrays. However, there really isn't any need to use an array here. Develop a closed-form expression for the value of a position, given three parameters: the order of the square, a row number, and a column number.

21.5.2. 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 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:



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 x 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.

21.5.3. 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 x N` upper triangular matrix can be stored in `[N x (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 MUUO operations that we have already studied are relevant. These include OPEN, CLOSE, STATZ, and RELEAS. In addition to these, we will need to learn the LOOKUP MUUO for selecting the input file, and the INPUT MUUO for reading data.

Buffered input is quite similar to buffered output: the INPUT MUUO makes use of buffers and buffer headers that have the same format as we have used for output. Again, our fundamental character reading routine will be governed by a control count and a byte pointer that we find in the buffer header. The major new problem we face when doing input processing is how to detect the end of file.

22.1. OPEN FOR INPUT

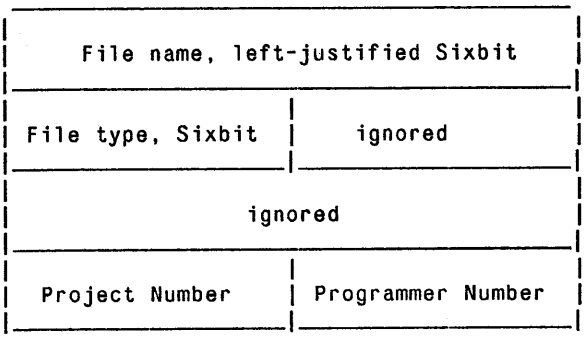
The OPEN MUUO is used for input in much the same way as for output. The fundamental exception is that for input we must specify the address of the input buffer header in the right half of the third argument word. The input buffer header is three or four words in the same format as the buffer header for output.

22.2. LOOKUP MUUO

For input, file selection is accomplished by means of the LOOKUP MUUO. The arguments to LOOKUP are similar to those for ENTER. The effective address of the LOOKUP MUUO specifies the address of a four-word argument block.

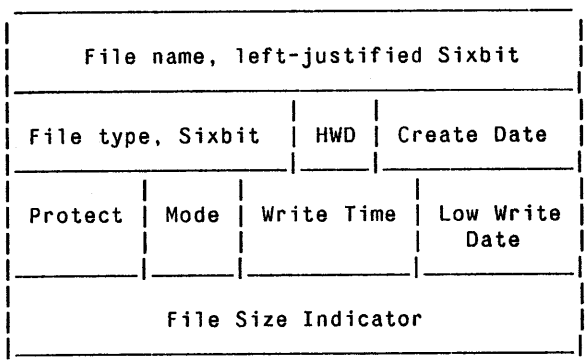
The first word of the block specifies the file name of the desired input file. File names, as with the ENTER MUUO are specified in the six-bit subset of ASCII. The left-half of the second word contains the file type or extension. The fourth word contains the project-programmer number of the disk area where the file can be found. In TOPS-10, it is usual to represent project-programmer names as two octal fields; the project number is right-justified in the left halfword and the the programmer number is right-justified in the right halfword. The format of the argument block needed for LOOKUP is show here:

Argument Block for LOOKUP



When a LOOKUP function has been executed successfully, the operating system modifies the argument block and returns some useful information in it. The format of the block after the successful execution of the LOOKUP is depicted below:

Result After a Successful LOOKUP



The field HWD, *high-order Write Date*, which is composed of bits 18:20 of the second word, taken together with bits 24:35 of the third word are the 15-bit file write date in the TOPS-10 format for dates: ((Year-1964)*12 + Month-1)*31 + Day-1. The time of day when the file was written is expressed in minutes since midnight in the 11-bit field in bits 13:23 of the third word. The mode field, bits 9:12 of the third word, reflect the output mode used by the program that wrote this file. Bits 0:8 of the third word specify the three file protection fields associated with this file. The file size indicator specifies the size of the file. For small files, this word is negative, and it should be swapped and then negated to reveal the count of data words in the file. For large files, this word is positive and it represents the number of 128-word records in the file.

Note that since TOPS-10 changes the contents of the argument block to LOOKUP, it is not possible to re-use the same argument block unless the program reinitializes it.

22.3. SIMPLE DISK INPUT, EXAMPLE 13A

The simplest program that displays the mechanism used for input is shown below. This program will read a given file, SAMPLE.DAT and type the contents of that file on the terminal.

```
        TITLE   Simple Disk Input.   Example 13a

A=1
B=2
C=3
D=4
P=17
;

INCHAN==10                              ;Channel for input

OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ P,]

PDLEN==100

PDLIST: BLOCK   PDLEN                   ;Room for the stack
IBUFH:  BLOCK   3                       ;Input channel buffer header
;

START:  RESET                           ;Normalize all I/O activity
        MOVE    P,[IOWD PDLEN,PDLIST]   ;Establish a stack pointer
        CALL    OPENFI                  ;Get the channel, Do Lookup
LOOP:   CALL    GETCHR                  ;get a character from the file
        JRST    EOF                     ;on end-of-file, jump out
        OUTCHR  A                       ;Print the character
        JRST    LOOP                    ;Do more
;

EOF:    OUTSTR  [ASCIZ/
All done.
/]
        EXIT
;

OPENFI: MOVEI    A,0                    ;Open a disk channel
        MOVSI    B,'DSK'                ;Device is DSK
        MOVEI    C,IBUFH                ;Input Buffer header address
        OPEN     INCHAN,A               ;Open the channel
        JRST     NODISK                 ;Can't get the channel.  Error
        MOVE     A,['SAMPLE']           ;Setup 4-words for LOOKUP
        MOVSI    B,'DAT'                ;File name in A, Type in B
        SETZB    C,D                    ;Zero C and D: Use logged-in PPN
        LOOKUP   INCHAN,A               ;Select the input file
        JRST     NOFILE                 ;Can't get the file name
        RET
;

GETCHR: SOSLE    IBUFH+2                ;Decrement buffer byte count
        JRST     GETCH1                 ;Get a character from the buffer
        INPUT    INCHAN,                ;Get another buffer
        STATZ    INCHAN,740000          ;Test for errors
        JRST     INERR                  ;An error
        STATZ    INCHAN,020000          ;Test for end of file
        RET                             ;No-skip return for eof
GETCH1: ILDB     A,IBUFH+1              ;get a byte from the buffer
        JUMPE    A,GETCHR               ;discard null bytes from file
CPOPJ1: AOS      (P)                    ;Return with Skip and char. in A
CPOPJ:  RET
;
```

```
INERR:  OUTSTR  [ASCIZ/Input Error
/]
        EXIT

NOFILE: OUTSTR  [ASCIZ/Can't find SAMPLE.DAT
/]
        EXIT

NODISK: OUTSTR  [ASCIZ/Can't obtain a channel for disk input
/]
        EXIT

        END     START
```

This program does not contain too much that is mysterious, yet it demonstrates disk input for the first time. The program makes the usual definitions of accumulators names and stack space.

The main program is quite simple. After its initialization tasks, the main program calls OPENFI to obtain an input channel and select a file for input. Once this has been accomplished, the main program enters a loop in which it calls GETCHR to obtain characters from the input file. These characters are sent to the terminal via an OUTCHR MUUO. The loop continues until the non-skip return from GETCHR signals that the end of the input file has been reached.

Let us now examine OPENFI and GETCHR in some detail. The program provides a buffer header control block called IBUFH. Its function is similar to that of the output buffer control block. The address of this control block appears in the right half of the third word of the argument block for OPEN. In the OPENFI routine, we use accumulators A, B, and C to hold the argument block for OPEN.

```
OPENFI: MOVEI   A,0             ;Open a disk channel
        MOVSI   B,'DSK'         ;Device is DSK
        MOVEI   C,IBUFH         ;Input Buffer header address
        OPEN    INCHAN,A        ;Open the channel
        JRST    NODISK          ;Can't get the channel.  Error
```

Mode zero, signifying 7-bit buffered characters is set up in register A. A MOVSI instruction is used to load the left half of register B with the device name DSK. Finally, the address of the input buffer control block is copied to the right half of register C. The program then performs the OPEN MUUO; assuming all goes well, the OPEN will skip.

Next, we select the file for input. We setup the four word block for LOOKUP in the accumulators. Note the similarity to the arguments needed for ENTER.

```
        MOVE    A,['SAMPLE']    ;Setup 4-words for LOOKUP
        MOVSI   B,'DAT'         ;File name in A, Type in B
        SETZB   C,D             ;Zero C and D: Use logged-in PPN
        LOOKUP  INCHAN,A        ;Select the input file
        JRST    NOFILE          ;Can't get the file name
        RET
```

If all goes well, the LOOKUP will skip and this routine will return to its caller.

The GETCHR is quite similar to the PUTCHR routine that we used for output. The third word of the input buffer header block (IBUFH+2) contains the count of characters that remain in the current buffer. Initially, this count is set to zero by means of the OPEN MUUO. The very first time this routine is called, because the count is zero, the SOSLE will skip and the INPUT MUUO will be performed.

If there are no buffers defined for this channel, the INPUT MUUO will cause buffers to be built. Then, the first record of the disk file is copied into the first buffer. The INPUT MUUO returns with an updated

byte count in IBUFH+2 and a byte pointer in IBUFH+1. After some tests (which we shall discuss later) the program arrives at GETCH1 where it reads a character from the buffer and, if the character is non-zero, returns it to the caller. Null bytes, which some editors install in files for padding, are discarded by jumping back to GETCHR.

Two STATZ MUUOs follow the INPUT MUUO. The first STATZ tests the status bits that indicate device or system errors. The STATZ skips if no such error indications are present. The second STATZ tests the status bit that signifies end of file. When the end of file status bits becomes one, the STATZ will avoid skipping, and the GETCHR routine will return without a skip.

```
GETCHR: SOSLE   IBUFH+2             ;Decrement buffer byte count
        JRST    GETCH1              ;Get a character from the buffer
        INPUT   INCHAN,             ;Get another buffer
        STATZ   INCHAN,740000       ;Test for errors
        JRST    INERR               ;An error
        STATZ   INCHAN,020000       ;Test for end of file
        RET                         ;No-skip return for eof
GETCH1: ILDB    A,IBUFH+1           ;get a byte from the buffer
        JUMPE   A,GETCHR            ;discard null bytes from file
CPOPJ1: AOS     (P)                 ;Return with Skip and char. in A
CPOPJ:  RET
```

22.4. SEARCH PROGRAM, EXAMPLE 13B

The next 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 that demonstrates this program. The user's type-in is underlined. Note that when the device name TTY: is used this program outputs to the terminal. When we come to look at the program, we'll discover that buffered mode output to the terminal uses precisely the same instructions as we have used for buffered output to the disk. This is our first example of *device independence*, by which the same system calls are used to affect a variety of devices.

```
.execute ex13b
MACRO:  File
LINK:   Loading
[LNKXCT FILE execution]
File name for input: b21.mss
File name for output: tty:
Please enter the search string: title

Search of B21.MSS for
TITLE
-----------------------------------------------------------------------------
        Title   Factorials up to 100!  Example 11
        TITLE   Plot Program - Example 12
        TITLE SIN Sine, Cosine and related function for FORLIB

There were 3 matches found in the file
```

```
File name for input: b21.mss
File name for output: ex13b.out
Please enter the search string: complicated
File name for input: who.me[3,4]        this file doesn't exist
Lookup failed to find file WHO.ME[3,4]
File name for input: _      type carriage return here

EXIT
^C
.type ex13b.out

Search of B21.MSS for
COMPLICATED
------------------------------------------------------------------------------
program will be long and complicated.  Indeed not!  It is only a few
The control structure for the loop at @t<MLOOP> is somewhat more complicated
we have to use more complicated programming techniques than simple
complicated as the number of dimensions increases.

There were 4 matches found in the file
EXIT
^C
.
```

22.4.1. Structured Programming

This program is the most complex that we have seen thus far; as programs become more complex, the 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    STOP                    ;None was specified. 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
```

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 13b

A=1
B=2
C=3
D=4
W=5
X=6
Y=7
Z=10
P=17

OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ P,]

INPCHN==10                              ;Define channel for input
OUTCHN==11                              ;Define channel for output

DEVNAM==0                               ;field names within a file block
FILNAM==1
FILTYP==2
FILPPN==3
FILBSZ==4                               ;size of a file block

;

;Main Program
START:  RESET
        MOVE    P,[IOWD PDLEN,PDLIST]   ;Set up the stack
NEXT:   CALL    GTINPF                  ;Get an input file
        JRST    STOP                    ;none was given.  Stop
        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

STOP:   EXIT

;
```

```
;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
        MOVEI   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: MOVE    B,[POINT 7,IBUF]        ;read one line from the file
GETLN1: CALL    GETCHR                  ;read a character
        JRST    EOFLIN                  ;Line ends here.
GETLN2: IDPB    A,B                     ;store the character in IBUF
        CAIE    A,12                    ;look for a line feed.
        JRST    GETLN1                  ;loop through line
        MOVEI   A,0
        IDPB    A,B                     ;Store null to end buffer
        JRST    CPOPJ1
;

EOFLIN: CAMN    B,[POINT 7,IBUF]        ;have we got anything on this line?
        RET                             ;nothing.  report end of file
        MOVEI   A,15                    ;Partial Line.  Add CR, LF, Null
        IDPB    A,B                     ;  and report it as a normal line
        MOVEI   A,12                    ;  Report eof next time
        JRST    GETLN2                  ;Add LF and Null.
;

;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"
        JRST    .+2                     ;not lower-case.
        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:    CLOSE   INPCHN,                 ;Close and release the input file
        RELEAS  INPCHN,
        MOVEI   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   A,MATCH                 ;count of matches.  2 or more?
        JRST    ZMATCH                  ;no matches found
        MOVE    Y,[CALL PUTCHR]         ;print to file
        CALL    DECOU1                  ;Print decimal to file
FINMES: MOVEI   B,[ASCIZ/ matches found in the file
/]
FINMS1: CALL    DOOUT
        CLOSE   OUTCHN,
        STATZ   OUTCHN,740000
        JRST    OUTERR
        RELEAS  OUTCHN,
        RET
;

ZMATCH: MOVEI   B,[ASCIZ/no/]
        CALL    DOOUT
        JRST    FINMES

ONEMAT: MOVEI   B,[ASCIZ/
There was one match found in the file
/]
        JRST    FINMS1
;

;low-level I/O routines
PUTCHR: SOSLE   OBUFH+2                 ;Decrement Buffer byte count
        JRST    PUTCH1                  ;Room left in buffer
        OUTPUT  OUTCHN,                 ;send old buffer
        STATZ   OUTCHN,740000           ;watch for errors
        JRST    OUTERR                  ;bad
PUTCH1: IDPB    A,OBUFH+1               ;add character to buffer
        RET

GETCHR: SOSLE   IBUFH+2                 ;Decrement buffer byte count
        JRST    GETCH1                  ;Buffer is non-empty. go get byte
        INPUT   INPCHN,                 ;get a new buffer
        STATZ   INPCHN,740000           ;watch for errors
        JRST    INERR                   ;bad
        STATZ   INPCHN,20000            ;check for end of file
        RET                             ;non-skip return for eof
GETCH1: ILDB    A,IBUFH+1               ;get a byte from the buffer
        JUMPE   A,GETCHR                ;discard null bytes
        JRST    CPOPJ1                  ;return byte in A, with skip
;

GTTYLN: MOVE    B,[POINT 7,TTYBUF]      ;Read one line to TTYBUF
GTTYL1: INCHWL  A                       ;Get one character
        CAIN    A,15                    ;discard CR
        JRST    GTTYL1
        CAIN    A,12                    ;convert LF to null
        MOVEI   A,0
        IDPB    A,B                     ;store in buffer,
        JUMPN   A,GTTYL1                ;loop until end of line
        RET
;
```

```
;Error handling
INERR:  OUTSTR  [ASCIZ/Input error.  Status = /]
        GETSTS  INPCHN,A
OCTERR: CALL    OCTOUT
        OUTSTR  CRLF
        EXIT

OUTERR: OUTSTR  [ASCIZ/Output error.  Status = /]
        GETSTS  OUTCHN,A
        JRST    OCTERR                  ;print octal error code & stop

NOOPEN: OUTSTR  [ASCIZ/Open failed for device /]
        MOVE    Y,[OUTCHR A]
        CALL    TYDEVN
        OUTSTR  CRLF
        RET

NOLOOK: OUTSTR  [ASCIZ/Lookup failed to find file /]
        CALL    TYFILN
        OUTSTR  CRLF
        RET

NOENTR: OUTSTR  [ASCIZ/Enter failed to select output file /]
        CALL    TYFILN
        OUTSTR  CRLF
        RET
;

;Get input file name from terminal. open it for 7-bit bytes.
GTINPF: OUTSTR  [ASCIZ/File name for input: /]
        CALL    GTTYLN                          ;Read a line from terminal
        LDB     A,[POINT 7,TTYBUF,6]            ;read first byte of response
        JUMPE   A,CPOPJ
        MOVEI   Z,IFILBK                        ;select input file block
        MOVE    W,[POINT 7,TTYBUF]
        CALL    SCNFIL                          ;scan a file name
        JRST    GTINPF                          ;can't parse file name
        MOVEI   A,0                             ;mode 0 for input
        MOVE    B,DEVNAM(Z)                     ;device name from SCNFIL
        MOVEI   C,IBUFH                         ;Address of input buffer header
        OPEN    INPCHN,A                        ;Open the channel
        JRST    [CALL NOOPEN                    ;if error, print message
                 JRST GTINPF]                   ;loop. try again
        MOVE    A,FILNAM(Z)                     ;Setup for LOOKUP
        MOVE    B,FILTYP(Z)                     ;Copy parameters from SCNFIL
        SETZ    C,
        MOVE    D,FILPPN(Z)
        LOOKUP  INPCHN,A                        ;obtain read access to file
        JRST    [CALL NOLOOK                    ;error: print message
                 JRST GTINPF]                   ;try again
        JRST    CPOPJ1                          ;return to caller w/skip
;
```

```
;Get output file name from terminal, open it for 7-bit bytes.
GTOUTF: OUTSTR  [ASCIZ/File name for output: /]
        CALL    GTTYLN                          ;Read a line from terminal
        MOVEI   Z,OFILBK                        ;select output file block
        MOVE    W,[POINT 7,TTYBUF]
        CALL    SCNFIL                          ;scan a file name
        JRST    GTOUTF                          ;don't understand the file name
        MOVEI   A,0                             ;Output mode 0: buffered ascii
        MOVE    B,DEVNAM(Z)                     ;device name from SCNFIL
        MOVSI   C,OBUFH                         ;output buffer header
        OPEN    OUTCHN,A                        ;open the channel
        JRST    [CALL NOOPEN                    ;error: print message
                 JRST GTOUTF]                   ; and loop
        MOVE    A,FILNAM(Z)                     ;setup for Enter
        MOVE    B,FILTYP(Z)                     ;use results from SCNFIL
        SETZ    C,
        MOVE    D,FILPPN(Z)
        ENTER   OUTCHN,A                        ;select file for output
        JRST    [CALL NOENTR                    ;error: print message
                 JRST GTOUTF]                   ;loop
        RET
;

;Get search string from terminal.  Copy as uppercase to BUFFER.
GTSTRG: OUTSTR  [ASCIZ/Please enter the search string: /]
        CALL    GTTYLN                  ;Read a line from terminal
        MOVE    W,[POINT 7,TTYBUF]      ;Point to string
;Convert lower-case to upper.  Place null at end of string.
        MOVE    B,[POINT 7,BUFFER]      ;Point to destination
GTSSCN: ILDB    A,W                     ;get a byte from the search string
        CAIL    A,"a"                   ;lower-case
        CAILE   A,"z"                   ;
        JRST    .+2                     ;
        TRZ     A,40                    ;is transformed to upper-case
        IDPB    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: MOVEI   B,[ASCIZ/
Search of /]
        CALL    DOOUT                   ;send "search of"
        MOVEI   Z,IFILBK                ;
        MOVE    Y,[CALL PUTCHR]         ;Instruction for TYFILN
        CALL    TFILNX                  ;"type file name"
        MOVEI   B,[ASCIZ/ for
/]
        CALL    DOOUT                   ;"for"
        MOVEI   B,BUFFER                ;send the text of the
        CALL    DOOUT                   ;search string
        MOVEI   B,[ASCIZ/
-------------------------------------------------------------------------
/]                                      ;a separation
DOOUT:  TLNN    B,-1                    ;skip if the left of B is non-zero
        HRLI    B,440700                ;set a byte point in Left of B
DOOUT1: ILDB    A,B                     ;get a byte
        JUMPE   A,CPOPJ                 ;exit if null
        CALL    PUTCHR                  ;send character to output
        JRST    DOOUT1                  ;loop to send more.

;
```

```
;File name and device type out.
TYDEVN: MOVE    B,DEVNAM(Z)             ;Must type device name.
        CALL    SIXOUT                  ;print as sixbit
        MOVEI   A,":"                   ;add colon after dev name
        XCT     Y
        RET

;Call TYFILN with Z = address of file descriptor block.
;Output file name to TTY.
;Call TFILNX with Z setup as above and Y containing an instruction to XCT
TYFILN: MOVE    Y,[OUTCHR A]            ;instr to execute to send data
TFILNX: SKIPN   B,DEVNAM(Z)             ;get the device name
        JRST    TFILN1                  ;skip this part if there's none
        CAME    B,['DSK   ']            ;don't print DSK either
        CALL    TYDEVN                  ;print the device name
TFILN1: SKIPN   B,FILNAM(Z)             ;print the file name, if present
        RET                             ;exit now if no file name
        CALL    SIXOUT                  ;print the file name
        HLLZ    B,FILTYP(Z)
        JUMPE   B,TFILN2                ;jump if no extension
        MOVEI   A,"."                   ;print period after name
        XCT     Y
        CALL    SIXOUT                  ;print file extension
TFILN2: SKIPN   FILPPN(Z)               ;print PPN if non-zero
        RET                             ;return quick
        MOVEI   A,"["                   ;print the square bracket
        XCT     Y
        HLRZ    A,FILPPN(Z)             ;print the Project part
        CALL    OCTOU1                  ; in octal
        MOVEI   A,","                   ;comma to separate p,pn
        XCT     Y
        HRRZ    A,FILPPN(Z)             ;print programmer part in octal
        CALL    OCTOU1
        MOVEI   A,"]"                   ;finish ppn.
        XCT     Y
        RET

;Print Sixbit of B via XCT Y
SIXOUT: JUMPE   B,CPOPJ                 ;sixbit output
        MOVEI   A,0                     ;clear high word
        LSHC    A,6                     ;slide the data up into A
        ADDI    A," "                   ;convert sixbit to ascii
        XCT     Y                       ;print it and loop
        JRST    SIXOUT

;OCTOUT:  Print octal of A on TTY.
;OCTOU1:  Print octal of A via XCT Y
OCTOUT: MOVE    Y,[OUTCHR A]            ;recursive octal printer
OCTOU1: IDIVI   A,10
        PUSH    P,B
        SKIPE   A
        CALL    OCTOU1
        POP     P,A
        ADDI    A,"0"
        XCT     Y
        RET
;
```

```
;DECOUT:  Print decimal of A on TTY.
;DECOU1:  Print decimal of A via XCT Y
DECOUT: MOVE    Y,[OUTCHR A]
DECOU1: IDIVI   A,12
        PUSH    P,B
        SKIPE   A
        CALL    DECOU1
        POP     P,A
        ADDI    A,"0"
        XCT     Y
        RET
;

SCNFIL: MOVSI   A,'DSK'                 ;assume device DSK
        MOVEM   A,DEVNAM(Z)             ;as the default
        SETZM   FILNAM(Z)               ;clear out everything else
        SETZM   FILTYP(Z)
        SETZM   FILPPN(Z)
        CALL    GETSIX                  ;device name or file name first
        CAIE    A,":"                   ;if there's a colon, this is dev
        JRST    SCNFL1                  ;no colon, this was the file name
        MOVEM   B,DEVNAM(Z)             ;save the device name
        CALL    GETSIX                  ;next thing must be the file name
SCNFL1: MOVEM   B,FILNAM(Z)             ;save the file name
        CAIE    A,"."                   ;if there's a period, get file type
        JRST    SCNFL2                  ;no file type present
        CALL    GETSIX                  ;read the file type
        HLLZM   B,FILTYP(Z)             ;save it
SCNFL2: CAIE    A,"["                   ;PPN next?
        JRST    CPOPJ1                  ;no.  assume we're happy
        CALL    GETOCT                  ;read an octal number
        HRLZM   B,FILPPN(Z)             ;store project number
        CAIE    A,","                   ;must see a comma next
        JRST    SCNERR                  ;error if it's not there
        CALL    GETOCT                  ;read the programmer part
        HRRM    B,FILPPN(Z)             ;save it
        CAIE    A,"]"                   ;must see the closing bracket
        JRST    SCNERR                  ;error
        CALL    GETSIX                  ;skip past the closing bracket
        JUMPN   B,SCNERR                ;any non-blank here is an error
        JRST    CPOPJ1                  ;return happy.

SCNERR: OUTSTR  [ASCIZ/I can't understand this file name
/]
        RET
;

;Read from byte pointer in W and return octal value in B
GETOCT: MOVEI   B,0                     ;initial value is zero
GETOC1: ILDB    A,W                     ;get a character
        CAIL    A,"0"
        CAILE   A,"7"
        RET                             ;not part of my number
        LSH     B,3                     ;shift old value
        ADDI    B,-"0"(A)               ;add in the next digit
        JRST    GETOC1                  ;loop
;
```

```
;get the next alpha-numeric sixbit term.
;input byte pointer in W.  Result in B, ascii delimiter in A
GETSIX: MOVEI   B,0                     ;Clear out result word
        MOVE    C,[POINT 6,B]           ;byte pointer into result word
GETSX1: ILDB    A,W                     ;get a character
        CAIL    A,"a"                   ;convert lowercase to sixbit
        CAILE   A,"z"
        JRST    GETSX2                  ;not lowercase. see what else it is
        TRZ     A,100                   ;lowercase to sixbit
        JRST    GETSX4                  ;go store this character

GETSX2: CAIL    A,"A"                   ;uppercase letter?
        CAILE   A,"Z"
        JRST    .+2
        JRST    GETSX3                  ;uppercase letter
        CAIL    A,"0"                   ;look for digits
        CAILE   A,"9"
        RET                             ;not part of a file name  return it
GETSX3: SUBI    A," "                   ;convert to sixbit
GETSX4: TLNE    C,770000                ;don't overflow out of B
        IDPB    A,C                     ;store character in B
        JRST    GETSX1                  ;loop

;

PDLEN==100
BUFLEN==100
PDLIST: BLOCK   PDLEN
BUFFER: BLOCK   BUFLEN                  ;search string buffer
TTYBUF: BLOCK   BUFLEN                  ;tty input line buffer
IBUF:   BLOCK   BUFLEN                  ;file input line buffer
MATCH:  0                               ;count of the number of matches
IBUFH:  BLOCK   3                       ;Input buffer header
OBUFH:  BLOCK   3                       ;Output buffer header
IFILBK: BLOCK   FILBSZ                  ;Input file name block
OFILBK: BLOCK   FILBSZ                  ;Output file name block
CRLF:   BYTE (7) 15,12

        END     START
```

Now we must discuss some of the routines in detail.

22.4.2. GTINPF - Get Input File

This subroutine prompts for the name of an input file. It calls GTTYLN to obtain a complete line from the terminal, which is placed in TTYBUF. If the line is empty, GTINPF returns without skipping; the main program will jump to STOP and exit. Assuming the line is not empty, GTINPF calls loads register Z with the address of a file descriptor block and register W with a byte pointer to the input line; these are parameters for the SCNFIL routine. SCNFIL, which we will examine later, reads the text from the input line and interprets it as the name of a device, file, file type, and project-programmer number. When SCNFIL is sucessful, it skips. If the file specification is not recognizable by SCNFIL, the direct return is taken and GTINPF starts over.

When SCNFIL is successful, it returns with the device, file name, file type, and ppn stored in the file descriptor block that register Z addresses. These are then used as data in the OPEN and LOOKUP MUUOs that follow. The field DEVNAM from the file descriptor block is used as an argument in OPEN. The file name, file type, and ppn are used in the LOOKUP. If either the OPEN or the LOOKUP fail, an appropriate error message is printed and the program loops back to GTINPF to try again. On success, the GTINPF routine returns with a skip.

The GTOUTF routine is very nearly the same as GTINPF. Note that the OPEN MUUO uses a different channel name and it places the address of the output buffer in the left half of the third word.

22.4.3. File Name Scan

The external file specification is converted to internal form by the SCNFIL routine. SCNFIL begins by initializing the file descriptor block that is passed to it by its caller. The device name DSK is stored in the file descriptor, at DEVNAM(Z), as a default. The balance of the block addressed by Z is cleared.

Next, SCNFIL calls the GETSIX routine. GETSIX uses the byte pointer in W and collects one alpha-numeric sixbit name in register B. It returns the name, and the delimiter character that follows the name is returned in A. If the delimiter character is a colon, then SCNFIL assumes that register B contains a device name. That device name is stored in DEVNAM(Z) and the next sixbit term is obtained; this term must be the file name. If there was no colon present, the routine assumes that register B contains a file name. At SCNFL1 the file name is stored into FILNAM(Z). If the file name is followed by a period, the routine gets a file extension and stores it in FILTYP(Z). At SCNFL2 the program decides if it is necessary to read a project-programmer name. If so, the two octal fields are read and stored in FILPPN(Z). If anything is encountered that confuses the SCNFIL routine, it complains and returns without skipping.

The routine GETSIX is relatively straightforward. Register B is cleared to zero; register C contains a byte pointer for depositing six-bit characters into B. Bytes are read via the pointer in register W. If the character is alpha-numeric, it is converted to sixbit and stored at GETSX4. The only strange thing in this routine is the use of the TLNE instruction at GETSX4. This TLNE instruction skips if bits 0:5 of register C are zero. These bits will be zero after six characters are deposited in B. Thus, if a seven-character or longer alphanumeric appears, only the first six letters will be stored in B; the remainder will be discarded.

22.4.4. GTSTRG - Get Search String

The GTSTRG routine prompts for a search string and calls GTTYLN to read it from the terminal. Then, the search string is converted to upper case and copied from TTYBUF to the area called BUFFER. The string in TTYBUF ends with a null; this null is copied to BUUFER.

```
GTSTRG: OUTSTR  [ASCIZ/Please enter the search string: /]
        CALL    GTTYLN                  ;Read a line from terminal
        MOVE    W,[POINT 7,TTYBUF]      ;Point to string
;Convert lower-case to upper.  Place null at end of string.
        MOVE    B,[POINT 7,BUFFER]      ;Point to destination
GTSSCN: ILDB    A,W                     ;get a byte from the search string
        CAIL    A,"a"                   ;lower-case
        CAILE   A,"z"                   ;
        JRST    .+2                     ;
        TRZ     A,40                    ;is transformed to upper-case
        IDPB    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
```

22.4.5. 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 converts the address given in B to a byte pointer and

then runs a loop that copies characters from that string, calling PUTCHR to store each character in the output buffer.

```
HEADER: MOVEI   B,[ASCIZ/
Search of /]
        CALL    DOOUT                   ;send "search of"
```

Next, HEADER writes the name of the input file in the output stream. This is accomplished by setting register Y to contain the instruction CALL PUTCHR and Z to contain the address of the input file descriptor block, and calling TFILNX. TFILNX converts the internal format file name to a string. Each character in the string is sent to the output file by executing the instruction found in Y.

```
        MOVEI   Z,IFILBK                ;
        MOVE    Y,[CALL PUTCHR]         ;Instruction for TYFILN
        CALL    TFILNX                  ;"type file name"
```

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.

```
        MOVEI   B,[ASCIZ/ for
/]
        CALL    DOOUT                   ;"for"
        MOVEI   B,BUFFER                ;send the text of the
        CALL    DOOUT                   ;search string
        MOVEI   B,[ASCIZ/
-------------------------------------------------------------------------
/]                                      ;a separation
DOOUT:  TLNN    B,-1                    ;skip if the left of B is non-zero
        HRLI    B,440700                ;set a byte point in Left of B
DOOUT1: ILDB    A,B                     ;get a byte
        JUMPE   A,CPOPJ                 ;exit if null
        CALL    PUTCHR                  ;send character to output
        JRST    DOOUT1                  ;loop to send more.
```

22.4.6. 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
        MOVEI   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.

22.4.7. GETLIN and EOFLIN

The GETLIN routine accumulates an input line into the region called IBUF. GETLIN uses repeated calls to the GETCHR routine to read a line from the file; the characters that are read are deposited into the buffer called IBUF.

```
GETLIN: MOVE    B,[POINT 7,IBUF]        ;read one line from the file
GETLN1: CALL    GETCHR                  ;read a character
        JRST    EOFLIN                  ;Line ends here.
GETLN2: IDPB    A,B                     ;store the character in IBUF
        CAIE    A,12                    ;look for a line feed.
        JRST    GETLN1                  ;loop through line
        MOVEI   A,0
        IDPB    A,B                     ;Store null to end buffer
        JRST    CPOPJ1

EOFLIN: CAMN    B,[POINT 7,IBUF]        ;have we got anything on this line?
        RET                             ;nothing.  report end of file
        MOVEI   A,15                    ;Partial Line.  Add CR, LF, Null
        IDPB    A,B                     ;  and report it as a normal line
        MOVEI   A,12                    ;  Report eof next time
        JRST    GETLN2                  ;Add LF and Null.
```

In the normal course of events, the GETCHR routine will return with a skip and characters will be deposited into IBUF until a line feed is stored, whereupon the routine will exit with a skip return.

When end of file occurs, the GETCHR routine returns without skipping. The program jumps to EOFLIN. If the input line is empty (nothing was deposited via B) then this routine returns without a skip. On the other hand, if there are characters present in IBUF they are part of the last line of the file. To these characters the program adds a carriage return and a line feed. GETLIN then returns with a skip, but the next time that it's called, it will have an empty line and end-of file, and so the next call to GETLIN will pass back the end-of-file indication.

22.4.8. 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.

22.4.8.1. 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 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
```

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.[1]

---

[1] 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.

22.4.8.2. 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.

22.4.9. FIN - Finish Routine

The FIN routine is responsible for writing a trailer into the output file and closing the files. First, the input file is closed (and released). 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
```

The code to do this is relatively obvious. After writing the concluding messages, `FIN` closes and releases the output file and returns.

22.5. EXERCISES

22.5.1. 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 x 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.

22.5.2. 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.

22.5.3. Crossword Puzzle

The object of this exercise is to write a program to "draw" a crossword puzzle diagram.[2] This is an example of an application where the problem of correctly formatting the output is nearly as challenging as the other computations.

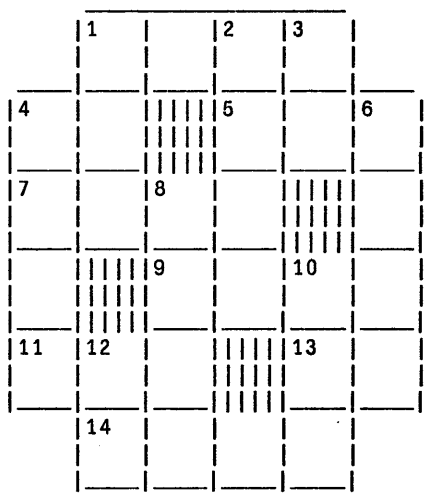
You will be given as input a matrix of zeros and ones. An entry of zero indicates a white square; a one indicates a black square. The output should be the diagram of the puzzle, with the appropriate white squares numbered for words *across* and *down*.

For example, given the matrix

[2]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:

```
                                bbbb|
Unnumbered White Squares:       bbbb|       "b" represents a blank space
                                ____|

                                nnbb|       nbbb|
Numbered White Squares:         bbbb|  or   bbbb|
                                ____|       ____|

                                |||||
Black Squares:                  |||||
                                |||||

-1 Squares, Depending on neighboring -1 Squares:

     bbbbb      bbbb|     bbbbb     bbbbb     bbbb|
     bbbbb      bbbb|     bbbbb     bbbbb     bbbb|
     bbbbb      bbbb|     _____     ____b     ____|
```

Each puzzle will appear in the input data as a 23 x 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
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-10. The structure of a directory is relatively simple; user-written programs have no difficulty in reading the directory and stepping through it file-by-file.

This program will gather the file name of 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. This information is then sorted by the name and extension of the file. Finally, the program will print the sorted directory listing, augmented by additional facts about each file.

23.1. DIRECTORY PROCESSING

The file name of a file directory is the same as the project programmer name of the user to whom the file belongs. The extension is `UFD`, meaning *user file directory*. The file directories are found in the directory belonging to project `1`, programmer `1`. The directory of `[1,1]` is called the *master file directory* or MFD. We read our file directory by performing a `LOOKUP` that specifies our own PPN as the file name, `UFD` as the extension and `<1,,1>` as the PPN of the file owner.

The data that we find in the directory file consists of two-word entries. The first word is the name of a file in the directory. The second word contains the file extension in the left halfword, and a number called a *file retrieval pointer* in the right halfword. A file name that is zero appears in those places where files have been deleted from the UFD. When the name is zero, we will discard the word that follows it.

Our program will perform the following functions:

- Open a channel for reading the UFD. For variety, we take this opportunity to demonstrate *unbuffered* or *dump-mode* input.
- Lookup the UFD corresponding to the project programmer name of the user who is running this program.
- Determine the size of the UFD, allocate memory space for it, and read the entire UFD into memory.
- Process the UFD entries: remove the entries for deleted files and sort the existing files by file name and file extension.
- Print a report for each file listing the name, extension, size, protection, mode and date of creation of the file.

23.2. DYNAMIC SPACE ALLOCATION

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. The `LINK` program helps us identify such an area of our address space. When `LINK` finishes loading a program into memory, 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`. The `RESET` MUUO copies the number in the left of `.JBSA` to the word called `.JBFF`.

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, 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 above this address.[1] Generally, every address at and above the word addressed by the left half of `.JBSA` is available to a program. However, there are some boundaries where care must be taken. No address in a program can be higher than `777777`.[2]

For this program we use the contents of `.JBFF` to determine the location in memory into which we will read the UFD. When we determine how much memory the UFD will occupy, we will update `.JBFF` to point beyond the memory space that the UFD uses.

It should be noted that in TOPS-10 the program must also negotiate with the operating system to expand its memory size. Thus, if we determine that the UFD does not fit in the space that is already allocated to the program, the program will request the use of additional memory by means of the `CORE` MUUO. The program's highest legal address can be found in `.JBREL`; if this number is not large enough, the program will request additional memory.

---

[1] Some programs that were written for TOPS-10 have partitioned their address space into two regions called *segments*. If you have to deal with one of these programs, avoid expanding the free space into the addresses at and above `400000`; in a two-segment program, the left half of `.JBSA` contains the first free address in the lower segment.

[2] Except in the unlikely event that TOPS-10 is made to support extended addressing.

23.3. 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 23.6.

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.

23.4. 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.

```
        TITLE   Directory I/O and Bubble Sort - Example 14

        EXTERN  .JBFF, .JBREL

A=1                                     ;Accumulator Definitions
B=2
C=3
D=4
W=5
X=6
Y=7
Z=10
P=17

OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ P,]

UFDCHN==0                               ;Channel for reading the UFD
LOOKCH==1                               ;Channel to LOOKUP each file
MFDPPN==<1,,1>                          ;ppn of master file directory
PDLEN==200

PDLIST: BLOCK   PDLEN                   ;stack
THSUSR: 0                               ;PPN of current user
UFDPTR: 0                               ;IOWD pointer to UFD in memory
LOOKBL: BLOCK   4                       ;data block for LOOKUP
CRLF:   BYTE(7)15,12

;Main program
START:  RESET                           ;normalize all I/O activity
        MOVE    P,[IOWD PDLEN,PDLIST]   ;establish a stack
        OUTSTR  [ASCIZ/Directory and Sort
/]
        CALL    RDUFD                   ;Open Channel, read the UFD.
        SKIPL   UFDPTR                  ;Non-empty UFD?
        JRST    STOP                    ;UFD is empty
        CALL    CMPRES                  ;Compress any holes
        SKIPL   UFDPTR                  ;any non-holes in the UFD?
        JRST    STOP                    ;UFD has no files in it
        CALL    SORT                    ;Sort the UFD
        CALL    PRINT                   ;print the results
STOP:   EXIT

;OPEN a DSK channel for Dump Mode input
RDUFD:  MOVEI   A,17                    ;Dump Mode (unbuffered)
        MOVSI   B,'DSK'                 ;Device DSK
        MOVEI   C,0                     ;No buffer control blocks
        OPEN    UFDCHN,A                ;Obtain a channel.
        JRST    NOOPEN                  ;Can't do it.
;LOOKUP the UFD for the current user
        GETPPN  A,                      ;Obtain PPN of current user.
        JFCL                            ;Sometimes GETPPN can skip
        MOVEM   A,THSUSR                ;remember the name of this user
        MOVSI   B,'UFD'                 ;get the file extension
        MOVEI   C,0
        MOVE    D,[MFDPPN]              ;PPN of the master file directory
        LOOKUP  UFDCHN,A                ;Seek file
        JRST    NOUFD                   ;can't find the UFD?
;Obtain the size, in words, of this UFD
        JUMPGE  D,RDUFD0                ;Jump if file length is a block count
        MOVS    D,D                     ;unswap the swapped negative word count
        MOVN    D,D                     ;D gets the positive wcount
        JRST    RDUFD1
```

```
RDUFD0: LSH     D,7                     ;shift up to make a word count.
;Here, D is the word count of the UFD.   Make sure we have enough core
RDUFD1: MOVE    A,.JBFF                 ;get the present free-core address
        MOVE    B,D                     ;Amount of memory needed
        ADDB    B,.JBFF                 ;This is the next free address
;If we need more memory than we have allocated, ask the system for more
        CAMG    B,.JBREL                ;does it fit in our allocation?
        JRST    RDUFD2                  ;yes, no need to ask for more
        CORE    B,                      ;request more memory
        JRST    NOCORE                  ;request denied
;We now have enough memory to hold the entire UFD.  Build an IOWD in A
;that describes the location and size of the region to read into.
;Set B to zero to terminate the dump-mode command list.  Return
;in UFDPTR the IOWD for the data area.
RDUFD2: MOVN    D,D                     ;-wc
        SUBI    A,1                     ;decrement original .JBFF
        HRL     A,D                     ;-WC,,Address-1 is now in A.
        MOVEM   A,UFDPTR                ;save as aobjn pointer to UFD in mem
        MOVEI   B,0                     ;A and B are a dump-mode command list
        INPUT   UFDCHN,A                ;Read entire UFD into memory.
        STATZ   UFDCHN,740000           ;test for errors
        JRST    UFDIER                  ;UFD input error
        CLOSE   UFDCHN,
        RELEAS  UFDCHN,
CPOPJ:  RET

;Here to remove holes from the UFD.  The UFD is copied onto itself,
;omitting the holes in the UFD.   Then UFDPTR is modified to reflect
;the shorter actual size of the data.   Also, to correct an anomaly
;in the collating sequence due to using signed comparisons, the sign
;bit of the file name and extension is complemented.
CMPRES: MOVE    A,UFDPTR                ;Source of data
        MOVE    B,UFDPTR                ;destination of data
CMPRS1: SKIPN   C,1(A)                  ;Skip if there's a file name here
        JRST    CMPRS2                  ;no file.  compress space
        TLC     C,400000                ;fix collating sequence
        MOVEM   C,1(B)
        HLLZ    C,2(A)
        TLC     C,400000                ;fix collating sequence
        MOVEM   C,2(B)
        ADD     B,[2,,2]                ;increment PUT pointer
CMPRS2: ADD     A,[2,,2]                ;increment take pointer
        JUMPL   A,CMPRS1
        HLLZ    B,B                     ;shorten the word count
        MOVN    B,B                     ;accounting for holes that
        ADDM    B,UFDPTR                ;have been removed.
        RET
```

```
;Enter with UFDPTR being an IOWD pointer to UFD entries
SORT:   MOVE    A,UFDPTR                ;
        ADD     A,[2,,2]
        JUMPGE  A,CPOPJ
        MOVEI   C,0                     ;number of exchanges, so far
SORT1:  MOVE    D,-1(A)                 ;get a name
        CAMGE   D,1(A)                  ;compare to the next name
        JRST    SORT2                   ;this pair is ok.
        CAME    D,1(A)                  ;test for equal names
        JRST    SORTX                   ;this pair must be swapped
        MOVE    D,0(A)                  ;get an extension
        CAMG    D,2(A)                  ;compare to the next extension
        JRST    SORT2                   ;ok.  keep these.
SORTX:  ADDI    C,1                     ;count an exchange was made
        MOVE    D,-1(A)                 ;exchange names
        EXCH    D,1(A)
        MOVEM   D,-1(A)
        MOVE    D,0(A)                  ;exchange extensions
        EXCH    D,2(A)
        MOVEM   D,0(A)
SORT2:  ADD     A,[2,,2]
        JUMPL   A,SORT1                 ;loop thru all pairs of items
        JUMPG   C,SORT                  ;jump if any exchanges were done
        RET

;Enter here to print the result.
PRINT:  MOVEI   A,17                    ;Open a channel by which we
        MOVSI   B,'DSK'                 ;we can do LOOKUPs
        MOVEI   C,0
        OPEN    LOOKCH,A
        JRST    NOOPEN

;Print the first line of output.
        OUTSTR  [ASCIZ/Directory Listing for [/]
        HLRZ    A,THSUSR                ;Print PPN as two octal quantities
        CALL    PRNOCT                  ;
        OUTCHR  [","]
        HRRZ    A,THSUSR
        CALL    PRNOCT
        OUTSTR  [ASCIZ/] at /]
        DATE    A,                      ;get the current date
        CALL    PRNDAT                  ;and print it
        OUTSTR  [ASCIZ/ /]
        TIMER   A,                      ;time in ticks since midnight
        IDIVI   A,74*74                 ;convert to minutes since midnight
        CALL    PRNTIM                  ;print time as HH:MM
        OUTSTR  CRLF
        OUTSTR  CRLF
        MOVE    Z,UFDPTR                ;scan through the file names
PRINT1: MOVE    A,1(Z)
        MOVE    B,2(Z)
        TLC     A,400000                ;unscramble the collating seq
        TLC     B,400000
        MOVEI   C,0
        MOVE    D,THSUSR
        MOVE    W,[A,,LOOKBL]           ;copy the lookup data to
        BLT     W,LOOKBL+3              ;LOOKBL
        LOOKUP  LOOKCH,LOOKBL
        JRST    [CALL NOLOOK            ;Unusual to not find it.
                 JRST PRINT2]           ;print message and continue
        CALL    PRNFIL                  ;Print data about this file
PRINT2: CLOSE   LOOKCH,                 ;disconnect file from channel
        ADD     Z,[2,,2]                ;loop thru the UFD
        JUMPL   Z,PRINT1
        RET
```

```
;Here to print one line for a file.  LOOKUP information in LOOKBL
PRNFIL: MOVEI   C,0                     ;count letters printed on line
        MOVE    B,LOOKBL                ;file name
        CALL    SIXOUT                  ;print name
        HLLZ    B,LOOKBL+1              ;file extension
        JUMPE   B,PRNFL1                ;skip if zero
        ADDI    C,1                     ;print a period and count it
        OUTCHR  ["."]
        CALL    SIXOUT                  ;print the file extension
PRNFL1: MOVEI   A," "                   ;print spaces to fill to column
        OUTCHR  A                       ;16...
        CAIGE   C,20
        AOJA    C,.-2
        SKIPL   A,LOOKBL+3              ;file size indicator
        JRST    PRNFL2                  ;if positive, this is block count
        MOVS    A,A                     ;Unswap the swapped negative WC
        MOVN    A,A                     ;make positive WC
        ADDI    A,177                   ;round up to whole blocks
        LSH     A,-7                    ;file size in blocks
PRNFL2: MOVEI   C,6                     ;print size in blocks
        CALL    DECFIL                  ;in six columns
        OUTSTR  [ASCIZ/  </]            ;Print the file protection
        LDB     A,[POINT 3,LOOKBL+2,2]  ;maybe this is the hard way
        ADDI    A,"0"                   ;but it's a repetition of some
        OUTCHR  A                       ;simple stuff
        LDB     A,[POINT 3,LOOKBL+2,5]
        ADDI    A,"0"
        OUTCHR  A
        LDB     A,[POINT 3,LOOKBL+2,8]
        ADDI    A,"0"
        OUTCHR  A
        OUTSTR  [ASCIZ/>  /]
        LDB     A,[POINT 4,LOOKBL+2,12] ;file mode word
        CAIGE   A,10
        OUTCHR  [" "]
        CALL    PRNOCT
        OUTSTR  [ASCIZ/  /]
        LDB     B,[POINT 3,LOOKBL+1,20] ;Assemble the file write date
        LDB     A,[POINT 12,LOOKBL+2,35]
        DPB     B,[POINT 3,A,23]
        CALL    PRNDAT                  ;file write date
        OUTSTR  [ASCIZ/  /]
        LDB     A,[POINT 11,LOOKBL+2,23] ;write time
        CALL    PRNTIM                  ;print time as minutes since midnight
        OUTSTR  CRLF
        RET

;print a TOPS-10 format date word
PRNDAT: IDIVI   A,37                    ;day-1 in B
        PUSH    P,A                     ;save month and year
        MOVEI   A,1(B)                  ;day number to A
        CALL    PRDEC2                  ;print day of month as 2 characters
        OUTCHR  ["-"]                   ;a hyphen
        POP     P,A                     ;get month/year
        IDIVI   A,14                    ;month number -1 to B
        OUTSTR  MONTAB(B)               ;print the name of the month and -
        ADDI    A,^D1964                ;print the year number
        CALL    PRNDEC
        RET
```

```
MONTAB: ASCIZ/Jan-/                     ;Table of month names for PRNDAT
        ASCIZ/Feb-/
        ASCIZ/Mar-/
        ASCIZ/Apr-/
        ASCIZ/May-/
        ASCIZ/Jun-/
        ASCIZ/Jul-/
        ASCIZ/Aug-/
        ASCIZ/Sep-/
        ASCIZ/Oct-/
        ASCIZ/Nov-/
        ASCIZ/Dec-/

PRDEC2: CAIGE   A,12                    ;print two characters of decimal
        OUTCHR  ["0"]
PRNDEC: IDIVI   A,12                    ;decimal printer.  counts
        PUSH    P,B                     ;characters printed in C
        SKIPE   A
        CALL    PRNDEC
        POP     P,A
        ADDI    A,"0"
        OUTCHR  A
        ADDI    C,1
        RET

DECFIL: IDIVI   A,12                    ;decimal print with leading fill
        PUSH    P,B                     ;call with number in A, column
        SUBI    C,1                     ;count in C
        JUMPE   A,DECFL2
        CALL    DECFIL
DECFL0: POP     P,A
        ADDI    A,"0"
        OUTCHR  A
        RET

DECFL2: JUMPLE  C,DECFL0
        OUTCHR  [" "]
        SOJA    C,DECFL2

PRNOCT: IDIVI   A,10                    ;octal printer
        PUSH    P,B
        SKIPE   A
        CALL    PRNOCT
        POP     P,A
        ADDI    A,"0"
        OUTCHR  A
        ADDI    C,1
        RET

SIXOUT: JUMPE   B,CPOPJ                 ;sixbit printer.  B/data
        MOVEI   A,0
        LSHC    A,6
        ADDI    A," "
        OUTCHR  A
        AOJA    C,SIXOUT

PRNTIM: IDIVI   A,74                    ;minutes in B, hours in A
        PUSH    P,B                     ;save minutes
        CALL    PRDEC2                  ;print hours in 2 columns
        OUTCHR  [":"]                   ;a colon
        POP     P,A                     ;get minutes
        JRST    PRDEC2                  ;print minutes
```

```
NOLOOK: OUTSTR  [ASCIZ/Lookup failed to find /]
        MOVE    B,LOOKBL
        CALL    SIXOUT
        HLLZ    B,LOOKBL+1
        SKIPE   B
        OUTCHR  ["."]
        CALL    SIXOUT
        OUTSTR  CRLF
        RET

;Error handling
NOOPEN: OUTSTR  [ASCIZ/Can't OPEN a channel for device DSK:
/]
        EXIT

NOUFD:  OUTSTR  [ASCIZ/Can't LOOKUP the UFD
/]
        EXIT

NOCORE: OUTSTR  [ASCIZ/Can't get enough core
/]
        EXIT

UFDIER: OUTSTR  [ASCIZ/UFD input error
/]
        EXIT

        END     START
```

23.5. 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.

```
START:  RESET                           ;normalize all I/O activity
        MOVE    P,[IOWD PDLEN,PDLIST]   ;establish a stack
        OUTSTR  [ASCIZ/Directory and Sort
/]
```

The main program continues by calling the RDUFD subroutine which reads the entire UFD into memory. RDUFD returns a description of where the UFD was placed in memory and how large it is. This description is in the form of an IOWD pointer, i.e., a negative word count in the left halfword and the address, minus one, in the right halfword. This pointer is returned in the word UFDPTR. If UFDPTR is positive, the UFD was empty and the program exits.

```
        CALL    RDUFD                   ;Open Channel, read the UFD.
        SKIPL   UFDPTR                  ;Non-empty UFD?
        JRST    STOP                    ;UFD is empty
```

Next, any holes in the UFD are removed. These holes are zero words corresponding to places where files once existed in the UFD. When a file is deleted, a zero is written in the UFD, obliterating the file name. The holes in the UFD are removed by the CMPRES routine. CMPRES returns an updated version of UFDPTR. If that version is positive, the UFD is empty and the program stops.

```
        CALL    CMPRES                  ;Compress any holes
        SKIPL   UFDPTR                  ;any non-holes in the UFD?
        JRST    STOP                    ;UFD has no files in it
```

Finally, the compressed version of the UFD is sorted by the SORT subroutine, which re-orders the files alphabetically by name and extension. The results are printed by PRINT, and the program stops.

```
        CALL    SORT                    ;Sort the UFD
        CALL    PRINT                   ;print the results
STOP:   EXIT
```

We will now examine each of these subroutines in detail.

23.5.1. RDUFD

The RDUFD routine accomplishes several tasks:

- It opens a disk channel for dump mode input.
- It performs a LOOKUP on the current user's user file directory.
- It determines the size of the UFD and allocates memory space for it, expanding memory if necessary.
- It reads the entire UFD into memory.
- It closes the file and releases the channel.

The channel named UFDCHN is opened in an unbuffered mode, mode 17. Because this is an unbuffered mode, buffer header control blocks need not be allocated for the channel. Thus, register C is zero for this OPEN.

```
RDUFD:  MOVEI   A,17                    ;Dump Mode (unbuffered)
        MOVSI   B,'DSK'                 ;Device DSK
        MOVEI   C,0                     ;No buffer control blocks
        OPEN    UFDCHN,A                ;Obtain a channel.
        JRST    NOOPEN                  ;Can't do it.
```

Next, the UFD must be selected for input by means of a LOOKUP MUUO. The file name of the UFD is the same as the PPN of the user. This program executes a GETPPN MUUO to obtain the PPN of the current user. The PPN is returned in the specified accumulator.

The file extension UFD is required in register B. The PPN [1,1], written as <1,,1> is loaded into register D and the LOOKUP is performed:

```
        GETPPN  A,                      ;Obtain PPN of current user.
        JFCL                            ;Sometimes GETPPN can skip
        MOVEM   A,THSUSR                ;remember the name of this user
        MOVSI   B,'UFD'                 ;get the file extension
        MOVEI   C,0
        MOVE    D,[MFDPPN]              ;PPN of the master file directory
        LOOKUP  UFDCHN,A                ;Seek file
        JRST    NOUFD                   ;can't find the UFD?
```

Following a successful LOOKUP, register D will contain an indicator of the file size. We expect that D will be negative, signifying a negative word count in the left halfword. If D happens to be positive, then D is a block count; the block count is multiplied by 128 (by means of a shift) to convert it to a word count:

```
        JUMPGE  D,RDUFD0                ;Jump if file length is a block count
        MOVN    D,D                     ;D gets the positive, swapped wcount
        MOVS    D,D                     ;unswap the word count
        JRST    RDUFD1

RDUFD0: LSH     D,7                     ;shift up to make a word count.
RDUFD1:
```

When the program arrives at RDUFD1, register D contains the positive word count of the file. This is added to the word found in .JBFF to obtain a new value for .JBFF. The old value of .JBFF is retained as the first address available for storing the data from the UFD. The new value of .JBFF is compared to .JBREL. The value in .JBREL reflects the actual amount of memory allocated to this program. If the new value of .JBFF exceeds the value in .JBREL then the UFD requires more room than is presently available in the memory space allocated to the program. By means of a CORE MUUO, more memory space is requested:

```
RDUFD1: MOVE    A,.JBFF                 ;get the present free-core address
        MOVE    B,D                     ;Amount of memory needed
        ADDB    B,.JBFF                 ;This is the next free address
        CAMG    B,.JBREL                ;does it fit in our allocation?
        JRST    RDUFD2                  ;yes, no need to ask for more
        CORE    B,                      ;request more memory
        JRST    NOCORE                  ;request denied
RDUFD2:
```

The program arrives at RDUFD2 with enough memory allocated so that the UFD will fit in. Register D contains the positive word count; this is negated. Register A contains the original value of .JBFF; this is decreased by one and the negative word count is copied to the left half of A. The result in A is an IOWD: a negative word count and a memory address minus one. This result is also saved as UFDPTR:

```
RDUFD2: MOVN    D,D                     ;-wc
        SUBI    A,1                     ;decrement original .JBFF
        HRL     A,D                     ;-WC,,Address-1 is now in A.
        MOVEM   A,UFDPTR                ;save as aobjn pointer to UFD in mem
```

In unbuffered input and output, the transfer of information is controlled by a *dump-mode command list.* The command list consists of words in memory. Except for the jumps, as explained below, the command list occupies consecutive words in memory. There are three kinds of words in the command list:

- IO command words. These have a negative left half which is a word count. The right half is the memory address, minus one, to or from which the transfer takes place.
- Jump words. These have a zero left half and a non-zero right half. TOPS-10 interprets this word as meaning that the command list continues at the address specified in the right half of the jump word.
- Halt word. This word is entirely zero.

Most of the time, a dump mode command list consists of just one IOWD and a zero to end the list. Although a dump mode command list can be located anywhere in the memory space of the program, we find it convenient to locate the list in the two consecutive accumulators A and B. In this case, A already contains the IOWD. Register B is set to zero to stop the list.

An INPUT MUUO is performed in which the effective address specifies the location of the first word in the command list:

```
        MOVEI   B,0                     ;A and B are a dump-mode command list
        INPUT   UFDCHN,A                ;Read entire UFD into memory.
```

The usual test of I/O status is made. The channel is closed and released. The RDUFD routine returns:

```
        STATZ   UFDCHN,740000           ;test for errors
        JRST    UFDIER                  ;UFD input error
        CLOSE   UFDCHN,
        RELEAS  UFDCHN,
CPOPJ:  RET
```

23.5.2. CMPRES

The CMPRES routine recopies the UFD data to lower addresses. We begin by setting registers A and B from UFDPTR. In this routine, register A is the *take pointer*: data is copied from the words addressed by A. Register B is the *put pointer*: the words addressed by A are copied to the area addressed by B. Register A advances by two for each file name that is processed. Register B advances when a non-zero name is stored:

```
CMPRES: MOVE    A,UFDPTR            ;Source of data
        MOVE    B,UFDPTR            ;destination of data
CMPRS1: SKIPN   C,1(A)              ;Skip if there's a file name here
        JRST    CMPRS2              ;no file.  compress space

        . . .                       ;copy data to area addressed by B

        ADD     B,[2,,2]            ;increment PUT pointer
CMPRS2: ADD     A,[2,,2]            ;increment TAKE pointer
        JUMPL   A,CMPRS1
```

When the data is copied, it also is changed. Bit 0 in both the file name and extension words is complemented before being stored. The reason for complementing the sign bit goes back to the representation of negative numbers: in a two's complement machine, the bit weight of bit 0 is the negative of what would otherwise be present. The negative weight of bit 0 creates some anomalies when the data represents characters instead of numbers. Without making this change to bit 0, the following relationships hold:

```
Sixbit/A/  = 410000,,0  <  Sixbit/0/  = 200000,,0
Sixbit/A0/ = 412000,,0  <  Sixbit/AA/ = 414100,,0
```

When a sixbit name begins with a letter, that name is a negative number. So a name that begins with a letter sorts before a name that starts with a digit. However, when a digit appears as the second or subsequent character in a name, the digit sorts before a letter. To remedy this problem, the program complements bit 0 before doing the sort. In effect, we are making the sort perform *unsigned comparisons*. On the whole, this is a desirable effect and it eliminates the anomaly mentioned above. After complementing, the results are more consistent:

```
Modified Sixbit/A/  = 010000,,0  >  Modified Sixbit/0/  = 600000,,0
Modified Sixbit/AA/ = 014100,,0  >  Modified Sixbit/A0/ = 012000,,0
```

The complement is performed very simply in the loop at CMPRS1:

```
        TLC     C,400000            ;fix collating sequence
        MOVEM   C,1(B)
        HLLZ    C,2(A)
        TLC     C,400000            ;fix collating sequence
        MOVEM   C,2(B)
```

It will be important to complement the name and extension before trying to print them. We shall see that this is done in PRINT.

Finally, CMPRES must return an updated descriptor in UFDPTR. This is accomplished by reducing the magnitude of the negative word count in the left half of UFDPTR. The amount of the reduction is determined by the unused word count present in the left half of register B. This count is made positive and added to the count in the left half of UFDPTR; adding a positive number to a negative count reduces the magnitude of the count:

```
        HLLZ    B,B                     ;shorten the word count
        MOVN    B,B                     ;accounting for holes that
        ADDM    B,UFDPTR                ;have been removed.
        RET
```

23.5.3. SORT

The main program calls SORT with UFDPTR containing a description of how many data items to sort and the location of the first of them. Bubble sort compares adjacent data elements. If any pair of elements is out of order, the program will interchange them. Let N denote the number of data items. 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.

First, SORT adds [2,,2] to the contents of UFDPTR. If the result is positive there is only one item in the list and SORT returns immediately, as one item is already sorted.

```
SORT:   MOVE    A,UFDPTR                ;Data descriptor
        ADD     A,[2,,2]                ;increase to address 2nd item
        JUMPGE  A,CPOPJ                 ;if no second, return now.
```

Usually, we are not given such an easy case. Register C is loaded with zero to signify that no exchanges have been made yet. Register A should be though of as a pointer to the second data item. The second file name is found in the word addressed by 1(A); the first file name is found at -1(A). The loop at SORT1 will generally compare two adjacent elements. If the comparison indicates that the elements are out of order, they will be interchanged. Register A will run from the second to the last element of the array.

At SORT1 the program determines if two adjacent elements are in order. If the first name is less than the second, then the pair is properly ordered; the program jumps to SORT2. If the first is greater than the second, they are out of order; the program jumps to SORTX where the pair will be interchanged. If the file names are equal, the program compares file extensions. Again, if the pair is found to be properly ordered, the program jumps to SORT2; if the pair must be exchanged, the program arrives at SORTX:

```
        MOVEI   C,0                     ;number of exchanges, so far
SORT1:  MOVE    D,-1(A)                 ;get a name
        CAMGE   D,1(A)                  ;compare to the next name
        JRST    SORT2                   ;this pair is ok.
        CAME    D,1(A)                  ;test for equal names
        JRST    SORTX                   ;this pair must be swapped
        MOVE    D,0(A)                  ;get an extension
        CAMG    D,2(A)                  ;compare to the next extension
        JRST    SORT2                   ;ok.  keep these.
SORTX:  . . .

SORT2:  ADD     A,[2,,2]
        JUMPL   A,SORT1                 ;loop thru all pairs of items
        JUMPG   C,SORT                  ;jump if any exchanges were done
        RET
```

Each time the program arrives at SORT2, the constant [2,,2] is added to A. If the result is negative, the program jumps to SORT1 to compare another pair. The ADD and JUMPL instructions have an effect similar to AOBJN; the program needs to advance by two positions (the name and the extension) each time, so two instructions are used to generalize AOBJN.

The count in A becomes positive after the next-to-last and the last elements have been compared. This signals the end of a pass. When a pass ends, if register C is non-zero, another pass is commenced by jumping

back to SORT. The program continues repeating these passes through the data until a pass ends in which C is zero. This signals that every pair of elements is properly arranged; therefore, the entire array is properly sorted.

The exchange of two data items takes place at SORTX. First, the exchange is counted in register C. Then the data items are swapped by means of the EXCH instructions:

```
SORTX:  ADDI    C,1                     ;count an exchange was made
        MOVE    D,-1(A)                 ;exchange names
        EXCH    D,1(A)
        MOVEM   D,-1(A)
        MOVE    D,0(A)                  ;exchange extensions
        EXCH    D,2(A)
        MOVEM   D,0(A)
```

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)*N/2 comparisons. Such a sort is said to be of *order* N^2 *work*. The N^2 stems from the fact that for large N, the size of the quantity (N-1)*N/2 is dominated by growth of the N^2 portion. It turns out, as we shall demonstrate in the next example, that N^2 is much too high a price to pay for sorting.

23.5.4. PRINT

The PRINT routine prints information about each of the files in the directory. Since the information is now sorted, the list that PRINT produces will likewise be sorted.

PRINT begins by opening another channel in dump mode. We are not planning to do any input or output on this channel, but we need the channel in order to perform LOOKUP MUUOs which we use to obtain further information about each file.

```
PRINT:  MOVEI   A,17                    ;Open a channel by which we
        MOVSI   B,'DSK'                 ;we can do LOOKUPs
        MOVEI   C,0
        OPEN    LOOKCH,A
        JRST    NOOPEN
```

Next, the program prints a heading that describes what information is about to follow. In order to keep this example simple, we'll confine ourselves to terminal output via OUTSTR and OUTCHR. The program prints the message "Directory Listing", the project and programmer number of the current user, and the date and time.

The PPN is printed by two calls to PRNOCT, an octal printing routine. First the left half of THSUSR is printed, then a comma, and then the programmer number from the right half of THSUSR.

The current date is obtained via the DATE MUUO. The date is returned in the specified register; in this case, that is A. The PRNDAT prints the date. The current time, measured in 60ths of a second since midnight is obtained via the TIMER MUUO. That number is divided by decimal 3600 to produce minutes since midnight. This is printed by PRNTIM, and a pair of CRLFs is written to end the line and to separate the heading from the output.

```
;Print the first line of output.
        OUTSTR  [ASCIZ/Directory Listing for [/]
        HLRZ    A,THSUSR                ;Print PPN as two octal quantities
        CALL    PRNOCT                  ;
        OUTCHR  [","]
        HRRZ    A,THSUSR
        CALL    PRNOCT
        OUTSTR  [ASCIZ/] at /]
        DATE    A,                      ;get the current date
        CALL    PRNDAT                  ;and print it
        OUTSTR  [ASCIZ/ /]
        TIMER   A,                      ;time in ticks since midnight
        IDIVI   A,74*74                 ;convert to minutes since midnight
        CALL    PRNTIM                  ;print time as HH:MM
        OUTSTR  CRLF
        OUTSTR  CRLF
```

Now comes the interesting part. Register Z is loaded with UFDPTR, the pointer to the array of sorted file names and extensions. The loop at PRINT1 loads register A with a file name, and register B with a file extension. Bit 0 in both registers A and B are then complemented, restoring the original names. Register C is loaded with a zero and D gets a copy of the project programmer name. This information is copied, by means of a BLT instruction, to the memory block called LOOKBL. A LOOKUP MUUO is then peformed using LOOKBL as the argument block. Assuming that the LOOKUP is successful, the program calls PRNFIL to print the information about one file. At PRINT2 the channel is closed, register Z is advanced, and if there is another file, the program jumps back to PRINT1 to process the next file.

```
        MOVE    Z,UFDPTR                ;scan through the file names
PRINT1: MOVE    A,1(Z)
        MOVE    B,2(Z)
        TLC     A,400000                ;unscramble the collating seq
        TLC     B,400000
        MOVEI   C,0
        MOVE    D,THSUSR
        MOVE    W,[A,,LOOKBL]           ;copy the lookup data to
        BLT     W,LOOKBL+3              ;LOOKBL
        LOOKUP  LOOKCH,LOOKBL
        JRST    [CALL NOLOOK            ;Unusual to not find it.
                 JRST PRINT2]           ;print message and continue
        CALL    PRNFIL                  ;Print data about this file
PRINT2: CLOSE   LOOKCH,                 ;disconnect file from channel
        ADD     Z,[2,,2]                ;loop thru the UFD
        JUMPL   Z,PRINT1
        RET
```

PRNFIL prints one line for each file. We begin by printing the file name and extension, as found in LOOKBL. In this fragment, register C counts the number of characters printed. This allows us to fill each line to the same column position, to obtain a pleasing listing:

```
;Here to print one line for a file.  LOOKUP information in LOOKBL
PRNFIL: MOVEI   C,0                     ;count letters printed on line
        MOVE    B,LOOKBL                ;file name
        CALL    SIXOUT                  ;print name
        HLLZ    B,LOOKBL+1              ;file extension
        JUMPE   B,PRNFL1                ;skip if zero
        ADDI    C,1                     ;print a period and count it
        OUTCHR  ["."]
        CALL    SIXOUT                  ;print the file extension
PRNFL1: MOVEI   A," "                   ;print spaces to fill to column
        OUTCHR  A                       ;16...
        CAIGE   C,20
        AOJA    C,.-2
```

The word in LOOKBL+3 contains an indication of the file size. This indication is converted (if necessary) to a block count. The file size in blocks is then printed:

```
        SKIPL   A,LOOKBL+3              ;file size indicator
        JRST    PRNFL2                  ;if positive, this is block count
        MOVN    A,A                     ;make positive swapped wc
        MOVS    A,A                     ;WC into right half
        ADDI    A,177                   ;round up to whole blocks
        LSH     A,-7                    ;file size in blocks
PRNFL2: MOVEI   C,6                     ;print size in blocks
        CALL    DECFIL                  ;in six columns
```

Next, the protection and mode fields from LOOKBL+2 are printed. The methods used are crude but effective.

```
        OUTSTR  [ASCIZ/  </]            ;Print the file protection
        LDB     A,[POINT 3,LOOKBL+2,2]  ;maybe this is the hard way
        ADDI    A,"0"                   ;but it's a repetition of some
        OUTCHR  A                       ;simple stuff
        LDB     A,[POINT 3,LOOKBL+2,5]
        ADDI    A,"0"
        OUTCHR  A
        LDB     A,[POINT 3,LOOKBL+2,8]
        ADDI    A,"0"
        OUTCHR  A
        OUTSTR  [ASCIZ/>  /]
        LDB     A,[POINT 4,LOOKBL+2,12] ;file mode word
        CAIGE   A,10                    ;skip if it prints in two columns
        OUTCHR  [" "]                   ;small num: add a leading space.
        CALL    PRNOCT
        OUTSTR  [ASCIZ/  /]
```

The date when the file was written is scattered in LOOKBL+1 and LOOKBL+2. The pieces are brought together and PRNDAT prints the file write date. The file write time and end of line are printed also.

```
        LDB     B,[POINT 3,LOOKBL+1,20] ;Assemble the file write date
        LDB     A,[POINT 12,LOOKBL+2,35]
        DPB     B,[POINT 3,A,23]
        CALL    PRNDAT                  ;file write date
        OUTSTR  [ASCIZ/  /]
        LDB     A,[POINT 11,LOOKBL+2,23] ;write time
        CALL    PRNTIM                  ;print time as minutes since midnight
        OUTSTR  CRLF
        RET
```

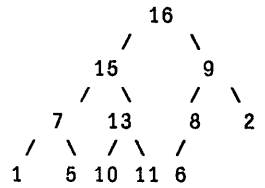
23.6. HEAPSORT

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*Log(N) operations. Since Log(N) is much smaller than N (and grows much more slowly than N) the quantity N*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.* A tree is a collection of data elements. 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 123). 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 below:



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.

23.6.1. 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*K and 2*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.

23.6.2. 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

[3]When we speak of N/2 in this discussion, we mean the integer portion of the quotient, as the IDIV instruction would compute.

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*X or 2*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].

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: 192837465.

Since there are nine items, the 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 the next items will be placed in the heap.

```
remaining: 1928
                    *
                  *   *
                *   3   7   4
               6 5
```

Working backwards through the series of remaining items, 1, 9, 7, 8, we add 8 to the fourth position in the heap:

```
remaining: 192
                    *
                  *   *
                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: 19
                    *
                  *   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: 19
                    *
                  *   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 are 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.

23.6.3. 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. 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.

23.6.4. 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.

23.6.5. 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.
```

23.6.6. 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 memory locations, N, HEAPX, and HEAPY must be defined also. A further complication arises as indices generally need to be doubled to account for the storage of two words per data item.

```
        SUBTTL  Heapsort Subroutine - Example 15

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]

;Call SORT with UFDPTR = array descriptor in IOWD format.

SORT:   HLRO    A,UFDPTR        ;-1,,-WC into A
        ASH     A,-1            ;convert -wc to negative item count
        MOVNM   A,N             ;Save the (positive) number of data items
        HRRZ    Y,UFDPTR        ;Address of the first item, -1
        SUBI    Y,1             ;Offset for 1-origin indexing
        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
;HEAPX is set up so that if an element number *2 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             ;HEAP[1] and HEAP[Old N] to exchange.
        CALL    HEAPEX          ;Exchange items HEAP[1] and HEAP[Y]
        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.
        CALL    HEAPEX          ;Exchange HEAP[X] with HEAP[Y]
        MOVE    X,Y             ;Copy Y to X.  Make sure that
        JRST    ENHEAP          ;  HEAP[Y] 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.
        CALL    COMPXY          ;Now compare HEAP[X] and HEAP[Y]
        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)
        CALL    COMPXY          ;compare items 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].
        PUSH    P,X             ;Save X.
        LSH     X,1             ;First comparand is HEAP[2*X]
        CALL    COMPXY          ;Other is HEAP[2*X+1].  Skip if [2*X] is larger
        SKIPA                   ;HEAP[2*X+1] is larger.  Y has 2*X+1
        SUBI    Y,1             ;HEAP[2*X] is larger.  Return 2*X in Y
        POP     P,X             ;return X unscathed.
        JRST    CPOPJ1          ;

;Compare HEAP[X] to HEAP[Y].  Skip if HEAP[X] is larger than HEAP[Y]
COMPXY: LSH     X,1             ;form index to two-word things
        LSH     Y,1
        MOVE    B,@HEAPX        ;B:=file name from HEAP[X]
        CAMGE   B,@HEAPY        ;Compare to file name at HEAP[2*X]
        JRST    CMPRET          ;HEAP[Y] is larger than HEAP[X]
        CAME    B,@HEAPY        ;are the names equal?
        JRST    CMPRT1          ;no.  Return quick
        TRO     X,1
        TRO     Y,1
        MOVE    B,@HEAPX        ;compare extensions
        CAML    B,@HEAPY        ;Skip if HEAP[X] is smaller
CMPRT1: AOS     (P)             ;HEAP[Y] is smaller than HEAP[X]
CMPRET: LSH     X,-1
        LSH     Y,-1
        RET

;Exchange HEAP[X] with HEAP[Y]
HEAPEX: LSH     X,1             ;Double the given indices.
        LSH     Y,1             ;
        MOVE    B,@HEAPY        ;Exchange HEAP[X] with HEAP[Y].
        EXCH    B,@HEAPX
        MOVEM   B,@HEAPY
        TRO     X,1             ;advance to the second word of HEAP[X]
        TRO     Y,1             ;advance to the second word of HEAP[Y]
        MOVE    B,@HEAPY        ;Exchange HEAP[X] with HEAP[Y].
        EXCH    B,@HEAPX
        MOVEM   B,@HEAPY
        LSH     X,-1
        LSH     Y,-1
        RET
```

23.6.7. 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`. The word count in `UFDPTR` is halved and the positive item count is stored in `N`, the number of items to sort. The next section is a bit tricky. The right half of `UFDPTR` contains one less than the address of the first data item. If we call the first data item `HEAP[1]`, and if we recall that each data item is really two words long, then the address of `HEAP[0]` is two less than the address of `HEAP[1]`. Therefore, the address of `HEAP[0]` is one less than the right half of `UFDPTR`. We load the right half of `UFDPTR` into register `Y` and subtract one; this is the address of the non-existent `HEAP[0]` word pair. The Heapsort algorithm compells us to locate the first data item in `HEAP[1]`; having the address of `HEAP[0]` will help us do indexing.

The left half of `Y` is then set to `X`, the name of an index register; this quantity is stored in `HEAPX`. `HEAPX`

will be used to address data items in the heap. Suppose X contains a doubled item number (item numbers are between 1 and N; doubled item numbers range from 2 to 2*N) of some item in the heap. Then the address expression @HEAPX will fetch the (first word of the) data item. When the computer sees the address expression @HEAPX, 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]; HEAPX also includes X in the index field. Therefore, the contents of X will be added to the address of HEAP[0] to form the address of HEAP[X], the desired item. The location HEAPY is set up in a similar fashion, but register Y appears in HEAPY as the index register. A reference to @HEAPY, when Y contains a doubled index, will obtain the item from HEAP[Y].

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.

Now, 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 HEAP[X] and the first son HEAP[Y] are compared. Comparisons are accomplished in the COMPXY routine, which worries about such things as doubling the index, and looking at the file extension if the file names are identical.

If the first son is larger than the father, MOVSEL will set Y to address the second son and jump to MOVSL1 to determine which of the two sons is largest. If the first son is smaller than the father, the program checks to see if there is a second son. If Y, the index of the first son, is less than N then the second son exists. If there is no second son, MOVSEL exits without skipping; there was only one son and it was smaller than its father.

If there is a second son, its index number, one larger than the index number of the first son, is computed

in Y. The second son is compared with the father. If the second son is larger, the routine returns with a skip; otherwise it returns without skipping. If this skip return is taken, Y contains the item number of the second son, which is the larger son.

At MOVSL1 we know the first son is larger than the father. We enter MOVSL1 with Y containing the index number of the second son. If Y greater than N, then there is only one son; Y is decremented to point to the first son and MOVSL1 returns with a skip. When there are two sons, they must be compared so we can return with Y set to the index number of the larger son.

Register X is saved on the stack. The value 2*X is loaded into X; Y still has 2*X+1. The two sons of X are compared. If HEAP[2*X] is larger than HEAP[2*X+1] then Y is decremented. The original value of X is restored from the stack and the routine returns with a skip.

23.6.8. Timing Analysis of Heapsort

We said earlier that on the order of N*Log(N) operations are necessary in the Heapsort algorithm. This number is result of some complicated analysis that we shall not repeat here. The rough outline of the analysis is straightforward and is sufficient for the present purpose.

The depth of a compact binary tree, such as the heap, is given by the integer part of the expression $1+Log_2(N)$, where N is the number of data items in the heap. When the ENHEAP subroutine is called to install a new item 1 in the heap, no more than $Log_2(N)$ comparisons and interchanges are necessary to place the item properly. Since ENHEAP is called N times to remove the sorted data from the heap, the total amount of work done is limited to $N*Log_2(N)$ comparisons. A similar approach can be taken to the analysis of building the heap. The ENHEAP routine is called N/2 times, each time doing no more than $Log_2(N)$ comparisons.

With the exception of multiplicative and additive constants, the amount of work performed by Heapsort is bounded by the expression $N*Log_2(N)$. Heapsort is said to be of order N*Log(N) time.[4]

23.7. EXERCISES

23.7.1. Cryptogram Program

Write a program to decode a simple substitution cypher.

In a substitution cypher, each letter of the unencrypted, "clear" text is changed to some particular letter in the encoded text.

For example, all "E" characters may become "Q" characters in the coded message.

Generally, the way to decode such a message is to count letter frequencies in the encoded message. English text follows particular patterns. The most frequent letter in usual text is "E". The letters "ETAIONSHRDLU" (which can also appear by jamming a linotype machine) are often considered to be the twelve most frequent letters in English, in descending order by frequency.

What your program has to do is

- Read the file specified by your instructor.

---

[4]The subscript 2 is dropped from the logarithm because that represents only a multiplicative constant.

- The first line contains the decryption key, i.e., the actual most frequent letter, second most frequent letter, third most frequent letter, etc., in sequence. This line contains twenty-six letters, followed by carriage return and line feed.
- The remainder of this file is the secret message. Read it; the input text will be less than 5000 characters in length. Count the occurrences of each character. Decrypt the text by changing the most frequent encoded character to the first character of the key string. The second most frequent character should be changed to the second character of the key string, etc. It will not be the case that two characters have exactly the same occurrence counts, except, zero occurrence counts are possible.
- Write a file that contains
    - The key string.
    - The encrypted text.
    - Each encrypted letter and its occurrence count, sorted to show the most frequent letter first. Although Bubble sort can be extremely slow in some cases, it would be suitable for sorting twenty-six items. Of course, any other sort you care to use is also acceptable.
    - The decrypted text.
- Both the upper-case and lower-case versions of one encoded letter will decode to the same letter. However, if the encrypted character is upper-case, then the decoding process should produce an upper-case character. Similarly a lower-case character in the encoded version should decode to a lower-case character. That is, if "Q" is decoded to "E" then "q" is decoded to "e".
- All spacing, punctuation, and line boundaries should be left as they appear in the coded text.

A sample input file follows:

```
EANSDHILOTRMUVCKWPGFBYJQXZ

 Ej hq wxaschw xagq cjjqzsqs
 Rxezy mtr rxew azs add ew vqzsqs
 Rxar ict xagq mtr wdtvmqfqs xqfq
 Hxedq rxqwq geweczw ses annqaf
 Azs rxew hqay azs esdq rxqvq
 Zc vcfq ieqdsezu mtr a sfqav
 Uqzrdqw, sc zcr fqnfqxqzs
 Ej ict nafscz, hq wxadd vqzs
 Azs aw E av az xczqwr Ntby
 Ej hq xagq tzqafzqs dtby
 Zch rc 'wbanq rxq wqfnqzr'w rczutq
 Hq wxadd vayq avqzsw qfq dczu
 Qdwq rxq Ntby a deaf badd
 Wc uccszeuxr tzrc ict add
 Uegq vq ictf xazsw ej hq mq jfeqzsw
 Azs Fcmez wxadd fqwrcfq avqzsw

        Heddeav Wxayqwnqafq, Veswtvvqf Zeuxrw Sfqav
XXXDYYYYBBBBBBBGGGGGGGG
```

The line at the end, XXXDYYYY etc, is padding to make sure that the occurrence counts are all unique. The following is the output corresponding to this input.

```
Encryption Program Output.

Key: EANSDHILOTRMUVCKWPGFBYJQXZ

Encrypted Text:

**********************************************************************

 Ej hq wxaschw xagq cjjqzsqs
 Rxezy mtr rxew azs add ew vqzsqs
 ....
 Uegq vq ictf xazsw ej hq mq jfeqzsw
 Azs Fcmez wxadd fqwrcfq avqzsw

        Heddeav Wxayqwnqafq, Veswtvvqf Zeuxrw Sfqav
XXXDYYYYBBBBBBBGGGGGGGG

**********************************************************************

Letter Frequencies:

Q 60
A 37
Z 32
W 29
S 28
X 26
E 25
 ....
I 5
K 0
L 0
O 0
P 0


Decrypted message:

**********************************************************************

 If we shadows have offended
 Think but this and all is mended
 That you have but slumbered here
 While these visions did appear
 And this weak and idle theme
 No more yielding but a dream
 Gentles, do not reprehend
 If you pardon, we shall mend
 And as I am an honest Puck
 If we have unearned luck
 Now to 'scape the serpent's tongue
 We shall make amends ere long
 Else the Puck a liar call
 So goodnight unto you all
 Give me your hands if we be friends
 And Robin shall restore amends

        William Shakespeare, Midsummer Nights Dream
HHHLKKKKCCCCCCCVVVVVVVV
```

23.7.2. Directory Cleaner

One of the usual problems that each user of TOPS-10 faces is the accumulation of useless files. Write a program to help a user decide which files to keep and which to delete. The directory cleaner program should request three specifications from the user. First, the ppn of a directory to examine; second, a file size (in records); third, a date.

The program should examine each file in the specified directory. If the file is larger than the given size argument and if it is older than the given date, print the name of the file, the size, and the creation date. Then ask the user if he wants to delete this file.

The RENAME MUUO can be used to delete a file. Consult [MCRM] for details of how to use it.

23.7.3. Fixed-Field Sorting Program

Write a program to sort data selected from a fixed-field in each line of a data file. Allow the user to specify four 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

A *record* is a block of memory with a particular structure that is defined by the programmer. The structure of a record, as defined by the program, may include fixed portions in particular places, and it may include a variable length portion that begins at a specific place. Each of the data items within a record is usually called a *field.*

One of the particular benefits of using records is that a record keeps all the information pertaining to a particular item in one place. 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. 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.

One technique is to use an array to contain record pointers. However, an array is not always suitable. An array has a fixed size; in many cases there is no way to know in advance how many different records there will be. Space for an array of record pointers can not be allocated until the number of records is 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 structure.* List structures 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).
- 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 terms from the programming language LISP). In the DECsystem-10 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. DICTIONARY PROGRAM - EXAMPLE 16

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 this is the 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:  CLOSE   ICHAN,                  ;close input file
        RELEAS  ICHAN,
        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
```

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 closing the output channel and exiting. We may assume that FINISH will end any output activity and close the output file.

```
        CALL    PRDICT                  ;print dictionary, both ways
        CLOSE   OCHAN,                  ;clean up output activity
        STATZ   OCHAN,740000
        JRST    OUTERR
        RELEAS  OCHAN,
        EXIT                            ;stop execution here
```

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.

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 do 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 follows:

```
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. (The line containing the definition of WRDBUF is not part of the record definition. It is included here because it is mentioned in the discussion of the PHASE pseudo-op.)

24.2.1. Suppressed Labels

The first new thing is the use of the characters :! to define a symbol. 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 will not be typed out by the symbolic disassembler in DDT. These labels are suppressed in this program because, as we shall see, they have values that would conflict with our accumulator definitions.

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 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.

We define the symbols WCOUNT, HSHLNK, etc., as suppressed labels because they take on values that would conflict with our accumulator names.

The DEPHASE pseudo-op makes both location counters the same. The value of the storage location counter is copied to the phase location counter; things go back 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 assembled and stored will be located at WRDBUF+40, the value of DEFS.

The .ORG pseudo-op reclaims the space wasted 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 really use this space for something else (it will be START) rather than for 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. We will use the memory location called .JBFF as the pointer to the next memory address that we can allocate for list space. The list head of the name list, NSHEAD, and the head of the count list, CSHEAD are both set to zero.

```
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
        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: HRRZ    A,HSHLNK(B)             ;link to next item
        JUMPE   A,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: HRRZ    A,HSHLNK(B)             ;link to next item
        JUMPE   A,PROCW3                ;Jump if 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)               ;Found it. 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
```

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 not obvious. We said that PROCW1 expects that register B contains a record pointer to the previous record. Although there is no record previous to the first record, we can pretend that there is. We load register B with the address of HASHTB-HSHLNK(A):

```
PROCWD: MOVE    B,[POINT 7,WRDBUF]      ;Pointer to the input word
        CALL    HSHFUN                  ;compute hash code for it.
        MOVEI   B,HASHTB-HSHLNK(A)      ;initial link address (B=prev record)
PROCW1: HRRZ    A,HSHLNK(B)             ;link to the next item
```

When we get to PROCW1 the first time, the HRRZ there will use B as a pointer to a record. The HRRZ 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 normal 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. We arrive 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 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.
        HRRZM   A,HSHLNK(B)             ;store addr of new item in last record
        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. Essentially, 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. 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 the possible values.

```
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
```

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 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.
```

24.3.1.3. BLDBLK

The purpose of the BLDBLK routine is to build a new dictionary record. First, BLDBLK must determine if there is enough space in memory to add another record. It does this by taking the address found in .JBFF, adding 50 to it, and checking to see if the sum is less than .JBREL. If the sum exceeds the current memory allotment, the program performs a CORE MUUO to obtain more space.

When space is available, the address of the first available word in free space is copied from .JBFF to register A; this will be the address of the new record. The HSHLNK and WCOUNT fields are set to zero.

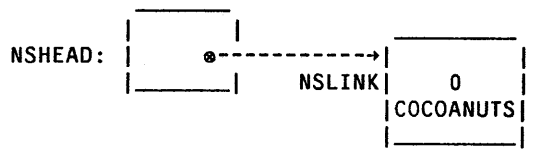
Register C is set up to point to the string portion of the new record. The text of the new word is copied to the new record. After copying the string, register C points to the last word in which text was stored; incrementing the right half of C results in a pointer to the next free word in memory; this is stored as the new value .JBFF.

```
BLDBLK: MOVE    A,.JBFF
        ADDI    A,50                    ;make sure we have enough room
        CAMGE   A,.JBREL                ;skip if we need more core
        JRST    BLDBL0                  ;we have enough
        CORE    A,                      ;request more
        JRST    NOCORE                  ;request denied
BLDBL0: HRRZ    A,.JBFF                 ;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.
        ADD     C,[POINT 7,NAMBLK]      ;make byte pointer to string in new rec
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
        HRRZM   C,.JBFF                 ;store as the next free word in mem.
```

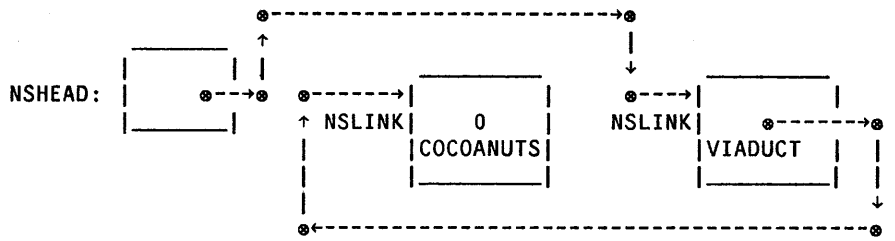
The head of the name-sort list, NSHEAD, is picked up at BLDBL2. This is the address of the most recently added record. The pointer to this other record is stored in the NSLINK and CSLINK portions of the new record we have built. The address of the new record is stored as NSHEAD and CSHEAD. These instructions 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                ;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
```

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 the stored in NSHEAD. Similar activities occur with the count-sort list. The result, ignoring the count-sort list, looks like this:



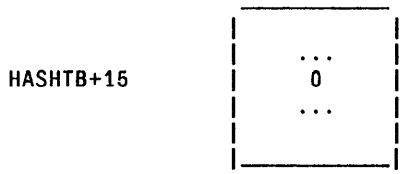
When another new word arrives, BLDBLK will make a dictionary entry for it. Suppose the new word is VIADUCT. Then the contents of NSHEAD will be copied to the NSLINK of the VIADUCT record. The address of the VIADUCT record will then be stored in NSHEAD. This changes the picture:



Each new record is added to the front of the name-sort list; NSHEAD will always point to the most recently added record.

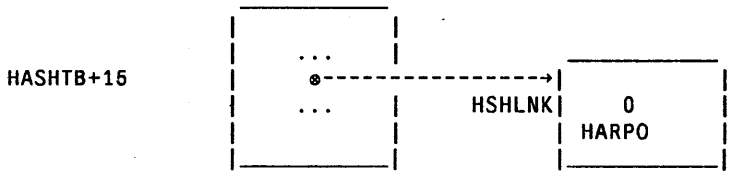
Unless the reader is familiar with pointers, it is easy to become lost. We continue our digression by giving a specific example of searching by hash code. Two words, HARPO and CHICO, both have the same hash code, octal 15.[1]

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.



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.

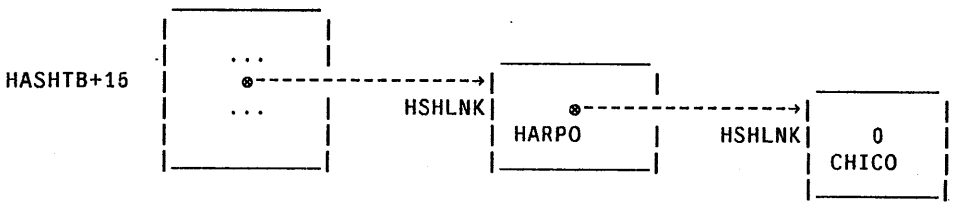
---

[1] The hash code for GROUCHO is 143.

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 right half of 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

It should be mentioned that 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.

```
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. 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
        EXTERN  .JBFF, .JBREL

A=1
B=2
C=3
D=4
W=5
X=6
Y=7
Z=10
P=17

OCHAN==1                                ;Define I/O channels
ICHAN==2

OPDEF   CALL    [PUSHJ P,]
OPDEF   RET     [POPJ P,]

PDLEN==40                               ;stack length
HSHTLN==177                             ;hash table size
BUFLEN==100

;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
        PHASE   0                       ;Define field names for a file block
DEVNAM:!        0                       ;Device name
FILNAM:!        0                       ;File name
FILTYP:!        0                       ;File extension
FILPPN:!        0                       ;PPN
FILBSZ==.                               ;size of a file block
        DEPHASE
        .ORG    DEFS

OBUFH:  BLOCK   3                       ;Output file buffer header
IBUFH:  BLOCK   3                       ;Input file buffer header

TTYBUF: BLOCK   BUFLEN                  ;tty input line buffer
IFILBK: BLOCK   FILBSZ                  ;Input file name block
OFILBK: BLOCK   FILBSZ                  ;Output file name block


NSHEAD: 0                               ;head of the name-sort list
CSHEAD: 0                               ;head of the count-sort list

PDLIST: BLOCK   PDLEN                   ;the stack
HASHTB: BLOCK   HSHTLN                  ;the hash table
WRDBUF: BLOCK   40                      ;The word buffer
```

```
        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:  CLOSE   ICHAN,                  ;close input file
        RELEAS  ICHAN,
        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
        CLOSE   OCHAN,                  ;clean up output activity
        STATZ   OCHAN,740000
        JRST    OUTERR
        RELEAS  OCHAN,
        EXIT                            ;stop execution here

        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.

ISLET:  CAIL    A,"A"
        CAILE   A,"Z"
        RET
CPOPJ1: AOS     (P)                     ;return with a skip.
CPOPJ:  RET

        SUBTTL  IO, Error Handling

;low-level I/O routines
PUTCHR: SOSLE   OBUFH+2                 ;Decrement Buffer byte count
        JRST    PUTCH1                  ;Room left in buffer
        OUTPUT  OCHAN,                  ;send old buffer
        STATZ   OCHAN,740000            ;watch for errors
        JRST    OUTERR                  ;bad
PUTCH1: IDPB    A,OBUFH+1               ;add character to buffer
        RET

GETCHR: SOSLE   IBUFH+2                 ;Decrement buffer byte count
        JRST    GETCH1                  ;Buffer is non-empty. go get byte
        INPUT   ICHAN,                  ;get a new buffer
        STATZ   ICHAN,740000            ;watch for errors
        JRST    INERR                   ;bad
        STATZ   ICHAN,20000             ;check for end of file
        RET                             ;non-skip return for eof
GETCH1: ILDB    A,IBUFH+1               ;get a byte from the buffer
        JUMPE   A,GETCHR                ;discard null bytes
        CALL    PUTCHR                  ;Copy this to the output file
        CAIL    A,"a"                   ;Is this character lower case?
        CAILE   A,"z"                   ;if so, make it upper
        JRST    CPOPJ1                  ;return byte in A, with skip
        TRZ     A,40
        JRST    CPOPJ1

GTTYLN: MOVE    B,[POINT 7,TTYBUF]      ;Read one line to TTYBUF
GTTYL1: INCHWL  A                       ;Get one character
        CAIN    A,15                    ;discard CR
        JRST    GTTYL1
        CAIN    A,12                    ;convert LF to null
        MOVEI   A,0
        IDPB    A,B                     ;store in buffer,
        JUMPN   A,GTTYL1                ;loop until end of line
        RET

;Error handling
INERR:  OUTSTR  [ASCIZ/Input error.  Status = /]
        GETSTS  ICHAN,A
OCTERR: CALL    OCTOUT
        OUTSTR  CRLF
        EXIT

OUTERR: OUTSTR  [ASCIZ/Output error.  Status = /]
        GETSTS  OCHAN,A
        JRST    OCTERR                  ;print octal error code & stop

NOOPEN: OUTSTR  [ASCIZ/Open failed for device /]
        MOVE    Y,[OUTCHR A]
        CALL    TYDEVN
        OUTSTR  CRLF
        RET
```

```
NOLOOK: OUTSTR  [ASCIZ/Lookup failed to find file /]
        CALL    TYFILN
        OUTSTR  CRLF
        RET

NOENTR: OUTSTR  [ASCIZ/Enter failed to select output file /]
        CALL    TYFILN
        OUTSTR  CRLF
        RET

NOCORE: OUTSTR  [ASCIZ/Insufficient core available
/]
        EXIT

;Get input file name from terminal. open it for 7-bit bytes.
GTINPF: OUTSTR  [ASCIZ/File name for input: /]
        CALL    GTTYLN                          ;Read a line from terminal
        LDB     A,[POINT 7,TTYBUF,6]            ;read first byte of response
        JUMPE   A,CPOPJ
        MOVEI   Z,IFILBK                        ;select input file block
        MOVE    W,[POINT 7,TTYBUF]
        CALL    SCNFIL                          ;scan a file name
        JRST    GTINPF                          ;can't parse file name

        MOVEI   A,0                             ;mode 0 for input
        MOVE    B,DEVNAM(Z)                     ;device name from SCNFIL
        MOVEI   C,IBUFH                         ;Address of input buffer header
        OPEN    ICHAN,A                         ;Open the channel
        JRST    [CALL NOOPEN                    ;if error, print message
                 JRST GTINPF]                   ;loop. try again
        MOVE    A,FILNAM(Z)                     ;Setup for LOOKUP
        MOVE    B,FILTYP(Z)                     ;Copy parameters from SCNFIL
        SETZ    C,
        MOVE    D,FILPPN(Z)
        LOOKUP  ICHAN,A                         ;obtain read access to file
        JRST    [CALL NOLOOK                    ;error: print message
                 JRST GTINPF]                   ;try again
        RET

;Get output file name from terminal, open it for 7-bit bytes.
GTOUTF: OUTSTR  [ASCIZ/File name for output: /]
        CALL    GTTYLN                          ;Read a line from terminal
        MOVEI   Z,OFILBK                        ;select output file block
        MOVE    W,[POINT 7,TTYBUF]
        CALL    SCNFIL                          ;scan a file name
        JRST    GTOUTF                          ;don't understand the file name

        MOVEI   A,0                             ;Output mode 0: buffered ascii
        MOVE    B,DEVNAM(Z)                     ;device name from SCNFIL
        MOVSI   C,OBUFH                         ;output buffer header
        OPEN    OCHAN,A                         ;open the channel
        JRST    [CALL NOOPEN                    ;error: print message
                 JRST GTOUTF]                   ; and loop
        MOVE    A,FILNAM(Z)                     ;setup for Enter
        MOVE    B,FILTYP(Z)                     ;use results from SCNFIL
        SETZ    C,
        MOVE    D,FILPPN(Z)
        ENTER   OCHAN,A                         ;select file for output
        JRST    [CALL NOENTR                    ;error: print message
                 JRST GTOUTF]                   ;loop
        RET
```

```
;File name and device type out.
TYDEVN: MOVE    B,DEVNAM(Z)             ;Must type device name.
        CALL    SIXOUT                  ;print as sixbit
        MOVEI   A,":"                   ;add colon after dev name
        XCT     Y
        RET

;Call TYFILN with Z = address of file descriptor block.
;Output file name to TTY.
;Call TFILNX with Z setup as above and Y containing an instruction to XCT
TYFILN: MOVE    Y,[OUTCHR A]            ;instr to execute to send data
TFILNX: SKIPN   B,DEVNAM(Z)             ;get the device name
        JRST    TFILN1                  ;skip this part if there's none
        CAME    B,['DSK   ']            ;don't print DSK either
        CALL    TYDEVN                  ;print the device name
TFILN1: SKIPN   B,FILNAM(Z)             ;print the file name, if present
        RET                             ;exit now if no file name
        CALL    SIXOUT                  ;print the file name
        HLLZ    B,FILTYP(Z)
        JUMPE   B,TFILN2                ;jump if no extension
        MOVEI   A,"."                   ;print period after name
        XCT     Y
        CALL    SIXOUT                  ;print file extension
TFILN2: SKIPN   FILPPN(Z)               ;print PPN if non-zero
        RET                             ;return quick
        MOVEI   A,"["                   ;print the square bracket
        XCT     Y
        HLRZ    A,FILPPN(Z)             ;print the Project part
        CALL    OCTOU1                  ; in octal
        MOVEI   A,","                   ;comma to separate p,pn
        XCT     Y
        HRRZ    A,FILPPN(Z)             ;print programmer part in octal
        CALL    OCTOU1
        MOVEI   A,"]"                   ;finish ppn.
        XCT     Y
        RET

;Print Sixbit of B via XCT Y
SIXOUT: JUMPE   B,CPOPJ                 ;sixbit output
        MOVEI   A,0                     ;clear high word
        LSHC    A,6                     ;slide the data up into A
        ADDI    A," "                   ;convert sixbit to ascii
        XCT     Y                       ;print it and loop
        JRST    SIXOUT

;OCTOUT:  Print octal of A on TTY.
;OCTOU1:  Print octal of A via XCT Y
OCTOUT: MOVE    Y,[OUTCHR A]            ;recursive octal printer
OCTOU1: IDIVI   A,10
        PUSH    P,B
        SKIPE   A
        CALL    OCTOU1
        POP     P,A
        ADDI    A,"0"
        XCT     Y
        RET
```

```
;DECOUT:  Print decimal of A on TTY.
;DECOU1:  Print decimal of A via XCT Y
DECOUT: MOVE    Y,[OUTCHR A]
DECOU1: IDIVI   A,12
        PUSH    P,B
        SKIPE   A
        CALL    DECOU1
        POP     P,A
        ADDI    A,"0"
        XCT     Y
        RET

SCNFIL: MOVSI   A,'DSK'                 ;assume device DSK
        MOVEM   A,DEVNAM(Z)             ;as the default
        SETZM   FILNAM(Z)               ;clear out everything else
        SETZM   FILTYP(Z)
        SETZM   FILPPN(Z)
        CALL    GETSIX                  ;device name or file name first
        CAIE    A,":"                   ;if there's a colon, this is dev
        JRST    SCNFL1                  ;no colon, this was the file name
        MOVEM   B,DEVNAM(Z)             ;save the device name
        CALL    GETSIX                  ;next thing must be the file name
SCNFL1: MOVEM   B,FILNAM(Z)             ;save the file name
        CAIE    A,"."                   ;if there's a period, get file type
        JRST    SCNFL2                  ;no file type present
        CALL    GETSIX                  ;read the file type
        HLLZM   B,FILTYP(Z)             ;save it
SCNFL2: CAIE    A,"["                   ;PPN next?
        JRST    CPOPJ1                  ;no.  assume we're happy
        CALL    GETOCT                  ;read an octal number
        HRLZM   B,FILPPN(Z)             ;store project number
        CAIE    A,","                   ;must see a comma next
        JRST    SCNERR                  ;error if it's not there
        CALL    GETOCT                  ;read the programmer part
        HRRM    B,FILPPN(Z)             ;save it
        CAIE    A,"]"                   ;must see the closing bracket
        JRST    SCNERR                  ;error
        CALL    GETSIX                  ;skip past the closing bracket
        JUMPN   B,SCNERR                ;any non-blank here is an error
        JRST    CPOPJ1                  ;return happy.

SCNERR: OUTSTR  [ASCIZ/I can't understand this file name
/]
        RET

;Read from byte pointer in W and return octal value in B
GETOCT: MOVEI   B,0                     ;initial value is zero
GETOC1: ILDB    A,W                     ;get a character
        CAIL    A,"0"
        CAILE   A,"7"
        RET                             ;not part of my number
        LSH     B,3                     ;shift old value
        ADDI    B,-"0"(A)               ;add in the next digit
        JRST    GETOC1                  ;loop

;get the next alpha-numeric sixbit term.
;input byte pointer in W.  Result in B, ascii delimiter in A
GETSIX: MOVEI   B,0                     ;Clear out result word
        MOVE    C,[POINT 6,B]           ;byte pointer into result word
GETSX1: ILDB    A,W                     ;get a character
        CAIL    A,"a"                   ;convert lowercase to sixbit
        CAILE   A,"z"
        JRST    GETSX2                  ;not lowercase. see what else it is
        TRZ     A,100                   ;lowercase to sixbit
        JRST    GETSX4                  ;go store this character
```

```
GETSX2: CAIL    A,"A"                   ;uppercase letter?
        CAILE   A,"Z"
        JRST    .+2
        JRST    GETSX3                  ;uppercase letter
        CAIL    A,"0"                   ;look for digits
        CAILE   A,"9"
        RET                             ;not part of a file name  return it
GETSX3: SUBI    A," "                   ;convert to sixbit
GETSX4: TLNE    C,770000                ;don't overflow out of B
        IDPB    A,C                     ;store character in B
        JRST    GETSX1                  ;loop

        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.
        MOVEI   B,HASHTB-HSHLNK(A)      ;initial link address (B=prev record)
;Enter PROCW1 with B = Address of Previous Record
PROCW1: HRRZ    A,HSHLNK(B)             ;link to next item
        JUMPE   A,PROCW3                ;Jump if 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.
        HRRZM   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,.JBFF
        ADDI    A,50                    ;make sure we have enough room
        CAMGE   A,.JBREL                ;skip if we need more core
        JRST    BLDBL0                  ;we have enough
        CORE    A,                      ;request more
        JRST    NOCORE                  ;request denied
BLDBL0: HRRZ    A,.JBFF                 ;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.
        ADD     C,[POINT 7,NAMBLK]      ;make byte pointer to string in new rec
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
        HRRZM   C,.JBFF                 ;store as the next free word in mem.
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
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
        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 list to sort.  Returns 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's 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.
;now, 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

;now, the situation is that there are two lists, A and B, both sorted.
;now, 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
```

24.4.1. 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;[2] a list that contains only one element is already sorted. 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.

---

[2]Zero length lists are handled by the SKIPE A,NSHEAD outside the first call to NSSORT.

```
        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.
```

We haven't yet explained how NSSORT works, but suppose that it does work. Having called NSSORT twice for the two short lists, we now have two short, sorted, lists. But we didn't want two sorted lists. We must merge the two short lists into one list. By *merge* we mean to 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 at 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.

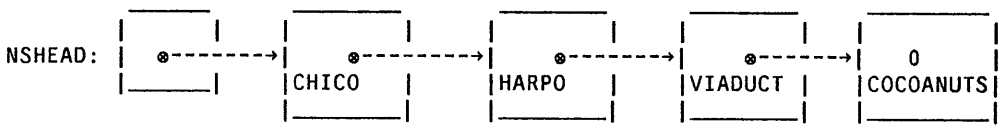
The first element of the A-list is removed and appended (via D) to the end of the output list, whose list head is in register C. D is updated so that it always points to the tail of the C-list. Register A is advanced past the head of the A-list by means of the instruction SKIPE A,NSLINK(A). If the SKIPE 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.

```
;now, the situation is that there are two lists, A and B, both sorted.
;now, 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
```

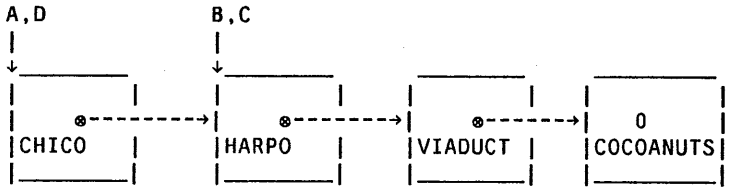
At this point we hope it is clear how NSSORT works. NSSORT divides the input list into two lists, each being half the length of the original. A pair of recursive calls to NSSORT further subdivides these lists. When a list is made that has only one element, NSSORT returns that list.

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.

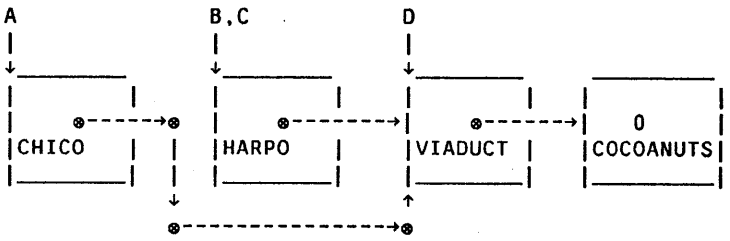
Suppose the original, unsorted list in NSHEAD is



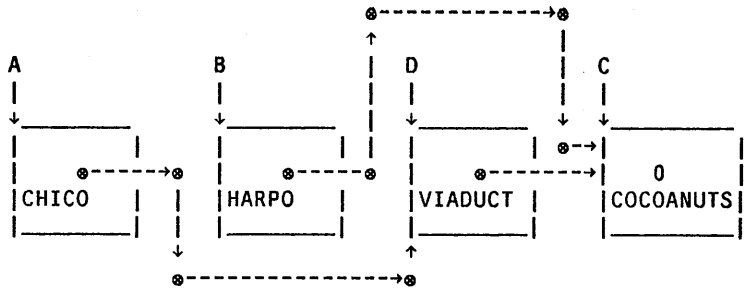
This diagram omits all but the NSLINK and NAMBLK fields of the records. Initially, register A points to the CHICO record. Since the link out of CHICO is not empty, the instructions at NSORT1 will be executed. The first time the program arrives at NSORT1, registers A, B, C, and D will be set up as indicated:



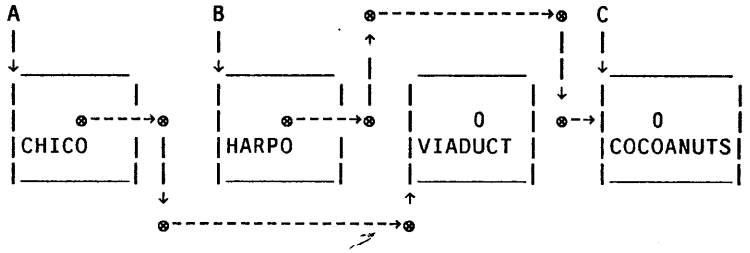
At NSORT1, register W gets a copy of the link out of the record that C addresses. C points to the HARPO record; W is loaded with a pointer to the VIADUCT record. The pointer in W is stored in the record addressed by D. Register D addresses the CHICO record; this changes the link out of the CHICO record 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:



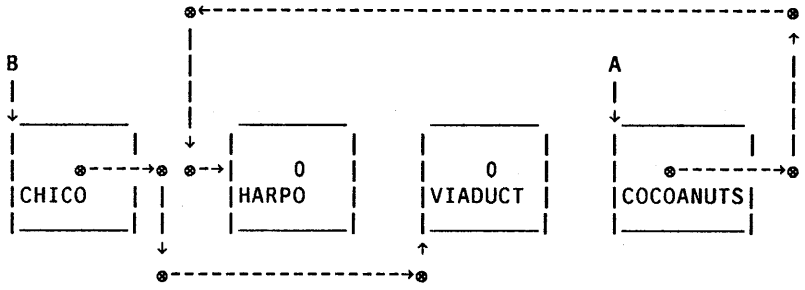
Next, W is loaded with a copy of the link out of the record that D addresses. Register D points to VIADUCT; W is loaded with a pointer to COCOANUTS. Then W is stored as the link out of the record that C addresses. This changes the link out of HARPO to point to COCOANUTS. Register C is loaded with a copy 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:



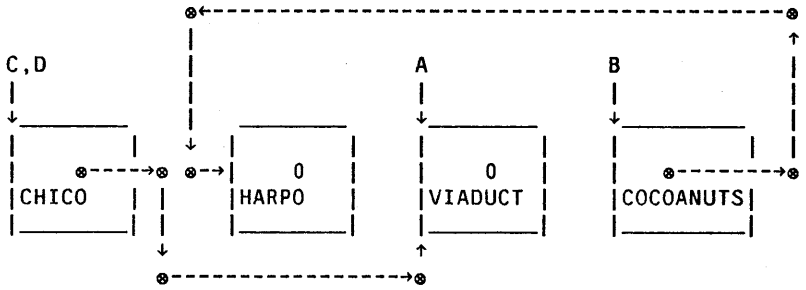
It is unimportant that A and B have exchanged meanings; all that matters is that both are pointers to short sorted lists.

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.

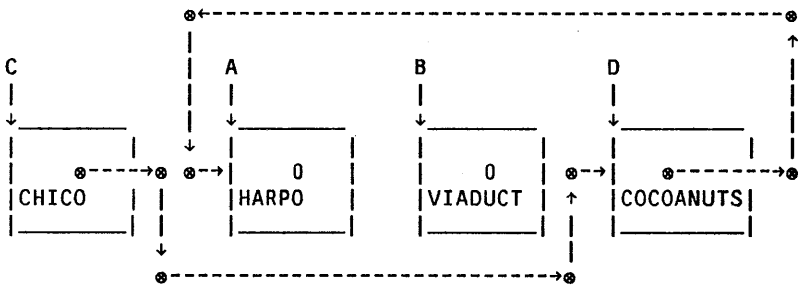
NSMERG calls NSCOMP; NSCOMP looks at the records addressed by A and B. If register A addresses a record that is alphabetically less that the record that B addresses, then NSCOMP will skip. Otherwise, NSCOMP will return without skipping. The first time that NSCOMP is called, it decides that CHICO is smaller 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.

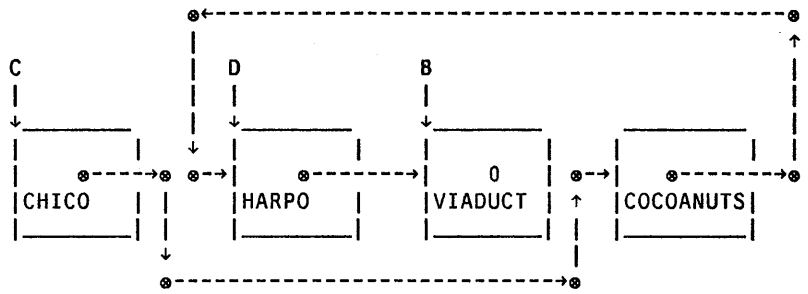


At NSMERG, again the records pointed to by A and B are compared. The COCOANUTS record, addressed by B, is smaller; A and B are exchanged again. The pointer to COCOANUTS is stored as the link out of the CHICO record addressed by D. Register D is then changed to point to COCOANUTS. Register A is advanced 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*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.

24.4.2. 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.

24.5. EXERCISES

24.5.1. 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. 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

   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.
        ; . . .
```

24.5.2. 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   .....
```

24.5.3. 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.

24.5.4. 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
Random Access I/O

In TOPS-10 the minimum unit of information transfer between memory and the disk is a *physical record* or *sector* that contains 128 words. When we perform random access I/O, we position the file in terms of these physical records.

Random access can be done either in buffered mode or in dump mode. However, random access in buffered mode presents a special problem that we'll discuss later, so our first example will use dump mode. In dump mode there are no buffers and input/output operations are synchronized with the INPUT and OUTPUT UUOs.

There are two MUUOs relating to random access I/O. These are

- USETI set file input pointer.
- USETO set file output pointer.

The input and output pointers are physical record numbers, with the first record in the file being number one. USETO and USETI require an I/O channel number in the accumulator field. The effective address is an immediate argument.

In the usual input and output cases random access is fairly straightforward. A channel is opened, a file is selected, and input or output may be done, with interspersed USETI or USETO operations.

25.1. EXAMPLE PROGRAM

Our first example of random access will show input processing. To keep this example short and focussed on the topic of random access, we present only a fragment of a larger program.

This fragment is taken from DART a file system backup utility. A backup utility should serve two purposes: first, the system manager must be assured that the file system can be restored from any calamity that may befall the disk; second, a user should be able to restore any file that he or she deletes or damages. A possible third purpose is to use the backup utility to perform archival storage; this is similar to the second pupose listed, but the record-keeping requirements are more stringent.

The DART program attempts to address all of these issues. In connection with archival record keeping, DART maintains a file that contains the name of every file every written on a backup tape, the file's date, and the tape number where it was written. In about eleven years of operation, DART has accumlated 2.5 million words of archival records to locate files on its collection of tapes.

Since a linear scan of 2.5 million words would take quite a lot of processing, DART has structured its data

base to permit random access. The fragment we present locates the backup records associated with one project-programmer name. This fragment is part of a larger command set that allows a user to specify particular file names, wild cards, etc.

The data file has the following format:

```
Record 1
          ____________________
        0 | ppn 1              |     the first PPN recorded in the file
        1 |       200 (octal)  |     file word number where info for
        2 | ppn 2              |         that PPN is recorded
        3 |    ppn 2 location  |     Second PPN and word position
        4 | ppn 3              |     Third PPN
        5 |    ppn 3 location  |        and word position
                ....
      174 | ppn 63 (decimal)   |     Last PPN in first index record
      175 |   ppn 63 location  |         and word position
      176 | next index record  |     Word number of next index record
      177 |      0             |
Record 2
          ____________________
      200 | ppn 1              |     Data area for PPN 1
      201 | File 1             |     First file name for PPN 1
      202 | Ext 1  |   count   |     First file extension & record count
      203 | tape # | file date |     For each instance of a backup copy of
      204 | tape # | file date |       this file, the file's date and the
      205 | tape # | file date |       tape number is recorded.  The right
      206 | tape # | file date |       half of the extension word records
      207 | tape # | file date |       the number of entries for this file.
      210 | File 2             |     When the count is exhausted, another
      211 | Ext 2  |   count   |       file name, extension and count appears
      212 | tape # | file date |
                ....
          |      0             |     A zero file name means a PPN follows
          | ppn 2              |
                ....

          |      0             |     A zero file name means a PPN follows
          | ppn 63             |
                ....
          |      0             |     After the entries for the first 63 PPNs
          |      0             |     have been recorded, zeros fill the balance
          |      0             |     of the current 128-word disk record.
          ____________________
    ...+0 | ppn 64             |     this is the second index record, that
    ...+1 |    ppn 64 location |       word 176 of the first index record
    ...+2 | ppn 65             |       points to.
    ...+3 |    ppn 65 location |
                ....
  ...+174 | ppn 126 (decimal)  |     Last PPN in second index record
  ...+175 |  ppn 126 location  |         and word position
  ...+176 | next index record  |     Word number of next index record
  ...+177 |      0             |
          ____________________
```

This format, index records with data interspersed, continues. The last index record generally isn't full; it is padded with zero words; in the last index record, the pointer to the next index record will also be zero.

Now, to our program fragment. We assume that the channel known as `FILE` has already been opened in dump mode and that a successful `LOOKUP` has been performed. The `FNDPPN` subroutine is entered with the sought-for PPN in register `W`. When the PPN is found, `FNDPPN` positions the file so the data area for this PPN will be the next thing read; it then returns with a skip. If the PPN can't be found, `FNDPPN` returns without a skip.

```
        SUBTTL  FNDPPN - Search in data file for PPN

;Enter here with FILE an OPEN channel on which a LOOKUP has been performed
;       W/ PPN to search for
;       Returns with a skip and the file positioned to read that PPN's data
;       Non-skip for no such PPN
;
FNDPPN: INPUT   FILE,[IOWD 200,IDXBUF   ;read a record from the file
                      0]
        STATZ   FILE,740000             ;test for usual errors
        JRST    INPERR
        STATZ   FILE,20000              ;end of file?
        RET                             ;We didn't expect this
        MOVE    X,[-176,,IDXBUF]        ;''AOBJN'' pointer to data
FNDPP1: CAMN    W,(X)                   ;compare a PPN
        JRST    FNDPP2                  ;found it!
        ADD     X,[2,,2]                ;skip to next PPN
        JUMPL   X,FNDPP1                ;loop unless done w/this index
        SKIPN   X,IDXBUF+176            ;fetch pointer to next index
        RET                             ;no pointer: can't find ppn
        LSH     X,-7                    ;divide by 128 to get record #
        USETI   FILE,1(X)               ;add 1 to record number (first record
        JRST    FNDPPN                  ;  is number 1) and read next index

;(X) points to the PPN's entry in the index record; 1(X) is the word address
;of this PPN in the file
FNDPP2: MOVE    A,1(X)                  ;get word number of this entry
        LSH     A,-7                    ;record number that contains this ppn
        USETI   FILE,1(A)               ;point to that record
        INPUT   FILE,[IOWD 200,DATBUF   ;read the record
                      0]
        MOVE    A,1(X)                  ;get the word number again
        ANDI    A,177                   ;Word number within the record
        MOVE    B,[POINT 36,DATBUF]
        ADD     B,A                     ;byte pointer to the word w/PPN
        MOVEM   B,FIBUF+1               ;simulate buffered input
        MOVEI   B,201                   ;nominal word count, offset by 1
        SUB     B,A                     ;reduce by words skipped at the front
        MOVEM   B,FIBUF+2               ;save ''buffer count''
        JRST    CPOPJ1


;Simulate buffered mode input using dump mode.
FILGET: SOSLE   FIBUF+2                 ;decrement buffer count
        JRST    FILGT1                  ;data remains in the buffer
        INPUT   FILE,[IOWD 200,DATBUF   ;get more data
                      0]
        STATZ   FILE,740000             ;test for usual errors
        JRST    INPERR
        STATZ   FILE,20000              ;test end of file
        RET
        MOVE    A,[POINT 36,DATBUF]
        MOVEM   A,FIBUF+1
        MOVEI   A,200
        MOVEM   A,FIBUF+2
FILGT1: ILDB    A,FIBUF+1
        JRST    CPOPJ1
```

The first point to notice is that USETI takes an *immediate* quantity as its argument. This is most usually accomplished by using an index register. In our example, we take a word number and divide it by decimal 128 (by means of a right shift by 7 bits). The result is the record number that contains the specified address. However, since the first record is number one (instead of zero as you might expect), we add one to the shifted quantity to form the argument to USETI. We use the construction 1(X) to accomplish the addition of one and the formation of the correct argument.

When we want a word that is not at the start of the record, we use the same technique to select the correct record. Then the low-order seven bits of the word number represent the offset within this record. At FNDPP2 the program reads the specified record. Then it sets up FIBUF+1 and FIBUF+2 as the byte pointer and count to simulate buffered mode input. The byte pointer is advanced by the quanity specified in the low-order seven bits of A. The byte count is reduced from octal 200 by the same amount. The program actually subtracts the number of data words to skip from 201. The extra one in the resulting count adjusts FIBUF+2 to account for the INPUT occurring outside the FILGET routine.

FILGET is shown for completeness. It runs a count and a byte pointer as though it was doing buffered input. However, these data items are set up explicitly by the program, as shown.

25.2. UPDATE-IN-PLACE

Update-in-place access allows a program to modify an existing file. Update-in-place immediately changes an existing file. (Our previous examples of output have created new versions of files that supercede any previous file of the same name.)

To obtain update-in-place access to a file, it is necessary to perform a LOOKUP and then an ENTER, where both MUUOs use the same file name and the same I/O channel. Now, it is possible to do USETI and read the selected physical record, modify that record in memory, do a USETO and OUTPUT and have the modifications occur without recopying the entire file.

The sequence of LOOKUP and ENTER provides a mode of update called *single access update.* In this mode, precisely one job can access the file; this mode cannot be obtained if, for instance, some other job is reading the same file when the ENTER is attempted. Except for this mention of it, we shall not discuss *multiple access update*, which can be achieved via the FILOP UUO; in multiple access update, all programs that access the file must agree among themselves how to manage the updates in an orderly fashion.

One use for (single access) update mode is to extend a file. In order to extend a file, update-in-place access is obtained. The file word count, as returned by LOOKUP is examined. If the word count indicates that data portion of the file ends in the middle of a record, that last partial record is read into memory. That record is augmented, and written out. The OUTPUT call will effect an increase in the file word count. If there is no partial record, the program simply does a USETO to position the file past the old end of file, and commences writing.

A short example of extending a file is shown below:

```
        MOVE    A,['DART  ']
        MOVSI   B,'LOG'
        SETZB   C,D
        LOOKUP  FILE,A                  ;obtain read access to the file
        JRST    NOLOOK                  ;can't do it
        JUMPGE  D,BIGFIL                ;jump if large file
        MOVS    D,D                     ;small file.  Swap to get -size
        MOVN    D,D                     ;get the positive size
        SKIPA
BIGFIL: LSH     D,7                     ;shift record count to Word count
        MOVEM   D,FILLEN                ;Store the file length
        SETZB   C,D
        ENTER   FILE,A                  ;obtain single access update
        JRST    NOENT                   ;can't have it (file busy?)
        MOVE    A,FILLEN                ;get the file length in words
        TRNN    A,177                   ;less than a whole block at end?
        JRST    NOPBLK                  ;no partial block at end.
        LSH     A,-7
        USETI   FILE,1(A)               ;select the last block of the file
        LDB     A,[POINT 7,FILLEN,35]   ;length of last block
        MOVN    A,A                     ;-length
        HRLZ    A,A                     ;-length,,0
        HRRI    A,DATBUF-1              ;-length,,DATBUF-1, an IOWD
        MOVEI   B,0
        INPUT   FILE,A                  ;read the last block
        STATZ   FILE,760000             ;check for errors
        JRST    INPERR
NOPBLK: LDB     A,[POINT 7,FILLEN,35]   ;offset to DATBUF
        MOVEI   B,200
        SUB     B,A                     ;count of words left in DATBUF
        ADD     A,[POINT 36,DATBUF]     ;byte pointer for deposits
        MOVEM   B,DATCNT
        MOVEM   A,DATPNT
        MOVE    A,FILLEN
        LSH     A,-7                    ;block number of last block
        USETO   FILE,1(A)               ;select block for output

        ;add data to DATBUF via byte pointer DATPNT, and count DATCNT
        ;the next OUTPUT UUO will re-write the last block of the file
        ;(or, if the last block was full, it will extend the file)
        ;subsequent to that OUTPUT UUO, further OUTPUTs will extend
        ;file.
```

25.3. RANDOM ACCESS IN BUFFERED MODES

In our examples thus far we have limited our random access I/O to dump mode. The problem in buffered mode is lack of synchronization between the program's INPUT UUOs and the device service routine's activity in filling buffers. Simply stated, with many buffers it's difficult to know just what the content of a particular buffer may represent.

There are three ways to make sense of buffered mode input using random access. First, we could use a buffer ring composed of only one buffer. With only one buffer, there's no doubt what it contains. Second, we could OPEN the channel with the IO.SYN status bit (bit 30) set; this bit forces synchronization between each INPUT (or OUTPUT) UUO and the device service routine. These two techniqus are essentially equivalent and both eliminate the advantage of buffered mode being able to overlap I/O while your program does computation. The third technique allows the program the advantages of buffered mode overlapped computation and I/O, at the expense of extra complexity when a USETI is performed.

To understand the problem better, suppose we're doing input using a buffer ring that contains five buffers. When our program performs its first INPUT UUO, TOPS-10 asks the disk service routine to fill the

program's buffers. An indeterminate number of buffers (but at least one) are filled and TOPS-10 resumes the execution of the program. While our program is running (unless all the buffers have been filled, or end of file has been encountered) the disk service routine may continue filling additional buffers.

Suppose our program now reads some data from the buffer and then decides to do a USETI. USETI waits for the device service routine to become quiescent; then it sets the file so that the next buffer filled by the device service routine will contain the desired record. However, an indeterminate number of buffers full of irrelevant data are present and must be discarded. After doing the USETI our program must examine the buffer ring to locate the first unused buffer.[1] Then, INPUT UUOs should be repeated until the buffer made available by the TOPS-10 matches the first unused buffer.

The following fragment suggests the steps needed to get to the right buffer. Assume the channel named FILE is open for buffered mode input. The buffer control block is located at FIBUF. The sign bit in the buffer link word will be set if the buffer contains data from the file that hasn't been yet been processed by the program.

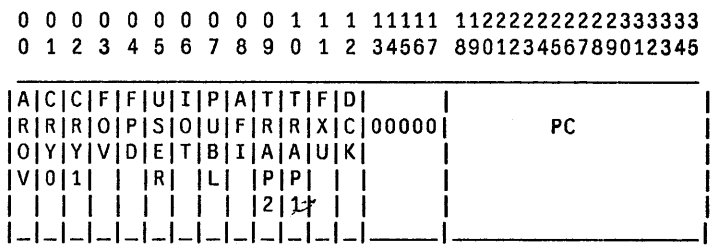
```
        USETI   FILE,....               ;Some USETI
        HRRZ    A,FIBUF                 ;address of the present buffer
        MOVE    B,A                     ;copy address of the current buffer.
FNDFRB: SKIPL   (B)                     ;skip if the buffer contains data
        JRST    FNDFB1                  ;B points to the first unused buffer
        HRRZ    B,(B)                   ;Advance to the next buffer
        CAME    B,A                     ;test for wrap around
        JRST    LOOP                    ;keep looking for an unused buffer
;Fall through if all buffers are in use.  B points to current buffer,
;which in the scheme of things will be the next free buffer.
FNDFB1: INPUT   FILE,                   ;make system advance to next buffer.
        HRRZ    A,FIBUF                 ;none of these INPUTs in this loop
        CAME    A,B                     ;calls the device service routine
        JRST    FNDFB1                  ;until the last one
;we have the right buffer now.  adjust the buffer byte count
;(assuming the usual form of buffered input reader)
        AOS     FIBUF+2
```

For USETO, TOPS-10 does the important work itself. Those buffers that we have told the system about (including the current one) are forced into the device service routine and written on the disk before changing the file position. It's probably a good idea after USETO to force the buffer byte count (e.g., FOBUF+2) to zero to force the next call to PUTCHR to perform an OUTPUT UUO.

[1] The program should guard against the possibility that all buffers are in use.

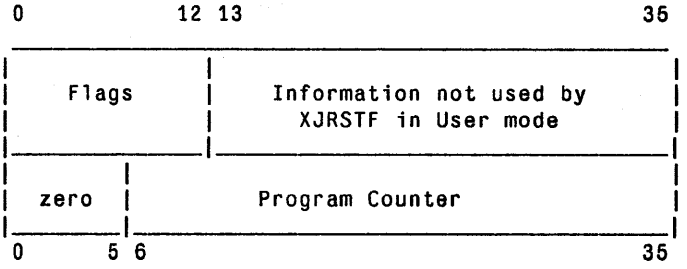
Appendix A
PC & Flags

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 Word

Subroutine calls that are executed in non-zero sections of the extended machine (DECSYSTEM-2060) 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.



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 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 -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.

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.

A 30-bit PC is stored with bits 0:5 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.

Appendix B
Instruction Nomenclature

E	The effective address of the I, X, and Y parts of the instruction.
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.
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 doubleword accumulator in which C(AC) is most significant.
PC	The 18-bit contents of the program counter.
∧	Boolean AND
∨	Boolean Inclusive OR
¬	Boolean Negation (One's Complement)
XOR	Boolean Exclusive OR
≡	Boolean Equivalence

Notation for Instruction Descriptions

Appendix C
DDT Commands

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 in the future. We believe the following is a correct (but not entirely complete) description of DDT in release 4 of TOPS-20. Some of this is a repetition of the material found in section 10. Our intention here 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.
- Any number that follows an escape, written as $n, is interpreted as a decimal number.

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 361 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

`CR` If a new value has been typed, that value is deposited. The open location is closed. Any temporary display modes that are in effect are cancelled; further displays will default to the prevailing permanent display mode.

`LF` 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.

`Back Space` The Back Space 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 ".".

`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.

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.
=	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.
_	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.
$H	Display quantities in halfword format.
$nO	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 byte typeout mask word, $3M, is used to define the locations of the byte boundaries. Each one 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 $nO command will be used.
$R	Display location addresses in relative (i.e., symbolic) mode.
$nR	Set the display radix for numbers to "n". The number "n" is decimal. ~~It is not meaningful for "n" to be larger than decimal 10.~~
$S	Display the contents of locations in symbolic mode, i.e., as instructions.
$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 $70 is helpful.
$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).
$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 right 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 accumulators and 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.

`adr2<adr1$nB` This is the command by which a breakpoint is installed. In this command, the address expression `adr1` specifies the location of the instruction at which to place the breakpoint. You cannot place a breakpoint at location zero.

The 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 address expression `adr2` specifies the address of the location to automatically open and display whenever this breakpoint is hit. You cannot automatically display location zero. If you omit the expression `adr2`, no location will be automatically displayed; omit the "<" character too.

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.

If a breakpoint is installed by a command with two consecutive escape characters, e.g., `$$nB`, then DDT will proceed from the breakpoint automatically. Automatic proceed continues until DDT detects that characters are available from the terminal when the breakpoint is executed. See the `$$P` description below.

The symbolic name `$nB` can be used to address the breakpoint block associated with breakpoint number "n". The breakpoint block contains several interesting data items associated with each breakpoint. For example, `$nB` contains the address of the instruction breakpointed by breakpoint "n". The locations following `$nB` 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$nB` Remove breakpoint 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]

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$X If the expression instr is a fullword quantity, then DDT will execute it as an instruction. If the expression has a zero left half, then the expression is interpreted as a repeat count as in the command n$X.

first<last>$$Z
Store zero into all words in the locations in the address range from the expression "first" to the expression "last".

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".

C.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 CR, LF, ^, 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.

---

[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.

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, at five 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 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"`	Convert "text" to Radix 50.

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, once again, is set from the first six letters of the first word that follows the TITLE statement. Once one 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

[2] If an input/output instruction is being assembled, the comma signifies the end of the device number field.

be repetitions of symbol names among the several programs. Opening one program's symbols serves to eliminate any ambiguity in such cases.

`$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:` 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`.

`sym#` While using DDT's instruction assembler (see section C.4, page 363), 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.

`$$<` 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

$1M	specifies the terminal control mask.
$2M	specifies the symbol offset quantity.
$0W	enables automatic write enable for deposits into write protected pages. Use two escapes to disable auto write enable.
$1W	enables automatic page create when pages that do not exist are examined or deposited into. Use two escapes to disable this feature.

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.

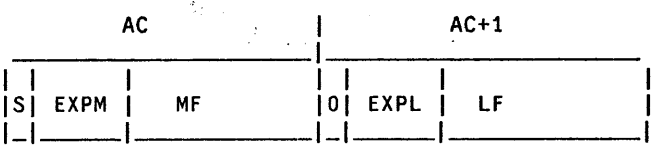
```
FADL    C(AC,AC+1) := C(AC) + C(E);   floating-point arithmetic.

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 at initialization.
- Erroneous arguments to monitor calls or 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.
- Don't 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-10 Reference Manual.* Maynard, Mass., Digital Equipment Corporation, 1978. AA-0988C-TB.

[MACRO] *MACRO-10 Assembler Reference Manual.* Maynard, Mass., Digital Equipment Corporation, 1978. AA-C780C-TB.

[MCRM] *TOPS-10 Monitor Calls Reference Manual.* Maynard, Mass., Digital Equipment Corporation, 1980. AA-0974C-TB.

[SYSREF] *DECsystem-10/DECSYSTEM-20 Processor Reference Manual.* Maynard, Mass., Digital Equipment Corporation, 1980. AA-H391A-TK.

*

Glossary

AC — An abbreviation for *accumulator.*

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.*

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-10 by which we specify the internal representation of characters. The table of ASCII characters is given in section 4.8, page 34.

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 25.

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. In the DECsystem-10 buses are used to connect the processor to the memory; the processor to I/O devices; controllers to the memory; and controllers to devices.

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 systems, 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 — 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: the interactive debugging program; see appendix C, page 359.

Decrement — Any decrease in value. Most usually a decrease in value by one.

DEC — Digital Equipment Corporation, the manufacturer of the DECsystem-10 and DECSYSTEM-20. 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. Programming this computer system is the subject of this book.

DECSYSTEM-20 — A timesharing computer system. 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. 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.

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 KL10-based systems, 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 41. In the extended PDP-10, addresses can be thirty bits.

Exec Mode — A processor mode in which the user mode restrictions do not apply. The TOPS-10 monitor normally operates in Exec mode.

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 353. See also *switch.*

Global	A symbol that is defined in one program and referenced in another. 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 global symbol is the name of a subroutine in a *library*, it must be declared as an `ENTRY` symbol. 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.
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 causes the normal instruction sequence to be changed. In TOPS-10, software interrupts can be enabled to alert the program to specific keyboard characters, incoming IPCF messages, and other events.
IPCF	Interprocess Communications Facility. A collection of monitor calls and systems programs that implement message passing between programs.
Job	In TOPS-10, a job is a software entity that consumes computer resources. A job can run a program.
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	KL10 is the manufacturing designation for the central processor that appears in the larger DECsystem-10 computers, the 1080, 1090, and 1091 configurations. The KL10 also is used in the DECSYSTEM-2040, 2050, and 2060 configurations.
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-10. 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, the massbus controller is an RH10 or an RH20; in the KS10, it is an RH11.

---

[1] The cache is not present in the 2040 configuration.

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. A DECsystem-10 may have either core or MOS semiconductor memory, or both.

Monitor — The TOPS-10 operating system itself, as distinct from the other components of the DECsystem-10 environment. See *operating system.*

MUUO — Monitor Unimplemented Operation. The MUUOs are the instructions by which a user program transfers control to the TOPS-10 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 MUUO instructions. The various different monitor functions are selected by the choice of a particular MUUO. Most often, these functions are simply called by their names, e.g., `OUTSTR`, `OPEN`, etc.

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 timesharing 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-10 operating system schedules hardware resources and provides the services requested via MUUO 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 — A page is `512` computer words. Programs in memory are composed of pages.

Page Map — In TOPS-10, a page map describes the actual pages that compose a program in memory. The page map of a program contains pointers to the specific pages in the virtual address space of that job.

PC — An abbreviation for *Program Counter.*

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, we use the term *PDP-10* to 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.

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.

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. Or, a physical record, 128 data words, in disk 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 25.

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 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-10 is one or more disk units that together represent one file system. A file structure contains one master file directory, various other file directories, one bit table for storage allocation, and user files.

A *Mountable Structure* is a file structure that can be mounted on (or dismounted from) physical disk units at will.

Switch — A means of controling 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-10 executes to grant the program the service that it requested. Sometimes, TOPS-10 decides that one program has run long enough; TOPS-10 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-10, the operating system is TOPS-10.

TOPS-10 — The operating system, utility programs, and environment found in the DECsystem-10 computer systems.

TOPS-20 — The operating system, command scanner, utilities, and environment found in the DECSYSTEM-20.

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-10, each word contains thirty-six bits and has its own address. In the DECsystem-10 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

ADD Instructions 163
ADJBP Instruction 111
ADJSP Instruction 75
AND Instructions 146
ANDCA Instructions 146
ANDCB Instructions 146
ANDCM Instructions 146
AOBJN Instruction 56
AOBJP Instruction 56
AOJ Instruction Class 54
AOS Instruction Class 52
ASCII Pseudo-op 33
ASCIZ Pseudo-op 16, 20, 33
ASH Instruction 161
ASHC Instruction 161

BLOCK Pseudo-Op 72
BLT Instruction 157
BYTE Pseudo-op 154

CAI Instruction Class 55
CALL Definition for PUSHJ 134
CAM Instruction Class 55
CLOSE MUUO 218
CLRBFI MUUO 204
COMMENT Pseudo-op 19

DADD Instruction 166
DDIV Instruction 166
DEFINE Pseudo-op 173
DEPHASE Pseudo-op 316
DF - Double-Precision Floating-Point Instructions 170
DFN Instruction 371
DIV Instructions 165
DMOVE Instructions 165
DMUL Instruction 166
DPB Instruction 108
DSUB Instruction 166

END Pseudo-op 17, 20
ENTRY Pseudo-op 201

EQV Instructions 146
EXCH Instruction 49
EXIT MUUO 21
EXTERN Pseudo-op 201

F - Floating-Point Instructions 170
FIX Instruction 171
FIXR Instruction 171
FLTR Instruction 171
FSC Instruction 172
FxxL Long Floating-Point Instructions 370

GETSTS MUUO 218

H - Halfword Instruction Class 121
HALT Instruction 130

IBP Instruction 108
IDIV Instructions 164
IDPB Instruction 109
IFDEF Conditional 175
IFE Conditional 175
IFG Conditional 175
IFGE Conditional 175
IFL Conditional 175
IFLE Conditional 175
IFN Conditional 175
IFNDEF Conditional 175
ILDB Instruction 109
IMUL Instructions 164
INBUF MUUO 219
INCHWL MUUO 65
INTERN Pseudo-op 201
IOR Instructions 146
IOWD Pseudo-op 73

JCRY0 Instruction 132
JCRY1 Instruction 132
JEN Instruction 130
JFCL Instruction 132
JFFO Instruction 133
JFOV Instruction 132
JOV Instruction 132

JRA Instruction 369
JRST Instruction 50, 129
JRSTF Instruction 130
JSA Instruction 369
JSP Instruction 131
JSR Instruction 130
JUMP Instruction Class 50

LDB Instruction 108
LIT Pseudo-op 90
LOOKUP MUUO 261
LSH Instruction 160
LSHC Instruction 160
LUUO Instructions 193

MOVE Instruction Class 47
MUL Instructions 165
MUUO Instructions 20

OPDEF Pseudo-op 134
OPEN MUUO for Input 261
OR Instructions 146
ORCA Instructions 146
ORCB Instructions 146
ORCM Instructions 146
.ORG Pseudo-op 317
OUTBUF MUUO 219
OUTCHR MUUO 62
OUTPUT MUUO 216
OUTSTR MUUO 15, 21

PHASE Pseudo-op 316
POINT Pseudo-op 110
POP Instruction 71
POPJ Instruction 126
PORTAL Instruction 130
PUSH Instruction 71
PUSHJ Instruction 126

RADIX50 Pseudo-op 384
RELEAS MUUO 218
RENAME MUUO 312
REPEAT Macro Operator 239
.REQUEST Pseudo-op 245
RESET MUUO 21
RET Definition for POPJ 134
ROT Instruction 161
ROTC Instruction 161

SEARCH Pseudo-op 177
SETA Instructions 146
SETCA Instructions 146
SETCM Instructions 146
SETM Instructions 146
SETO Instructions 146
SETZ Instructions 146
SIXBIT Pseudo-op 215
SKIP Instruction Class 52
SOJ Instruction Class 54
SOS Instruction Class 53
STATO MUUO 218
STATZ MUUO 218
SUB Instructions 163
SUBTTL Pseudo-op 200

T - Test Instruction Class 143
TITLE Pseudo-op 19

UFA Instruction 371
USETI MUUO 347
USETO MUUO 347

XCT Instruction 133
XJRSTF Instruction 364
XOR Instructions 146

*

Index

.JBDDT, Job DDT Starting Address word 94
.JBOPC, Job Old PC word 94

! OR Operator in MACRO 147, 186
& AND Operator in MACRO 146
^! XOR Operator in MACRO 147
^- NOT Operator in MACRO 148
^D Force Decimal Radix in MACRO 179
^L JFFO Operator in MACRO 133
_ (Underscore) Shift Operator in MACRO 160

Absolute Address 89
AC (Accumulator) 11, 377
AC (Accumulator) Field in Instructions 37
Accumulator 11, 377
ADD Instruction Class 163
Address 5, 8, 377
Address Break Trap 355
Address Polynomials, Array Addressing via 240, 254
Addressing, Indexed 42
Addressing, Indirect 45
ADJSP instruction 75
Algorithm 377
AND Instructions 146
AND Operator in MACRO, & 146
ANDCA Instructions 146
ANDCB Instructions 146
ANDCM Instructions 146
AOBJN, AOBJP Instructions 56
AOJ Instructions 54
AOS Instructions 53
Arithmetic in Binary 25
Arithmetic in Octal 31
Arithmetic Instructions 163
Arithmetic Overflow 30
Arithmetic Overflow Flag in PC, AROV 126
Arithmetic Overflow Flag in PC, TRAP1 126
Arithmetic Shift Instructions 161
Arithmetic, Floating-Point 166
AROV, PC flag 126
Array 225
Array Addressing via Indirect Addressing 237
Array Addressing via Side-Tables 237
Array, Efficiency Considerations 255
Array, Index Register to Address 42
Array, Multi-Dimensional 254
Array, Two-Dimensional 237
ASCII Character Table 34
ASCII Characters 33
ASCII Pseudo-op 33
ASCIZ Pseudo-op 16, 20, 33
ASH Instruction 161
ASHC Instruction 161
Assembler Program 2, 377
Assembler Program, Functions of 16
Assembly Language 2
Assembly of Instructions into Memory Words 37
Assembly Switch 175

Base Eight (Octal) 31
Base Two Numbers 25
Binary Number System 25
Binary Relocatable File 91
Bit 5, 377
BLOCK Pseudo-Op 72
BLT Instruction 157
Boolean Instructions 146
Bottom Test in Loop 82
Breakpoint 93
B Shifting Operator in Numbers 202
Bucketing 317
Bus 378
Byte 107, 378
Byte Instructions 107
Byte Instructions, example 112
Byte Pointer 107
Byte Pointer, Assembly of 110
BYTE Pseudo-op 154

C (Channel) Bus 378
Cache Memory 255
CALL Definition for PUSHJ 134
Central Processing Unit 9, 378
Character 378
Character Output to Terminal, OUTCHR 62

Character Representation, ASCII Code 33
Character Table, ASCII 34
CLOSE MUUO 218
CLRBFI MUUO 204
Column Major Form, Array Storage 237
COMMENT Pseudo-op 19
Comments in programs 19
Compare Instructions 55
Comparisons, CAI Class 55
Comparisons, CAM Class 55
Complement 378
Concealed Program 130
Conditional Assembly 175
Conditional Jumps, AOJ Class 54
Conditional Jumps, JUMP Class 50
Conditional Jumps, SOJ Class 54
Conditional Skips, AOS Class 53
Conditional Skips, SKIP Class 52
Conditional Skips, SOS Class 54
Context Switch 84
Control Characters in ASCII 33
Control Structures, Bottom Test Loop 82
Control Structures, Flags 119
Control Structures, Subroutines 126
Control Structures, Top Test Loop 82
Conversion, Fixed-Point to Floating-Point 171
Conversion, Floating-Point to Fixed-Point 171
Core-Image 85
CPU 9, 378
CREF Listing 89
Cross-Reference Listing 89

DADD Instruction 166
Data Movement, MOVE Class 47
Data Movement, EXCH Instruction 49
Data Representation, ASCII Characters 33
Data Representation, Floating-Point 166
Data Structure, Array 225
Data Structure, Linked Lists 313
Data Structure, Records 285, 313
Data Structures, Use of Macros for 184
Data, Representation of 25
DCK, PC flag 126
DDIV Instruction 166
DDT 3, 93, 359
Debugging using DDT 93, 359
Decimal Input Scanner 184
Decimal Output Printer 179
Decimal Radix, ^D in MACRO 179
DECsystem-10 9
DEFINE Pseudo-op 173
DEPHASE Pseudo-op 316
DFN Instruction 371
Directory of Files 285
DIV Instruction Class 165
Divide by Zero, DCK Flag in PC 126
Divide Check Flag in PC, DCK 126
DMOVE Instruction Class 165
DMUL Instruction 166
Documenting programs 19
Dormant Page 255
Double-Precision Fixed-Point Arithmetic 166
Double-Precision Floating-Point 168
Double-Word Instructions 165
DSUB Instruction 166
Dump-Mode Command List 295
Dynamic Space Allocation 286

E (Execution) Bus 379
E Box 379
Effective Address 41
Effective Address, Examples 42
Effective Addressing, Index Registers in 42
Effective Addressing, use of Indirect Address 45
Effective Index 227
Efficiency 84, 255, 324
END Pseudo-op 17, 20
ENTRY Pseudo-op 201
EQV Instructions 146
Example 1 15
Example 2-A 59
Example 2-B 61
Example 3 65
Example 4-A 79
Example 4-B 112
Example 5 115
Example 6-A 134
Example 6-B 148
Example 7 176
Example 8 194
Example 9 213
Example 10 219
Example 11 227
Example 12 241
Example 13a 262
Example 13b 265
Example 14 285
Example 15 300
Example 16 314
Exercises 22, 35, 63, 120, 155, 172, 235, 256, 281, 309, 342
EXIT MUUO 21
Exponent, Floating-Point Numbers 166
EXTERN Pseudo-op 201
External Symbols 201

File Directory Processing 285
File Input 261
File, Universal 385
FIX Instruction 171
Fixed-Point Addition 163
Fixed-Point Arithmetic 163
Fixed-Point Arithmetic, Double-Precision 166
Fixed-Point Division 164, 165
Fixed-Point Multiplication 164, 165
Fixed-Point Subtraction 163
Fixed-Point to Floating-Point Conversion 171
FIXR Instruction 171
Flag, Use of 119
Flags, PC 125, 353
Floating Overflow Flag in PC, FOV 126

Floating Underflow Flag in PC, FXU 126
Floating-Point Arithmetic Exceptions 170
Floating-Point Input Scanner 205
Floating-Point Instructions 170
Floating-Point Output Printer 206
Floating-Point to Fixed-Point Conversion 171
Floating-Point, Double-Precision 168
Floating-Point, Normalization of Results 169
FLTR Instruction 171
Fortran Library 245, 251
FOV, Overflow in Floating-Point 170
FOV, PC flag 126
Fraction, Floating-Point Numbers 166
Free Space Management 286
FSC Instruction 172
FXU, PC flag 126
FXU, Underflow in Floating-Point 170
FxxL Long Floating-Point Instructions 370

GETSTS MUUO 218
Global Symbols 201

Half-killed Symbol 105
Halfword Instructions 121
HALT Instruction 130
Hardware Instructions 2
Hash Bucket 317
Hash Code 317
Hash Index 317
Hashing Function 317
Heapsort 300
Help! 93, 359, 373

I (Indirect) Field in Instructions 37
IDIV Instruction Class 164
IFDEF Conditional 175
IFE Conditional 175
IFG Conditional 175
IFGE Conditional 175
IFL Conditional 175
IFLE Conditional 175
IFN Conditional 175
IFNDEF Conditional 175
IMUL Instruction Class 164
INBUF MUUO 219
INCHWL MUUO 65
Index 388
Index of Instructions 387
Index Register 11
Index Register in Effective Address 42
Indexed Addressing 42
Indirect Addressing 45
Indirect Addressing, of Arrays 237
Input from File 261
Input Operations via JSYS Instructions 3
Input, Characters from Terminal, INCHWL 65
Instruction 1, 2, 9, 37
Instruction Fields 37
Instruction Index 387
Instruction Nomenclature 357
Instruction Set 2
INTERN Pseudo-op 201
Internal Symbols 201
IOR Instructions 146
IOWD 295
IOWD Pseudo-op 73

.JB41 LUUO Trap Instruction 201
.JBFF Job First Free Address 286, 318
.JBREL Job Highest Address 286
.JBSA Job Start Address 286
.JBUUO LUUO Image 201
JCRY0 Instruction 132
JCRY1 Instruction 132
JEN Instruction 130
JFCL Instruction 132
JFFO Instruction 133
JFFO Operator in MACRO, ^L 133
JFOV Instruction 132
Job Data Area (JOBDAT) 22, 201, 286
Job Data Area, .JBDDT 94
Job Data Area, .JBOPC 94
JOBDAT, Job Data Area 22, 201, 286
Joke 3, 27, 323
JOV Instruction 132
JRA Instruction 369
JRST Instruction 50, 129
JRSTF Instruction 130
JSA Instruction 369
JSP Instruction 131
JSR Instruction 130
JSYS Instructions 3
Jump Instructions 50

LINK Program 91
List, Data Structure 313
Lists, Sort by Merging 336
LIT Pseudo-op 90
Literal in Assembler 66
Loader Program 91
Location Counter 316, 317
Logical Shift Instructions 160
LOOKUP MUUO 261
Lookup, Hash Code 317
Loop Control: AOBJN and AOBJP 56
Loop, Bottom Test 82
Loop, Top Test 82
Loops in Programs 57
LSH Instruction 160
LSHC Instruction 160
LUUO Instructions 193

M Box 381
Machine Language 1
Macro Facility 173
Macro, Definition of 173
Macros, Arguments to 174
Massbus 381
Massbus Controller 381
Matrix 237

Memory 5
Memory Management 286
Merge Sorting of Lists 336
MUL Instruction Class 165
MUUO Instructions 15, 20
MUUO Operations, Overhead of 84

Negative Numbers, Representation of 27
Nested Skips 56
Nesting of Subroutines 128
No Divide, DCK Flag in PC 126
No-op 46, 132
Normalization of Floating-Point Results 169
NOT Operator in MACRO, ^- 148
Notation for Instruction Descriptions 48, 357
Number Conversions 171

Octal Representation 31
One's Complement 378
OPDEF Pseudo-op 134
OPEN MUUO for Input 261
Operating System 3, 209
OR Instructions 146
OR Operator in MACRO, ! 147, 186
ORCA Instructions 146
ORCB Instructions 146
ORCM Instructions 146
.ORG Pseudo-op 317
OUTBUF MUUO 219
OUTCHR MUUO 62
OUTPUT MUUO 216
Output Operations via JSYS Instructions 3
Output, Character to Terminal, OUTCHR 62
Output, Efficiency of Terminal 84
Output, String to Terminal, OUTSTR 21
OUTSTR MUUO 15, 21
Overflow 30
Overflow, Arithmetic 30
Overflow, Arithmetic, AROV Flag in PC 126
Overflow, Arithmetic, TRAP1 Flag in PC 126
Overflow, Floating, Flag in PC 126
Overflow, Floating-Point 170
Overflow, Pushdown, Flag in PC 126
Overhead of MUUO Operations 84

Page Fault 255
Pagination of Program Source File 200
PC 9, 125, 353
PC Flags 125, 353
PDP-10 9
Phase Location Counter 316
PHASE Pseudo-op 316
POINT Pseudo-op 110
Pointer, Byte 107
POP Instruction 71
POPJ Instruction 126
PORTAL Instruction 130
Processor (CPU) 9, 378
Program Control: AOBJN and AOBJP 56
Program Control: AOJ Class 54
Program Control: AOS Class 53
Program Control: CAI Class 55
Program Control: CAM Class 55
Program Control: JRST Instruction 50
Program Control: JUMP Class 50
Program Control: SKIP Class 52
Program Control: SOJ Class 54
Program Control: SOS Class 54
Program Counter 9
Program Counter Format 125, 353
Program Organization 200
Program Starting Address 17
Program Symbol Table Name 19
Pseudo-Operators 19
Pseudo-Operators in the Assembler 16
Pseudo-ops 16, 19
Public (Unconcealed) Program 130
PUSH Instruction 71
Pushdown List 71
Pushdown Overflow Flag in PC, TRAP2 126
PUSHJ Instruction 126

Record Pointer 313
Record, Data Structure 313
Records, Index Register to Access 44
Recursion 129, 182
Recursive Subroutine 129
Recursive Subroutines 179, 182
Register 10, 11
Register, Index, in Effective Address 42
REL File 91
RELEAS MUUO 218
Relocatable Address 89
Relocatable Code 91
Relocatable File 91
Relocation Constant 92
RENAME MUUO 312
REPEAT Macro Operator 239
Representation of Characters, ASCII Code 33
Representation of Data 25
Representation of Instructions in Memory 37
Representation, Floating-Point 166
.REQUEST Pseudo-op 245
Reserve Space in Memory 72
RESET MUUO 21
RET Definition for POPJ 134
Return from Subroutine 126
Return, Skip, from Subroutine 128
ROT Instruction 161
Rotate Instructions 161
ROTC Instruction 161
Row Major Form, Array Storage 237

S (Storage) Bus 383
Satisfied Condition 175
SEARCH Pseudo-op 177
Search, Hash Code 317
Semicolon Character, Meaning of 19
SETA Instructions 146
SETCA Instructions 146