System/360-370
Assembler Language
(DOS)

System/360-370 Assembler Language (DOS)

Kevin McQuillen

Mike Murach & Associates, Inc.
1067 West Morris Avenue
Fresno, California 93705

Printed in the United States of America.
Library of Congress Catalog Card
Number: 74-76436

Thanks to IBM and Science Research Associates, Inc. for permission to reprint or adapt the materials listed.

Reprinted by permission of International Business Machines Corporation: figures 1-1, 1-7, 2-5, 2-6, 2-15, 6-3, 9-9, 9-15, 9-16, 10-12, 10-13, 10-14, 11-1, 12-2, 12-4, 12-5, 12-6, 12-11, 14-2, 14-6, 14-13, 14-14, 15-10, 15-12, 15-19, 15-20, 16-8, 16-10, A-1, A-2, and A-3; and forms GX24-6599, GX20-1776, GX28-6509, and GX20-8021.

From PRINCIPLES OF BUSINESS DATA PROCESSING.

Contents

Preface for Instructors

System/360-370 Assembler Language (DOS) has been designed for use in both introductory and advanced courses. Specifically, this book teaches System/360-370 assembler language for the DOS or DOS/VS operating system. In chapters 1 through 5, I believe this book provides the most effective introduction to assembler-language programming that is currently available. In chapters 6 through 16, I believe this book provides the most effective advanced material on assembler-language programming that is on the market today.

Because most students taking this course will have taken a previous course in data processing or programming, or will have had equivalent experience in the field, it is assumed that the readers have certain skills. In particular, the readers should be able to do the following:

1 Describe the components of a typical card, tape, or direct-access system.

2 Explain what is meant by the term *continuous form*.

3 Decode the data in an uninterpreted punched card.

4 Describe how data is keypunched into a standard punched card.

5 Explain what is meant by *loading a program*.

6 Describe blocked tape records.

Since this is a very limited set of requirements, it is possible to use this book for a first course in computing by giving the class a computer and keypunching demonstration plus an introductory lecture that provides the background material indicated above. Similarly, if the courses previously taken by some students haven't covered all the material required by these prerequisites, the omitted material can be covered in a single lecture or demonstration. If all six prerequisites should have been met in a previous course, you may want to give a pretest on the first day of class to make sure that they actually have been met; you can then review accordingly.

This is the first book published by our company, and it was developed in a way that I think adds some much needed professionalism to the process of preparing instructional materials. To begin with, more than three months were spent in analyzing the possible subjects to be included in this book, selecting the actual content, and organizing that content based on a theory of instruction. Since few writers have the luxury of this much time for planning, one of the major shortcomings of most texts is the content selection and organization. Second, this book was written by a first-rate industry specialist, Kevin McQuillen, on a full time basis. In contrast, many books in the field are written on nights and weekends as secondary projects to some full time job. Finally, Kevin's manuscript was extensively rewritten and edited in an attempt to bridge the gap between professional and novice. I think the resulting product indicates that this method of preparing instructional materials is an effective one.

BOOK FEATURES

Content Selection

One important feature of this book is the breadth and usefulness of its content. This is true because the content was selected based on an analysis of the tasks done by a professional assembler-language programmer. For each task required of him, there is explanatory material in this book. As a result, I think you will find that this book is the most complete assembler-language book currently available. If you check the table of contents, you will find material on diagnostics, debugging, tape and disk concepts, operating systems, job-control language, and keypunching. Although all of these subjects are related to essential programming tasks, it is common for one or more of them to be omitted from assembler-language texts.

On the other hand, subjects that aren't related to the tasks of a professional programmer have been omitted from this text. This makes the book relevant. Although you may think that all programming books use task analysis as a guideline for selecting content, I can find numerous examples of irrelevant material in the sampling of assembler-language books that I have in my library. Multiplying binary numbers, decoding the PSW word, writing channel commands, coding privileged instructions, using physical IOCS, knowing the interrupt system—all are irrelevant to the assembler-language programmer but are included in one text or another. (Granted some of these are related to the tasks of a software specialist, but that isn't the context in which these subjects are presented.)

The only material in this book that isn't based on a strict task analysis is the topic on floating-point arithmetic and the topic on advanced macro writing. Although these topics are more relevant to the software specialist than to the assembler-language programmer, they have been included to give some important exposure. Otherwise, you can be sure that the coding and techniques illustrated in this book are also those found in industry. In contrast, a student who uses another text will all too often discover that the techniques illustrated are not those of the real world.

Modular Organization

A second feature of this book is its organization, sometimes referred to as "modular" organization. After reading the first five chapters of the text, the student can continue with any of its other parts, or modules. In particular, the book is organized as follows:

Part	Chapters	Part Title	Prerequisite Parts	Design
1	1-2	Required Background	—	Sequential
2	3-5	Assembler Language: The Core Content	1	Sequential
3	6-11	Advanced Assembler-Language Subjects	1,2	Random
4	12-13	Magnetic Tape Programming	1,2	Sequential
5	14-15	Direct-Access Programming	1,2	Sequential
6	16	The Operating System	1,2	

This means that the chapters in parts 4 and 5 should be studied in sequence, but the chapters in part 3 can be studied in any order. Within the few limitations indicated in this table, it is you who will determine the sequence of instruction and the material to be emphasized. Similarly, the student is free to jump to a topic that interests him without fear of missing related background material. In short, the course can be teacher-directed or student-directed, but it will not be textbook-directed.

In addition to the teaching flexibility that modular organization gives, there is an important educational reason for organizing a book in this way. Briefly stated, this organization forces the author to present the essence, or "core content," of the subject in just a few chapters early in the book. This in turn means that the student is shown all the important relationships between the elements of the subject early in the course. Because one of the major problems of learning is the failure to see the relationships between the parts, the emphasis on core content makes learning more efficient.

I might add that although many books are advertised as modular, few actually are. To be truly modular, the essence, totality, or core content of the subject (call it what you will) must be presented early in the course and all subsequent modules must need only this core content as prerequisite material. When a book is designed in this way, its format proceeds from a theory of education rather than from marketing considerations.

Educational Methodology

I think the primary feature of this book is that it works—you can actually learn how to program in assembler language from it. I know because I learned DOS assembler language from Kevin's raw manuscript without any outside help of any kind. Since that time, we've refined the manuscript considerably and have made it more effective didactically.

As I see it, there are two main reasons for this book's effectiveness. One of these reasons is its modular approach. In general, there are two basic approaches to teaching assembler language—the parts-to-the-whole method and the modular method. The first teaches the separate elements of the language until a great deal of detail has been covered and only at that time are a few of these elements put together in a complete program. Using this approach, it isn't uncommon for the first complete program to be presented in the second half of the book, and sometimes the first complete program is presented very late in the book. For instance, one of the leading texts presents its first complete program on page 285, while another presents its first complete program in chapter 14 of a 15-chapter book.

The problem with this parts-to-the-whole method of teaching is twofold. First, a student doesn't have the perspective to appreciate the relationships between the parts until he is familiar with a complete program. As a result, he learns the parts through memory rather than through some underlying structure or concept. Second, from a point of view of classroom teaching, this method is impractical. Normally, you must wait at least several weeks before a student has learned enough of the parts in order to be able to write a complete program. In the meantime, motivation dwindles, and what should be an exciting problem-solving class becomes a frustrating struggle to learn the massive amount of detail associated with the language. Also, if the assignment of computer laboratory time is a fixed number of hours per week from the start of the course, the instructor usually has to create supplementary material so students can run programs, or segments of programs, in the early weeks of the course.

As indicated earlier, this book uses the second (modular)

approach to the teaching of assembler language. After some background material is provided in chapters 1 and 2 (some of which may be review), topic 1 of chapter 3 presents a complete program including card input, printer output, data movement, editing, arithmetic, and logic. As soon as this basic program is understood, the student can begin to write significant programs of his own. Before chapter 3 is completed, though, two refinements of this first program and two additional programs are presented so a total of five complete programs are shown in chapter 3. By this time, a full subset of the language has been presented, and the student is prepared to do independent work in a computer lab.

Following the subset presented in chapter 3, chapter 4 offers a definitive presentation on correcting diagnostics, preparing test data, and debugging programs, including the analysis of core dumps. Since these skills are essential to assembler-language programming and to successful lab work, it is amazing that they are often treated so lightly in other texts. To complete the core content, chapter 5 presents a collection of elements and techniques that expands the subset and makes the book truly modular.

Once a student has satisfactorily completed part 2—in particular, chapters 3 and 4—the most demanding part of the instructor's job is finished. With an understanding of the structure of the language and all the related skills for coding, testing, and debugging a program, learning other assembler-language elements and techniques becomes part of a pattern. If a student can see the need for an element or technique and can see how that element or technique relates to the whole task of programming, mastering the material is a manageable task.

The second reason for the book's didactic effectiveness is its illustrative material. Although many authors rely heavily on verbal description, we know from experience that a programming language cannot be learned without extensive illustrative material. In fact, the illustrative material is far more important than the descriptive material. For this reason, this book has a carefully planned sequence of program listings. In all, there are 24 complete program listings, 45 self-contained segments of coding, several diagnostic and debugging listings, and numerous supporting examples. As soon as I saw the illustrative material that Kevin had selected for this book, I knew the product would be effective.

Apparatus by Topic

Because learning depends on what the student does, not upon what he sees or hears, each topic is followed by terminology lists, behavioral objectives, and, whenever relevant, problems and their solutions. The terminology lists are listings of the new words presented in the text. The intent is for the student to scan the list to check his comprehension of the terminology. If he understands the words, he can proceed. If he feels that he doesn't have a clear understanding of a term, he can reread applicable sections or note the term so he can question its meaning in class. In any case, the intent of the list is for use as a quick review; a student shouldn't be expected to write definitions of the words.

Following the terminology lists are behavioral objectives that describe the activities a student should be able to do upon completion of the topic. The intent of these objectives is to give the student a clear picture of what his learning goals should be. Since this book deals with programming, the primary objectives have to do with solving various types of programming problems using assembler language. In addition, there are objectives that deal with related skills such as reading core dumps, describing a type of file organization, choosing blocking factors for disk files, and preparing DOS job-control cards. Although some students will ignore the objectives, others will be more efficient learners because of them.

Although some instructors seem to feel that preparing and using objectives is busywork, I would like to see them in all textbooks. At the least, listing objectives would force the author to focus more clearly on what he is trying to accomplish. Without objectives, I think an author all too often concerns himself with writing a definitive work rather

than concentrating on the goals of education.

At any rate, I believe objectives can contribute significantly to the success of a course. If students are convinced that the objective lists describe *all* activities that they will be expected to perform, their learning will become much more directed. In each class I've taught, I have found students who wouldn't rest until they felt they could satisfy all of the course objectives. Because some students find it hard to believe an instructor is telling them everything they will be required to do, it is important for you to refer to the objectives as you review a topic in class. If the objectives are made prominent in all classroom activity, I am convinced that teaching has a greater likelihood of success.

Since it is unlikely that two people will agree on a list of objectives for a course, you will probably want to modify the objectives to suit your class. If, for example, you are using the text for a computer science course, you may want to add objectives that emphasize software development. The objective lists, then, are only a starting point. However, if only the objectives given in the text are fulfilled, I would say that you have taught a highly successful course.

When the objectives deal with problem solving, they are followed by problems and their solutions. These problems are intended to give practice in the skills described by the objectives. As much as possible, these problems have been designed to show how the elements and techniques described in the text are used in a different context. There are no multiple-choice, true/false, matching, or fill-in questions, because those types of activities have nothing to do with the important objectives of a programming course.

So there is immediacy to the problem-solving activity, solutions are presented immediately after the problems. This has the advantage of letting a student know that he is right when he is right, and just as important, of letting him know right away when he's wrong. For those students who wouldn't otherwise know how to get started in solving a problem, the solutions are an essential part of the learning process. Although compiling and testing programs on a computer system has the same effect as doing the problems and checking the solutions, studying the solutions of a professional can correct many false notions and bad habits before problems are actually tried on a computer system. Because of the expense of computer time, this is a practical consideration.

What about students who don't actually do the problems but look to the solutions? The experience is still valuable. Although the best way to learn is to actually do the problems and then compare the answers with the solutions provided, it may not always be the most efficient way of learning—particularly, for the brightest students or for those with experience in another language. In the interest of expediency, then, a student may read a problem, conceive a solution, and compare this conception with the actual solution. The important thing is that assembler language be viewed in the context of its application. Looking at the problems in this way, they can be seen simply as a means of presenting additional assembler-language applications.

Lab Problems

Appendix F presents a progression of programming problems. Since these problems include test data listings, they are ideal for lab problems. In addition, they can be used for classroom exercises or tests. If a student can write programs for all of the types of problems given, I feel sure he is well qualified to become an entry-level programmer in industry.

TEACHING NOTES

Because there is no instructor's guide for this text, we have tried to make the text itself as complete—both for teacher and student—as is practical. For this reason, the following teaching notes are included in this preface.

1 If students taking this course have a strong background in data processing or programming, it is possible that they will have already mastered the material in chapter 1 (introductory concepts), chapter 12 (tape concepts), and chapter 14 (direct-access concepts). For this reason,

you may want to give a pretest for these chapters based on the objectives at the ends of the topics. To a lesser extent, the students may also be familiar with the material of chapter 2 (CPU concepts) and chapter 16 (DOS and JCL). Here again, a pretest can determine which students need to master which objectives.

2 One of the problems you may encounter when conducting the computer lab is that the necessary I/O modules aren't in the relocatable library. As a result, the object programs won't link-edit properly.

For this reason, table 1 in this preface lists all of the I/O modules that might be needed for running the solutions to the lab problems of appendix F. This table also includes all the modules used by the programs illustrated in this text with the exception of the variable-length tape programs and DA-file disk programs. To find out which of these modules are already stored in the relocatable library of your system, you can use the DSERV program as follows:

```
// JOB DISPLAY DIRECTORY
// EXEC DSERV
   DSPLY RD
/*
/&
```

If modules are missing, you can either assemble the modules and add them to the library or you can direct the students to code their DTFs using operands that correspond to the modules already in the library.

To catalog a module in the relocatable library, an object module is first created using a job deck like this:

```
// JOB ASSEMBLE I/O MODULE
// OPTION DECK,NOSYM
// EXEC ASSEMBLY
        ISMOD SEPASMB=YES,IOROUT=LOAD
        END
/*
/&
```

In this example, the I/O module named IJHZLZZZ is created. Then, the object module can be catalogued in the relocatable library with a job deck like this:

```
// JOB CATALOG
// EXEC MAINT
   CATALR IJHZLZZZ
   (IJHZLZZZ object deck)
/*
/&
```

3 If your students have keypunched source and JCL decks before, appendix A should give them all the information they need for keypunching assembler-language decks. Otherwise, a keypunching demonstration that includes the mounting of a program card on a program drum should be given.

4 Because it can make the running of lab problems more efficient, I recommend that tape or sequential-disk programming be taught immediately after the core content. Then, a card-to-tape or card-to-disk utility can be used to store the test decks in tape or sequential-disk files, and the problems for chapters 6 through 12 can be run as tape-to-printer or disk-to-printer programs rather than card-to-printer programs. In general, this presents no conceptual problems for the students, but it can greatly improve computer efficiency.

5 Although the material in this text is self-sufficient, there are several references to IBM manuals that provide additional information. As a result, one or more copies of the relevant manuals should be available to the members of the class. These manuals are listed at the end of the introduction.

6 Because of its modular organization, didactic approach, illustrative material, and apparatus, I believe some new solutions to old teaching problems are possible when this text is used. Perhaps the most significant teaching problem encountered in a programming course is the range of aptitudes of the students. For instance, some students will grasp the material by merely reading the

TABLE 1. Modules for Illustrations and Lab Problems

DTF Options	Module Name	Assembly Statement
Card I/O Modules:		
One I/O area, no work area	IJCFZIZ0	CDMOD SEPASMB=YES
One I/O area, work area	IJCFZIW0	CDMOD SEPASMB=YES,WORKA=YES
Two I/O areas, work area	IJCFZIB0	CDMOD SEPASMB=YES,IOAREA2=YES,WORKA=YES
One I/O area, no work area, output file	IJCFZOZ0	CDMOD SEPASMB=YES,TYPEFLE=OUTPUT
Printer I/O Modules:		
One I/O area, no work area	IJDFZZZZ	PRMOD SEPASMB=YES
Two I/O areas, work area	IJDFZZIW	PRMOD SEPASMB=YES,IOAREA2=YES,WORKA=YES
One I/O area, PRTOV macro	IJDFZPZZ	PRMOD SEPASMB=YES,PRINTOV=YES
One I/O area, no work area, CNTRL and PRTOV macros	IJDFCPZZ	PRMOD SEPASMB=YES,CONTROL=YES,PRINTOV=YES
One I/O area, work area, CNTRL and PRTOV macros	IJDFCPZW	PRMOD SEPASMB=YES,CONTROL=YES,PRINTOV=YES, WORKA=YES
Two I/O areas, work area, CNTRL and PRTOV macros	IJDFCPIW	PRMOD SEPASMB=YES,CONTROL=YES,PRINTOV=YES, IOAREA2=YES,WORKA=YES
One I/O area, no work area, ASA control characters	IJDFAZZZ	PRMOD SEPASMB=YES,CTLCHR=ASA
One I/O area, work area, ASA control characters	IJDFAZZW	PRMOD SEPASMB=YES,CTLCHR=ASA,WORK=YES
Two I/O areas, work area, ASA control characters	IJDFAZIW	PRMOD SEPASMB=YES,CTLCHR=ASA,IOAREA2=YES, WORKA=YES
Tape I/O Modules:		
Fixed-length, no work area	IJFFZZZZ	MTMOD SEPASMB=YES
Fixed-length, work area	IJFFZZWZ	MTMOD SEPASMB=YES,WORKA=YES
Sequential Disk Modules:		
Fixed-length, input file	IJGFIZZZ	SDMODFI SEPASMB=YES
Fixed-length, output file	IJGFOZZZ	SDMODFO SEPASMB=YES
Fixed-length, update file	IJGFUZZZ	SDMODFU SEPASMB=YES
ISAM Disk Modules:		
Load, one I/O area	IJHZLZZZ	ISMOD SEPASMB=YES,IOROUT=LOAD
Load, two I/O areas	IJHZLGZZ	ISMOD SEPASMB=YES,IOROUT=LOAD,IOAREA2=YES
Sequential retrieval, one I/O area	IJHZRSZZ	ISMOD SEPASMB=YES,IOROUT=RETRVE,TYPEFLE=SEQNTL
Sequential retrieval, two I/O areas	IJHZRGZZ	ISMOD SEPASMB=YES,IOROUT=RETRVE,TYPEFLE=SEQNTL, IOAREA2=YES
Random retrieval, no cylinder index in core	IJHZRRZZ	ISMOD SEPASMB=YES,IOROUT=RETRVE,TYPEFLE=RANDOM
Random retrieval, cylinder index in core	IJHZRRCZ	ISMOD SEPASMB=YES,IOROUT=RETRVE,TYPEFLE=RANDOM, CORINDX=YES
Add unblocked records	IJHUIZZZ	ISMOD SEPASMB=YES,IOROUT=ADD,RECFORM=FIXUNB
Add blocked records	IJHBIZZZ	ISMOD SEPASMB=YES,IOROUT=ADD,RECFORM=FIXBLK

text, some will require minor assistance in addition to reading the text, some will require extensive help in the form of lecture and discussion, and some just shouldn't be taking the course. Because this text gets into the problem-solving aspects of the subject in chapters 2 and 3, the aptitudes of your students should be apparent early in the course, in time for effective counseling. Then, the brightest students can be assigned lab problems and be allowed to work independently; others can be assigned less demanding problems and given periodic assistance; and the marginal students can be given full assistance and supervision.

CONCLUSION

In conclusion, I'd like to say that we have tried very hard to make this text as effective as possible. Nevertheless, I know that we have much to learn about preparing instructional materials. To this end, we intend to do controlled tests of this product in order to see at what points in this text learning problems develop. Maybe then we can improve our methodology in future editions and in future products. We also welcome your comments, criticisms, suggestions, or questions.

Mike Murach
Fresno, California

Introduction

In the mid-1960s, computer industry experts began to predict that the use of assembler language would decline and die within ten years or so. Today, various experts continue to predict the replacement of assembler languages by various software or hardware developments. In fact, however, assembler language is the second most widely used programming language for business applications—second only to COBOL. Furthermore, the use of assembler language does not appear to have declined in the last three years.

Because of this widespread use, a professional programmer for a medium or large sized computer system is almost certain to come in contact with assembler language at some time in his career. But that's only one of the reasons for studying assembler language. In addition, the ability to write and debug assembler-language programs allows you to write routines that cannot be written in high-level languages, helps you write more efficient high-level language programs, and gives you the tools with which to debug sophisticated problems that may result from using high-level-language code. In short, a high-level-language programmer isn't in complete control of a computer system unless he knows assembler language.

In case you don't know it, there are many different assembler languages. For instance, each type of computer system has its own assembler language. In addition, there may be more

than one version of assembler language for a single computer. For the IBM System/360-370, there are two main versions of assembler language. They can be referred to as the DOS and OS assembler languages. DOS assembler language is the most widely used version, and that's what this book is about.

With few exceptions, the content of this book has been selected based on frequency of use in industry. As a result, you can feel sure that this book is a true representation of what you'll find in the business world. However, because of the range of this book—it covers everything from elementary coding to advanced table handling, translating, macro writing, and disk file handling—much of the material presented here is not known or used by the average professional programmer. For this reason, I feel confident that you will have more than met the requirements of a starting programmer in industry if you are able to apply all of the material in this book to business programming problems.

Before you actually begin to use this book, there are several things you ought to know about it.

1 In most cases, a student taking this course will have taken an introductory course or will have programmed in another language. As a result, this book assumes that you can do the following:
   a. Describe the components of a typical card, tape, or direct-access system.
   b. Explain what is meant by the term *continuous form*.
   c. Decode the data in an uninterpreted punched card.
   d. Describe how data is keypunched into a standard punched card.
   e. Explain what is meant by *loading a program*.
   f. Describe blocked tape records.

   Since this is a very limited set of requirements, your background is probably much stronger than required. If so, the material will be that much easier for you. In particular, chapters 1, 12, and 14 are likely to be mainly review for you, and chapters 2 and 16 are likely to be advanced versions of material you have already been introduced to.

2 This book is not designed so that you must read its sixteen chapters in sequence. Instead, its chapters are divided into six parts as indicated by this table:

Part	Chapters	Title	Prerequisite Parts	Design
1	1-2	Required Background	—	Sequential
2	3-5	Assembler Language: The Core Content	1	Sequential
3	6-11	Advanced Assembler-Language Subjects	1,2	Random
4	12-13	Magnetic Tape Programming	1,2	Sequential
5	14-15	Direct-Access Programming	1,2	Sequential
6	16	The Operating System	1,2	

   This means that after completing the first two parts, you can continue with any other part of the book. In parts 4 and 5 the chapters should be read in sequence, but part 3 is designed so the chapters can be read in any order you choose.

   The advantage of this type of organization is that you can read the parts and chapters in the sequence that interests you, not in the sequence the author thinks is best. If, for example, you are interested in writing a file-handling program for a direct-access file, you can skip to part 5 immediately after completing part 2. Similarly, a COBOL programmer who wants to know how to write an assembler-language subprogram might skip to chapter 9 immediately after part 2.

   From an educational point of view, this method of organization is effective because it gives you an honest representation of the language early in the course. By the time you complete chapter 3, you will know how to code complete assembler-language programs for card input and printer output. Then chapter 4 shows you how to prepare and debug programs, and chapter 5 presents some

professional coding techniques. As a result, after completing part 2 you will have a good appreciation for the nature of assembler language as well as a good idea of your ability to master the language.

If you are studying this language on your own and want a recommended sequence of study, I recommend the following:

Chapters 1, 2, 3, 4, 5	The Core Content
Chapters 7, 8, 9, 11	Selected Advanced Subjects
Chapters 12, 13	Magnetic Tape Programming
Chapter 16 (Tape JCL)	DOS and JCL for Tape Files
Chapters 14, 15	Direct-Access Programming
Chapter 16 (The Remainder)	Other JCL
Chapter 10	Macro Writing
Chapter 6	Binary Arithmetic

Don't feel, however, that you should rigidly adhere to this sequence. Whenever your interest in a subject is aroused, read the appropriate chapter. If, for example, a question arises about tape input and output when you are reading chapter 5, turn to part 4 next. There is no greater assurance that learning will take place than to study a subject in search of an answer.

3 At the end of each topic or chapter there are terminology lists of all the new terms. The intent of these lists is not that you be able to define the words in them, but that you feel you understand the words. After you read a topic, glance at the list and note any word whose meaning is unclear to you. Then, reread the related material. Once the words are fixed in your mind, continue.

4 Following the terminology lists for each topic or chapter there are one or more behavioral objectives. These objectives describe the activities (behavior) that you should be able to perform upon completion of a topic or chapter. The theory is that you will be a more effective learner if you know what you are expected to do and what you will be tested on. This contrasts with the traditional classroom treatment in which a student is forced to guess as to what he will be tested on.

In general, behavioral objectives can be divided into two classes: *knowledge objectives* and *application objectives*. A knowledge objective requires you to list, identify, describe, or explain aspects of a subject. For example, the first objective in chapter 1 is: Identify the model number and I/O device numbers of the System/360-370 that you will be writing programs for. Once you have been told what these numbers are, you should have no trouble fulfilling this objective. Although other knowledge objectives will be more involved and more difficult than this one, given the objective and source of knowledge, you should be able to perform the activity described in the objective.

Application objectives, on the other hand, require you to apply knowledge to problems. Since assembler-language programming is concerned entirely with problem solving, the primary objectives of this book are application objectives. In general, knowledge objectives are only stated when they are a prerequisite for understanding some aspect of assembler language. If only one objective were given for this entire book, it would be something like this: Given a business programming problem of any degree of difficulty, solve it using assembler language.

5 Following the behavioral objectives there are one or more problems for each application objective. These problems are intended to get you involved. There is much truth in the maxim: I hear and I forget; I see and I remember; I *do* and I understand. If there is one message coming from research in education, it is that meaningful learning depends on what the learner does—not on what he sees, hears, or reads.

Because the intent of this book is to teach assembler-language programming, the problems for the most part ask you to apply assembler language to significant programming tasks. There are no fill-in answers, no multiple choice questions, and no true/false statements because those types of activity have nothing to do with writing assembler-language programs. As much as possible, the problems are intended to stimulate

the kind of thinking that would be necessary if you were actually doing the job of a programmer. Because the problems often require you to apply assembler language to situations that go beyond the applications shown in the topics themselves, I hope that at times the problems will help you to experience the joy of discovery and to receive the reward of deeper understanding.

Solutions are presented immediately after the problems. This allows you to confirm that you are right when you are right, but it also lets you learn from being wrong. By checking the solution when you finish a problem, you can discover when you are wrong and correct false notions before they become habits.

Should you actually work each problem in detail before checking the solution? This may be the surest way of learning, but it isn't necessarily the most efficient or the most practical way. As long as you determine what the problem is and conceive the essential elements of a solution before you check the given solution, learning should take place.

One important message: *don't skip the problems.* Reading, like listening, can be a very passive activity. Have you ever, for example, read an entire chapter of a book and then realized you didn't understand any of it? The problems for each topic or chapter will provide you with check points to reinforce the reading you've done, to teach you application, to keep the learning process active, and to help you progress toward deeper understanding.

6 Although this book provides all the information needed to write a wide variety of programs, it is not a complete description of all of the details or instructions of assembler language. As a result, several IBM manuals are named as reference materials in various chapters of the book. If you would like to get a complete set of these reference manuals (though it's certainly not necessary), you can order the following through IBM:

Order No.	Title
GX20-1703	*System/360 Reference Card*
GX20-1850	*System/370 Reference Card*
GC24-5037	*DOS Supervisor and I/O Macros*
GC24-3414	*DOS Assembler Language* (old form number, but still applicable)
GC33-4010	*OS/VS and DOS/VS Assembler Language* (replaces GC24-3414)
GA22-6821	*System/360 Principles of Operation*
GA22-7000	*System/370 Principles of Operation*
GC24-5036	*DOS System Control and System Service Programs*
GC24-3465	*DOS Utility Program Specifications*

At the least, you should order the reference card that applies to the system you will be using.

Part 1

Part 1

Required Background

Before you can start to learn about assembler language, you need some background. This part presents the minimum background needed for a meaningful beginning in assembler language programming. Since this book assumes some previous exposure to data processing and programming —either from a course you have taken or from experience you have had in industry—chapter 1 presents only those introductory concepts that are necessary for assembler-language programming. Chapter 2 presents the System/360–370 machine concepts that are critical to assembler language programming. Although chapter 2 may seem very complex and detailed, it will prepare you for rapid progress when learning assembler-language in part 2.

1

Preliminary Concepts and Terminology

This chapter consists of three topics. Topic 1 presents components of a typical IBM System/360 or System/370 computer system. Topic 2 describes the steps involved in writing programs in System/360–370 assembler language. Topic 3 presents the concept of stacked-job processing and discusses the necessary job-control cards for preparing 360–370 assembler-language programs. Because you may already be familiar with some of this material, you may want to look at the terminology lists, behavioral objectives, and problems given at the end of each topic. They will help you determine which topics you need to read and which you can skip.

TOPIC ONE System/360–370 The IBM System/360 is the most widely used computer system in history. Because its central processing units (CPUs) are available in many different sizes and can be combined with many different input/output

(I/O) devices, the System/360 can be used in relatively small businesses as well as in the largest businesses in the world. For example, the user of a small computer might have a System/360 Model 22, while the user of a large computer might have a 360 Model 195. Between the Model 22 and the Model 195 are Models 25, 30, 40, 50, and so forth, so there is a model size appropriate for most processing requirements.

The IBM System/370 is the successor to the 360. Because it makes use of more recent technological developments, the cost of computer processing can be lower with the 370 than with the 360. Like the 360, the System/370 is available in many model sizes—the Model 115, the 125, the 135, and so forth—and can be combined with many different I/O devices.

Because the 370 was designed to be compatible with the 360, they are conceptually the same from a programmer's point of view, even though electronic components or other hardware elements may differ. In fact, programs written for the System/360 will run on a 370 without any changes as long as the same I/O devices are used. This makes it easy for a computer user to switch from a 360 to a 370.

On the other hand, the System/370 has some capabilities that go beyond those of the 360. If these capabilities are used, a 370 program will not run on a System/360. Whenever one of these capabilities is presented in this book, it is noted so that the 360 user can avoid using it.

Figure 1-1 illustrates a typical, small sized System/370, the Model 135. It consists of a CPU, a card reader and a card punch in the same physical unit (called a reader/punch), a printer, four tape drives, and a disk unit consisting of five disk drives. The manufacturer's model numbers for the I/O devices are printed above the units so you can become familiar with these numbers. In this case, the components used are a Model 135 CPU, a 1052 console typewriter, a 2540 reader/punch, a 1403 printer, four 2400 tape drives, and a 2314 disk unit with five individual drives, or *spindles*, contained within it.

In general, the devices of this typical system have standard capabilities. The 1052 console typewriter can print output data one character at a time, or it can receive input data through the keyboard. Because both sides of the 2540 reader/punch work independently, the reader side can read cards while, at the same time, the punch side punches cards. The 1403 printer is a high-speed printer that prints an entire line at one time. The Model 2400 tape drives are devices that can write data on or read data from a magnetic tape. The 2314 disk unit can write data on or read data from disk packs mounted on the spindles within the unit. Because in-depth understanding of tape and disk devices is required of the assembler-language programmer, the 2400 and 2314 are covered in detail in chapters 12 and 14.

Most System/360s have the same I/O devices as the system just described. However, a System/370 more commonly has a 3330 disk unit instead of a 2314. It is a faster disk unit with greater storage capacity and it too is described in detail in chapter 14. In addition, the System/370 might have a 3211 printer instead of the 1403, a 3505 card reader and 3525 card punch instead of the 2540 reader/punch, and Model 3400 tape drives instead of the 2400s. It might also have a video-display console device in place of the 1052 console typewriter, so that console output is displayed on a television-like screen rather than being typed out. In this case, a keyboard beneath the display screen is used to enter console input data. In all cases, however, the primary difference is input or output speed rather than any important conceptual difference.

This book is designed to teach you how to write programs for the System/360 or 370. Because these systems are conceptually the same, they are generally spoken of as if they were the same, even though there are some technological differences. Throughout this book, you will find them spoken of as one system that is referred to as the System/360-370.

Terminology

spindle

Objective

Identify the model number and I/O device numbers of the System/360-370 for which you will be writing programs.

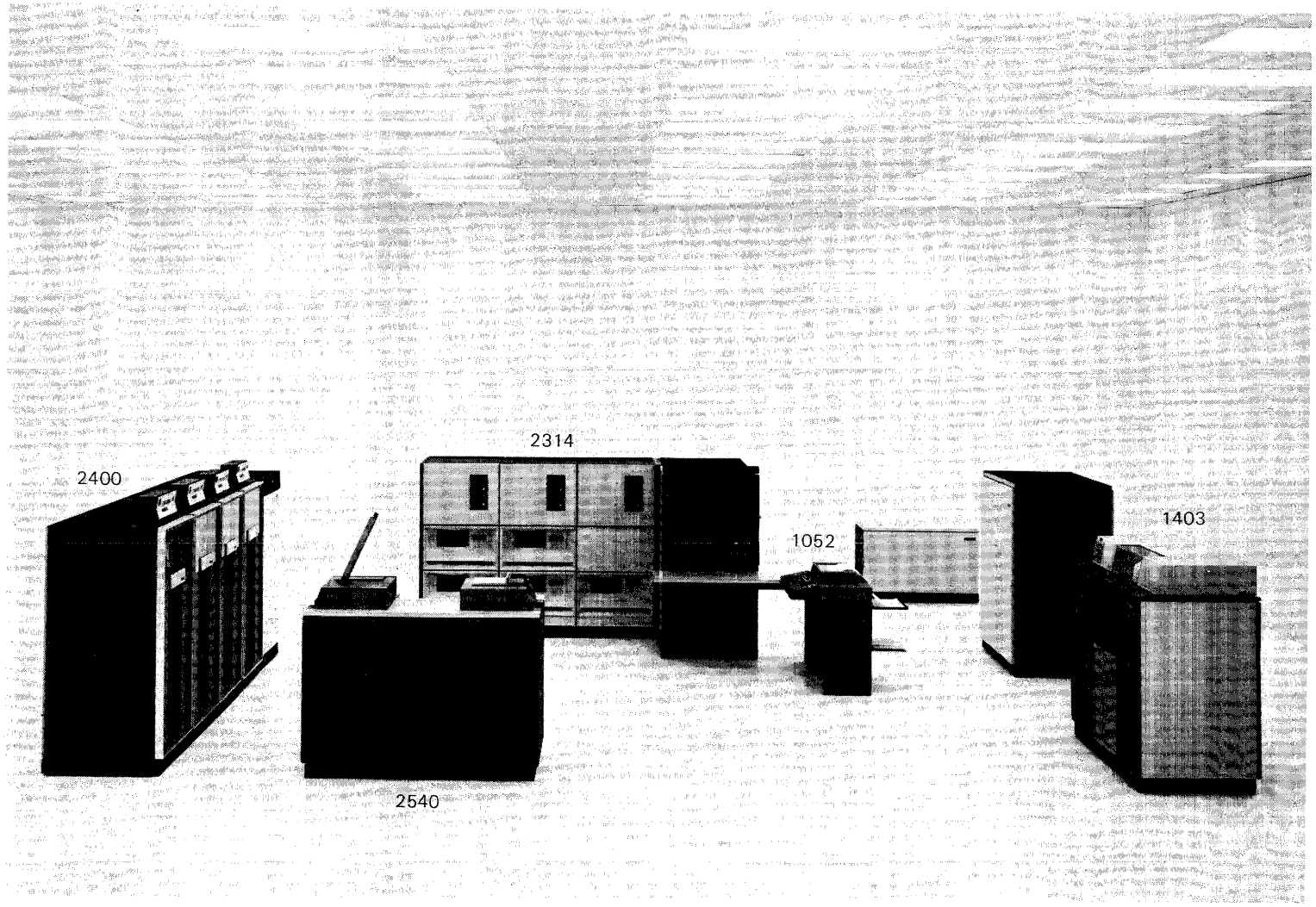


FIGURE 1-1 A Model 135 System/370

TOPIC TWO Writing a Program in System/360–370 Assembler Language When writing a program in System/360-370 assembler language, a programmer goes through five phases: (1) he defines the problem to be programmed; (2) he plans a solution to the problem; (3) he codes the solution in assembler language; (4) he tests his program to be sure that it does what is intended; and (5) he documents the program. This topic is devoted to an explanation of these five steps.

DEFINING THE PROBLEM

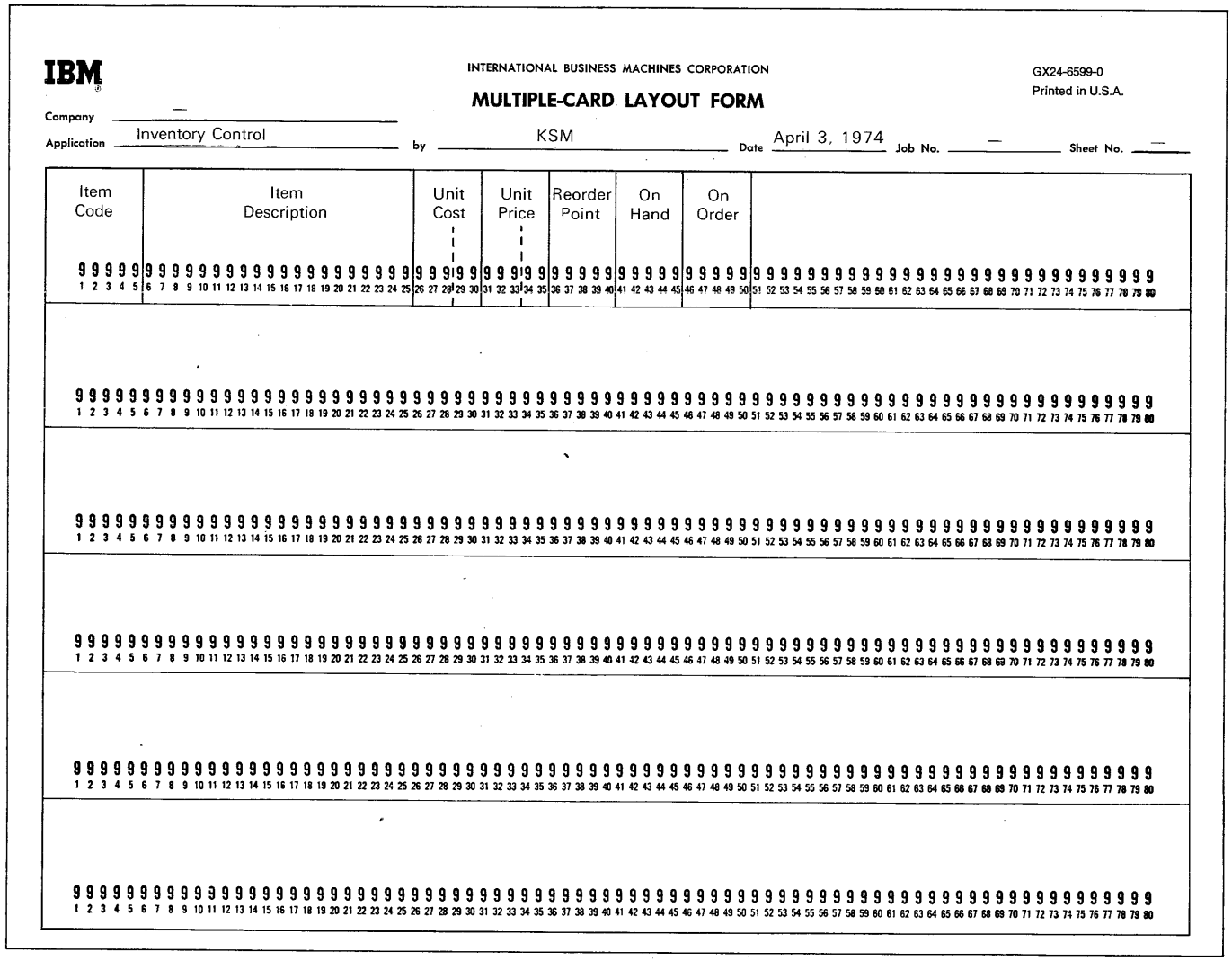
Defining the problem is simply making sure that you know what the program you are going to write is supposed to do. You must understand what the input is going to be, what the output of the program must be, and what calculations or other procedures must be followed to derive the required output from the input. Two documents that are often used for defining card input and printer output are the *card-layout form* and the *print chart*.

Card-layout forms show which fields the input cards are going to have and which card columns each field is in. These forms have many different formats. Some give the layout of several different types of cards; some give the layout for only one card. The form illustrated in figure 1-2 is a *multiple-card-layout form* that can give the layouts of up to six different cards. In this case, only one card layout is given, that of an inventory-balance card. By studying this form, you should be able to see, for example, that the item-number field is in columns 1–5, the unit cost, with two decimal positions, is in columns 26–30, and so on.

A print chart, such as the one in figure 1-3, shows the layout of a printed report or other document. It indicates both the print positions to be used for each item of data on the report and the headings to be printed. On the print chart in figure 1-3, the heading INVESTMENT REPORT is to be printed in print positions 15–31, the column heading ITEM CODE is to be printed on lines 4 and 5 in print positions 7–10, and so on. Similarly, the item-number field for each data line (as opposed to a heading line) is to be printed in print positions 6–10 and the amount invested is to be printed in positions 41–49. At the end of the report, two lines are to be skipped and then a total of the amount invested is to be printed; this total is indicated by two asterisks printed in positions 51 and 53. Although figure 1-3 indicates only 60 print positions from left to right, a complete print chart usually has 144 or more—at least as many as there are available on the printer being used.

A print chart also shows which punches in the *carriage-control* (or *forms-control*) *tape* of the printer are to be used by the program. A carriage-control tape is a paper tape that is punched to correspond to the lines of a printed form. Figure 1-4, for example, shows the carriage-control tape used to print an invoice form. A 1-punch in the carriage-control tape corresponds to the first address line of the invoice, a 2-punch corresponds to the miscellaneous line, a 3-punch to the first line in the body of the form, and a 5-punch to the total line. As you can see, in the carriage-control tape there are 12 punching positions called *channels*, all of which can be used if necessary. Although the channel-1 punch is commonly used to represent the first printing line of a form, the use of channels 2–11 is determined by the system designer or programmer.

After the carriage-control tape is planned, the required holes are punched in the tape with a hand punch. Usually, the series of punches for each form is repeated two or more times so that one carriage-control tape is the length of two or more forms (since one form length is usually too short to put in the carriage-control mechanism properly). Then, the tape is cut to the length of the appropriate number of forms and the ends are pasted together to form a loop. When this loop is placed in the carriage-control mechanism of the printer, the control tape moves in conjunction with the continuous form being printed. As a result, if the printer in the example just described is instructed to skip to a 1-punch in the control tape, the continuous form is skipped to the first address line of the invoice. If the printer skips to channel 2, the continuous form is skipped to the miscellaneous line. Although it is difficult to visualize the operation of a carriage-control tape as you read about it, it is quite easy to understand when you actually see it work.



IBM

INTERNATIONAL BUSINESS MACHINES CORPORATION

GX24-6599-0
Printed in U.S.A.

MULTIPLE-CARD LAYOUT FORM

Company —

Application Inventory Control by KSM Date April 3, 1974 Job No. — Sheet No. —

Item Code	Item Description	Unit Cost	Unit Price	Reorder Point	On Hand	On Order	
1 2 3 4 5	6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25	26 27 28 29 30	31 32 33 34 35	36 37 38 39 40	41 42 43 44 45	46 47 48 49 50	51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80

FIGURE 1-2 Multiple-Card Layout Form

Channel 12 has a special purpose. It is used to indicate where the last printing line (other than the total line) should appear on the form. As you can see in figure 1-4, the 12-punch corresponds to the line before the total line. The printer can be programmed to skip to any one of the other 11 punches whenever a 12-punch is sensed, thus skipping to

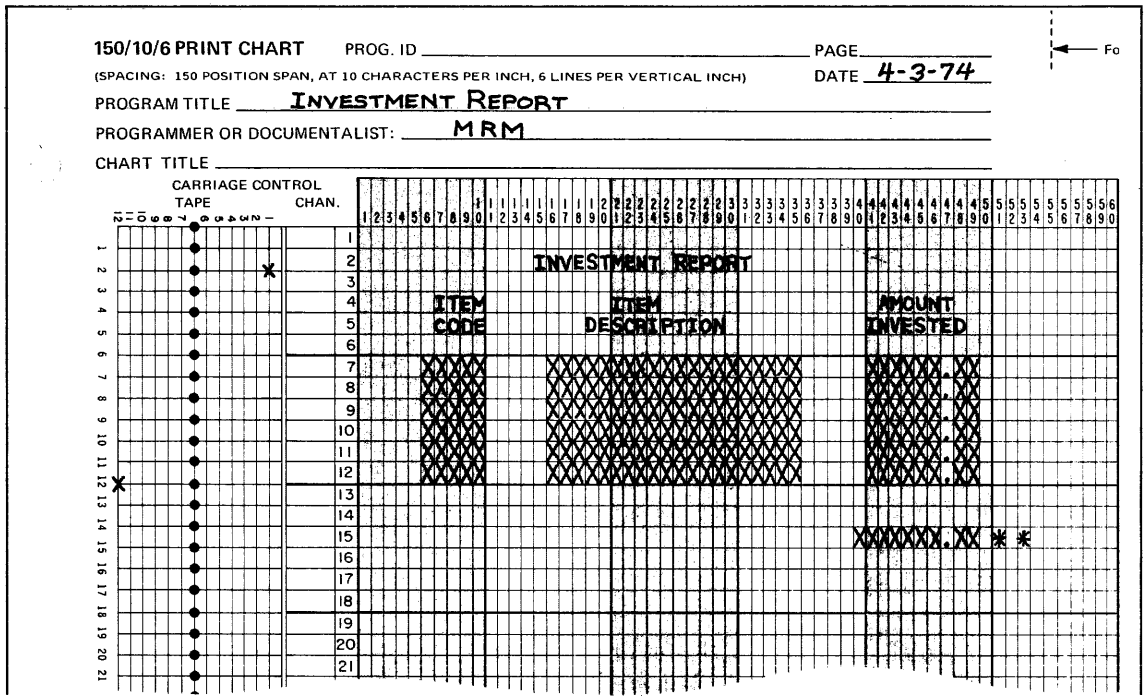


FIGURE 1-3 Print Chart

the next form. This skipping from one form to another is commonly referred to as *forms overflow*.

On the left portion of the print chart the carriage-control punches to be used by the program are given. In figure 1-3, channel 1 is to be used for the first heading line, line 2, of each form. In addition, channel 12 is to be used for the last data line to be printed before forms overflow takes place. When printing standard blank forms, as opposed to printed forms such as invoices or payroll checks, you can assume that a carriage-control tape with at least a 1-punch for the first printing line and a 12-punch for the last printing line on each page will be present in the printer at the time of program execution. This is standard operating procedure.

In addition to the card-layout form, a programmer may be given other written program specifications before writing a program. For example, he may be given formulas to be used for calculating output results, tape or disk record-layout forms, and a narrative summary of the processing that is to take place.

Because certain aspects of the problem are likely to be inadequately defined, the programmer usually questions the person who assigned the program to him until he feels the entire problem has been clearly defined. Suppose, for instance, you are asked to write a program that prepares a report like the one in figure 1-3 using a deck of input cards that have the format shown in figure 1-2. What additional

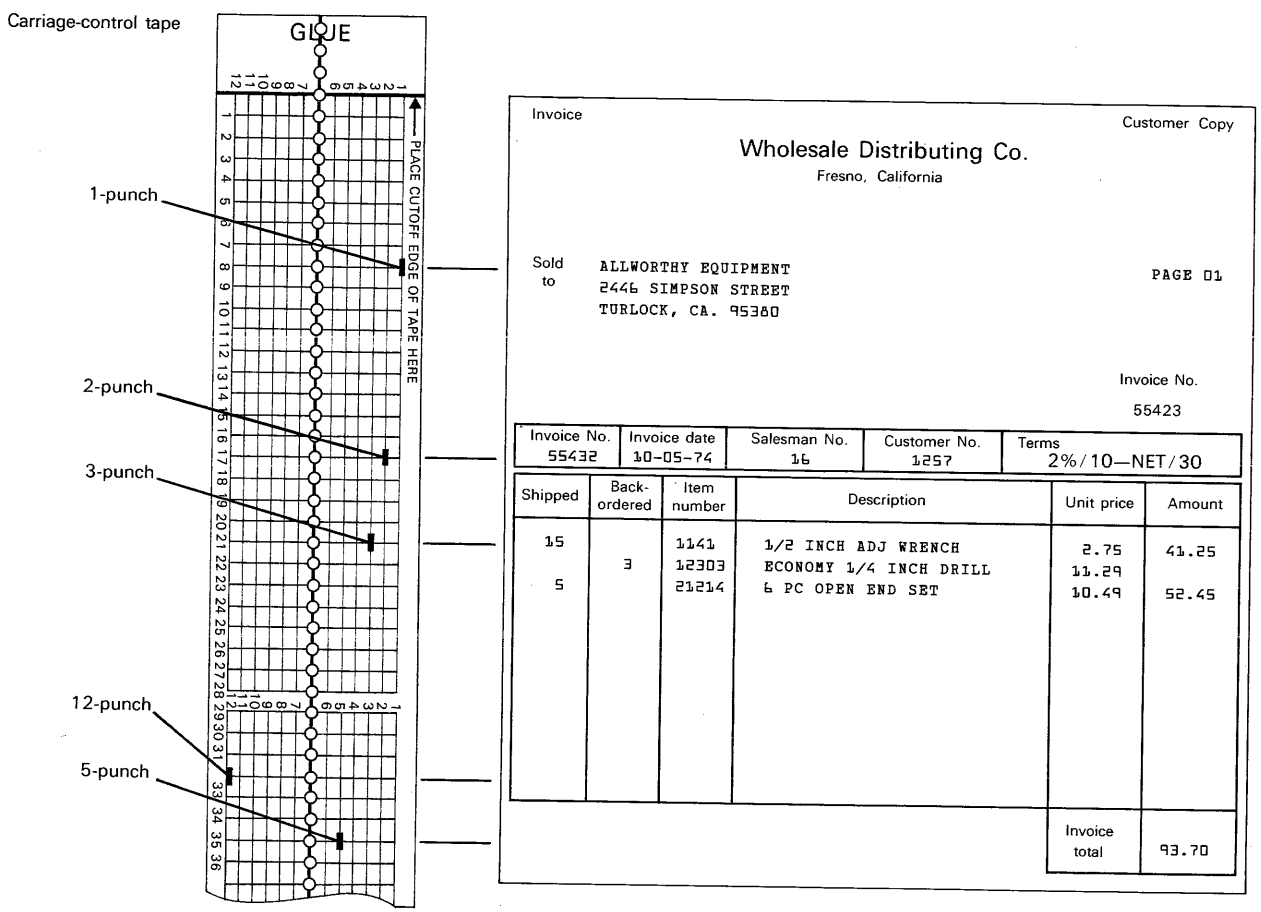


FIGURE 1-4 Carriage Control

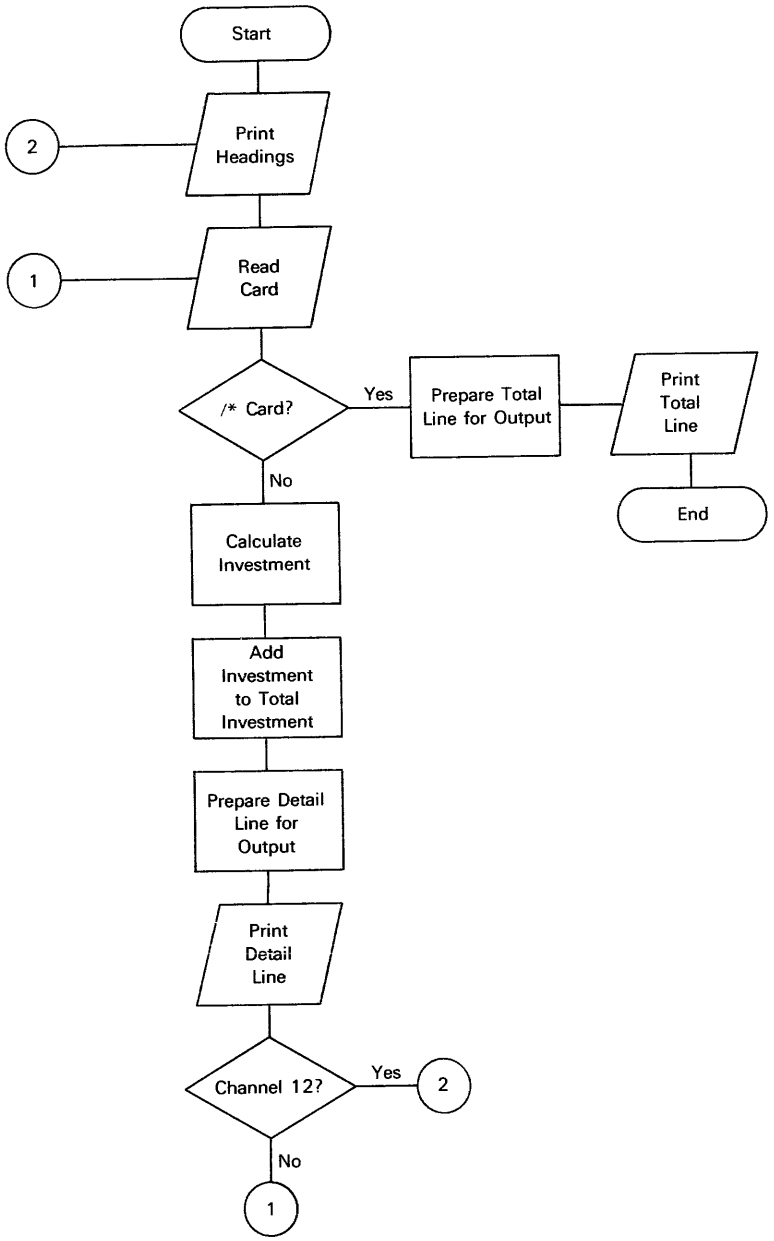


FIGURE 1-5 Program Flowchart

information do you think you would need to get from the person who assigned the problem to you? Here are some suggestions.

1 How is the amount invested calculated? Is it the on-hand balance multiplied by unit cost, or is it on-hand multiplied by unit price?
2 Should the input deck be checked to be sure it is in numerical sequence by item number? (This is a common programming practice.)
3 Should one line be printed for each card in the input deck or would it be better to print a line for only certain items—perhaps those items with an inventory investment of over $10,000?

The point is that you must know exactly what the program is intended to do before you can write it. Often, programming errors stem from an incomplete understanding of what the program is supposed to do.

PLANNING THE SOLUTION

Planning the solution to a programming problem means deciding what must be done to solve that problem. At this point the programmer is generally concerned with the sequences of instructions to be used and the logic required to derive the output from the input. When planning an assembler-language program, the programmer draws a *program flowchart* of the planned solution—something like the flowchart in figure 1-5. The flowchart is then used as an aid in coding the program.

Basically, there are two levels at which a program flowchart can be drawn: general and detailed. A *general flowchart* (sometimes called a *macro flowchart*) gives the major functional and logical requirements of the program. It is often prepared by a system designer, as opposed to a programmer, and is too general to be used as a guideline for coding.

On the other hand, the *detailed flowchart* (sometimes called a *micro flowchart*) is intended to be a guide for

coding. As a result, it directly corresponds to the logic of the eventual program. In this book, only detailed flowcharts are used.

Figure 1-5 is a detailed flowchart for an inventory program that prepares a report like the one specified by the print chart in figure 1-3 using a deck of balance-forward cards that have the format given in figure 1-2. The last card in the input deck, called a /* card, has a slash in column 1 and an asterisk in column 2. This is the way the end of every System/360-370 input card deck is identified. When the /* card is read, the program is supposed to print the total amount invested for all input cards.

The flowchart uses the five different symbols shown in figure 1-6. These symbols conform to the flowcharting standards approved by the American National Standards Institute and will be used throughout this book. Although many companies have not yet converted to these standards, almost all companies use symbols that vary in only a minor way from the standard symbols.

Can you follow the flowchart in figure 1-5? As a general rule, you start at the top and read down and to the right, unless arrows indicate otherwise. When you come to a circle with a number in it (called a connector circle), you continue at the connector circle containing the same number. For example, after printing a detail line (a line representing one input record), the program reaches the connector circle with a 2 in it if forms overflow is to take place (that is, if a channel-12 punch has been sensed by the printer). This means that the flow of the program continues at the connector circle leading into the Print Headings symbol (or block). If the 12-punch hasn't been sensed, the program reaches the connector circle containing a 1, so the program continues with the block containing Read Card.

When drawing flowcharts, the main concern is that all processing and all branches required by the program are indicated on the flowchart. If the words used in the symbols clearly indicate the operations that are to take place, they serve their purpose. When drawing a flowchart for a specific language, though, the programmer normally uses words or code that correspond to the language to be used. Later on in this book, then, you will see flowcharts using words that correspond to System/360-370 assembler language.

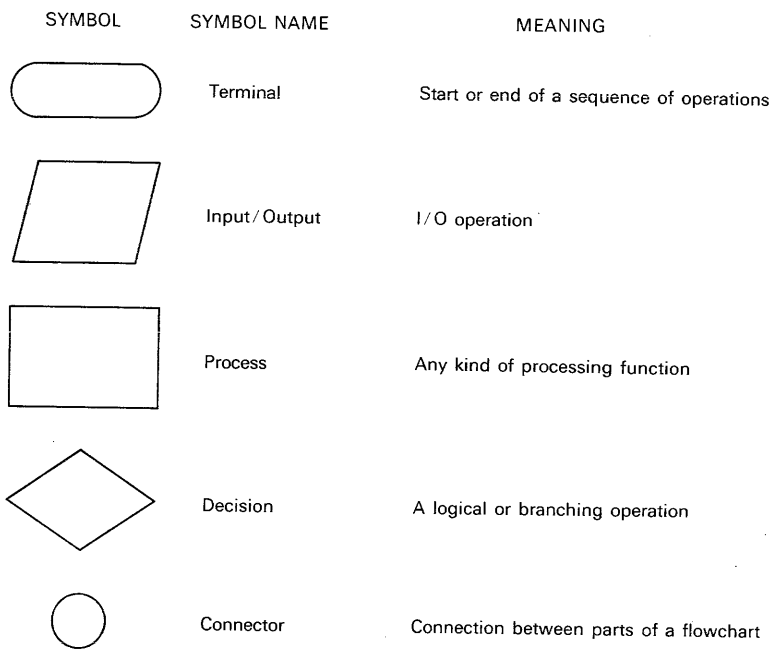


FIGURE 1-6 Program Flowcharting Symbols

As an aid in drawing the symbols, a programmer uses a plastic flowcharting template such as the one shown in figure 1-7. To use the template, the programmer uses his pencil to trace around the outside edge of the cut-out symbol. This method allows him to draw a more exact symbol than he could if he drew it freehand. Because using the template is somewhat slower than drawing freehand, a programmer is likely to make one or more freehand sketches of a flowchart before using the template to draw the final version.

Incidentally, in both industry and programming classes, a programmer often omits drawing the flowchart in his

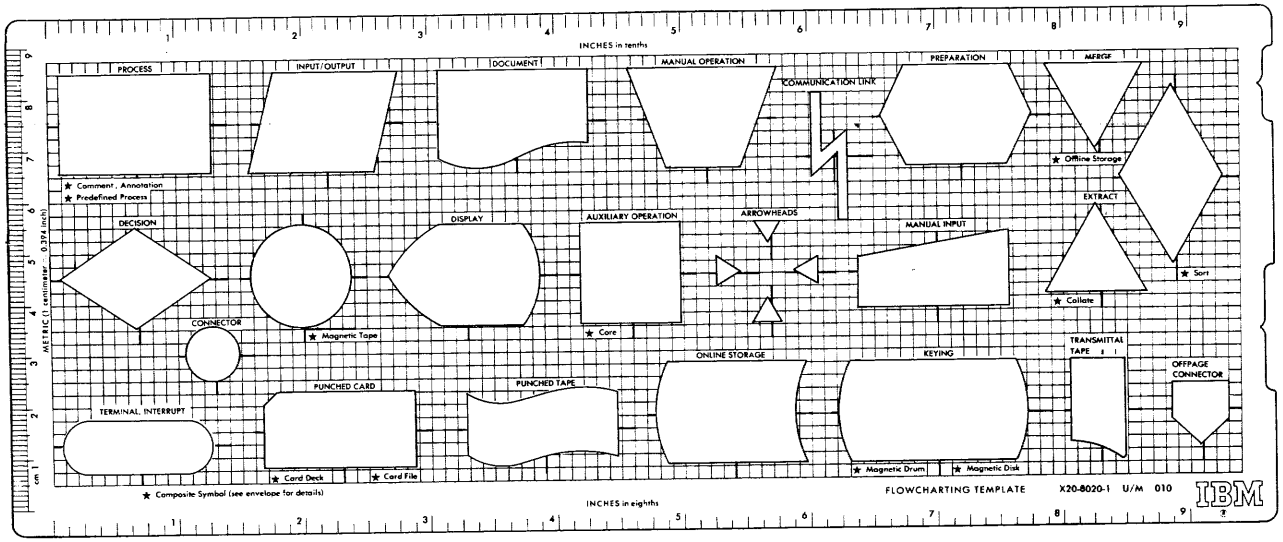


FIGURE 1-7 Flowcharting Template

eagerness to write a program. Quite frankly, most people enjoy coding and testing programs more than drawing flowcharts. Nevertheless, the most efficient way of preparing a program, except for very simple ones, is to draw a complete flowchart first. By using the flowchart as a guide, coding and testing can be done more accurately and in less time than they could otherwise.

CODING

Coding a program in System/360-370 assembler language involves writing the detailed code that eventually gets translated into a machine-language program—that is, a program that can be run by the computer. The major purpose of this book is to teach you how to code in Basic Assembler Language (*BAL*) for the System/360-370. When coding BAL, special coding sheets are used and, when the program is finished, one card is keypunched for each coding line. The resulting deck of cards is called the *source deck*.

Because the source deck may contain keypunching or programming errors, the programmer *desk checks* the source deck after it is keypunched. He usually does this by studying a listing of the contents of the source cards, but he can also do it by studying the printing done by the keypunch at the top of the source cards. Figure 1-8 illustrates these two alternatives. If the programmer finds any errors, he makes the appropriate corrections to the source deck. When he is sure that the source deck is as accurate as he can make it, it is ready to be *assembled*.

Assembling a BAL source deck means converting the source code of the source deck into an *object program*. The object program may be punched into cards, called an *object deck*, or it may be stored on magnetic tape or disk. In any case, the object program, which is in machine language, can be *loaded* into the storage of the computer and then *executed*. In other words, the object program is ready to run.

The translation of a BAL source deck into an object program is called an *assembly*. This is done by the computer

itself under control of a translator program called an *assembler*. A typical assembly is illustrated in figure 1-9. This assembly takes place in two steps.

1 The assembler, which is stored on magnetic disk as an object program, is loaded into the computer.

2 The computer executes the assembler, thus processing the source deck and creating the object program, which is written on the disk ready to be loaded and executed. During the assembly, an *assembly listing* is printed by the computer.

Figure 1-9 points out that, in most cases, an object program is stored on a disk rather than being punched into cards.

The assembly listing is a listing of the source deck as well as a listing of various reference tables to which you will be introduced in chapter 4. If any errors are caught by the assembler during the assembly (as is usually the case), one or more *diagnostic messages* (or *diagnostics*) are printed as part of the assembly listing and the object program is not created. Each diagnostic calls attention to one source deck error.

If there are diagnostics, the programmer makes the necessary corrections to the source deck and the deck is reassembled. The process is repeated until there are no more diagnostics in the assembly listing. At this point, the program is ready to be tested.

TESTING

To test a program, the programmer tries his object program on *test data*. This test data is intended to try all of the conditions that may occur when the program is actually used. For card-to-printer programs, the test data is punched into cards, but for more complex programs, the test data may include card decks, tapes, disks, and any other form of input used. When the program has been executed, the programmer compares the actual output with the output he expected to get. If they agree, he can assume that the program does what it is intended to do.

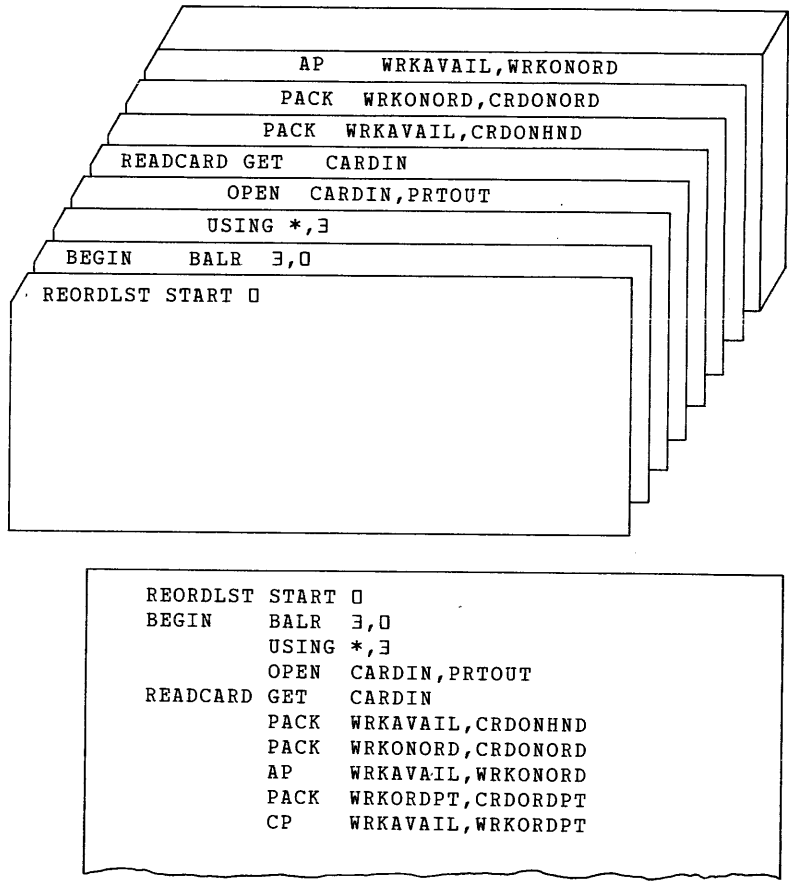
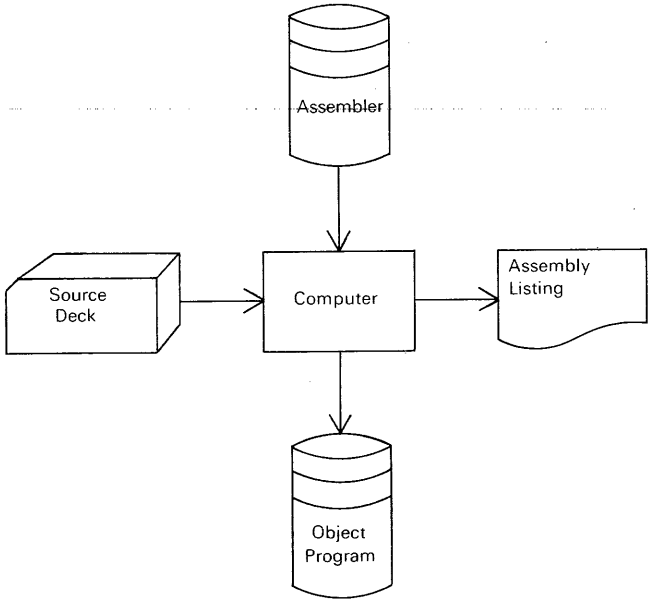


FIGURE 1-8 Source Cards and Source Listing

More likely, however, the actual output and the intended output will not agree the first time the program is executed. The programmer must then *debug* the program. He must find the errors (bugs), make the necessary corrections to the source deck, reassemble the source program, and make another *test run*. This process is continued until the program executes as intended.

Although debugging techniques vary depending on the language used, testing is often the most difficult phase of



1. The assembler is loaded from disk into storage.
2. The assembler is executed, thus processing the source deck, printing the assembly listing, and writing the object program on disk.

FIGURE 1-9 Assembly

programming. In a program consisting of thousands of instructions, the original deck may have dozens of bugs, require dozens of reassemblies, and take weeks to debug.

In actual practice, rather than making just one test run on a new program, a series of test runs is made using different sets of test data. The test data for the first test run is usually low in volume—perhaps only a dozen input records—and may be designed to test only the main processing functions of the program. After the program has been debugged using this data, it may be tested on data that tries all conditions that may possibly come up during the execution of the program. This test data is usually of much greater volume than the first test data. After the program checks out with this data, a test run may be made using actual, or "live," test data. Then, an entire group of programs may be tested together to make sure that the output from one program is valid input for the next program. Only after a program has proved itself under conditions that are as close to real as possible is the program considered ready for use.

DOCUMENTING THE PROGRAM

Documentation in data processing terminology refers to the collection of records that specifies what is being done and what is going to be done within a data processing system. For each program in an installation, there is a collection of records referred to as *program documentation*. One of the jobs of a programmer is to provide this documentation.

Why is programming documentation necessary? Because data processing requirements change. For example, tax laws change: the social security tax percent has increased periodically over the last several years as has the maximum amount of social security tax that must be paid in any one year. Company policies also change: discounts may vary from year to year, production departments may switch to new forecasting techniques, and accounting practices may change. For each change, all affected programs must be modified.

In fact, change in data processing requirements is so common that large companies employ special maintenance programmers whose entire job is to modify existing programs. This frees the other programmers to work on new programs without interruption. Without adequate documentation, however, maintenance programmers could not make changes within a reasonable period of time. Even when a programmer modifies his own programs, though, documentation is invaluable. Three months after writing a program, you can barely recognize it.

The following are some of the more important documents likely to be required by a company's documentation standards.

1 Specifications that give the detailed requirements of the program
2 Layouts of all input and output records on special layout forms

3 A flowchart of the entire program
4 The assembly listing created during the last assembly
5 Listings of both the input data used for testing and the output results of the test runs

Most of these documents are prepared and used as the program is developed. The flowchart, for instance, is both an aid for coding the program and a record of the completed program. Nevertheless, a programmer normally spends considerable time refining and finishing documentation after he has completed a program. To avoid wasting their programmer's time, most companies try to reach a balance between too much and too little documentation. With few exceptions, however, they have too little rather than too much.

CONCLUSION

Figure 1-10 represents the proportion of time an assembler-language programmer might spend on each of the five phases of programming. The most important point to note is that coding is less than half of the programmer's job.

Terminology

card-layout form
print chart
multiple-card-layout form
carriage-control tape
forms-control tape
channel
forms overflow
program flowchart
general flowchart
macro flowchart
detailed flowchart
micro flowchart
BAL
source deck
desk check
assemble
object program
object deck
to load a program
to execute a program
assembly
assembler
assembly listing
diagnostic message
a diagnostic
test data
debug
test run
documentation
program documentation

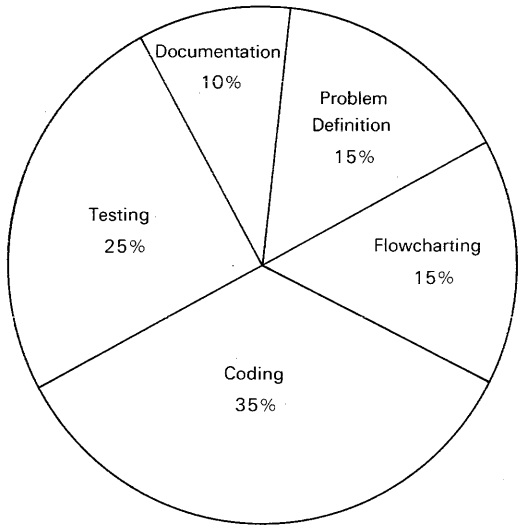


FIGURE 1-10 How a Programmer's Time Is Spent

Objectives

1 Given the card-layout form and the print chart for a programming problem, identify the card columns or print positions of any input or output data item. Also, identify any carriage-control punches to be used by the program and state which portions of printed output these punches correspond to.
2 Given a description of a programming problem, draw a program flowchart for its solution.
3 In general terms, describe how a BAL source deck is assembled into an object program.
4 List the five phases a programmer goes through when writing a program.

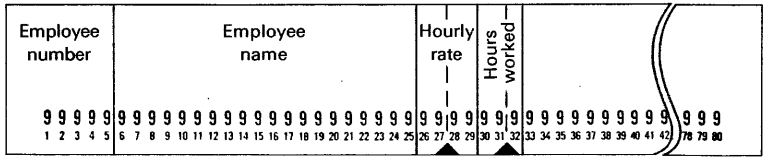
Problems

NOTE: If you have never before been introduced to card-layout forms, print charts, or flowcharting, you should do the problems that follow. Otherwise, you will probably

want to skip them. As you work your way through this book, you will be presented with progressively more complex examples and problems involving the analysis of layout forms and print charts, and the drawing of flowcharts.

1 (Objective 1)

a. In the card shown below, which columns contain the hourly-rate field?



b. How many decimal positions does the hours-worked field have?

c. What data is punched in columns 6-25?

d. Based on the print chart that follows, in what print positions should the first heading line, OVERTIME REPORT, be printed?

e. In which print positions should the final total of overtime hours be printed?

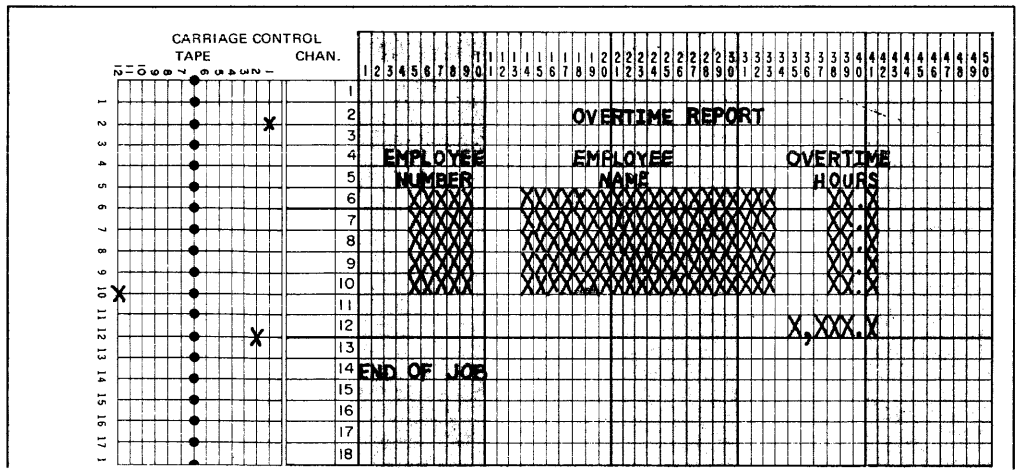
f. In which print positions should the message, END OF JOB, be printed?

g. What character should print in print position 40 of each data line?

h. Which carriage-control punches are used by the program, and what printing lines do they correspond to?

i. Should the headings be printed on all pages or only on the first page of the report?

2 (Objective 2) The input and output of a program is described by the card-layout form and print chart shown in problem 1. To determine overtime, forty is subtracted from the number of hours worked. If an employee has worked overtime, one line should be printed showing the number of overtime hours worked; otherwise, no line should be printed for the employee. Draw a program flowchart for this problem.



Solutions

1 a. 26–29
b. 1
c. Employee name
d. 18–32
e. 35–41
f. 1–10
g. A decimal point
h. These carriage-control punches are used: channel 1 corresponds to the first heading line; channel 2 corresponds to the total line; channel 12 corresponds to the overflow line—that is, the last data line for each page. Remember that channel 12 always indicates the overflow line.
i. You can't tell from a print chart, but it is common practice to print the headings on all pages of a report.

2 Figure 1-11 is a flowchart that gives the basic requirements of a solution. However, solutions may vary considerably from person to person.

You may have noticed that the printing blocks for detail lines in the flowchart are preceded by blocks showing the data being arranged in the output area. Although this may puzzle you now, you should understand it after you have been introduced to BAL coding in chapter 3. In fact, program flowcharting itself will become more meaningful after you have become familiar with the language you are going to use for coding. If you have difficulty with flowcharting at this stage of your development, don't let it bother you.

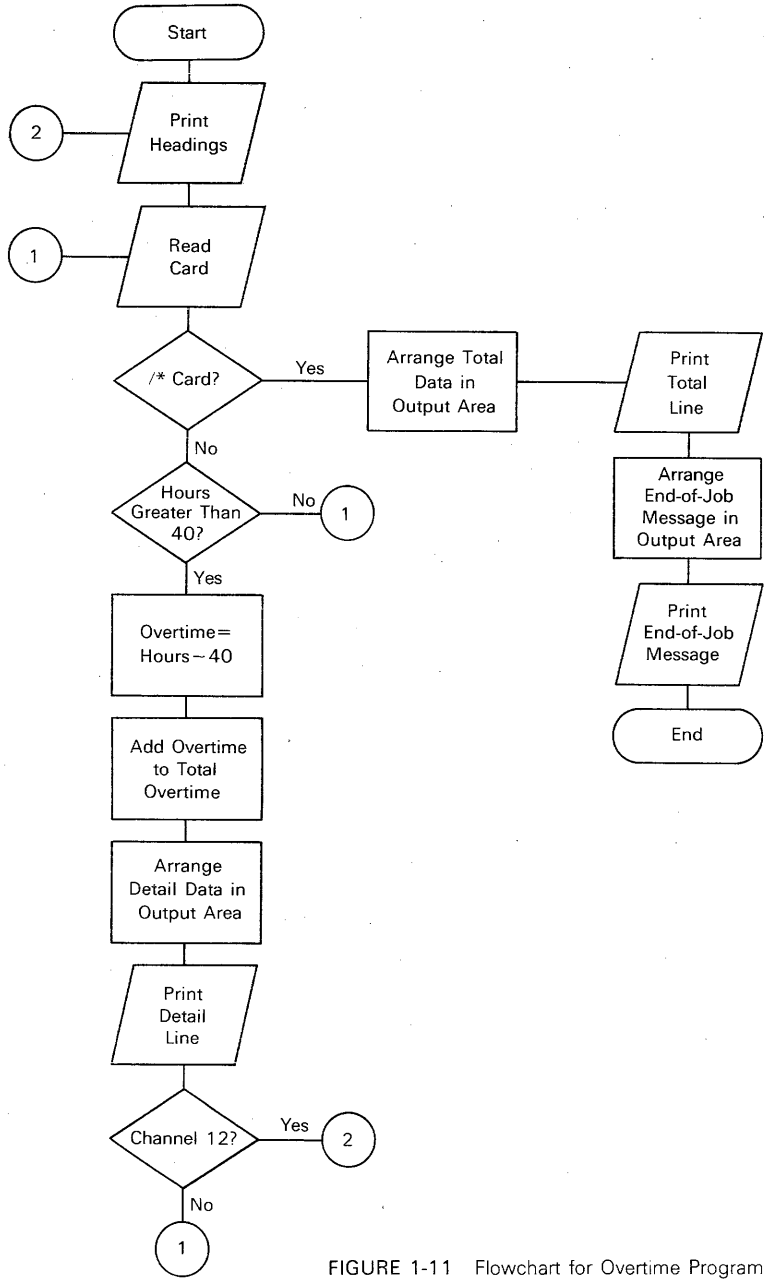


FIGURE 1-11 Flowchart for Overtime Program

TOPIC THREE Introduction to DOS and Job-Control Language

When a company uses a System/360 or System/370, an *operating system* is supplied with the equipment. An operating system is a collection of programs designed to improve the efficiency of a computer installation. It does this in two ways. First, an operating system improves operational efficiency by providing programs that make the transition

from one program to another more efficient. This job-to-job transition is often referred to as *stacked-job processing*. Second, an operating system increases programming efficiency by providing various language translators, such as an assembler, along with other programs that eliminate or reduce the programming efforts required of a computer user. Although this chapter deals primarily with stacked-job processing and related operational procedures, chapter 16 describes some of the other programs and capabilities of an operating system.

With the System/360-370, several different operating systems are available. The two most widely used are the *Disk Operating System (DOS)* and the full Operating System (OS). These systems vary in the features they provide and the amount of storage they require. For example, DOS requires a computer with at least 16,000 storage positions and provides translators for assembler language, COBOL, FORTRAN, PL/I, and RPG. OS, the most advanced of the IBM operating systems, requires a computer with at least 64,000 storage positions and supplies translators for an advanced version of FORTRAN in addition to translators for assembler language, PL/I, RPG, COBOL, and ALGOL. DOS is in general use on System/360 models 22, 25, 30, 40, and occasionally 50 as well as on System/370 models 115, 125, 135, and occasionally 145. OS, on the other hand, is in general use on System/360 models 50 and up, and on System/370 models 145 and up.

Although the Basic Assembler Languages supplied with DOS and OS are the same in many respects, there are enough differences between them to make treatment of both DOS and OS BAL in the same book confusing. Therefore, this book deals exclusively with Basic Assembler Language for the Disk Operating System, or *DOS BAL*.

STACKED-JOB PROCESSING

Before stacked-job processing was developed, a computer system stopped when it finished executing a program. The operator then removed the output from the program (such as card decks or magnetic tapes) and made ready the I/O units for the next program. Next, he loaded the program—usually in the form of an object deck—and placed any cards to be processed in the card reader. The program was then ready to be executed.

The trouble with this intervention of the operator between programs was that it wasted computer time. If a company ran 120 programs a day and the operator took 30 seconds to set up each program, it meant one hour of lost computer time. With computer time on a medium sized system costing $100 or more per hour, such lost time would be very costly.

When stacked-job processing is used, the computer rather than the computer operator loads programs. To make this possible, all of a company's programs are stored in *libraries* on a *system-residence device*, which for the Disk Operating System is a disk device. This allows programs to be directly accessed from the system-residence device and loaded into storage at a high rate of speed.

At the start of a day's computer operations, then, the computer operator loads a *supervisor program* (or *supervisor*) into storage, and control of the computer is transferred to this program. The supervisor, which is one of the programs of the operating system, is responsible for loading all of the other programs that are to be executed from the system-residence device. The supervisor program, which may require 6000 storage positions or more, remains in storage during the execution of all other programs. As you will see later, the supervisor has other responsibilities in addition to handling job-to-job transition, but, for now, concentrate on its use in stacked-job processing.

To tell the supervisor which programs are supposed to be executed, the operator places a stack of *job-control cards* (such as the stack in figure 1-12) in the card reader. These cards give the names of the programs to be executed along with such information as which tape should be mounted on

which tape drive. There are usually several job-control cards for each program to be executed, and, if a program requires card input, the data deck follows the job-control cards. In figure 1-12, programs 2 and 3 require card input, while programs 1, 4, and 5 do not. The stack of job-control cards is commonly referred to as a *job deck*.

When the computer has finished executing one program, loading and executing the next actually takes place in four steps. These steps are illustrated in figure 1-13. First, control of the computer passes from the completed program to one of the instructions of the supervisor program. Second, the supervisor loads a program called the *job-control program* from the system-residence device and passes control to it. This program, which is one of the programs of the operating system, is responsible for reading and processing the job-control cards for the next program, Third, the job-control program processes the job-control cards. If there are any errors in the job-control cards or if any necessary information has been omitted, the program prints a message on the console typewriter. When all job cards have been processed, control passes back to the supervisor. Finally, the supervisor loads the next program from the system-residence device and passes control to its first instruction.

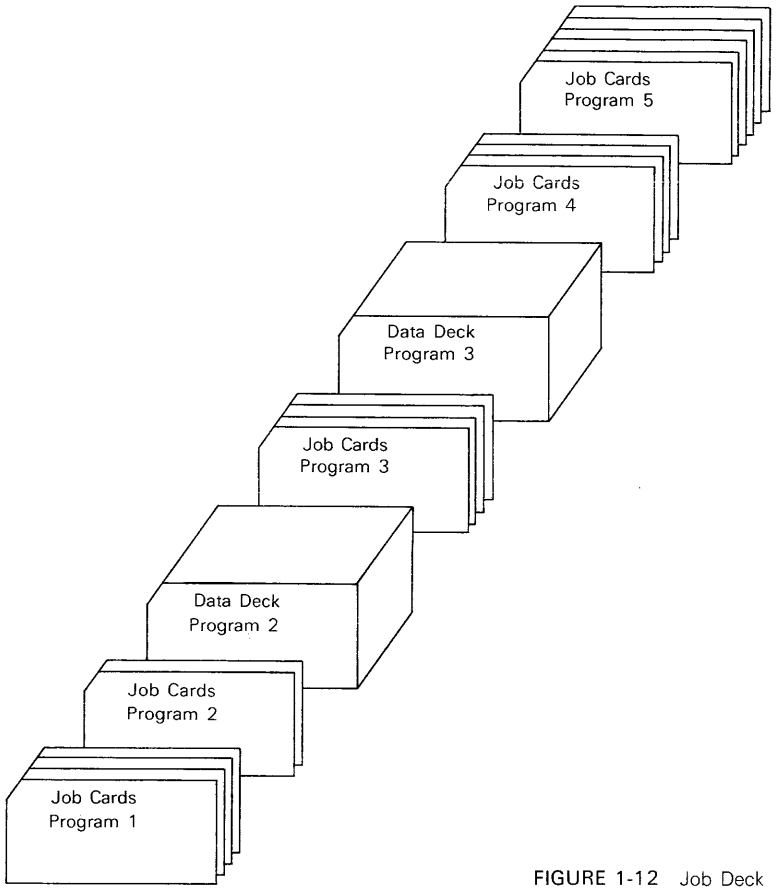


FIGURE 1-12 Job Deck

JOB-CONTROL LANGUAGE

In order to assemble and test a program on a DOS system, the programmer must be able to create the required job-control cards. Because job-control cards require specifications that are in a sense a language of their own, the code used in job-control cards is often referred to as *job-control language*. In the remainder of this topic, enough elementary DOS job-control language is presented so that by the end of this chapter you will be able to assemble and test card-to-printer programs. In chapter 16, additional specifications of DOS job-control language are presented so that you will be able to test tape and disk programs as well as use some of the other features of the Disk Operating System.

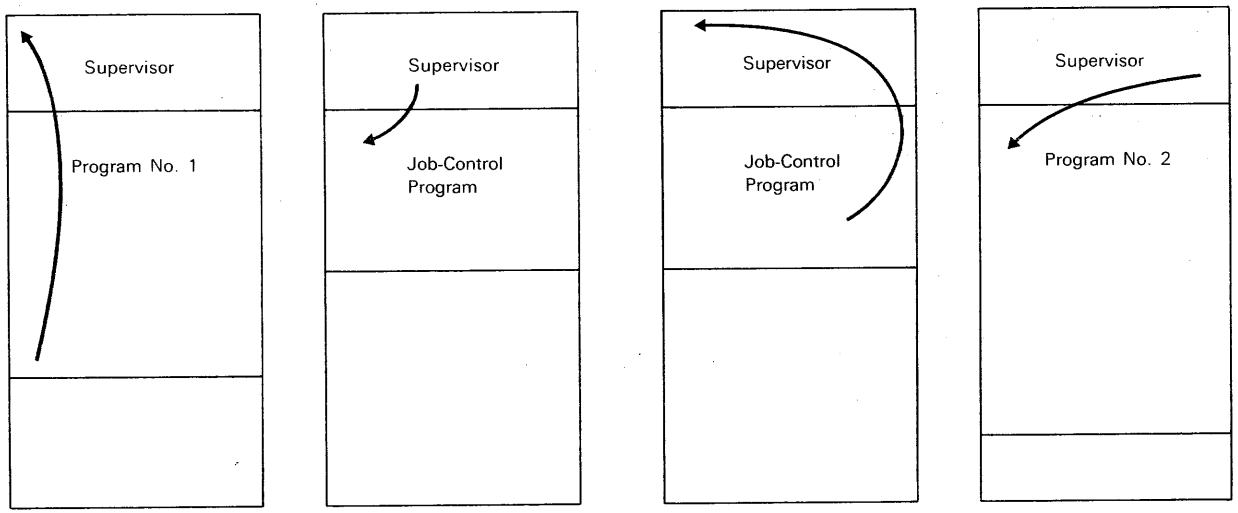


FIGURE 1-13 Job-to-Job Transition in Stacked-Job Processing

Assembling a Program

Figure 1-14 illustrates the job-control cards needed to assemble a BAL source deck. It begins with a JOB card that gives a *jobname* to the *job* to be performed. The JOB card's format, which is typical of many job-control cards, begins with two slashes (columns 1 and 2) followed by one or more blanks and the operation—in this case, JOB. This is followed by one or more blanks and the operand or operands—in this case, the jobname MCQUILLN, which is a shortening of the programmer's name, MCQUILLEN. After one or more blanks, the operand or operands of a job-control card can be followed by comments (any data) that are ignored by the operating system.

The jobname can be up to eight characters long. These characters can be alphabetic or alphabetic and numeric, but the jobname must start with a letter. Although the jobname is used in certain messages printed by the operating system, it has little significance for the student programmer. As a result, your name or a shortening of it (eight characters maximum with no blanks or special characters) is an acceptable jobname for job decks you create.

The second card, // EXEC ASSEMBLY (called the EXEC card), causes the assembler to be loaded into storage and executed. ASSEMBLY is the name given to the assembler stored in the program library of the system-residence device. When the assembler is executed, the source deck is read and the assembly listing along with diagnostics is printed.

The /* card (slash in column 1, asterisk in column 2) indicates to the assembler that there are no more source cards. The /& card (slash in column 1, ampersand in column 2) indicates that the job is completed. In DOS, a job may consist of one or more *job steps*. For each job step, there is one EXEC card between the JOB card, which is always the

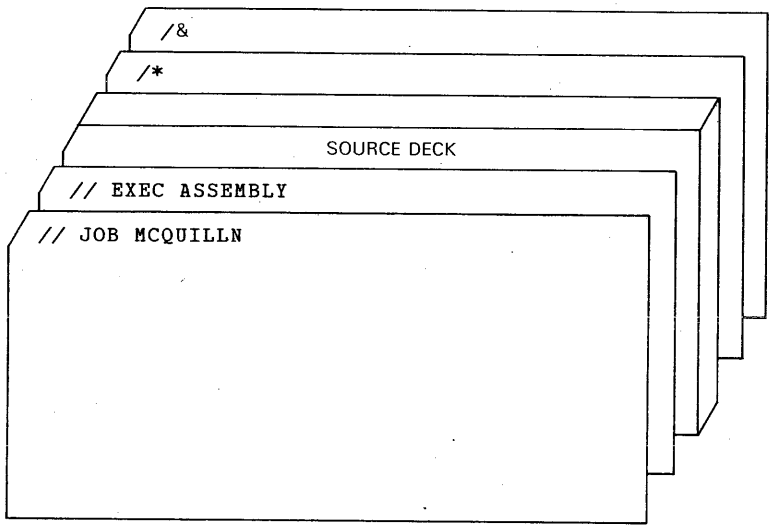


FIGURE 1-14 Job-Control Cards for Assembly

first card for a job, and the /& card, which is always the last card for a job. The job in this example consists of one job step since there is only one EXEC card.

Usually, when a program is assembled using the cards just described, no object deck is punched and no object program is written on disk. The reason is that all DOS systems have a number of standard options that are automatically performed. For a typical DOS system, the standard assembly options are *not* to punch an object deck and *not* to write an object program on disk.

The standard options can be overruled by using the OPTION card. For example, if a programmer wants the object deck to be punched, he can use the following cards for assembly.

```
// JOB MCQUILLN
// OPTION DECK
// EXEC ASSEMBLY
   (source deck)
/*
/&
```

Similarly, the following cards will cause the object program to be stored on disk.

```
// JOB MCQUILLN
// OPTION LINK
// EXEC ASSEMBLY
   (source deck)
/*
/&
```

Because of the time involved in punching cards and the difficulty in handling object decks, the DECK option is rarely used, while the LINK option is used whenever a programmer is ready to test his program. When an OPTION card is used, the options stay in effect until the next /& card or the next JOB card, at which time the standard options are put back into effect.

Assembling and Testing a Program

Before an object program can be executed using the Disk Operating System, it must be processed by the *linkage-editor program,* which is another of the programs of the operating system. The linkage editor stores the object program in one of the libraries of the system-residence device in a form that can be loaded by the supervisor. Assembling and testing a BAL program, then, is actually done in three steps: first, the source deck is converted to an object program that is stored on disk; second, the linkage-editor program converts the object program into a form ready for execution and stores it in one of the DOS libraries; third, the object program is loaded and executed.

Figure 1-15 shows the job-control cards required for assembling and executing a BAL program involving card input and printer output. The cards for the first job step are:

```
// JOB MCQUILLN
// OPTION LINK
// EXEC ASSEMBLY
   (source deck)
/*
```

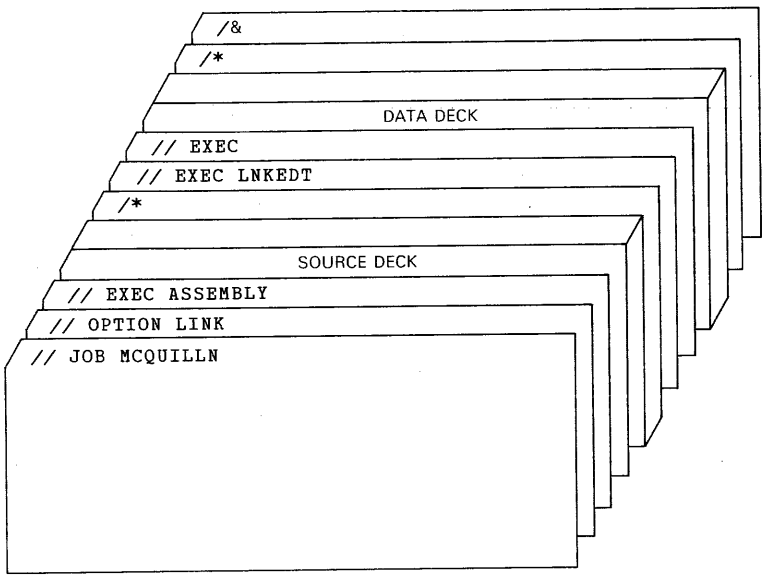


FIGURE 1-15 Assembling and Testing

As explained earlier, this causes the assembly to take place, and the resulting object program to be stored on disk.

The second job step is executing the linkage-editor program. The job-control card for this step is:

```
// EXEC LNKEDT
```

LNKEDT is the name for the linkage-editor program. It is loaded from the system-residence device and executed. When this job step is finished, the object program to be tested is stored in one of the libraries of the system-residence device, ready for execution.

The third job step is testing the program. The job-control cards for this step are:

```
// EXEC
   (data deck)
/*
```

Because the EXEC card doesn't specify a program name (as in // EXEC ASSEMBLY or // EXEC LNKEDT), it indicates that the program can be found in a temporary library area on the system-residence device. When the next program to be tested is link-edited, it will replace this first program in the temporary library area.

When preparing BAL programs, the job setups illustrated in figures 1-14 and 1-15 are all that is needed for card-to-printer programs. Normally, the first setup —assemble only, as shown in figure 1-14—is used until all diagnostics have been corrected. Then, the second job setup—as shown in figure 1-15—is used for a series of test runs. Each time an error is debugged, the source deck is changed, and the program is reassembled and tested again.

Terminology

operating system
stacked-job processing
Disk Operating System
DOS
DOS BAL
library
system-residence device
supervisor program
supervisor
job-control card
job deck
job-control program
job-control language
jobname
job
job step
linkage-editor program

Objective

Make up job decks for assembling and for assembling and testing a BAL program that requires card input and printer output.

2

System/360-370 CPU Concepts

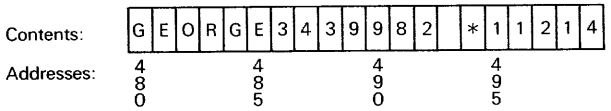
To program in Basic Assembler Language, a programmer needs considerable knowledge of the internal organization and operation of the System/360-370. This chapter is designed to provide that knowledge. Topic 1 describes how data and instructions are stored in the System/360-370 and how some commonly used instructions operate. Topic 2 presents some concepts that you must understand in order to be able to code I/O operations in assembler language.

TOPIC ONE Data and Instructions When a program is loaded into a computer, it is placed in the storage of the CPU. That's why a computer program is often referred to as a stored program. That's also why the length and complexity of a program depend to a certain extent on the storage capacity of the computer. Small computers may have only a limited amount

of storage—say 8000 *storage positions*—while some of the largest computers have more than one million storage positions. Because the word *kilo* refers to one thousand, *K* is often used to refer to 1000 storage positions: a 16K computer has approximately 16,000 storage positions. (I say approximately because one K is actually 1024 storage positions, so 16K represents 16,384 storage positions. In normal conversation, however, the excess storage positions are dropped.)

Associated with each of the storage positions is a number that identifies it; this number is called the *address* of the storage position. An 8K computer, for instance, has addresses ranging from 000 to 8095. Therefore, you might talk about the contents of the storage position with address 180, the contents of storage position 4482, and so on.

In a System/360-370, data can be stored in four different forms. In one form, one character is stored in each storage position. To illustrate this form, suppose the following boxes represent the twenty storage positions from 480 through 499.



In this case, you can say that storage position 480 contains the letter G, storage position 487 contains the number 4, position 494 contains an asterisk, and position 493 contains a blank. Or you can say that there is a 2 at address 497 and the number 343 is stored in positions 486 through 488. This is simply the way programmers talk about storage and its contents.

Several consecutive storage positions that contain one item of data such as item number or unit price are commonly referred to as a *field*. For example, storage positions 486-490 above (this is read as 486 through 490) might represent a balance-on-hand field, while positions 495-499 might represent an item-number field. To address a field, a System/360-370 instruction specifies the address of the leftmost storage position as well as the number of storage positions in the field. Thus, address 486 with a length of 5 would address the field in positions 486-490, while address 1024 with a length of 20 would address the field in positions 1024-1043.

Of course, a storage position isn't really a small box with a character of data in it. Instead, each storage position consists of a number of electronic components that are called *binary components* because they can be switched to either of two conditions. These two conditions are commonly referred to as "on" and "off."

In the System/360, the binary component used is the *magnetic core*. This tiny, doughnut-shaped component can be magnetized in either of two directions: clockwise or counterclockwise. When magnetized in one direction, a core is said to be on; when magnetized in the other direction, a core is said to be off. A string of cores makes up one storage position in a computer, and thousands of cores in planes make up a computer's storage. Because most storage consists of magnetic cores, you will often hear storage referred to as *core storage*.

In order to represent data, the cores at a storage position are turned on or off in selected combinations by wires that run through the center of the cores. Each combination of on or off cores represents a digit or digits, a letter, or a special character. Figure 2-1, for example, represents three storage positions. By decoding the combinations at each storage position, it can be determined that the characters B, 2, and 9 are stored. (The shaded cores are on, the white cores are off.)

Because the codes of a computer are represented by binary components, the *binary digits* 0 and 1 can be used to represent binary code. If 1 is used to represent an on-core and 0 is used to represent an off-core, 11000010 can be used to represent the first string of cores in figure 2-1. Since a binary digit is often referred to as a *bit* (a contraction of *bi*nary and dig*it*) magnetic cores are often called bits also.

In the System/370, magnetic cores are not used. Instead, transistor-like solid materials are used as the binary components. Regardless of the components used, though, the principles are the same: each binary component can be

referred to as a bit, eight bits represent one storage position, and one or more storage positions represent a field.

Actually, each storage position of the System/360-370 is commonly called a *byte* of data. Furthermore, this byte, and thus the storage position itself, is made up of nine bits—eight data bits plus one *parity bit*. The parity bit is used as a check on operations that take place within the CPU. Each time a byte of data is moved into or out of storage during the execution of a program, the byte is *parity checked*. That is, the number of on-bits in the byte is checked to make sure that it is an odd number. If the number of on-bits is even, as in the code 011110011, an error is indicated and the system stops executing the program.

The parity bit at a storage position in the System/360-370 is used to make any bit combination odd. As a result, the complete code for the letter A is 011000001, where the leftmost bit is the parity bit and the rightmost eight bits are the data bits. Similarly, the complete code for the number 9 is 111111001, where the leftmost bit has been turned on to make the number of on-bits odd.

As I mentioned before, there are four forms in which data can be stored in the System/360-370. Three of these are covered in this chapter. They are *EBCDIC*, *packed decimal*, and *fixed-point binary* (or just *binary*). The fourth form, which is generally not used by the assembler-language programmer, is called *floating-point binary*; it is presented in chapter 6.

Regardless of the form of data, the parity bit is used in each byte of storage. However, the parity bit is usually ignored when discussing specific codes or storage forms, and the System/360-370 is usually said to have an eight-bit, rather than a nine-bit, byte. For this reason, no further mention of parity checking will be made in this chapter. After all, parity checking is simply an electronic check on the accuracy of internal operations.

DATA STORAGE

Because it is awkward to work with binary codes, *hexadecimal* notation is commonly used to represent the

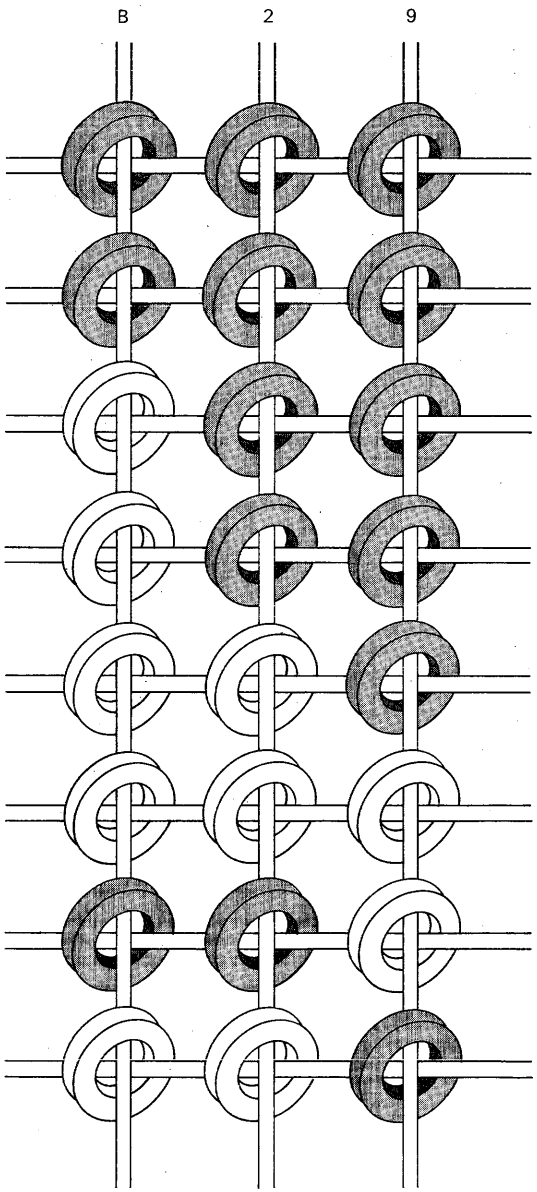


FIGURE 2-1 Core Storage

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10

FIGURE 2-2 Hexadecimal Chart

contents of System/360-370 storage. The intent of hexadecimal is to provide a method of shorthand whereby one group of four binary digits is replaced with one hexadecimal character. Figure 2-2 shows the relationship between binary, decimal, and hexadecimal notation. Thus, the binary 1111 is written as F in hexadecimal (or *hex*); the binary 1001 is a hex 9; and binary 0110 is a hex 6. As you will discover later, much of the debugging done in assembler language requires working with hex notation.

EBCDIC

In EBCDIC code (pronounced ee'-bee-dick or ib'-si-dick), each byte of storage contains one character of data. Because eight bits can be arranged in 256 different combinations, 256 combinations could be coded in EBCDIC. As a result, EBCDIC can be used to represent the letters of the alphabet (both upper and lower case), the decimal digits, and many special characters. However, not all of the 256 combinations are used.

Figure 2-3 gives the EBCDIC codes for the more commonly used characters. It shows the binary code for each character, the punched-card code, and the hexadecimal code. As indicated in the figure, it is common to divide an EBCDIC code into two halves. The leftmost four bits represent the *zone bits*; the rightmost four bits represent the *digit bits*. When this is done, you can see a relationship between EBCDIC and punched card code. For letters and numbers, the zone bits 1100, or hex C, correspond to a 12-punch; the bits 1101, or hex D, to an 11-punch; the bits 1110, or hex E, to a 0-punch; and 1111, or hex F, to no zone punch.

The digit bits, on the other hand, correspond even more closely to the card punches. For example, hex 0 corresponds to a 0-punch; hex 1 to a 1-punch, and hex 9 to a 9-punch.

For special characters, other zone bit combinations are used. Thus, 01011011 is used for the dollar sign and 01001101 for the left parenthesis. Although it is sometimes handy for a programmer to know the EBCDIC codes for numbers and letters, decoding special characters is usually unnecessary. If it is required, the codes can easily be looked up in reference tables.

To represent EBCDIC data in storage, hex can be used as in this example:

```
E2 C1 D4 40 40 40 40 40 40 40
```

Here, the name SAM is stored in ten bytes (hex 40 represents a blank). Similarly, a six-byte numeric EBCDIC field containing 1234 can be shown as:

```
F0 F0 F1 F2 F3 F4
```

This EBCDIC form of representing numbers is often referred to as *zoned decimal*. (You should notice from these two examples that an alphanumeric field is normally left-justified with blanks filling out the unused bytes to the right of the data, while a numeric field is normally right-justified with zeros filling out the unused bytes in the left of the field.)

To represent the sign of a zoned-decimal field, the zone portion of the rightmost byte of the field is used. If the zone portion is 1111 (hex F) or 1100 (hex C), the number is positive; if the zone portion is 1101 (hex D), the number is negative. Thus, a positive 1234 in four storage positions can be shown as:

```
F1 F2 F3 C4
```

A negative 1234 can be shown as:

```
F1 F2 F3 D4
```

Packed Decimal

Packed decimal, in contrast to zoned decimal, is a more compact form of System/360-370 storage. Except for the rightmost byte of a packed-decimal field, two decimal digits are stored in each eight-bit byte. The rightmost byte of the field contains a decimal digit in its zone half and the sign of the field in its digit half. Using hex notation, the number +12345, stored in three bytes, is shown as follows:

```
12 34 5C
```

(Remember that either C or F is a valid sign for a positive field.) If the number −12345 is stored in four bytes, it can be shown as:

```
00 12 34 5D
```

This illustrates that leading zeros must fill out the positions of a packed-decimal number if the number has fewer digits than the field allows.

Packed-decimal fields in hex are relatively easy to decode because the decimal digits 0 through 9 are also 0 through 9 in hex. The only problem, then, is determining the sign of the field by analyzing the digit portion of the rightmost byte of the field.

Binary

In a System/360-370, two or four consecutive bytes are used

Character	EBCDIC Zone Bits	EBCDIC Digit Bits	Punched Card Code	Hexadecimal Code
.	0100	1011	12-3-8	4B
(	0100	1101	12-5-8	4D
+	0100	1110	12-6-8	4E
&	0101	0000	12	50
$	0101	1011	11-3-8	5B
*	0101	1100	11-4-8	5C
)	0101	1101	11-5-8	5D
;	0101	1110	11-6-8	5E
-	0110	0000	11	60
/	0110	0001	0-1	61
,	0110	1011	0-3-8	6B
%	0110	1100	0-4-8	6C
?	0110	1111	0-7-8	6F
#	0111	1011	3-8	7B
'	0111	1101	5-8	7D
=	0111	1110	6-8	7E
"	0111	1111	7-8	7F
A	1100	0001	12-1	C1
B	1100	0010	12-2	C2
C	1100	0011	12-3	C3
D	1100	0100	12-4	C4
E	1100	0101	12-5	C5
F	1100	0110	12-6	C6
G	1100	0111	12-7	C7
H	1100	1000	12-8	C8
I	1100	1001	12-9	C9
J	1101	0001	11-1	D1
K	1101	0010	11-2	D2
L	1101	0011	11-3	D3
M	1101	0100	11-4	D4
N	1101	0101	11-5	D5
O	1101	0110	11-6	D6
P	1101	0111	11-7	D7
Q	1101	1000	11-8	D8
R	1101	1001	11-9	D9
S	1110	0010	0-2	E2
T	1110	0011	0-3	E3
U	1110	0100	0-4	E4
V	1110	0101	0-5	E5
W	1110	0110	0-6	E6
X	1110	0111	0-7	E7
Y	1110	1000	0-8	E8
Z	1110	1001	0-9	E9
0	1111	0000	0	F0
1	1111	0001	1	F1
2	1111	0010	2	F2
3	1111	0011	3	F3
4	1111	0100	4	F4
5	1111	0101	5	F5
6	1111	0110	6	F6
7	1111	0111	7	F7
8	1111	1000	8	F8
9	1111	1001	9	F9

FIGURE 2-3 EBCDIC

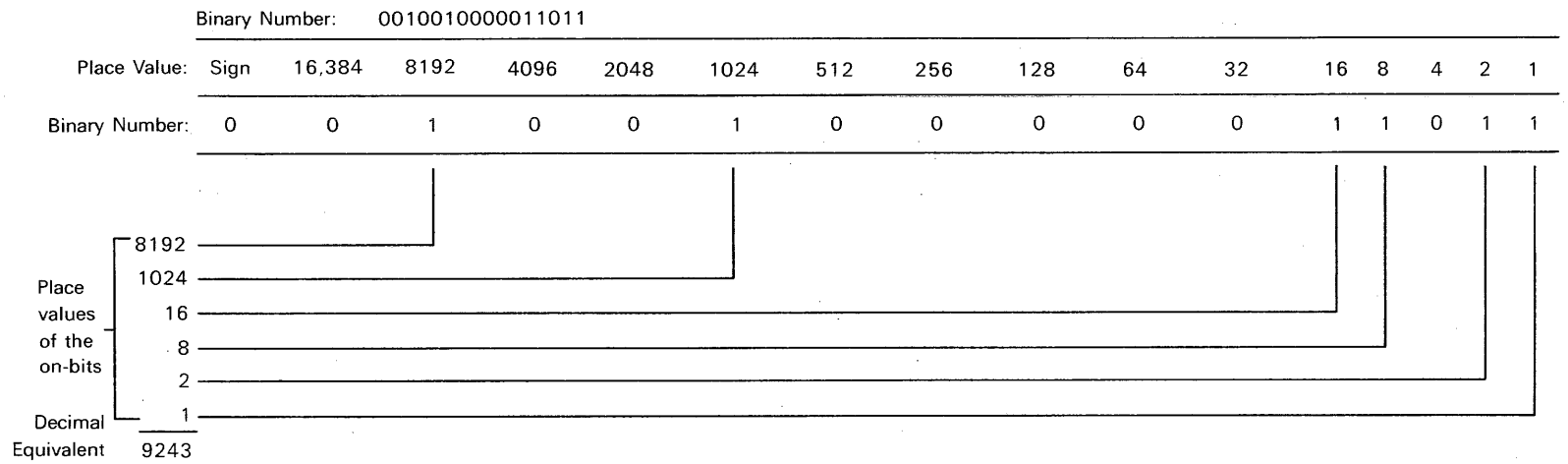


FIGURE 2-4 Binary Coding

for a binary field. Thus, 16 or 32 bits are used to represent a binary number. If the number is represented by using binary digits, it can be converted by assigning a *place value* to each bit. The place values start at the rightmost bit position with a value of 1 and double for each position to the left. Thus, the place values for a 32-bit binary number are from right to left 1, 2, 4, 8, 16, 32, 64, and so on, until the next to the leftmost bit, the 31st bit, is 1,073,741,824. The leftmost bit is used to indicate the sign of the number—if 0, the number is positive; if 1, the number is negative. In a 16-bit binary number, 15 bits are used for the number, while the leftmost bit indicates the sign.

To illustrate binary coding, consider the following binary number: 0010010000011011. Since the leftmost bit is 0, the number is positive. By adding the place values of the on-bits, you can determine that the decimal equivalent is 9243. The decoding process is illustrated in figure 2-4. The maximum value of a 16-bit binary number is a positive 32,767; the maximum value of a 32-bit number is a positive 2,147,483,647.

Using hex, the binary storage of decimal 9243 can be shown as:

```
24 1B
```

If 9243 is stored as a four-byte binary number, it can be shown as:

```
00 00 24 1B
```

To convert hex representation of a positive binary number to decimal, it is common to use a chart such as the one in figure 2-5. By looking up the decimal value of each hex digit in the corresponding column in the chart and then adding the decimal values, the decimal number can be derived. Thus, hex A in the rightmost four bits of a binary number has a value of 10; hex 4 in the next four bits has a value of 64; and so on. When these decimal values are added, the decimal number 15,178 is derived from hex 3B4A. (Incidentally, the hex chart simply reflects the place values of a hex number. The rightmost hex digit has a place value of 1; the next hex digit to its left has a place value of 16; the hex digit to its left has a place value of 256; and so on.)

Because negative binary numbers are not simply binary numbers with a 1-bit preceding them, you will not be able to decode negative hex numbers by using this technique. Fortunately, when common business programming techniques

FULLWORD															
HALFWORD								HALFWORD							
BYTE				BYTE				BYTE				BYTE			
BITS: 0123		4567		0123		4567		0123		4567		0123		4567	
Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	268,435,456	1	16,777,216	1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	536,870,912	2	33,554,432	2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	805,306,368	3	50,331,648	3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	1,073,741,824	4	67,108,864	4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	1,342,177,280	5	83,886,080	5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	1,610,612,736	6	100,663,296	6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	1,879,048,192	7	117,440,512	7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	2,147,483,648	8	134,217,728	8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	2,415,919,104	9	150,994,944	9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	2,684,354,560	A	167,772,160	A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	2,952,790,016	B	184,549,376	B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	3,221,225,472	C	201,326,592	C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	3,489,660,928	D	218,103,808	D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	3,758,096,384	E	234,881,024	E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	4,026,531,840	F	251,658,240	F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15
Place values: 8		7		6		5		4		3		2		1	

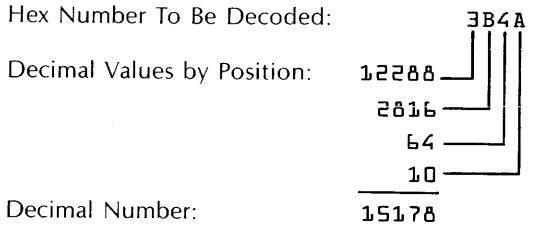


FIGURE 2-5 Decoding a Hex Number

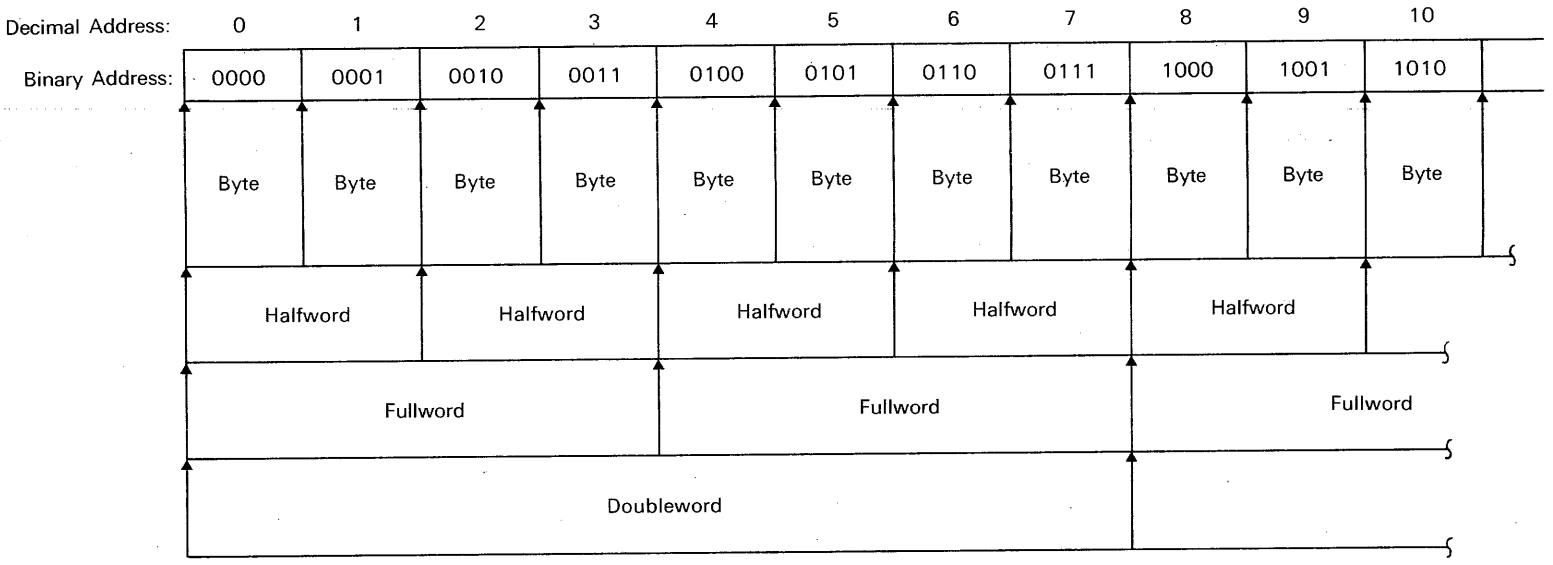


FIGURE 2-6 Storage Boundaries

are used, binary fields should never be negative. For the most part, then, it is enough to know that any hex representation of a binary number in which the leftmost hex digit is 8 or greater is a negative number.

On the System/360-370, a two-byte binary field is referred to as a *halfword* and a four-byte binary field as a *fullword*. For some conversion operations, an eight-byte field known as a *doubleword* is also used. These fields must have the proper *boundary alignment* before they can be operated upon by System/360 instructions, although boundary alignment is not required by the System/370. On the 360, a halfword, for example, must start at a storage address that is a multiple of 2 (like address 0, 2, 4, 6, 8, or 10). This is called a *halfword boundary*. Similarly, a fullword must start at a *fullword boundary* (an address that is a multiple of 4), and a doubleword must start at a *doubleword boundary* (an address that is a multiple of 8). Figure 2-6 summarizes this terminology as it applies to the first ten positions of storage.

With three different types of data representation, the number of storage positions required for a numeric field on the System/360-370 is determined by the number of digits in the field and the data format used. If, for example, a field is supposed to contain the number +205,597,474, it requires nine bytes of storage using zoned decimal, five bytes using packed decimal, and a fullword (four bytes) using binary. If a field consists of only one decimal digit such as the number +7, one byte is required using zoned decimal, one byte using packed decimal, and a halfword (two bytes) using binary. Figure 2-7 summarizes these examples.

INSTRUCTION FORMATS

While a program is being executed, both the instructions of the program and the data being processed are contained in storage. The instructions, in coded form, indicate both the operations that are to be performed and the addresses and lengths of the fields that are to be operated upon. In the

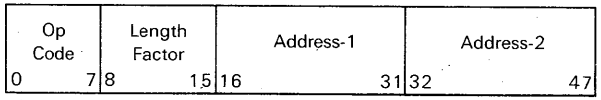
System/360-370, instructions are two, four, or six storage positions in length, depending on the function of the instruction.

To illustrate the parts of a typical instruction, consider one of its basic move instructions called the *move-characters instruction*. This instruction causes the data from one field to be moved unchanged to another field. If, for example, a move-characters instruction specifies that an EBCDIC field in bytes 1551-1555 should be moved to bytes 1701-1705, the execution of the instruction can be shown in hex notation as:

	Receiving Field	Sending Field
Before:	F0 F4 F3 F9 F9	F0 F0 F7 F0 F1
After:	F0 F0 F7 F0 F1	F0 F0 F7 F0 F1

The effect is that data in one field, called the *sending field*, is duplicated in the second field, called the *receiving field*. In this case, a zoned-decimal value of 701 replaces the previous contents of the receiving field.

The format of this move-characters instruction is:



The first byte (bits 0-7) of this six-byte instruction contains the *operation code*. There is a unique operation code for each of the System/360-370 instructions, and, in the case of the move-characters instruction, the operation code is 11010010, or hex D2. The second byte of the instruction (bits 8-15) is a *length factor*, an eight-bit binary number from 0-255 that indicates how many storage positions should be moved. A length factor of 0 indicates one byte is to be moved; a length factor of 1 indicates two bytes are to be moved; and so on. In other words, length factor plus one is the number of bytes to be moved.

The last four bytes of the instruction represent two addresses that specify the starting locations of the fields involved in the instruction. If the length factor is a binary 9 (indicating that ten bytes should be moved), the ten bytes of data starting at the location specified as address-2 are moved to the ten bytes of storage starting at the location specified as address-1. In this, as in most System/360-370 instructions, the address-2 field is the sending field, while the address-1 field is the receiving field.

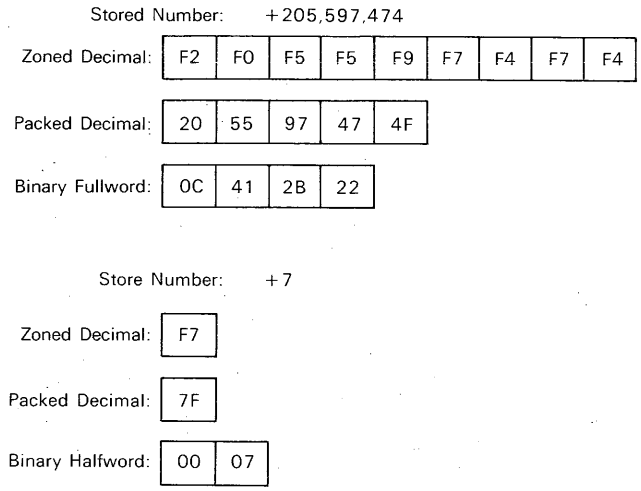
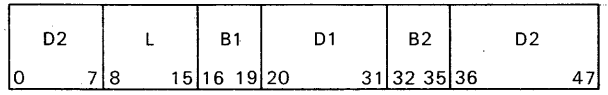


FIGURE 2-7 Storage Use

An address in a System/360-370 instruction is actually made up of two parts: 4 bits that specify a *base register* and 12 bits that represent a *displacement factor*. The base register can be any one of the 16 *general purpose registers* that are components of the System/360-370's CPU. These general purpose registers consist of 32 bit positions—the equivalent of a fullword. When a *register* is used as a base register, the rightmost 24 bit positions are used to represent a *base address*, which is always a positive number. To get the actual address, the base address is added to the displacement factor specified in the instruction.

To be more specific, then, the format of the move-characters instruction is:



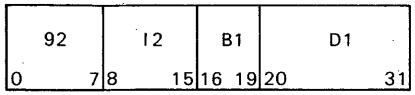
Here, D2 is the actual operation code in hex, L is the length factor, B1 and B2 are four-bit binary numbers that specify one of the 16 registers (numbered 0 through 15 or hex 0 through F), and D1 and D2 are 12-bit binary numbers that represent displacement factors.

With this as background, you should be able to understand that the instruction described above (moving bytes 1551-1555 to 1701-1705) might be represented in hex as:

```
D20432A5320F
```

By breaking this down, you can determine that the operation code is D2; the number of bytes to be moved is 5 (hex 4); address-1 consists of a base address in register 3 plus a displacement of hex 2A5 (decimal 677); and address-2 consists of a base address in register 3 plus hex 20F (decimal 527). If register 3 contains hex 400 (decimal 1024) at the time the instruction is executed, bytes 1551-1555 will be moved to bytes 1701-1705. It is this type of decoding of the parts of an instruction that is done by the CPU at the time that an instruction is executed.

A second form of the move instruction, often called the *move-immediate instruction*, illustrates a four-byte instruction format. When it is executed, one byte of data is moved from the instruction itself to a receiving field. The instruction's format is:



Here, the operation code is hex 92. There is no length code since the move-immediate always involves only one byte, and the data to be moved is stored in the second byte of the instruction (bits 8-15). Bits 16-31 give the base register and displacement of the receiving field.

Suppose, for example, that the move-immediate instruction in hex is:

```
92C1B100
```

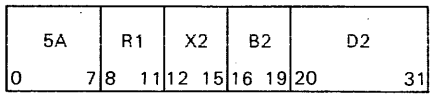
This would move the letter A (hex C1) into the byte at the address computed by adding the contents of register 11 (hex B) to hex 100. If register 11 contains hex 1000, A is moved into the byte at address hex 1100, or decimal 4352. (By using the chart in figure 2-5, you can see that hex 1000 is 4096; hex 100 is 256; 4096 + 256 = 4352.)

Another four-byte instruction format is illustrated by the binary add instruction. This instruction adds the binary number in one fullword of storage to the contents of a general purpose register. If, for example, an instruction indicates that the fullword at address 8000 should be added to the contents of register 7, the instruction execution might be shown in hex as:

	Register 7	Fullword
Before:	00 00 00 10	00 00 00 0A
After:	00 00 00 1A	00 00 00 0A

Thus, the register is the receiving field because it receives the result (hex 1A), which is the sum of register 7's initial contents (hex 10) plus the fullword value (hex 0A).

The format of the fullword add instruction follows:



Here, the operation code is 5A, the receiving register is specified in bits 8-11, and the address of the fullword is given in bits 16-31. No length factor is needed because this instruction always operates on a fullword. (Bits 12-15 can be used to specify an *index register*, but this use, called *indexing*, is not common and is not covered in this book. If bits 12-15 are zero, they are ignored. When indexing is used, the address is derived by adding the base address,

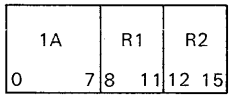
displacement, and the contents of the index register.)

A two-byte System/360-370 instruction is illustrated by the register-to-register add instruction. In this case, a binary number in one register is added to a binary number in another register as in the following example.

	Register 1	Register 2
Before:	00 00 04 01	00 00 00 83
After:	00 00 04 84	00 00 00 83

The contents of register 2 are added to the contents of register 1, and the result is stored in register 1.

The format of this register-to-register instruction is:



Quite simply, the register numbers of the two fields are given along with the operation code, hex 1A.

These, then, are examples of the six-, four-, and two-byte instructions of the System/360-370. In order of presentation, they are often referred to as SS format (storage-to-storage), SI format (immediate-to-storage), RX format (storage-to-register), and RR format (register-to-register). These formats as well as RS format (another form of storage-to-register instructions) and a variation of SS format are presented in figure 2-8.

As you proceed through this book, you will see several examples of each of these types of instructions as they are used within actual Basic Assembler Language programs. By referring to the formats in figure 2-8, you will be able to understand how the assembler-language code relates to the machine language. You will also be able to appreciate the value of assembler language when you see how the language frees the programmer from dealing with actual instruction formats.

INSTRUCTION SET

The term *instruction set* applies to the collection of machine instructions—all valid operation codes—that a computer can execute. Of the 150 or more instructions in the System/360-370 instruction set, most assembler-language programmers use only about 50 of them. The rest are used for highly specialized functions that are rarely needed for business programming.

Any computer's instruction set can be broken down into these functional groups:

1 Data-movement instructions
2 Arithmetic instructions
3 Logical instructions
4 Input/output (I/O) instructions

For the remainder of this topic, examples of System/360-370 instructions are presented for each of these four groups. Because there is a close relationship between the machine-language instructions and assembler language, this material will help you learn the actual assembler code presented in the next chapter more easily.

Data-Movement Instructions

Move-Characters (MVC) You have already been introduced to *data-movement instructions* in the form of the move-characters instruction in the examples of instruction formats. This instruction places the data from the second field specified (called the second *operand*) into the first field specified (called the first operand). The instruction moves one byte of data at a time, processing from left to right through each field.

(Incidentally, MVC is actually the assembler-language operation code for the move-characters instruction. It is called a *mnemonic operation code* because the letters of the code are designed to aid memorization of the code, and the word *mnemonic* refers to memory. In the rest of this topic, the mnemonic operation codes are given along with the description of the instruction. You will see these codes used in the next chapter.)

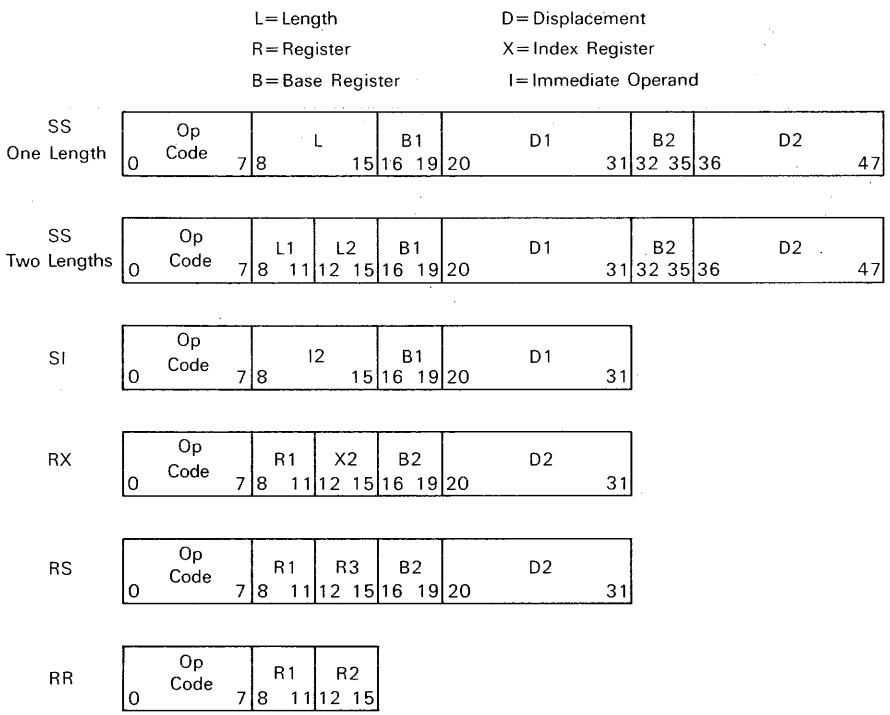


FIGURE 2-8 System/360-370 Instruction Formats

Move-Immediate (MVI) The move-immediate instruction was introduced with the examples of instruction formats. It takes a byte from the instruction and places it into a byte of storage.

Edit (ED) The edit instruction can be used to refine (*edit*) the numeric data in a packed field prior to printing. In its simplest form, for example, the edit instruction might change leading zeros in a packed field to blanks in an EBCDIC field as follows:

	Operand-1	Operand-2
Before:	40 20 20 20 20 20	00 15 8C
After:	40 40 40 F1 F5 F8	00 15 8C

Here, operand-1 must be an *edit pattern* that determines the form in which operand-2 is to be edited. To change lead zeros to blanks (called *zero suppression*), the pattern must consist of a lead blank (hex 40) followed by as many hex 20s as there are digits in the packed field. In this example, since there are five digits in the packed field, operand-1 consists of five hex 20s preceded by a blank. (Incidentally, hex 20 is one of the 256 EBCDIC combinations that does not have a character to represent it. When used in an edit pattern, hex 20 is called a *digit selector*.)

When a more complex pattern is used, the edit instruction can be used to insert a decimal point or commas into a number. For example, the packed-decimal number 0512389 can be edited as follows:

Group	Sending Field	Receiving Field Pattern	Edited Result Field	Printed Result Field
1	12345C	402020202020	40F1F2F3F4F5	12345
Lead Zero	00123F	402020202020	404040F1F2F3	123
Suppression	00000C	402020202020	404040404040	
2	00511F	402020202120	404040F5F1F1	511
Significance	00001C	402020202120	4040404040F1	1
Starting	00000C	402020202120	4040404040F0	0
	123456789C	4020202068202020682020120	40F1F2F36BF4F5F66BF7F8F9	123,456,789
	000123456C	4020202068202020682020120	4040404040F1F2F36BF4F5F6	123,456
	000000123C	4020202068202020682020120	4040404040404040F1F2F3	123
3	000000000C	4020202068202020682020120	404040404040404040404040F0	0
Decimal Point	123456789C	40206B2020206B20202014B2020	40F16BF2F3F46BF5F6F74BF8F9	1,234,567.89
and Comma	000000123C	40206B2020206B20202014B2020	404040404040404040F14BF2F3	1.23
Insertion	000000005C	40206B2020206B20202014B2020	40404040404040404040404BF0F5	.05
	1234567C	402020202020214B2020	40F1F2F3F4F54BF6F7	12345.67
	0000123F	402020202020214B2020	40404040F14BF2F3	1.23
	0000005D	402020202020214B2020	4040404040404BF0F5	.05

FIGURE 2-9 Editing

	Receiving Field	Sending Field
Before:	40 20 20 6B 20 20 21 4B 20 20	05 12 38 9C
After:	40 40 F5 6B F1 F2 F3 4B F8 F9	05 12 38 9C

Because hex 6B is the comma and hex 4B is a decimal point, the resulting receiving field would print as: 5,123.89.

Figure 2-9 gives several examples of the patterns required for suppressing lead zeros and inserting commas and decimal points. In all cases, the leftmost pattern character, called the *fill character*, is the blank so that lead zeros are changed to blanks. Also, either a digit selector (hex 20) or a *significance starter* (hex 21) must be used wherever a digit from the sending field is to be placed. The significance starter is used for the same purpose as a digit selector except that it also indicates where lead zeros should start printing. In group 2, for example, the pattern 40 20 20 20 21 20 causes a packed-decimal field containing zero to be converting into a form that prints as 0. In other words, lead zeros print if they come to the right of the significance starter.

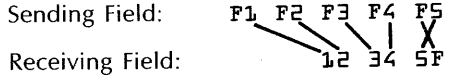
In group 3, commas and decimal points are used in patterns as *message characters*. As instruction execution proceeds from left to right through the sending and receiving fields, the message characters are unchanged if a *significant digit* (a non-zero digit) or the significance starter has been encountered. Thus, the commas and decimal points are inserted into the edited result. On the other hand, if significance hasn't been started by either a significant digit or the significance starter, a message character is changed to a blank (the fill character). Although message characters can be any hex code other than hex 20, 21, or 22, hex 6B (the comma) and hex 4B (the decimal point) are most commonly used.

With these examples as guides, you should now be able to create a pattern that will do simple editing on any packed field. In chapter 8 examples are given of more sophisticated editing patterns. The main point to remember now is that each pattern must consist of a leftmost fill character as well as one digit selector or significance-starter for each digit in the packed-decimal sending field. Logically, there can only be one significance starter in an edit pattern. An important thing to notice in the examples is that the sign of a sending field is converted to hex F when edited, regardless of whether it is plus (hex C or F) or minus (hex D).

Pack (PACK) When data is read into storage from a punched card, it is in EBCDIC form—one character per byte. However, since all System/360-370 arithmetic must be done on packed-decimal or binary fields, EBCDIC fields must be converted to packed decimal or binary before they can be operated upon. The *pack* instruction, then, is one of the conversion instructions. It takes an EBCDIC sending field and converts it into packed decimal in the receiving field as in this example:

```
                Receiving Field     Sending Field
Before:         99 99 99            F1 F2 F3 F4 F5
After:          12 34 5F            F1 F2 F3 F4 F5
```

As you can see, the zone and digit halves of the rightmost character of the sending field are reversed in the rightmost byte of the receiving field. Thereafter, the digit portions of the bytes in the sending field are packed two digits per byte in the receiving field. A more detailed view of the operation of the pack instruction can be shown as:



The pack instruction, which—in contrast to the move or edit instruction—proceeds from right to left during execution, uses the second form of SS format shown in figure 2-8. This means that both sending and receiving fields have length factors. Therefore, it is possible to *pad* a field as in this example:

```
Sending Field (Length 4):     F2 F6 F4 F8
Receiving Field (Length 4):   00 02 64 8F
```

In other words, unfilled half-bytes are padded with zeros.

It is also possible to *truncate* a field as in this example:

```
Sending Field (Length 5):     F6 F3 F2 F0 F4
Receiving Field (Length 2):         20 4F
```

Because the receiving field is too small to receive all digits in the sending field, two significant digits are truncated.

Unpack (UNPK) In contrast, the *unpack* instruction converts a packed-decimal field into an EBCDIC field as in this example:

```
Sending Field:          56 43 7F
Receiving Field:   F5 F6 F4 F3 F7
```

As in the case of the pack instruction, both operands have length factors, so padding can take place:

```
Sending Field (Length 2):                 03 4C
Receiving Field (Length 5):   F0 F0 F0 F3 C4
                              padded
                               bytes
```

And truncation can take place:

```
                                truncated
                                 digits
Sending Field (Length 3):       12 91 2D
Receiving Field (Length 2):        F1 D2
```

In addition to pack and unpack, there are instructions that convert numbers from packed decimal to binary and vice versa. These data-movement instructions are covered in chapter 5.

Arithmetic Instructions

When arithmetic is done on packed-decimal fields, it is referred to as *decimal arithmetic*. Because less conversion is required for decimal arithmetic than for binary arithmetic, decimal arithmetic is used for most of the arithmetic done in business programs. The five basic decimal instructions are add, subtract, multiply, divide, and zero-and-add.

Add-Decimal (AP) When the add-decimal instruction is executed, operand-2 is added to operand-1 and the result replaces operand-1. The second SS format is used for this instruction, so fields of different lengths—up to 16 bytes each—can be added as in this example:

	Operand-1	Operand-2
Before:	10 00 0F	01 0F
After:	10 01 0C	01 0F

Since 10 is added to 10000, the result is 10010. Either positive or negative fields can be added, and the sign of the result field is always hex C for positive or hex D for negative.

One thing to watch when coding decimal operations is *arithmetic overflow*. This takes place when the receiving field is not large enough for the result. In the following example, arithmetic overflow occurs.

	Operand-1	Operand-2
Before:	87 11 0F	40 00 0F
After:	27 11 0C	40 00 0F

Since the actual result is 127110, the leftmost digit has been truncated due to the overflow. Because arithmetic overflow during the execution of a program will generally cause the program to be stopped and cancelled, overflow should be avoided by making the receiving field large enough for any possible result.

Subtract-Decimal (SP) The subtract-decimal instruction operates in the same manner as the add instruction except that the operation performed is subtraction. Once again, the programmer must avoid overflow by making the receiving field large enough for any possible result.

Multiply-Decimal (MP) When the multiply-decimal instruction is executed, the first operand (the multiplicand) is multiplied by the second operand (the multiplier), and the result replaces the first operand. The multiplicand can be up to 16 bytes long and the multiplier up to 8 bytes long. To prevent overflow, the first operand should have at least as many high order zeros as the second operand has digits.

	Operand-1	Operand-2
Before:	00 08 0C	50 0F
After:	40 00 0C	50 0F

Divide-Decimal (DP) The divide-decimal instruction is similar to multiply decimal with one additional complication: the remainder. The first operand (the dividend), can be up to 16 bytes long, and the second operand (the divisor), can be up to 8 bytes long. The divisor is divided into the dividend, and the resulting quotient *and* the remainder replace the dividend. Here is an example:

	Operand-1	Operand-2
Before:	00 00 00 05 3C	00 7C
After:	00 00 7C 00 4C	00 7C
	quotient remainder	

The remainder portion of the result is always the same number of bytes as the divisor and is located in the rightmost bytes of the operand-1 area. The quotient portion of the result, complete with valid sign, occupies the rest of the dividend area. Because of this, the programmer must make sure that operand-1 has at least as many leading bytes filled with zeros as there are bytes in the divisor.

Zero-and-Add (ZAP) The zero-and-add instruction operates in the same manner as an add-decimal instruction that adds a packed field to another packed field containing a value of zero. Here's an example of a zero-and-add instruction:

	Operand-1	Operand-2
Before:	12 45 9C	1C
After:	00 00 1C	1C

In other words, operand-1 was first changed to zero and then operand-2 was added to it.

Logical Instructions

The basic logical instruction and the basis of logic in the computer is the *branch instruction*. When a program is initially loaded into storage, the object program specifies the storage positions in which the instructions are to be stored and the storage position at which the computer is to begin executing the program. When the computer finishes executing one instruction, it continues with the next instruction in storage. After executing the instruction in bytes 1000–1005, for example, the computer executes the instruction starting at address 1006. The only exception to this is the branch instruction. When the branch instruction is executed, it can cause the computer to break the sequence and jump to the instruction beginning at the address specified in the branch instruction.

When a branch instruction branches whenever it is executed, it is called an *unconditional branch*. Suppose that an unconditional-branch instruction stored in positions 4032–4035 specifies a branch to address 801. When the instruction is executed, the computer will continue with the instruction starting in storage position 801.

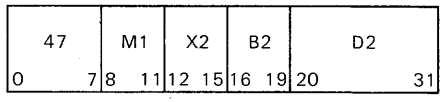
Conditional-branch instructions cause branching only when specified conditions are met. For example, a conditional-branch instruction might branch only if the result of an arithmetic instruction is negative. Suppose such an instruction occupies storage positions 2044–2047 and specifies that the computer should branch to address 1000 if the result of the preceding arithmetic instruction is negative. If the result is zero or positive, the computer continues with the instruction starting at address 2048, the next instruction in sequence. If the result is negative the computer continues by executing the instruction starting at address 1000.

One of the most useful conditions to be branched upon is based on the results of a comparison between two fields in storage. This branch instruction is used in conjunction with the other type of logical instruction, the *compare instruction*. System/360-370 has compare instructions that compare EBCDIC, packed decimal, and binary fields. In the instruction descriptions that follow, the compare-decimal instruction is presented followed by two of the System/360–370 branch instructions. Other compare and branch instructions are presented in other chapters of this book.

Compare-Decimal (CP) The compare-decimal instruction specifies that two packed-decimal fields are to be compared. When it is executed, the computer determines the relationship between the fields: Are they equal? Is the first field greater in value than the second? Is the first field less in value than the second? Based on this comparison, a *condition code* that can be used by subsequent branch instructions is set.

The condition code can be thought of as four bits that are located in the CPU and that can be tested by a branch instruction. For the compare-decimal instruction, if the leftmost bit of the condition code is turned on (bit 0), it indicates that operand-1 and operand-2 are equal. If bit 1 is turned on, it indicates that operand-1 is less than operand-2. If bit 2 is turned on, it indicates that operand-1 is greater than operand-2. The rightmost condition code bit (bit 3) isn't used by the compare-decimal instruction.

Branch-on-Condition (BC) The most widely used branch instruction, called the branch-on-condition, has this RX format:



When this instruction is executed, it compares the *mask* bits (bits 8–11) with the present condition code. If the on-bit in the condition code has a corresponding on-bit in the instruction mask, the program branches to the address specified in bits 12–31 of the instruction (base register plus displacement plus the contents of an index register if bits

12–15 specify a register number other than zero). If the on-bit in the condition code does not have a corresponding 1-bit in the mask, the program continues with the next instruction in sequence.

To illustrate, suppose that the condition code is 0100 after a compare decimal instruction. Then, if the mask in the branch-on-condition instruction is 0100 or 0110, the branch takes place. On the other hand, if the mask is 1011 or 0001, the branch will not take place.

The mask bits in an instruction can be used in any of the 16 combinations that are possible with four bits. This makes it possible for a branch-on-condition instruction to branch on multiple conditions. If, for example, the mask is 1010 following a compare-decimal instruction, a branch will take place when the operands are equal (condition code is 1000) or when operand-1 is greater than operand-2 (condition code is 0010). If all four mask bits are on, the branch is unconditional (it will take place every time); if all four bits are off, it will never branch.

Figure 2-10 summarizes the condition code settings of all of the instructions affecting the condition code that have been presented thus far. The branch-on-condition instruction can branch based on the results of any of these instructions. Note that many instructions, such as the move-characters or multiply-decimal, do not change the condition code in any way.

Branch-and-Link-Register (BALR) The branch-and-link-register instruction is a two-byte instruction in RR format. When this instruction is executed, it places the address of the next instruction in storage in the first register specified. Then, it branches to the address that is given by the contents of the second register specified. However, if zero is specified for the second register, no branch takes place.

To illustrate the execution of this instruction, suppose it is stored in bytes 5000–5001 and specifies register 8 as operand 1, and register 9, which contains the binary equivalent of 2048, as operand 2. When the branch-and-link-register instruction is executed, address 5002 will be stored in register 8 after which the program will branch to address

Condition Code Bit Setting	0	1	2	3
Add Decimal	Zero	Minus	Plus	Overflow
Compare Decimal (A:B)	Equal	A Low	A High	—
Edit	Zero	Minus	Plus	—
Subtract Decimal	Zero	Minus	Plus	Overflow
Zero-and-Add	Zero	Minus	Plus	Overflow

FIGURE 2-10 Condition Code Settings

2048. In contrast, if zero was specified for register 2, 5002 would be stored in register 8 and no branch would take place. You will see this no-branching use of the branch-and-link-register instruction in the next chapter.

I/O Instructions

The System/360–370 *I/O instructions* are perhaps the most complex instructions in the instruction set. In fact, a computer user's program never executes I/O instructions. Instead, all I/O operations are started by the *supervisor program* supplied with the Disk Operating System. Whenever a user's program requires an I/O operation, it branches to the supervisor which in turn causes the appropriate instructions to be executed by *channels*, special components outside of the CPU that are designed for executing I/O instructions. These and some related considerations are described more fully in topic 2 of this chapter.

For now, though, you should become familiar with the overall result of an I/O operation. For example, an input instruction like the card-read instruction might specify that a card is to be read and its data is to be stored in the 80 storage positions beginning with address 5501. In this case, bytes 5501–5580 are called the *card-input area*, or just the *input area*, of storage. When the read instruction is executed, the data that is read from one card replaces the data the card-input area originally contained. All data is stored in EBCDIC code with the data from card column 1 in byte 5501, the data from card column 2 in byte 5502, and so on, until

the data from card column 80 is in byte 5580.

A negative card field is usually indicated by an 11-punch in the rightmost column of the field. Thus, a negative 12544 in a five-column field is read into storage as 1254M (M is the combination of an 11-punch and a 4-punch). In hex, this would be

```
F1 F2 F5 F4 D4
```

and, if packed, the field would be

```
12 54 4D
```

Thus, you can see that the negative sign is maintained from input into the packed form.

An output instruction like a print instruction specifies the bytes from which the output line is to be printed (called the *printer-output area*, or just *output area*, of storage). If a print instruction specifies that a line should be printed from bytes 6601–6732 the content of byte 6601 is printed in print position 1 on the printer (on the far left of the form), the content of byte 6602 is printed in print position 2, and so on.

The data in the printer-output area must be in EBCDIC form prior to printing. Furthermore, a numeric field that carries a hex C or D sign must be edited if it is to print properly. To illustrate, suppose a zoned-decimal field contains a positive 00557 in this form:

```
F0 F0 F5 F5 C7
```

If it is printed, it will print as 0055G since G is the EBCDIC equivalent of hex C7. By packing the field and using the edit instruction prior to printing, however, the lead zeros can be suppressed and the hex C sign changed to hex F.

A similar problem occurs when printing negative fields. For example, a negative 12 will print as 1K. However, if a negative field is packed and edited using the patterns presented thus far, the sign is converted to hex F, which will cause the field to print as 12. To indicate that the field is negative, either a more complex edit pattern or additional instructions must be used.

CONCLUSION

The material you have just finished reading may seem to contain an overwhelming amount of detail. Nevertheless, this is the level of machine knowledge required of the BAL programmer. Because of this, the preliminary knowledge covered in this topic will make your introduction to assembler language in the next chapter much more manageable. If at this time you have a general understanding of the forms in which data and instructions are stored and an appreciation for what happens when the 13 instructions presented here are executed, you should be able to continue with no problems. Later on, should you feel the need, you can refer back to the specific details presented in this topic.

Terminology

storage position
K
address
field
binary component
magnetic core
core storage
binary digit
bit
byte
parity bit
parity checking
EBCDIC
packed decimal
fixed-point binary
binary
floating-point binary
hexadecimal
hex
zone bits
digit bits
zoned decimal
place value
halfword
fullword
doubleword
boundary alignment
halfword boundary
fullword boundary
doubleword boundary
move-characters instruction
sending field
receiving field
operation code
length factor
base register
displacement factor
general purpose register
register
base address
move-immediate instruction
index register

indexing
instruction set
data-movement instruction
operand
mnemonic operation code
edit
edit pattern
zero suppression
digit selector
fill character
significance starter
message character
significant digit
pack
pad
truncate
unpack
arithmetic instruction
decimal arithmetic
arithmetic overflow
logical instruction
branch instruction
unconditional branch
conditional branch
compare instruction
condition code
mask
I/O instruction
supervisor program
channel
card-input area
input area
printer-output area
output area

Objectives

1 Given the hexadecimal code for an EBCDIC, packed-decimal, or binary field, tell what data the field contains. (In the case of a negative binary field, simply indicate that it is negative.)
2 Given all related specifications, codes, and data for any of the 13 instructions described in this topic, indicate what will happen when the instruction is executed. The 13 instructions described are: move-characters, move-immediate, edit, pack, unpack, add-decimal, subtract-decimal, multiply-decimal, divide-decimal, zero-and-add, compare-decimal, branch-on-condition, and branch-and-link-register.
3 Explain what an input or output area is.

Problems

1 (Objective 1) Give the values of the zoned-decimal fields with the following hexadecimal representations?
   a. F0 F0 F0 F5
   b. F4 F3 F9 F2 D2
   c. F0 F1 C1
2 (Objective 1) Using the chart in figure 2-3, tell what the EBCDIC fields with the following hexadecimal representations contain:
   a. C2 C1 D3 40 F1 C1 40 40 40 40
   b. D2 4B 40 E2 4B 40 E2 D4 C9 E3 C8
3 (Objective 1) Give the values of the packed-decimal fields with the following hex representations.
   a. 12 99 5C
   b. 07 7F
   c. 00 00 12 8D
4 (Objective 1) Give the values of the binary fields with the hex representations that follow. If a number is negative, indicate only that it is negative. (The easiest way to convert these fields to decimal is to use the table in figure 2-5 to find the decimal value for each hex digit and add the decimal values.)
   a. 00 41
   b. 80 01
   c. 2A 1F
   d. 00 0B 1D 05
   e. A0 04 AB D1
5 (Objective 2) Below is a table with three columns headed PATTERN, SENDING FIELD (HEX), and EDITED RESULT (CHARACTER). Assuming in each case that an edit instruction edits the data using the pattern given, what would the edited result be if character rather than hex notation was used? (See sample answer given for part "a." below.)

	PATTERN	SENDING FIELD (HEX)	EDITED RESULT (CHARACTER)
a.	40 20 20 21 20 20	05 44 3C	5443
b.	40 20 20 21 20 20	00 00 5C	
c.	40 20 20 21 20 20	39 84 2D	
d.	40 20 6B 20 20 20 6B 20 20 20	12 00 00 5C	
e.	40 20 6B 20 20 20 6B 20 20 20	00 08 00 0D	
f.	40 20 6B 20 20 20 6B 20 20 20	00 00 00 0C	
g.	40 20 20 6B 20 20 21 4B 20 20	00 00 12 3F	
h.	40 20 20 6B 20 20 21 4B 20 20	00 00 00 5C	

6 (Objective 2) Suppose an unpack instruction is used to convert the following packed fields into EBCDIC. Give the hex notation for the unpacked result in a five-byte receiving field.

a. 00 11 2F
b. 55 5D
c. 70 09 45 0C

7 (Objective 2) Complete the following table for the subtract-decimal instruction using hex notation for the packed-decimal result.

	OPERAND-1 (BEFORE)	OPERAND-2	OPERAND-1 (AFTER)
a.	45 55 7C	02 5C	
b.	45 55 7C	65 55 7C	
c.	80 0D	02 0C	
d.	80 0C	00 5D	

8 (Objective 2) Complete the following table for the divide-decimal instruction using hex notation for the packed-decimal result.

	OPERAND-1 (BEFORE)	OPERAND-2	OPERAND-1 (AFTER)
a.	00 00 00 58 4C	01 0C	
b.	00 00 87 40 9C	50 0C	
c.	00 00 00 00 01 39 8C	02 50 0F	

9 (Objective 2) Complete the following table for the compare-decimal instruction using binary notation for the condition code.

	OPERAND-1	OPERAND-2	CONDITION CODE
a.	18 0C	18 0D	
b.	28 1C	28 1F	
c.	01 1F	01 2F	

10 (Objective 2) Suppose register-1 in a branch-and-link register instruction is hex 6 and register 2 is hex zero. If the instruction is stored starting at address 6192, what happens when the instruction is executed?

Solutions

1 a. 5
b. −43922
c. +11

2 a. BAL 1A
b. K. S. SMITH

3 a. +12995
b. +77
c. −128

4 a. 65
b. negative
c. 10783
d. 728,325
e. negative

5 Edited Result
a. 5443
b. 05
c. 39842
d. 1,200,005
e. 8,000
f.
g. 1.23
h. .05

6 a. F0 F0 F1 F1 F2
b. F0 F0 F5 F5 D5
c. F0 F9 F4 F5 C0

7 OPERAND-1 (AFTER)
a. 45 53 2C
b. 20 00 0D
c. 82 0D
d. 80 5C

8 OPERAND-1 (AFTER)
a. 00 05 8C 00 4C
b. 00 17 4C 40 9C
c. 00 00 00 0C 01 39 8C

9 CONDITION CODE
   a. 0010
   b. 1000
   c. 0100

10 Address 6194 is stored in register 6 and the program continues with the instruction starting at address 6194.

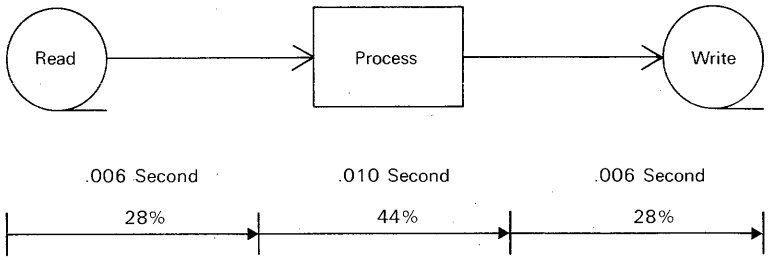


FIGURE 2-11 The Problem of Idle Computer Components

TOPIC TWO Overlap and I/O Instructions Until the middle 1960s, most computer systems could perform only one operation at a time. For instance, a typical card system read a card, processed it, and printed an output line on the printer. It then repeated this sequence for the next card record. Similarly, a typical tape system read input, processed it, and gave output—but only one operation at a time. The trouble with this method of processing was that the components of the computer system were idle, even though the system was running.

To illustrate, suppose a tape system executes a program that reads a tape record, processes it, and writes a tape record. Figure 2-11 might then represent the relative amounts of time that the CPU and the tape drives would be busy during execution of the program. (These percentages would, of course, vary depending on the speed of the components.) During the time the system is running, the CPU in this example is busy 44 percent of the time, while each tape drive is busy only 28 percent of the time.

To overcome the problem of having idle components, systems that could *overlap* I/O operations with CPU processing were developed. The difference between nonoverlapped and overlapped processing is illustrated in figure 2-12. On system A, the nonoverlapped system, nine time intervals are required to read, process, and write three records. On system B, the overlapped system, nine records have been read, eight have been processed, and seven have been written at the end of nine time intervals. Although this example makes the unlikely assumption that reading, processing, and writing take equal amounts of time, the message is clear: overlap can significantly increase the amount of work that a computer system can do.

One reason earlier computers weren't able to overlap is that the CPU executed all of the instructions of a program, one after the other. In contrast, the CPU of an overlapped system like the System/360-370 doesn't execute I/O instructions. Instead, *channels* are used to execute the I/O instructions, while the CPU executes arithmetic, logical, and data-movement instructions. The I/O instructions that the channels execute are called *channel commands*. Figure 2-13 illustrates the components of an overlapped system: one channel executes an input command, a second channel executes an output command, and the CPU executes other instructions of the program—all at the same time. (Although the CPU is generally considered to consist of storage and control circuitry, the CPU and storage are drawn separately in this figure to show that both CPU and channels can access data from storage.)

Whenever data is transferred from a channel to storage or from storage to a channel, CPU processing is interrupted for one *storage-access cycle* (or *access cycle*). Because of the tremendous difference between access-cycle speeds and I/O speeds, however, this is a minor interruption. To illustrate, suppose cards read by a 600-card-per-minute card reader are being processed by a computer that transfers one byte of data during each access cycle. If the time for each access cycle is 1.5 microseconds (millionths of a second), which is the access speed of the System/360 Model 22, CPU processing will be interrupted a total of 120 microseconds

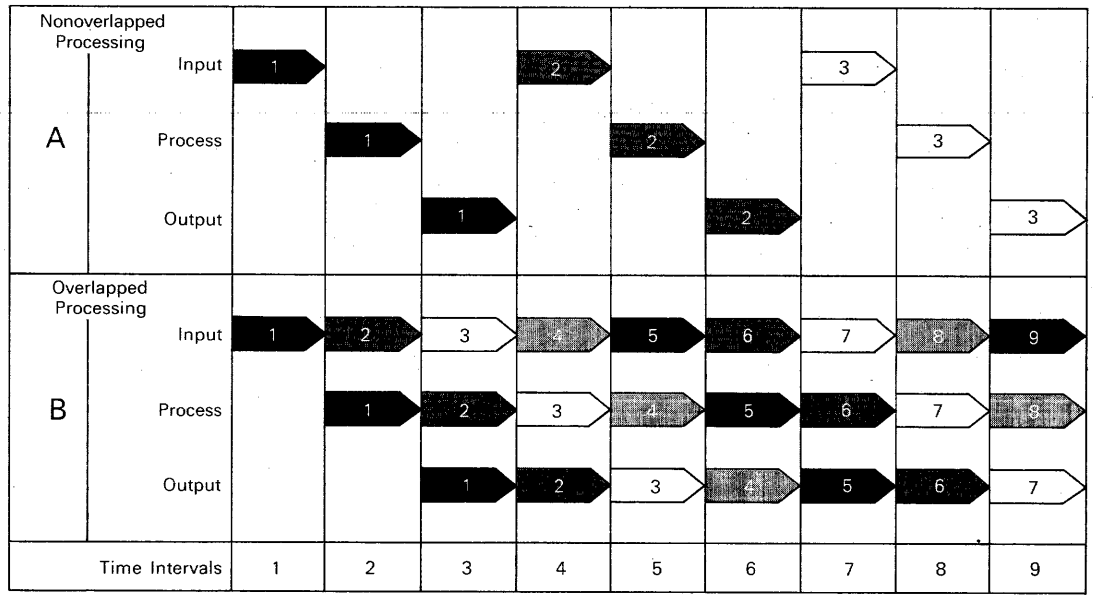


FIGURE 2-12 Overlapped and Nonoverlapped Processing

to read all 80 card columns into storage. In contrast, the card reader takes 1/10 of a second, or 100,000 microseconds to read all 80 card columns. That means that while each card is read, the CPU can spend over 99 percent of its time—99,880 out of 100,000 microseconds to be exact—executing other instructions of the program.

Although tape and disk devices are many times faster than card readers and printers, this same type of inequality is likely to exist between the speeds of these I/O devices and the access-cycle speeds. For example, a tape drive with a 50,000-byte-per-second transfer rate reads or writes one byte of data every 20 microseconds. If the CPU requires 1 microsecond to transfer the byte to or from storage, 19 microseconds per byte are available for other processing. In other words, because of the overlap capability, the CPU can spend 95 percent of its time executing other instructions.

Since a channel, like a CPU, can execute only one operation at a time, the number of overlapped operations that a system can have is limited by the number of channels on the system. For instance, a one-channel system can overlap one I/O operation with CPU processing, a three-channel system can overlap three I/O operations with CPU processing. The one exception to this is the *multiplexor channel*, which can read or write on two or more slow speed I/O devices at one time.

The multiplexor channel has the ability to alternate between several I/O devices. If, for example, a card reader, card punch, and printer are attached to a multiplexor channel, the channel can accept one byte of data from the card reader, send one byte to the card punch, send one byte to the printer, and then return for another byte from the card reader. By switching from one device to another, the single channel makes possible the overlapping of several different devices. Here again, this is possible because of the extreme difference in speeds between I/O devices and access cycles.

Figure 2-14 is a typical configuration of a System/360

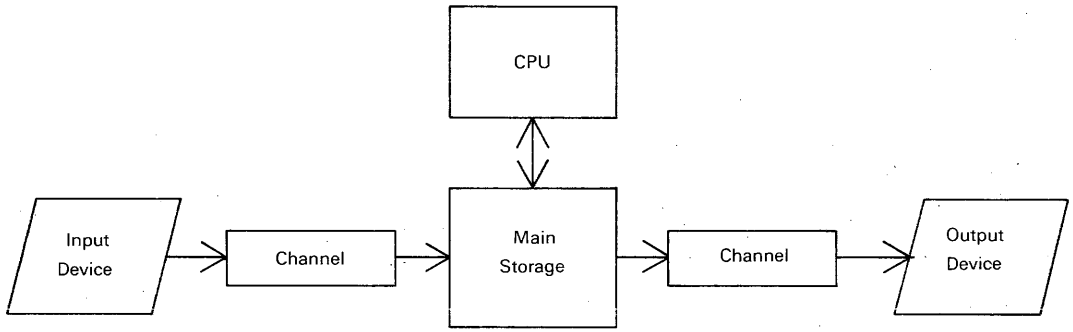


FIGURE 2-13 Computer System with Overlap Capability

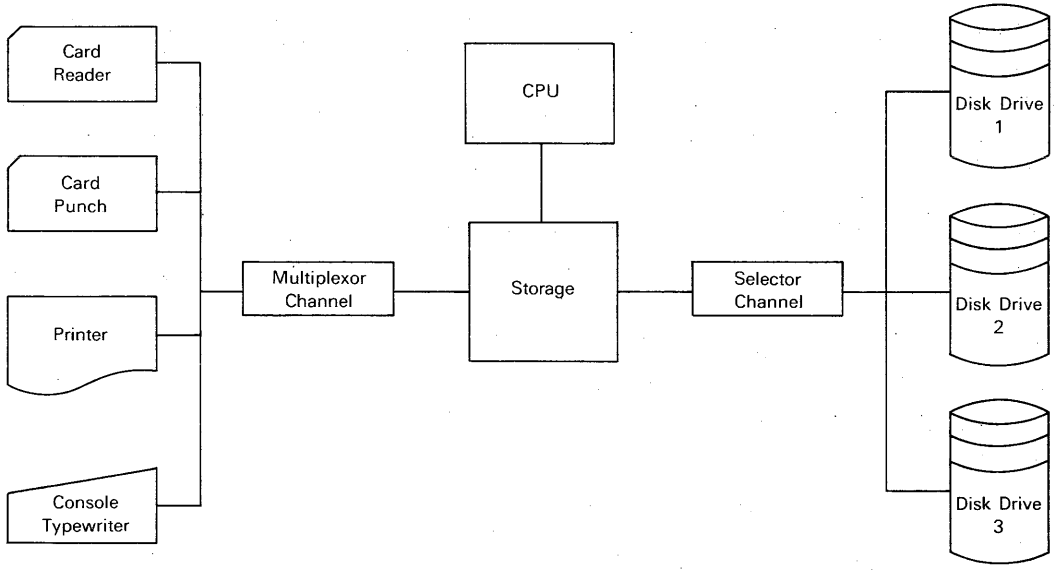
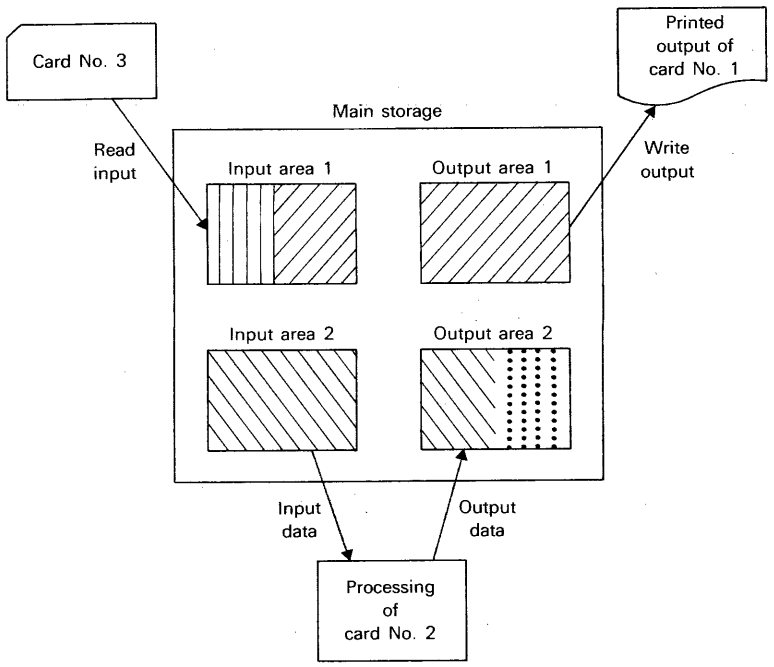


FIGURE 2-14 Typical System/360 Configuration

with three disk drives. All of its slow speed devices—the card reader, card punch, printer, and console typewriter—are attached to a multiplexor channel, while the disk drives are attached to the other kind of channel—a *selector channel.* This system, then, can overlap card reading, printing, punching, console typewriter operations, and reading or writing on one of the disk drives. Because a selector channel can perform only one operation at a time, however, reading from disk drive 1 and writing on disk drive 2 cannot be overlapped. Since disk drives are considerably faster than



Note: Data is being moved into input area 1 and output area 2

FIGURE 2-15 Use of Dual I/O Areas for Overlapped Processing

the slow speed devices, it is more important for slow speed operations to be overlapped than for disk operations. In general, a System/360-370 consists of one multiplexor channel and one or more selector channels.

One programming complexity resulting from overlap is that two I/O areas in storage must be used for each I/O operation. If only one input area of storage were to be used for a card reading operation, for example, the second card would be read into the input area while the first card was being processed—thus destroying the first card's data. Instead, the second card must be read into a second input area of storage while the first card is being processed in the first input area. Then, the third card is read into the first input area, while the second card is processed in the second input area. This switching from one input area to the other, which is shown schematically in figure 2-15, must be continued throughout the program. Similarly, dual I/O areas must be used for all other I/O operations that are overlapped.

To relieve the programmer of the responsibility of both switching I/O areas and coping with the other complexities (such as writing channel commands) that are associated with overlap, the Disk Operating System supplies (1) *I/O modules* and (2) a supervisor program. The appropriate I/O modules, which are loaded into storage along with the assembler-language object program, handle the switching of I/O areas required for overlap. The supervisor, which remains in storage during the execution of all programs, issues the channel commands to the channels for execution.

Although there is a complex relationship between the user's program, the I/O modules, and the supervisor, it is a complexity that the programmer need not be concerned with. As you will see in the next chapter, the programmer codes specifications for each of the I/O files used by his program, and the rest is done by the assembler in combination with the I/O modules and the supervisor. For now, the primary messages are (1) the concept of overlap, and (2) the notion that two I/O areas are required for an overlapped file.

Terminology

overlap
channel
channel command
storage-access cycle
access cycle
multiplexor channel
selector channel
I/O module

Objectives

1 Explain how overlap increases the productivity of a computer system.
2 Explain why two I/O areas are required for each input device if overlap is to take place.

Part 2

Part 2

Assembler Language: The Core Content

This part—in particular, chapter 3—presents the critical material of this book. Once you have mastered it, you will be able to write complete assembler-language programs of great complexity. Furthermore, the rest of the material in this book should be relatively easy in comparison to this core content. *As a result, you should be prepared to put far more effort into the mastery of the material in this part than you will put into any of the other parts of this text.*

3

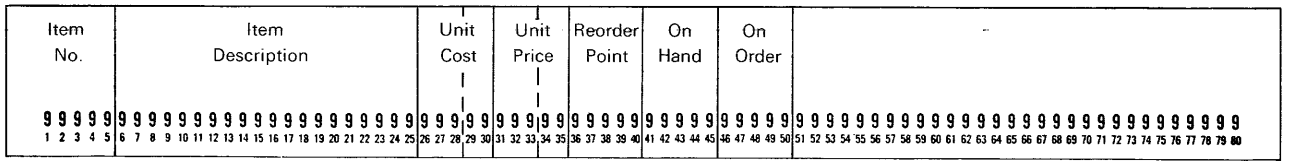
A Subset of Assembler Language

This chapter introduces you to DOS Basic Assembler Language. It does so by presenting programming problems and the BAL solutions for those problems. Since the solutions use BAL code that represents the basic capabilities of the programming language, the code presented can be called a *subset* of the complete Basic Assembler Language. Topic 1 introduces the subset, and topics 2 and 3 present elements that complete the subset.

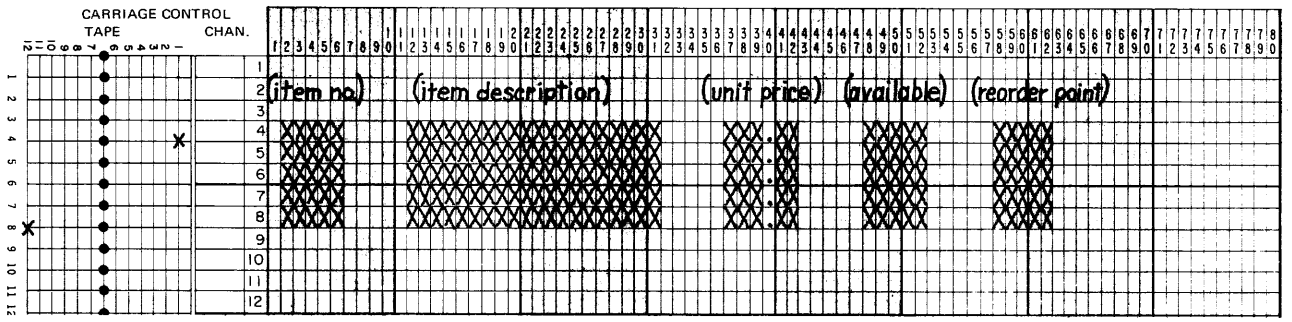
The purpose of this chapter is to get you to see the complete picture right away—that is, how the elements of assembler language work together in a complete program. When you have completed this chapter, you should be able to write complete BAL programs of considerable complexity. Then, the remaining chapters can build on this base of knowledge.

TOPIC ONE An Introduction to Assembler Language Figure 3-1 shows the characteristics of an inventory-control program

Card Layout:



Print Chart:



Processing Specifications:

1. Add on-hand to on-order to derive available.
2. Print a line on the reorder listing only when available is less than the reorder point.

FIGURE 3-1 The Reorder-Listing Problem

that is to be written in assembler language. The input is a deck of inventory cards, one card for each inventory item. The output of the program is a listing of those items in inventory that need to be reordered. The illustration shows the card format and the print positions for each item printed on the output listing. As you can see from the processing specifications, a line is printed whenever an item's available stock (on-hand plus on-order) is less than its reorder point. To simplify your introduction to assembler language, no headings and no final totals are to be printed on this report.

Figure 3-2 is a program flowchart for use in the solution of this problem. After doing some setup for the program (housekeeping), the program reads a card, calculates the amount of available stock, and compares it with the reorder point. If available is greater than or equal to the reorder point, the program branches back and reads another card. Otherwise, it constructs an output line, prints the line, and repeats the processing cycle. When end-of-file (EOF) is reached (that is, a /* card is read), the program ends.

When an assembler-language programmer codes this

program, he uses coding forms similar to the one in figure 3-3. The form in the figure is, in fact, the first page of this reorder-listing program. If you study the form, you will see that 80 columns are indicated from the left to the right margin. When the program is complete, one source-deck card is keypunched for each coding line.

To help both programmer and keypunch operator, groups of columns on the BAL coding form are labeled. Columns 1-8 are called either *label* or *name* and can be used to give a symbolic name to a data field or an instruction. Thus, the branch-and-link register instruction in line 2 of the program has been given the name BEGIN.

Columns 10-14 are called *operation* and columns 16-71 are called *operand*. This means that mnemonic operation codes are given in columns 10-14 and various types of symbolic and actual operands are given in columns 16-71. In the operation field, you can see the codes BALR, MVC, PACK, and so on, to which you were introduced in chapter 2. In columns 16-71, you can see operands such as *, 3, CARDIN, and PRTOUT. You will learn how to code these operands later.

Column 72 of the coding form is used to indicate that a coding line is continued on the next line. This will be explained later in this chapter, so ignore this column for now.

The last eight columns of the form, columns 73-80, can be used to identify and sequence the cards of the source deck. In this program, ORDR will be punched in columns 73-76 with a sequence number in columns 77-80. The sequence numbers are by tens so that, if a line of coding needs to be added to a deck later on, it can be done by using an appropriate numeric value. For example, line 0165 would go between lines 0160 and 0170. Although these columns can help identify a deck or keep it in order, columns 73-80 do not affect the resulting object program in any way. Because of this, the identification and sequence code is often added after the program is completely written, and, in some cases, these columns are left blank.

The heading of the assembler coding form gives such information as program name, date, and programmer. In addition, it gives punching instructions that tell the keypunch operator what symbols that are likely to be confusing represent. In figure 3-3, for example, the programmer is telling the keypunch operator that a Ø represents the number zero while O represents the letter O. Similarly, a Z is written with a bar through it (Z) to distinguish it from a 2, and the number one is written as a vertical line to distinguish it

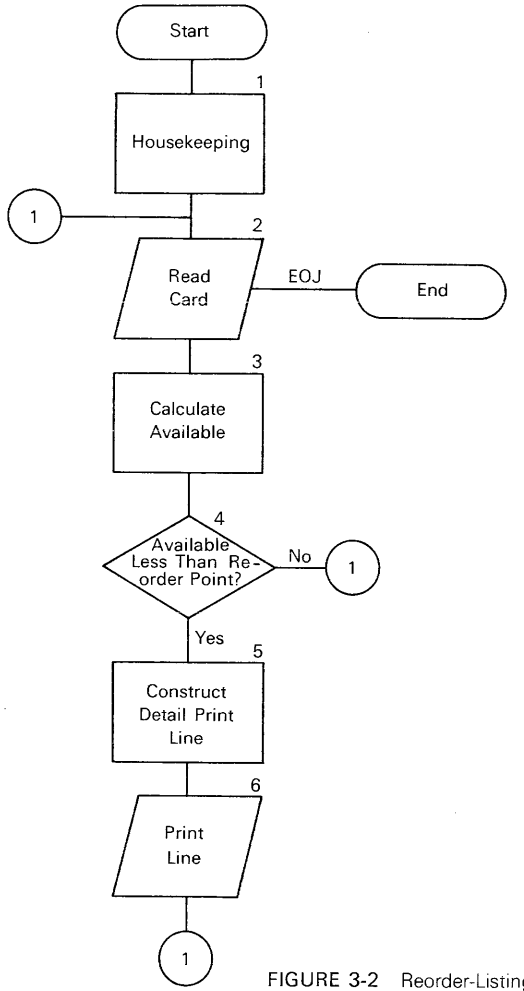
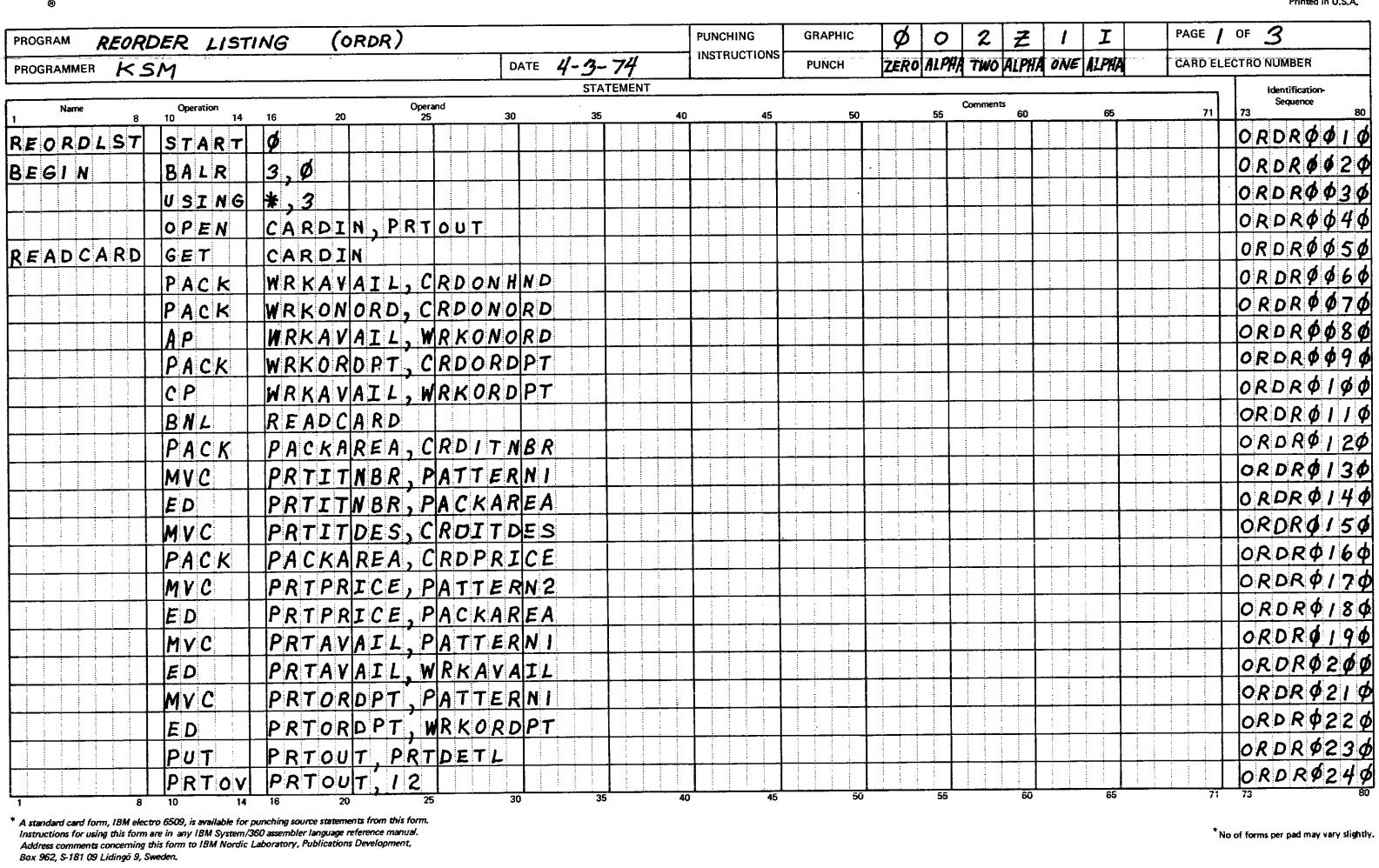


FIGURE 3-2 Reorder-Listing Flowchart



IBM®

IBM System/360 Assembler Coding Form

GX28-6509-6 U/M 050* Printed in U.S.A.

PROGRAM REORDER LISTING (ORDR) | PUNCHING INSTRUCTIONS | GRAPHIC ∅ 0 2 Z 1 I | PAGE 1 OF 3

PROGRAMMER KSM | DATE 4-3-74 | PUNCH ZERO ALPHA TWO ALPHA ONE ALPHA | CARD ELECTRO NUMBER

STATEMENT				Identification-Sequence
Name (1–8)	Operation (10–14)	Operand (16–71)	Comments	(73–80)
REORDLST	START	∅		ORDR∅∅1∅
BEGIN	BALR	3,∅		ORDR∅∅2∅
	USING	*,3		ORDR∅∅3∅
	OPEN	CARDIN,PRTOUT		ORDR∅∅4∅
READCARD	GET	CARDIN		ORDR∅∅5∅
	PACK	WRKAVAIL,CRDONHND		ORDR∅∅6∅
	PACK	WRKONORD,CRDONORD		ORDR∅∅7∅
	AP	WRKAVAIL,WRKONORD		ORDR∅∅8∅
	PACK	WRKORDPT,CRDORDPT		ORDR∅∅9∅
	CP	WRKAVAIL,WRKORDPT		ORDR∅1∅∅
	BNL	READCARD		ORDR∅11∅
	PACK	PACKAREA,CRDITNBR		ORDR∅12∅
	MVC	PRTITNBR,PATTERN1		ORDR∅13∅
	ED	PRTITNBR,PACKAREA		ORDR∅14∅
	MVC	PRTITDES,CRDITDES		ORDR∅15∅
	PACK	PACKAREA,CRDPRICE		ORDR∅16∅
	MVC	PRTPRICE,PATTERN2		ORDR∅17∅
	ED	PRTPRICE,PACKAREA		ORDR∅18∅
	MVC	PRTAVAIL,PATTERN1		ORDR∅19∅
	ED	PRTAVAIL,WRKAVAIL		ORDR∅2∅∅
	MVC	PRTORDPT,PATTERN1		ORDR∅21∅
	ED	PRTORDPT,WRKORDPT		ORDR∅22∅
	PUT	PRTOUT,PRTDETL		ORDR∅23∅
	PRTOV	PRTOUT,12		ORDR∅24∅

* A standard card form, IBM electro 6509, is available for punching source statements from this form. Instructions for using this form are in any IBM System/360 assembler language reference manual. Address comments concerning this form to IBM Nordic Laboratory, Publications Development, Box 962, S-181 09 Lidingö 9, Sweden.

* No of forms per pad may vary slightly.

FIGURE 3-3 The Assembler-Language Coding Form

from the letter *I*, which is written with top and bottom cross members.

A source program can also be illustrated as in figure 3-4. This is an *80-80 listing* of the source deck, so the starting characters in the columns of names, operations, and operands correspond to card columns 1, 10, and 16. (An 80-80 listing is a printout of the contents of a deck of punched cards with the contents of column 1 in print position 1, the contents of column 2 in print position 2, and so on, finishing with the contents of column 80 in print position 80.) In the remainder of this book, all programs will be illustrated as in figure 3-4, but remember that you use BAL coding forms when actually programming.

Take time now to look at the program in figure 3-4. It is the complete program for printing the reorder listing from the inventory cards. As indicated by the brackets to the right of the listing, the program can be divided into (1) instructions, (2) file definitions, and (3) data definitions. The file definitions give the characteristics of the input and output files; the data definitions define fields that are operated upon by the program; and the instructions use names defined by the file and data definitions to describe the sequence of operations required by the program.

To highlight various portions of this program, the programmer has used *comment cards* (or *comments*). A comment card has an asterisk in column 1 and a comment or note written by the programmer in the remaining columns of the card. During assembly, the contents of the cards are printed, but otherwise the cards are ignored. As a result, they can be used by the programmer to make the program listing easier to follow, but they do not affect the resulting object code.

When writing a card-to-printer program like this one, the programmer uses three documents: the card-layout form, the print chart, and the flowchart. In general, the programmer does not write the program straight through from top to bottom, but rather uses two or more coding forms at the same time. When I wrote this program, for example, I wrote the file definitions and the data definitions for the card and printer I/O areas first; then, I began writing instructions on one coding sheet and, whenever needed, I wrote data definitions for any work areas on another coding sheet. After I had written all of the coding lines, I added the identification code and sequence numbers in columns 73-80.

FILE DEFINITIONS

For each file used by a program, there must be a *file definition*. These file definitions are often referred to as *DTFs* (Define The File). In the reorder-listing program there is one DTF statement for the card file, which uses the operation code DTFCD, and one for the printer, which uses the operation code DTFPR. (Although you may not be used to thinking of printer output as a file, it is common practice in programming to refer to it as a print file.)

The card input file is defined by the DTFCD statement in line 290 of figure 3-4. In the name portion of the statement, a label that becomes the *filename* for the card file is given—that is, CARDIN. This label (or name) must (1) start with a letter, (2) consist of only letters and numbers, and (3) be seven characters or less in length. As you will see later, this filename is used when coding input operations in the instruction portion of the program.

After the operation code (DTFCD) in columns 10-14, the operands begin in column 16. These operands give specific characteristics of the file. Each operand consists of a *keyword* (like DEVADDR or IOAREA1) followed by an equals sign (=) followed by programmer-supplied words (like SYSIPT or CRDINPA).

Figure 3-5 summarizes some of the most commonly used keywords of the DTFCD. As you can see, DEVADDR and IOAREA1 are required, but the others are selected by the programmer as they are needed. As you will see later in this chapter, IOAREA2 and WORKA are used if overlap is required. DEVICE is used if the 2540 card reader/punch is not the device being used for card operations. For example, the DEVICE operand might be DEVICE=2520 if the 2520 card reader is used. EOFADDR is commonly used for a card-input file so that a branch will take place when a /* card is read,

```
REORDLST START 0                                                        ORDR0010
BEGIN    BALR  3,0                                                      ORDR0020
         USING *,3                                                      ORDR0030
         OPEN  CARDIN,PRTOUT                                            ORDR0040
READCARD GET   CARDIN                                                   ORDR0050
         PACK  WRKAVAIL,CRDONHND                                        ORDR0060
         PACK  WRKONORD,CRDONORD                                        ORDR0070
         AP    WRKAVAIL,WRKONORD                                        ORDR0080
         PACK  WRKORDPT,CRDORDPT                                        ORDR0090
         CP    WRKAVAIL,WRKORDPT                                        ORDR0100
         BNL   READCARD                                                 ORDR0110
         PACK  PACKAREA,CRDITNBR                                        ORDR0120
         MVC   PRTITNBR,PATTERN1                                        ORDR0130
         ED    PRTITNBR,PACKAREA                                        ORDR0140
         MVC   PRTITDES,CRDITDES                                        ORDR0150
         PACK  PACKAREA,CRDPRICE                                        ORDR0160
         MVC   PRTPRICE,PATTERN2                                        ORDR0170
         ED    PRTPRICE,PACKAREA                                        ORDR0180
         MVC   PRTAVAIL,PATTERN1                                        ORDR0190
         ED    PRTAVAIL,WRKAVAIL                                        ORDR0200
         MVC   PRTORDPT,PATTERN1                                        ORDR0210
         ED    PRTORDPT,WRKORDPT                                        ORDR0220
         PUT   PRTOUT,PRTDETL                                           ORDR0230
         PRTOV PRTOUT,12                                                ORDR0240
         B     READCARD                                                 ORDR0250
CRDEOF   CLOSE CARDIN,PRTOUT                                            ORDR0260
         EOJ                                                            ORDR0270
* THE CARD FILE DEFINITION FOLLOWS                                      ORDR0280
CARDIN   DTFCD DEVADDR=SYSIPT,IOAREA1=CRDINPA,EOFADDR=CRDEOF            ORDR0290
* THE PRINTER FILE DEFINITION FOLLOWS                                   ORDR0300
PRTOUT   DTFPR DEVADDR=SYSLST,IOAREA1=PRTOUTA,BLKSIZE=132,PRINTOV=YES, XORDR0310
               WORKA=YES                                                ORDR0320
* THE DATA DEFINITIONS FOR THE CARD INPUT AREA FOLLOW                   ORDR0330
CRDINPA  CS    0CL80                                                    ORDR0340
CRDITNBR CS    CL5                                                      ORDR0350
CRDITDES CS    CL20                                                     ORDR0360
         CS    CL5                                                      ORDR0370
CRDPRICE CS    CL5                                                      ORDR0380
CRDORDPT CS    CL5                                                      ORDR0390
CRDONHND CS    CL5                                                      ORDR0400
CRDONORD CS    CL5                                                      ORDR0410
         CS    CL30                                                     ORDR0420
* THE DATA DEFINTION OF THE PRINTER OUTPUT AREA FOLLOWS                 ORDR0430
PRTOUTA  CS    CL132                                                    ORDR0440
* THE DATA DEFINITIONS FOR THE PRINTER WORK AREA FOLLOW                 ORDR0450
PRTDETL  CS    0CL132                                                   ORDR0460
PRTITNBR DS    CL6                                                      ORDR0470
         DC    5C' '                                                    ORDR0480
PRTITDES DS    CL20                                                     ORDR0490
         DC    4C' '                                                    ORDR0500
PRTPRICE DS    CL7                                                      ORDR0510
         DC    4C' '                                                    ORDR0520
PRTAVAIL DS    CL6                                                      ORDR0530
         DC    4C' '                                                    ORDR0540
PRTORDPT DS    CL6                                                      ORDR0550
         DC    70C' '                                                   ORDR0560
* THE DATA DEFINITIONS THAT FOLLOW DEFINE OTHER WORK AREAS NEEDED       ORDR0570
* BY THE PROGRAM                                                        ORDR0580
PATTERN1 DC    X'402020202020'                                          ORDR0590
PATTERN2 DC    X'40202021482020'                                        ORDR0600
WRKAVAIL DS    PL3                                                      ORDR0610
WRKONORD DS    PL3                                                      ORDR0620
WRKORDPT DS    PL3                                                      ORDR0630
PACKAREA DS    PL3                                                      ORDR0640
         END   BEGIN                                                    ORDR0650
```

Instructions

File Definitions

Data Definitions

FIGURE 3-4 Reorder-Listing Program

and TYPEFLE=OUTPUT is used when defining a card-output file punched by the card punch.

In line 290 in figure 3-4, the DEVADDR keyword for the card reader specifies SYSIPT. Until you read chapter 16 and learn how these SYS codes relate to the Disk Operating System, always use SYSIPT for a card reader. For IOAREA1, the programmer has used CRDINPA, a name defined in the data definition in line 340, and for EOFADDR, he has used CRDEOF, a name defined by the instruction in line 260. By the time you have completed this topic, you will understand how the file definitions, data definitions, and instructions are related.

The file definition for the printer file is the DTFPR statement that begins in line 310 of figure 3-4 and continues in line 320. As in the DTFCD, the code in columns 1-7 of the statement gives a filename to the file; thus, the programmer has given the name PRTOUT to the printer file. After the operation code, a list of keyword operands gives the characteristics of the file. In this case, the operand list requires two coding lines, so a continuation code is used in column 72. This code can be any character whatsoever (in the example, X is used), and the operands on the coding line that follows must start in column 16.

Figure 3-6 summarizes the keywords for the DTFPR. Here again, DEVADDR and IOAREA1 are required, and the others are selected by the programmer as he needs them. For example, DEVICE=3211 must be used if the printer is a 3211 instead of the more common 1403. All of the other keywords in this list will be used somewhere in this chapter, so they need not be studied now.

In the program in figure 3-4, the DEVADDR keyword specifies SYSLST. This word should be used for all printers until you have read chapter 16 and understand how other SYS numbers can be used. The IOAREA1 keyword specifies PRTOUTA, a name defined in the data definition in line 310, and BLKSIZE (block size) specifies 132 because the output area consists of 132 bytes to accommodate the 132 print positions of the printer. PRINTOV=YES indicates that forms overflow will be accomplished by using the instruction code PRTOV, and WORKA=YES indicates that a work area will be used in addition to an output area. You will soon see how these words coordinate with the data definitions and instructions.

Priority	Keyword	Programmer Code	Default	Remarks
Required	DEVADDR	SYSnnn		SYSIPT is the usual code for a card reader, SYSPCH for a card punch.
Required	IOAREA1	Name of I/O area		
Optional	IOAREA2	Name of second I/O area		Use for overlap; WORKA must also be coded.
Optional	WORKA	YES		YES if second I/O area is specified; a work area can also be used with a single I/O area. GET or PUT names work area.
Optional	DEVICE	1442 2501 2520 2540	2540	Usually omitted if 2540.
Optional	EOFADDR	Label of first instruction of EOF routine		Required for input file.
Optional	TYPEFLE	INPUT OUTPUT	INPUT	Usually omitted for input. Must be OUTPUT for punching.

FIGURE 3-5 DTFCD Operand Summary

One other point about writing DTFs: they are often written with one operand per coding line. For example, the DTFCD in figure 3-4 could be written as:

```
CARDIN   DTFCD DEVADDR=SYSIPT,                    X
               IOAREA1=CRDINPA,                   X
               EOFADDR=CRDEOF
```

The critical requirements for coding DTFs in this way are (1) that the operands be separated only after commas, (2) that some continuation character be punched in column 72 on the card preceding a continuation line, and (3) that all continuation lines start in column 16. Whether you code one operand per line or as many operands as possible per line is a matter of personal preference, but it is easier to change a DTF if one operand is coded on each line.

Priority	Keyword	Programmer Code	Default	Remarks
Required	DEVADDR	SYSnnn		SYSLST is most common.
Required	IOAREA1	Name of I/O area for print lines		
Optional	BLKSIZE	Length of output lines	121	For 1403, usually 132 or 133 for control character use.
Optional	CONTROL	YES		Required if CNTRL macro is used; otherwise omit. If used, omit CTLCHR.
Optional	CTLCHR	ASA YES		Required if line control characters are used. Usual code is ASA; requires ASA line control character as first in print line. YES requires 360 line control character. If used, omit CONTROL.
Optional	DEVICE	1403 1443 3211	1403	Usually omitted for 1403.
Optional	IOAREA2	Name of second I/O area		Use for overlap; WORKA must also be coded.
Optional	WORKA	YES		Required if second I/O area used. PUT names work area. Can also be used for single I/O area.
Optional	PRINTOV	YES		Required if PRTOV macro will be used. Otherwise omit.

FIGURE 3-6 DTFPR Operand Summary

DATA DEFINITIONS

Data definitions are used to give symbolic names to fields in storage. These symbolic names can then be used as operands in the instructions of the program, thus relieving the programmer from the responsibility of keeping track of the addresses and lengths of the fields to be operated upon. In general, the programmer writes data definitions for the I/O areas and for any other areas or fields required by the program. The latter are called *work areas* or *work fields*, which means that they are not an I/O area or any part of one.

The data definitions for the card-input area (lines 340-420) in figure 3-4 follow:

```
CRDINPA  DS    0CL80
CRDITNBR DS    CL5
CRDITDES DS    CL20
         DS    CL5
CRDPRICE DS    CL5
CRDORDPT DS    CL5
CRDONHND DS    CL5
CRDONORD DS    CL5
         DS    CL30
```

For all of the coding lines, the operation code DS (which stands for Define Storage) is used. In columns 1-8 of the statements, names are given by the programmer. These names must (1) start with a letter, (2) consist entirely of letters and numbers, and (3) be eight characters or less in length. The first coding line assigns the name CRDINPA to the entire card-input area and the eight coding lines that follow define fields within this 80-byte area. The name for the card-input area must be the same as the name given in the DTFCD that follows the keyword IOAREA1. When deciding on names, you should try to make them as easy as possible to remember and use. Thus, I have given the name CRDINPA to the card (CRD) input (INP) area (A) and the name CRDITNBR to the card-item-number field.

The operand for each DS statement has three parts: (1) a *duplication factor*, (2) a *type code*, and (3) a *length modifier*. In the statement

```
CRDINPA  DS    0CL80
```

the duplication factor is 0, the type code is C for EBCDIC (or character), and the length modifier is L80 indicating a length of 80 bytes.

A duplication factor of zero simply means that the fields that follow are within the area being defined. Since CRDINPA

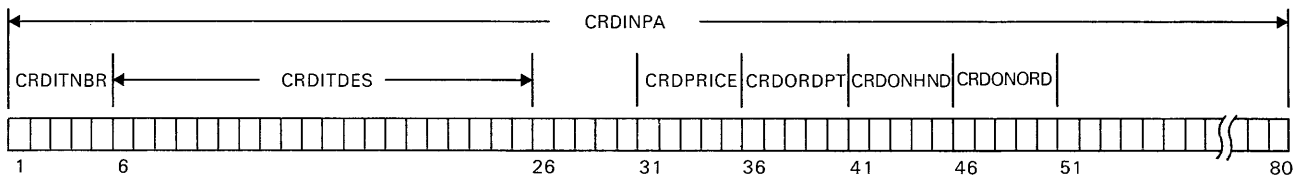


FIGURE 3-7 Card Input Area

has a length of 80, the zero duplication factor means that the DS statements for the next 80 bytes are within the card-input area. Thus, CRDITNBR, CRDITDES, and so on, are within the input area.

When the duplication factor is omitted, it is assumed to be 1. Thus,

```
CRDITNBR DS    CL5
```

defines a field named CRDITNBR that is five bytes long, should receive EBCDIC data, and is bytes 1-5 of the area named CRDINPA. Similarly, the remaining fields in the input area are defined so that they correspond to the fields in the input cards. Since columns 26-30 of the card (the unit-cost field) aren't used by the program, bytes 26-30 of the input area aren't named. Since card columns 51-80 aren't used, the last DS statement within the card-input area is coded without a name and with a length modifier of 30. Figure 3-7 illustrates both the names assigned by the data definitions and their relationship to the input card.

The printer-output area is defined in line 440 as:

```
PRTOUTA  DS    CL132
```

Since the duplication factor is omitted, it is assumed to be one. Since the printer has 132 print positions, the length modifier is L132. The name given to the output area must be the same as the one specified by the IOAREA1 operand in the DTFPR. No fields within the printer area are defined, because a printer work area is used (WORKA=YES). This work area will be moved to the output area prior to printing.

The data definitions of the printer work area (lines 460-560) are:

```
PRTDETL  DS    0CL132
PRTITNBR DS    CL6
         DC    5C' '
PRTITDES DS    CL20
         DC    4C' '
PRTPRICE DS    CL7
         DC    4C' '
PRTAVAIL DS    CL6
         DC    4C' '
PRTORDPT DS    CL6
         DC    70C' '
```

These definitions use the operation code DS as well as the operation code DC, which stands for Define Constant. The DC statement is coded in the same manner as a DS except that a value is also given to the field. This value is given between apostrophes (or single quotation marks) following the length modifier.

Because PRTDETL has a duplication factor of zero, it indicates that the 132 bytes defined by the following DS and DC statements are within this work area. Thus, PRTITNBR is the name for the first 6 bytes of the printer work area. This field is given a length of 6 bytes rather than the 5 bytes

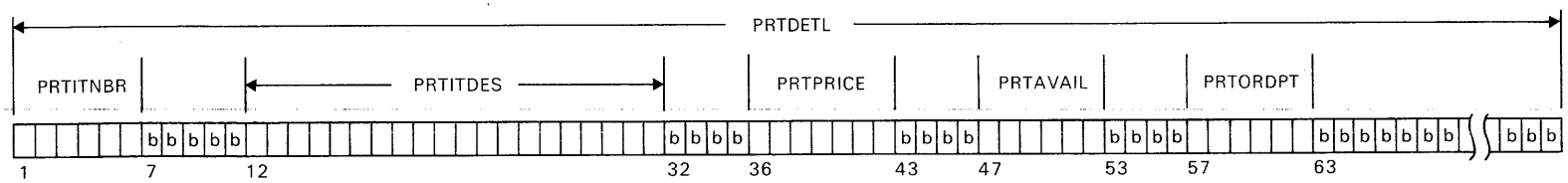


FIGURE 3-8 Printer Work Area

indicated by the print chart because the edit pattern for item number requires 6 bytes.

The five bytes following PRTITNBR are defined by this DC statement:

```
DC    5C' '
```

Here, no length modifier is given so the length is derived from the *nominal value*—that is, the value between the apostrophes. Since one blank is coded between the apostrophes, the length factor is assumed to be one. Since the duplication factor is five, this one-byte blank is duplicated five times, so there will be five blanks between the item-number and item-description fields. This, of course, corresponds to the spacing indicated by the print chart. An alternate way of writing this operand would be C'bbbbb' (where b is one blank), which means a duplication factor of one and a length factor of five taken from the nominal value.

The remaining DS and DC statements in the printer work area are defined in a similar manner. Thus, PRTITDES is a 20-byte field followed by four blanks. Then, PRTPRICE is a 7-byte field followed by four blanks; and so on. The PRTPRICE, PRTAVAIL, and PRTORDPT fields are one byte longer than indicated by the print chart so that the edit patterns for these fields will fit properly. The last DC for this area has an operand of 70C'b' where b is one blank so 70 blanks fill out the area. In figure 3-8, the names assigned to this printer work area are shown schematically.

The last series of data definitions (lines 590-640) are for various work fields used by the program. They use the type codes X for hexadecimal and P for packed-decimal. For example,

```
PATTERN1 DC    X'402020202020'
```

defines a constant named PATTERN1 with a length of six containing the hex codes

```
40 20 20 20 20 20
```

When the type code X is used, two hex characters in the nominal value represent one byte of storage.

Similarly,

```
WRKAVAIL DS    PL3
```

defines a three-byte field named WRKAVAIL that should receive packed-decimal data. If a DC is used for a packed field, the nominal value is a decimal number with or without a leading plus or minus sign. The numeric value is then placed in the field in the appropriate packed form.

One type of code not shown that you may find useful is Z, which stands for zoned-decimal. It too takes a decimal number, with or without a sign, for a nominal value in a DC statement. And the numeric value is stored in the field in zoned-decimal form.

Although these examples should give you a good idea of how to code DS and DC statements, there are some other points you should become familiar with. First, you can give an implied length to a DS statement by using a nominal value just as you can with a DC statement. Thus,

```
CRDITNBR DS    C'99999'
```

defines a five-byte field named CRDITNBR. This is sometimes done deliberately to show how large a number a field can hold, but more often it results from using a DS code by mistake when a DC was intended. You should realize that the DS statement is assembled properly, but no value is placed in the field.

Second, a zero duplication factor can be used to define subfields within fields as well as fields within areas. Thus,

```
DATE     DS    0CL6
MONTH    DS    CL2
DAY      DS    CL2
YEAR     DS    CL2
```

defines two-byte month, day, and year fields within a six-byte date field. The date field in turn could be part of an input area, so that as many levels of subdefinitions as are needed can be coded in assembler language. This is explained more fully in chapter 5.

Third, you should be aware that *padding* and *truncation* can occur in a DC statement. For zoned, packed, or hex fields, padding occurs to the left with zero values as in these examples:

SOURCE CODE		RESULTING FIELD (HEX)
DC	Z'15'	F1 F5
DC	ZL4'15'	F0 F0 F1 F5
DC	P'15'	01 5C
DC	PL3'315'	00 31 5C
DC	XL3'0F'	00 00 0F

For character fields, padding occurs to the right with blanks, as in these examples:

SOURCE CODE		RESULTING FIELD (HEX)
DC	C'A'	C1
DC	CL3'A'	C1 40 40
DC	CL6'NAME'	D5 C1 D4 C5 40 40
DC	CL5' '	40 40 40 40 40

Similarly, truncation occurs on the left for zoned, packed, or hex fields, and on the right for character fields, as in these examples:

SOURCE CODE		RESULTING FIELD
DC	CL3'NAME'	D5 C1 D4
DC	CL1'YES'	E8
DC	ZL2'-132'	F3 D2
DC	PL1'+12'	2C
DC	PL3'400382'	00 38 2C
DC	XL2'140FB12C'	B1 2C

Although padding is used intentionally by BAL programmers, truncation usually results from a programming error. As a result, you wouldn't code a DC statement for truncation, but you should know what will happen if you do it by error.

Note in these examples that for X or C type codes the hex or character codes in the nominal value are the codes placed in the field at the time of program execution. In contrast, for P or Z type codes the numeric value expressed in the nominal value is placed in the field in the appropriate data format.

The last point concerning DS and DC statements is that, in order to code a character (C) constant containing an ampersand (&) or apostrophe ('), two ampersands or apostrophes must be given in the nominal value. For instance,

```
DC    C'A''B'
```

defines a three-byte field containing the characters A'B. The DC statement

```
DC    CL5'S && R'
```

defines a five-byte field containing S & R.

INSTRUCTIONS

The instructions of this program are found in the first 27 lines. They correspond to the program flowchart in figure 3-2. As a programmer codes instructions, he refers to the flowchart for the logic flow and uses the names defined in the file and data definitions as instruction operands.

Housekeeping

The first four instructions of the program represent the *housekeeping* block of the flowchart. In almost all programs, there is some housekeeping or setup (sometimes called *initialization*) required before the actual processing of the program begins. The housekeeping instructions in figure 3-4 are:

```
REORDLST START 0
BEGIN    BALR  3,0
         USING *,3
         OPEN  CARDIN,PRTOUT
```

START The first instruction does not cause any object code to be assembled. It is one of several *assembler instructions* (also known as *assembler commands*) that the programmer uses to control the assembly of his program. The START instruction signals the assembler that the source code is starting and tells the assembler the address at which the object code should start. In almost all cases, the object program will be loaded at a different address than the one specified, so zero is the easiest value to use. (Remember that an object program is link-edited before being executed and the linkage editor determines the actual starting address of the program.)

The name given for the START instruction becomes the *program name*—in this case, REORDLST. A name for the START instruction, or any other instruction, is formed by using the same rules as those used for field names: (1) a name must start with a letter; (2) it must use only letters and numbers; (3) it must be eight characters or less in length.

BALR The branch-and-link-register instruction is the first *machine instruction* (also known as *imperative instruction*) of the reorder-listing program. When it is assembled, one machine-language instruction will result from it.

The BALR instruction has two operands: 3 and 0. When two or more operands are written, they must be separated by commas with no intervening blanks. Unless otherwise indicated, all values specified in assembler language are decimal rather than hexadecimal.

Remember, in chapter 2 you learned that the BALR instruction causes the address of the next instruction to be placed in operand-1 (in this case, register 3). If the register specified by operand-2 is 0 (as in this case), no branching takes place and the program continues with the next instruction. If, for example, this program is loaded starting at address 5200, the BALR instruction will occupy bytes 5200–5201. Then, when the BALR is executed, address 5202 will be stored in register 3 and the program will continue with the next instruction.

The purpose of this BALR instruction is to store a base address in a base register. Thus, register 3 will be used as the base register for this program. If a program is loaded into storage and a proper address is not loaded into the base register used by the program, the program will not execute as intended. As a result, a BALR instruction with zero for operand-2 is normally the first machine instruction of a program.

USING A USING instruction, or statement, is another assembler command since it generates no object code. It tells the assembler program which register is going to be used as the base register. The first operand (in this case, *) tells the assembler at which point in the program the base register should start being used; the second operand (in this case, 3) specifies the register to be used. Since * means "at this point" or "starting now," the USING statement of the reorder-listing program tells the assembler to use register 3 as the base register for all addresses following the USING statement. Since the preceding BALR instruction has already stored the address of the next instruction in register 3, the USING statement and BALR instruction are properly coordinated.

The BALR and USING instructions, coded as shown, are the standard method of loading and specifying the base register. Therefore, you will have similar instructions at the start of each of your programs. Since registers 0, 1, 2, 13, 14, and 15 are used by various programs of the Disk Operating

System, these registers are generally not used as base registers. Instead, registers 3-12 are used. More specifically, it is a common practice to use registers 3, 4, and 5 for base registers and registers 6-12 for other register operations. As a standard procedure, then, you may want to use register 3 as the first base register of your programs.

A single base register can accommodate a program up to 4096 bytes long since the maximum displacement value is 4095 bytes. If a program is larger than this, additional base registers must be loaded and assigned. The coding required to load and specify multiple base registers is explained in chapter 11.

OPEN The OPEN instruction is a *macro instruction* (or just *macro*), which means that more than one object instruction is assembled from it. (Actually, the assembler first converts the macro into two or more source instructions, and then converts the source instructions into object instructions.) In contrast, only one object instruction is assembled from each machine (or imperative) instruction.

An OPEN statement must be executed before an input file can be read or an output file written. You can think of opening a file as checking to be sure the device is ready to operate. The operands of the OPEN statement are the filenames given each input file by the DTFs. For the reorder-listing program, the operands are CARDIN for the DTFCD and PRTOUT for the DTFPR. Although this OPEN statement opens only two files, there is no limit to the number of files that can be opened in a single statement.

The Mainline Routine

The *mainline routine* is the part of a program that accomplishes the main processing of the program. It is usually a loop that includes input, processing, and output. In the flowchart in figure 3-2, blocks 2 through 6, including branching back to connector circle 1, make up the mainline routine.

The first instruction of the mainline routine is:

```
READCARD GET    CARDIN
```

The GET instruction is a macro instruction that causes a record to be read into storage. In this case the operand is CARDIN, so one punched card is read. Since the DTFCD specifies no work area, and since IOAREA1=CRDINPA, the card data is read into the 80-byte field named CRDINPA.

The GET statement, then, corresponds to block 2 (READ CARD) of the flowchart. Because the DTF for the card reader specifies EOFADDR=CRDEOF, the GET instruction will cause a branch to the instruction named CRDEOF when a /* card is read. The flowline out of the card-input block represents this branching. As you can see in the listing in figure 3-4, CRDEOF is the name given to the instruction coded in line 260.

The second block of the mainline routine (block 3 of the flowchart) indicates that the available inventory amount should be calculated. The program does this as follows:

```
PACK  WRKAVAIL,CRDONHND
PACK  WRKONORD,CRDONORD
AP    WRKAVAIL,WRKONORD
```

In other words, the input field named CRDONHND is packed into the work field named WRKAVAIL; the input field named CRDONORD is packed into the work field named WRKONORD; and WRKONORD is added to WRKAVAIL (which then contains the value of the on-hand field). After the three instructions are executed, WRKAVAIL contains the available amount (on-hand plus on-order). You should notice that the names used as operands in these instructions are identical to the names given in the data definitions.

The third block of the mainline routine (block 4) is a decision block testing whether the available amount is less than the reorder point. Its instructions are:

```
PACK  WRKORDPT,CRDORDPT
CP    WRKAVAIL,WRKORDPT
BNL   READCARD
```

Because the compare-decimal (CP) instruction is used to compare available and reorder point, the input-reorder-point field (CRDORDPT) is packed before comparison. Then,

Classification	Operation Code	Meaning
Unconditional	B	Branch unconditionally
After Compare Instructions	BH	Branch on A High
	BL	Branch on A Low
	BE	Branch on A Equal B
	BNH	Branch on A Not High
	BNL	Branch on A Not Low
	BNE	Branch on A Not Equal B
After Arithmetic Instructions	BO	Branch on Overflow
	BP	Branch on Plus
	BM	Branch on Minus
	BZ	Branch on Zero
	BNP	Branch on Not Plus
	BNM	Branch on Not Minus
	BNZ	Branch on Not Zero

FIGURE 3-9 Branch Operation Codes

WRKAVAIL and WRKORDPT are compared and the condition code is set based on this comparison.

The third instruction of this decision block is a branch instruction. Rather than specifying a specific mask for branching, the assembler-language programmer uses a distinct operation code for each type of branch. These codes are summarized in figure 3-9. Thus, the operation code B specifies an unconditional branch; the operation code BE, following a compare instruction, specifies a branch when operand-1 equals operand-2; and BP, following an arithmetic operation, specifies a branch if the result is positive. In the reorder-listing program,

```
BNL     READCARD
```

means that the program should branch to the statement named READCARD if WRKAVAIL is not less than WRKORDPT. If WRKAVAIL is less than WRKORDPT, the program continues with the next instruction.

This branching result could also be coded this way:

```
BH      READCARD
BE      READCARD
```

If available is greater than reorder point, the first branch takes place. If not, the second branch instruction is executed. Then, if available is equal to reorder point, the branch to READCARD takes place; if not, the next instruction in sequence is executed. (You should realize that all of the operation codes in figure 3-9 cause one branch-on-condition instruction to be assembled; only the mask differs, depending on the operation code.)

Incidentally, the branch-on-condition instruction can also be coded by using a decimal number that represents the mask. For instance, this instruction

```
BC      15,READCARD
```

gives a mask of decimal 15 (hex F or binary 1111) so it is an unconditional branch to the instruction named READCARD. Similarly, if operand 1 is an 8 (hex 8 or binary 1000), the branch only takes place if the first bit in the condition code is on. Following a compare decimal instruction, then,

```
BC      8,READCARD
```

has the same effect as

```
BE      READCARD
```

By using a decimal number from 0 through 15, any mask can be coded so the instruction can branch on any combination of condition codes. This form of coding is rarely used, however, since codes like BE, BH, and BNL are so much easier to use.

The fourth block in the mainline routine (block 5) says "construct detail print line." This means arrange the data in the printer work area in a form suitable for printing. This normally involves moving alphanumeric fields into the output work area and editing numeric fields into the output work area so that lead zeros are suppressed and commas and decimal points are inserted wherever needed.

Lines 120–220 represent this block of processing. For instance, the first three instructions following the branch instruction edit the input-item-number field into the output work area:

```
PACK   PACKAREA,CRDITNBR
MVC    PRTITNBR,PATTERN1
ED     PRTITNBR,PACKAREA
```

In the first instruction, the five-byte input field is packed into the three-byte work field named PACKAREA. Then, the hex pattern 402020202020 is moved into the first six bytes of the printer work area. Finally, the packed item number is edited into the pattern thus suppressing lead zeros.

The next instruction moves the input-item-description field, unchanged, into the output work area. Then, the next seven instructions edit the unit price, available, and reorder point fields. Since available and reorder point are already in packed form, they do not need to be packed before editing. In all cases, an edit pattern is moved into the appropriate output field before the edit instruction is executed.

The last block of the mainline routine (block 6) is the output block. It indicates that a line should be printed and, if necessary, forms overflow should take place. The instructions representing this block are:

```
PUT    PRTOUT,PRTDETL
PRTOV  PRTOUT,12
```

The PUT instruction is a macro instruction that causes output. Because the device is a printer, one line is printed each time the PUT is executed, and single spacing takes place. Since the DTFPR specifies WORKA=YES, two operands are required for the PUT. The first operand gives the filename as defined by the DTFPR (PRTOUT); the second operand gives the name of the work area used (in this case, PRTDETL). When the instruction is executed, the data in the work area is moved to the output area (IOAREA1= PRTOUTA) and then printed.

The PRTOV instruction that follows is a macro instruction that accomplishes forms overflow. It can be used because PRINTOV=YES has been specified in the DTFPR. The first operand gives the filename for the printer; the second operand gives the number of the carriage-control channel that indicates the last printing line (usually, 12). When channel 12 is sensed by the printer, an automatic skip to channel 1 takes place.

The final instruction of the mainline routine is:

```
B      READCARD
```

This means an unconditional branch to the instruction named READCARD. Thus, the program then repeats the processing for the next card in the input deck.

The End-of-Job Routine

When the end of input is detected, most programs branch to instructions that make up an *end-of-job routine*, or *EOJ routine*. This routine may or may not cause final totals to print but, at the least, it does close the files and signal the supervisor that the next program in the job deck can be executed.

The end-of-job routine for the reorder-listing program is:

```
CRDEOF  CLOSE CARDIN,PRTOUT
        EOJ
```

Since the name CRDEOF is given for the keyword EOFADDR in the DTFCD, these instructions are executed when a /* card is read.

The CLOSE statement parallels the OPEN statement. It is a macro instruction that closes the files to further processing, and its operands are the filenames for the input and output files taken from the DTFs.

The final macro instruction is operation code EOJ with no operands. It signals the supervisor to load and execute the next program.

THE END STATEMENT

The last statement in a BAL source deck must always be the END statement. It, like the START statement, is an assembler

command. It tells the assembler that there are no more source cards and its operand tells the assembler where program execution should begin. In most cases, the operand is the name of the BALR instruction that loads the base register at the start of the program since this is the first executable instruction of the program. Thus, the END statement for the reorder-listing program (line 650) specifies BEGIN as the operand—that is, the name of the BALR instruction.

SUMMARY

As you can see from this example, an assembler-language programmer actually writes three types of instructions: assembler commands, machine instructions, and macro instructions. In addition, he writes file definitions and data definitions. He gives names to all DTFs and to the data fields and instructions that are referred to by other instructions. These names are then used as operands in the instructions that refer to them.

Although this example gives the instructions first, then the file definitions, and, finally, the data definitions, the assembler does not require any specific order. It is also common practice to have the file definitions first, the instructions second, and the data definitions last. For the sake of efficiency, it is best to use the order shown in the reorder-listing program or the common variation just described.

By comparing assembler-language code with the actual machine instructions presented in chapter 2, you can see how assembler language aids the programmer. First, by using symbolic names for files, data, and instructions, the programmer doen't have to keep track of addresses and length codes. Second, by using mnemonic operation codes, the programmer doesn't have to remember hex machine-operation codes. Third, by using macros, many machine instructions can be assembled from a single line of BAL coding. In particular, remember the difficulty I mentioned in coding machine-language I/O instructions—the use of channel commands and so forth. By defining the file with a DTF and opening the file with an OPEN, the programmer can thereafter use a GET macro for any input operation and a PUT macro for any output operation.

Terminology

subset
label
name
operation
operand
80-80 listing
comment card
comment
file definition
DTF
filename
keyword
data definition
work area
work field
duplication factor
type code
length modifier
nominal value
padding
truncation
housekeeping
initialization
assembler instruction
assembler command
program name
machine instruction
imperative instruction
macro instruction
macro
mainline routine
end-of-job routine
EOJ routine

Objectives

1 Given the program listing in figure 3-4 and samples of input data, describe the result of any instruction or any group of instructions in the program.
2 Given the description of a storage field or area, write acceptable data definitions for it using C, Z, P, or X type codes. (For this and all other problems requiring coding, you should use BAL coding sheets.)

Problems

1 (Objective 1) The data that follow represent five input cards for the reorder-listing program shown in figure 3-4:

CARD COLUMNS	1-5	6-25	26-30	31-35	36-40	41-45	46-50
CARD 1	00101	GENERATOR	04000	04900	00100	00070	00050
	00103	HEATER SOLENOID	00330	00440	00050	00034	00000
	03244	GEAR HOUSING	06500	07900	00010	00012	00000
	03981	PLUMB LINE	00210	00240	00015	00035	00000
CARD 5	/*						

Assuming columns 51-80 of all five cards are blank and that the object program is loaded at decimal address 8000, answer the following questions.

a. What is the first object instruction to be executed by the program and what is the result of executing that instruction?
b. After the GET instruction has been executed for card 1, what data, in hex, do CRDPRICE and CRDONORD contain?
c. After the instructions in lines 060 and 070 have been executed, what data, in hex, do WRKAVAIL and WRKONORD contain?
d. After the AP instruction in line 080 has been executed, what data, in hex, does WRKAVAIL contain?
e. After the PACK instruction in line 090 has been executed, what data, in hex, does WRKORDPT contain?
f. What is the line number of the next instruction to be executed after the BNL instruction in line 110 for card 1 has been executed?
g. After the GET instruction has been executed for card 2, what data, in hex, do CRDPRICE and CRDONORD contain?
h. After the AP and PACK instructions in lines 080 and 090 have been executed, what, in hex, do WRKAVAIL and WRKORDPT contain?
i. What is the line number of the next instruction to be executed after the BNL instruction in line 110 has been executed for card 2?
j. What, in hex, do PATTERN1 and PRTITNBR contain after the MVC instruction in line 130 has been executed?
k. What, in hex, does PRTITNBR contain after the ED instruction in line 140 has been executed? How will this data print?
l. When the PUT instruction is executed for card 2, what will print in the first 42 print positions?
m. What is the name of the printer I/O area, the name of the printer work area, and the name of the printer file?
n. Assume that a line has been printed and no 12-channel punch in the carriage-control tape has been sensed. What happens when the PRTOV macro is executed?
o. If channel 12 is sensed when the PUT is executed, what happens when the PRTOV macro is executed?
p. What happens when the branch instruction in line 250 is executed?
q. When the GET instruction for card 5 is executed, what happens?
r. What happens when the EOJ macro is executed?

2 (Objective 2) Write the data definitions for the card input area that has these fields:

CARD COLUMNS	FIELD
1	Card Code
2-4	Salesman Number
5-9	Customer Number
15-19	Invoice Number
20-25	Invoice Date
26-47	Customer Name
48-54	Invoice Amount

3 (Objective 2) Write data definitions for the following:

a. A six-byte zoned-decimal field
b. Twenty bytes of storage containing blanks
c. A five-byte packed-decimal field containing a value of minus 5

d. A five-byte zoned-decimal field containing a value of plus 390
e. A two-byte binary field containing a value of decimal 12
f. A seven-byte EBCDIC field containing O'CLOCK
g. Six fields of seven bytes each that are to receive zoned-decimal data

Solutions

1 a. BALR is the instruction. After it is executed, 8002 is stored in register 3 and the program continues with the OPEN instruction.
b. F0 F4 F9 F0 F0 and F0 F0 F0 F5 F0
c. 00 07 0C and 00 05 0C
d. 00 12 0C
e. 00 10 0C
f. Line 050, the GET instruction
g. F0 F0 F4 F4 F0 and F0 F0 F0 F0 F0; in other words, the data for card 1 is replaced by the data for card 2.
h. 00-03 4C and 00 05 0C
i. Line 120
j. 40 20 20 20 20 20 and 40 20 20 20 20 20
k. Hex 40 40 40 F1 F0 F3 which will print as 103.
l. `103 HEATER SOLENOID 4.40`
m. PRTOUTA, PRTDETL, and PRTOUT
n. Nothing. The program continues with the next instruction.
o. A skip to channel 1 occurs and the program continues with the next instruction in the sequence.
p. A branch to line 050 takes place.
q. The card data is read into CRDINPA, the /* in columns 1 and 2 is detected, and the program branches to the CLOSE instruction in line 260.
r. The program ends and a branch to the supervisor takes place.

2
```
CDINPUTA DS    0CL80
CDCODE   DS    CL1
CDSLSNO  DS    CL3
CDCSTNO  DS    CL5
         DS    CL5
CDINVNO  DS    CL5
CDINVDTE DS    CL6
CDCSTNME DS    CL22
CDINVAMT DS    CL7
         DS    CL26
```

3 a.
```
         DS    ZL6
```
b.
```
         DC    CL20'b' where b is a blank
or       DC    20C'b'
or       DC    C'bbbbbbbbbbbbbbbbbbbb'
```
c.
```
         DC    PL5'-5'
```
d.
```
         DC    ZL5'390'
or       DC    ZL5'+390'
```
e.
```
         DC    X'000C'
or       DC    XL2'0C'
```
f.
```
         DC    CL7'O''CLOCK'
```
g.
```
         DS    6ZL7
```

TOPIC TWO Refining the Reorder-Listing Program Because it is the first program presented, the reorder-listing program in topic 1 has been simplified. In actual practice, a program such as this would print headings at the top of each page of the report and would likely print a count of the number of records processed at the end of the report. This count would be used to make sure that all inventory records were processed—that is, no cards were missing from the input deck. These additional printing lines are shown on the print chart in figure 3-10.

A flowchart for this refined reorder-listing program is shown in figure 3-11 and the BAL coding is shown in figure 3-12. In addition to the extra printing on the report, the refined program overlaps card reading and printing with processing. Such overlap would normally be required of a program.

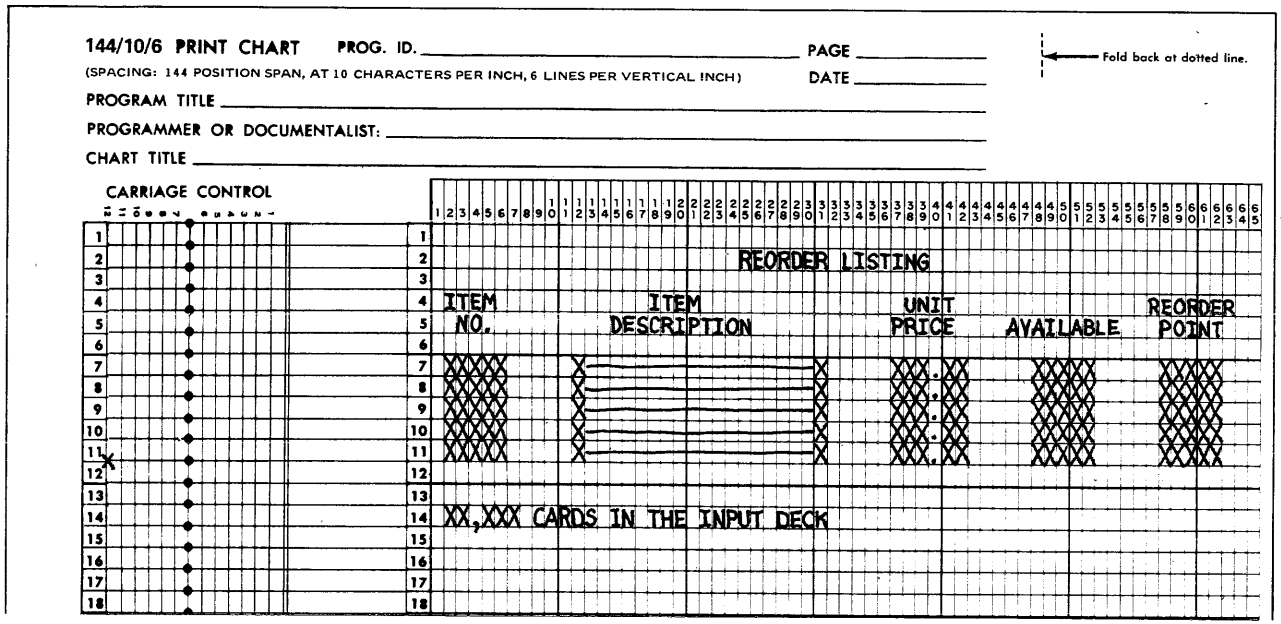


FIGURE 3-10 Refined Reorder-Listing Print Chart

REMARKS

In addition to comment cards, the program listing in figure 3-12 uses *remarks* to make the program easier to follow. Remarks are programmer notes following the operands of an instruction; they do not affect the object code in any way. There must be one or more blanks between the last operand of the instruction and the first character of the remark, and the remark must not go beyond column 71 of the source cards. It is common to code all remarks starting in the same column such as column 40 or 45. By using remarks, a programmer documents his program as he prepares it.

OVERLAP

The only changes required for overlap are in the DTFs for the card and printer, and in the data definitions for the associated I/O areas. Since two I/O areas are required for each file, the keywords IOAREA1 and IOAREA2 are used for both card and printer files. Two I/O areas are then defined for each file in the data definitions section of the program. For the card reader, they are CRDIO1 and CRDIO2, two 80-byte areas; for the printer, they are PRTIO1 and PRTIO2, two 132-byte areas.

When card or print operations are overlapped with processing, a work area is generally used. After a card is read into either I/O area, the data is moved to the work area. Before data is printed from either printer I/O area, the data is moved into the appropriate I/O area from the appropriate work area. Thus, WORKA=YES is specified for both files. When the GET is used, as in line 110 of figure 3-12, it requires two operands: the first gives the filename for the

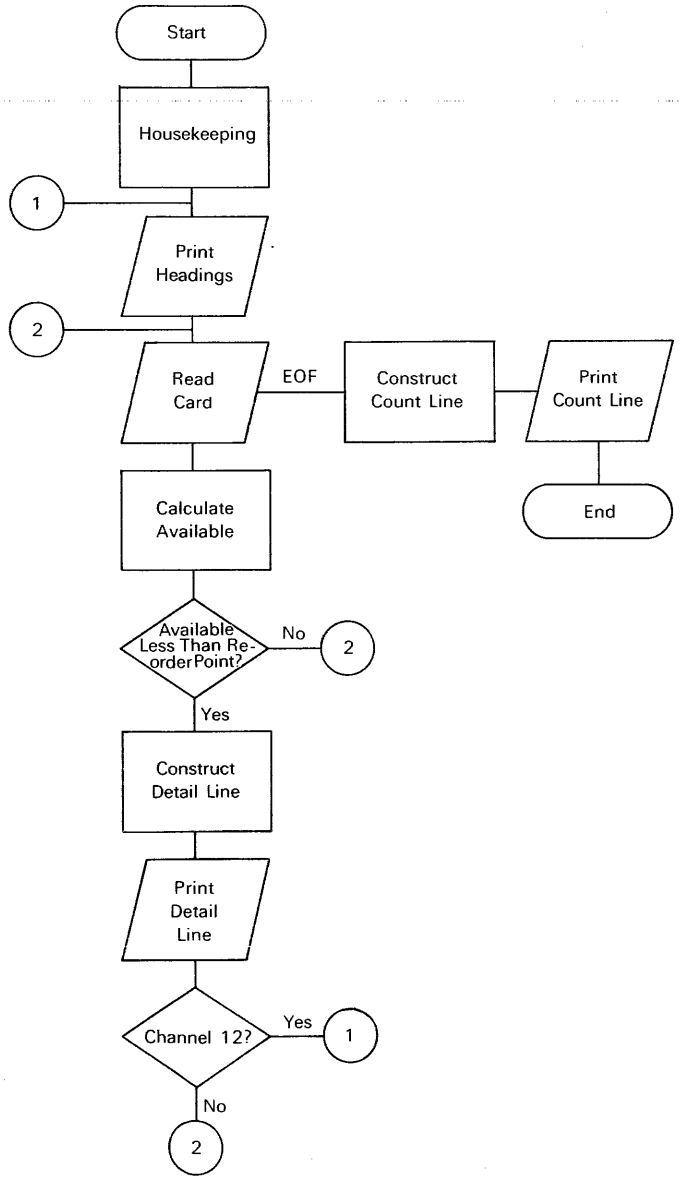


FIGURE 3-11 Refined Reorder-Listing Flowchart

card reader, the second gives the name of the work area used. These are the only changes required for overlap to take place in the program.

SKIPPING AND SPACING

When a PUT macro is executed for the printer, the form is moved one space after printing. If other than single spacing is required, the CNTRL macro can be used. To use it, CONTROL=YES must be specified in the DTFPR.

The CNTRL macro, as used in lines 050, 070, and 350 of the refined reorder-listing program, can have four operands. The first operand gives the filename of the printer and the second operand is either SK for skip or SP for space. If SK is used, the third operand gives a number from 1 through 12 indicating the channel to skip to immediately, and the fourth operand gives the channel to skip to *after* printing. The following example causes a skip to channel 1 immediately, and a skip to channel 3 after printing.

```
CNTRL PRNTFLE,SK,1,3
```

If operand 3 or 4 is omitted, it means no skipping at that time. Thus, line 050 of the refined reorder-listing program causes a skip to channel 1 immediately and no skipping after printing.

```
HEADING  CNTRL PRTOUT,SK,1
```

A CNTRL macro that would cause a skip to channel 1 after printing and have no skipping before would be coded this way:

```
HEADING  CNTRL PRTOUT,SK,,1
```

When SP is used for the second operand, the digits 1, 2, or 3 can be used for the third and fourth operands. The third operand indicates the number of lines to space immediately, the fourth operand specifies the number of lines to space *after* printing. Thus, line 070 indicates: space one line immediately.

```
CNTRL PRTOUT,SP,1
```

When a CNTRL macro indicating spacing after printing is not used before a PUT macro is executed, single spacing after printing automatically takes place.

PRINTING HEADINGS

To print headings, work areas that contain the required heading data are defined. Thus, the three heading lines needed are defined in work areas named HDGLINE1, HDGLINE2, and HDGLINE3. The data to be printed is defined using DCs, and continuation lines for the DCs are used whenever needed. These definitions are given in lines 660 through 770 of figure 3-12.

When PUT macros are issued to print headings, the appropriate work area is given as the second operand. To illustrate, consider the heading routine in figure 3-12:

```
HEADING  CNTRL PRTOUT,SK,1
         PUT   PRTOUT,HDGLINE1
         CNTRL PRTOUT,SP,1
         PUT   PRTOUT,HDGLINE2
         PUT   PRTOUT,HDGLINE3
         CNTRL PRTOUT,SP,1
```

Here, three different work areas are specified as the operands of three successive PUT macros. The CNTRL macro is used as needed to control spacing and skipping.

The data definitions for the constants in HDGLINE2 and HDGLINE3 illustrate how continuation lines are coded. The coding in the first line continues through column 71, a continuation punch is used in column 72, and the coding continues in column 16 of the continuation line. This can be done for any assembler-language coding lines including DTFs. Although DTFs can also be broken any time after a comma, other operand lists cannot be broken. For macro instructions, there can be as many continuation lines as are needed; for other assembler language instructions, there can be a maximum of two continuation lines.

So that the headings are printed on each page of the report, the PRTOV macro specifies a third operand:

```
PRTOV PRTOUT,12,HEADING
```

The third operand (HEADING) gives the name of the instruction to branch to when a channel-12 punch is sensed by the preceding PUT instruction. When a third operand is used, automatic skipping does not take place; instead, the heading routine must cause skipping by using the CNTRL macro.

LITERALS

To count the number of input records read by the refined reorder-listing program, a field named COUNT is defined in line 1030. This is a three-byte packed field with a starting value of zero. Then, after each record is read, the value 1 is added to COUNT using the AP instruction with a *literal* as an operand:

```
AP     COUNT,=P'1'
```

A literal is a value given by the instruction operand itself. Thus, the AP instruction just shown is the equivalent of the instruction

```
AP     COUNT,PCON1
```

and the data definition

```
PCON1  DC     P'1'
```

By using the literal, the amount of coding is reduced.

To code a literal operand, use an equals sign followed by a type code and a nominal value enclosed in single quotes. All of the DC type codes are valid, so any of the following are acceptable literals:

```
=P'-15'
=Z'+5000'
=X'40'
=C'ABC'
```

A literal can be used as the sending field in an instruction, but it cannot be used as the receiving field.

```
REORDLST START 0                                                        ORDR0010
BEGIN    BALR  3,0                    LOAD BASE REGISTER               ORDR0020
         USING *,3                                                     ORDR0030
         OPEN  CARDIN,PRTOUT                                           ORDR0040
HEADING  CNTRL PRTOUT,SK,1            SKIP TO CHANNEL ONE              ORDR0050
         PUT   PRTOUT,HDGLINE1        PRINT FIRST HEADING LINE         ORDR0060
         CNTRL PRTOUT,SP,1            SPACE PRINTER ONE LINE           ORDR0070
         PUT   PRTOUT,HDGLINE2        PRINT SECOND HEADING LINE        ORDR0080
         PUT   PRTOUT,HDGLINE3        PRINT THIRD HEADING LINE         ORDR0090
         CNTRL PRTOUT,SP,1            SPACE PRINTER ONE LINE           ORDR0100
READCARD GET   CARDIN,CRDWRKA         READ CARD INTO WORK AREA         ORDR0110
         AP    COUNT,=P'1'            ADD ONE TO COUNT                 ORDR0120
         PACK  WRKAVAIL,CRDONHND                                       ORDR0130
         PACK  WRKONORD,CRDONORD                                       ORDR0140
         AP    WRKAVAIL,WRKONORD      ADD ON HAND AND ON ORDER         ORDR0150
         PACK  WRKORDPT,CRDORDPT                                       ORDR0160
         CP    WRKAVAIL,WRKORDPT      COMPARE AVAILABLE & ORDER POINT  ORDR0170
         BNL   READCARD                                                ORDR0180
         PACK  PACKAREA,CRDITNBR                                       ORDR0190
         MVC   PRTITNBR,PATTERN1                                       ORDR0200
         ED    PRTITNBR,PACKAREA      EDIT ITEM NUMBER FIELD           ORDR0210
         MVC   PRTITDES,CRDITDES      MOVE ITEM DESCRIPTION            ORDR0220
         PACK  PACKAREA,CRDPRICE                                       ORDR0230
         MVC   PRTPRICE,PATTERN2                                       ORDR0240
         ED    PRTPRICE,PACKAREA      EDIT UNIT PRICE                  ORDR0250
         MVC   PRTAVAIL,PATTERN1                                       ORDR0260
         ED    PRTAVAIL,WRKAVAIL      EDIT AVAILABLE                   ORDR0270
         MVC   PRTORDPT,PATTERN1                                       ORDR0280
         ED    PRTORDPT,WRKORDPT      EDIT ORDER POINT                 ORDR0290
         PUT   PRTOUT,PRTDETL         PRINT DETAIL LINE                ORDR0300
         PRTOV PRTOUT,12,HEADING      IF END OF PAGE, GO TO HEADING    ORDR0310
         B     READCARD               GO TO READCARD                   ORDR0320
CRDEOF   ED    CNTPATRN,COUNT         EDIT COUNT                       ORDR0330
         CNTRL PRTOUT,SP,2            SPACE PRINTER TWO LINES          ORDR0340
         PUT   PRTOUT,CNTLINE         PRINT COUNT LINE                 ORDR0350
         CLOSE CARDIN,PRTOUT                                           ORDR0360
         ECJ                                                           ORDR0370
* THE CARD FILE DEFINITION FOLLOWS                                     ORDR0380
CARDIN   DTFCD DEVADDR=SYSIPT,IOAREA1=CRDIO1,IOAREA2=CRDIO2,          XORDR0390
               EOFADDR=CRDEOF,WORKA=YES                                ORDR0400
* THE PRINTER FILE DEFINITION FOLLOWS                                  ORDR0410
PRTOUT   DTFPR DEVADDR=SYSLST,                                        XORDR0420
               IOAREA1=PRTIO1,                                        XORDR0430
               IOAREA2=PRTIO2,                                        XORDR0440
               WORKA=YES,                                             XORDR0450
               BLKSIZE=132,                                           XORDR0460
               CONTROL=YES,                                           XORDR0470
               PRINTOV=YES                                             ORDR0480
* THE DATA DEFINITIONS FOR THE TWO CARD I/O AREAS FOLLOW               ORDR0490
CRDIO1   DS    CL80                                                   ORDR0500
CRDIO2   DS    CL80                                                   ORDR0510
```

FIGURE 3-12 Refined Reorder-Listing Program

```
* THE DATA DEFINITIONS FOR THE TWO PRINTER AREAS FOLLOW                  ORDR0520
PRTIO1   DS    CL132                                                     ORDR0530
PRTIO2   DS    CL132                                                     ORDR0540
* THE DATA DEFINITIONS FOR THE CARD WORK AREA FOLLOW                     ORDR0550
CRDWRKA  DS    0CL80                                                     ORDR0560
CRDITNBR DS    CL5                                                       ORDR0570
CRDITDES DS    CL20                                                      ORDR0580
         DS    CL5                                                       ORDR0590
CRDPRICE DS    CL5                                                       ORDR0600
CRDORDPT DS    CL5                                                       ORDR0610
CRDONHND DS    CL5                                                       ORDR0620
CRDONORD DS    CL5                                                       ORDR0630
         DS    CL30                                                      ORDR0640
* THE DATA DEFINITIONS FOR THE PRINTER HEADING LINES FOLLOW              ORDR0650
HDGLINE1 DS    0CL132                                                    ORDR0660
         DC    24C' '                                                    ORDR0670
         DC    C'REORDER LISTING'                                        ORDR0680
         DC    93C' '                                                    ORDR0690
HDGLINE2 DS    0CL132                                                    ORDR0700
         DC    C' ITEM                ITEM                  UNIT       XORDR0710
               REORDER'                                                  ORDR0720
         DC    69C' '                                                    ORDR0730
HDGLINE3 DS    0CL132                                                    ORDR0740
         DC    C'  NO.          DESCRIPTION          PRICE    AVAILABLEXORDR0750
               POINT'                                                    ORDR0760
         DC    70C' '                                                    ORDR0770
* THE DATA DEFINITIONS FOR THE PRINTER DETAIL LINE FOLLOW                ORDR0780
PRTDETL  DS    0CL132                                                    ORDR0790
PRTITNBR DS    CL6                                                       ORDR0800
         DC    5C' '                                                     ORDR0810
PRTITDES DS    CL20                                                      ORDR0820
         DC    4C' '                                                     ORDR0830
PRTPRICE DS    CL7                                                       ORDR0840
         DC    4C' '                                                     ORDR0850
PRTAVAIL DS    CL6                                                       ORDR0860
         DC    4C' '                                                     ORDR0870
PRTORDPT DS    CL6                                                       ORDR0880
         DC    70C' '                                                    ORDR0890
* THE DATA DEFINITIONS FOR THE FINAL TOTAL LINE FOLLOW                   ORDR0900
CNTLINE  DS    0CL132                                                    ORDR0910
CNTPATRN DC    X'4020206B202020'                                         ORDR0920
         DC    C' CARDS IN THE INPUT DECK'                               ORDR0930
         DC    101C' '                                                   ORDR0940
* THE DATA DEFINITIONS THAT FOLLOW DEFINE OTHER WORK AREAS NEEDED         ORDR0950
* BY THE PROGRAM                                                         ORDR0960
PATTERN1 DC    X'402020202020'                                           ORDR0970
PATTERN2 DC    X'40202021 4B2020'                                        ORDR0980
WRKAVAIL DS    PL3                                                       ORDR0990
WRKONORD DS    PL3                                                       ORDR1000
WRKORDPT DS    PL3                                                       ORDR1010
PACKAREA DS    PL3                                                       ORDR1020
COUNT    DC    PL3'0'                                                    ORDR1030
         END   BEGIN                                                     ORDR1040
```

FIGURE 3-12 Refined Reorder-Listing Program (Continued)

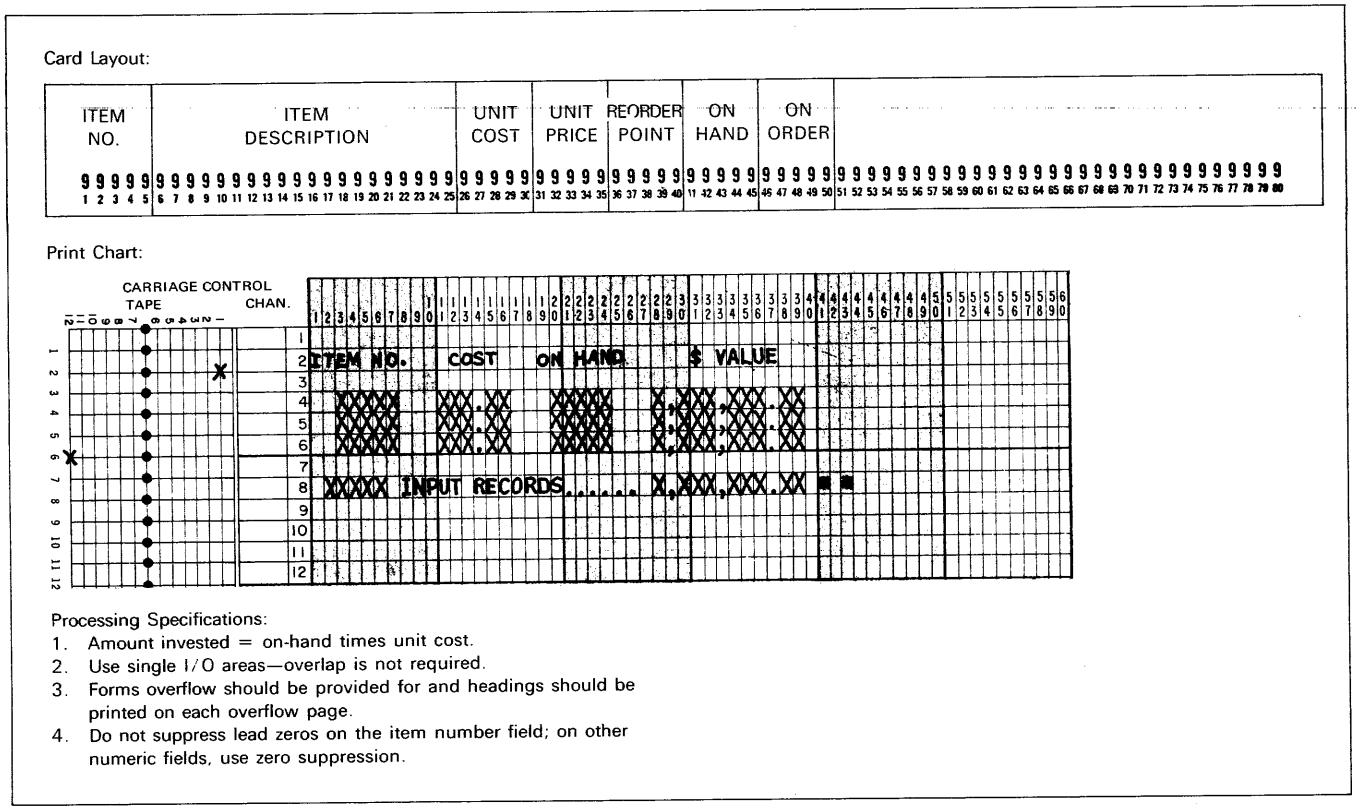


FIGURE 3-13 Investment-Listing Problem

PRINTING THE COUNT

When the end-of-file is reached, the program branches to CRDEOF. The count is then printed using this code:

```
CRDEOF   ED     CNTPATRN,COUNT
         CNTRL  PRTOUT,SP,2
         PUT    PRTOUT,CNTLINE
```

First, COUNT is edited into an edit pattern that is part of the printer work area for the count line. Next, the CNTRL macro moves the form two spaces up. Then, the PUT macro causes the count line to be printed.

CONCLUSION

If you understand this refined reorder-listing program, you should be able to start coding programs of your own. Although the SP, MP, DP, and ZAP haven't been illustrated in a program yet, you should be able to use them because they are analogous to the AP instruction coding. Similarly, you should be able to code the UNPK instruction because it is the reverse of the PACK. (You may want to refer back to chapter 2 to check the operational details of these instructions.)

One problem you might have using the DP instruction is

separating the remainder from the quotient. This can be done by using a zero duplication factor for the entire result field and defining subfields for the quotient and remainder as in these data definitions:

```
DIVIDEND DS    0CL5
DIVQUO   DS    CL3
DIVREM   DS    CL2
```

Then, the quotient can be referred to as DIVQUO, the remainder as DIVREM.

Terminology

remark
literal

Objectives

Given a programming problem, code a program for its solution. The problem will have card input and printer output and may require any functions described in this or previous topics.

Problems

1 Figure 3-13 defines a problem that consists of printing an investment report from a deck of inventory cards like those used for the reorder-listing program. At the end of the report, a total line consisting of a count of the number of input cards and a total of the amounts invested is to be printed.
   a. Draw a program flowchart for this problem.
   b. Code a solution for this problem. (As a guideline, code the DTFs first, the data definitions for I/O areas and related work areas second, and the instructions and other work areas last.)

2 Figure 3-15 is a solution for problem 1b above. Now suppose there is a request for an extra total line that gives

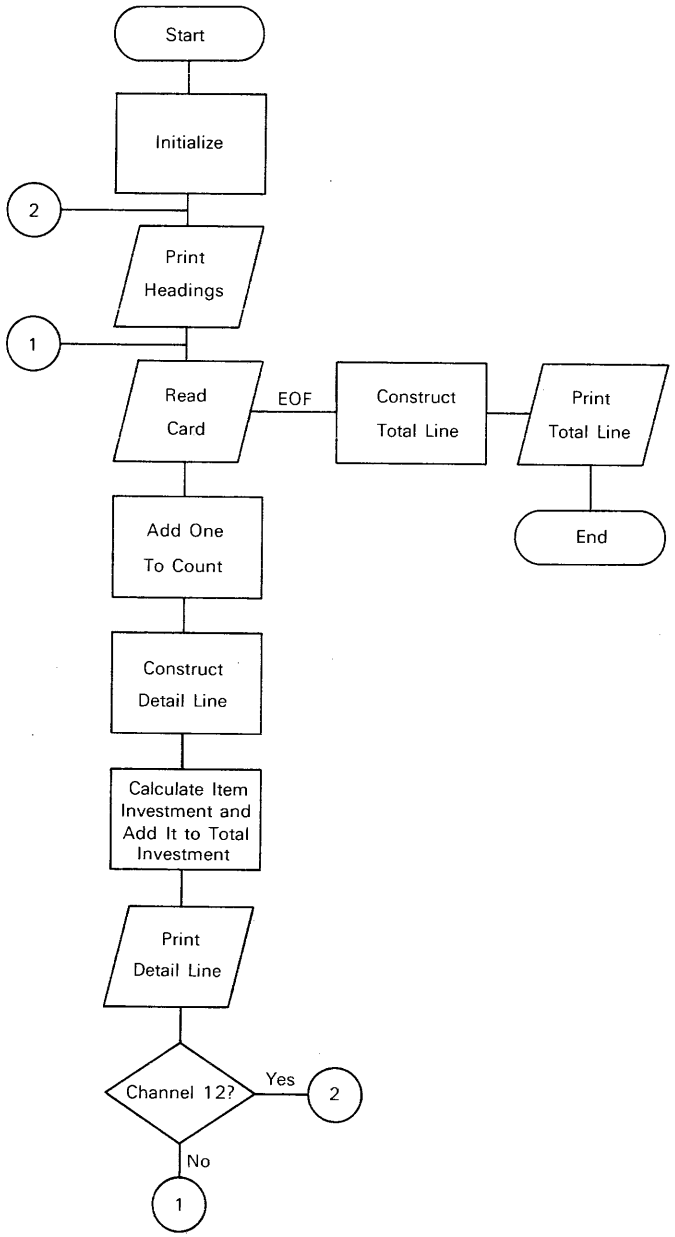


FIGURE 3-14 Investment-Listing Flowchart

```
INVVALUE START 0                                                    INVL0010
BEGIN    BALR  3,0                                                  INVL0020
         USING *,3                  SPECIFY REG 3 AS BASE REGISTER  INVL0030
         OPEN  CARDIN,PRTOUT        OPEN CARD AND PRINTER FILES     INVL0040
HEADING  CNTRL PRTOUT,SK,1          SKIP PRINTER TO TOP OF PAGE     INVL0050
         PUT   PRTOUT,PRTHEAD       PRINT HEADING LINE              INVL0060
         CNTRL PRTOUT,SP,2          SPACE PRINTER 2 LINES           INVL0070
READCARD GET   CARDIN               READ CARD INTO I/O AREA         INVL0080
         AP    COUNT,=P'1'                                          INVL0090
         MVC   PRTITNBR,CRDITNBR    MOVE ITEM NBR INTO PRINT LINE   INVL0100
         PACK  WRKITCST,CRDITCST                                    INVL0110
         MVC   PRTITCST,PATTERN1                                    INVL0120
         ED    PRTITCST,WRKITCST    EDIT ITEM COST                  INVL0130
         PACK  WRKBOH,CRDBOH                                        INVL0140
         MVC   PRTBOH,PATTERN2                                      INVL0150
         ED    PRTBOH,WRKBOH        EDIT ON HAND                    INVL0160
         PACK  WRKITVAL,CRDITCST                                    INVL0170
         MP    WRKITVAL,WRKBOH      MULTIPLY COST AND ON HAND       INVL0180
         AP    TOTVALUE,WRKITVAL    ADD AMOUNT TO TOTAL AMOUNT      INVL0190
         MVC   PRTITVAL,PATTERN3                                    INVL0200
         ED    PRTITVAL,WRKITVAL    EDIT AMOUNT                     INVL0210
         PUT   PRTOUT,PRTDETL       PRINT DETAIL LINE               INVL0220
         PRTOV PRTOUT,12,HEADING    IF END OF PAGE, GO TO HEADING   INVL0230
         B     READCARD                                             INVL0240
EOFRT    CNTRL PRTOUT,SP,2          SPACE PRINTER TWO LINES         INVL0250
         ED    PRTTOTVL,TOTVALUE    EDIT TOTAL AMOUNT               INVL0260
         ED    PRTCOUNT,COUNT       EDIT RECORD COUNT               INVL0270
         PUT   PRTOUT,PRTTOTAL      PRINT TOTAL LINE                INVL0280
         CLOSE CARDIN,PRTOUT        CLOSE FILES                     INVL0290
         EOJ                        END OF JOB MACRO                INVL0300
```

FIGURE 3-15 Investment-Listing Program

the average amount invested in inventory for each item and uses this format:

```
AVERAGE INVESTMENT PER ITEM IS XX,XXX.XX
```

Code the changes that would have to be made to the original program to cause the new total line to print on the line immediately after the original total line.

Solutions

1 a. Figure 3-14 is an acceptable solution.
  b. Figure 3-15 is an acceptable solution.

2 These instructions would be inserted between lines 350 and 360 of the program:

```
         ZAP   PKDIV,TOTVALUE
         DP    PKDIV,COUNT
         ED    PRTAVE,QUOTIENT
         PUT   PRTOUT,PRTAVELN
```

These work area definitions would have to be added to the data definitions:

```
PRTAVELN DS    0CL132
         DC    C'AVERAGE INVESTMENT PER ITEM IS'
PRTAVE   DC    X'4020206B2020214B2020'
         DC    92C' '
ZNDIV    DS    ZL9
PKDIV    DS    0PL8
         DS    PL1
QUOTIENT DS    PL4
         DS    PL3
```

```
* FILE DEFINITIONS                                                    INVL0310
CARDIN   DTFCD DEVADDR=SYSIPT,                                       XINVL0320
               IOAREA1=CRDIPTA,                                      XINVL0330
               EOFADDR=EOFRT                                          INVL0340
PRTOUT   DTFPR DEVADDR=SYSLST,                                       XINVL0350
               IOAREA1=PRTOPTA,                                      XINVL0360
               BLKSIZE=132,                                          XINVL0370
               CONTROL=YES,                                          XINVL0380
               PRINTOV=YES,                                          XINVL0390
               WORKA=YES                                              INVL0400
CRDIPTA  DS    0CL80                                                  INVL0410
CRDITNBR DS    CL5                                                    INVL0420
         DS    CL20                                                   INVL0430
CRDITCST DS    CL5                                                    INVL0440
         DS    CL10                                                   INVL0450
CRDBOH   DS    CL5                                                    INVL0460
         DS    CL35                                                   INVL0470
PRTOPTA  DS    CL132                                                  INVL0480
PRTHEAD  DS    0CL132                                                 INVL0490
         DC    C'ITEM NO.    COST    ON HAND      $ VALUE'            INVL0500
         DC    95C' '                                                 INVL0510
PRTDETL  DS    0CL132                                                 INVL0520
         DC    2C' '                                                  INVL0530
PRTITNBR DS    CL5                                                    INVL0540
         DC    2C' '                                                  INVL0550
PRTITCST DS    CL7                                                    INVL0560
         DC    2C' '                                                  INVL0570
PRTBOH   DS    CL6                                                    INVL0580
PRTITVAL DS    CL15                                                   INVL0590
         DC    93C' '                                                 INVL0600
PRTTOTAL DS    0CL132                                                 INVL0610
PRTCOUNT DC    X'402020202020'                                        INVL0620
         DC    C' INPUT RECORDS******'                                INVL0630
PRTTOTVL DC    X'40206B2020206B2020214B2020'                          INVL0640
         DC    C' **'                                                 INVL0650
         DC    89C' '                                                 INVL0660
PATTERN1 DC    X'4020202 14B2020'                                     INVL0670
PATTERN2 DC    X'402020202020'                                        INVL0680
PATTERN3 DC    X'402020206B2020206B2020214B2020'                      INVL0690
WRKITCST DS    PL3                                                    INVL0700
WRKBOH   DS    PL3                                                    INVL0710
WRKITVAL DS    PL6                                                    INVL0720
COUNT    DC    PL3'0'                                                 INVL0730
TOTVALUE DC    PL5'0'                                                 INVL0740
         END   BEGIN                                                  INVL0750
```

FIGURE 3-15 Investment-Listing Program (Continued)

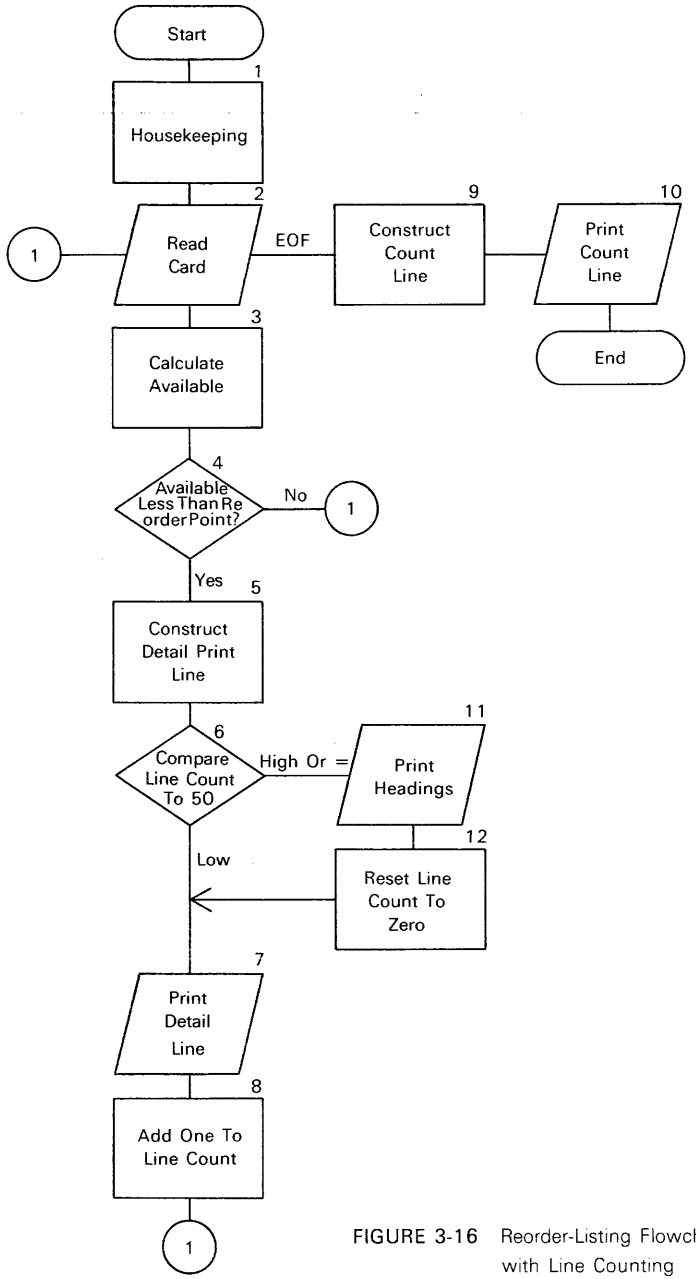


FIGURE 3-16 Reorder-Listing Flowchart with Line Counting

TOPIC THREE Completing the Basic Subset In topics 1 and 2 you were introduced to the basic structure of the assembler-language program. In this topic you will build on that base with some additional BAL coding techniques and instructions. You will learn how to control skipping and spacing in a more professional way, how to code punched-card output, and so on.

ASA CHARACTERS AND LINE COUNTING

The use of the CNTRL and PRTOV macro for forms control is a technique that is fast being replaced by the technique of *line counting* and the use of *control characters*. When line counting is used, the program does not rely on a printer for output. If a printer is not available at the time of program execution, the print file is recorded on magnetic tape or disk and held until it is convenient to read the file and print its contents. Thus, lack of a printer doesn't hold up the most significant aspects of processing. On large systems, it is the normal mode of operation to write print files on tape or disk prior to printing. As a result, line counting is required for such installations.

The flowchart in figure 3-16 and the program listing in figure 3-17 indicate how the reorder-listing program must be changed if line counting is used. This program functions in exactly the same manner as the program in figure 3-12: a reorder listing is printed with headings and a final count line. The remarks on the instructions in figure 3-17 indicate some of the major changes that have been made in the previous program.

If you compare the flowchart in figure 3-16 with the one in figure 3-11, you will see the differences in logic. Instead of checking for a channel-12 punch, the line-counting routine checks to see whether 50 lines have been printed on a page. If yes, the program skips to the next page, prints headings, resets a field (called the line-count field) to zero, and continues. If no, the program prints a detail line, adds one to the line-count field, and continues.

Because line counting is used, the DTFPR for the program does not specify CONTROL=YES or PRINTOV=YES.

Instead, it specifies CTLCHR=ASA, which means the control characters recommended by the American National Standards Institute will be used for forms control. Other control characters can be used, but the move toward standardization is very strong in the data processing industry.

When control characters are used, one byte at the start of an output area indicates the skipping or spacing to be performed when the area is printed. Thus, BLKSIZE in the DTFPR specifies 133—one control byte followed by 132 bytes corresponding to the 132 print positions. If you look at the data definitions for the printer-output and work areas, you will see the extra byte added at the start of each area.

The four most commonly used ASA control characters are:

CHARACTER	MEANING
blank	Space one line before printing
0	Space two lines before printing
-	Space three lines before printing
1	Skip to channel 1 before printing

The complete range of ASA control characters and their uses is covered in Appendix B of the IBM DOS Supervisor and I/O Macros manual, number GC24-5037.

The line counting logic is given in lines 250-330 of the program. The first two lines correspond to block 6 of the flowchart in figure 3-16:

```
CP    LINECNT,=P'50'
BL    PRTDET
```

In other words, the field named LINECNT is compared to a packed value of 50. If LINECNT is less than 50, a branch to PRTDET takes place and a detail line is printed. If LINECNT is equal to or higher than 50, no branch takes place and the three heading lines are printed. Since LINECNT is given a starting value of 50, no branch takes place the first time through the routine and the succeeding PUT instructions are executed.

The instructions that follow the PUT instructions are:

```
ZAP   LINECNT,=P'0'
MVI   PRTDETL,C'0'
```

The ZAP instruction changes the value of LINECNT from 50 to 0. The MVI moves the control character 0 into the first byte of the work area for the detail output line. Note that in an immediate instruction (the SI format), the immediate operand is specified using a type code and nominal value in the same manner as for a DC statement. However, no equals sign is needed, in contrast with a literal operand.

The next three lines of code are:

```
PRTDET   PUT   PRTOUT,PRTDETL
         AP    LINECNT,=P'1'
         B     READCARD
```

Since the control character in the PRTDETL area has just been set to zero, the printer is spaced twice before printing the detail line. After a detail line is printed, the AP instruction adds one to LINECNT and the program branches back to the GET instruction. During the processing for the next card, the control character for the detail line is set to blank causing single spacing to take place until page overflow occurs and headings are printed again.

RELATIVE ADDRESSING AND EXPLICIT LENGTHS

Lines 130 and 140 of figure 3-17 illustrate the use of relative addressing and explicit lengths to set all the characters of a printer work area (PRTDETL) to blanks:

```
MVI   PRTDETL,X'40'
MVC   PRTDETL+1(132),PRTDETL
```

The MVI instruction moves one blank into the first byte of this area. This, in fact, is the control byte so this instruction prepares the area for single spacing. Even though PRTDETL has a length of 133 bytes, only 1 byte is affected because the MVI has an implied length of 1 byte.

```
REORDLST START 0                                                        ORDR0010
BEGIN    BALR  3,0                                                      ORDR0020
         USING *,3                                                      ORDR0030
         OPEN  CARDIN,PRTOUT                                            ORDR0040
READCARD GET   CARDIN,CRDWRKA                                           ORDR0050
         AP    COUNT,=P'1'          ADD ONE TO COUNT                   ORDR0060
         PACK  WRKAVAIL,CRDONHND                                        ORDR0070
         PACK  WRKONORD,CRDONORD                                        ORDR0080
         AP    WRKAVAIL,WRKONORD                                        ORDR0090
         PACK  WRKORDPT,CRDORDPT                                        ORDR0100
         CP    WRKAVAIL,WRKORDPT                                        ORDR0110
         BNL   READCARD                                                 ORDR0120
         MVI   PRTDETL,X'40'        MOVE BLANK TO CONTROL BYTE          ORDR0130
         MVC   PRTDETL+1(132),PRTDETL BLANK PRINTER WORK AREA           ORDR0140
         PACK  PACKAREA,CRDITNBR                                        ORDR0150
         MVC   PRTITNBR,PATTERN1                                        ORDR0160
         ED    PRTITNBR,PACKAREA                                        ORDR0170
         MVC   PRTITDES,CRDITDES                                        ORDR0180
         PACK  PACKAREA,CRDPRICE                                        ORDR0190
         MVC   PRTPRICE,PATTERN2                                        ORDR0200
         ED    PRTPRICE,PACKAREA                                        ORDR0210
         MVC   PRTAVAIL,PATTERN1                                        ORDR0220
         ED    PRTAVAIL,WRKAVAIL                                        ORDR0230
         MVC   PRTORDPT,PATTERN1                                        ORDR0240
         ED    PRTORDPT,WRKORDPT                                        ORDR0250
         CP    LINECNT,=P'50'       COMPARE LINE COUNT TO 50           ORDR0260
         BL    PRTDET               BRANCH ON LOW TO PRTDET            ORDR0270
         PUT   PRTOUT,HDGLINE1      PRINT FIRST HEADING LINE           ORDR0280
         PUT   PRTOUT,HDGLINE2      PRINT SECOND HEADING LINE          ORDR0290
         PUT   PRTOUT,HDGLINE3      PRINT THIRD HEADING LINE           ORDR0300
         ZAP   LINECNT,=P'0'        RESET LINE COUNT TO ZERO           ORDR0310
         MVI   PRTDCTL,C'0'         MOVE ZERO TO DETAIL CONTROL BYTE   ORDR0320
PRTDET   PUT   PRTOUT,PRTDETL       PRINT DETAIL LINE                  ORDR0330
         AP    LINECNT,=P'1'        ADD ONE TO LINE COUNT              ORDR0340
         B     READCARD                                                 ORDR0350
CRDEOF   ED    CNTPATRN,COUNT       EDIT COUNT                         ORDR0360
         PUT   PRTOUT,CNTLINE       PRINT COUNT LINE                   ORDR0370
         CLOSE CARDIN,PRTOUT                                           ORDR0380
         EOJ                                                           ORDR0390
         ORDR0400
* THE CARD FILE DEFINITION FOLLOWS                                     XORDR0410
CARDIN   DTFCD DEVADDR=SYSIPT,IOAREA1=CRDIO1,IOAREA2=CRDIO2,           ORDR0420
               EOFADDR=CRDEOF,WORKA=YES                                ORDR0430
* THE PRINTER FILE DEFINITION FOLLOWS                                  XORDR0440
PRTOUT   DTFPR DEVADDR=SYSLST,                                         XORDR0450
               IOAREA1=PRTIO1,                                         XORDR0460
               IOAREA2=PRTIO2,                                         XORDR0470
               WORKA=YES,                                              XORDR0480
               BLKSIZE=133,                                            ORDR0490
               CTLCHR=ASA                                              ORDR0500
* THE DATA DEFINITIONS FOR THE TWO CARD I/O AREAS FOLLOW                ORDR0510
CRDIO1   DS    CL80                                                    ORDR0520
CRDIO2   DS    CL80                                                    ORDR0530
* THE DATA DEFINITIONS FOR THE TWO PRINTER AREAS FOLLOW                 ORDR0540
PRTIO1   DS    CL133                                                   ORDR0550
PRTIO2   DS    CL133
```

FIGURE 3-17 Reorder-Listing Program with Line Counting

```
* THE DATA DEFINITIONS FOR THE CARD WORK AREA FOLLOW                    ORDR0560
CRDWRKA  DS    0CL80                                                    ORDR0570
CRDITNBR DS    CL5                                                      ORDR0580
CRDITDES DS    CL20                                                     ORDR0590
         DS    CL5                                                      ORDR0600
CRDPRICE DS    CL5                                                      ORDR0610
CRDORDPT DS    CL5                                                      ORDR0620
CRDONHND DS    CL5                                                      ORDR0630
CRDONORD DS    CL5                                                      ORDR0640
         DS    CL30                                                     ORDR0650
* THE DATA DEFINITIONS FOR THE PRINTER HEADING LINES FOLLOW             ORDR0660
HDGLINE1 DS    0CL133                                                   ORDR0670
         DC    C'1'                                                     ORDR0680
         DC    24C' '                                                   ORDR0690
         DC    C'REORDER LISTING'                                       ORDR0700
         DC    93C' '                                                   ORDR0710
HDGLINE2 DS    0CL133                                                   ORDR0720
         DC    C'0'                                                     ORDR0730
         DC    C' ITEM              ITEM                 UNIT          XORDR0740
               REORDER'                                                 ORDR0750
         DC    69C' '                                                   ORDR0760
HDGLINE3 DS    0CL133                                                   ORDR0770
         DC    C' '                                                     ORDR0780
         DC    C'  NO.          DESCRIPTION          PRICE    AVAILABLEXORDR0790
               POINT'                                                   ORDR0800
         DC    70C' '                                                   ORDR0810
* THE DATA DEFINITIONS FOR THE PRINTER DETAIL LINE FOLLOW               ORDR0820
PRTDETL  DS    0CL133                                                   ORDR0830
PRTDCTL  DS    CL1                                                      ORDR0840
PRTITNBR DS    CL6                                                      ORDR0850
         DC    5C' '                                                    ORDR0860
PRTITDES DS    CL20                                                     ORDR0870
         DC    4C' '                                                    ORDR0880
PRTPRICE DS    CL7                                                      ORDR0890
         DC    4C' '                                                    ORDR0900
PRTAVAIL DS    CL6                                                      ORDR0910
         DC    4C' '                                                    ORDR0920
PRTORDPT DS    CL6                                                      ORDR0930
         DC    70C' '                                                   ORDR0940
* THE DATA DEFINITIONS FOR THE FINAL TOTAL LINE FOLLOW                  ORDR0950
CNTLINE  DS    0CL133                                                   ORDR0960
         DC    C'-'                                                     ORDR0970
CNTPATRN DC    X'4020206B202020'                                        ORDR0980
         DC    C' CARDS IN THE INPUT DECK'                              ORDR0990
         DC    101C' '                                                  ORDR1000
* THE DATA DEFINITIONS THAT FOLLOW DEFINE OTHER WORK AREAS NEEDED        ORDR1010
* BY THE PROGRAM                                                        ORDR1020
PATTERN1 DC    X'402020202020'                                          ORDR1030
PATTERN2 DC    X'40202021 4B2020'                                       ORDR1040
WRKAVAIL DS    PL3                                                      ORDR1050
WRKONORD DS    PL3                                                      ORDR1060
WRKORDPT DS    PL3                                                      ORDR1070
PACKAREA DS    PL3                                                      ORDR1080
COUNT    DC    PL3'0'                                                   ORDR1090
LINECNT  DC    P'50'                                                    ORDR1100
         END   BEGIN                                                    ORDR1110
```

FIGURE 3-17 Reorder-Listing Program with Line Counting (Continued)

The second instruction, the MVC, uses a *relative address* and an *explicit length* for the first operand, PRTDETL+1(132). This means that the address used in the actual machine instruction should be one greater than the address assigned to PRTDETL and the length should be 132 bytes instead of 133. If, for example, the address 8280 is assigned to PRTDETL, this operand refers to the 132 bytes beginning at address 8281. Since the second operand is PRTDETL and since the MVC instruction moves from left to right during execution, the blank in the first byte of PRTDETL is moved through the remaining 132 bytes of the area. For instance, the blank in byte 8280 is moved to byte 8281, then the blank in 8281 is moved to 8282, and so on, until 132 blanks have been moved.

Incidentally, in many programs this blanking technique is used to set a print area to blanks before moving data into the area. This prevents any characters left over from previous print lines from being printed. If you look at the PRTDETL data definitions, you can see that the bytes between fields are not defined as blank constants since these bytes will be blanked during program execution.

To code a relative address, you use a plus or minus sign followed by a decimal number. Thus, CRDPRICE+3 addresses the first decimal position of the price field and CRDPRICE−25 addresses the first byte of the description field.

An explicit length shows as a decimal number in parentheses following a data name or relative address. Thus, CRDPRICE(3) refers to the first three bytes of the price field while CRDPRICE+3(2) refers to the last two bytes of the field. When using relative addressing, an explicit length is often given; otherwise, the length is taken from the data definitions of the fields being operated upon. As a result, CRDPRICE+3 refers to the five bytes starting with the fourth byte of the price field. Since this would take three bytes from the reorder-point field, an explicit length of two or less would probably be used.

You can use an explicit length on any operand that has a length factor in the actual instruction. Thus, the MVC instruction can have an explicit length only on the first operand, but the AP instruction can have an explicit length on both operands as in this example:

```
AP    FIELDA+1(4),FIELDB+3(2)
```

By referring to an instruction's format, you can determine which operands can have explicit lengths.

An explicit length should be used whenever an operand length other than the one given in the data definition is required. For instance, PRTDETL in figure 3-17 is assigned a length *attribute* of 133 during assembly. (An attribute is simply one of the characteristics assigned to a symbol.) As a result, the instruction

```
MVC   PRTDETL+1,PRTDETL
```

will move 133 bytes of data since the length used in the machine instruction is the length attribute of the first operand. Since only 132 bytes should be moved, an explicit length is needed. Otherwise, the first byte following the print area will also be set to a blank.

OVERLAPPED OPERANDS

The blanking technique illustrated above also illustrates *overlapped operands*. In short, for various types of data-movement instructions, it is possible to have the same field specified in both operands. In blanking the print area, PRTDETL is given for both operands. Similarly, it is possible to pack a field in its own area. Thus, this code is valid:

```
PACK  CRDPRICE,CRDPRICE
```

If the CRDPRICE field contains 00440 before execution, it contains this data after execution:

```
00 00 00 44 0F
```

The field can then be edited as follows:

```
PACK  CRDPRICE,CRDPRICE
MVC   PRTPRICE,PATTERN2
ED    PRTPRICE,CRDPRICE+2(3)
```

In the edit instruction, only the rightmost three bytes of the

CRDPRICE field are edited, so the number of digits in the sending field matches the number of digit places in the edit pattern—a requirement of the edit instruction.

CARD OUTPUT

The OPEN, CLOSE, GET, and PUT instructions are used for almost all I/O operations in assembler language. To go from one I/O device to another, the only code that changes is that used in the DTF. To punch cards, for instance, the PUT instruction is used along with a DTFCD with appropriate keyword operands such as:

```
CRDOUT   DTFCD DEVADDR=SYSPCH,                X
               IOAREA1=CRDOUTA,               X
               TYPEFLE=OUTPUT
```

SYSPCH should be used for the DEVADDR operand until you learn other valid specifications, and TYPEFLE must be OUTPUT. In this case, since a work area isn't specified, the data in the output area (CRDOUTA) will be punched when the following macro is executed:

```
         PUT   CRDOUT
```

If you refer back to figure 3-5, you can see other operands that you might need for card punching. In particular, to overlap card punching with other operations, you use the keywords IOAREA2 and WORKA=YES. Then, if the DTF is named CRDOUT and the work area is named CRDOUTW, a card will be punched when this macro is executed:

```
         PUT   CRDOUT,CRDOUTW
```

Remember that all input or output files must be opened before processing and closed before issuing the EOJ macro.

CHARACTER COMPARISONS

Thus far, you know how to compare packed-decimal fields, but you do not know how to compare EBCDIC fields. For them, the CLC (compare-logical-characters) and the CLI (compare-logical-immediate) instructions are used. In contrast to the compare-decimal instruction, which bases the comparison on numeric value, these instructions are called *logical* since they compare bit patterns. Although the operands are normally in EBCDIC format, the comparison will take place regardless of data format.

The CLC instruction can be used to compare two areas of storage up to 256 bytes in length. Like the MVC instruction, the length code is taken from the first operand as in this example:

```
         CLC   FIELDA,FIELDB
```

Here, FIELDA will be compared to a field of equal length beginning at the address represented by the label FIELDB. Following the compare-logical-characters instruction, you would normally code a branch instruction to test the result of the comparison.

The CLI instruction operates in the same manner as the CLC, except that one byte of data in the instruction itself is compared with one byte of data in the field named. The immediate operand is given as operand-2 in the assembler-language code as:

```
         CLI   FIELDZ,C'0'
```

This instruction compares the first byte of FIELDZ to the hex constant F0 stored in the instruction. If FIELDZ contained four bytes and you wanted to check the rightmost byte for hex F0, you could code a relative address this way:

```
         CLI   FIELDZ+3,X'F0'
```

Remember that a valid immediate operand (operand-2) is type code X, Z, P, or C followed by a one-byte nominal value enclosed in single quotes so X'F0', C'0', and Z'0' are equivalent.

When two logical fields are compared, they are considered equal if the bit patterns in both fields are identical. But what if they aren't? Is X-12-13 lower or higher than X12345? The answer to this depends on the binary value of the bytes being compared. In this case, since the CLC instruction operates from left to right, the leftmost bytes

would be compared first and would be found to be equal. Then, the bit pattern for the hyphen (-) would be compared with the bit pattern for 1, or 01100000 would be compared with 11110000. Since 01100000 is less than 11110000 when binary values are considered, X-12-13 is considered less than X12345 by the System/360-370.

The System/360-370 sequence of some commonly used EBCDIC characters, called the *collating sequence*, is (from lowest to highest):

the blank
.
(
&
)
-
/
,
#
'
"
the letters A-Z
the numbers 0-9

Therefore, T/X is less than TOM, H&R is less than H-R, and T2S is less than T29. The sequence of all 256 bit patterns possible in a byte is given on the IBM "green card" (form X20-1703 for the System/360; form GX20-1850 for the System/370). (Since this card has much valuable information for a BAL programmer, you should get one. Hereafter, this card will be referred to simply as the IBM green card. Curiously enough, the card for the 370 is yellow, even though it is commonly referred to as the green card.)

If fields with different formats are compared, it is normally a programming error. For instance, if a CLC instruction compares an EBCDIC field containing 123 (hex F1F2F3) with a packed-decimal field containing 58234 (hex 58234C), the EBCDIC field will be considered higher. (The leftmost four bits are 1111 as compared with 0101.) Data formats, then, are critical when using compare instructions.

ROUNDING ARITHMETIC RESULTS

System/360-370 arithmetic instructions operate only on whole numbers. For instance, a unit price of $4.49 is treated as 449 when arithmetic operations are performed. As a result, the programmer is responsible for keeping track of the location of decimal points and, if necessary, for rounding the results of calculations. To code the operations for rounding, relative addressing and explicit length codes are commonly used along with two new move instructions, the move-numerics (MVN) and the move-with-offset (MVO).

MVN The move-numerics instruction operates in the same way as a MVC instruction, but only the digit (numeric) halves of the fields are moved. To illustrate:

```
Instruction:    MVN    RECEIVE,SEND

                RECEIVE          SEND
Before:     AA BB CC DD    12 34 56 78
After:      A2 B4 C6 D8    12 34 56 78
```

As you can see, the zone portions of the bytes in the receiving field are unchanged.

The MVN instruction is useful in rounding numbers according to the techniques illustrated in figures 3-18 and 3-19. In technique one of figure 3-18, the problem is to multiply a two-decimal-position amount field by a two-decimal-position percent field and obtain a rounded two-decimal-position result. After the multiply instruction is executed, a four-decimal-position result is stored in RESULT4. Fifty (50) is added to this result by using a literal in the AP instruction. The effect of this is to increase the second decimal position by one if the third decimal position is 5 or over. That is, the answer is rounded to two decimal positions provided the rightmost two digits are truncated. In the sample data in figure 3-18, the third and fourth decimal positions contain 41 so when 50 is added to them the result is 91 and the second decimal position is unchanged.

Source Code	Example of Field Changes During Execution			
	AMOUNT2	PERCENT2	RESULT4	RNDRSLT2
STARTING VALUES:	12 34 56 7C	02 3F	99 99 99 99 99 99	99 99 99 99
ZAP RESULT4,AMOUNT2			00 00 12 34 56 7C	
MP RESULT4,PERCENT2			00 02 83 95 04 1C	
AP RESULT4,=P'50'			00 02 83 95 09 1C	
MVC RNDRSLT2,RESULT4+1				02 83 95 09
MVN RNDRSLT2+3(1),RESULT4+5				02 83 95 0C
TECHNIQUE ONE				
STARTING VALUES:	12 34 56 7C	02 3F	99 99 99 99 99 99	99 99 99 99
ZAP RESULT4,AMOUNT2			00 00 12 34 56 7C	
MP RESULT4,PERCENT2			00 02 83 95 04 1C	
AP RESULT4,=P'50'			00 02 83 95 09 1C	
MVN RESULT4+4(1),RESULT4+5			00 02 83 95 0C 1C	
ZAP RNDRSLT2,RESULT4(5)				02 83 95 0C
TECHNIQUE TWO				
AMOUNT2 DS PL4			12345.67	
PERCENT2 DS PL2			.23	
RESULT4 DS PL6			2839.5041	
RNDRSLT2 DS PL4			Rounded: 2839.50	
DATA DEFINITIONS			SAMPLE PROBLEM	

FIGURE 3-18 Rounding an Even Number of Decimal Places

The next two instructions complete the rounding by dropping the rightmost two digits of RESULT4 as the result is moved to RNDRSLT2:

```
MVC   RNDRSLT2,RESULT4+1
MVN   RNDRSLT2+3(1),RESULT4+5
```

When the MVC is executed, the second through fifth bytes of RESULT4 are moved into the four-byte RNDRSLT2 field. (Because the length is taken from operand-1 and RNDRSLT2 is defined as PL4, four bytes are moved; otherwise, explicit lengths could be used to move the appropriate number of bytes.) When the MVN is executed, the sign of RESULT4 is moved to RNDRSLT2.

Technique two in figure 3-18 is similar. After the literal 50 is added to the result field, the sign of RESULT4 is moved one byte to the left with this instruction:

```
MVN   RESULT4+4(1),RESULT4+5
```

Here, an explicit length is needed, so only one byte is operated upon. After the instruction is executed, the leftmost five bytes of RESULT4 have the rounded result value. These bytes are then moved to the RNDRSLT2 field by the ZAP instruction, which uses an explicit length of five for the sending field. Because the receiving field is only four bytes, the leftmost byte of RESULT4 will be truncated.

MVO The move-with-offset instruction has the unique attribute of offsetting the data one half-byte to the left during the move, as:

```
Instruction:    MVO    RECEIVE,SEND

                Before          After
SEND:           AA BB CC        AA BB CC
                                  \  \  \
RECEIVE:    FF FF FF FF     0A AB BC CF
```

Notice that the rightmost half-byte of the receiving field is not changed. The data from the sending field is placed in the receiving field, but it is offset one half byte to the left. Also notice that in the receiving field the high order (leftmost) half-bytes that are not filled from the sending field are padded to hex zeros.

Truncation can also occur. For example, if the field RECEIVE is only three bytes, the leftmost half byte is truncated:

```
Instruction:    MVO    RECEIVE,SEND

                Before          After
SEND:       AA BB CC        AA BB CC
                              /  /  /
RECEIVE:    FF FF FF        AB BC CF
                           /
                        truncated
```

The operation of the MVO instruction, then, is similar to that of the packed decimal instructions, and not to the operations of the other move instructions. The MVO instruction uses length factors on both operands, and execution moves from right to left. Padding with hex zeros and truncation can both occur.

The common use of the MVO instruction is in rounding packed-decimal numbers to an odd number of digits. In figure 3-19, technique one illustrates a five-decimal-position result being rounded to two decimal positions. Since three positions must be dropped, the constant that is added to the result is 500 rather than 50 as in the previous example. Using the sample data, the second decimal position is increased by one (from 6 to 7) when the AP instruction is executed.

Next, the MVO instruction moves the digits from the leftmost four bytes of RESULT5 to RNDRSLT2:

```
MVO    RNDRSLT2,RESULT5(4)
```

The explicit length of four is used so that three decimal positions are dropped during the move. Because the move is offset, the digit half of the rightmost byte of RNDRSLT2 is not changed by the move. Instead, the MVN instruction moves the sign from RESULT4 to RNDRSLT2:

```
MVN   RNDRSLT2+3(1),RESULT5+5
```

The second technique in figure 3-19 shows how overlapped fields can be used with the MVO instruction. When the MVO instruction is executed, it moves one byte at a time like this:

```
Instruction:    MVO    RESULT5,RESULT5(4)

                       RESULT5
Before:          00 28 02 47 20 9C
                    \  \  \  \  \
After Byte 1:    00 28 02 47 24 7C
                    \  \  \  \  \
After Byte 2:    00 28 02 40 24 7C
                    \  \  \  \  \
After Byte 3:    00 28 02 80 24 7C
                    \  \  \  \  \
After Byte 4:    00 20 02 80 24 7C

Padding:         00 00 02 80 24 7C
```

This technique for rounding is not used if all decimal positions are needed in subsequent calculations. When a result is rounded in its own field, the truncated decimal positions are not available for later processing.

MOVE ZONES

One other instruction that can be handy for manipulating numeric data is the move-zones instruction. When it is executed, the zone halves of the bytes in the sending field are moved to the zone halves of the bytes in the receiving field. For example,

```
MVZ    FIELDA(3),=X'DODODO'
```

moves hex Ds into the zone portions of the first three bytes of FIELDA.

Source Code	Example of Field Changes During Execution			
	AMOUNT2	PERCENT3	RESULT5	RNDRSLT2
STARTING VALUES:	12 34 56 7C	22 7C	99 99 99 99 99 99	99 99 99 99
ZAP RESULT5,AMOUNT2			00 00 12 34 56 7C	
MP RESULT5,PERCENT3			00 28 02 46 70 9C	
AP RESULT5,=P'500'			00 28 02 47 20 9C	
MVO RNDRSLT2,RESULT5(4)				02 80 24 79
MVN RNDRSLT2+3(1),RESULT5+5				02 80 24 7C
TECHNIQUE ONE				
STARTING VALUES:	12 34 56 7C	22 7C	99 99 99 99 99 99	99 99 99 99
ZAP RESULT5,AMOUNT2			00 00 12 34 56 7C	
MP RESULT5,PERCENT3			00 28 02 46 70 9C	
AP RESULT5,=P'500'			00 28 02 47 20 9C	
MVO RESULT5,RESULT5(4)			00 00 02 80 24 7C	
TECHNIQUE TWO				
AMOUNT2 DS PL4			12345.67	
PERCENT DS PL2			.227	
RESULT5 DS PL6			2802.46709	
RNDRSLT2 DS PL4			Rounded: 2802.47	
DATA DEFINITIONS			SAMPLE PROBLEM	

FIGURE 3-19 Rounding an Odd Number of Decimal Places

SUMMARY

You have now been introduced to a complete subset of BAL. This subset allows you to program the major functions of a computer system: input, output, forms control, arithmetic, and logic. By using only the elements of this subset, you can write programs of great complexity.

Terminology

line counting
control character
relative address
explicit length
attribute
overlapped operands
logical comparison
collating sequence

Objective

Given a programming problem, code an acceptable BAL solution for it. The problem may require any of the elements of the subset.

Problems

1 Suppose the reorder-listing program in figure 3-17 must be modified. Instead of triple spacing before printing the count line, the program is supposed to skip to the next page. Also, a count of the number of items to be reordered must be kept, and additional total lines must be printed so the total lines will look like this:

```
XX,XXX CARDS IN INPUT DECK
XX,XXX ITEMS TO BE REORDERED
XX.X% OF THE ITEMS ARE TO BE REORDERED
```

Using appropriate line numbers, indicate what changes, deletions, or additions must be made to the code in figure 3-17.

2 Write a program to punch identification codes and sequence numbers into a BAL source deck that is placed in the card punch. The identification code and number of source cards in the source deck is to be read from a card at the start of the program. Columns 1-4 of the card will contain the identification characters to be punched in source columns 73-76. The sequence numbers are to be punched in columns 77-80 of the source cards, starting with 0010 and counting by tens. The number of source cards is punched in card columns 5-8 of the input card. Card punching need not be overlapped with processing.

Solutions

1 Add this line between lines 120 and 130:

```
        AP      ORDCOUNT,=P'1'
```

Add these lines between lines 360 and 370:

```
        ED      OCNTPATT,ORDCOUNT
        PUT     PRTOUT,OCNTLINE
        ZAP     DIVIDEND,ORDCOUNT
        MP      DIVIDEND,=P'10000'
        DP      DIVIDEND,COUNT
        AP      DIVIDEND(5),=P'5'
        MVO     DIVIDEND(5),DIVIDEND(4)
        ED      PCTPATT,DIVIDEND+3(2)
        PUT     PRTOUT,PCTLINE
```

Substitute this line for line 960:

```
        DC      C'1'
```

Add these lines to the data definitions:

```
OCNTLINE DS     0CL133
         DC     C'0'
OCNTPATT DC     X'4020206B202020'
         DC     C'ITEMS TO BE REORDERED'
         DC     103C' '
PCTLINE  DS     0CL133
         DC     C'0'
PCTPATT  DC     X'4020214B20'
         DC     C'% OF THE ITEMS ARE TO BE REORDERED'
         DC     92C' '
ORDCOUNT DC     PL3'0'
DIVIDEND DS     PL8
```

2 Figure 3-20 is an acceptable solution. Notice the use of the UNPK and MVZ instructions to get the sequence number into an acceptable form for punching. The MVZ instruction changes the plus sign in the zoned-decimal output field from hex C to hex F.

```
IDSEQ    START 0                                                    ISEQ0010
IDCARD   DTFCD DEVADDR=SYSIPT,IOAREA1=ID                            ISEQ0020
SRCECRD  DTFCD DEVADDR=SYSPCH,IOAREA1=SOURCEIO,TYPEFLE=OUTPUT       ISEQ0030
PROGSTRT BALR  3,0                                                  ISEQ0040
         USING *,3                                                  ISEQ0050
         OPEN  IDCARD,SRCECRD                                       ISEQ0060
         GET   IDCARD                                               ISEQ0070
         PACK  PCRDCNT,CRDCNT                                       ISEQ0080
PUNCHCRD MVC   IDCODE,IDCHAR                                        ISEQ0090
         AP    PSEQNO,=P'10'                                        ISEQ0100
         UNPK  SEQNO,PSEQNO                                         ISEQ0110
         MVZ   SEQNO+3(1),=X'F0'                                    ISEQ0120
         PUT   SRCECRD                                              ISEQ0130
         SP    PCRDCNT,=P'1'                                        ISEQ0140
         CP    PCRDCNT,=P'0'                                        ISEQ0150
         BH    PUNCHCRD                                             ISEQ0160
         CLOSE IDCARD,SRCECRD                                       ISEQ0170
         EOJ                                                        ISEQ0180
ID       DS    0CL80                                                ISEQ0190
IDCHAR   DS    CL4                                                  ISEQ0200
CRDCNT   DS    CL4                                                  ISEQ0210
         DS    CL72                                                 ISEQ0220
SOURCEIO DS    0CL80                                                ISEQ0230
         DC    72C' '                                               ISEQ0240
IDCODE   DS    CL4                                                  ISEQ0250
SEQNO    DS    CL4                                                  ISEQ0260
PSEQNO   DC    PL3'0'                                               ISEQ0270
PCRDCNT  DS    PL3                                                  ISEQ0280
         END   PROGSTRT                                             ISEQ0290
```

FIGURE 3-20 Sequence Punching Program

4

Diagnostics and Debugging

Coding is by no means the end of the programmer's job. Before he is finished with a program, the code must be keypunched into a source deck, the source deck must be assembled—without errors—to create an object program, and the object program must be tested to prove that it does what it is intended to do. In actual practice, the programmer desk checks his program before and after keypunching, corrects diagnostic messages until the assembly process finally detects no errors, creates test data for the test run, and debugs any errors found during test runs. Topic 1 of this chapter covers desk checking and diagnostics; topic 2 covers testing and debugging.

TOPIC ONE Desk Checking and Diagnostics A programmer normally *desk checks* his program before and after keypunching. This saves computer time and programmer time because desk

checking almost always catches an error or two thus saving an assembly or a test run.

Before you have a program keypunched, you should desk check your coding forms using guidelines similar to these:

1 Scan the operation codes for spelling errors or oversights. Occasionally, you'll slip and code MOVE for MVC, or MULT for MP. Pay particular attention to DS and DC instructions. If a DS is coded when a DC is correct, it may well assemble without error, but it can cause significant errors during testing.

2 Make a list of all the labels that appear as operands. (As you compile the list, you may pick up a couple of misspellings.) Then check the list to be sure that every label is defined by appearing in the label column of some statement. Also, watch out for names that are defined more than once; they are a common problem in longer programs.

3 Double check the numbers used in relative addresses. It often helps to draw pictures of the fields with sample data in place.

4 Sketch record layouts as defined in your program and compare them with the layouts given in the problem definition. Make sure that the I/O areas and record work areas are long enough since overlaying constants with input data is a common programming error. For example, suppose a card input area is defined in error as:

```
CARDINPA DS    0CL80
AMOUNTA  DS    CL7
AMOUNTB  DS    CL8
AMOUNTC  DS    CL5
         DS    CL40
```

Suppose also that these definitions are followed by this constant:

```
PATTERN  DC    X'40202020206B2020206B2020214B2020'
```

In this case, when the GET instruction for the card file is executed, the card-input data will overlay the edit pattern and cause a program error when the program attempts to use the edit pattern.

5 Look over the DTFs in your program. Be sure that all required commas are in place. The keywords must be spelled correctly or they will be ignored. If you have continuation cards, check for a character in column 72 and make sure that the continuation lines start in column 16. Remember that you can have blanks between two operands, but not in the middle of one.

In addition to checking for such clerical errors, you should check for logic errors. A logic error won't cause an assembly error but it will cause testing difficulties. Logic errors are easiest to find while the program is still fresh in your mind. Here are some suggestions for finding logic errors.

1 The initial data in your work fields, print-line areas, and so on, may be important. If so, you must either define them with an initial value, or code instructions to set or reset these areas. Check to make sure that these values are established properly.

2 Check the flow of the program. For this it is often helpful to use some actual data and follow it through the flow of the program. What happens if the conditional branches of your program are not taken? Is the fall-through (no-branch) condition coded properly? Are the operation codes of the branch instructions correct? Frequently, the equal condition is not used properly. Remember, the branch-low and the branch-not-high instructions handle the equal condition in opposite ways. Also, don't assume that a line counter that should never exceed 50 won't somehow get to be 51. If the branch to your heading routine tests for equal to 50, you may have a report with no headings.

3 Now, even though the program appears to flow correctly, compare it to your flowchart. You may find that a possible condition has been overlooked. Little things, like forgetting to open and close files are often found this way.

When you have checked for clerical and logical errors in this manner, your program is ready for keypunching. Unfortunately, keypunching is an opportunity for errors to

return to the source code. Therefore, either get an 80-80 listing of the source deck or use the cards themselves to compare with your corrected coding sheets. You may find, for instance, that the way you write a 5 is mistaken for an S by the keypunch operator. You may also find some random keypunch errors. It pays to spend a few minutes to catch keypunch errors since it usually saves an assembly or test run.

THE ASSEMBLY LISTING

When a program is assembled, a number of different types of output are printed as part of the *assembly listing*. These are illustrated for the reorder-listing program with ASA control characters in figure 4-1 (parts 1-7). Each type of output starts on a new page and continues for as many pages as needed. In sequence, the normal output consists of (1) an external symbol dictionary, (2) the assembly listing, (3) the relocation dictionary, (4) the cross-reference listing, and (5) the diagnostic listing. Because the external symbol dictionary and the relocation dictionary are of little use to the beginning assembler-language programmer, you can ignore them and they will be omitted from other assembly listings in this book.

The assembly listing itself is parts 2-4 of figure 4-1 (although the entire assembly output is also commonly referred to as the assembly listing). Since the assembly listing contains considerable information, I will go over it in detail.

From left to right, there are six columns of information on the assembly listing. The rightmost column, headed SOURCE STATEMENT, is an 80-80 listing of the source deck. In addition, however, instructions generated from macro instructions are shown. These are preceded by a plus sign. For example, the GET macro has generated 5 statements, the first of which is a comment, as follows:

```
+* 360N-CL-453 GET       CHANGE LEVEL 3-0
+READCARD L     1,=A(CARDIN) GET DTF TABLE ADDRESS
+         L     0,=A(CRDWRKA) GET WORK AREA ADDRESS
+         L     15,16(1) GET LOGIC MODULE ADDRESS
+         BAL   14,8(15) BRANCH TO GET ROUTINE
```

On the other hand, the DTFPR, which is also a macro (though it hasn't been referred to as such before), has generated 20 lines of source code.

After the END statement in the source listing are data definitions representing the literals used by the program. For example, the last three statements represent the literals used for packed values 50, 0, and 1. The other literal definitions shown represent those generated by macros. They use type codes that are covered later in this book.

To the left of the source statements is a column headed STMT, short for statement number. This column gives a number to each of the source statements of the program. The statement numbers are then used in the cross-reference and diagnostic listings to refer to the source statements.

```
                                EXTERNAL SYMBOL DICTIONARY                PAGE    1

SYMBOL   TYPE  ID   ADDR   LENGTH LD ID

REORDLST  SD   01 000000 000634
IJDFAZIW  ER   02
```

FIGURE 4-1 (Part 1) External Symbol Dictionary

```
                                                                                                              PAGE    1

  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                                          FDOS CL3-9 12/13/73

000000                                1 REORDLST START 0
000000 0530                           2 BEGIN    BALR  3,0                        LOAD BASE REGISTER
000002                                3          USING *,3
                                      4          OPEN  CARDIN,PRTOUT
                                      5+* 360N-CL-453 OPEN     CHANGE LEVEL 3-3                             3-3
000002 0700                           6+         CNOP  0,4
000004                                7+         DC    0F'0'
000004 4110 35FE            00600     8+         LA    1,=C'$$BOPEN
000008 4500 3012            00014     9+IJJ00001 BAL   0,*+4+4*(3-1)
00000C 00000000                      10+         DC    A(CARDIN)
           *** ERROR ***
000010 00000118                      11+         DC    A(PRTOUT)
000014 0A02                          12+         SVC   2
                                     13 READCARD GET   CARDIN,CARDWRKA            READ CARD INTO WORK AREA
                                     14+* 360N-CL-453 GET      CHANGE LEVEL 3-0
000016 5810 360E            00610    15+READCARD L     1,=A(CARDIN) GET DTF TABLE ADDRESS
00001A 5800 3612            00614    16+         L     0,=A(CARDWRKA) GET WORK AREA ADDRESS
00001E 58F1 0010            00010    17+         L     15,16(1) GET LOGIC MODULE ADDRESS
000022 45EF 0008            00008    18+         BAL   14,8(15) BRANCH TO GET ROUTINE
000026 FA20 35F2 3630 005F4 00632    19          AP    COUNT,=P'1'                ADD ONE TO COUNT
00002C F224 35E6 3318 005E8 0031A    20          PACK  WRKAVAIL,CRDONHND
000032 F224 35E9 331D 005EB 0031F    21          PACK  WRKONORD,CRDONORD
                                     22          ADD   WRKAVAIL,WRKONORD
           *** ERROR ***
000038 F224 35EC 3313 005EE 00315    23          PACK  WRKORDPT,CRDORDPT
00003E F922 35E6 35EC 005E8 005EE    24          CP    WRKAVAIL,WRKORDPT .
000044 47B0 3014            00016    25          BNL   READCARD
000048 9240 34CF      004D1          26          MVI   PRTDETL,X'40'              MOVE BLANK TO CONTROL BYTE
00004C D283 34D0 34CF 004D2 004D1    27          MVC   PRTDETL+1(132),PRTDETL     BLANK PRINTER WORK AREA
000052 F224 35EF 32F0 005F1 002F2    28          PACK  PACKAREA,CRDITNBR
                                     29          MVL   PRTITNBR,PATTERN1
           *** ERROR ***
000058 DE05 34D0 35EF 004D2 005F1    30          ED    PRTITNBR,PACKAREA          EDIT ITEM NUMBER FIELD
00005E D213 34DF 32F5 004E1 002F7    31          MVC   PRTITDES,CRDITDES          MOVE ITEM DESCRIPTION
000064 F224 35EF 330E 005F1 00310    32          PACK  PACKAREA,CRDPRICE
00006A D206 34F3 35DF 004F5 005E1    33          MVC   PRTPRICE,PATTERN2
000070 DE06 34F3 35EF 004F5 005F1    34          ED    PRTPRICE,PACKAREA          EDIT UNIT PRICE
000076 D205 34FE 35D9 00500 005DB    35          MVC   PRTAVAIL,PATTERN1
00007C DE05 34FE 35E6 00500 005E8    36          ED    PRTAVAIL,WRKAVAIL          EDIT AVAILABLE
000082 D205 3508 35D9 0050A 005DB    37          MVC   PRTORDPT,PATTERN1
000088 DE05 3508 35EC 0050A 005EE    38          ED    PRTORDPT,WRKORDPT          EDIT ORDER POINT
00008E F911 35F5 362E 005F7 00630    39          CP    LINECNT,=P'50'             COMPARE LINE COUNT TO 50
000094 4740 30D0            000D2    40          BL    PRTDET                     BRANCH ON LOW TO PRTDET
                                     41          PUT   PRTOUT,HDGLINE1            PRINT FIRST HEADING LINE
                                     42+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                         3-5
000098 5810 3616            00618    43+         L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
00009C 5800 361A            0061C    44+         L     0,=A(HDGLINE1) GET WORK AREA ADDRESS
0000A0 58F1 0010            00010    45+         L     15,16(1) GET LOGIC MODULE ADDRESS      3-5
0000A4 45EF 000C            0000C    46+         BAL   14,12(15) BRANCH TO PUT ROUTINE         3-5
                                     47          PUT   PRTOUT,HDGLINE2            PRINT SECOND HEADING LINE
                                     48+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                         3-5
0000A8 5810 3616            00618    49+         L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000AC 5800 361E            00620    50+         L     0,=A(HDGLINE2) GET WORK AREA ADDRESS
0000B0 58F1 0010            00010    51+         L     15,16(1) GET LOGIC MODULE ADDRESS      3-5
0000B4 45EF 000C            0000C    52+         BAL   14,12(15) BRANCH TO PUT ROUTINE         3-5
                                     53          PUT   PRTOUT,HDGLINE3            PRINT THIRD HEADING LINE
                                     54+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                         3-5
0000B8 5810 3616            00618    55+         L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000BC 5800 3622            00624    56+         L     0,=A(HDGLINE3) GET WORK AREA ADDRESS
0000C0 58F1 0010            00010    57+         L     15,16(1) GET LOGIC MODULE ADDRESS      3-5
0000C4 45EF 000C            0000C    58+         BAL   14,12(15) BRANCH TO PUT ROUTINE         3-5
0000C8 F810 35F5 3631 005F7 00633    59          ZAP   LINECNT,=P'0'              RESET LINE COUNT TO ZERO
0000CE 92F0 34CF      004D1          60          MVI   PRTDCTL,C'0'               MOVE ZERO TO DETAIL CONTROL BYTE
                                     61 PRTDET   PUT   PRTOUT,PRTDETL             PRINT DETAIL LINE
                                     62+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                         3-5
0000D2 5810 3616            00618    63+PRTDET   L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000D6 5800 3626            00628    64+         L     0,=A(PRTDETL) GET WORK AREA ADDRESS
0000DA 58F1 0010            00010    65+         L     15,16(1) GET LOGIC MODULE ADDRESS      3-5
0000DE 45EF 000C            0000C    66+         BAL   14,12(15) BRANCH TO PUT ROUTINE         3-5
0000E2 FA10 35F5 3630 005F7 00632    67          AP    LINECNT,=P'1'              ADD ONE TO LINE COUNT
0000E8 47F0 3014            00016    68          B     READCARD
```

FIGURE 4-1 (Part 2) Assembly Listing

```
                                                                                                                    PAGE    2

  LOC   OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                                                  FDOS CL3-9 12/13/73

0000EC DE06 3555 35F2 00557 005F4    69 CRDEOF   ED    CNTPATRN,COUNT          EDIT COUNT
                                     70          PUT   PRTOUT,CNTLINE          PRINT COUNT LINE
                                     71+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                          3-5
0000F2 5810 3616             00618   72+         L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000F6 5800 362A             0062C   73+         L     0,=A(CNTLINE) GET WORK AREA ADDRESS
0000FA 58F1 0010             00010   74+         L     15,16(1) GET LOGIC MODULE ADDRESS      3-5
0000FE 45EF 000C             0000C   75+         BAL   14,12(15) BRANCH TO PUT ROUTINE        3-5
                                     76          CLOSE CARDIN,PRTOUT
                                     77+* 360N-CL-453 CLOSE     CHANGE LEVEL 3-3                           3-3
000102 0700                          78+         CNOP  0,4
000104                               79+         DC    0F'0'
000104 4110 3606             00608   80+         LA    1,=C'$$BCLOSE'
000108 4500 3112             00114   81+IJJC0008 BAL   0,*+4+4*(3-1)
00010C 00000000                      82+         DC    A(CARDIN)
         *** ERROR ***
000110 00000118                      83+         DC    A(PRTOUT)
000114 0A02                          84+         SVC   2
                                     85          EOJ
                                     86+* 360N-CL-453 EOJ       CHANGE LEVEL 3-0
000116 0A0E                          87+         SVC   14
                                     88 * THE CARD FILE DEFINITION FOLLOWS
                                     89 CARDIN   DTFCR DEVADDR=SYSIPT,IOAREA1=CRDIO1,IOAREA2=CRDIO2,          X
                                                       EOFADDR=CRDEOF,WORKA=YES
          *** ERROR ***
                                     90 * THE PRINTER FILE DEFINITION FOLLOWS
                                     91 PRTOUT   DTFPR DEVADDR=SYSLST,                                        X
                                                       IOAREA1=PRTIO1,                                        X
                                                       IOAREA2=PRTIO2,                                        X
                                                       WORKA=YES,                                             X
                                                       BLKSIZE=133,                                           X
                                                       CTLCHR=ASA
                                     92+* 360 N-CL-453 DTFPR       CHANGE LEVEL 3-9                         3-9
000118                               93+         DC    0D'0'
000118 000080000000                  94+PRTOUT   DC    X'000080000000' RES. COUNT, COM. BYTES, STATUS BYTES 3-9
00011E 00                            95+         DC    AL1(0) LOGICAL UNIT CLASS
00011F 03                            96+         DC    AL1(3) LOGICAL UNIT
000120 00000140                      97+         DC    A(*+32) CCW  ADDR.
000124 00000000                      98+         DC    4X'00' CCB-ST BYTE,CSW CCW ADDRESS
000128 00                            99+         DC    AL1(0) 3-9
000129 000000                       100+         DC    VL3(IJDFAZIW) ADDR OF LOGIC MODUL3-8
00012C 08                           101+         DC    X'08' DTF TYPE  (PRINTER)
00012D 36                           102+         DC    AL1(54) SWITCHES
00012E 09                           103+         DC    X'09' NORMAL  COMM. CODE
00012F 09                           104+         DC    X'09' CONTROL COMM. CODE
000130 000001E9                     105+         DC    A(PRTIO1+1) ADDRESS OF DATA IN IOAREA1
000134 00000000                     106+         DC    4X'00' BUCKET                          3-5
000138 0700                         107+         NOPR  0 PUT LENGTH IN REG12 (ONLY UNDEF.
00013A 4700 0000             00000  108+         NOP   0 LOAD USER POINTER REG
00013E 0000                         109+         DC    2X'00' NOT USED                        3-5
000140 0900026E20000084             110+         CCW   9,PRTIO2+1,X'20',133-1
000148                              111+IJJZ0010 EQU   *
                                    112 * THE DATA DEFINITIONS FOR THE TWO CARD I/O AREAS FOLLOW
000148                              113 CRDIO1   DS    CL80
000198                              114 CRDIO2   DS    CL80
                                    115 * THE DATA DEFINITIONS FOR THE TWO PRINTER AREAS FOLLOW
0001E8                              116 PRTIO1   DS    CL133
00026D                              117 PRTIO2   DS    CL133
                                    118 * THE DATA DEFINITIONS FOR THE CARD WORK AREA FOLLOW
0002F2                              119 CRDWRKA  DS    0CL80
0002F2                              120 CRDITNBR DS    CL5
0002F7                              121 CRDITDES DS    CL20
00030B                              122          DS    CL5
000310                              123 CRDPRICE DS    CL5
000315                              124 CRDORDPT DS    CL5
00031A                              125 CRDONHND DS    CL5
00031F                              126 CRDONORD DS    CL5
000324                              127          DS    CL30
                                    128 * THE DATA DEFINITIONS FOR THE PRINTER HEADING LINES FOLLOW
000342                              129 HDGLINE1 DS    0CL133
000342 F1                           130          DC    C'1'
000343 4040404040404040             131          DC    24C' '
```

FIGURE 4-1 (Part 3) Assembly Listing (Continued)

```
                                                                                                                    PAGE    3

  LOC   OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                                                    FDOS CL3-9 12/13/73

00035B D9C5D6D9C4C5D940              132            DC    C'REORDER LISTING'
00036A 4040404040404040              133            DC    93C' '
0003C7                               134 HDGLINE2   DS    0CL133
0003C7 F0                            135            DC    C'0'
                                     136            DC    C' ITEM              ITEM                  UNIT              X
0003C8 40C9E3C5D4404040                                      REORDER'
000407 4040404040404040              137            DC    69C' '
00044C                               138 HDGLINE3   DS    0CL133
00044C 40                            139            DC    C' '
                                     140            DC    C'  NO.          DESCRIPTION            PRICE    AVAILABLEX
00044D 4040D5D64B404040                                      POINT'
000488 4040404040404040              141            DC    70C' '
                                     142 * THE DATA DEFINITIONS FOR THE PRINTER DETAIL LINE FOLLOW
0004D1                               143 PRTDETL    DS    0CL133
0004D1                               144 PRTDCTL    DS    CL1
0004D2                               145 PRTITNBR   DS    CL6
0004D8 4040404040                    146            DC    5C' '
0004DD 40404040                      147            DC    4C' '
0004E1                               148 PRTITDES   DS    CL20
0004F5                               149 PRTPRICE   DS    CL7
0004FC 40404040                      150            DC    4C' '
000500                               151 PRTAVAIL   DS    CL6
000506 40404040                      152            DC    4C' '
00050A                               153 PRTORDPT   DS    CL6
000510 4040404040404040              154            DC    70C' '
                                     155 * THE DATA DEFINITIONS FOR THE FINAL TOTAL LINE FOLLOW
000556                               156 CNTLINE    DS    0CL133
000556 60                            157            DC    C'-'
000557 4020206B202020                158 CNTPATRN   DC    X'4020206B202020'
00055E 40C3C1D9C4E240C9              159            DC    C' CARDS IN THE INPUT DECK'
000576 4040404040404040              160            DC    101C' '
                                     161 * THE DATA DEFINITIONS THAT FOLLOW DEFINE OTHER WORK AREAS NEEDED
                                     162 * BY THE PROGRAM
0005DB 402020202020                  163 PATTERN1   DC    X'402020202020'
0005E1 40202021 4B2020               164 PATTERN2   DC    X'40202021 4B2020'
0005E8                               165 WRKAVAIL   DS    PL3
0005EB                               166 WRKONORD   DS    PL3
0005EE                               167 WRKORDPT   DS    PL3
0005F1                               168 PACKAREA   DS    PL3
0005F4 00000C                        169 COUNT      DC    PL3'0'
0005F7 050C                          170 LINECNT    DC    P'50'
000000                               171            END   BEGIN
000600 5B5BC2D6D7C5D540              172                  =C'$$BOPEN '
000608 5B5BC2C3D3D6E2C5              173                  =C'$$BCLOSE'
000610 000C0000                      174                  =A(CARDIN)
         *** ERROR ***
000614 00000000                      175                  =A(CARDWRKA)
         *** ERROR ***
000618 00000118                      176                  =A(PRTOUT)
00061C 00000342                      177                  =A(HDGLINE1)
000620 000003C7                      178                  =A(HDGLINE2)
000624 0000044C                      179                  =A(HDGLINE3)
000628 000004D1                      180                  =A(PRTDETL)
00062C 00000556                      181                  =A(CNTLINE)
000630 050C                          182                  =P'50'
000632 1C                            183                  =P'1'
000633 0C                            184                  =P'0'
```

FIGURE 4-1 (Part 4) Assembly Listing (Continued)

```
                                    RELOCATION DICTIONARY                    PAGE    1

POS.ID    REL.ID    FLAGS    ADDRESS                                         12/13/73

  01        01        0C      000010
  01        01        0C      000110
  01        01        0C      000120
  01        02        18      000129
  01        01        0C      000130
  01        01        08      000141
  01        01        0C      000618
  01        01        0C      00061C
  01        01        0C      000620
  01        01        0C      000624
  01        01        0C      000628
  01        01        0C      00062C
```

FIGURE 4-1 (Part 5) Relocation Dictionary

```
                                    CROSS-REFERENCE                          PAGE    1

SYMBOL     LEN   VALUE  DEFN     REFERENCES                                   12/13/73

BEGIN     00002 000000 00002    0171
CARDIN    ****UNDEFINED*****    0010  0015  0082  0174
CARDWRKA  ****UNDEFINED*****    0016  0175
CNTLINE   00133 000556 00156    0073  0181
CNTPATRN  00007 000557 00158    0069
COUNT     00003 0005F4 00169    0019  0069
CRDEOF    00006 0000EC 00069
CRDIO1    00080 000148 00113
CRDIO2    00080 000198 00114
CRDITDES  00020 0002F7 00121    0031
CRDITNBR  00005 0002F2 00120    0028
CRDONHND  00005 00031A 00125    0020
CRDONORD  00005 00031F 00126    0021
CRDORDPT  00005 000315 00124    0023
CRDPRICE  00005 000310 00123    0032
CRDWRKA   00080 0002F2 00119
HDGLINE1  00133 000342 00129    0044  0177
HDGLINE2  00133 0003C7 00134    0050  0178
HDGLINE3  00133 00044C 00138    0056  0179
IJJC0008  00004 000108 00081
IJJ00001  00004 000008 00009
IJJZ0010  00001 000148 00111
LINECNT   00002 0005F7 00170    0039  0059  0067
PACKAREA  00003 0005F1 00168    0028  0030  0032  0034
PATTERN1  00006 0005DB 00163    0035  0037
PATTERN2  00007 0005E1 00164    0033
PRTAVAIL  00006 000500 00151    0035  0036
PRTDCTL   00001 0004D1 00144    0060
PRTDET    00004 0000D2 00063    0040
PRTDETL   00133 0004D1 00143    0026  0027  0027  0064  0180
PRTIO1    00133 0001E8 00116    0105
PRTIO2    00133 00026D 00117    0110
PRTITDES  00020 0004E1 00148    0031
PRTITNBR  00006 0004D2 00145    0030
PRTORDPT  00006 00050A 00153    0037  0038
PRTOUT    00006 000118 00094    0011  0043  0049  0055  0063  0072  0083  0176
PRTPRICE  00007 0004F5 00149    0033  0034
READCARD  00004 000016 00015    0025  0068
REORDLST  00001 000000 00001
WRKAVAIL  00003 0005E8 00165    0020  0024  0036
WRKONORD  00003 0005EB 00166    0021
WRKORDPT  00003 0005EE 00167    0023  0024  0038
```

FIGURE 4-1 (Part 6) Cross-Reference Listing

```
                                        DIAGNOSTICS                          PAGE    1

 STMT  ERROR CODE   MESSAGE                                                   12/13/73

   10  IJY024       NEAR OPERAND COLUMN   3--UNDEFINED SYMBOL
   22  IJY088       UNDEFINED OPERATION CODE
   29  IJY088       UNDEFINED OPERATION CODE
   82  IJY024       NEAR OPERAND COLUMN   3--UNDEFINED SYMBOL
   89  IJY088       UNDEFINED OPERATION CODE
  174  IJY024       NEAR OPERAND COLUMN   4--UNDEFINED SYMBOL
  175  IJY024       NEAR OPERAND COLUMN   4--UNDEFINED SYMBOL


    7 STATEMENTS FLAGGED IN THIS ASSEMBLY
```

FIGURE 4-1 (Part 7) Diagnostic Listing

The leftmost column of information is headed LOC, which stands for *location counter*. The location counter is a field that is used during assembly to keep track of the starting location of each machine instruction or data field of the program. For the first machine instruction, BALR, the location is the value given in the START instruction—in this case, zero. For each subsequent instruction or data definition, the location is determined by adding the length of the instruction or field to the previous location counter value. Since BALR is a two-byte instruction, the second machine instruction has a location of 000002.

Notice that the location values are given in hex. If you scan the LOC column, you can see that the increments from instruction to instruction are two, four, or six bytes. These, of course, are the lengths of the System/360-370 instructions. If a source line doesn't generate any object code, the location counter value is not increased.

For the data definitions, the location counter is increased by the total number of bytes reserved—that is, the duplication factor times the length attribute. As an interesting example, look at statements 119 and 120;

```
CRDWRKA  DS    0CL80
CRDITNBR DS    CL5
```

Because the duplication factor is zero in statement 119, both CRDWRKA and CRDITNBR have the same location counter value, hex 2F2. Thus the contents of card column 1 can be addressed by either label although CRDWRKA will have a length attribute of 80 and CRDITNBR of 5.

The second column from the left is headed OBJECT CODE. This column shows the actual machine language assembled for a statement just as it will be stored in the computer. It is in hex notation with two hex digits representing one byte of storage. If a statement causes no object code to be generated, this column is blank.

For instructions, there are two, four, or six bytes represented in the object code column. The format of this data corresponds to the instruction formats presented in chapter 2. The BALR statement, for instance, is translated into two bytes of object code, hex 0530. The first byte is the operation code, hex 05; the second byte holds two register numbers, 3 and 0. Similarly, the MVC instruction in statement 31 is translated into hex D21334D932F5. By referring to the MVC instruction format, you can determine that D2 is the operation code, hex 13 is the length, address 1 consists of a base address in register 3 plus a displacement of hex 4D9, and so on.

For DC statements, the object code column usually shows only the first eight bytes of the field. If you want all of the defined data to print, you can use a PRINT assembler command which is described in chapter 11. Remember that DS statements do not cause object code to be created even if they are given nominal values.

You might notice that ***ERROR*** is printed in the

object code column when the assembler cannot create object code due to a diagnostic error. Later on, the diagnostics refer to these statements so the errors can be corrected.

The ADDR1 and ADDR2 columns give the location counter values for operands 1 and 2 of each instruction. If the operand is a register or if no operand is present, the appropriate column is left blank. As you will see, the location values are helpful when you are debugging.

Part 6 of figure 4-1 is the *cross-reference listing*. In the left column of information (SYMBOL), the labels of all instructions or fields used in the program are listed in alphabetic order. The second column (LEN) gives the length attribute of each label. Thus, BEGIN, which is the label of the BALR instruction, has a length attribute of 2; CRDITDES has a length of 20. The VALUE column gives the location counter value assigned to the label and the DEFN column gives the number of the statement that defines the label. Finally, in the REFERENCES column are listed the numbers of all statements that use the label as an operand. If necessary, multiple lines are used for the reference numbers. As you will see, this listing is a handy reference to use when correcting diagnostics or debugging.

The last page of the assembly listing is the *diagnostic listing*. For each diagnostic message, the statement number of the error statement is given along with a code for the type of error and an error message. The error code, like IJY088, refers to an expanded explanation of the message that can be found in IBM's *DOS Assembler Language* manual, number GC24-3414. In general, however, the messages on the assembly listing tell you all you need to know so you should rarely need this manual.

CORRECTING DIAGNOSTICS

The assembler's ability to find errors is limited. It can find invalid operation codes, invalid formats for operands, undefined labels, and missing commas, but it cannot detect errors in the flow of a program. Since each statement is examined individually by the assembler, it is possible to write a series of statements that assemble without error but that make no sense at all as far as the function of a program is concerned.

You should also know that the assembler does not require strict alignment of statement parts even though the coding form indicates that operation codes should begin in column 10 and operands in column 16. Although the label must start in column 1, the operation code can start anywhere as long as one or more blanks separate it from the label. Similarly, the operands can start anywhere as long as one or more blanks separate them from the operation code. If a statement doesn't have a label, the operation code can start anywhere after column 1. Of course, the sequence from left to right in the statement is critical; it must be label, operation code, and operands in that order.

There are many diagnostic messages that can be produced by the assembler. The majority of errors, however, are covered by just a few messages. In figures 4-1 and 4-2, examples of the most common types of errors are given.

In figure 4-1, there are six diagnostic messages that all relate to invalid operation codes. The second diagnostic message, for example, is UNDEFINED OPERATION CODE and refers to statement 22. Since the invalid ADD rather than the valid AP was used for the operation code, the assembler was unable to create the object code and the diagnostic was printed. Similarly, the third diagnostic was caused by the invalid operation code MVL (probably a keypunching error) instead of the intended MVC.

The fifth diagnostic is for an undefined operation code in statement 89. Here, DTFCR was used when DTFCD would have been correct. Because of this error, the label of the DTFCD, CARDIN, is not considered to be defined. This is shown in the cross-reference listing.

The latter error accounts for the other three diagnostic messages, all of which refer to an undefined symbol. Two of these diagnostics refer to instructions generated by the OPEN and CLOSE macros—statements 10 and 82. The third diagnostic refers to a literal definition generated by the GET macro—statement 174: =A(CARDIN). This refers to the address of CARDIN, which cannot be determined due to the error in the DTF operation code.

```
                                                                                                    PAGE    1

  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                                  FDOS CL3-9 12/13/73

000000                                  1 REORDLST START 0
000000 0530                             2 BEGIN    BALR  3,0                     LOAD BASE REGISTER
000002                                  3          USING *,3
                                        4          OPEN  CARDIN,PRTOUT
                                        5+* 360N-CL-453 OPEN     CHANGE LEVEL 3-3                          3-3
000002 0700                             6+         CNOP  0,4
000004                                  7+         DC    0F'0'
000004 4110 3636              00638     8+         LA    1,=C'$$BOPEN '
000008 4500 3012              00014     9+IJJ00001 BAL   0,*+4+4*(3-1)
00000C 00000128                        10+         DC    A(CARDIN)
000010 00000160                        11+         DC    A(PRTOUT)
000014 0A02                            12+         SVC   2
                                       13 READCARD GET   CARDIN,CARDWRKA         READ CARD INTO WORK AREA
                                       14+* 360N-CL-453 GET      CHANGE LEVEL 3-0
000016 5810 3646              00648    15+READCARD L     1,=A(CARDIN) GET DTF TABLE ADDRESS
00001A 5800 364A              0064C    16+         L     0,=A(CARDWRKA) GET WORK AREA ADDRESS
00001E 58F1 0010              00010    17+         L     15,16(1) GET LOGIC MODULE ADDRESS
000022 45EF 0008              00008    18+         BAL   14,8(15) BRANCH TO GET ROUTINE
000026 FA20 3632 3668 00634 0066A      19          AP    COUNT,=P'1'             ADD ONE TO COUNT
00002C F224 3629 335B 0062B 0035D      20          PACK  WRKAVAIL,CRDONHND
000032 F224 362C 3360 0062E 00362      21          PACK  WRKONORD,CRDONORD
000038 FA22 3629 362C 0062B 0062E      22          AP    WRKAVAIL,WRKONORD       ADD ON HAND AND ON ORDER
00003E F204 362F 3356 00631 00358      23          PACK  WRKORDPT,CRDORDPT
000044 F920 3629 362F 0062B 00631      24          CP    WRKAVAIL,WRKORDPT
00004A 0000 0000              00000    25          BNL   READCRD
         *** ERROR ***
00004E 9240 3512        00514          26          MVI   PRTDETL,X'40'           MOVE BLANK TO CONTROL BYTE
000052 D283 3513 3512 00515 00514      27          MVC   PRTDETL+1(132),PRTDETL  BLANK PRINTER WORK AREA
000058 F224 362F 3338 00631 0033A      28          PACK  PACKAREA,CRDITNBR
00005E D205 3513 361C 00515 0061E      29          MVC   PRTITNBR,PATTERN1
000064 DE05 3513 362F 00515 00631      30          ED    PRTITNBR,PACKAREA       EDIT ITEM NUMBER FIELD
00006A D213 351E 333D 00520 0033F      31          MVC   PRTITDES,CRDITDES       MOVE ITEM DESCRIPTION
000070 F224 362F 3351 00631 00353      32          PACK  PACKAREA,CRDPRICE
000076 D206 3536 3622 00538 00624      33          MVC   PRTPRICE,PATTERN2
00007C DE06 3536 362F 00538 00631      34          ED    PRTPRICE,PACKAREA       EDIT UNIT PRICE
000082 0000 0000 0000 00000 00000      35          MVC   PRTAVAIL, PATTERN1
         *** ERROR ***
000088 DE05 3541 3629 00543 0062B      36          ED    PRTAVAIL,WRKAVAIL       EDIT AVAILABLE
00008E D205 354B 361C 0054D 0061E      37          MVC   PRTORDPT,PATTERN1
000094 DE05 354B 362F 0054D 00631      38          ED    PRTORDPT,WRKORDPT       EDIT ORDER POINT
00009A F901 3635 3666 00637 00668      39          CP    LINECNT,=P'50'          COMPARE LINE COUNT TO 50
0000A0 4740 30DC              000DE    40          BL    PRTDET                  BRANCH ON LOW TO PRTDET
                                       41          PUT   PRTOUT,HDGLINE1         PRINT FIRST HEADING LINE
                                       42+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                      3-5
0000A4 5810 364E              00650    43+         L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000A8 5800 3652              00654    44+         L     0,=A(HDGLINE1) GET WORK AREA ADDRESS
0000AC 58F1 0010              00010    45+         L     15,16(1) GET LOGIC MODULE ADDRESS     3-5
0000B0 45EF 000C              0000C    46+         BAL   14,12(15) BRANCH TO PUT ROJTINE         3-5
                                       47          PUT   PRTOUT,HDGLINE2         PRINT SECOND HEADING LINE
                                       48+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                      3-5
0000B4 5810 364E              00650    49+         L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000B8 5800 3656              00658    50+         L     0,=A(HDGLINE2) GET WORK AREA ADDRESS
0000BC 58F1 0010              00010    51+         L     15,16(1) GET LOGIC MODULE ADDRESS     3-5
0000C0 45EF 000C              0000C    52+         BAL   14,12(15) BRANCH TO PUT ROJTINE         3-5
                                       53          PUT   PRTOUT,HDGLINE3         PRINT THIRD HEADING LINE
                                       54+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                      3-5
0000C4 5810 364E              00650    55+         L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000C8 5800 365A              0065C    56+         L     0,=A(HDGLINE3) GET WORK AREA ADDRESS
0000CC 58F1 0010              00010    57+         L     15,16(1) GET LOGIC MODULE ADDRESS     3-5
0000D0 45EF 000C              0000C    58+         BAL   14,12(15) BRANCH TO PUT ROUTINE         3-5
0000D4 00C0 0000 0000 00000 00000      59          ZAP   LINECNT,P'0'
         *** ERROR ***
0000DA 92F0 3512        00514          60          MVI   PRTDCTL,C'0'            MOVE ZERO TO DETAIL CONTROL BYTE
                                       61 PRTDET   PUT   PRTOUT,PRTDETL          PRINT DETAIL LINE
                                       62+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                      3-5
0000DE 5810 364E              00650    63+PRTDET   L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000E2 5800 365E              00660    64+         L     0,=A(PRTDETL) GET WORK AREA ADDRESS
0000E6 58F1 0010              00010    65+         L     15,16(1) GET LOGIC MODULE ADDRESS     3-5
0000EA 45EF 000C              0000C    66+         BAL   14,12(15) BRANCH TO PUT ROJTINE         3-5
0000EE FA00 3635 3668 00637 0066A      67          AP    LINECNT,=P'1'           ADD ONE TO LINE COUNT
0000F4 47F0 3014              00016    68          B     READCARD
```

FIGURE 4-2 Assembly Output

```
                                                                                                PAGE    2

  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                                    FDOS CL3-9 12/13/73

0000F8 DE06 3598 3632 0059A 00634    69 CRDEOF    ED     CNTPATRN,COUNT              EDIT COUNT
                                     70           PUT    PRTOUT,CNTLINE              PRINT COUNT LINE
                                     71+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                     3-5
0000FE 5810 364E            00650    72+          L      1,=A(PRTOUT) GET DTF TABLE ADDRESS
000102 5800 3662            00664    73+          L      0,=A(CNTLINE) GET WORK AREA ADDRESS
000106 58F1 0010            00010    74+          L      15,16(1) GET LOGIC MODULE ADDRESS      3-5
00010A 45EF 000C            0000C    75+          BAL    14,12(15) BRANCH TO PUT ROJTINE          3-5
                                     76           CLOSE  CARDIN,PRTOUT
                                     77+* 360N-CL-453 CLOSE    CHANGE LEVEL 3-3                     3-3
00010E 0700                          78+          CNOP   0,4
000110                               79+          DC     0F'0'
000110 4110 363E            00640    80+          LA     1,=C'$$BCLOSE'
000114 4500 311E            00120    81+IJJC0008 BAL     0,*+4+4*(3-1)
000118 00000128                      82+          DC     A(CARDIN)
00011C 00000160                      83+          DC     A(PRTOUT)
000120 0A02                          84+          SVC    2
                                     85           EOJ
                                     86+* 360N-CL-453 EOJ      CHANGE LEVEL 3-0
000122 0A0E                          87+          SVC    14
                                     88 * THE CARD FILE DEFINITION FOLLOWS
                                     89 CARDIN    DTFCD DEVADDR=SYSIPT,IOAREA1=CRDIO1,IOAREA2=CRDIO2,          X
                                                         EOFADDR=CRDEOF WORKA=YES
                                     90+* 360N-CL-453   DTFCD    CHANGE LEVEL 3-10                    3-10
                                     91                  0,NEITHER IOREG NOR WORKA STATED FOR A 2 IOAREAS FILE.  X
                                                         WORKA ASSUMED
000124 00000000
000128                               92+          DC     0D'0'
000128 000080000000                  93+CARDIN    DC     X'000080000000' RES. COUNT,COM. BYTES,STATUS BTS
00012E 00                            94+          DC     AL1(0) LOGICAL UNIT CLASS
00012F 01                            95+          DC     AL1(1) LOGICAL UNIT
000130 00000148                      96+          DC     A(IJCX0010) CCW ADDRESS
000134 C0000000                      97+          DC     4X'00' CCB-ST BYTE,CSW CCW ADDR.
000138 00                            98+          DC     AL1(0) 3-3
000139 000000                        99+          DC     VL3(IJCFZIB0) ADDRESS OF LOGIC MODULE    3-3
00013C 02                           100+          DC     X'02' DTF TYPE (READER)
00013D 05                           101+          DC     AL1(5) SWITCHES
00013E 02                           102+          DC     AL1(2) NORMAL COMM.CODE
00013F 02                           103+          DC     AL1(2) CNTROL COMM.CODE
000140 000001E0                     104+          DC     A(CRDIO2) ADDR. OF IOAREA2
000144 000000F8                     105+          DC     A(CRDEOF) EOF ADDRESS                      3-8
000148 0200019020000050             106+IJCX0010 CCW     2,CRDIO1,X'20',80
000150 4700 0000            00000   107+          NOP    0 LOAD USER POINTER REG.
000154 D24F D000 E000 00000 00000   108+          MVC    0(80,13),0(14) MOVE IOAREA TO WORKA
00015A                              109+IJJZ0010 EQU     *
                                    110 * THE PRINTER FILE DEFINITION FOLLOWS
                                    111 PRTOUT    DTFPR DEVADDR=SYSLST,                                         X
                                                        IOAREA2=PRTIO2,                                         X
                                                        WORKA=YES,                                              X
                                                        BLKSIZE=133,                                            X
                                                        CTLCHR=ASA
                                    112+* 360 N-CL-453 DTFPR       CHANGE LEVEL 3-9                    3-9
                                    113                 0,NO IOAREA1 SPECIFIED. SET TO *
00015A 000000000000
000160                              114+          DC     0D'0'
000160 000080000000                 115+PRTOUT    DC     X'000080000000' RES. COUNT, COM. BYTES, STATUS BYTES 3-9
000166 00                           116+          DC     AL1(0) LOGICAL UNIT CLASS
000167 03                           117+          DC     AL1(3) LOGICAL UNIT
000168 00000188                     118+          DC     A(*+32) CCW  ADDR.
00016C 00000000                     119+          DC     4X'00' CCB-ST BYTE,CSW CCW ADDRESS
000170 00                           120+          DC     AL1(0) 3-9
000171 000000                       121+          DC     VL3(IJDFAZIW) ADDR OF LOGIC MODUL3-8
000174 08                           122+          DC     X'08' DTF TYPE  (PRINTER)
000175 36                           123+          DC     AL1(54) SWITCHES
000176 09                           124+          DC     X'09' NORMAL  COMM. CODE
000177 09                           125+          DC     X'09' CONTROL COMM. CODE
000178 00000179                     126+          DC     A(*+1) ADDRESS OF DATA IN IOAREA1
00017C 00000000                     127+          DC     4X'00' BUCKET                 3-5
000180 0700                         128+          NOPR   0 PUT LENGTH IN REG12 (ONLY UNDEF.
000182 4700 0000            00000   129+          NOP    0 LOAD USER POINTER REG
000186 0000                         130+          DC     2X'00' NOT USED               3-5
000188 090002B620000084             131+          CCW    9,PRTIO2+1,X'20',133-1
```

FIGURE 4-2 Assembly Output (Continued)

```
                                                                                                                    PAGE     3

  LOC   OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                                                    FDOS CL3-9 12/13/73

000190                             132+IJJZ0011 EQU   *
                                   133 * THE DATA DEFINITIONS FOR THE TWO CARD I/O AREAS FOLLOW
000190                             134 CRDIO1   DS    CL80
0001E0                             135 CRDIO2   DS    CL80
                                   136 * THE DATA DEFINITIONS FOR THE TWO PRINTER AREAS FOLLOW
000230                             137 PRTIO1   DS    CL133
0002B5                             138 PRTIO2   DS    CL133
                                   139 * THE DATA DEFINITIONS FOR THE CARD WORK AREA FOLLOW
00033A                             140 CRDWRKA  DS    0CL80
00033A                             141 CRDITNBR DS    CL5
00033F                             142 CRDITDES DS    CL20
                                   143          DS    C5
         *** ERROR ***
000353                             144 CRDPRICE DS    CL5
000358                             145 CRDORDPT DS    CL5
00035D                             146 CRDONHND DS    CL5
000362                             147 CRDONORD DS    CL5
000367                             148          DS    CL30
                                   149 * THE DATA DEFINITIONS FOR THE PRINTER HEADING LINES FOLLOW
000385                             150 HDGLINE1 DS    0CL133
000385 F1                          151          DC    C'1'
0003B6 4040404040404040            152          DC    24C' '
0003CE D9C5D6D9C4C5D940            153          DC    C'REORDER LISTING'
0003AD 4040404040404040            154          DC    93C' '
00040A                             155 HDGLINE2 DS    0CL133
00040A F0                          156          DC    C'0'
                                   157          DC    C' ITEM              ITEM                  UNIT              X
00040E 40C9E3C5D4404040                            REORDER'
00044A 4040404040404040            158          DC    69C' '
00048F                             159 HDGLINE3 DS    0CL133
00048F 40                          160          DC    C' '
                                   161          DC    C'  NO.          DESCRIPTION            PRICE    AVAILABLEX
000490 4040D5D64B404040                            POINT'
0004CE 4040404040404040            162          DC    70C' '
                                   163 * THE DATA DEFINITIONS FOR THE PRINTER DETAIL LINE FOLLOW
000514                             164 PRTDETL  DS    0CL133
000514                             165 PRTDCTL  DS    CL1
000515                             166 PRTITNBR DS    CL6
00051B 4040404040                  167          DC    5C' '
000520                             168 PRTITDES DS    CL20
000534 40404040                    169          DC    4C' '
000538                             170 PRTPRICE DS    CL7
00053F 40404040                    171          DC    4C' '
000543                             172 PRTAVAIL DS    CL6
000549 40404040                    173          DC    4C' '
00054D                             174 PRTORDPT DS    CL6
000553 4040404040404040            175          DC    70C' '
                                   176 * THE DATA DEFINITIONS FOR THE FINAL TOTAL LINE FOLLOW
000599                             177 CNTLINE  DS    0CL133
000599 60                          178          DC    C'-'
00059A 4020206B202020              179 CNTPATRN DC    X'4020206B202020'
0005A1 40C3C1D9C4E240C9            180          DC    C' CARDS IN THE INPUT DECK'
0005B9 4040404040404040            181          DC    101C' '
                                   182 * THE DATA DEFINITIONS THAT FOLLOW DEFINE OTHER WORK AREAS NEEDED
                                   183 * BY THE PROGRAM
00061E 402020202020                184 PATTERN1 DC    X'402020202020'
000624 402020214B2020              185 PATTERN2 DC    X'402020214B2020'
00062E                             186 WRKAVAIL DS    PL3
00062E                             187 WRKONORD DS    PL3
                                   188 WRKORDPT DC    PL3
         *** ERROR ***
000631                             189 PACKAREA DS    PL3
000634 00000C                      190 COUNT    DC    PL3'0'
                                   191 LINECNT  DC    '50'
         *** ERROR ***
000000                             192          END   BEGIN
000638 5B5BC2D6D7C5D540            193                =C'$$BOPEN '
000640 5B5BC2C3D3D6E2C5            194                =C'$$BCLOSE'
000648 00000128                    195                =A(CARDIN)
00064C 000C0000                    196                =A(CARDWRKA)
         *** ERROR ***
```

FIGURE 4-2 Assembly Output (Continued)

PAGE 4

FDOS CL3-9 12/13/73

LOC	OBJECT CODE	ADDR1	ADDR2	STMT	SOURCE STATEMENT
000650	00000160			197	=A(PRTOUT)
000654	00000385			198	=A(HDGLINE1)
000658	0000040A			199	=A(HDGLINE2)
00065C	0000048F			200	=A(HDGLINE3)
000660	00000514			201	=A(PRTDETL)
000664	00000599			202	=A(CNTLINE)
000668	050C			203	=P'50'
00066A	1C			204	=P'1'

FIGURE 4-2 Assembly Output (Continued)

CROSS-REFERENCE

PAGE 1

12/13/73

SYMBOL	LEN	VALUE	DEFN	REFERENCES							
BEGIN	00002	000000	00002	0192							
CARDIN	00006	000128	00093	0010	0015	0082	0195				
CARDWRKA	****UNDEFINED*****			0016	0196						
CNTLINE	00133	000599	00177	0073	0202						
CNTPATRN	00007	00059A	00179	0069							
COUNT	00003	000634	00190	0019	0069						
CRDEOF	00006	0000F8	00069	0105							
CRDIO1	00080	000190	00134	0106							
CRDIO2	00080	0001E0	00135	0104							
CRDITDES	00020	00033F	00142	0031							
CRDITNBR	00005	00033A	00141	0028							
CRDONHND	00005	00035D	00146	0020							
CRDONORD	00005	000362	00147	0021							
CRDORDPT	00005	000358	00145	0023							
CRDPRICE	00005	000353	00144	0032							
CRDWRKA	00080	00033A	00140								
HDGLINE1	00133	000385	00150	0044	0198						
HDGLINE2	00133	00040A	00155	0050	0199						
HDGLINE3	00133	00048F	00159	0056	0200						
IJCX0010	00008	000148	00106	0096							
IJJC0008	00004	000114	00081								
IJJO0001	00004	000008	00009								
IJJZ0010	00001	00015A	00109								
IJJZ0011	00001	000190	00132								
LINECNT	00001	000637	00191	0039	0059	0067					
PACKAREA	00003	000631	00189	0028	0030	0032	0034				
PATTERN1	00006	00061E	00184	0029	0037						
PATTERN2	00007	000624	00185	0033							
PRTAVAIL	00006	000543	00172	0035	0036						
PRTDCTL	00001	000514	00165	0060							
PRTDET	00004	0000DE	00063	0040							
PRTDETL	00133	000514	00164	0026	0027	0027	0064	0201			
PRTIO1	00133	000230	00137								
PRTIO2	00133	0002B5	00138	0131							
PRTITDES	00020	000520	00168	0031							
PRTITNBR	00006	000515	00166	0029	0030						
PRTORDPT	00006	00054D	00174	0037	0038						
PRTOUT	00006	000160	00115	0011	0043	0049	0055	0063	0072	0083	0197
PRTPRICE	00007	000538	00170	0033	0034						
READCARD	00004	000016	00015	0068							
READCRD	****UNDEFINED*****			0025							
REORDLST	00001	000000	00001								
WRKAVAIL	00003	00062B	00186	0020	0022	0024	0036				
WRKONORD	00003	00062E	00187	0021	0022						
WRKORDPT	00001	000631	00188	0023	0024	0038					

FIGURE 4-2 Assembly Output (Continued)

```
                                        DIAGNOSTICS                                    PAGE    1

                                                                                       12/13/73

STMT  ERROR CODE   MESSAGE

  25   IJY024      NEAR OPERAND COLUMN    1--UNDEFINED SYMBOL
  35   IJY039      NEAR OPERAND COLUMN   10--INVALID DELIMITER
  59   IJY085      NEAR OPERAND COLUMN    9--INVALID SYNTAX IN EXPRESSION
  91   IJY037      MNOTE STATEMENT
 113   IJY037      MNOTE STATEMENT
 143   IJY107      NEAR OPERAND COLUMN    2--INVALID OPERAND
 188   IJY107      NEAR OPERAND COLUMN    4--INVALID OPERAND
 191   IJY031      NEAR OPERAND COLUMN    1--UNKNOWN TYPE
 196   IJY024      NEAR OPERAND COLUMN    4--UNDEFINED SYMBOL


    9 STATEMENTS FLAGGED IN THIS ASSEMBLY
```

FIGURE 4-2 Assembly Output (Continued)

Figure 4-2 illustrates some other common error messages. The diagnostic for statement 34, for example, indicates INVALID DELIMITER. A delimiter is a character, normally a comma or parenthesis, that is used to separate operands. In the case of statement 35, a comma is present, but it is followed by a blank instead of the next operand. The statement should be corrected by omitting the blank.

The diagnostic for statement 59 indicates INVALID SYNTAX IN EXPRESSION. The word *syntax* refers to the arrangement of words within a sentence, so syntax in programming terminology refers to the number of operands, the way they are expressed, and so on. In this case, the second operand of the ZAP instruction should be either a label or a literal. Since the equals sign is missing before the intended literal, the assembler detected invalid syntax.

The diagnostic for statement 188 indicates INVALID OPERAND. Actually, however, it's the operation code that's wrong. It should be DS instead of DC. Because the operand was coded without a nominal value, the assembler assumed the operand was incorrect.

The reverse of this mistake—coding DS for DC—is also common. In such a case, however, the error will not be caught by the assembler because it is acceptable to have a nominal value in a DS operand. This is a more serious error than coding DC for DS because the program is assembled but no starting value is placed in the field as it was intended to be. As a result, when the program is executed, an error will normally result. Hopefully, you will catch this kind of error when you are desk checking, but you can expect one to slip through occasionally.

The diagnostic message for statement 91 illustrates a special type of error. The MNOTE message is generated when an error is present in a macro statement. It usually means that one of the operands for the macro is missing, has an invalid value, or is in conflict with another operand. In this case, statement 91 says that two I/O areas have been given for the DTFCD so either an I/O register or work area must be used also (thus far, only work areas have been illustrated). Statement 91 also says that WORKA=YES is assumed by the assembler to be present.

The problem is that there is a blank instead of a comma between the WORKA=YES operand and the preceding operand. As a result, the assembler treats WORKA=YES as a remark. However, since the assembler has assumed WORKA=YES, the DTFCD can be left as it is—the assumption is the same as the code intended. If the program has to be reassembled to correct other errors, however, you would probably correct this MNOTE by punching a comma between the last two keyword operands of the DTFCD.

In correcting diagnostics, you should normally take them in sequence and correct operation codes and operands as

necessary. If you come to a message that you can't figure out, you can skip it at first since one of the later diagnostics may indicate its cause. If after going through all diagnostics, there are still a few you can't figure out, you might try reassembling the program with all other errors corrected. On some occasions, this will solve your problem because the problem diagnostics were caused by errors you have already corrected.

Terminology

desk checking
assembly listing
location counter
cross-reference listing
diagnostic listing
syntax

Objective

Given an assembly listing including diagnostics, correct the source code as required.

Problem

Refer to figure 4-2.

a. How should statement 25 be corrected?
b. How should statement 143 be corrected?
c. How should statement 191 be corrected?
d. How should the DTFPR, statement 111, be corrected? (See MNOTE statement 113.)

Solution

a. The operand should be READCARD.
b. The operand should be CL5.
c. The operand should be P'50' or PL2'50'.
d. IOAREA1=PRTIO1 should be added to the operand list.

TOPIC TWO Testing and Debugging Testing and debugging refer to the process of executing the object program to see if it works, finding any errors in the program and correcting them, and then trying the program again. One of the major pitfalls in the process of programming is inadequate testing. In other words, a program is not tried on enough combinations of input data to be certain all of its routines work. Nevertheless, the program is considered to be tested and is put into production. Then when actual data causes meaningless output to be produced or program cancellation, a crisis often occurs. Can you imagine the scurrying that goes on when a program that prints payroll checks is cancelled prematurely? Or when a payroll check for double the amount owed to an employee is discovered?

CREATING TEST DATA

To guard against inadequate testing, a program is often tested at several levels. First, a program is tested on a small volume of data that is designed for testing the major functional aspects of the program. If that test is successful, the program is tested on a larger volume of data designed to test all conditions that may occur during actual processing. If a program is large, there may be several low-volume tests and several high-volume tests. After the program is successful with all of this test data, the program is often tested in conjunction with other programs. This latter test is often referred to as a *system test.*

The purpose behind a system test is to discover any misunderstandings there may be between the programmers involved. For example, one programmer may think an output file is supposed to be in one format, while another programmer whose program uses the output file as an input file expects it to be in another format. As a result, based on the results of their own test data, both programmers may consider their programs adequately tested when actually one or both programs are in error. The final phase of the system test may involve live data from portions of the system that are already in production.

Test data may be available to a programmer when the program is assigned, but in many cases the programmer must prepare at least some of the test data himself—for example, the low-volume test data. To prepare test data a programmer

should use guidelines similar to these:

1 Outline each of the conditions that must be tested. Start with the simplest and proceed to the most complex.
2 Create test data that will test all conditions that you have outlined. This data should test minimum and maximum values for input and output fields as well as the branching logic.
3 To make testing more manageable, break up the test data into a series of test runs ranging from the simple to the complex. For instance, start with test data that will test the major logic flow of the program and show isolated effects of the program; then go on to test data that will test minor conditions and show combination effects; finally test all input data in one run.
4 Within a single test run, you may also want to arrange the conditions in sequence, from simple to complex. That way some of the simple conditions may get tested even if the program is cancelled due to an error discovered when handling one of the more complex conditions.
5 Be careful to design the test cases with carefully controlled variations. If you change too much from one test run to the next, you may have trouble figuring out what caused one test to run and the next to fail.

How would you create test data for the reorder-listing program? First, you should decide on the conditions that must be tested. Since there is only one decision point other than for page overflow, there are only three major conditions to test: available equals reorder point, available less than reorder point, and available greater than reorder point. As a minor condition, page overflow should also be tested.

Second, you should create data for these conditions. You will need only 3 cards for the three conditions named, but you will need more than 50 cards to test the page-overflow logic. Furthermore, it is usually best to test page-overflow routines with enough data to produce more than two pages of output. This will catch coding that works for the first page overflow, but not for the second. To get enough data to be able to test overflow, you can duplicate each basic data card as many times as you need to.

Third, since this program treats each record individually and has only one processing path, I would propose only two test runs. The first would consist of half a dozen cards to prove the basic functions; the second would have enough cards to test page overflow. Because of the simplicity of this particular program, however, you may well choose to combine the test runs into a single run.

After a test run has been made, you compare the actual output of the test run with the expected output. If the program involves tape or disk input or output, appropriate listings of the input and output files must be made before and after the test run. In a disk-update run, for example, the contents of the disk file must be printed before testing and after testing to see what changes were made in the file. For this type of test, you may have to write programs to list the file contents or you may use *utility programs* that are supplied with the Disk Operating System. These general programs that can be used for listing files are described in chapter 16. In any event, if the actual output disagrees with the expected output, you must find the cause of the error, change the source code, and test again.

After a test run, you should document it. The documentation for each test run should consist of:

1 The assembly listing for the run.
2 Listings of each input file used.
3 The computer listing created during the test run.
4 Listings of all output files created during the test run.
5 A list of changes that are to be made to the source program prior to the next test.

Because a programmer in a typical data processing department is frequently coding, testing, and debugging several programs at once, this documentation can save you some amount of backtracking and confusion. Also, when you have finished testing and debugging and it is time to do the final documentation of the program, you will find that you already have much of the material required.

DEBUGGING

Debugging is one of the most challenging jobs a programmer has. In a large, complex program, debugging an error can be like solving a mystery. From the outward clues you trace backwards figuring out what happened until you find the culprit: a coding error or input error.

When a program is tested, there are two possible outcomes: (1) it can run to completion (EOJ), or (2) it can be abnormally terminated or cancelled. If the program runs to completion, the programmer discovers errors or bugs by comparing actual output with expected output. If the program is long or the calculations are complex, it may be very difficult to determine why the output is not as expected. In such a case, you may want to add lines of code that print messages as the program executes. These messages can show the contents of selected storage fields or indicate what routine is being executed. In this way they can help you find the bugs. After the errors have been corrected, the extra statements can be removed from the source deck, the program reassembled, and a final test run made.

If a program is abnormally terminated, it is referred to as a *program check*. A program check indicates that the program tried to do something invalid—like trying to execute the add decimal (AP) instruction on EBCDIC data. When a program check occurs, the programmer must find the instruction that led to the program check, find the cause of the error, and correct it.

To illustrate the analysis of a program check, I will use the reorder-listing program. In the source code of this program I have included a minor error that will cause the program to fail. Of the several test cases that I would design to fully test the program, I will use these two basic ones for this illustration:

	Case 1	Case 2
Item Number	00101	00103
Description	GENERATOR	HEATER SOLENOID
On-Hand Qty	00070	00034
On-Order Qty	00050	00000
Reorder Point	00100	00050

The first item does not require reordering so it should not appear on the report; the second item should appear on the report.

Figure 4-3 shows the output of the test run. Because assembly and testing were combined in one job, the assembly listing is followed by the *link-edit map* (part 5) which is followed by a *storage dump* (parts 7 and 8). The storage dump, also known as a *storage printout* or *core dump*, lists the contents of storage at the time of the program check.

The program-check message (part 6) indicates (1) the location of the instruction that was being executed when the program was cancelled and (2) the cause of the cancellation. In this case, the instruction that was being executed is located at hex address 689A and the cause is DATA EXCEPTION.

Some of the most common causes of program checks are listed in figure 4-4; the complete list is given on the IBM "green card" and is explained in the *Principles of Operation* manuals, GA22-6821 (for the System/360) and GA22-7000 (for the System/370). *Data exception* indicates that an operand has invalid data—like EBCDIC data for a packed-decimal instruction. *Operation exception* indicates that the operation code of an instruction is invalid. This could happen if a branch instruction specifies a data name instead of an instruction label. A *decimal-overflow exception* occurs when a decimal arithmetic instruction leads to a result larger than the receiving field can hold. And a *decimal-divide exception* occurs when the quotient of a DP instruction is larger than the receiving field can hold. By far the most common, though, is the data exception.

To find the instruction that caused the program check, the programmer refers to the link-edit map (part 5 of figure 4-3). The link-edit map tells the actual storage positions used for the program. The LOCORE and HICORE columns give the lowest address and the highest address in hex used by the program—in figure 4-3, these are hex 6800 and 6FE9.

In the LABEL column, the names of all *object modules* that make up the complete program (or *phase*) are given. In this case, the phase consists of modules named REORDLST,

```
                                                                                                    PAGE    1

  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                                   FDOS CL3-8 12/13/73

000000                                 1 REORDLST START 0
000000 0530                            2 BEGIN    BALR   3,0                      LOAD BASE REGISTER
000002                                 3          USING  *,3
                                       4          OPEN   CARDIN,PRTOUT
                                       5+* 360N-CL-453 OPEN     CHANGE LEVEL 3-3                            3-3
000002 0700                            6+         CNOP   0,4
000004                                 7+         DC     0F'0'
000004 4110 3646             00648     8+         LA     1,=C'$$BOPEN '
000008 4500 3012             00014     9+IJJ00001 BAL    0,*+4+4*(3-1)
00000C 00000128                       10+         DC     A(CARDIN)
000010 00000160                       11+         DC     A(PRTOUT)
000014 0A02                           12+         SVC    2
                                      13 READCARD GET    CARDIN,CRDWRKA           READ CARD INTO WORK AREA
                                      14+* 360N-CL-453 GET      CHANGE LEVEL 3-0
000016 5810 3656             00658    15+READCARD L      1,=A(CARDIN) GET DTF TABLE ADDRESS
00001A 5800 365A             0065C    16+         L      0,=A(CRDWRKA) GET WORK AREA ADDRESS
00001E 58F1 0010             00010    17+         L      15,16(1) GET LOGIC MODULE ADDRESS
000022 45EF 0008             00008    18+         BAL    14,8(15) BRANCH TO GET ROUTINE
000026 FA20 363A 3678 0063C 0067A     19          AP     COUNT,=P'1'              ADD ONE TO COUNT
00002C F224 362E 3360 00630 00362     20          PACK   WRKAVAIL,CRDONHND
000032 F224 3631 3365 00633 00367     21          PACK   WRKONORD,CRDONORD
000038 FA22 362E 3631 00630 00633     22          AP     WRKAVAIL,WRKONORD        ADD ON HAND AND ON ORDER
00003E F224 3634 335B 00636 0035D     23          PACK   WRKORDPT,CRDORDPT
000044 F922 362E 3634 00630 00636     24          CP     WRKAVAIL,WRKORDPT
00004A 47B0 3014             00016    25          BNL    READCARD
00004E 9240 30DC       000DE          26          MVI    PRTDET,X'40'             MOVE BLANK TO CONTROL BYTE
000052 D283 3518 3517 0051A 00519     27          MVC    PRTDETL+1(132),PRTDETL   BLANK PRINTER WORK AREA
000058 F224 3637 3338 00639 0033A     28          PACK   PACKAREA,CRDITNBR
00005E D205 3518 3621 0051A 00623     29          MVC    PRTITNBR,PATTERN1
000064 DE05 3518 3637 0051A 00639     30          ED     PRTITNBR,PACKAREA        EDIT ITEM NUMBER FIELD
00006A D213 3523 3342 00525 00344     31          MVC    PRTITDES,CRDITDES        MOVE ITEM DESCRIPTION
000070 F224 3637 3356 00639 00358     32          PACK   PACKAREA,CRDPRICE
000076 D206 353B 3627 0053D 00629     33          MVC    PRTPRICE,PATTERN2
00007C DE06 353B 3637 0053D 00639     34          ED     PRTPRICE,PACKAREA        EDIT UNIT PRICE
000082 D205 3546 3621 00548 00623     35          MVC    PRTAVAIL,PATTERN1
000088 DE05 3546 362E 00548 00630     36          ED     PRTAVAIL,WRKAVAIL        EDIT AVAILABLE
00008E D205 3550 3621 00552 00623     37          MVC    PRTORDPT,PATTERN1
000094 DE05 3550 3634 00552 00636     38          ED     PRTORDPT,WRKORDPT        EDIT ORDER POINT
00009A F911 363D 3676 0063F 00678     39          CP     LINECNT,=P'50'           COMPARE LINE COUNT TO 50
0000A0 4740 30DC             000DE    40          BL     PRTDET                   BRANCH ON LOW TO PRTDET
                                      41          PUT    PRTOUT,HDGLINE1          PRINT FIRST HEADING LINE
                                      42+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                        3-5
0000A4 5810 365E             00660    43+         L      1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000A8 5800 3662             00664    44+         L      0,=A(HDGLINE1) GET WORK AREA ADDRESS
0000AC 58F1 0010             00010    45+         L      15,16(1) GET LOGIC MODULE ADDRESS     3-5
0000B0 45EF 000C             0000C    46+         BAL    14,12(15) BRANCH TO PUT ROUTINE        3-5
                                      47          PUT    PRTOUT,HDGLINE2          PRINT SECOND HEADING LINE
                                      48+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                        3-5
0000B4 5810 365E             00660    49+         L      1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000B8 5800 3666             00668    50+         L      0,=A(HDGLINE2) GET WORK AREA ADDRESS
0000BC 58F1 0010             00010    51+         L      15,16(1) GET LOGIC MODULE ADDRESS     3-5
0000C0 45EF 000C             0000C    52+         BAL    14,12(15) BRANCH TO PUT ROUTINE        3-5
                                      53          PUT    PRTOUT,HDGLINE3          PRINT THIRD HEADING LINE
                                      54+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                        3-5
0000C4 5810 365E             00660    55+         L      1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000C8 5800 366A             0066C    56+         L      0,=A(HDGLINE3) GET WORK AREA ADDRESS
0000CC 58F1 0010             00010    57+         L      15,16(1) GET LOGIC MODULE ADDRESS     3-5
0000D0 45EF 000C             0000C    58+         BAL    14,12(15) BRANCH TO PUT ROUTINE        3-5
0000D4 F810 363D 3679 0063F 0067B     59          ZAP    LINECNT,=P'0'            RESET LINE COUNT TO ZERO
0000DA 92F0 3517       00519          60          MVI    PRTDETL,C'0'             MOVE ZERO TO DETAIL CONTROL BYTE
                                      61 PRTDET   PUT    PRTOUT,PRTDETL           PRINT DETAIL LINE
                                      62+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                        3-5
0000DE 5810 365E             00660    63+PRTDET   L      1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000E2 5800 366E             00670    64+         L      0,=A(PRTDETL) GET WORK AREA ADDRESS
0000E6 58F1 0010             00010    65+         L      15,16(1) GET LOGIC MODULE ADDRESS     3-5
0000EA 45EF 000C             0000C    66+         BAL    14,12(15) BRANCH TO PUT ROUTINE        3-5
0000EE FA10 363D 3678 0063F 0067A     67          AP     LINECNT,=P'1'            ADD ONE TO LINE COUNT
0000F4 47F0 3014             00016    68          B      READCARD
0000F8 DE06 359D 363A 0059F 0063C     69 CRDEOF   ED     CNTPATRN,COUNT           EDIT COUNT
                                      70          PUT    PRTOUT,CNTLINE           PRINT COUNT LINE
                                      71+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                        3-5
```

FIGURE 4-3 (Part 1) Test Run Output

```
                                                                                                                PAGE    2

  LOC  OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                                                  FDOS CL3-8 12/13/73

0000FE 5810 365E             00660    72+          L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
000102 5800 3672             00674    73+          L     0,=A(CNTLINE) GET WORK AREA ADDRESS
000106 58F1 0010             00010    74+          L     15,16(1) GET LOGIC MODULE ADDRESS      3-5
00010A 45EF 000C             0000C    75+          BAL   14,12(15) BRANCH TO PUT ROUTINE        3-5
                                      76           CLOSE CARDIN,PRTOUT
                                      77+* 360N-CL-453 CLOSE     CHANGE LEVEL 3-3                                  3-3
00010E 0700                           78+          CNOP  0,4
000110                                79+          DC    0F'0'
000110 4110 364E             00650    80+          LA    1,=C'$$BCLOSE'
000114 4500 311E             00120    81+IJJC0008 BAL    0,*+4+4*(3-1)
000118 00000128                       82+          DC    A(CARDIN)
00011C 00000160                       83+          DC    A(PRTOUT)
000120 0A02                           84+          SVC   2
                                      85           EOJ
                                      86+* 360N-CL-453 EOJ       CHANGE LEVEL 3-0
000122 0A0E                           87+          SVC   14
                                      88 * THE CARD FILE DEFINITION FOLLOWS
                                      89 CARDIN    DTFCD DEVADDR=SYSIPT,IOAREA1=CRDIO1,IOAREA2=CRDIO2,          X
                                                         EOFADDR=CRDEOF,WORKA=YES
                                      90+* 360N-CL-453   DTFCD    CHANGE LEVEL 3-9                                  3-9
000124 00C00000
000128                                91+          DC    0D'0'
000128 000080000000                   92+CARDIN    DC    X'000080000000' RES. COUNT,COM. BYTES,STATUS BTS
00012E 00                             93+          DC    AL1(0) LOGICAL UNIT CLASS
00012F 01                             94+          DC    AL1(1) LOGICAL UNIT
000130 00000148                       95+          DC    A(IJCX0010) CCW ADDRESS
000134 00000000                       96+          DC    4X'00' CCB-ST BYTE,CSW CCW ADDR.
000138 00                             97+          DC    AL1(0) 3-3
000139 000000                         98+          DC    VL3(IJCFZIB0) ADDRESS OF LOGIC MODULE    3-3
00013C 02                             99+          DC    X'02' DTF TYPE (READER)
00013D 05                            100+          DC    AL1(5) SWITCHES
00013E 02                            101+          DC    AL1(2) NORMAL COMM.CODE
00013F 02                            102+          DC    AL1(2) CNTROL COMM.CODE
000140 C00001E0                      103+          DC    A(CRDIO2) ADDR. OF IOAREA2
000144 000000F8                      104+          DC    A(CRDEOF) EOF ADDRESS                      3-8
000148 020C019020000050              105+IJCX0010 CCW   2,CRDIO1,X'20',80
000150 47C0 0000             00000   106+         NOP   0 LOAD USER POINTER REG.
000154 D24F D000 E000 00000  00000   107+         MVC   0(80,13),0(14) MOVE IOAREA TO WORKA
00015A                               108+IJJZ0010 EQU   *
                                     109 * THE PRINTER FILE DEFINITION FOLLOWS
                                     110 PRTOUT   DTFPR DEVADDR=SYSLST,                                          X
                                                        IOAREA1=PRTIO1,                                          X
                                                        IOAREA2=PRTIO2,                                          X
                                                        WORKA=YES,                                               X
                                                        BLKSIZE=133,                                             X
                                                        CTLCHR=ASA
                                     111+* 360 N-CL-453 DTFPR       CHANGE LEVEL 3-8                          3-8
00015A 000C00000000
000160                               112+         DC    0D'0'
000160 000080000000                  113+PRTOUT   DC    X'000080000000' RES. COUNT,COM. BYTES,STATUS BTS
000166 00                            114+         DC    AL1(0) LOGICAL UNIT CLASS
000167 03                            115+         DC    AL1(3) LOGICAL UNIT
000168 00000188                      116+         DC    A(*+32) CCW  ADDR.
00016C 00C00000                      117+         DC    4X'00' CCB-ST BYTE,CSW CCW ADDRESS
000170 00                            118+         DC    AL1(0) 3-3
000171 000000                        119+         DC    VL3(IJDFAZIW) ADDR OF LOGIC MODUL3-8
000174 08                            120+         DC    X'08' DTF TYPE  (PRINTER)
000175 36                            121+         DC    AL1(54) SWITCHES
000176 09                            122+         DC    X'09' NORMAL  COMM. CODE
000177 09                            123+         DC    X'09' CONTROL COMM. CODE
000178 00000231                      124+         DC    A(PRTIO1+1) ADDRESS OF DATA IN IOAREA1
00017C 00000000                      125+         DC    4X'00' BUCKET                          3-5
000180 0700                          126+         NOPR  0 PUT LENGTH IN REG12 (ONLY UNDEF.
000182 4700 0000             00000   127+         NOP   0 LOAD USER POINTER REG
000186 0000                          128+         DC    2X'00' NOT USED                    3-5
000188 090002B620000084              129+         CCW   9,PRTIO2+1,X'20',133-1
000190                               130+IJJZ0011 EQU   *
                                     131 * THE DATA DEFINITIONS FOR THE TWO CARD I/O AREAS FOLLOW
000190                               132 CRDIO1   DS    CL80
0001E0                               133 CRDIO2   DS    CL80
                                     134 * THE DATA DEFINITIONS FOR THE TWO PRINTER AREAS FOLLOW
```

FIGURE 4-3 (Part 2) Test Run Output

```
                                                                                                                    PAGE     3

  LOC   OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                                                    FDOS CL3-8 12/13/73

000230                               135 PRTIO1   DS     CL133
0002B5                               136 PRTIO2   DS     CL133
                                     137 * THE DATA DEFINITIONS FOR THE CARD WORK AREA FOLLOW
00033A                               138 CRDWRKA  DS     0CL80
00033A                               139 CRDITNBR DS     CL5
00033F                               140          DS     CL5
000344                               141 CRDITDES DS     CL20
000358                               142 CRDPRICE DS     CL5
00035D                               143 CRDORDPT DS     CL5
000362                               144 CRDONHND DS     CL5
000367                               145 CRDONORD DS     CL5
00036C                               146          DS     CL30
                                     147 * THE DATA DEFINITIONS FOR THE PRINTER HEADING LINES FOLLOW
00038A                               148 HDGLINE1 DS     0CL133
00038A F1                            149          DC     C'1'
00038B 4040404040404040              150          DC     24C' '
0003A3 D9C5D6D9C4C5D940              151          DC     C'REORDER LISTING'
0003B2 4040404040404040              152          DC     93C' '
00040F                               153 HDGLINE2 DS     0CL133
00040F F0                            154          DC     C'0'
                                     155          DC     C' ITEM              ITEM                  UNIT              X
000410 40C9E3C5D4404040                               REORDER'
00044F 4040404040404040              156          DC     69C' '
000494                               157 HDGLINE3 DS     0CL133
000494 40                            158          DC     C' '
                                     159          DC     C'  NO.          DESCRIPTION           PRICE    AVAILABLEX
000495 4040D5D64B404040                               POINT'
0004D3 4040404040404040              160          DC     70C' '
                                     161 * THE DATA DEFINITIONS FOR THE PRINTER DETAIL LINE FOLLOW
000519                               162 PRTDETL  DS     0CL133
000519                               163 PRTDCTL  DS     CL1
00051A                               164 PRTITNBR DS     CL6
000520 4040404040                    165          DC     5C' '
000525                               166 PRTITDES DS     CL20
000539 40404040                      167          DC     4C' '
00053D                               168 PRTPRICE DS     CL7
000544 40404040                      169          DC     4C' '
000548                               170 PRTAVAIL DS     CL6
00054E 40404040                      171          DC     4C' '
000552                               172 PRTORDPT DS     CL6
000558 4040404040404040              173          DC     70C' '
                                     174 * THE DATA DEFINITIONS FOR THE FINAL TOTAL LINE FOLLOW
00059E                               175 CNTLINE  DS     0CL133
00059E 60                            176          DC     C'-'
00059F 4020206B202020                177 CNTPATRN DC     X'4020206B202020'
0005A6 40C3C1D9C4E240C9              178          DC     C' CARDS IN THE INPUT DECK'
0005BE 4040404040404040              179          DC     101C' '
                                     180 * THE DATA DEFINITIONS THAT FOLLOW DEFINE OTHER WORK AREAS NEEDED
                                     181 * BY THE PROGRAM
000623 402020202020                  182 PATTERN1 DC     X'402020202020'
000629 402020214B2020                183 PATTERN2 DC     X'402020214B2020'
000630                               184 WRKAVAIL DS     PL3
000633                               185 WRKONORD DS     PL3
000636                               186 WRKORDPT DS     PL3
000639                               187 PACKAREA DS     PL3
00063C 00000C                        188 COUNT    DC     PL3'0'
00063F                               189 LINECNT  DS     P'50'
000000                               190          END    BEGIN
000648 5B5BC2D6D7C5D540              191                 =C'$$BOPEN '
000650 5B5BC2C3D3D6E2C5              192                 =C'$$BCLOSE'
000658 00000128                      193                 =A(CARDIN)
00065C 0000033A                      194                 =A(CRDWRKA)
000660 00000160                      195                 =A(PRTOUT)
000664 0000038A                      196                 =A(HDGLINE1)
000668 0000040F                      197                 =A(HDGLINE2)
00066C 00000494                      198                 =A(HDGLINE3)
000670 00000519                      199                 =A(PRTDETL)
000674 0000059E                      200                 =A(CNTLINE)
000678 050C                          201                 =P'50'
00067A 1C                            202                 =P'1'
00067B 0C                            203                 =P'0'
```

FIGURE 4-3 (Part 3) Test Run Output

CROSS-REFERENCE PAGE 1

12/13/73

SYMBOL	LEN	VALUE	DEFN	REFERENCES
BEGIN	00002	000000	00002	0190
CARDIN	00006	000128	00092	0010 0015 0082 0193
CNTLINE	00133	00059E	00175	0073 0200
CNTPATRN	00007	00059F	00177	0069
COUNT	00003	00063C	00188	0019 0069
CRDEOF	00006	0000F8	00069	0104
CRDIO1	00080	000190	00132	0105
CRDIO2	00080	0001E0	00133	0103
CRDITDES	00020	000344	00141	0031
CRDITNBR	00005	00033A	00139	0028
CRDONHND	00005	000362	00144	0020
CRDONORD	00005	000367	00145	0021
CRDORDPT	00005	00035D	00143	0023
CRDPRICE	00005	000358	00142	0032
CRDWRKA	00080	00033A	00138	0016 0194
HDGLINE1	00133	00038A	00148	0044 0196
HDGLINE2	00133	00040F	00153	0050 0197
HDGLINE3	00133	000494	00157	0056 0198
IJCX0010	00008	000148	00105	0095
IJJC0008	00004	000114	00081	
IJJO0001	00004	000008	00009	
IJJZ0010	00001	00015A	00108	
IJJZ0011	00001	000190	00130	
LINECNT	00002	00063F	00189	0039 0059 0067
PACKAREA	00003	000639	00187	0028 0030 0032 0034
PATTERN1	00006	000623	00182	0029 0035 0037
PATTERN2	00007	000629	00183	0033
PRTAVAIL	00006	000548	00170	0035 0036
PRTDCTL	00001	000519	00163	0060
PRTDET	00004	0000DE	00063	0026 0040
PRTDETL	00133	000519	00162	0027 0027 0064 0199
PRTIO1	00133	000230	00135	0124
PRTIO2	00133	0002B5	00136	0129
PRTITDES	00020	000525	00166	0031
PRTITNBR	00006	00051A	00164	0029 0030
PRTORDPT	00006	000552	00172	0037 0038
PRTOUT	00006	000160	00113	0011 0043 0049 0055 0063 0072 0083 0195
PRTPRICE	00007	00053D	00168	0033 0034
READCARD	00004	000016	00015	0025 0068
REORDLST	00001	000000	00001	
WRKAVAIL	00003	000630	00184	0020 0022 0024 0036
WRKONORD	00003	000633	00185	0021 0022
WRKORDPT	00003	000636	00186	0023 0024 0038

NO STATEMENTS FLAGGED IN THIS ASSEMBLY

FIGURE 4-3 (Part 4) Test Run Output

12/13/73	PHASE	XFR-AD	LOCORE	HICORE	DSK-AD	ESD TYPE	LABEL	LOADED	REL-FR
	REORDER	006800	006800	006FF1	5D 11 4	CSECT	REORDLST	006800	006800
						CSECT	IJCFZIB0	006E80	006E80
						CSECT	IJDFAZIW	006F10	006F10

FIGURE 4-3 (Part 5) Test Run Output—Link-Edit Map

```
0S03I PROGRAM CHECK INTERRUPTION - HEX LOCATION 00689A - CONDITION CODE 2 - DATA EXCEPTION
0S00I JOB MCQ       CANCELED
```

FIGURE 4-3 (Part 6) Test Run Output—Program Check Message

```
         MCQ                   12/13/73                                                                   18.50.33              PAGE    1

GR 0-7   00006B3A 00006928 00016FFF 40006802   00016F84 FFFFFF7C 00006800 00003F98
GR 8-F   00005AA6 0A0407F1 00004010 00004010   00004FD0 00005FD0 A0006826 00006E80
FP REG   0C000000 00000000 00000000 00000000   00000000 00000000 00000000 00000000
CCMREG   BG ADDR IS 000328

000000   00000000 00000000 00000000 00000000   00000000 00000328 FF050000 00000000   ................   ................
000020   FF050007 40003B66 FF150007 E00068A0   5B5BC2C5 D6D1F340 FF05000E 80004C28   .... ...........   $$BEOJ3 ......¤.
000040   00003C88 08000000 00003C78 00000000   F4D43600 022B5A46 00040000 0F001C20   ................   4M..............
000060   00040000 00000560 00040000 00001BE0   00000000 00000E5C 00040000 000003FC   .......-........   ........*.......
000080   00000000 00000000 00000000 00000003   00050003 06B006B0 41BB0072 45700144   ................   ................
0000A0   940FBBE1 41A0C2D4 45700E26 18A84190   01844180 BA2C47F0 00D806B0 06B006B0   ......BM........   ........0.Q.....
0000C0   06E006B0 C6B006B0 41BB0017 41BB0050   45700144 41800184 9640A001 9120A00C   ...............&   ......... ......
0000E0   471000E8 9260A001 95E2A002 4780B0F2   95C1A002 4780B0F2 9561A002 4780010C   ...Y.-...S.....2   .A......2./.....
000100   9104A000 4780010C 9203008F 9281A000   4BA00382 487AC280 49A00396 41AAC2C4   ................   ......B.......BD
000120   077894F9 703BD701 70587058 9283A000   9680A001 440006B6 0788947F A0014570   ...9..P.........   ................
000140   B95607F8 42B000E5 58B00280 9180A000   07175890 A0044880 022E5000 902C6000   ...8...V........   ...........&...-.
000160   90586020 90606040 90686060 9070D21B   90100260 D2079008 80009680 A00007F7   ..-..-- ..--..K.   ...-K..........7
000180   4570014C D205B51A B521DC05 B51AC2C4   1BAADD06 B51A000C 43A10007 42A00357   ...¤K.........BD   ................
0001A0   41AAC2C4 D2000541 03579540 05414740   01C89560 05414780 01C29240 054147F0   ..BDK...... ...    .H.-.....B. ...0
0001C0   01C8D200 05410540 45900292 4400A004   5890A004 4220A000 9140A001 47100242   .HK.... .......    ......... ......
0001E0   D2070260 90086800 90586820 90606840   90686860 90704BA0 038241AA C280D201   K..-..........-.   ...-........B.K.
000200   0016A000 98989010 82000260 45900292   4400A004 5890A004 98189030 91020021   ...........-....   ................
000220   47800228 92000542 989F0260 82000038   9284C324 D2070260 B5609890 B5688200   ...........-....   ..C.K..-.-......
000240   02609680 A0004110 003047F0 B8CA9603   00399601 00388200 0038615C 44F00000   .-..........0...   ............/*.0..
000260   FF050007 40003B66 00001000 00002000   00003000 00004000 000038B0 8000178A   .... ...........   ...... .........
000280   00001000 00002000 00003000 00004000   234ED500 05410542 07899068 05285880   ............ .     ..N.............
0002A0   00505980 B51047D0 02B28A80 000847F0   02B68880 00085F80 00541078 5E700544   .&.............0   ................
0002C0   50800544 95400542 4740030E 1B664A70   05385D60 053C4060 05385880 05541867   &.... ... ......   ...-.. -........
0002E0   5E608020 50608020 58800550 18675E60   80205060 80205880 054C5E70 80205070   .-. .&-.....&...-   ..&-......¤....&.
000300   8020D200 05420541 98680528 07F91B66   43600542 88600002 58860548 5E70801C   ..K.........9..    .-...-..........
000320   5070801C 47F00302 F1F261F1 F361F7F3   40004000 00000000 00000000 00000000   &....0..12/13/73    . .............
000340   D4C3D840 40404040 00016FFF 00006FF1   00006FF1 00000010 0001FFFF FA5D7E80   MCQ     .......1   ...1............
000360   28A07EC0 00AB2425 242C24CD 24CE0000   26D826DC 26E04CF1 F2F1F3F7 F3F3F4F7   ................   .Q....¤121373347
000380   000022C4 C0000014 13601482 15201538   15400010 23E80010 5B5BC2D6 00060002   ...D.....-......   . ...Y..$$BO....
0003A0   010015A8 22800000 00000000 03281000   000003B4 00000000 000027F8 00000044   ................   ............8....
0003C0   00C00000 --SAME--                                                             ....
0003E0   00000000 C0000000 00000528 000028A0   00000000 00003C78 00002CD4 9238022F   ................   ............M....
000400   909F0260 98BE0280 41900A60 48A00356   4AA00382 9180A000 47100436 58B0A004   ...-.......-....   ................
000420   9018B030 988E0280 90690528 4590046E   98690528 07F99601 A00098BE 02809069   ................   .....9..........
000440   05284590 046E9869 052895FF A00F0789   5000B584 48E0022E D207B560 E00058E0   ................   &.......K..-....
000460   028C94FD B561D21B B5680260 07F9D200   05410542 58800050 5980B510 47D00488   ...../K....-.9K.   .......&........
000480   8A800008 47F0048C 88800008 5F800054   10785E70 05449180 A0004780 04AC95FF   .....0..........   ................
0004A0   A00F4780 04ACD200 05400542 91300542   477004FA 91400542 07195080 05441B66   ......K.. ......   ..... ....&.....
0004C0   4A70053A 5D60C53C 4060053A 58800554   18675E60 80245060 80245880 05501867   .....-.. -......   ...-..&-.....&..
0004E0   5E608024 50608024 5880054C 5E708024   50708024 92400542 07F99528 022F0789   .-..&-.....¤....   &.... ...9......
000500   50800544 1B664360 05428860 00025886   05485E70 801C5070 801C9520 022F0789   &......-...-....   ......&.........
000520   92400542 C7F90000 0000000E 90000184   000022D4 00000A60 00000000 00000001   . ...9..........   ...M...-........
000540   10101010 FEC9755E 00002A08 000028A0   00002918 00002990 00000000 000000C8   .....I..........   ................H
000560   909F0260 9220022F 4590040C 4190020C   95000023 4780069E 95250023 47B000BC   ...-............   ................
000580   48600022 1A654870 02904866 700007F6   181F1B66 4121000F 4570B23E 4860B490   .-.............6   .............-..
0005A0   1B224320 10074130 002E1B23 474005BA   4130002E 1B2347B0 05BC1A23 422007A9   ........... ..     ................
0005C0   43201007 47F00718 472000C8 423007A9   48200356 4322C283 1A23950B 100747F0   .....0.....H....   ......B.........0
0005E0   07144720 00C64400 07449680 1002960C   100407F9 18584143 00024354 00004145   .....F..........   ...9............
000600   50001A44 4A40B488 95FF4000 477005FA   42840000 07F94400 0FE095C0 C31C0749   &.... .... .....   .....9......C...
000620   95FF3002 C7894570 0CBC9180 30064780   BA269104 30064710 BA264860 30009F00   ................   ...........-....
000640   60004720 065A4400 07449E00 60004780   B9764710 0E6847F0 06701B77 43703000   -...........-...   ........0.......
000660   4377C31D 4937C334 47800646 47F0B976   91060045 4770B1E2 95D03004 47800688   ..C...C.......0..   .......S........
000680   95700044 4780B976 95190023 077947F0   BA2695FF 07810789 18005000 B58495FF   ...............0   ...........&.....
0006A0   07814780 B9769240 05424860 03569560   03574780 0590D502 A00503A1 477006CE   ...... ...-...-    ......N.........
0006C0   41110000 4910B2B4 47B006CE 1B664121   000F4570 B23ED502 10090359 47B000C4   ................   ......N........D
0006E0   440007C8 587003F4 D5027001 035947B0   00C41B33 43301007 95011006 477005C8   ...H...4N.......   .D..............H
000700   92FF07A9 D200070D A00D4123 000BD500   1007A00E 47B000C8 41822000 48700374   ....K.........N.   ........H.......
000720   43387000 4930B2B2 47B005E2 89300003   4A300368 95190023 47800616 951B0023   ............S....   ................
000740   47800616 D4031002 C33091F0 30044780   BD74D501 0022C27C 4780C032 D501B480   ....M...C..0....   ..N...B@....N...
000760   B48E4720 077E4930 B4A84770 077E950F   00234780 077E9110 30064780 B9764180   ................   ................
000780   00014148 80001A44 4A40B488 18584A50   0394D200 07814000 50104000 92FF4000   ......... .....&   ..K... .&. ... .
0007A0   42205000 42605014 92035028 91F03004   4780BD94 47F007BC 5880B59C 44000FE0   ..&..-&...&..0..   .....0..........
0007C0   9560C31C 47700836 D20203F5 10095860   03F49507 60004770 0836D202 03F56001   .-C......K..5...-   .4..-......K..5-.
0007E0   587003F4 1B444340 30054C40 B47E5A40   03F84144 0000D503 70014000 47800836   ...4.... ..¤ ...   .8....N... .....
000800   9120100C 47100830 91051002 47700830   91406004 47800830 94BF6004 91101002   ................   . -.......-.....
000820   47800828 96401002 96141002 9601100C   D2034000 700195FF 30024770 05F44280   ..... ..........   K. ...........4..
000840   30029198 30060779 43203000 4322C31D   4860300C 95003000 47800874 95C03004   ..............C.   .-...............
000860   4770086E 9F006000 072947F0 08749F00   60000769 91003006 07194060 08B49550   ......-....0....   -.......... -...&
```

FIGURE 4-3 (Part 7) Test Run Output—Storage Dump

```
        MCG                 12/13/73                                                                    PAGE   6

004000  D9C5D6D9 C4C5D940 D2831014 10830A16   180607F7 00000000 00000000 00000000   REORDER K.......  ...7............
004020  00000000 --SAME--                                                           ....
006800  05300700 41103646 45003012 00006928   00006960 0A025810 36565800 365A58F1   ................  ...-...........1
006820  001045EF 0008FA20 363A3678 F224362E   3360F224 36313365 FA22362E 3631F224   ............2...  .-2...........2.
006840  3634335B F922362E 363447B0 30149240   30DCD283 35183517 F2243637 3338D205   ...$9.........   ..K.....2.....K.
006860  35183621 DE053518 3637D213 35233342   F2243637 3356D206 353B3627 DE06353B   .........K......  2.....K.........
006880  3637D205 35463621 DE053546 362ED205   35503621 DE053550 3634F911 363D3676   ..K...........K.  .&.....&..9.....
0068A0  474030DC 5810365E 58003662 58F10010   45EF000C 5810365E 58003666 58F10010   . ...........1..  ..............1..
0068C0  45EF000C 5810365E 5800366A 58F10010   45EF000C F810363D 367992F0 35174010   ............1...  ....8.......0.. .
0068E0  365E5800 366E58F1 001045EF 000CFA10   363D3678 47F03014 DE06359D 363A5810   ........1.......  .....0..........
006900  365E5800 367258F1 001045EF 000C0700   4110364E 4500311E 00006928 00006960   ........1.......  ..............-
006920  0AC20A0E CCCCCCC0 0000C000 09C00001   00006948 00006950 00006E80 02C50202   ................  ........&.....E..
006940  020069E0 C00068F8 02006990 20000050   47000000 D24FD000 E0000000 00000000   .......8........&  ....K...........
006960  00008000 CCC00003 00006988 00003AD0   00006F10 08B60909 00006A31 00000000   ................  ...............
006980  07004700 C0000000 09006AB6 20000084   615C4040 40404040 40404040 40404040   ................  /*
0069A0  40404040 --SAME--
0069E0  F0F0F1F0 F3C8C5C1 E3C5D940 E2D6D3C5   D5D6C9C4 40404040 40F0F0F3 F3F0F0F0   00103HEATER SOLE  NOID      0033000
0069A0  F4F4F0F0 F0F0F5F0 F0F0F0F3 F4F0F0F0   F0F04040 40404040 40404040 40404040   4400005000034000  00
006A20  40404040 40404040 40404040 40404040   00000000 00000000 00000000 00000000                     ................
006A40  00000000 --SAME--                                                           ....
006B20  00000000 00000000 00000000 00000000   00000000 00000000 0000F0F0 F1F0F3C8   ................  ..........00103H
006B40  C5C1E3C5 D940E2D6 D3C5D5D6 C9C44040   404040F0 F0F3F3F0 F0F0F4F4 F0F0F0F0   EATER SOLENOID       0033000440000
006B60  F5F0F0F0 F0F3F4F0 F0F0F0F0 40404040   40404040 40404040 40404040 40404040   500003400000
006B80  40404040 40404040 4040F140 40404040   40404040 40404040 40404040 40404040                1
006BA0  404040D9 C5D6D9C4 C5D940D3 C9E2E3C9   D5C74040 40404040 40404040 40404040      REORDER LISTI  NG
006BC0  40404040 --SAME--
006C00  40404040 40404040 40404040 404040F0   40C9E3C5 D4404040 40404040 40404040                  0    ITEM
006C20  40C9E3C5 D4404040 40404040 40404040   40404040 40E4D5C9 E3404040 40404040   ITEM                    UNIT
006C40  40404040 40404040 D9C5D6D9 C4C5D940   40404040 40404040 40404040 40404040         REORDER
006C60  40404040 --SAME--
006C80  40404040 40404040 40404040 40404040   40404040 404040D5 D64B4040 40404040                        NO.
006CA0  404040C4 C5E2C3D9 C9D7E3C9 D6D54040   40404040 40404040 40D7D9C9 C3C54040     DESCRIPTION                PRICE
006CC0  4040C1E5 C1C9D3C1 C2D3C540 4040D7D6   C9D5E340 40404040 40404040 40404040   AVAILABLE   PO  INT
006CE0  40404040 --SAME--
006D00  40404040 40404040 40404040 40404040   40404040 40404040 40404040 40F1F0F3                        103
006D20  40404040 40D940E2 D6D3C5D5 D6C9C440   40404040 F0F0F3F3 F0404040 40404040        R SOLENOID         00330
006D40  F44BF4F0 40404040 40404040 F3F44040   40404040 4040F5F0 40404040 40404040   4.40          34           50
006D60  40404040 --SAME--
006D80  40404040 40404040 40404040 40404040   40404040 40404040 40404040 40406040                          -
006DA0  20206B20 202040C3 C1D9C4E2 40C9D540   E3C8C540 C9D5D7E4 E340C4C5 C3D24040   ..... CARDS IN    THE INPUT DECK
006DC0  40404040 --SAME--
006E20  40404040 20202020 20402020 214B2020   00034000 000F0005 0F00440F 0000DC20        .....  .....   ..¤.............
006E40  20204020 20214B20 5B5BC2D6 D7C5D540   5B5BC2C3 D3D6E2C5 00006928 00006B3A   ..  .....$$BOPEN   $$BCLOSE........,:
006E60  00006960 C0C06B8A 00006C0F 00006C94   00006D19 00006D9E 050C1C0C 00004358   ...-.,.,.........  ...............
006E80  0A320000 0A320000 47F0F01A 0A320000   C9D1C3C6 E9C9C2F0 F3F99140 10154710   .........00.....  IJCFZIB039. ....
006EA0  F0289640 10150A00 91801002 4710F032   0A0750E0 F08C58E0 1020D202 10211019   0..  .........0..  ..&.0.....K.....
006EC0  50E01018 91011004 4780F066 91401002   4710F05C 58E0F08C 47F0F026 94BF1015   &.........0..  ..  ..0*..0..00.....
006EE0  58E0101C 07FED501 F086E000 4770F074   47F0F05C 50D0F088 18D04400 102C98DE   .....N.0.....0.  .00*&.0.........
006F00  F0880A00 07FE615C 00005FD0 A0006826   0A320000 0A320000 0A320000 47F0F01A   0...../*.........  ..............00.
006F20  C9D1C4C6 C1E9C9E6 F3F99180 10024710   F0240A07 90CEF0B0 58E01018 06E018D0   IJDFAZIW39......  0.....0.........
006F40  D2001017 D00048C0 102E44C0 F0BC1BCC   18DC43C0 101741E0 001043DE F0C119DC   K.........0...  ..............0A..
006F60  4780F05A 46E0F04A 0A3243DE F0D143C0   10161 9CD 4770F080 91801016 4780F080   ..0....0.....0J..  ......0.........0.
006F80  920B1028 CA009180 10024710 F0800A07   42D01028 42D01016 0A009180 10024710   .........0....  ...............
006FA0  F0940A07 92011028 58E01028 D2021029   101950E0 101898CE F0B00A00 07FE0000   0...........K...  ..&.....0.......
006FC0  00000000 00000000 00000000 D200E000   D000F2C2 C1F9F8F7 F6F5F4F3 C3F14E60   ..........K....  ..2BA9876543C1.-
006FE0  F04093DB D3CBC3BB B3ABA39B E38B031B   130B0000 00000000 00000000 00000000   0 ..L.C.....T...  ...............
007000  00000000 --SAME--                                                           ....
016FE0  00000000 00000000 00000000 00000000   00000000 00000000 00000000 00000000   ................  ...............
```

FIGURE 4-3 (Part 8) Test Run Output—Storage Dump

Code	Exception Type	Explanation
01	Operation	An invalid operation code has been encountered. Often caused by a branch to a data name.
05	Addressing	An instruction has attempted to use an address that is beyond the highest valid storage address. Usually results from an invalid address in a register used in an explicit operand.
06	Specification	An instruction that requires boundary alignment of the data has tried to refer to a storage address that is not on the required boundary.
07	Data	Data that is invalid for the instruction being executed has been found. Most often, invalid data for a packed-decimal instruction is the cause.
0A	Decimal Overflow	A packed-decimal add, subtract, or multiply has resulted in a value that is too large for the receiving field. Often caused by bad input data.
0B	Decimal Divide	The quotient resulting from a divide-decimal instruction is too large for the quotient field. Usually caused by dividing by zero.

FIGURE 4-4 Common Program Checks

the name used in the START statement of the reorder-listing program, and IJCFZIBO and IJDFAZIW, the names of two I/O modules supplied with DOS. These I/O modules assist the REORDLST module in controlling the card-reading and line-printing operations.

The LOADED column gives the starting address of each module and the REL-FR gives the *relocation factor* for each module. The relocation factor is most important for debugging because it relates the actual storage locations of your program to the location counter values given in the assembly listing. The actual starting location of any instruction or data field is the sum of the relocation factor and the location counter value.

Now, to get back to the question: Which instruction caused the program check? To find out, subtract the relocation factor of REORDLST (6800) from the instruction location given in the program-check message (689A). The result is the location counter value of the offending instruction.

There are a couple of ways of subtracting one hex number from another, but the most efficient way is to do it in the same way you would do decimal subtraction:

```
 689A
-6800
 ----
   9A
```

Start with the right column of digits and move left. Hex 0 from hex A is hex A; hex 0 from hex 9 is hex 9; hex 68 from hex 68 is 0. So the location counter value is hex 9A.

Since most relocation factors end in zeros, the subtraction process is usually simple. But what if the relocation factor is 4D88 and the program check address 5C3C? Then, the problem is:

```
 5C3C
-4D88
 ----
```

Because this problem involves "borrowing" from the next column, I'll show the subtraction process one column at a time.

The first column is hex C minus hex 8. With practice, you will eventually learn the hex subtraction tables and know right off that the answer is hex 4. At the start, however, you will probably find it easiest to convert both values to decimal, subtract, and convert the result back to hex. Thus, hex C is decimal 12, hex 8 is decimal 8, 12 minus 8 is decimal 4, and decimal 4 is hex 4.

For the second column of digits, I must "borrow" since 8 can't be subtracted from 3:

```
 5C3C
-4D88
 ----
    4
```

To borrow, you reduce the digit to the left by one and increase the number being subtracted from by hex 10. Thus, the second column becomes hex 13 minus hex 8. Since hex 13 is the same as hex 10 (decimal 16) plus 3, the decimal

value of hex 13 is 19. Then, 19 minus 8 is 11. Converting back to hex, decimal 11 equals hex B.

The third column again involves borrowing:

```
  B
 5C3C
-4D88
  B4
```

Since 1 was borrowed from C, the problem is hex B minus hex D, which can't be done. After borrowing, it becomes hex 1B minus hex D. Converting to decimal, it is 27 (16 + 11) minus 13 equals 14. Converting back to hex, the result is E.

Since the fourth column has been borrowed from, it now has a value of 4.

```
  4B
 5C3C
-4D88
 E B4
```

Hex 4 minus hex 4 is 0, so the final result is hex EB4.

In any event, you must be able to subtract the relocation factor from the program-check value regardless of the digits involved. If you do it in a way comparable to decimal subtraction, you will soon gain considerable proficiency at this operation. Although hex subtraction may seem like a lengthy process the first few times, you will soon be able to derive the location counter value in a matter of seconds.

Once you have the location counter value—like 9A in this debugging example—find the instruction in the assembly listing. In this case, it is statement 39, the CP instruction:

```
CP     LINECNT,=P'50'
```

Since the cause of the program check is data exception, you can now examine the data fields involved by looking them up in the storage dump.

To find the data fields, you must first calculate their starting addresses. You do this by adding their location counter value to the relocation factor given in the link-edit map. Since the ADDR1 column in statement 39 gives 63D as the location counter value for LINECNT, the first storage byte of the field is hex 63D plus hex 6800. You can also find the location counter value for a field by looking it up in the cross-reference listing.

To add two hex numbers, use a method comparable to decimal addition. Thus,

```
 6800
+ 63D
 6E3D
```

In this case, the addition was simple, you had only to remember that hex 8 plus hex 6 equals hex E with no carry. If carrying is involved, it is done as in this column-by-column example:

```
 6E8 8
+ 1D3
     B
```

For the first column, hex 8 is added to hex 3 giving hex B. If necessary, you can convert both numbers to decimal, add them, and convert back to hex. Thus, hex 8 plus hex 3 is decimal 8 plus 3, which is equal to 11, which is hex B.

```
 6E8 8
+ 1D3
    B
```

For the second column, a carry is involved:

```
   1
 6E8 8
+ 1D3
  05 B
```

Converting to decimal, it is 8 plus 13 equals 21. Converting back to hex, the result is 15 (hex 10 has a value of 16, hex 5 has a value of 5: 16 + 5 = 21). So 5 is written below the second column and 1 is carried to the third column.

For the third column, it is 1 plus E plus 1, resulting in another carry:

```
  11
 6E8 8
+ 1D3
  05 B
```

Hex E is decimal 14 plus 1 plus 1 is 16. Since decimal 16 is hex 10, the 1 is carried.

The fourth column is simply 6 plus 1 so the problem is complete as follows:

```
  6E8 8
+ 1D3
  705 B
```

This brings us back to finding LINECNT in the storage dump starting at byte 6E3D. At the start of the storage dump, the contents of the 16 registers are shown. This is followed by a listing of the contents of storage. Each line of the dump displays 32 bytes of storage in groups of four, with the address of the first byte of the line in the left margin. The line next to address 6800, for example, displays the 16 bytes from 6800-680F counting in hex and the 16 bytes from 6810-681F as follows:

```
ADDRESS     STORAGE CONTENTS
6800-6803       05300700
6804-6807       41100363E
6808-680B       45003012
680C-680F       00006928

6810-6813       00006960
6814-6817       0A025810
6818-681B       364E5800
681C-681F       365258F1
```

If you check the object code for the first several instructions in the assembly listing against the object code starting at address 6800 in the storage dump, you will find it identical.

To the right of the eight columns representing the contents of the 32 bytes of storage in hex notation are two columns representing the same 32 bytes in character notation. If a period is given for a storage position, it means the byte does not contain an EBCDIC code that has a character representation. Otherwise, the character representing the code is printed. If you look to the dump line starting at address 69E0, for example, you will see the characters for columns 1-32 of the input card that was being processed when the program check occurred:

```
00103HEATER SOLE  NOID      0033000
```

These bytes are the first part of the card I/O area.

To find byte 6E3D in the storage dump, scan down the column of addresses on the left side of the storage dump until you come to the nearest address below 6E3D—namely, address 6E20. Then, count over to the hex digits representing address 6E3D. Since the LINECNT field is two bytes long, you will want to examine bytes 6E3D and 6E3E, the shaded bytes shown here:

```
006E20  40402020 20202040 2020214B 20200003
        4C000000F 00050F00 440F0000 20000000
```

As you can see, this field does not contain valid packed-decimal data and that was the cause of the data exception. Now, why does the field LINECNT contain invalid data?

Since LINECNT was defined as a constant, I must assume that the error is not in any input data, but in my source code:

```
LINECNT  DS    P'50'
```

And now I see the error. Because I coded DS instead of DC, the program assembled correctly but no starting value was given to the field. By changing DS to DC and reassembling, this aspect of the program will be debugged.

You might have noticed that the storage printout given in figure 4-3 is only the first and last page of the actual six-page printout. Pages 2 through 5 of the dump are omitted since they represent the contents of the supervisor area and some unused storage positions. Generally, the contents of the supervisor area don't apply to a debugging problem, but in some cases you may want to know the contents of one or more of the general registers. These contents are shown in the lines headed GR 0-7 and GR 8-F on the first page of the

```
12/13/73    PHASE   XFR-AD  LOCORE  HICORE  DSK-AD    ESD TYPE  LABEL     LOADED  REL-FR
          PHASE***  004000  004000  00471F  5D 11 4   CSECT     REORDLST  004000  004000
                                                       CSECT     IJCFZIB0  004690  004690
* UNREFERENCED SYMBOLS                                EXTRN     IJDFZPIW
001 UNRESOLVED ADDRESS CONSTANTS
```

FIGURE 4-5 (Part 1) Test Run Output

```
0S03I PROGRAM CHECK INTERRUPTION - HEX LOCATION 000000 - CONDITION CODE 0 - OPERATION EXCEPTION
0S00I JOB MCQ       CANCELED
```

FIGURE 4-5 (Part 2) Test Run Output

storage printout. For example, the content of register 3, the base register for this program, is 40006802. Since only the rightmost three bytes are involved in base addressing, the base address is hex 006802, which is the address of the first instruction following the BALR.

Sometimes the cause of a program check is an absent or incorrect I/O module. Figure 4-5 shows the test run output of the reorder listing program in the PRTOV macro form. Notice that the address in the program-check message is 000000. A very low storage address like this—000008, 00000C, and 000010 are also common—indicates that one of the I/O modules required by the program could not be found. The link-edit map will show that at least one I/O module, IJDFZPIW in figure 4-5, could not be found and resulted in an *unresolved address constant* in the program.

This problem is corrected by assembling an I/O module with the proper name. The *DOS Supervisor and I/O Macros* manual, number GC24-5037, illustrates the special macros used to generate these I/O modules. In this case, you would code a PRMOD macro:

```
PRMOD PRTOV=YES,IOAREA2=YES,WORKA=YES,SEPASMB=YES
```

The object module that is punched as the output of the assembly of this macro must then be *catalogued* in the *relocatable library* to make it available for link-editing. The relocatable library and the catalog procedure are described in chapter 16. Although you can't be expected to correct a problem like this at this stage of your development, you should be able to determine the cause of the program check.

You may also encounter a program check in which the instruction address given in the message is within an I/O module. For example, suppose the printer module for the test of figure 4-5 is assembled and catalogued, and the test is run again. The link-edit map will now have a LOADED address and relocation factor for IJDFZPIW: say 4720. This time, another program check occurs and the address of the offending instruction is 0046B4. You can see from the link-edit map that this instruction is not in the reorder-listing program, but is within the card I/O module, IJCFZIBO. Since you don't have a listing of these instructions and they would be difficult to debug if you did, this could stop you. Fortunately, the cause of a program check in an I/O module is almost always the result of an improperly coded DTF. By analyzing the DTF operands, you should be able to find an inconsistency that has led to the error.

SUMMARY

Because many different procedures were described when going through these debugging examples, let me recap the general sequence in debugging program checks.

1 Find the program-check message and record the instruction location and the cause. For example: 689A, data exception.
2 Check the link-edit map to see which module the offending instruction is in. If it is in an I/O module, look for some inconsistency in your DTFs. If it is in your program, continue.
3 Derive the location counter value of the instruction by subtracting the relocation factor of your program from the offending instruction address.
4 Look up the instruction in the assembly listing of your program by using the location counter value.
5 At this point, you may want to locate instructions or fields in the storage dump depending on the type of exception you're dealing with. To find the starting byte of any field or instruction, add its location counter value to the relocation factor.
6 Examine areas of the storage dump and analyze the assembly listing to determine the specific cause of the error. This can be simple, as in the data exception example, or it can be very complicated.

If this seems like a lengthy process to go through to find one trivial error, take heart. Once you have found the cause of a program check a few times, you will be able to locate instructions and fields in the storage dump with considerable speed.

Terminology

system test
utility program
program check
link-edit map
storage dump
storage printout
core dump
data exception
operation exception
decimal-overflow exception
decimal-divide exception
object module
phase
relocation factor
unresolved address constant
catalogued
relocatable library

Objectives

1 Given a programming problem, design adequate test data for it.
2 Given a programming problem and program, debug the program. In particular, be able to determine the cause of of a program check.

Problems

1 (Objective 1) A set of test data for the reorder-listing program follows. What aspects of the program will not be tested by this data?

```
00101GENERATOR          040000490000100000700005O
00103HEATER SOLENOID    003300044000050000340000O
03244GEAR HOUSING       065000790000010000120000O
03981PLUMB LINE         002100024000015000350000O
04638START SWITCH       009000098000030000160000O
```

```
                    REORDER LISTING

ITEM          ITEM                    UNIT                    REORDER
 NO.      DESCRIPTION                PRICE     AVAILABLE      POINT
```

FIGURE 4-6 (Part 1) Test Run Output—Problem 2

```
0S03I PROGRAM CHECK INTERRUPTION - HEX LOCATION 0068DE - CONDITION CODE 0 - OPERATION EXCEPTION
0S00I JOB MCQ       CANCELED
```

FIGURE 4-6 (Part 2) Test Run Output—Problem 2

2 (Objective 2) Assume that the bug illustrated in figure 4-3 is corrected and the program is tested again on the test data given in problem 1. This time the heading lines print, and another program check occurs. The output for this test run is given in figure 4-6. Assuming the assembly listing and link-edit map are the same as those in figure 4-3 with the one previous error corrected, what is the bug this time?

3 (Objective 2) Figure 4-7 shows a test run of the reorder-listing program in the CNTRL and PRTOV macro form. Explain the cause of the program check and how to correct it.

```
        MCG                     12/13/73                                                                     18.32.17                 PAGE   1

GR 0-7   00006C94 00006960 00016FFF 40006802    00016F84 FFFFFF7C 00006800 00003F98
GR 8-F   00005AA6 CA0407F1 00004010 00004010    00004FD0 00005FD0 800068D4 00006F10
FP REG   00000000 C0000000 00000000 00C00000    00000000 00000000 00000000 00000000
CCMREG   BG ADDR IS 000328

000000   00000000 00000000 00000000 00000000    00000000 00000328 FF050000 00000000    ................   ................
000020   FF050007 40003B66 FF150001 C00068E4    5B5BC2C5 D6D1F340 FF05000E 80001226    .... ..........U   $$BEOJ3 ........
000040   00003C88 08000000 00003C78 00000000    F9D8F000 022B5A46 00040000 0F001C20    ................   9Q0.............
000060   00040000 00000560 00040000 00001BE0    00000000 00000E5C 00040000 000003FC    .......-........   .......*........
000080   00000000 00000000 00000000 00000003    00050003 06B00680 41BB0072 45700144    ................   ................
0000A0   940FBBE1 41A0C2D4 45700E26 18A84190    01844180 BA2C47F0 00D806B0 06B006B0    ......BM........   .......0.Q......
0000C0   06B006B0 06B006B0 41BB0017 41BB0050    45700144 41800184 9640A001 9120A00C    ...............&   ......... ......
0000E0   471000E8 9260A001 95E2A002 4780B0F2    95C1A002 4780B0F2 9561A002 4780010C    ...Y.-...S.....2   .A.....2./......
000100   9104A000 4780010C 9203008F 9281A000    48A00382 487AC280 49A00396 41AAC2C4    ................   ......B.......BD
000120   077894F9 703BD701 70587058 9283A000    9680A001 44000686 0788947F A0014570    ...9..P.........   ................
000140   B95607F8 42B000E5 58B00280 9180A000    07175890 A0044880 022E5000 902C6000    ...8...V........   ..........&...-.
000160   90586020 90606040 90686060 9070D21B    90100260 D2079008 80009680 A00007F7    ..-..-- ..--..K.   ...-K..........7
000180   4570014C D205B51A B521DC05 B51AC2C4    1BAADD06 B51A000C 43A10007 42A00357    ...¤K.........BD   ................
0001A0   41AAC2C4 D20C0541 03579540 05414740    01C89560 05414780 01C29240 054147F0    ..BDK...... ...    .H.-.....B. ...0
0001C0   01C8D200 05410540 45900292 4400A004    5890A004 4220A000 9140A001 47100242    .HK.... ........   ......... ......
0001E0   D2070260 90086800 90586820 90606840    90686860 90704BA0 038241AA C280D201    K..-..........-    ...-........B.K.
000200   0016A000 98989010 82000260 45900292    4400A004 5890A004 98189030 91020021    ...........-....   ................
000220   47800228 92000542 989F0260 82000038    9284C324 D2070260 B5609890 B5688200    ...........-....   ..C.K..-.-......
000240   02609680 A0004110 003047F0 B8CA9603    00399601 00388200 0038615C 44F00000    .-.........0....   ........../*.0..
000260   FF050007 40003B66 00001000 00002000    00003000 00004000 000038B0 8000178A    .... ...........   ...... .........
000280   00001000 00002000 00003000 00004000    234ED500 05410542 07899068 05285880    .............. .   ..N.............
0002A0   00505980 B51047D0 02B28A80 000847F0    02B68880 00085F80 00541078 5E700544    .&.............0   ................
0002C0   50800544 95400542 4740030E 1B664A70    05385D60 053C4060 05385880 05541867    &.... ... ......   ...-.. -........
0002E0   5E608020 50608020 58800550 18675E60    80205060 80205880 054C5E70 80205070    .-..&-.....&...-   ..&-......¤...&.
000300   8020D200 C5420541 98680528 07F91B66    43600542 88600002 58860548 5E70801C    ..K..........9..   .-...-..........
000320   5070801C 47F00302 F1F261F1 F361F7F3    40004000 00000000 00000000 00000000    &....0..12/13/73    . .............
000340   D4C3D840 40404040 00016FFF 00006FF1    00006FF1 00000010 0001FFFF FA5D7E80    MCQ     .......1   ...1............
000360   28A07EC0 00AB2425 242C24CD 24CE0000    26D826DC 26E04CF1 F2F1F3F7 F3F3F4F7    ................   .Q....¤121373347
000380   000022C4 C0000014 13601482 15201538    15400010 23E80010 5B5BC2D6 00060002    ...D......-.....   . ...Y..$$BO....
0003A0   010015A8 22800000 00000000 03281000    000003B4 00000000 000027F8 00000044    ................   ...........8....
0003C0   00C00000 --SAME--                                                            ....
0003E0   00C00000 00000000 00000528 000028A0    00000000 00003C78 00002CD4 9238022F    ................   ...........M....
000400   909F0260 98BE0280 41900A60 48A00356    4AA00382 9180A000 47100436 58B0A004    ...-.......-....   ................
000420   9018B030 98BE0280 90690528 4590046E    98690528 07F99601 A00098BE 02809069    ................   .....9..........
000440   05284590 046E9869 052895FF A00F0789    5000B584 48E0022E D207B560 E00058E0    ................   &.......K..-....
000460   028C94FD B561D21B B5680250 07F9D200    05410542 58800050 5980B510 47D00488    ...../K....-.9K.   .......&........
000480   8A800008 47F0C49C 88800008 5F800054    10785E70 05449180 A0004780 04AC95FF    .....0..........   ................
0004A0   A00F4780 04ACD200 05400542 91300542    477004FA 91400542 07195080 05441B66    ......K.. ......   ..... ....&.....
0004C0   4A70053A 5D60C53C 4060053A 58800554    18675E60 80245060 80245880 05501867    .....-.. -......   ...-..&-.....&..
0004E0   5E608024 50608024 5880054C 5E708024    50708024 92400542 07F99528 022F0789    .-..&-.....¤....   &.... ...9......
000500   50800544 1B664360 05428860 00025886    0585E070 801C5070 801C9520 022F0789    &......-...-....   ......&.........
000520   92400542 C7F90000 0000000E 90000184    000022D4 00000A60 00000000 00000001    . ...9..........   ...M...-........
000540   10101010 FECE7A31 00002A08 000028A0    00002918 00002990 00000000 000000C8    ................   ...............H
000560   909F0260 9220022F 4590040C 4190020C    95000023 4780069E 95250023 47B000BC    ...-............   ................
000580   48600022 1A664870 02904866 700007F6    181F1B66 4121000F 4570B23E 4860B490    .-.............6   .............-..
0005A0   1B224320 10074130 002E1B23 474005BA    4130002E 1B2347B0 058C1A23 422007A9    ............. ..   ................
0005C0   43201007 47F00718 472000C8 423007A9    48200356 4322C283 1A23950B 100747F0    .....0.....H....   ......B........0
0005E0   07144720 00C64400 07449680 1002960C    100407F9 18584143 00024354 00004145    .....F..........   ...9............
000600   50001A44 4A40B488 95FF4000 477005FA    42840000 07F94400 0FE095C0 C31C0749    &.... .... .....   .....9......C...
000620   95FF3002 07894570 0CBC9180 30064780    BA269104 30064710 BA264860 30009F00    ................   ...........-....
000640   60004720 065A4400 07449E00 60004780    B9764710 0E6847F0 06701B77 43703000    -...........-...   .......0........
000660   4377C31D 4937C334 47800646 47F0B976    91060045 4770B1E2 95D03004 47800688    ..C...C......0..   .......S........
000680   95700044 4780B976 95190023 077947F0    BA2695FF 07810789 1B005000 B58495FF    ...............0   ..........&.....
0006A0   07814780 B9769240 05424860 03569560    03574780 0590D502 A00503A1 477006CE    ....... ...-...-   ......N.........
0006C0   41110000 4910B2B4 47B006CE 1B664121    000F4570 B23ED502 10090359 47B000C4    ................   ......N........D
0006E0   440007C8 587003F4 D5027001 035947B0    00C41B33 43301007 95011006 477005C8    ...H...4N.......   .D.............H
000700   92FF07A9 D200700D A00D4123 000BD500    1007A00E 47B000C8 41822000 48700374    ....K.........N.   .......H........
000720   43387000 4930B2B2 47B005E2 89300003    4A300368 95190023 47800616 951B0023    ...........S....   ................
000740   47800616 D4031002 C33091F0 30044780    BD74D501 0022C27C 4780C032 D501B480    ....M...C..0....   ..N...B@....N...
000760   848E4720 077E4930 B4A84770 077E950F    00234780 077E9110 30064780 B9764180    ................   ................
000780   00004148 E0001A44 4A40B488 18584A50    0394D200 07814000 50104000 92FF4000    ......... .....&   ..K... .&. ... .
0007A0   42205000 42605014 92035028 91F03004    4780BD94 47F007BC 5880B59C 44000FE0    ..&..-&...&..0..   .....0..........
0007C0   9560C31C 47700836 D20203F5 10095860    03F49507 60004770 0836D202 03F56001    .-C.....K..5...-   .4..-.....K..5-.
0007E0   587003F4 18444340 30054C40 847E5A40    03F84144 0000D503 70014000 47800836    ...4... ..¤ ...    .8....N... .....
000800   9120100C 4710C830 91051002 47700830    91406004 47800830 94BF6004 91101002    ................   . -.......-.....
000820   47800828 96401002 96141002 9601100C    D2034000 700195FF 30024770 05F44280    ..... ..........   K. ..........4..
000840   30029198 30060779 43203000 4322C31D    48603000 95003000 47800874 95C03004    ..............C.   .-..............
000860   4770086E 9F006000 072947F0 08749F00    60000769 91003006 07194060 08849550    ......-....0....   -......... -...&
```

FIGURE 4-6 (Part 3) Test Run Output—Problem 2

```
         MCG                 12/13/73                                                                                          PAGE   6

004000   D9C5D6D9 C4C5D940 D2831014 10830A16    180607F7 00000000 00000000 00000000    REORDER K.......  ...7............
004020   00000000 --SAME--                                                              ....
006800   05300700 41103646 45003012 00006928    00006960 0A025810 36565800 365A58F1    ................  ...-...........1
006820   001045EF 0008FA20 363A3678 F224362E    3360F224 36313365 FA22362E 3631F224    ............2...  .-2...........2.
006840   3634335B F922362E 363447B0 30149240    3517D283 35183517 F2243637 3338D205    ...$9..........   ..K.....2.....K.
006860   35183621 DE053518 3637D213 3523333D    F2243637 3356D206 353B3627 DE06353B    ..........K.....  2.....K.........
006880   3637D205 35463621 DE053546 362ED205    35503621 DE053550 3634F911 363D3676    ..K...........K.  .&.....&..9.....
0068A0   474030DC 5810365E 58003662 58F10010    45EF000C 5810365E 58003666 58F10010    . ...........1..  ..............1..
0068C0   45EF000C 5810365E 5800366A 58F10010    45EF000C F810363D 367992F0 30DCF010    ..............1..  ....8.......0..0.
0068E0   365E5800 366E58F1 001045EF 000CFA10    363D3678 47F03014 DE06359D 363A5810    .......1........  .....0..........
006900   365E5800 367258F1 001045EF 000C0700    4110364E 4500311E 00006928 00006960    .......1........  ...............-
006920   0A020A0E C0C00000 0000C000 09000001    00006948 00006950 00006E80 02C50202    ................  .......&.....E..
006940   020069E0 000068F8 02006990 20000050    47000000 D24FD000 E0000000 00000000    .......8.......&  ....K...........
006960   00008000 08000003 00006988 00006990    00006F10 08B60B40 01006AB6 00000000    ................  ....... .......
006980   07004700 C0000000 01006A31 20000084    615C4040 40404040 40404040 40404040    ................  /*
0069A0   40404040 --SAME--
0069E0   F0F0F1F0 F3C8C5C1 E3C5D940 E2D6D3C5    D5D6C9C4 40404040 40F0F0F3 F3F0F0F0    00103HEATER SOLE  NOID      0033000
006A00   F4F4F0F0 F0F0F5F0 F0F0F0F3 F4F0F0F0    F0F04040 40404040 40404040 40404040    4400005000034000  00
006A20   40404040 40404040 40404040 40404040    404040D5 D6B4040 40404040 404040C4                          NO.         D
006A40   C5E2C3D9 C9D7E3C9 D6D54040 40404040    40404040 40D7D9C9 C3C54040 4040C1E5    ESCRIPTION             PRICE     AV
006A60   C1C9D3C1 C2D3C540 4040D7D6 C9D5E340    40404040 40404040 40404040 40404040    AILABLE   POINT
006A80   40404040 --SAME--
006AA0   40404040 40404040 40404040 40404040    40404040 40F040C9 E3C5D440 40404040                          0 ITEM
006AC0   40404040 404040C9 E3C5D440 40404040    40404040 40404040 404040E4 D5C9E340          ITEM                    UNIT
006AE0   40404040 40404040 40404040 4040D9C5    D6D9C4C5 D9404040 40404040 40404040                  RE  ORDER
006B00   40404040 --SAME--
006B20   40404040 40404040 40404040 40404040    40404040 40404040 4040F0F0 F1F0F3C8                                00103H
006B40   C5C1E3C5 D940E2D6 D3C5D5D6 C9C44040    404040F0 F0F3F3F0 F0F0F4F4 F0F0F0F0    EATER SOLENOID        0033000440000
006B60   F5F0F0F0 F0F3F4F0 F0F0F0F0 40404040    40404040 40404040 40404040 40404040    500003400000
006B80   40404040 40404040 4040F140 40404040    40404040 40404040 40404040 40404040              1
006BA0   404040D9 C5D6D9C4 C5D940D3 C9E2E3C9    D5C74040 40404040 40404040 40404040       REORDER LISTI  NG
006BC0   40404040 --SAME--
006C00   40404040 40404040 40404040 404040F0    40C9E3C5 D4404040 40404040 40404040                   0   ITEM
006C20   40C9E3C5 D4404040 40404040 40404040    40404040 40E4D5C9 E3404040 40404040     ITEM                  UNIT
006C40   40404040 40404040 D9C5D6D9 C4C5D940    40404040 40404040 40404040 40404040           REORDER
006C60   40404040 --SAME--
006C80   40404040 40404040 40404040 40404040    40404040 404040D5 D64B4040 40404040                             NO.
006CA0   404040C4 C5E2C3D9 C9D7E3C9 D6D54040    40404040 40404040 40D7D9C9 C3C54040       DESCRIPTION                PRICE
006CC0   4040C1E5 C1C9D3C1 C2D3C540 4040D7D6    C9D5E340 40404040 40404040 40404040      AVAILABLE   PO  INT
006CE0   40404040 --SAME--
006D00   40404040 40404040 40404040 40404040    40404040 40404040 40404040 40F1F0F3                                  103
006D20   40404040 40C8C5C1 E3C5D940 E2D6D3C5    D5D6C9C4 40404040 40404040 40404040         HEATER SOLE  NOID
006D40   F44BF4F0 40404040 40404040 F3F44040    40404040 4040F5F0 40404040 40404040    4.40          34          50
006D60   40404040 --SAME--
006D80   40404040 40404040 40404040 40404040    40404040 40404040 40404040 40406040
006DA0   20206B20 202040C3 C1D9C4E2 40C9D540    E3C8C540 C9D5D7E4 E340C4C5 C3D24040    ...... CARDS IN   THE INPUT DECK
006DC0   40404040 --SAME--
006E20   40404040 20202020 20402020 214B2020    00034C00 000F0005 0F00440F 00002C00         ..... ......  ..¤.............
006E40   0C204020 20214B20 5B5BC2D6 D7C5D540    5B5BC2C3 D3D6E2C5 00006928 00006B3A    .. .....$$BOPEN   $$BCLOSE........
006E60   00006960 00006B8A 00006C0F 00006C94    00006D19 00006D9E 050C1C0C 00004358    ...-............  ................
006E80   0A320000 0A320000 47F0F01A 0A320000    C9D1C3C6 E9C9C2F0 F3F99140 10154710    .........00.....  IJCFZIB039. ....
006EA0   F0289640 10150A00 91801002 4710F032    0A0750E0 F08C58E0 1020D202 10211019    0.. .........0.   ..&.0.....K.....
006EC0   50E01018 91011004 4780F066 91401002    4710F05C 58E0F08C 47F0F026 94BF1015    &.........0.. ..  ..0*..0..00.....
006EE0   58E0101C 07FED501 F086E000 4770F074    47F0F05C 50D0F088 18D04400 102C98DE    ......N.0.....0.  .00*&.0.........
006F00   F0880A00 07FE615C 00005FD0 A0006826    0A320000 0A320000 0A320000 47F0F01A    0...../*........  ..............00.
006F20   C9D1C4C6 C1E9C9E6 F3F99180 10024710    F0240A07 90CEF0B0 58E01018 06E018D0    IJDFAZIW39......  0.....0.........
006F40   D2001017 D00048C0 102E44C0 F0BC1BCC    18DC43C0 101741E0 001043DE F0C119DC    K...........0...  ..............0A..
006F60   4780F05A 46E0F04A 0A3243DE F0D143C0    101619CD 4770F080 91801016 4780F080    ..0...0......0J..  ......0........0.
006F80   920B1028 0A009180 10024710 F0800A07    42D01028 42D01016 0A009180 10024710    ............0...  ................
006FA0   F0940A07 92011028 58E01028 D2021029    101950E0 101898CE F0B00A00 07FE0000    0...........K...  ..&.....0......
006FC0   00004FD0 00005FD0 B00068D4 D200E000    D000F2C2 C1F9F8F7 F6F5F4F3 C3F14E60    ............MK...  ..2BA9876543C1.-
006FE0   F04093DB D3CBC3BB B3ABA39B E38B031B    130B0000 00000000 00000000 00000000    0 ..L.C.....T...  ................
007000   00000000 --SAME--                                                              ....
016FE0   00000000 00000000 00000000 00000000    00000000 00000000 00000000 00000000    ................  ................
```

FIGURE 4-6 (Part 4) Test Run Output—Problem 2

```
                                                                                                          PAGE    1

  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT                                         FDOS CL3-8 12/13/73

000000                                   1 REORDLST START 0
000000 0530                              2 BEGIN    BALR  3,0                    LOAD BASE REGISTER
000002                                   3          USING *,3
                                         4          OPEN  CARDIN,PRTOUT
                                         5+* 360N-CL-453 OPEN    CHANGE LEVEL 3-3                          3-3
000002 0700                              6+         CNOP  0,4
000004                                   7+         DC    0F'0'
000004 4110 3656             00658       8+         LA    1,=C'$$BOPEN '
000008 4500 3012             00014       9+IJJ00001 BAL   0,*+4+4*(3-1)
00000C 00000148                         10+         DC    A(CARDIN)
000010 00000180                         11+         DC    A(PRTOUT)
000014 0A02                             12+         SVC   2
                                        13 HEADING  CNTRL PRTOUT,SK,1            SKIP TO CHANNEL ONE
                                        14+* 360N-CL-453  CNTRL  CHANGE LEVEL  3-7
000016 5810 3666             00668      15+HEADING  L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
00001A 4100 008B             0008B      16+         LA    0,139 GET OPERATION CODE
00001E 58F1 0010             00010      17+         L     15,16(1) GET LOGIC MODULE ADDRESS
000022 05EF                             18+         BALR  14,15 BRANCH TO CNTRL ROUTINE
                                        19          PUT   PRTOUT,HDGLINE1          PRINT FIRST HEADING LINE
                                        20+* 360N-CL-453 PUT        CHANGE LEVEL 3-5                       3-5
000024 5810 3666             00668      21+         L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
000028 5800 366A             0066C      22+         L     0,=A(HDGLINE1) GET WORK AREA ADDRESS
00002C 58F1 0010             00010      23+         L     15,16(1) GET LOGIC MODULE ADDRESS      3-5
000030 45EF 000C             0000C      24+         BAL   14,12(15) BRANCH TO PUT ROUTINE          3-5
                                        25          CNTRL PRTOUT,SP,1            SPACE PRINTER ONE LINE
                                        26+* 360N-CL-453  CNTRL  CHANGE LEVEL  3-7
000034 5810 3666             00668      27+         L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
000038 4100 000B             0000B      28+         LA    0,11 GET OPERATION CODE
00003C 58F1 0010             00010      29+         L     15,16(1) GET LOGIC MODULE ADDRESS
000040 05EF                             30+         BALR  14,15 BRANCH TO CNTRL ROUTINE
                                        31          PUT   PRTOUT,HDGLINE2          PRINT SECOND HEADING LINE
                                        32+* 360N-CL-453 PUT        CHANGE LEVEL 3-5                       3-5
000042 5810 3666             00668      33+         L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
000046 5800 366E             00670      34+         L     0,=A(HDGLINE2) GET WORK AREA ADDRESS
00004A 58F1 0010             00010      35+         L     15,16(1) GET LOGIC MODULE ADDRESS      3-5
00004E 45EF 000C             0000C      36+         BAL   14,12(15) BRANCH TO PUT ROUTINE          3-5
                                        37          PUT   PRTOUT,HDGLINE3          PRINT THIRD HEADING LINE
                                        38+* 360N-CL-453 PUT        CHANGE LEVEL 3-5                       3-5
000052 5810 3666             00668      39+         L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
000056 5800 3672             00674      40+         L     0,=A(HDGLINE3) GET WORK AREA ADDRESS
00005A 58F1 0010             00010      41+         L     15,16(1) GET LOGIC MODULE ADDRESS      3-5
00005E 45EF 000C             0000C      42+         BAL   14,12(15) BRANCH TO PUT ROUTINE          3-5
                                        43          CNTRL PRTOUT,SP,1            SPACE PRINTER ONE LINE
                                        44+* 360N-CL-453  CNTRL  CHANGE LEVEL  3-7
000062 5810 3666             00668      45+         L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
000066 4100 000B             0000B      46+         LA    0,11 GET OPERATION CODE
00006A 58F1 0010             00010      47+         L     15,16(1) GET LOGIC MODULE ADDRESS
00006E 05EF                             48+         BALR  14,15 BRANCH TO CNTRL ROUTINE
                                        49 READCARD GET   CARDIN,CRDWRKA           READ CARD INTO WORK AREA
                                        50+* 360N-CL-453 GET     CHANGE LEVEL 3-0
000070 5810 3676             00678      51+READCARD L     1,=A(CARDIN) GET DTF TABLE ADDRESS
000074 5800 367A             0067C      52+         L     0,=A(CRDWRKA) GET WORK AREA ADDRESS
000078 58F1 0010             00010      53+         L     15,16(1) GET LOGIC MODULE ADDRESS
00007C 45EF 0008             00008      54+         BAL   14,8(15) BRANCH TO GET ROUTINE
000080 FA20 3653 368A 00655 0068C       55          AP    COUNT,=P'1'              ADD ONE TO COUNT
000086 F224 3647 337E 00649 00380       56          PACK  WRKAVAIL,CRDONHND
00008C F224 364A 3383 0064C 00385       57          PACK  WRKONORD,CRDONORD
000092 FA22 3647 364A 00649 0064C       58          AP    WRKAVAIL,WRKONORD        ADD ON HAND AND ON ORDER
000098 F224 364D 3379 0064F 0037B       59          PACK  WRKORDPT,CRDORDPT
00009E F922 3647 364D 00649 0064F       60          CP    WRKAVAIL,WRKORDPT
0000A4 4780 306E             00070      61          BNL   READCARD
0000A8 F224 3650 3356 00652 00358       62          PACK  PACKAREA,CRDITNBR
0000AE D205 3532 363A 00534 0063C       63          MVC   PRTITNBR,PATTERN1
0000B4 DE05 3532 3650 00534 00652       64          ED    PRTITNBR,PACKAREA        EDIT ITEM NUMBER FIELD
0000BA D213 353D 335B 0053F 0035D       65          MVC   PRTITDES,CRDITDES        MOVE ITEM DESCRIPTION
0000C0 F224 3650 336F 00652 00371       66          PACK  PACKAREA,CRDPRICE
0000C6 D206 3555 3640 00557 00642       67          MVC   PRTPRICE,PATTERN2
0000CC DE06 3555 3650 00557 00652       68          ED    PRTPRICE,PACKAREA        EDIT UNIT PRICE
0000D2 D205 3560 363A 00562 0063C       69          MVC   PRTAVAIL,PATTERN1
0000D8 DE05 3560 3647 00562 00649       70          ED    PRTAVAIL,WRKAVAIL        EDIT AVAILABLE
0000DE D205 356A 363A 0056C 0063C       71          MVC   PRTORDPT,PATTERN1
```

FIGURE 4-7 (Part 1) Test Run Output—Problem 3

```
                                                                                                                    PAGE    2

  LOC   OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                                                  FDOS CL3-8 12/13/73

0000E4 DE05 356A 364D 0056C 0064F    72           ED    PRTORDPT,WRKORDPT        EDIT ORDER POINT
                                     73           PUT   PRTOUT,PRTDETL           PRINT DETAIL LINE
                                     74+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                              3-5
0000EA 5810 3666             00668   75+          L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000EE 5800 367E             00680   76+          L     0,=A(PRTDETL) GET WORK AREA ADDRESS
0000F2 58F1 0010             00010   77+          L     15,16(1) GET LOGIC MODULE ADDRESS      3-5
0000F6 45EF 000C             0000C   78+          BAL   14,12(15) BRANCH TO PUT ROUTINE         3-5
                                     79           PRTOV PRTOUT,12,HEADING        IF END OF PAGE, GO TO HEADING
                                     80+* 360N-CL-453 PRTOV    CHANGE LEVEL 3-0
0000FA 5810 3666             00668   81+          L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
0000FE 5800 3682             00684   82+          L     0,=A(HEADING) GET ROUTINE ADDRESS
000102 58F1 0010             00010   83+          L     15,16(1) GET LOGIC MODULE ADDRESS
000106 45EF 0004             00004   84+          BAL   14,4(15) BRANCH TO PRTOV ROUTINE
00010A 47F0 306E             00070   85           B     READCARD                 GO TO READCARD
00010E DE06 35B6 3653 00588 00655    86 CRDEOF    ED    CNTPATRN,COUNT           EDIT COUNT
                                     87           CNTRL PRTOUT,SP,2              SPACE PRINTER TWO LINES
                                     88+* 360N-CL-453  CNTRL  CHANGE LEVEL  3-7
000114 5810 3666             00668   89+          L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
000118 4100 0013             00013   90+          LA    0,19 GET OPERATION CODE
00011C 58F1 0010             00010   91+          L     15,16(1) GET LOGIC MODULE ADDRESS
000120 05EF                          92+          BALR  14,15 BRANCH TO CNTRL ROUTINE
                                     93           PUT   PRTOUT,CNTLINE           PRINT COUNT LINE
                                     94+* 360N-CL-453 PUT          CHANGE LEVEL 3-5                              3-5
000122 5810 3666             00668   95+          L     1,=A(PRTOUT) GET DTF TABLE ADDRESS
000126 5800 3686             00688   96+          L     0,=A(CNTLINE) GET WORK AREA ADDRESS
00012A 58F1 0010             00010   97+          L     15,16(1) GET LOGIC MODULE ADDRESS      3-5
00012E 45EF 000C             0000C   98+          BAL   14,12(15) BRANCH TO PUT ROUTINE         3-5
                                     99           CLOSE CARDIN,PRTOUT
                                    100+* 360N-CL-453 CLOSE    CHANGE LEVEL 3-3                                 3-3
000132 0700                         101+          CNOP  0,4
000134                              102+          DC    0F'0'
000134 4110 365E             00660  103+          LA    1,=C'$$BCLOSE'
000138 4500 3142             00144  104+IJJC0013 BAL   0,*+4+4*(3-1)
00013C 00000148                     105+          DC    A(CARDIN)
000140 00000180                     106+          DC    A(PRTOUT)
000144 0A02                         107+          SVC   2
                                    108           EOJ
                                    109+* 360N-CL-453 EOJ      CHANGE LEVEL 3-0
000146 0A0E                         110+          SVC   14
                                    111 * THE CARD FILE DEFINITION FOLLOWS
                                    112 CARDIN    DTFCD DEVADDR=SYSIPT,IOAREA1=CRDIO1,IOAREA2=CRDIO2,            X
                                                        EOFADDR=CRDEOF,WORKA=YES
                                    113+* 360N-CL-453   DTFCD    CHANGE LEVEL 3-9                               3-9
000148                              114+          DC    0D'0'
000148 000080000000                 115+CARDIN    DC    X'000080000000' RES. COUNT,COM. BYTES,STATUS BTS
00014E 00                           116+          DC    AL1(0) LOGICAL UNIT CLASS
00014F 01                           117+          DC    AL1(1) LOGICAL UNIT
000150 00000168                     118+          DC    A(IJCX0015) CCW ADDRESS
000154 00000000                     119+          DC    4X'00' CCB-ST BYTE,CSW CCW ADDR.
000158 00                           120+          DC    AL1(0) 3-3
000159 000000                       121+          DC    VL3(IJCFZI80) ADDRESS OF LOGIC MODULE    3-3
00015C 02                           122+          DC    X'02' DTF TYPE (READER)
00015D 05                           123+          DC    AL1(5) SWITCHES
00015E 02                           124+          DC    AL1(2) NORMAL COMM.CODE
00015F 02                           125+          DC    AL1(2) CNTROL COMM.CODE
000160 00CC0200                     126+          DC    A(CRDIO2) ADDR. OF IOAREA2
000164 C000010E                     127+          DC    A(CRDEOF) EOF ADDRESS                                    3-8
000168 020001B020000050             128+IJCX0015 CCW   2,CRDIO1,X'20',80
000170 470C 0000             00000  129+          NOP   0 LOAD USER POINTER REG.
000174 D24F D000 E000 00000 00000   130+          MVC   0(80,13),0(14) MOVE IOAREA TO WORKA
00017A                              131+IJJZ0015 EQU   *
                                    132 * THE PRINTER FILE DEFINITION FOLLOWS
                                    133 PRTOUT    DTFPR DEVADDR=SYSLST,                                          X
                                                        IOAREA1=PRTIO1,                                          X
                                                        IOAREA2=PRTIO2,                                          X
                                                        WORKA=YES,                                               X
                                                        BLKSIZE=132,                                             X
                                                        PRINTOV=YES
                                    134+* 360 N-CL-453 DTFPR       CHANGE LEVEL 3-8                           3-8
00017A 000000000000
000180                              135+          DC    0D'0'
```

FIGURE 4-7 (Part 2) Test Run Output—Problem 3

```
                                                                                                                    PAGE    3

  LOC  OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                                                    FDOS CL3-8 12/13/73

000180 000084000400                  136+PRTOUT   DC    X'000084000400' RES. COUT,COM. BYTES,STATUS BTS
000186 00                            137+         DC    AL1(0) LOGICAL UNIT CLASS
000187 03                            138+         DC    AL1(3) LOGICAL UNIT
000188 000001A8                      139+         DC    A(*+32) CCW  ADDR.
00018C 00000000                      140+         DC    4X'00' CCB-ST BYTE,CSW CCW ADDRESS
000190 00                            141+         DC    AL1(0) 3-3
000191 000000                        142+         DC    VL3(IJDFZPIW) ADDR OF LOGIC MODUL3-8
000194 08                            143+         DC    X'08' DTF TYPE  (PRINTER)
000195 16                            144+         DC    AL1(22) SWITCHES
000196 09                            145+         DC    X'09' NORMAL  COMM. CODE
000197 09                            146+         DC    X'09' CONTROL COMM. CODE
000198 00000250                      147+         DC    A(PRTIO1+0) ADDRESS OF DATA IN IOAREA1
00019C 00000000                      148+         DC    4X'00' BUCKET                          3-5
0001A0 0700                          149+         NOPR  0 PUT LENGTH IN REG12 (ONLY UNDEF.
0001A2 4700 0000           00000     150+         NOP   0 LOAD USER POINTER REG
0001A6 0000                          151+         DC    2X'00' NOT USED                        3-5
0001A8 090002D420000084              152+         CCW   9,PRTIO2+0,X'20',132-0
0001B0                               153+IJJZ0016 EQU   *
                                     154 * THE DATA DEFINITIONS FOR THE TWO CARD I/O AREAS FOLLOW
0001B0                               155 CRDIO1   DS    CL80
000200                               156 CRDIO2   DS    CL80
                                     157 * THE DATA DEFINITIONS FOR THE TWO PRINTER  AREAS FOLLOW
000250                               158 PRTIO1   DS    CL132
0002D4                               159 PRTIO2   DS    CL132
000358                               160 CRDWRKA  DS    0CL80
                                     161 * THE DATA DEFINITIONS FOR THE CARD WORK AREA FOLLOW
000358                               162 CRDITNBR DS    CL5
00035D                               163 CRDITDES DS    CL20
000371                               164 CRDPRICE DS    CL5
000376                               165          DS    CL5
00037B                               166 CRDORDPT DS    CL5
000380                               167 CRDONHND DS    CL5
000385                               168 CRDONORD DS    CL5
00038A                               169          DS    CL30
                                     170 * THE DATA DEFINITIONS FOR THE PRINTER HEADING LINES FOLLOW
0003A8                               171 HDGLINE1 DS    0CL132
0003A8 4040404040404040              172          DC    24C' '
0003C0 D9C5D6D9C4C5D940              173          DC    C'REORDER LISTING'
0003CF 4040404040404040              174          DC    93C' '
00042C                               175 HDGLINE2 DS    0CL132
                                     176          DC    C' ITEM             ITEM                 UNIT              X
00042C 40C9E3C5D4404040                              REORDER'
00046B 4040404040404040              177          DC    69C' '
0004B0                               178 HDGLINE3 DS    0CL132
                                     179          DC    C'  NO.          DESCRIPTION            PRICE    AVAILABLEX
0004B0 4040D5D64B404040                              POINT'
0004EE 4040404040404040              180          DC    70C' '
                                     181 * THE DATA DEFINITIONS FOR THE PRINTER DETAIL LINE FOLLOW
000534                               182 PRTDETL  DS    0CL132
000534                               183 PRTITNBR DS    CL6
00053A 4040404040                    184          DC    5C' '
00053F                               185 PRTITDES DS    CL20
000553 40404040                      186          DC    4C' '
000557                               187 PRTPRICE DS    CL7
00055E 40404040                      188          DC    4C' '
000562                               189 PRTAVAIL DS    CL6
000568 40404040                      190          DC    4C' '
00056C                               191 PRTORDPT DS    CL6
000572 4040404040404040              192          DC    70C' '
                                     193 * THE DATA DEFINITIONS FOR THE FINAL TOTAL LINE FOLLOW
0005B8                               194 CNTLINE  DS    0CL132
0005B8 4020206B202020                195 CNTPATRN DC    X'4020206B202020'
0005BF 40C3C1D9C4E240C9              196          DC    C' CARDS IN THE INPUT DECK'
0005D7 4040404040404040              197          DC    101C' '
                                     198 * THE DATA DEFINITIONS THAT FOLLOW DEFINE OTHER WORK AREAS NEEDED
                                     199 * BY THE PROGRAM
00063C 402020202020                  200 PATTERN1 DC    X'402020202020'
000642 402020214B2020                201 PATTERN2 DC    X'402020214B2020'
000649                               202 WRKAVAIL DS    PL3
00064C                               203 WRKONORD DS    PL3
00064F                               204 WRKORDPT DS    PL3
```

FIGURE 4-7 (Part 3) Test Run Output—Problem 3

```
                                                                    PAGE    4

  LOC   OBJECT CODE     ADDR1 ADDR2  STMT   SOURCE STATEMENT                FDOS CL3-8 12/13/73

000652                                205 PACKAREA DS     PL3
000655 00000C                         206 COUNT    DC     PL3'0'
000000                                207          END    BEGIN
000658 5B5BC2D6D7C5D540               208                 =C'$$BOPEN '
000660 5B5BC2C3D3D6E2C5               209                 =C'$$BCLOSE'
000668 00000180                       210                 =A(PRTOUT)
00066C 000003A8                       211                 =A(HDGLINE1)
000670 0000042C                       212                 =A(HDGLINE2)
000674 000004B0                       213                 =A(HDGLINE3)
000678 00000148                       214                 =A(CARDIN)
00067C 00000358                       215                 =A(CRDWRKA)
000680 00000534                       216                 =A(PRTDETL)
000684 00000016                       217                 =A(HEADING)
000688 000005B8                       218                 =A(CNTLINE)
00068C 1C                             219                 =P'1'
```

FIGURE 4-7 (Part 4) Test Run Output—Problem 3

```
12/13/73    PHASE   XFR-AD  LOCORE  HICORE  DSK-AD    ESD TYPE  LABEL     LOADED  REL-FR

         PHASE***   004000  004000  0047C5  5D 11 4   CSECT     REORDLST  004000  004000

                                                      CSECT     IJCFZIB0  004690  004690

                                                      CSECT     IJDFZPIW  004720  004720
                                                      *  ENTRY  IJDFZZIW  004720
```

FIGURE 4-7 (Part 5) Test Run Output—Problem 3

```
0S04I ILLEGAL SVC - HEX LOCATION 004720 - SVC CODE 32
0S00I JOB MCQ       CANCELED
```

FIGURE 4-7 (Part 6) Test Run Output—Problem 3

Solutions

1 a. No test case has the available stock exactly equal to the reorder point.
   b. Page overflow isn't tested.
   c. Maximum values aren't tried in the numeric fields.
   d. Minimum values—like a 5 cent item—aren't tried in the unit price field.
   e. A full 20 character name isn't tried in the description field.

2 Statement 60 has moved a zero into the first instruction generated by the PUT macro for the detail line. Operand-1 of this instruction should be PRTDCTL, not PRTDET. This problem illustrates that once you find the instruction that caused the operation exception, it requires logic to discover how the instruction was altered. In this case, you can check the instruction in the storage dump, find hex F0 in its first byte, and question which other instruction might have moved this data into the instruction. In some cases, you must go through the program listing, instruction by instruction, until you find the one that caused the problem.

3 The DTFPR macro, statement 133, was coded without the operand CONTROL=YES even though the CNTRL macro was used for the file. As a result, the wrong I/O module was link-edited. To correct the problem, the operand must be added to the DTFPR and the program reassembled. This time a different I/O module will be linked.

5

Expanding the Basic Subset

This chapter expands the basic BAL subset presented in chapter 3 by presenting some techniques and coding commonly used by professional programmers. Although the elements presented don't involve any additional computing functions, they do make it possible for you to code more efficiently. Because the elements presented here are in common use in industry, this material will also help you to understand programs written by professionals. After you complete this chapter, you will be able to read any of the chapters in part 3.

TOPIC ONE **Register Operations** In chapter 3, you learned how a register is used as a base register for addressing. In addition, registers are useful in performing repetitive operations and in controlling the flow of a program. This topic describes how to use registers for these two purposes.

On the System/360, there are about 58 instructions that involve registers. The System/370 offers at least 5 more. Take heart, though, in this chapter only 18 register instructions are presented. This group of instructions represents those most widely used by professional programmers.

In general, register instructions have the following format:

```
OPCODE  OPERAND1,OPERAND2
```

Here, OPERAND1 is the number of a register and is the receiving operand; OPERAND2 is either a label representing a fullword of storage or a register number. In System/360 instructions, the storage fields referred to must be at fullword boundaries, but this is not required for System/370 instructions. A few of the register instructions have three operands, in which case operand-2 is a register number and operand-3 is a label for a fullword of storage.

Since register instructions commonly operate on fullwords of storage, and occasionally on doublewords, you must know how to define fullwords and doublewords. The type codes for these fields are F and D as in these examples:

```
SAMPLEF1 DC    F'1'
SAMPLEF2 DS    4F
SAMPLEF3 DC    F'-125'
SAMPLED1 DS    D
SAMPLED2 DS    3D
```

Because the length is always four bytes for a fullword and eight for a doubleword, no length factor is required. On the System/360, a fullword or doubleword is given a location counter value that will lead to proper boundary alignment if the program is loaded starting at a doubleword boundary (as is always the case). Duplication factors can be used as required, and the nominal value is a signed or unsigned decimal number. Thus, SAMPLEF1 will have a starting binary value of +1; SAMPLEF3 will have a starting value of minus 125. It is important to note, however, that a nominal value cannot be used with type code D since it will cause a floating-point constant to be assembled as described in chapter 6.

LOADING AND STORING REGISTERS

The first step to take in using a register is to get some data into it. Here are four machine instructions used for that purpose.

Load (L) The load instruction can load data from any fullword into any one of the 16 registers as in this example:

```
L     4,FWORD
```

This instruction places the contents of the fullword named FWORD into register 4. The first operand can be any register number from 0 through 15; the second operand can be the name of any fullword.

Load Register (LR) A register can also be loaded from another register as in this load-register instruction:

```
LR    4,7
```

Here, the second operand is also the number of one of the 16 registers. The effect of the instruction is to duplicate the contents of register 7 in register 4.

Load Multiple (LM) The load-multiple instruction will load two or more consecutive registers from an equal number of consecutive words of storage as in this example:

```
LM    4,6,WORD1
```

In this case, registers 4, 5, and 6 are loaded with the data in the 12-byte area beginning with the byte referred to by the name WORD1. Register 4 is loaded with bytes 1 through 4, register 5 with bytes 5 through 8, and register 6 with bytes 9 through 12. This can be illustrated as follows:

```
        LM    4,6,WORD1
        .
        .
        .
WORD1   DC    F'1'
WORD2   DC    F'2'
WORD3   DC    F'3'
```

	Before	After
Register 4:	00 56 0A 71	00 00 00 01
Register 5:	00 00 0B AA	00 00 00 02
Register 6:	34 B1 CC 00	00 00 00 03

In any assembler-language instruction that refers to multiple registers, the *wrap-around* concept applies. Wrap-around means that the register numbers form an endless sequence with register 0 following register 15. A load-multiple instruction, for example, can be coded like this:

```
LM      14,1,FOURWORD
```

Then, register 14 will be loaded with the first fullword of storage beginning at FOURWORD, register 15 with the second word, register 0 with the third, and register 1 with the fourth.

Load Address (LA) The last of the four load instructions is the load-address instruction which can be coded as:

```
LA      6,ANYDATA
```

Here, register 6 is loaded with the *address* of ANYDATA, not the data stored there. Suppose, for example, that the label ANYDATA refers to a two-byte field containing hex C1C2 at location 004B60. After the instruction has been executed, register 6 contains hex 004B60, the address referred to by the label. The address can be any byte in storage—it doesn't have to be at a fullword boundary.

Store (ST) and Store Multiple (STM) The store and store-multiple instructions provide for saving the contents of registers in one or more fullwords of storage. These instructions are two of the cases in which the first operand is the sending operand and the second is the receiving operand. (This, of course, is the opposite of the normal instruction format.) For instance, this store instruction places the contents of register 14 in the fullword named SAVE14:

```
ST      14,SAVE14
```

In store-multiple instructions, the contents of consecutive registers are stored in consecutive words:

```
STM     7,10,SAVE7
```

In this example, the contents of registers 7, 8, 9, and 10 are stored in the four fullwords beginning at the byte addressed by the label SAVE7.

The wrap-around concept applies to the store-multiple instruction also. For instance, this instruction is commonly used to save all the registers except number 13:

```
STM     14,12,SAVEAREA
```

In this case, 15 registers are stored in this sequence: 14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.

REGISTER ARITHMETIC AND COMPARISON

The four basic arithmetic functions—add, subtract, multiply, and divide—can all be done using register instructions. Since you only need addition and subtraction in order to manipulate addresses for repetitive processing, this chapter only describes the add and subtract instructions. The multiply and divide instructions are covered in chapter 6.

The add and subtract instructions are available in the register-to-storage and the register-to-register forms. In each case, the first operand is a register number and is the operand that will receive the result. The second operand can be a fullword in storage or it can be another register.

Here are some examples of the register-to-storage instructions:

```
A       6,BCON4
S       9,BCON4
```

If BCON 4 is a fullword containing a value of four, four is added to the contents of register 6 in the first example and subtracted from the contents of register 9 in the second example. In each case, the result replaces the original contents of the register specified.

In the register-to-register form, the contents of the second register are added to or subtracted from the contents of the first register as shown in these examples:

```
AR      6,7
SR      9,8
```

Here, the binary value in register 7 is added to the value in register 6, and the value in register 8 is subtracted from the value in register 9. The result replaces the contents of the first register.

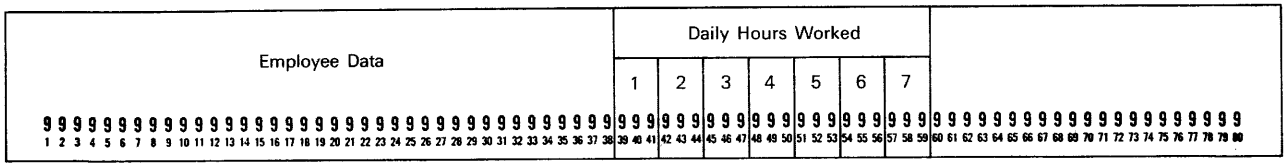
The register-compare instruction is available for comparing results stored in registers. It can be coded in register-to-storage or register-to-register form:

```
C       3,BCON50
CR      3,12
```

The first instruction compares the binary value in register 3 with the binary value stored in the fullword labeled BCON50. The second instruction compares the value in register 3 with the value in register 12. As in the case of the compare-decimal or compare-logical instructions, you would expect to find these compare instructions followed by a conditional branch instruction. A branch-high (BH) instruction, for example, would cause a branch only if the value in register 3 were greater than the value in BCON50 in the first example, or greater than the value in register 12 in the second example.

EXPLICIT ADDRESSES AND LENGTHS

To illustrate the use of register operations for repetitive processing, consider this problem. A payroll program reads an input card that has this format:



To calculate the number of hours an employee worked during the week, one of the routines of the program must add the seven daily-hours-worked fields. Although this routine could be done using the chapter 3 subset, figure 5-1 shows how to do it more efficiently by using register operations.

The first three instructions of the routine set up registers 4 and 6 for repetitive processing and set the weekly-hours field to zero. As you will see, register 6 is used as a substitute base register for the daily-hours-worked fields, while register 4 is used to control the number of times the ADDLOOP routine is repeated. The SR instruction, then, sets register 4 to zero by subtracting the contents of register 4 from itself. The second instruction, the LA instruction, places the address of the first daily-hours-worked field in register 6. And the third instruction, the ZAP instruction, zeros the weekly-hours-worked field, WKLYHRS.

The next two instructions pack the hours-worked fields and add them to the field named WKLYHRS. These instructions will be repeated seven times, once for each hours-worked field in the input card. Because the instructions use *explicit addresses* and *lengths*, the same instructions will reference all seven input fields.

In chapter 3, you were introduced to explicit lengths—namely, numbers in parentheses following labels or relative addresses that indicate the length of the operand. In addition, though, the assembler-language programmer can specify explicit base registers and explicit displacements. This is illustrated by the first instruction in the ADDLOOP routine:

```
ADDLOOP  PACK  0(3,6),0(3,6)
```

displacement — length — base register

base register — length — displacement

Since the address of the data from card column 39 has been stored in register 6, this instruction packs the three-byte field at the address in base register 6 to the three-byte field at the address in base register 6. In other words, it packs the input field in its own area. Since the displacement value is zero, the base address is not modified.

The next instruction also illustrates explicit addressing:

```
AP      WKLYHRS,0(3,6)
```

This adds the three-byte field at the address in base register 6 to the field named WKLYHRS.

The next four instructions cause the ADDLOOP to be executed six more times:

```
A       4,BCON1
A       6,BCON3
C       4,BCON7
BL      ADDLOOP
```

First, 1 is added to the counter, register 4. Second, 3 is added to the base register, register 6. Third, register 4 is compared with a value of 7. Finally, a branch to ADDLOOP takes place if the content of register 4 is less than the value 7. Since register 4 starts with a value of zero, the branch to ADDLOOP will take place six times for each input card. Since the content of register 6 is increased by three each time it goes through the loop, the PACK and AP instructions will operate on successive card fields each time they go through the loop. If, for example, the base address is 6000 the first time the PACK instruction is executed, it will be 6003 the next time through.

Figure 5-2 gives the instruction formats for using explicit base registers, displacements, and lengths in all of the instructions presented in chapters 1 through 5 of this book. The IBM green card gives the explicit formats for all System/360-370 instructions. In general, though, an operand in storage can be expressed as a displacement (D), length (L), and base register (B) in this form:

```
D(L,B)
```

That is, a number representing a displacement is followed by

```
          .
          .
          .
          SR      4,4
          LA      6,CARDIN+38
          ZAP     WKLYHRS,=P'0'
ADDLOOP   PACK    0(3,6),0(3,6)
          AP      WKLYHRS,0(3,6)
          A       4,BCON1
          A       6,BCON3
          C       4,BCON7
          BL      ADDLOOP
          .
          .
          .
CARDIN    DS      CL80
WKLYHRS   DS      PL3
BCON1     DC      F'1'
BCON3     DC      F'3'
BCON7     DC      F'7'
```

FIGURE 5-1 Payroll Routine Using Register Operations

two numbers in parentheses, the first being the length factor, the second being the number of the base register.

Other codes used in figure 5-2 are I for an immediate operand and X for an index register. Since the intended use of an index register is rarely used and is not covered in this book, you would normally code a zero for the index register. Thus, an explicit operand in the fullword add instruction could be specified as:

```
A       4,8(0,5)
```

Here, the fullword at the address given in register 5 plus a displacement of 8 is added to register 4.

As an alternative, you could let the index register serve as the base register when using explicit addresses. Thus, the add instruction above could be written as:

```
A       4,8(5)
```

Name	Code	Type	Operands
Add	AR	RR	R1,R2
Add	A	RX	R1,D2(X2,B2)
Add Decimal	AP	SS	D1(L1,B1),D2(L2,B2)
Compare	CR	RR	R1,R2
Compare	C	RX	R1,D2(X2,B2)
Compare Decimal	CP	SS	D1(L1,B1),D2(L2,B2)
Compare Logical	CLC	SS	D1(L,B1),D2(B2)
Compare Logical	CLI	SI	D1(B1),I2
Convert to Binary	CVB	RX	R1,D2(X2,B2)
Convert to Decimal	CVD	RX	R1,D2(X2,B2)
Divide Decimal	DP	SS	D1(L1,B1),D2(L2,B2)
Edit	ED	SS	D1(L,B1),D2(B2)
Load	LR	RR	R1,R2
Load	L	RX	R1,D2(X2,B2)
Load Address	LA	RX	R1,D2(X2,B2)
Load Multiple	LM	RS	R1,R3,D2(B2)
Move	MVC	SS	D1(L,B1),D2(B2)
Move	MVI	SI	D1(B1),I2
Move Numeric	MVN	SS	D1(L,B1),D2(B2)
Move with Offset	MVO	SS	D1(L1,B1),D2(L2,B2)
Move Zones	MVZ	SS	D1(L,B1),D2(B2)
Multiply Decimal	MP	SS	D1(L1,B1),D2(L2,B2)
Pack	PACK	SS	D1(L1,B1),D2(L2,B2)
Store	ST	RX	R1,D2(X2,B2)
Store Multiple	STM	RS	R1,R3,D2(B2)
Subtract	SR	RR	R1,R2
Subtract	S	RX	R1,D2(X2,B2)
Subtract Decimal	SP	SS	D1(L1,B1),D2(L2,B2)
Unpack	UNPK	SS	D1(L1,B1),D2(L2,B2)
Zero and Add	ZAP	SS	D1(L1,B1),D2(L2,B2)

FIGURE 5-2 Instruction Formats for Explicit Operands

This means that the fullword addressed by the sum of register 5 and the displacement is added to register 4. Since no base register is specified, the index register operates as the base register.

When coding explicit operands, you must remember that the formats for different instructions have different parts. For instance, the register and immediate instructions have no length factors and the MVC instruction has a length factor on operand-1 only. Because each explicit value is critical to the resulting object code, you must check the formats to be sure each explicit value means what you think it means.

CONTROLLING PROGRAM FLOW

Registers can be used for program control in two ways. First, they can be used for linking between parts of a program. The link is usually made by saving the *return address*, then branching. The return address is the address of the instruction to be executed when control is returned to the mainline routine.

The branch-and-link instruction performs both of these functions: it saves the address of the instruction that follows and branches to the address specified. This instruction is available in register-to-storage and register-to-register forms as in these examples:

```
BAL   11,SEARCH
BALR  11,9
```

When executed, the address of the instruction following the branch-and-link is saved in the register specified in the first operand. Then, a branch is made to the address specified by the second operand. In the BAL format, the second operand is an instruction label. In the BALR format, the second operand is a register containing a branch address.

To avoid repeating similar code in multiple parts of a program, it is often convenient to code parts of a program as *subroutines*. A subroutine is simply a group of instructions outside of the mainline routine that is designed to perform some specific task. For instance, a print routine is often coded as a subroutine. If you had four or five different types of print lines to construct and print, you could avoid repeating the series of statements that check for page overflow, PUT the line to the printer, and clear the print-line area to blanks by coding them as a subroutine. Then, in each

of the routines that construct the various print lines you would code a branch-and-link, perhaps like this:

```
BAL   11,PRINT
```

This would cause a branch to the print subroutine. Then, to return to the mainline routine, the last statement in the subroutine would be:

```
BR    11
```

The branch-register (BR) instruction is an unconditional branch to the address given in the register—in this case, register 11. This notion of branching to and from a subroutine is expanded in chapter 9 and is illustrated in figure 9-2.

The second way registers are used for program control is as counters—that is, to count the number of times a routine is executed (this is shown in figure 5-1). When you want to use registers in this way, you can take advantage of the branch-on-count instruction as in these examples:

```
BCT   4,LOOP
BCTR  4,9
```

Each time the branch-on-count instruction is executed, the value in the register specified as the first operand is reduced by one. When the resulting value in the register is zero, control falls through to the next instruction in sequence. Otherwise, a branch is made to the instruction given in the second operand. The second operand is an instruction label in the BCT format and a register containing an instruction address in the BCTR format.

Figure 5-3 shows how the branch-on count instruction can be used to control the payroll routine initially presented in figure 5-1. In this case, the BCT instruction is used in place of the add, compare, and branch-low instructions of figure 5-1. The first time through the loop, register 4 is decreased from its initial value of seven to six. Since the value is not yet zero, the branch to ADDLOOP takes place. On the seventh time through the loop, the BCT instruction will again reduce register 4 by one. This time the result will be zero so the branch to ADDLOOP will not occur and control will

```
          .
          .
          L     4,BCON7
          LA    6,CARDIN+38
          ZAP   WKLYHRS,=P'0'
ADDLOOP   PACK  0(3,6),0(3,6)
          AP    WKLYHRS,0(3,6)
          A     6,BCON3
          BCT   4,ADDLOOP
          .
          .
          .
CARDIN    DS    CL80
WKLYHRS   DS    PL3
BCON3     DC    F'3'
BCON7     DC    F'7'
```

FIGURE 5-3 Payroll Routine with Branch-on-Count Refinement

pass to the next instruction in sequence.

The payroll routine can be simplified one more step by using the load-address instruction, in place of the load instruction, to load initial values into registers. This technique, shown in figure 5-4, eliminates the need to define constants. For example, to load the initial value of seven into register 4, you can code a load-address instruction as:

```
LA    4,7
```

This loads the value 7, as though it were an address, into register 4. If you check the operand formats in figure 5-2, you will see that the 7 is treated as though it were the displacement for an operand with no index and no base register.

In figure 5-4, the load-address instruction is also used to increase the value in a register by a positive value:

```
LA    6,3(6)
```

If you look at the formats in figure 5-2 again, you will see

```
                .
                .
                .
                LA      4,7
                LA      6,CARDIN+38
                ZAP     WKLYHRS,=P'0'
ADDLOOP         PACK    0(3,6),0(3,6)
                AP      WKLYHRS,0(3,6)
                LA      6,3(6)
                BCT     4,ADDLOOP
                .
                .
                .
CARDIN          DS      CL80
```

FIGURE 5-4 Payroll Routine with Load-Address Refinement

that the address in index register 6 plus a displacement of three is to be placed in register 6. This, then, has the effect of increasing the contents of register 6 by three. You can increase the value in a register by any fixed amount with a load-address instruction coded in this way.

BINARY DATA CONVERSIONS

The pack and unpack instructions offer a way to convert numeric data from EBCDIC format to packed-decimal, and vice versa. Two instructions in the register group—convert-to-binary and convert-to-decimal—provide conversion capability between packed-decimal and binary formats. (There is no direct conversion between EBCDIC and binary. EBCDIC data must first be packed, then converted to binary.)

The convert-to-binary (CVB) instruction converts a packed-decimal value into a binary value. The packed-decimal number must be stored in a doubleword area and the resulting binary value is stored in a register. On System/360, the doubleword packed-decimal field must begin on a doubleword boundary, but this is not necessary on the System/370.

Here is a sample CVB instruction:

```
                CVB     12,DBLEPACK
```

The packed-decimal value stored in the doubleword named DBLEPACK is converted to binary and loaded into register 12. As in all packed-decimal instructions, the data in the area named DBLEPACK must have a valid sign in the rightmost half-byte and decimal digits in all other half-bytes. If not, a data exception program check will occur. Since packed-decimal fields are not normally defined as doubleword areas, you would usually define a doubleword work area and use the zero-and-add (ZAP) instruction to insert your packed decimal value.

The convert-to-decimal (CVD) instruction is the opposite of the convert-to-binary. It converts a binary value in a register to packed-decimal format and stores it in a doubleword area. The CVD is another of the instructions in which operand-2 is the receiving operand:

```
                CVD     10,DBLEAREA
```

Here, the binary value in register 10 is packed into the doubleword named DBLEAREA.

Just for illustration, figure 5-5 is the payroll routine shown earlier in another form. This time the input data is converted to binary and the sum of hours is computed in binary. The total is then converted back to packed decimal. After each of the daily-hours-worked fields has been packed into the doubleword area, it is converted to binary in register 7 and then added to the weekly-hours total in register 5. When the hours for all seven days have been accumulated, control falls through the BCT instruction and the total hours in register 5 is converted back to packed decimal. The CVD instruction makes use of the same doubleword that had been used in the CVB instruction.

HALFWORD INSTRUCTIONS

Although they are infrequently used, you should also be aware that there are instructions that operate on halfwords—in particular, load halfword (LH), store halfword (STH), add

halfword (AH), and subtract halfword (SH). These instructions operate like the fullword instructions except that operand 2 is always a halfword. When a halfword value is loaded into, added to, or subtracted from a register, however, all four bytes of the register are used. Because these instructions duplicate the functions of the fullword instructions, they are only needed in some special cases, a couple of which are illustrated later in this book.

To define a halfword field, the type code H is used. This type code will cause two bytes of storage with proper boundary alignment to be assembled as in these examples:

```
DS    H
DS    4H
DC    H'-1'
DC    H'+420'
```

Terminology

wrap-around
explicit address
explicit length
return address
subroutine

Objective

Apply the register techniques and instructions presented in this topic to appropriate aspects of a programming problem.

Problem

A deck of cards contains temperatures. Each card can have a maximum of 40 two-column temperatures that can be above or below zero. However, if two columns that should contain a temperature are blank, the rest of the card will be blank also. Thus, a card can have from 1 to 40 temperatures recorded in it. Write a routine that adds the temperatures in each card, assuming the cards are read into an area defined as follows:

```
TEMPS     DS     CL80
```

```
          .
          .
          .
          SR     5,5
          LA     4,7
          LA     6,CARDIN+38
ADDLOOP   PACK   DBLEWRK,0(3,6)
          CVB    7,DBLEWRK
          AR     5,7
          LA     6,3(6)
          BCT    4,ADDLOOP
          CVD    5,DBLEWRK
          ZAP    WKLYHRS,DBLEWRK
          .
          .
          .
CARDIN    DS     CL80
DBLEWRK   DS     D
WKLYHRS   DS     PL3
```

FIGURE 5-5 Payroll Routine with Data Conversions

Use register operations and the branch-on-count instruction to control the repetitive processing. Perform the addition in binary so the sum of the temperatures for a card is stored in a fullword defined as follows:

```
CARDSUM   DS     F
```

Solution

Figure 5-6 is an acceptable solution. Register 4 is used as the counter, register 5 as the base register, register 6 as the accumulator for the temperature sum, and register 7 as the conversion register for input values.

TOPIC TWO Storage Definition Techniques

Chapter 3 presented the basic methods of defining I/O areas, work areas, and work fields. However, there are some other

```
            .
            .
            .
            LA     4,40
            LA     5,TEMPS
            SR     6,6
ADDTEMPS    CLC    0(2,5),=X'4040'
            BE     ENDRT
            PACK   DBLEWRK,0(2,5)
            CVB    7,DBLEWRK
            AR     6,7
            LA     5,2(5)
            BCT    4,ADDTEMPS
ENDRT       ST     6,CARDSUM
            .
            .
            .
TEMPS       DS     CL80
DBLEWRK     DS     D
CARDSUM     DS     F
```

FIGURE 5-6 Temperature Routine

elements and techniques that can lead to more efficient coding. In particular, this chapter reviews the use of zero duplication factors and introduces the ORG instruction and dummy sections.

ZERO DUPLICATION FACTORS

Figure 5-7 illustrates how zero duplication factors are used for referring to various portions of an accounts receivable card. In other words, fields can be defined within fields that are within areas. In fact, any number of levels within levels can be defined. Thus, BALFWD refers to the 13 bytes that hold data from card columns 58-70; BALDATE refers to the 6 bytes that hold data from card columns 58-63; and YEAR refers to the 2 bytes that hold data from card columns 62 and 63. Whenever a zero duplication factor is used, the location counter value does not increase during the assembly.

THE ORG INSTRUCTION

Often an input file will contain several types of records. For example, the input to a billing program, as shown in figure 5-8, might contain a set of cards for each shipment made. The first card of each set contains the customer name and billing address and has a 1 in column 80. The second card specifies the date of shipment and the shipping address and has a 2 in column 80. And for each item sent on a shipment, there is a third type of card; it specifies the identification number and quantity of one item of the shipment.

Rather than use three separate work areas for these cards, the ORG instruction can be used to define all three formats in a single area. This is illustrated in figure 5-9. When the ORG instruction is specified with an operand, it tells the assembler to set the location counter to the location counter value of the operand. Thus, CARD2 and its fields are given location values that are in the CARD1 area. Similarly, CARD3 and its fields are given location values in the CARD1 area. The final ORG instruction, the ORG instruction without an operand, tells the assembler to set the location counter to a location value representing the next available location—in this case, the first byte following the card-input area.

The operand for the ORG statement can consist of names, decimal values, or the asterisk (which stands for the current location counter value) and can be combined in arithmetic expressions also. Thus, the first ORG statement in figure 5-9 would have the same effect if coded as

```
ORG    *-80
```

This means the current location counter value minus 80. Similarly, the second ORG instruction would have the same

```
          .
          .
ARCARD    DS    0CL80
CUSTID    DS    0CL20
CUSTNBR   DS    CL5
CUSTNAME  DS    CL15
CUSTADDR  DS    0CL37
STREET    DS    CL15
CITY      DS    CL15
STATE     DS    CL2
ZIP       DS    CL5
BALFWD    DS    0CL13
BALDATE   DS    0CL6
MONTH     DS    CL2
DAY       DS    CL2
YEAR      DS    CL2
BALAMT    DS    ZL7
          DS    CL10
```

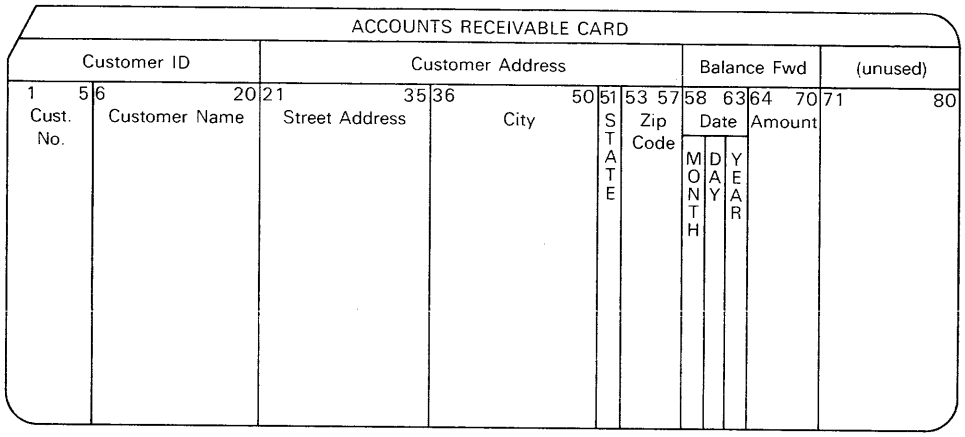


FIGURE 5-7 Use of Zero Duplication Factor

effect if coded as any one of the following:

```
ORG   CARD1
ORG   CARD2
ORG   *-80
ORG   C1IDNUM
ORG   C2IDNUM
ORG   C2DATE-4
```

Here's another example that illustrates more dramatically the effect of the ORG instruction without an operand:

```
LOC                STATEMENT
00A4    CARD1      DS    CL80
00F4    PRT1       DS    CL132
0178    PRT2       DS    CL132
00A4               ORG   CARD1
00A4    CARD2      DS    CL80
00A4               ORG   CARD2
00A4    CARD3      DS    CL80
01FC               ORG
```

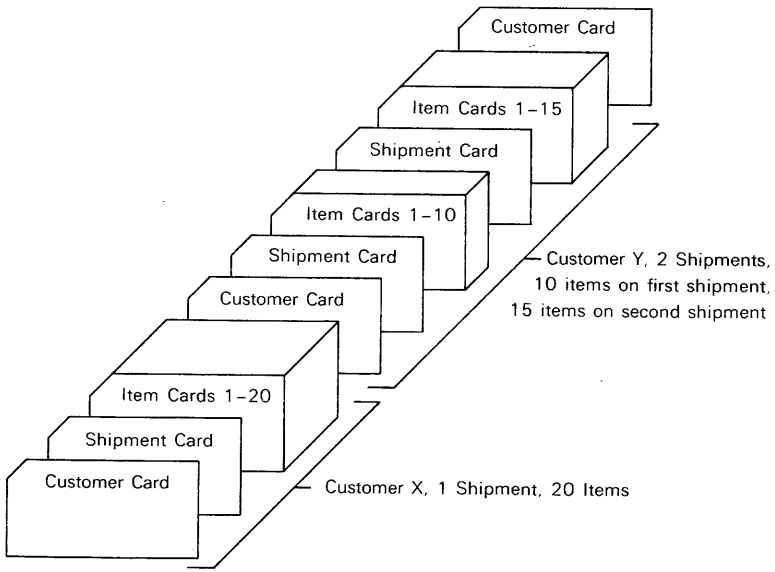


FIGURE 5-8 Multiple Input Card Formats

```
LOC CTR VALUE          STATEMENT
02A0              CARD1     DS     0CL80
02A0              C1IDNUM   DS     CL4
02A4              C1NAME    DS     CL25
02BD              C1STRT    DS     CL15
02CC              C1CITY    DS     CL15
02DB              C1STATE   DS     CL2
02DD              C1ZIP     DS     CL5
02E2                        DS     CL13
02EF              C1TYPE    DS     CL1
02A0                        ORG    CARD1
02A0              CARD2     DS     0CL80
02A0              C2IDNUM   DS     CL4
02A4              C2DATE    DS     CL6
02AA                        DS     CL19
02BD              C2ADDR1   DS     CL15
02CC              C2ADDR2   DS     CL15
02DB              C2ADDR3   DS     CL2
02DD              C2ADDR4   DS     CL5
02E2                        DS     CL13
02EF              C2TYPE    DS     CL1
02A0                        ORG    CARD1
02A0              CARD3     DS     0CL80
02A0              C3IDNUM   DS     CL4
02A4              C3DATE    DS     CL6
02AA              C3ITEM    DS     CL8
02B2              C3QTY     DS     CL6
02B8                        DS     CL65
02EF              C3TYPE    DS     CL1
02F0                        ORG
```

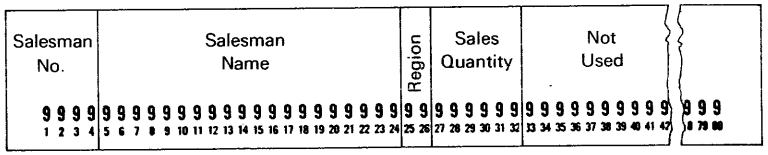
FIGURE 5-9 Use of ORG Instruction

Two print line areas follow the first card area and advance the location counter value 132 bytes each. The definitions of the second and third card formats are preceded by ORG instructions, so they redefine the first card area. The last ORG, with no operand, resets the location counter to the next available byte. This would be the first byte beyond the 132-byte area named PRT2. (You'll have to convert 132 to hex and add it to hex 0178 to check the value shown above.)

DUMMY SECTIONS

The use of a *dummy section* is very much like using explicit addresses to control repetitive processing as discussed in topic 1. A dummy section gives the same control, but it lets you code labels instead of explicit addresses. This helps to prevent coding errors and makes it easier for a programmer to come back to a program later on and change it.

As an example, suppose you want to process a file of cards that has been copied to magnetic tape. The cards contain the monthly sales of your company's salesmen in this format:



Your program is to total the sales and print a report.

A normal program definition of an input area for this file might look like this:

```
SLSCARD   DS     0CL80
SLSNBR    DS     CL4
SLSNAME   DS     CL20
SLSREG    DS     CL2
SLSQTY    DS     CL6
          DS     CL48
```

In tape processing, however, the input records are usually blocked so let's assume the data from ten cards has been stored in each tape block. Your program must then process an 800-byte input block that contains ten 80-byte records.

The routine in figure 5-10 reads a tape block and then prints a line for each record in the block using the register operations presented in topic 1. After a line has been printed, using a subroutine named PRINTSUB, the sales quantity for each record is added to a field named TOTSUM. The fields of each 80-byte record are referred to by using explicit displacements from register 5, a substitute base register. For example,

```
        MVC   PRTREG,24(5)
```

means move the field addressed by base register 5 plus a displacement of 24 to the field named PRTREG. The processing loop is executed ten times using base register 6 as a counter for each input block.

If the card format is described as a dummy section, however, labels can replace the explicit operands as follows:

```
          USING CRDMAP,5
          .
          .
          .
CRDMAP    DSECT
SLSCARD   DS    0CL80
SLSNBR    DS    CL4
SLSNAME   DS    CL20
SLSREG    DS    CL2
SLSQTY    DS    CL6
          DS    CL48
PROGNAME  CSECT
```

DSECT is an assembler instruction and generates no machine code. It tells the assembler program that the storage definitions that follow form a dummy section having its own base register. Instead of displacements from the beginning of the program, labels within the DSECT are assigned displacements from the beginning of the DSECT. In addition,

```
          .
          .
          .
READTAPE  GET   TAPEFIL
          LA    5,IOAREA
          LA    6,10
NXTCRD    MVC   PRTNBR,0(5)
          MVC   PRTNAME,4(5)
          MVC   PRTREG,24(5)
          PACK  26(6,5),26(6,5)
          MVC   PRTQTY,PATTERN
          ED    PRTQTY,26(6,5)
          BAL   11,PRINTSUB
          AP    TOTSUM,26(6,5)
          LA    5,80(5)
          BCT   6,NXTCRD
          B     READTAPE
          .
          .
          .
IOAREA    DS    10CL80
          .
          .
          .
```

FIGURE 5-10 Use of Explicit Addressing to Deblock Records

a separate USING statement assigns a substitute base register for the dummy section by using the DSECT label as the first operand. Then, when a field within the DSECT is used as an operand, the substitute base register and the DSECT displacement make up the address that is assembled in the instruction.

The USING statement mentioned above tells the assembler program to use register 5 as the base register for any label that appears within the DSECT labeled CRDMAP.

```
            .
            .
            .
            USING *,3
            USING CRDMAP,5
            .
            .
            .
READTAPE    GET   TAPEFIL
            LA    5,IOAREA
            LA    6,10
NXTCRD      MVC   PRTNBR,SLSNBR
            MVC   PRTNAME,SLSNAME
            MVC   PRTREG,SLSREG
            PACK  SLSQTY,SLSQTY
            MVC   PRTQTY,PATTERN
            ED    PRTQTY,SLSQTY
            BAL   11,PRINTSUB
            AP    TOTSUM,SLSQTY
            LA    5,80(5)
            BCT   6,NXTCRD
            B     READTAPE
            .
            .
IOAREA      DS    10CL80
CRDMAP      DSECT
SLSCARD     DS    0CL80
SLSNBR      DS    CL4
SLSNAME     DS    CL20
SLSREG      DS    CL2
SLSQTY      DS    CL6
            DS    CL48
PROGNAME    CSECT
            .
            .
            .
```

FIGURE 5-11 Use of a Dummy Section to Deblock Records

Normally, this USING statement is put at the beginning of the program with any other USING statements there may be. It is then the programmer's responsibility to have the correct value in register 5 whenever a label within the DSECT is referred to.

The CSECT (Control Section) instruction at the end of the dummy section ends the dummy section and restores the program base register and location counter value. The label field of the CSECT instruction must be the program name taken from the label field of the START instruction. A DSECT should always end with either the DSECT of another dummy section or a CSECT statement.

The card processing routine of the blocking example can then be coded using DSECT labels as in figure 5-11. The machine code generated by this routine will be exactly the same as that generated by the program shown in figure 5-10. The label SLSNAME, for instance, will generate a displacement of 4 bytes beyond alternate base register 5 and will have a length attribute of 20 bytes.

Here's another DSECT example. An inventory master record on tape contains basic inventory information plus 25 warehouse location data groups:

BYTES	FIELD
1-10	Item Number
11-25	Item Description
26-30	Total Quantity on Hand
31-430	25 Storage Location Groups

Each 16-byte storage location group contains the warehouse number, bin number, quantity stored in the bin, and date it was stored as:

BYTES	FIELD
1-2	Warehouse Number
3-5	Bin Number
6-10	Quantity Stored
11-16	Date Stored (MMDDYY)

If a dummy section is used for the storage location data, each of the 25 groups can be processed using labels. As a result, I might code the definition of the master record as:

```
LOC                STATEMENT
0D30       MSTR      DS     0CL430
0D30       ITEMNBR   DS     CL10
0D3A       ITEMDESC  DS     CL15
0D4B       ONHAND    DS     PL5
0D50       STORAGE   DS     CL400
0000       LOCATION  DSECT
0000       WHSE      DS     CL2
0002       BIN       DS     CL3
0005       LOCQTY    DS     PL5
000A       STORDATE  DS     0CL6
000A       MONTH     DS     CL2
000C       DAY       DS     CL2
000E       YEAR      DS     CL2
0EE0       TAPEPROG  CSECT
0EE0       QTYWORK   DS     PL5
```

The location counter values that might be assembled are included to illustrate the effect of the DSECT. Notice that the actual storage area for the 25 sixteen-byte location groups is reserved by the DS statement labeled STORAGE. The dummy section itself reserves no area and the displacements (location counter values) for the fields within the DSECT are relative to the start of the DSECT. The CSECT statement restores the location counter value for the program so that QTYWORK begins at the first location beyond the 400 bytes of storage. (Incidentally, the location-quantity and on-hand fields are defined as packed-decimal fields, which is valid for tape fields.)

Figure 5-12 gives the input area definitions along with a processing routine that adds up all the warehouse location quantities and compares the sum to the total on-hand value. Register 8 is used as the base address for the dummy section, so the address of the 400 bytes named STORAGE is loaded into register 8 at the start of the routine. Register 9 is used as a counter, so the number of times the loop is to be executed (25) is loaded into it before the loop begins. Each time through the loop, then, register 8 is increased by 16 (the number of bytes used for each warehouse location) and the warehouse location quantity is added to QTYWORK. After

```
                .
                .
                .
                USING *,3
                USING LOCATION,8
                .
                .
                .
                LA    8,STORAGE
                LA    9,25
                ZAP   QTYWORK,=P'0'
ADDLOC          AP    QTYWORK,LOCQTY
                LA    8,16(8)
                BCT   9,ADDLOC
                CP    ONHAND,QTYWORK
                BNE   ERROR
                .
                .
                .
MSTR       DS     0CL430
ITEMNBR    DS     CL10
ITEMDESC   DS     CL15
ONHAND     DS     PL5
STORAGE    DS     CL400
LOCATION   DSECT
WHSE       DS     CL2
BIN        DS     CL3
LOCQTY     DS     PL5
STORDATE   DS     0CL6
MONTH      DS     CL2
DAY        DS     CL2
YEAR       DS     CL2
TAPEPROG   CSECT
QTYWORK    DS     PL5
                .
                .
                .
```

FIGURE 5-12 Use of a Dummy Section to Process 25 Storage Locations

the loop has been executed 25 times, the sum of the warehouse quantities is compared with the on-hand balance for the item. If they are not equal, a branch to an error routine takes place.

Terminology

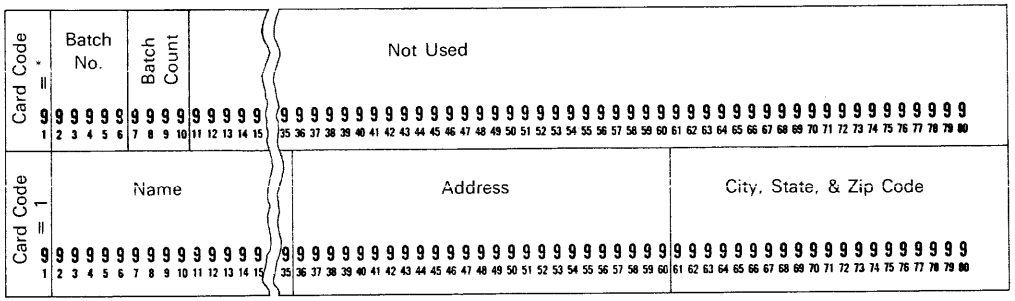
dummy section

Objective

Apply zero duplication factors, the ORG instruction, and dummy sections to appropriate aspects of programming problems.

Problems

1 A card-to-printer program is to read a control card at the start of the program and then list the name-and-address cards that follow. The format for the cards is:



Assuming a DTFCD named CARDFLE specifies one I/O area named CARDIO1 and no work area, use the ORG statement to define the input areas for the card types. Then, assuming the file has been opened, write a routine that reads the lead card, packs the batch number and count in fields named PBNO and PBCNT, reads the first name-and-address card, and moves its fields to print fields named PRTNAME, PRTADDR, and PRTCTY. If the control card isn't present, the routine should branch unconditionally to an address in register 10. If subsequent cards don't have a 1 in column 1, the program should branch to a subroutine named INVCODE.

2 In figure 5-4, explicit addresses are used to add seven daily-hours-worked fields. Use a dummy section to modify this routine so a label can be used for the hours-worked fields. (Assume that the program name is PR52).

Solutions

1 Figure 5-13 is an acceptable solution.
2 Figure 5-14 is an acceptable solution.

```
          .
          .
          .
          GET   CARDFLE
          CLI   BATCHCDE,C'*'
          BE    CONTINUE
          BR    10
CONTINUE  PACK  PBNO,BATCHNO
          PACK  PBCNT,BATCHCNT
READRTN   GET   CARDFLE
          CLI   NACODE,C'1'
          BNE   INVCODE
          MVC   PRTNAME,NANAME
          MVC   PRTADDR,NAADDR
          MVC   PRTCTY,NACTY
          .
          .
          .
CARDIO    DS    0CL80
BATCHCDE  DS    CL1
BATCHNO   DS    CL5
BATCHCNT  DS    CL4
          DS    CL70
          ORG   CARDIO
NACARD    DS    0CL80
NACODE    DS    CL1
NANAME    DS    CL34
NAADDR    DS    CL25
NACTY     DS    CL20
          ORG
          .
          .
          .
```

FIGURE 5-13 Solution 1—Using the ORG Statement

```
          .
          .
          .
          USING *,3
          USING HOURS,6
          .
          .
          .
          LA    4,7
          LA    6,HRSDATA
          ZAP   WKLYHRS,=P'0'
ADDLOOP   PACK  DLYHRS,DLYHRS
          AP    WKLYHRS,DLYHRS
          LA    6,3(6)
          BCT   4,ADDLOOP
          .
          .
          .
CARDIN    DS    0CL80
          DS    CL38
HRSDATA   DS    CL21
HOURS     DSECT
DLYHRS    DS    CL3
PRS2      CSECT
          DS    CL21
WKLYHRS   DS    PL3
          .
          .
          .
```

FIGURE 5-14 Solution 2—Using a Dummy Section

Part 3

Part 3

Advanced Assembler-Language Subjects

Once you have mastered the material in part 2, you can study any chapter in this part. You don't have to read these chapters in any particular sequence and you can skip any chapters that you aren't interested in. For instance, the business programmer may want to skip chapter 6, Binary Arithmetic, and go directly to the chapters on table handling and subprograms.

6

Binary Arithmetic

There are two forms of binary arithmetic instructions for the System/360–370: fixed-point and floating-point. Although both forms are faster than decimal arithmetic, decimal instructions are used for most mathematical routines that are written in assembler language. Why? Because binary arithmetic requires more data conversion than decimal arithmetic. Before fixed-point instructions can be used, the fields to be operated upon must be converted to fixed-point binary format. Before floating-point instructions can be used, the fields to be operated upon must be converted to floating-point format. After the operations have been performed, the results must be converted back to packed-decimal format prior to editing or to EBCDIC format. In a business program, then, this extra conversion counterbalances any advantage gained by the speed of binary arithmetic.

This chapter is divided into two topics. Topic 1 describes those fixed-point instructions that haven't been covered in

chapter 5, along with some related techniques. Topic 2 describes floating-point instructions. Because the material in both topics is rarely used by business programmers, the student of business programming may want to skip this chapter. This material is of primary interest to the computer scientist or software specialist.

TOPIC ONE Fixed-Point Arithmetic Topic 1 of chapter 5 presents most of the assembler-language elements related to fixed-point arithmetic. It shows how to define a halfword, fullword, or doubleword, how to convert a packed field to binary and vice versa, how to load registers, how to code fixed-point addition and subtraction instructions, how to store results, and how to compare fixed-point fields. These aspects of the fixed-point instruction set are useful in address manipulation and loop control as illustrated in chapter 5 and in table handling as illustrated in chapter 7.

MULTIPLY AND DIVIDE

To complete the fixed-point instruction set, you only need to learn about the multiply and divide instructions. You will then be able to code any arithmetic routine using the fixed-point instructions as well as the decimal instructions.

Multiply (M, MR, MH) The fixed-point multiply instruction has three forms: multiply-fullword (M), multiply-register (MR), and multiply-halfword (MH). In a fullword instruction (M or MR), the multiplicand must be placed in the odd register of a pair of even-odd registers—such as registers 6 and 7 or registers 10 and 11—before the instruction is executed. Although the contents of the even register are ignored, it is a common practice to clear it before the multiply instruction anyway. The even register of the even-odd pair is specified as the first operand and the resulting product is placed in the even-odd pair as a doubleword value:

```
SR    6,6           CLEAR REGISTER 6
L     7,MULTCAND    LOAD MULTIPLICAND
M     6,MLTPLYR     MULTIPLY
STM   6,7,DOUBLE    STORE PRODUCT
```

The sign of the product is developed according to the rules of algebra: like signs produce positive products, unlike signs produce negative products.

Generally, the range of values you deal with causes all significant bits in the product to be contained in the odd register of the even-odd pair. If this is the case and you know that the product will always be positive, you can ignore the even register and just store the contents of the odd register as the product:

```
M     6,MLTPLYR
ST    7,PRODUCT
```

If you're not sure that the product will be positive but know that the value of the product will be completely contained in one register, you can transfer the sign from the even to the odd register by using:

```
M     6,MLTPLYR
OR    7,6
ST    7,PRODUCT
```

The OR instruction will affect only the leftmost bit in register 7. If the sign bit in register 6 is on, it will turn on the sign bit in register 7. If the sign bit is off, register 7 will be unchanged. As a result, the odd register will contain the product as a one-word binary value, complete with correct sign. (The OR instruction is covered in detail in chapter 8.)

In the multiply-halfword (MH) instruction, only one register is required and any register, odd or even, can be used. The multiplicand must be in the rightmost half of the register and the multiplier must be on a halfword boundary. The product is a fullword value that replaces the contents of the register specified:

```
LH    9,FACTOR1
MH    9,FACTOR2
ST    9,FWORD
```

In either the halfword or the fullword multiply instructions, no overflow is possible (it is possible in decimal instructions).

Divide (D, DR) The fixed-point divide instruction also requires an even-odd pair of registers as operand-1. The

dividend must occupy the pair as a doubleword value and the divisor must be a fullword or a register:

```
D       6,FWORD1
DR      4,9
```

The resulting quotient is stored as a fullword value in the even register with its sign determined by the normal algebraic rules. The remainder is stored in the odd register, also a fullword value, with its sign the same as the dividend's was:

```
SR      8,8
L       9,DIVIDEND
D       8,DIVISOR
ST      9,QUOTNT
ST      8,REMNDER
```

If the values of the dividend and divisor are such that the quotient won't fit in the even register, a fixed-point divide exception occurs and the program is usually cancelled.

NEGATIVE RESULTS

Because arithmetic routines sometimes lead to negative results, it is occasionally (but rarely) necessary during debugging to determine the value of a negative fixed-point field or register. In the System/360-370, negative binary numbers are indicated by a 1 in the leftmost bit of the word. The rest of the word contains the binary number but in *two's complement form*. To get the two's complement of a binary number, reverse the bits and add 1 to the rightmost bit as in this example:

```
NUMBER:                0000000001011101
REVERSE THE BITS:      1111111110100010
ADD 1:                                1
TWO'S COMPLEMENT:      1111111110100011
```

Here, the two's complement of decimal 93 is derived. Thus, negative 93 stored in a halfword is this series of bits:

```
1111 1111 1010 0011
```

When debugging, you may want to find the decimal value of a fixed-point field or register. If the leftmost bit is positive (indicated by a hex value of seven or less in the leftmost half-byte of the word), you can convert to decimal as described in chapter 2 by using a hex chart like the one in figure 2-5. On those rare occasions when the value is negative, you can determine the decimal value by following these steps.

Fullword Hex Notation:	FFE4
1. Convert to Binary:	1111 1111 1110 0100
	-1
2. Subtract 1:	1111 1111 1110 0011
3. Reverse Bits:	0000 0000 0001 1100
4. Convert to Decimal:	-28

FIGURE 6-1 Converting a Negative Hex Value to Decimal

1 Convert the hex digits to a binary configuration.
2 Subtract binary 1 from the number, which is in two's complement form.
3 Reverse the bits to get the binary number.
4 Convert the binary number directly to decimal or convert to hex and then to decimal using conversion charts.

This procedure is illustrated in figure 6-1.

CONCLUSION

One question you might ask is: When should the fixed-point instructions be used for an arithmetic routine? The general answer is: only when a long series of arithmetic operations is to be performed on a small number of fields—the type of repetitive processing that you wouldn't ordinarily find in a business program. If a field is involved in only five or six operations, as is the case in most business routines, you should use decimal arithmetic because it requires less data conversion.

Furthermore, an arithmetic routine involving a long series of arithmetic operations on a small number of fields isn't likely to be written in assembler language. For this type

```
AVGPROB  START 0
BEGIN    BALR  3,0
         USING *,3
         OPEN  CARD,REPORT
         SR    4,4
         SR    5,5
         SR    6,6
         SR    7,7
         SR    8,8
NXTCRD   GET   CARD
         PACK  DWORK,CHGT
         CVB   9,DWORK
         PACK  DWORK,CWGT
         CVB   10,DWORK
         AR    5,9
         AR    7,10
         A     8,BCON1
         B     NXTCRD
CRDEOF   MVI   PRTIO,X'40'
         MVC   PRTIO+1(131),PRTIO
         DR    4,8
         CVD   5,DWORK
         ED    RAVGHGT,DWORK+6
         DR    6,8
         CVD   7,DWORK
         ED    RAVGWGT,DWORK+6
         MVC   PRTIO(32),RHGTLINE
         PUT   REPORT
         MVC   PRTIO(32),RWGTLINE
         PUT   REPORT
         CLOSE CARD,REPORT
         EOJ
CARD     DTFCD DEVADDR=SYSIPT,IOAREA1=CRDIO,EOFADDR=CRDEOF
REPORT   DTFPR DEVADDR=SYSLST,IOAREA1=PRTIO,BLKSIZE=132
PRTIO    DS    CL132
CRDIO    DS    0CL80
CSNAME   DS    CL20
CHGT     DS    CL2
CWGT     DS    CL3
         DS    CL55
RHGTLINE DC    C'THE AVERAGE HEIGHT IS'
RAVGHGT  DC    X'40202021'
         DC    C' INCHES'
RWGTLINE DC    C'THE AVERAGE WEIGHT IS'
RAVGWGT  DC    X'40202021'
         DC    C'POUNDS'
DWORK    DS    D
BCON1    DC    F'1'
         END   BEGIN
```

FIGURE 6-2 Fixed-Point Arithmetic

of routine, a language such as FORTRAN or BASIC is much easier to work with. As a result, the assembler-language programmer rarely needs to use the fixed-point instructions for arithmetic routines although the object code resulting from FORTRAN or BASIC programs will often involve fixed-point arithmetic routines. Remember, however, that fixed-point addition and subtraction are commonly used for address manipulation, loop control, and table handling.

Terminology

two's complement form

Objective

Use the fixed-point instructions in appropriate arithmetic routines.

Problem

Use fixed-point arithmetic in a program to calculate the average height and weight of a group of male students. The height to the nearest inch and weight to the nearest pound are punched in cards with this format:

CARD COLUMNS	FIELD
1-20	Student Name
21-22	Height
23-24	Weight

The averages are to be printed in two final total lines with these formats:

```
THE AVERAGE HEIGHT IS XXX INCHES
THE AVERAGE WEIGHT IS XXX POUNDS
```

Your program should read the EBCDIC values, convert them to fixed-point, sum the height and weight totals, and, when the end-of-file is reached, calculate the averages.

Solution

Figure 6-2 is an acceptable solution.

TOPIC TWO Floating-Point Arithmetic

Floating-point is the most powerful form of System/360-370 arithmetic for several reasons. First, it's faster than fixed-point or decimal arithmetic. Second, the floating-point data format can store a wider range of values than the other forms. Third, the floating-point format can store fractional values and, when the floating-point instructions are executed, they automatically align the decimal positions.

In business data processing, however, there is little need for this power. In fact, due to the problem of converting data to and from floating-point format, floating-point instructions are never used for business arithmetic. Instead, the floating-point facilities are used for solving scientific problems—ones in which very large or very small values may be involved or ones in which decimal alignment may be extremely difficult to keep track of when using fixed-point arithmetic.

The vast majority of scientific programming is not done in assembler language, however. For a scientific programming problem, a mathematical language such as FORTRAN is much easier to use. As a result, unless you write a compiler some day, it is very unlikely that you will ever actually use floating-point instructions in a production program. For this reason, the material in this topic should be studied with one major objective: to develop a better understanding of high-level languages that use floating-point instructions in the resulting object code. After you complete this topic, for example, you should have a better understanding of how precision errors can occur in FORTRAN.

DATA FORMATS

Floating-point data is stored in a fullword (*short form*) or a doubleword (*long form*) as illustrated in figure 6-3. In either form, the leftmost bit represents the sign of the field: "off" represents plus, and "on," minus. The next seven bits represent the *exponent* of the number and the remaining bits (bits 8-31 in the short form, 8-63 in the long form) represent the *fraction*. This is something like the exponent notation that you may have used in chemistry or physics in which a value like 1,563,487 can be written 0.1563487E+7 meaning

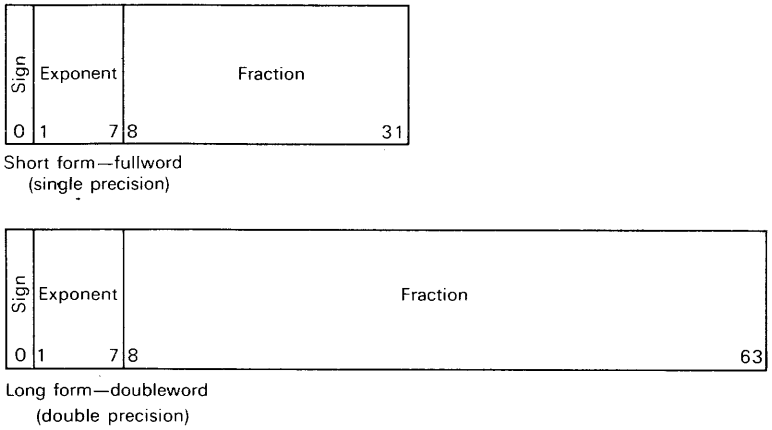


FIGURE 6-3 Floating-Point Data Formats

0.1563487 × 10^{7}. The exponent—in this case, +7—is sometimes called the *characteristic*; the fraction—in this case, 0.1563487—is sometimes called the *mantissa*.

The exponent in floating-point format is expressed as a power of 16 (instead of 10). This exponent value is stored as a binary number in bits 1-7. To allow both plus and minus exponents, the exponent is stored in *excess-64 format*. This means that the value stored as the exponent is 64 more than the actual exponent. If, for example, the value stored is 65, it really means 65 minus 64, or 1. Thus, the fraction is multiplied by 16 to the first power. If the exponent value is less than 64, it indicates a negative exponent—for instance, a value of 62 means 62 minus 64, or an exponent of −2. (Negative exponents allow storage of very small fractional values as in 0.18 × 10^{-12} which equals .00000000000018.) Because the seven exponent bits can represent a number from 0 through 127, the exponent can range from −64 to +63. Thus, the range of a floating-point number in either form is approximately from 10^{-78} to 10^{+75}.

The fraction portion of a floating-point number assumes a decimal point before bit number 8. Thus, the place values of bits 8 through 31 in the short form, or bits 8 through 63 in the long form, are 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, and so on. The fractional values of these place values are 1/2, 1/4, 1/8, 1/16, and so on. Figure 6-4 illustrates some short-form floating-point numbers in binary along with their decimal equivalents. Unlike fixed-point format, negative fractions are not in complement form.

The difference between short and long form is the number of digits that can be carried in the fraction. This is referred to as the *precision* of a number. In short form, which is often called *single precision*, the equivalent of about 7 decimal digits can be stored in the fraction; in long form, which is often called *double precision*, the equivalent of about 16 decimal digits can be stored.

A floating-point value is *normalized* if the fraction bits have been shifted left as much as possible so that the exponent is at the minimum possible value. In contrast, the last example in figure 6-4 is *unnormalized*. In this case, the fraction can be shifted left four bits and the exponent reduced by one without changing the value stored. Since normalization allows a maximum number of fraction bits to be carried, thereby offering the highest level of precision, floating-point fields are usually normalized. Although there are eight instructions for manipulating unnormalized fields and there are cases when these instructions have value, they are rarely used and, therefore, aren't covered in this book.

To define a floating-point field, you use the type codes E and D. Type code E indicates short form and type D indicates long form as in these examples:

```
SINGLE   DS    E
DOUBLE   DS    D
FPVAL1   DC    E'3.141596'
FPVAL2   DC    D'1.86E+5'
FPVAL3   DC    E'-1.0E-6'
```

The nominal value in a floating-point operand can be written in regular decimal form or in exponent notation. If exponent notation is used, the exponent in decimal must be preceded by E and a plus or minus sign. The E type code reserves a

Sign	Exponent	Fraction	Determining value
0	1000000	10100000 00000000 00000000	Exponent=(64−64)=0→$16^0=2^0$ Fraction=$(2^{-1}+2^{-3})$ Value=$2^0(2^{-1}+2^{-3})=(2^{-1}+2^{-3})=(.500+.125)=0.625$
1	1000001	01100000 00000000 00000000	Exponent=(65−64)=1→$16^1=2^4$ Fraction=$(2^{-2}+2^{-3})$ Value=$-2^4(2^{-2}+2^{-3})=-(2^2+2^1)=-(4+1)=-5$
0	0111110	01000000 00000000 00000000	Exponent=(62−64)=$-2\rightarrow16^{-2}=2^{-8}$ Fraction=(2^{-2}) Value=$2^{-8}(2^{-2})=(2^{-10})=\frac{1}{1024}=.009765625$
0	1000100	00001001 00000000 01000000	Exponent=(68−64)=4→$16^4=2^{16}$ Fraction=$(2^{-5}+2^{-8}+2^{-18})$ Value=$2^{+16}(2^{-5}+2^{-8}+2^{-18})=(2^{11}+2^8+2^{-2})=(2048+256+0.25)=2304.25$

FIGURE 6-4 Floating-Point Numbers and Their Equivalents

fullword of storage, the D form reserves a doubleword, both with proper boundary alignment.

Figure 6-5 illustrates some additional constant definitions with the hex constant shown as it will appear both on an assembly listing and in a storage dump. Notice that hex 40 in the exponent portion of the field is equivalent to a zero exponent (64 minus 64). Note also that a value like 0.1 can't be expressed exactly in floating-point. In the single precision format, for example, the hex representation is 4019999A where 40 is the exponent and 19999A is the fraction. This is an approximate value of .09999999.

INSTRUCTIONS

As in fixed-point arithmetic, all arithmetic operations for floating-point are performed in registers. For this purpose, the floating-point feature (optional on some models of System/360 but standard on others, and standard on all System/370s) provides four *floating-point registers*. These are doubleword registers (distinct from the general-purpose registers) that are used only for floating-point operations. The registers are numbered 0, 2, 4, and 6, and accommodate both short- and long-form data. For short-form instructions, only the first of the two words in each register is used, while the long-form instructions use both words. There are 44 floating-point instructions on the System/360; they include loading, storing, addition, subtraction, multiplication, division, and comparison. Thirty-two of these instructions (summarized in figure 6-6) are presented in this topic.

Store There are two forms of the floating-point store instruction:

```
STE   2,FPSAVE    Store Short
STD   4,FPDBLE    Store Long
```

In these instructions, as in all floating-point instructions, the character E in the operation code means short form and the character D means long form. The operands of all the

Hex Code on Assembly Listing	Source Code		Object Code in Storage Dump
+00.000000	DC	E'0'	00000000
+41.100000	DC	E'1'	41100000
+43.FFF000	DC	E'4095'	43FFF000
+44.100000	DC	E'4096'	44100000
-41.100000	DC	E'-1'	C1100000
-43.FFF000	DC	E'-4095'	C3FFF000
-44.100000	DC	E'-4096'	C4100000
+40.800000	DC	E'0.5'	40800000
+41.180000	DC	E'1.5'	41180000
+40.19999A	DC	E'0.1'	40199999A
+3F.28F5C3	DC	E'0.01'	3F28F5C3
+3E.418937	DC	E'0.001'	3E418937
+48.4C2CBC	DC	E'12.78E+8'	484C2CBC
+51.56BC76	DC	E'1E+20'	5156BC76
-6A.BF9572	DC	E'-2.8E+50'	EABF9572
+00.00000000000000	DC	D'0'	0000000000000000
+41.10000000000000	DC	D'1'	4110000000000000
+40.80000000000000	DC	D'0.5'	4080000000000000
+40.19999999999999A	DC	D'0.1'	401999999999999A
+17.3BD0F495A9703E	DC	D'0.1E-49'	173BD0F495A9703E
+40.1F9ADD3739635F	DC	D'12345.6789E-5'	401F9ADD3739635F

FIGURE 6-5 Floating-Point Representation

floating-point instructions also have identical requirements: the first operand must always be one of the floating-point register numbers; the second operand must be either a fullword (short form) or a doubleword (long form) at a proper boundary, or another floating-point register.

Load In addition to the basic forms of the load instruction:

```
LE    4,FWORD    Load Short
LD    6,DWORD    Load Long
```

there are register-to-register load instructions:

```
LER   0,4    Load Short RR
LDR   2,0    Load Long RR
```

Several special forms of the load instruction, as indicated in figure 6-6, load a value while forcing the sign of the value to be positive, negative, or the opposite of its original value. Because negative floating-point numbers aren't stored in complement form, the load complement instructions change the sign of a field by reversing only the sign bit.

Add and Subtract The add and subtract instructions follow a similar pattern. Both short and long forms are available in storage-to-register and register-to-register format:

```
AE      0,FACT1      Add Short
AER     0,6          Add Short RR
AD      4,DFACT      Add Long
ADR     4,6          Add Long RR
SE      2,SUBWORD    Subtract Short
SER     4,2          Subtract Short RR
SD      6,DBLESUB    Subtract Long
SDR     6,0          Subtract Long RR
```

When these instructions are executed, the value in the second operand is added to or subtracted from the first operand with the result replacing the first operand. Any alignment of decimal points is performed automatically, and the result is normalized before it is placed in the first operand register.

Multiply The floating-point multiply instruction can have either a storage field or another floating-point register as the second operand:

```
ME      0,FACT       Multiply Short
MER     2,4          Multiply Short RR
MD      4,DFACT      Multiply Long
MDR     2,6          Multiply Long RR
```

When the instruction is executed, the first operand is multiplied by the second with the product replacing the first operand. Like signs result in a positive result, unlike signs in a negative result. An *exponent overflow* exception occurs if the resulting exponent value is greater than 127 and *exponent underflow* occurs if it's less than 0.

Divide Floating-point divide is very similar:

```
DE      2,DIV        Divide Short
DER     2,0          Divide Short RR
DD      4,DDIVS      Divide Long
DDR     2,6          Divide Long RR
```

When executed, the quotient replaces the dividend in the first operand register and no remainder is saved. Exponent overflow and underflow can occur as in multiply, and an attempt to divide by zero leads to a floating-point divide exception.

Operation Code	Instruction Name
STE	Store Short
STD	Store Long
LE	Load Short
LD	Load Long
LER	Load Short RR
LDR	Load Long RR
LPER	Load Positive Short
LPDR	Load Positive Long
LNER	Load Negative Short
LNDR	Load Negative Long
LCER	Load Complement Short
LCDR	Load Complement Long
AE	Add Short
AD	Add Long
AER	Add Short RR
ADR	Add Long RR
SE	Subtract Short
SD	Subtract Long
SER	Subtract Short RR
SDR	Subtract Long RR
ME	Multiply Short
MD	Multiply Long
MER	Multiply Short RR
MDR	Multiply Long RR
DE	Divide Short
DD	Divide Long
DER	Divide Short RR
DDR	Divide Long RR
CE	Compare Short
CD	Compare Long
CER	Compare Short RR
CDR	Compare Long RR

FIGURE 6-6 Floating-Point Instructions

Compare The floating-point compare instructions provide an arithmetic comparison of floating-point values:

```
CE     2,VAL1     Compare Short
CER    0,2        Compare Short RR
CD     6,VAL2     Compare Long
CDR    2,4        Compare Long RR
```

After the compare instruction, you would use an ordinary branch instruction to test the condition code and branch accordingly.

One thing you should understand about floating-point arithmetic is that it doesn't always give exact results. For one thing, some decimal values (like 0.1) can't be expressed exactly in hexadecimal floating-point format. For another thing, hex digits on the right of a fraction may be lost during an arithmetic operation.

To illustrate, suppose the add-long instruction adds two values, one with a hex exponent of 40, another with a hex exponent of 43. Prior to addition, the fraction with the smaller exponent is shifted right three hex digits to align the fractions. This means the rightmost 12 bits of the smaller number are moved out of the register. After addition, the result is normalized (if necessary), so the result is shifted left—which can result in lost hex digits.

To reduce the number of lost digits in floating-point arithmetic operations, the electronic circuitry that actually does the shifting has four extra bit positions to the right of the fraction that is shifted. These four bit positions are called a *guard digit*. Then, if a fraction is shifted right three hex digits, only two of the digits are lost when the result is normalized. Nevertheless, the result of a floating-point operation is likely to be less precise than a fixed-point or decimal operation.

This imprecision is an important notion to grasp because it can affect the coding of branching operations. To illustrate, suppose a program compares a floating-point constant of 0.1 with a field in which a decreasing value eventually should reach 0.1. When they are equal, the program should branch. Since the constant 0.1 can't be represented exactly and since the manipulated value may lose digits as it approaches 0.1, the two fields may never be exactly equal. Because of this, a branch code of BNL or BNH should be used so the program will branch when the manipulated value equals or passes the constant value. If BE is used, the program may never branch.

DATA CONVERSION

The main reason floating-point isn't used in business programming is that data conversion is a problem. Since there is no instruction that converts other data formats to floating-point format, you must code a conversion routine similar to the one in figure 6-7. This routine converts an EBCDIC input field to floating point. This routine is relatively simple because the input format is rigid: the leftmost column of the ten-column field contains the sign, followed by a four-column integer, followed by a decimal point, followed by a four-column decimal fraction. Imagine how much more complex a FORTRAN conversion routine must be since the decimal point can be anywhere in a field and E notation is optional.

In figure 6-7, the sign position is tested first. If it is minus, a negative zone is moved to both the integer and fraction portions of the field. After this, the integer and fraction portions are converted to floating point separately and then added together.

These instructions are used to convert the integer portion of the field:

```
CONV     PACK   DWORK,CINT
         CVB    7,DWORK
         M      6,BCON1
         O      6,EXPONENT
         STM    6,7,DWORK
         LD     0,DWORK
         AD     0,FPZERO
```

After the digits are converted to binary in register 7, the sign of the binary value is moved to the leftmost bit of register 6 by multiplying by binary one (BCON1), thus making a doubleword binary value. A valid floating-point exponent value is then placed into register 6 using the O (OR) instruction (described in detail in chapter 8). In this case, an exponent of hex 4E (the equivalent of +14) is placed in the register without changing any of the other bits. Since there are 14 hex digits in the fraction portion of a double precision

field, this means the decimal point is assumed to be to the right of all the hex digits in the fraction. In other words, the value stored in registers 6 and 7 is a valid, unnormalized whole number. This value is then loaded into floating-point register 0 and a value of zero is added to it. After the addition takes place, the result is automatically normalized so the original integer value is shifted left as many hex digits as appropriate and the exponent is reduced accordingly.

To convert the fraction portion, these instructions are used:

```
PACK   DWORK,CFRAC
CVB    7,DWORK
M      6,BCON1
O      6,EXPONENT
STM    6,7,DWORK
LD     2,DWORK
DD     2,FRACADJ
```

This conversion takes place in the same manner as the integer conversion until after the decimal fraction is loaded as an integer into floating-point register 2. Then, the value is divided by 10,000 so it takes on its true fractional value. After the integer and fractional portions of the input field have been added together, the floating-point value is stored in the field named FPVAL.

Terminology

short form	single precision
long form	double precision
exponent	normalized
fraction	unnormalized
characteristic	floating-point register
mantissa	exponent overflow
excess-64 format	exponent underflow
precision	guard digit

Objective

Give two reasons why a numeric result obtained using floating-point instructions may not be exactly equal to a result obtained using fixed-point instructions.

```
         .
         .
         .
         CLI   CSIGN,C'-'
         BNE   CONV
         MVZ   CINT+3(1),=X'D0'
         MVZ   CFRAC+3(1),=X'D0'
CONV     PACK  DWORK,CINT
         CVB   7,DWORK
         M     6,BCON1
         O     6,EXPONENT
         STM   6,7,DWORK
         LD    0,DWORK
         AD    0,FPZERO
         PACK  DWORK,CFRAC
         CVB   7,DWORK
         M     6,BCON1
         O     6,EXPONENT
         STM   6,7,DWORK
         LD    2,DWORK
         DD    2,FRACADJ
         ADR   0,2
         STD   0,FPVAL
         .
         .
         .
DWORK    DS    D
FPVAL    DS    D
FPZERO   DC    D'0'
FRACADJ  DC    D'1.0E+4'
EXPONENT DC    X'4E000000'
BCON1    DC    F'1'
         .
         .
         .
CRDVAL   DS    0CL10    (SXXXX.XXXX)
CSIGN    DS    C
CINT     DS    CL4
CPOINT   DS    C
CFRAC    DS    CL4
```

FIGURE 6-7 Floating-Point Conversion Routine

7

Table Handling

Tables are used in many data processing applications. For example, a tax table may be used to look up the amount of income tax to be withheld from paychecks. To find the premium to be charged for an insurance policy, rating tables are often used. And in many statistical analyses, tables are printed to show how data breaks down into categories.

This chapter is divided into two topics. The first topic describes the coding for simple tables, called single-level tables. The second topic describes the coding and techniques for more advanced tables, called multi-level tables.

TOPIC ONE Single-Level Tables A *single-level table* is a table that tabulates data for one variable factor. For instance, the rate table in figure 7-1 is a single-level table in which the pay class is the variable. As the pay class increases, so does the rate of pay.

To illustrate the coding for a single-level table, suppose you wanted to store the rate table in figure 7-1 and use it to look up employees' pay rates. The following is one way the table can be defined in assembler language:

```
PAYTABLE DS    0CL40
         DC    C'01'
         DC    PL2'221'
         DC    C'02'
         DC    PL2'239'
         DC    C'03'
         DC    PL2'258'
         DC    C'04'
         DC    PL2'277'
         DC    C'05'
         DC    PL2'297'
         DC    C'06'
         DC    PL2'317'
         DC    C'07'
         DC    PL2'338'
         DC    C'08'
         DC    PL2'360'
         DC    C'09'
         DC    PL2'381'
         DC    C'10'
         DC    PL2'403'
```

This table consists of ten entries with each entry consisting of one pay class and one pay rate. Each pay class is two-bytes long in EBCDIC form; each pay rate is two-bytes long in packed-decimal form.

Figure 7-2 illustrates another way of defining the same table. This time each table entry is defined in hex:

```
PAYTABLE DS    0CL40
         DC    X'F0F1221C'
         DC    X'F0F2239C'
         .
         .
         .
         DC    X'F1F0403C'
         DC    X'FFFFFFFF'
```

Pay Class	Pay Rate
1	2.21
2	2.39
3	2.58
4	2.77
5	2.97
6	3.17
7	3.38
8	3.60
9	3.81
10	4.03

FIGURE 7-1 A Single-Level Table

The resulting object code is identical to that of the previous table definition with the exception of one additional table entry—an entry containing hex FFFFFFFF. This entry will be used to indicate the end of the table in storage.

A single-level table like this is normally searched sequentially. Starting at the first entry, the input pay class is compared with the pay-class value in the table. If the pay classes match, the corresponding pay rate is used to calculate the worker's pay. If the pay-class values don't match, the input pay class is compared with the next entry in the table. If the end of the table is reached without finding a match, it is assumed that the input pay class is invalid.

This sequential search technique is illustrated by the table-lookup routine in figure 7-2. This routine uses register 8 as a substitute base register so that the search can be done in a loop. First, register 8 is loaded with the address of the table by using the LA instruction. Then, the CLC instruction compares the two-byte pay-class field in the input card, CRDPAYCL, to the first two bytes of the first table entry. If they are equal, the BE instruction transfers control to the instruction named PAYFOUND. PAYFOUND is the first instruction of the routine that calculates the worker's pay. Since the pay rate to be used is two bytes beyond the

address in register 8, the following instruction is the first one in the PAYFOUND routine:

```
PAYFOUND ZAP   PAYWORK,2(2,8)
```

It places the packed-decimal pay rate from the table (the two-byte field addressed by register 8 plus a displacement of 2) into the field named PAYWORK.

If CRDPAYCL and the table-entry pay class don't match, control falls through the branch to the next instruction. Then, the CLI instruction checks to see if the end of the table has been reached. If the table entry is hex FF, indicating the end of the table, the BE instruction following the CLI will branch to an error routine named NOTFOUND. If neither a match nor end-of-table is found, the program must go through the loop again to examine the next table entry. When this happens, the LA instruction increases register 8 by four so that it contains the address of the next table entry, and the program branches to the beginning of the compare loop, CMPCLASS.

This type of table definition and lookup technique, sometimes called *factor matching*, will work for a great variety of tables. For one thing, the variable factors don't have to form a continuous sequence. If, for example, some old pay classes are deleted and some new ones added, the table might look like this after a year or two:

Pay Class	Pay Rate
1	2.25
2	2.43
4	2.85
5	3.08
6	3.27
7	3.46
9	3.88
10	4.09
11	4.23
12	4.47

```
          .
          .
          .
          LA    8,PAYTABLE
CMPCLASS  CLC   CRDPAYCL,0(8)
          BE    PAYFOUND
          CLI   0(8),X'FF'
          BE    NOTFOUND
          LA    8,4(8)
          B     CMPCLASS
          .
          .
          .
PAYFOUND  ZAP   PAYWORK,2(2,8)
          .
          .
          .
PAYTABLE  DS    0CL40
          DC    X'F0F1221C'
          DC    X'F0F2239C'
          DC    X'F0F3258C'
          DC    X'F0F4277C'
          DC    X'F0F5297C'
          DC    X'F0F6317C'
          DC    X'F0F7338C'
          DC    X'F0F8360C'
          DC    X'F0F9381C'
          DC    X'F1F0403C'
          DC    X'FFFFFFFF'
PAYWORK   DS    PL2
          .
          .
          .
```

FIGURE 7-2 Table-Lookup Routine—Factor Matching

However, the table-lookup coding in figure 7-2 will still work provided the table definitions have been changed to

```
          .
          .
          .
          PACK  PAYCLASS,CRDPAYCL
          CP    PAYCLASS,LOWLIMIT
          BL    CLASSERR
          CP    PAYCLASS,HILIMIT
          BH    CLASSERR
          LA    12,PAYTABLE
          ZAP   TABCOUNT,LOWLIMIT
COMPLOOP  CP    TABCOUNT,PAYCLASS
          BE    RATEFND
          AP    TABCOUNT,LOWLIMIT
          LA    12,2(12)
          B     COMPLOOP
RATEFND   ZAP   WRATE,0(2,12)
          .
          .
          .
TABCOUNT  DS    PL2
HILIMIT   DC    P'10'
LOWLIMIT  DC    P'1'
PAYCLASS  DS    PL2
WRATE     DS    PL2
PAYTABLE  DS    0CL20
          DC    PL2'221'
          DC    PL2'239'
          DC    PL2'258'
          DC    PL2'277'
          DC    PL2'297'
          DC    PL2'317'
          DC    PL2'338'
          DC    PL2'360'
          DC    PL2'381'
          DC    PL2'403'
```

FIGURE 7-3 Positional-Table-Lookup Routine—Counting Entries

correspond with the changes made in the table.

Also, in such a case, it doesn't matter what order the table entries are in. I could have defined the table with the entries in reverse order or completely out of order. The proper pay rate will still be found by matching with the pay class.

The sequence of entries can, however, affect how efficient the table lookup is in terms of processing time. Since the table is always searched starting with the first entry and proceeding towards the last, the most-used entry should be the first one in the table and the least-used entry should be the last. If, for example, 65 of the 100 jobs in a factory are assigned pay class 4, why waste all the time it takes to compare against classes 1, 2, and 3 before getting to class 4? Instead, class 4 should be the first entry in the table. Similarly, the rest of the table should be sequenced according to frequency of use. You can imagine what a difference this type of sequencing could make if class 10 were the one used most often.

POSITIONAL-TABLE LOOKUP

The variable factor in the table in figure 7-1 is pay class. Since this factor forms an unbroken sequence, you can also look up pay rates by their position in the table. When using this type of table, called a *positional table*, you need only include the pay rates, not the pay classes:

```
PAYTABLE DS    0CL20
         DC    PL2'221'
         DC    PL2'239'
         DC    PL2'258'
         DC    PL2'277'
         DC    PL2'297'
         DC    PL2'317'
         DC    PL2'338'
         DC    PL2'360'
         DC    PL2'381'
         DC    PL2'403'
```

To look up pay rate in this table, you must use the input pay class as some sort of index. One way to do this is to start at the beginning of the table and count through the entries until the count is equal to the input pay class. This technique is illustrated in figure 7-3.

The first five instructions of this routine check to see that the input pay class is a value in the range of one through ten. If the value is less than one or more than ten, the program branches to a routine named CLASSERR and the lookup is not performed. If the pay-class field is a valid number, the address of the table is loaded into register 12 and a value of one is placed in the count field. The program then enters the table-lookup loop.

In the lookup loop, the counter field, TABCOUNT, is compared with the packed input pay-class field, PAYCLASS. If they are equal, register 12 points to the proper pay rate and the program branches to RATEFND. If they're not equal, TABCOUNT is increased by one, register 12 is increased by two so it points to the next pay-rate entry, and the program repeats the loop. Since the program has already checked to see that the input pay class is in the proper range, there should always be a match. In the RATEFND instruction (the first instruction of the pay-calculation routine) the table value addressed by register 12 is placed in the field named WRATE.

Figure 7-4 illustrates a second way to use the input pay class as an index to the positional pay-rate table. Here, the idea is to convert the input pay class to a displacement value. By adding this displacement value to the starting address of the table, the proper pay rate can be addressed directly.

The first four instructions of this routine perform the high/low check of the input pay-class value. First, the pay class is packed into a doubleword work area so it will be ready to convert to binary. If the input pay class is within the proper range (from one through ten) the lookup continues. If not, the routine branches to CLASSERR and the lookup is not performed.

```
          .
          .
          .
          PACK  DBLEWORK,CRDPAYCL
          CP    DBLEWORK,LOWLIMIT
          BL    CLASSERR
          CP    DBLEWORK,HILIMIT
          BH    CLASSERR
          CVB   7,DBLEWORK
          S     7,=F'1'
          M     6,BCON2
          LA    8,PAYTABLE
          AR    7,8
          ZAP   WRATE,0(2,7)
          .
          .
          .
BCON2     DC    F'2'
DBLEWORK  DS    D
HILIMIT   DC    P'10'
LOWLIMIT  DC    P'1'
PAYTABLE  DS    0CL20
          DC    PL2'221'
          DC    PL2'239'
          DC    PL2'258'
          DC    PL2'277'
          DC    PL2'297'
          DC    PL2'317'
          DC    PL2'338'
          DC    PL2'360'
          DC    PL2'381'
          DC    PL2'403'
WRATE     DS    PL2
          .
          .
          .
```

FIGURE 7-4 Positional-Table-Lookup Routine—Direct Addressing

The next five instructions determine the address of the proper pay rate:

```
CVB     7,DBLEWORD
S       7,=F'1'
M       6,BCON2
LA      8,PAYTABLE
AR      7,8
```

First, the input pay class is converted to binary in register 7. Second, the value in register 7 is reduced by one. Third, the value in register 7 is multiplied by two. (Since the multiply instruction, described in chapter 6, requires an even-odd pair of registers as the first operand, the first operand above is registers 6 and 7. When the multiply instruction is executed, the product is placed in both registers. Since in this case the product will be between zero and 18, all significant bits in the answer are in register 7.) Fourth, the address of the pay table is loaded into register 8. Finally, the address in register 8 is added to the displacement in register 7 so that register 7 contains the address of the appropriate table value. The next instruction

```
ZAP     WRATE,0(2,7)
```

stores this rate in a work field named WRATE and the pay calculation begins.

Note that if a sequence starts at a value other than one, this direct-addressing technique can still be used. In such a case, you would subtract an appropriate value from the variable factor at the start of the routine. Similarly, the multiplication factor is determined by the length of the table entries. If, for example, the pay classes start at 11 and are four bytes long, 11 would be subtracted from pay class at the start of the routine and the multiplication factor would be four.

The direct-addressing technique, like the counting technique, works best if the variable factors form an unbroken sequence. If there are only a few "holes" in the sequence as in 1, 2, 3, 5, 6, 7, 9, 10, 11, and 12, you can fill them with dummy entries. However, if there are many holes in the table, the technique becomes inefficient.

In general, positional-lookup techniques are not as flexible as the factor-matching techniques. I recommend that you use a positional table only when you can use the direct-addressing technique. Otherwise, use factor matching with the table arranged according to frequency of use.

LOADING A TABLE

In many cases, particularly if a table needs frequent changes, the table used in a program isn't defined in the program. Instead, the table is read into storage from a file (cards, tape, or disk) at the beginning of the program. Then the table can be changed without changing the program that uses it.

Using the pay-table example again, you might find the table values maintained in a small card deck. The file could have several table entries on each card or only one entry per card. Let me use the second case, one entry per card, to illustrate the code needed to load this table. Any program that used this pay table would have to use a similar routine to read the table file and construct the table in storage.

I'll assume for this example that the format of the table is the same as that defined in figure 7-2. However, I'll allow up to 15 entries, each one consisting of a two-byte EBCDIC pay class and a two-byte packed-decimal pay rate. The pay-table input cards have the pay class punched in columns 1 and 2 and the corresponding pay rate in columns 5-8 with an assumed decimal point between columns 6 and 7. The last card in the table file has nines in columns 1 and 2 to indicate the end of the file; this is a common practice for input tables. Figure 7-5 illustrates a routine for loading this table into storage.

The pay table is loaded by looping through a GET macro that reads each input table-entry card. Before the loop is entered, register 5 is loaded with the address of the table area. After a table card has been read, the CLC instruction compares the pay-class field in the input card for all nines. If all nines, the end of the table input has been reached and control is passed to the main processing

routine. If the pay class isn't nines, it is considered a valid table entry. In this case, the pay class is moved from the card area to the first two bytes of the table-entry field whose address is in register 5:

```
MVC   0(2,5),TCRDCLAS
```

The first operand indicates a displacement of zero from the address in register 5 and a length code of two bytes. Then, the pay rate assigned to the pay class is packed from the card-input field into the second two bytes of the table entry by using an explicit first operand:

```
PACK  2(2,5),TCRDRATE
```

After the first table entry has been constructed, the LA instruction increases the address in register 5 by the length of a table entry, four bytes. It will now point to the next table-entry field to be filled. The branch back to the GET statement completes the loop.

SUMMARY

In summary, handling tables in assembler language is quite easy. Generally, you search tables with simple loops by using explicit operands and manipulating the entry addresses in a substitute base register. Whenever you must design a table and the search routine that goes with it, your objectives should be (1) to cover all possible conditions and (2) to maximize processing efficiency.

Terminology

single-level table
factor matching
positional table

Objective

Given a programming problem involving single-level tables, code a BAL solution.

```
          .
          .
          .
          LA    5,PAYTABLE
READTCRD  GET   CARDIN,TCRD
          CLC   TCRDCLAS,=C'99'
          BE    PROCESS
          MVC   0(2,5),TCRDCLAS
          PACK  2(2,5),TCRDRATE
          LA    5,4(5)
          B     READTCRD
          .
          .
          .
TCRD      DS    0CL80
TCRDCLAS  DS    CL2
          DS    CL2
TCRDRATE  DS    CL4
          DS    CL72
          .
          .
          .
PAYTABLE  DS    15CL4
          .
          .
          .
CARDIN    DTFCD DEVADDR=SYSIPT,....
          .
          .
          .
```

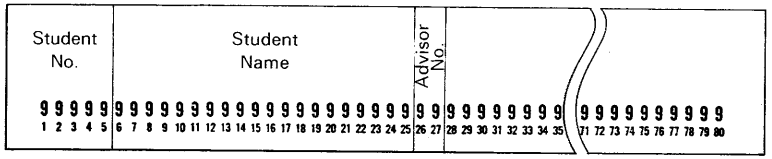
FIGURE 7-5 Table-Loading Routine

```
ADVLIST START 0
BEGIN    BALR  3,0
         USING *,3
         OPEN  CARDIN,REPORT
PROCESS  MVI   PRTIO,X'40'
         MVC   PRTIO+1(131),PRTIO
         GET   CARDIN
         MVC   PRTSNBR,CRDSNBR
         MVC   PRTSNAME,CRDSNAME
         LA    4,ADVTABLE
TABLOOP  CLC   CRDANBR,0(4)
         BE    ADVFOUND
         CLC   TABEND,0(4)
         BE    ADVFOUND
         LA    4,22(4)
         B     TABLOOP
ADVFOUND MVC   PRTANAME,2(4)
         PUT   REPORT
         B     PROCESS
EOFCRD   CLOSE CARDIN,REPORT
         EOJ
CARDIN   DTFCD DEVADDR=SYSIPT,IOAREA1=CRDIO,EOFADDR=EOFCRD
REPORT   DTFPR DEVADDR=SYSLST,IOAREA1=PRTIO,BLKSIZE=132
ADVTABLE DS    0CL176
         DC    C'12MR. KAUFFMAN       '
         DC    C'18MS. BAIM           '
         DC    C'24MS. CORDER         '
         DC    C'37MR. RYAN           '
         DC    C'42MRS. HOGG          '
         DC    C'51MR. VANHISE        '
         DC    C'56MR. BERDOU         '
         DC    C'99INVALID ADVISOR NO.'
TABEND   DC    C'99'
CRDIO    DS    0CL80
CRDSNBR  DS    CL5
CRDSNAME DS    CL20
CRDANBR  DS    CL2
         DS    CL53
PRTIO    DS    0CL132
PRTSNBR  DS    CL5
         DS    CL5
PRTSNAME DS    CL20
         DS    CL5
PRTANAME DS    CL20
         DS    CL77
         END   BEGIN
```

FIGURE 7-6 Advisor-Lookup Program

Problem

Write a program to list the advisors assigned to the students as recorded in a deck of cards having this format:



The report should have this format:

STUDENT NO.	STUDENT NAME	ADVISOR NAME
PP 1–5	PP 11–30	PP 36–55

In order to print the advisor name, look it up by number in the following table which you must define in your program:

Advisor No.	Advisor Name
12	Mr. Kauffman
18	Ms. Baim
24	Ms. Corder
37	Mr. Ryan
42	Mrs. Hogg
51	Mr. Vanhise
56	Mr. Berdou

If an advisor number can't be found in the table, print an error message in the advisor-name area. (Note: Don't bother with headings and line control; concentrate on the table-handling aspects of the problem.)

Solution

Figure 7-6 is an acceptable solution. Notice how the end-of-table entry in the table area is used to print the INVALID ADVISOR NO. message.

TOPIC TWO Multilevel Tables For many table-lookup problems, single-level tables can't do the job. In such cases *multilevel tables* must be used. Income tax withholding tables, for instance, vary based on two factors: amount of pay and number of dependents. Because two variables are involved, an income tax table can be referred to as a two-level table.

Three-level tables are also relatively common. Figure 7-7, for example, is a three-level insurance-rating table. The three variable factors—age, sex, and job class—determine the premium to be charged for accident insurance.

	Men		Women	
Age	Class 1	Class 2	Class 1	Class 2
18-34	$23.50	$27.05	$24.75	$28.45
35-39	24.00	27.55	25.80	29.50
40-44	24.60	28.15	27.10	30.80
45-49	25.30	28.85	29.10	32.80
50-54	26.30	29.85	31.55	35.25
55-59	28.00	31.55	35.00	38.70

FIGURE 7-7 Three-Level Rating Table

A two- or three-level table can be designed as a factor-matching type, positional type, or a combination of the two. For the table in figure 7-7, I would use factor matching for the age variable, and positional lookup for the sex and job class variables. I would choose factor matching for the age factor for two reasons: (1) because each table entry serves a range of ages, not just one age, and (2) because the size and range of the age brackets are likely to change. If the age factor were to be treated in positional terms, a change in the number of age brackets would cause a change in the table search routine. On the other hand, the number of sex and job class categories are fixed. These factors can therefore be handled most efficiently by using table position.

```
          .
          .
          .
          LA     7,RATETAB
          PACK   AGEWRK,INPAGE
          CP     AGEWRK,0(2,7)
          BL     AGEERR
AGELOOP   CP     0(2,7),=P'99'
          BE     AGEERR
          CP     AGEWRK,2(2,7)
          BNH    AGEFOUND
          LA     7,16(7)
          B      AGELOOP
AGEFOUND  LA     7,4(7)
          CLI    INPSEX,C'M'
          BE     MALE
          LA     7,6(7)
MALE      CLI    INPJOBCL,C'1'
          BE     RATEFND
          LA     7,3(7)
RATEFND   ZAP    RATEWRK,0(3,7)
          .
          .
          .
AGEWRK    DS     PL2
RATEWRK   DS     PL3
RATETAB   DS     0CL112
          DC     PL2'18'
          DC     PL2'34'
          DC     PL3'2350'
          DC     PL3'2705'
          DC     PL3'2475'
          DC     PL3'2845
          DC     PL2'35'
          DC     PL2'39'
          DC     PL3'2400'
          DC     PL3'2755'
          DC     PL3'2580'
          DC     PL3'2950'
          DC     PL2'40'
          DC     PL2'44'
          DC     PL3'2460'
          DC     PL3'2815'
          DC     PL3'2710'
          DC     PL3'3080'
          DC     PL2'45'
          DC     PL2'49'
          DC     PL3'2530'
          DC     PL3'2885'
          DC     PL3'2910'
          DC     PL3'3280'
          DC     PL2'50'
          DC     PL2'54'
          DC     PL3'2630'
          DC     PL3'2985'
          DC     PL3'3155'
          DC     PL3'3525'
          DC     PL2'55'
          DC     PL2'59'
          DC     PL3'2800'
          DC     PL3'3155'
          DC     PL3'3500'
          DC     PL3'3870'
          DC     PL2'99'
          DC     PL2'99'
          DC     4PL3'9999'
```

FIGURE 7-8 Three-Level Table Lookup

I would define this three-level table as shown in figure 7-8. Here, there are six entries for each age bracket as follows:

```
RATETAB  DS    0CL112
         DC    PL2'18'          Low age limit
         DC    PL2'34'          High age limit
         DC    PL3'2350'        Rate for men, class 1
         DC    PL3'2705'        Rate for men, class 2
         DC    PL3'2475'        Rate for women, class 1
         DC    PL3'2845'        Rate for women, class 2
          .
          .
          .
         DC    PL2'55'          Low age limit
         DC    PL2'59'          High age limit
         DC    PL3'2800'        Rate for men, class 1
         DC    PL3'3155'        Rate for men, class 2
         DC    PL3'3500'        Rate for women, class 1
         DC    PL3'3870'        Rate for women, class 2
         DC    PL2'99'
         DC    PL2'99'
         DC    4PL3'999999'
```

Age Segment 1

Age Segment 6

End-of-Table Entry

The last entry has nines in the age-bracket entries to indicate the end of the table.

Figure 7-8 also gives a table-lookup routine for finding the proper rate when age, sex code (M or F), and job-class code (1 or 2) are input fields. First, the address of the table is loaded into register 7, the input age is packed, and it is compared with the lower limit of the first age-bracket entry:

```
CP     AGEWRK,0(2,7)
```

If the input age is lower than this lower limit of the table, a branch to AGEERR takes place and the table lookup isn't performed.

The program next enters a factor-matching loop for the proper age bracket. In this loop, a lower limit of 99 indicates end-of-table has been reached and a branch to AGEERR takes place. To find the proper bracket, the input age is compared with the higher limit of each age segment. If the age is lower than or equal to this higher limit, the proper bracket has been found. Otherwise, 16 is added to register 7 (the size of each age segment) and the loop is repeated.

When the age bracket is found, four is added to register 7. Then, the sex code is compared to M. If they are equal, the program goes to MALE and register 7 is left unchanged. If unequal, indicating a female, the program adds six to the address in register 7 so it points to the first of the women's rates instead of the men's.

Finally, a comparison of the input job class is made. If the job class is equal to one, the register is pointing at the proper rate-table entry. If it isn't one, the input pay class must be two. In the latter case, three is added to the address in register 7 so it points to the rate that should be used.

As you can see, then, this table-lookup routine is actually three single-level table lookups combined. The age-group lookup picks a table segment rather than an individual table entry. Then, within an age segment, the sex code picks a smaller segment composed of two individual rate entries. Finally, the job class is used to select one of the two rate entries.

One assumption made by this lookup routine is that neither sex nor job-class code will be invalid. In actual practice, the routine would not make this assumption. Instead, all valid codes would be tested for equal comparisons and appropriate messages would be printed if invalid codes were found.

This same type of table structure can be used for more than three levels. Each level of the structure corresponds to one input factor and can use either factor-matching or positional-organization techniques. Most complex tables are stored in files so they can be updated without changes to the programs that use them. As a result, you must keep future changes in mind when you design tables and be sure to reserve space for these changes in your processing programs. You must also code your processing routine so it is adaptable to table changes. Notice in figure 7-8 that the routine used will work whether the number of age

brackets is increased or decreased, whether the lower limit of the first age bracket is lowered, or whether any of the other age limits are changed.

Terminology

multilevel table

Objective

Given a programming problem involving a multilevel table, code a BAL solution.

Problem

Assume that the premium table processed in figure 7-8 is stored on cards rather than being stored as constants within the program. The table file consists of six cards with this format:

CARD COLUMNS	DATA
1-2	Lower age limit
3-4	Upper age limit
5-8	Rate for men class 1
9-12	Rate for men class 2
13-16	Rate for women class 1
17-20	Rate for women class 2

The seventh table card has nines in columns 1-4. Write a routine that loads these table entries into the table defined in figure 7-8 assuming the card file, named CARDTAB, uses one I/O area with this definition:

```
CARDIO   DS    0CL80
LLIMIT   DS    CL2
HLIMIT   DS    CL2
MC1      DS    CL4
MC2      DS    CL4
WC1      DS    CL4
WC2      DS    CL4
         DS    CL60
```

Solution

The following instructions are an acceptable load routine:

```
         LA    7,RATETAB
LOADRT   GET   CARDTAB
         CLC   LLIMIT,=C'99'
         BE    PROCESS
         MVC   0(4,7),LLIMIT
         PACK  4(3,7),MC1
         PACK  7(3,7),MC2
         PACK  10(3,7),WC1
         PACK  13(3,7),WC2
         LA    7,16(7)
         B     LOADRT
```

8

Editing, Bit Manipulation, and Translation

One of the weaknesses of a high-level language such as COBOL or FORTRAN lies in its limited ability to manipulate or translate individual bytes or the bits within those bytes. After presenting some additional patterns for the System/360-370 edit instructions, this chapter presents the BAL instructions for bit manipulation and byte translation.

EDITING

Some simple edit patterns for the edit instruction have been presented in chapters 2 and 3. However, additional editing capabilities are made possible by use of the edit instruction and the edit-and-mark instruction.

Edit (ED) Chapter 2 presents edit patterns used to suppress lead zeros and to insert commas and decimal points into

Group	Sending Field	Receiving-Field Pattern	Edited-Result Field	Printed-Result Field
1 Non-Blank Fill Character	12345C	5C2020202020	5CF1F2F3F4F5	*12345
	00123F	5C2020202020	5C5C5CF1F2F3	***123
	00000C	5C2020202020	5C5C5C5C5C5C	******
	123456789C	5B206B2020206B2020214B2020	5BF16BF2F3F46BF5F6F74BF8F9	$1,234,567.89
	000000123C	5B206B2020206B2020214B2020	5B5B5B5B5B5B5B5B5BF14BF2F3	$$$$$$$$$1.23
	000000005C	5B206B2020206B2020214B2020	5B5B5B5B5B5B5B5B5B5B4BF0F5	$$$$$$$$$.05
2 Message Characters for Negative Fields	01234C	402020202020 60	4040F1F2F3F440	1234
	01234D	402020202020 60	4040F1F2F3F460	1234-
	00000C	402020202020 60	40404040404040	
	00123C	402020214B202040C3D9	404040F14BF2F3404040	1.23
	00001D	402020214B202040C3D9	404040404BF0F140C3D9	.01 CR
	00000C	402020214B202040C3D9	404040404BF0F0404040	.00
3 Date Editing	0020474F	40202021612020612020	404040F261F0F461F7F4	2/04/74
	0121974F	40202021602020602020	4040F1F260F1F960F7F4	12-19-74
	0120274F	40202021402020402020	4040F1F240F0F240F7F4	12 02 74
4 Field Separators	123C123C123F	402021202222220212022222202120	40F1F2F34040F1F2F34040F1F2F3	123 123 123
	100C000C001C	402021202222220212022222202120	40F1F0F040404040F040404040F1	100 0 1
	123C12345C	402021202222222202020214B2020	40F1F2F3404040F1F2F34BF4F5	123 123.45
	001C00000C	402121202222222202020214B2020	404040F1404040404040404BF0F0	1 .00

FIGURE 8-1 Advanced Editing Patterns

numbers (see figure 2-9). Figure 8-1 illustrates some additional editing patterns. The first of these patterns shows how a nonblank fill character can be used. By using a dollar sign or asterisk as the fill character, a string of characters is made to precede the first significant digit in the edited result. This feature is commonly used when printing checks so the amount of the check cannot easily be tampered with.

The second group of patterns shows methods of indicating negative values. If you remember, in the patterns discussed in chapter 2, all results, negative or positive, are printed as positive values. This is not the case with the patterns discussed in this chapter. When one of the patterns shown in figure 8-1 is used, message characters to the right of the rightmost digit position are left unchanged if the number being edited is negative. If the number being edited is positive, these rightmost message characters are replaced by a fill character. As a result, one or more message characters can be printed after a negative number. Any message characters can be used to indicate a negative field, but the most common ones are CR, DB, and the minus sign.

The third group of patterns illustrates date-field editing. Here, the message characters are inserted into a seven-digit date field.

The fourth group of patterns illustrates the use of one edit instruction to edit two or more fields. To do this, the fields to be edited must be located in successive bytes of storage and the receiving field must have one or more *field separators* (hex 22s) between the individual edit

patterns. As you can see in figure 8-1, only one fill character (in the leftmost byte) is used in the receiving field. This fill character is used for all fields to be edited and also replaces all field separators during editing.

Edit-and-Mark (EDMK) The edit-and-mark instruction is used primarily in programs that print money values. The EDMK instruction operates in exactly the same manner as the edit instruction except that it also saves the address of the first significant digit in the edited area. This address is stored in register 1. This address can then be used to place a dollar sign right next to the first digit of the edited value. This is often referred to as using a *floating dollar sign*. Here's a sample routine that uses the edit-and-mark instruction:

```
          MVC   PRTVALUE,PATTERN
          EDMK  PRTVALUE,VALUE
          S     1,=F'1'
          MVI   0(1),C'$'
          .
          .
          .
PRTVALUE  DS    CL7
VALUE     DS    PL3
PATTERN   DC    XL7'402020214B2020'
```

If the value is 1.23, the edited result will print as $1.23. Notice that one must be subtracted from register 1 before the dollar sign can be inserted into the edited result.

A complication occurs when the sending field contains no significant digits until after the significance starter in the receiving field has been reached. In such a case, because significance is started by the significance starter and not a significant digit, no address will be loaded into register 1 by the EDMK instruction. Then, when the MVI instruction is executed, there won't be a proper address in register 1. To avoid this, the EDMK instruction should be preceded by an instruction that loads register 1 with a default value as shown in figure 8-2. Then, if no significant digit is found, the dollar sign will be placed in the appropriate position. If, for example, the value to be edited by the routine in figure 8-2 is .05, the edited-result field will print as: $.05.

```
          .
          .
          .
          MVC   PRTVALUE,PATTERN
          LA    1,PRTVALUE+4
          EDMK  PRTVALUE,VALUE
          S     1,=F'1'
          MVI   0(1),C'$'
          .
          .
          .
PRTVALUE  DS    CL7
VALUE     DS    PL3
PATTERN   DC    XL7'402020214B2020'
          .
          .
          .
```

FIGURE 8-2 Using the EDMK Instruction

BIT MANIPULATION

Bit manipulation instructions are often called "bit twiddlers" because they are used to test or change selected bits in an eight-bit byte. The three most important instructions in this group are the OR, the AND, and the test-under-mask instruction.

OR (OI, OC, O, OR) When an OR instruction is executed, the bits in the sending field (operand-2) are used to modify the bits in the receiving field (operand-1) according to this table:

Operand-2 \ Operand-1	0	1
0	0	1
1	1	1

OR Table

```
          .
          .
READCARD  GET   PAYCRD
          LA    4,7
          LA    5,PHRSDAY1
          LA    6,0
LOOP1     LA    7,3
LOOP2     TM    0(5),X'F0'
          BO    OKDIGIT
          OI    FLDERR,X'FF'
OKDIGIT   LA    5,1(5)
          BCT   7,LOOP2
          TM    FLDERR,X'FF'
          BZ    NXTFLD
          LA    6,1(6)
          NI    FLDERR,X'00'
NXTFLD    BCT   4,LOOP1
          C     6,BCON0
          BE    NOERRS
          B     ERRS
          .
          .
          .
FLDERR    DC    X'00'
BCON0     DC    F'0'
PAYCRD    DS    0CL80
PEMPNBR   DS    CL4
PEMPNAME  DS    CL30
PPAYRATE  DS    CL4
PHRSDAY1  DS    CL3
PHRSDAY2  DS    CL3
PHRSDAY3  DS    CL3
PHRSDAY4  DS    CL3
PHRSDAY5  DS    CL3
PHRSDAY6  DS    CL3
PHRSDAY7  DS    CL3
          DS    CL21
          .
          .
```

FIGURE 8-3 Input Validation Routine

byte might be named ERRSWTCH and its bits might have these meanings:

```
Bit 1—Error in FIELD1
Bit 2—Error in FIELD2
Bit 3—Error in FIELD3
Bit 4—Error in FIELD4
```

You could then use a test-under-mask instruction to check ERRSWTCH for one or more on-bits before processing the input transaction:

```
          TM    ERRSWTCH,B'11110000'
          BZ    PROCESS
```

Here, the branch to PROCESS will occur only if all of the error-switch bits tested (bits 1-4) are off. If one or more is on, control will fall through the branch to the next instruction, which should be the first instruction of the error print routine.

In a switch-setting routine like this, you can use only one bit, instead of four, to indicate an input-data error. But this means you won't be able to tell, in the print routine, which of the four fields caused the error. Since one objective of a validation routine is to make it easy for data clerks to correct any errors, you would probably want to indicate the error field or fields on the error listing.

Figure 8-3 illustrates bit manipulation in a short validation routine for payroll cards. These cards contain the number of hours worked on each of the seven days of the week in seven three-column fields (one decimal position). The validation routine uses the TM instruction to test each byte in each of the seven input fields for a valid numeric character. If the zone portion of each input byte is all ones (hex F), the byte contains a valid number and the next byte is tested. If it isn't valid, hex FF is ORed into the byte named FLDERR. After each field is checked, the TM instruction tests FLDERR to see if it is hex FF. If yes, one is added to register 6, which counts the number of invalid fields in a card, and the NI instruction turns off the bits in FLDERR. The inner loop in this routine (LOOP2) is repeated three times for each hours-worked field and

the outer loop (LOOP1) is repeated seven times for each card.

TRANSLATION

Most high-level languages are very limited when it comes to translating data from one code to another. They also are limited when it comes to handling free-form input data—for instance, address fields of variable lengths with the end of each field indicated by a slash. For such tasks, a BAL subprogram using the System/360-370 translate, translate-and-test, and execute instructions is often used.

Translate (TR) The translate instruction can translate the bit pattern of each byte in a field to any other bit pattern. This instruction works in conjunction with a table that is defined by the program and that gives the bit pattern of the replacement code. Since there are 256 different patterns for an eight-bit byte, the maximum size of the translate table is 256 bytes but, as you will see, it is sometimes possible to use a smaller table. To code the instruction, the programmer uses the name of the field to be translated as operand-1 and the name of the table to be used as operand-2:

```
TR    FIELDA,TABLE
```

When the instruction is executed, the field is translated from left to right, one byte at a time. To find the appropriate code in the table, the code in the operand-1 field is treated as a binary value and added to the address of the first byte of the table. The byte at the resulting address then replaces the byte in the operand-1 field.

To illustrate, suppose the following table is used:

```
TABLE   DS    0CL256
        DC    192X'00'
        DC    X'010203040506070809'
        DC    7X'00'
        DC    X'0A0B0C0D0E0F101112'
        DC    8X'00'
        DC    X'131415161718191A'
        DC    23X'00'
```

Suppose also that the input field (FIELDA in the above instruction) contains MOD14. When the instruction is executed, the binary value of M is first added to the address of TABLE. Since M is hex D4, or binary 11010011, it has a value of 212. As a result, the 213th byte in the table is substituted for the letter M. Since this byte contains hex 0D, the first byte of FIELDA will contain hex 0D after execution.

The instruction continues to execute in this way on the remaining bytes in FIELDA. Since the letter O has a binary value of 214, the 215th table value, hex 0F, is substituted for it. Since the letter D has a binary value of 196, the 197th table value, hex 04, is substituted for it. Since the numbers 1 (hex F1) and 4 (hex F4) have binary values of 241 and 244, hex 00 is substituted for each of them. When the instruction finishes its execution, FIELDA contains this data:

```
0D 0F 04 00 00
```

In practice, you can usually avoid defining a table of 256 bytes because the input data is normally restricted to a smaller range. For example, if you are translating the alphabetic characters of the EBCDIC code to some other bit patterns, the input range is from hex C1 (A) to hex E9 (Z). As a result, the translate table used to cover the range need be only 41 bytes long. (Hex C1 through hex E9 equals binary values 193 through 233.)

To reference this table in the translate instruction, you adjust the beginning address of the second operand so the lowest binary value in the input range results in a displacement of zero. Since the low end of the range is hex C1 in this example, the translate instruction should be coded as:

```
TR    DATA,TABLE-193
```

The effect is that an input character A will be translated into the first byte of the table:

```
(TABLE-193) + X'C1' = (TABLE-193) + 193
                    = TABLE + 0
```

```
          .
          .
          .
          GET   TAPEFLE,TAPEREC
          TR    TAPEREC,TRANSTAB
          .
          .
          .
TRANSTAB  DC    X'F0F1F2F3F4F5F6F7F8F9'
          DC    X'7D7E7A4D6E50F0'
          DC    X'C1C2C3C4C5C6C7C8C9'
          DC    X'5E4B5D4D5DF0'
          DC    X'D0D1D2D3D4D5D6D7D8D9'
          DC    X'7B5B5C7D5ED04C61'
          DC    X'E2E3E4E5E6E7E8E9'
          DC    X'7C6B4D605D4A'
TAPEREC   DS    CL200
          .
          .
          .
```

FIGURE 8-4 Translation Routine

Note, however, that a bit pattern that isn't between hex C1 and hex E9 will not be translated properly.

Though such a need is rare, the translate instruction is ideal for translating from one data code to another. For instance, I once wrote a program to translate a group of magnetic tape files written by a Honeywell 200 computer to IBM System/360 format. The Honeywell tapes were written in *octal code*. Each character was made up of six bits treated as two groups of three. The program I wrote had to read the tape records into storage, allowing the System/360 hardware to add two high-order zero bits to each six-bit character, and translate the resulting bit patterns to EBCDIC. Of course, I had to know what EBCDIC characters each of the six-bit octal codes represented so I could make up an appropriate translation table for the program. This translation routine and table is illustrated in figure 8-4. Notice that once the table has been created, the translate instruction does all the work.

In this example, the translation table didn't have to be 256 bytes long since each input byte had only six significant bits with two high-order zero bits added by the hardware. As a result, the maximum hex input value was B'00111111', or X'3F.' Since the range X'00' to X'3F' represents 64 combinations, my table only had to be 64 bytes long.

Translate-and-Test (TRT) The translate-and-test instruction operates in almost the same manner as the translate instruction. The data bytes of the first operand are used as displacements from the second operand address, which is the address of a table. Instead of replacing the data byte with the corresponding byte in the table, however, the TRT instruction only checks to see if the byte in the table is hex 00. If so, processing continues with the next byte in the first operand field. If not, execution of the TRT instruction is halted, the address of the byte in the first operand is put in register 1, and the nonzero byte from the table is inserted into the rightmost byte of register 2.

The translate-and-test instruction is used to find certain characters in an input stream. For example, you might use it in the following manner to find the first blank in an input card:

```
          TRT   INPAREA,TRTTABLE
          .
          .
          .
INPAREA   DS    CL80
TRTTABLE  DS    0CL256
          DC    64X'00'
          DC    X'40'
          DC    191X'00'
```

When the TRT is executed, each of the bytes of data in the input area (processing from left to right) will be used as a displacement from the start of the table, TRTTABLE. Since all of the table bytes except the 65th byte (displacement

of +64) are hex zeros, only a blank in the input field (hex 40) will cause the TRT to stop. Then, the address of the blank is put in register 1, and the nonzero table character, also a blank in this case, is put into the rightmost byte of register 2. If necessary, then, you can use the address in register 1 to calculate the length of the data field in the input area.

When the translate-and-test instruction is executed, there are three possible resulting conditions:

Condition Code On-Bit	Condition
Bit 0	All bytes in input field have corresponding hex zeros in the table.
Bit 1	A nonzero table byte has been found.
Bit 2	The last byte in the input field has a corresponding nonzero byte in the table.

Since you may not know whether a field will contain a byte with a corresponding nonzero table byte, you can use the branch-on-condition instruction to alter the processing sequence according to the condition code. For instance,

```
BC      8,NOCHAR
```

will branch if only zero values are found for a field, while

```
BC      2,LSTBYTE
```

will branch if a nonzero value has been found for the last byte in the operand-1 field.

Execute (EX) As illustrated in figure 8-5, the execute instruction is often used in conjunction with the translate-and-test instruction. When the execute instruction is executed, it does two things. First, it ORs the rightmost byte of the register that is specified as the first operand with the second byte (bits 8-15) of the instruction specified as the second operand. Second, it executes the operand-2 instruction using the results of the OR as the second byte

```
          .
          .
          .
          LA      4,INPAREA
          TRT     INPAREA,TRTTABLE
          SR      1,4
          S       1,=F'1'
          EX      1,MOVEINST
          .
          .
INPAREA   DS      CL80
TRTTABLE  DS      0CL256
          DC      64X'00'
          DC      X'40'
          DC      191X'00'
MOVEINST  MVC     0(0,5),0(4)
          .
          .
          .
```

FIGURE 8-5 The TRT and EX Instructions

of this instruction without actually changing the byte in the instruction. When it finishes, the instruction following the execute instruction is next to be executed.

To illustrate, suppose INPAREA as defined in figure 8-5 has this data in bytes 1-16:

```
THIS IS THE LAST
```

Suppose also that the address of INPAREA is decimal 8000. If this is the case, when the LA instruction is executed, 8000 will be loaded into register 4. When the TRT instruction is executed, it will stop only when it encounters a blank in INPAREA because the table is all hex zeros except for the 65th table entry. Since the first blank in INPAREA is the fifth byte of the field, the TRT instruction will load address 8004 into register 1 and place hex 40 in the rightmost byte of register 2. Then, the routine subtracts the contents of

```
LABELS   START 0
BEGIN    BALR  3,0
         USING *,3
         OPEN  CARD,PRINT
NEWCARD  LA    4,CARDIO
         LA    5,79
         GET   CARD
NXTFLD   EX    5,TRTINST
         BC    8,LASTFLD
         BC    4,LASTSLSH
         SR    1,4
         S     1,=F'1'
         EX    1,MOVEFLD
         PUT   PRINT
         AR    4,1
         A     4,=F'2'
         SR    5,1
         S     5,=F'1'
         MVI   PRTIO,C' '
         MVC   PRTIO+1(132),PRTIO
         B     NXTFLD
LASTFLD  LA    1,CARDIO+79
PRTLSTLN SR    1,4
         EX    1,MOVEFLD
         PUT   PRINT
         MVC   PRTIO+1(132),PRTIO
         MVI   PRTIO,C'1'
         B     NEWCARD
LASTSLSH LA    1,CARDIO+78
         B     PRTLSTLN
CRDEOF   CLOSE CARD,PRINT
         EOJ
CARD     DTFCD DEVADDR=SYSIPT,EOFADDR=CRDEOF,IOAREA1=CARDIO
PRINT    DTFPR DEVADDR=SYSLST,IOAREA1=PRTIO,CTLCHR=ASA,BLKSIZE=133
CARDIO   DS    CL80
PRTIO    DC    CL133'1'
TRTTABLE DS    0CL256
         DC    96X'00'
         DC    X'61'
         DC    159X'00'
TRTINST  TRT   0(0,4),TRTTABLE
MOVEFLD  MVC  PRTIO+1(0),0(4)
         END   BEGIN
```

FIGURE 8-6 Free-Form Labeling Program

register 4 from register 1, leaving a value of 4, the length of the first input word. Because the length stored in an instruction is one less than the number of bytes operated upon, the program next subtracts one from register 1 leaving a value of 3.

The last instruction of the routine is the execute instruction. First, it ORs the rightmost byte in register 1 with the second byte of the MVC instruction named MOVEINST. Since the move instruction has been given a length of zero, its second byte is hex 00, and the result of the OR operation

is hex 03. Then, the execute instruction causes the MVC instruction to be executed using the length code of hex 03. The result is that the word addressed by register 4 with a length of hex 3 (the input word THIS) is moved to the four bytes starting at the address given by register 5. Note, however, that the execute instruction wouldn't work as intended if the length code specified in the MVC instruction wasn't hex zeros because the length code desired wouldn't OR properly.

Figure 8-6 illustrates an entire program that uses free-form input. The input is a deck of address cards with one complete address in each card; the output is a number of two-, three-, or four-line mailing labels. The difficult part of the program is determining where the input data for one address line ends and input data for the next begins; this is because a single slash is used to separate input lines. Furthermore, the final address line in a card may or may not be ended by a slash. After a label has been printed, the program should skip to channel 1 using ASA control characters before printing the next label.

At the start of the program, the address of the input area is loaded into register 4; the length of the input area minus one is loaded into register 5; and a card is read. Then, this execute instruction is executed:

```
EX      5,TRTINST
```

Since register 5 contains the length of the input area minus one, the instruction named TRTINST

```
TRTINST  TRT    0(0,4),TRTTABLE
```

is executed using the field at the address in register 4, and having a length of 80 as the first operand—in other words, the field is the card-input area.

The two BC instructions following the execute instruction branch if only zero values are found for the input area or if the nonzero value found is in the last byte of the first operand. Since either of these conditions indicates the last address field in a card, the program branches to appropriate last-line routines. Otherwise, these instructions are encountered:

```
SR      1,4
S       1,=F'1'
EX      1,MOVEFLD
PUT     PRINT
```

Here, the length of the address field minus one is calculated in register 1. Then, register 1 is used in the execute instruction to modify the length in the MVC instruction named MOVEFLD. When its execution is complete, one address field has been moved to the output area, at which time one line has been printed on the label.

After printing a line, these instructions are executed:

```
AR      4,1
A       4,=F'2'
SR      5,1
S       5,=F'1'
MVI     PRTIO,C' '
MVC     PRTIO+1(132),PRTIO
B       NXTFLD
```

First, register 4 is adjusted so it addresses the first byte of the next address field. Second, register 5 is adjusted so it contains the length minus one of the remaining bytes in the input area. Third, the print area including the control character is cleared to blanks. (Since address lines of different lengths are being printed, data from a long line would overlap data from a shorter line and print again if the area were not cleared.) Finally, the program branches back to the first execute instruction so the loop is repeated for the next address line.

If the last address field is indicated, an appropriate length is developed in register 1, and the MVC instruction is executed via the execute instruction. Next, a line is printed, the print area is cleared, and the line-control character is set to 1 to indicate a skip to channel 1 when the next line is printed. The program then returns to NEWCARD to set up registers 4 and 5 and read another card.

SUMMARY

The elements presented in this chapter illustrate some of the power of assembler language. For instance, it is very

2 The following is an acceptable solution:

```
          .
          .
          .
          TRT    IPARTNUM,TRTTABLE
          BC     8,NOERR
          OI     ERRBYTE,B'00100000'
          .
          .
          .
TRTTABLE  DS     0CL256
          DC     64X'FF'
          DC     X'00'
          DC     128X'FF'
          DC     9X'00'
          DC     7X'FF'
          DC     9X'00'
          DC     8X'FF'
          DC     8X'00'
          DC     6X'FF'
          DC     10X'00'
          DC     6X'FF'
```

Note that only the table bytes for the blank, the letters, and the digits are hex zeros. All others are hex Fs to indicate an invalid character.

3 Figure 8-8 is an acceptable solution. Note that the first length in the PACK instruction, the length of the receiving field is unchanged by the ORing effect of the execute instruction because the corresponding bits in the register will be zeros.

9

Subroutines and Subprograms

This chapter is divided into two topics. The first describes the use of subroutines; the second describes the use of subprograms. The use of subroutines is important because it can eliminate the duplication of code within a program and coding within a department. The use of subprograms is important because it allows a large programming task to be broken down into smaller independent tasks. Also, subprograms allow a program written in one language—for instance, COBOL—to use a routine written in another language such as Basic Assembler Language.

TOPIC ONE Subroutines A *subroutine* is a group of instructions that is used within a larger routine or within a complete program. Usually, a subroutine is coded outside of the mainline routine of a program and is branched to whenever it is needed.

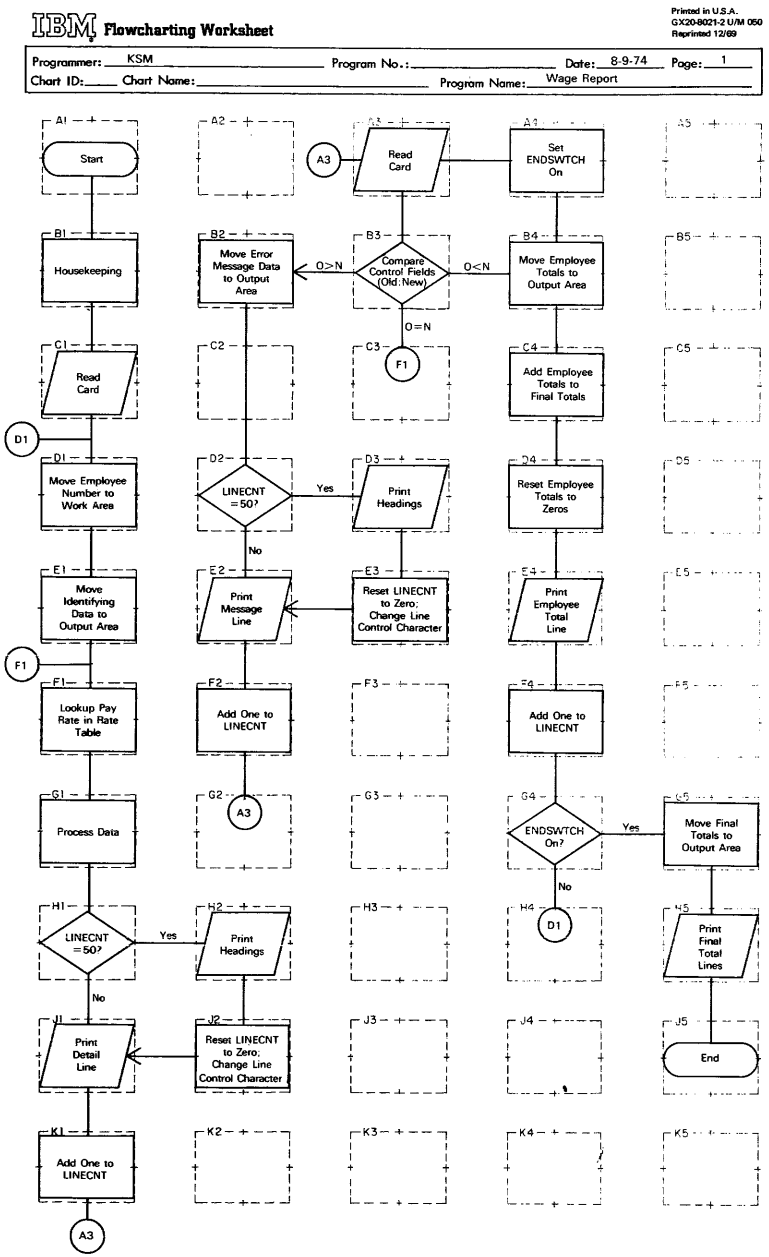


FIGURE 9-1 Wage Report Flowchart

The last instruction of the subroutine must then return to the mainline routine.

Subroutines are useful because there are many cases when the same routine is required in two or more portions of a program. By coding the routine as a subroutine, it need be coded only once. This reduces the amount of coding required by the programmer and the amount of storage required by the program.

The flowchart in figure 9-1 has been designed to illustrate the value of subroutines. This flowchart presents the logic for a program that reads a deck of employee job cards and prints a report showing the amount of pay earned on each type of job, the total amount of pay for each employee, and the grand total of pay for all employees. The input cards are in sequence by employee number, one report line is printed for each input card, and a group-total line is printed after all the cards for one employee have been processed. If an out-of-sequence card is detected, an error message is printed and another card is read.

This flowchart is prepared on a standard flowcharting form that is often used by professional programmers. This form consists of 50 boxes that are numbered so that each block of the flowchart can be identified. For example, box C1 refers to the I/O symbol containing the words Read Card; J5 refers to the terminal symbol containing the word End. By using the same letter/number combinations in connector symbols, it becomes easy to follow connections within a flowchart. For instance, the connector symbol exiting from box F2, indicating a branch to connector symbol A3, connects with the symbol to the left of box A3.

The key point of this flowchart is the decision symbol in block B3. Here, the control field (employee number) of the card just read (the new card) is compared with the control field of the previous card (the old card). If the employee numbers are equal (O = N), the new card is processed. If the old employee number is less than the new one (O < N), a group-total line for the previous employee is printed, the group totals are reset to zeros, and the new card is processed for the next employee. If the old employee number is greater than the new one (O > N), an out-of-sequence error message is printed.

By analyzing this flowchart, you can see that a line-counting routine is used both before the detail line is printed and before the error-message line is printed. Also, 1 is added to the line-count field after each type of line (detail, message, or group-total) is printed. As a result, by treating the line-count routine as a subroutine and using one work area for all three print lines, duplication of several lines of code can be eliminated.

Figure 9-2 illustrates the coding in the mainline routine, the coding in the subroutine, and the *linkage* between them. To get to the subroutine, the mainline routine uses one of the BAL instructions originally presented in chapter 5. For instance, the first branch to the subroutine in figure 9-2 uses this instruction:

```
BAL    11,LCNTRT1
```

When this instruction is executed, the address of the instruction following the BAL instruction is placed in the register specified as operand-1—in this case, register 11. Then, the BAL instruction causes a branch to the instruction named as operand-2—in this case, LCNTRT1.

The subroutine is coded as usual with one exception: when the subroutine finishes, it must branch to the address stored in register 11. To do this, the subroutine uses the BR instruction as follows:

```
BR     11
```

The BR instruction is an unconditional branch to the address that is given in the register specified by the operand.

When a BAL instruction is used to branch to a subroutine, the address in the operand-1 register is called the *return address*, and the operand-2 label is called the *entry point* of the subroutine. In the example in figure 9-2, LCNTRT1 is the entry point, and the address in register 11 is the return address. A subroutine can have more than one entry point, if necessary; this is illustrated by the subroutine in figure 9-2, which has two entry points. The second entry point, used for printing total lines, is named LCNTRT2.

The instruction of the subroutine that returns to the mainline routine is called the *exit point*. Depending on the type of function performed by the subroutine, it may have

```
* MAINLINE ROUTINE
WAGERPT  START 0
BEGIN    BALR  3,0
         USING *,3
         OPEN  CARDIN,PRTOUT
         .
         .
         .
         BAL   11,LCNTRT1     PRINT DETAIL LINE
         .
         .
         .
         BAL   11,LCNTRT2     PRINT TOTAL LINE
         .
         .
         .
         BAL   11,LCNTRT1     PRINT MESSAGE LINE
         .
         .
         .
* LINE COUNTING AND PRINTING SUBROUTINE
LCNTRT1  CP    LINECNT,=P'50'
         BL    LCNTRT2
         PUT   PRTOUT,HDGLINE1
         PUT   PRTOUT,HDGLINE2
         PUT   PRTOUT,HDGLINE3
         ZAP   LINECNT,=P'0'
         MVI   WORKAREA,C'0'
LCNTRT2  PUT   PRTOUT,WORKAREA
         AP    LINECNT,=P'1'
         BR    11
* END OF SUBROUTINE
         .
         .
         .
```

FIGURE 9-2 Simple Subroutine Linkage

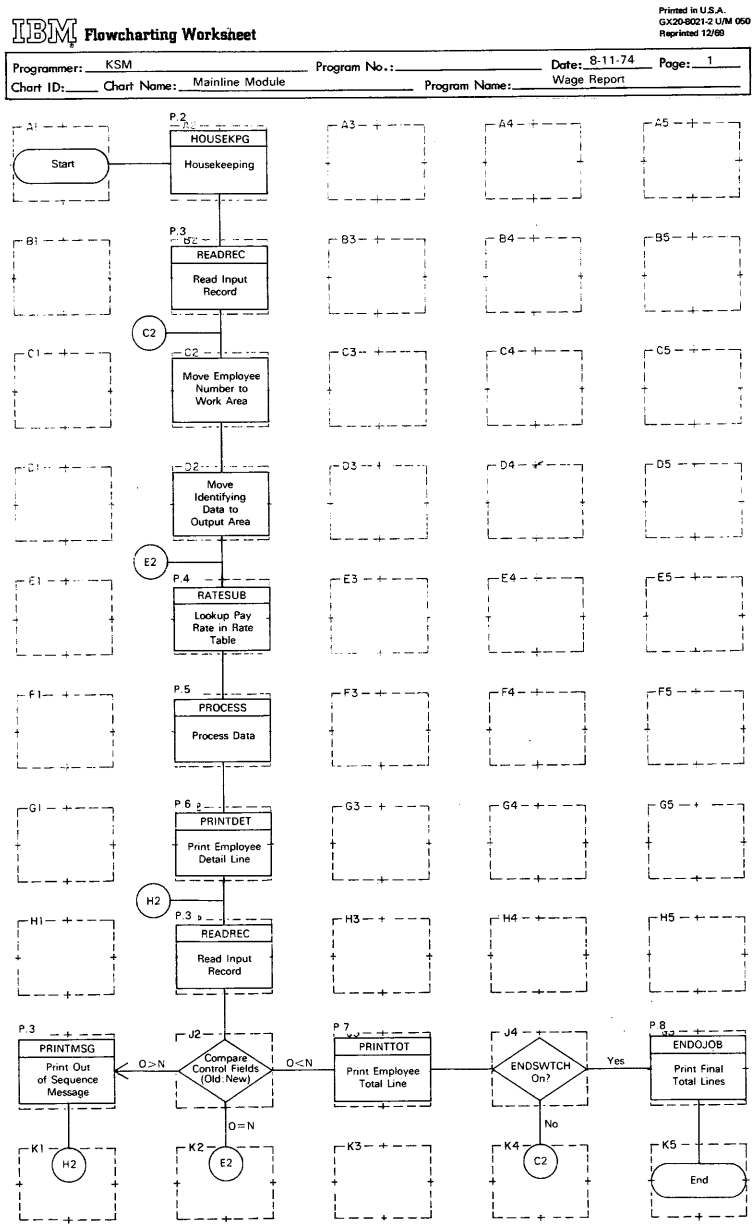


FIGURE 9-3 Wage Report Flowchart—Modular Design

multiple exit points. For instance, completion of normal processing by a subroutine might have one exit point while some error routine within the subroutine might use a different exit point.

Any register can be used for linking to a subroutine, but remember that most I/O operations use registers 14, 15, 0, and 1. If you were to use register 14 as a link register and then perform some input or output in your subroutine, the original return address in register 14 would be destroyed. However, you can use these registers if you save their contents as the first step of the subroutine and restore them just before exiting the subroutine. This save and restore technique is illustrated later in this topic.

MODULAR PROGRAM DESIGN

Although the subroutine linkage just described is valuable because it reduces coding, the concept of subroutines is also important when designing programs. In particular, the concept of *modular program design* relies upon the use of subroutines. The idea is to break a program down into a number of separate *modules*—one *mainline module* and one or more *subroutine modules*. As much as possible, each module should be independent of the other modules. The advantage of modularity is that the logic of the total program becomes more manageable. Instead of one extensive program, the program becomes a group of small, understandable programming segments. This makes it easier to code the program, and, if a problem occurs during testing, it makes it easier to locate the routine (module) that caused the error. Similarly, if the program needs to be changed later on due to changed processing requirements, locating, modifying, and testing the routine that needs to be changed can be done more efficiently than if the program weren't modular.

To achieve modularity, the mainline module should indicate all of the major processing routines as well as the logical decisions required to direct the program to these routines. For instance, figure 9-3 represents the mainline module of the wage-report program that was originally

flowcharted in figure 9-1. *Striping* is used at the top of each symbol that represents a subroutine module to indicate the name of the module it represents. As a result, the wage-report program consists of the mainline module plus eight other modules named HOUSEKPG, READREC, RATESUB, PROCESS, PRINTDET, PRINTMSG, PRINTTOT, and ENDOJOB.

Although this mainline module consists of only 13 blocks, it could well be the mainline for a program requiring hundreds of instructions; this depends upon the complexity of the subroutine modules. As a general rule, even in complex programs, the mainline module shouldn't require more than two dozen flowcharting blocks. In fact, because the mainline module's primary purpose is to direct the flow of processing to the other modules, it may well be the shortest module of the program.

After the mainline module has been flowcharted, each of the subroutines can be flowcharted. To continue the concept of modularity, each subroutine module may be broken into additional, more specific modules, depending, of course, on the length and complexity of the original subroutine module. Since the idea is to make each programming segment manageable, it is a good idea to keep each module between 50 and 200 statements long.

To make it easy to relate subroutine flowcharts to a higher level flowchart such as the mainline flowchart, cross-references are used. Examples of use of cross-references can be found in the mainline flowchart of figure 9-3 and in the subroutine flowcharts of figure 9-4. In block J1 of the mainline flowchart, for example, the stripe gives the name PRINTMSG to a subroutine. Page three (P. 3) is written to the upper left of this flowcharting symbol to indicate that the flowchart for the PRINTMSG subroutine can be found on page 3 of this set of flowcharts (page 3 is represented by figure 9-4). To complete the cross-referencing, PRINTMSG is written in the terminal symbol at the start of that subroutine flowchart on page 3, and page 1—a reference back to the mainline flowchart—is written to the upper left of its terminal symbol. Similarly, flowchart page numbers and module names are used to cross-reference the other

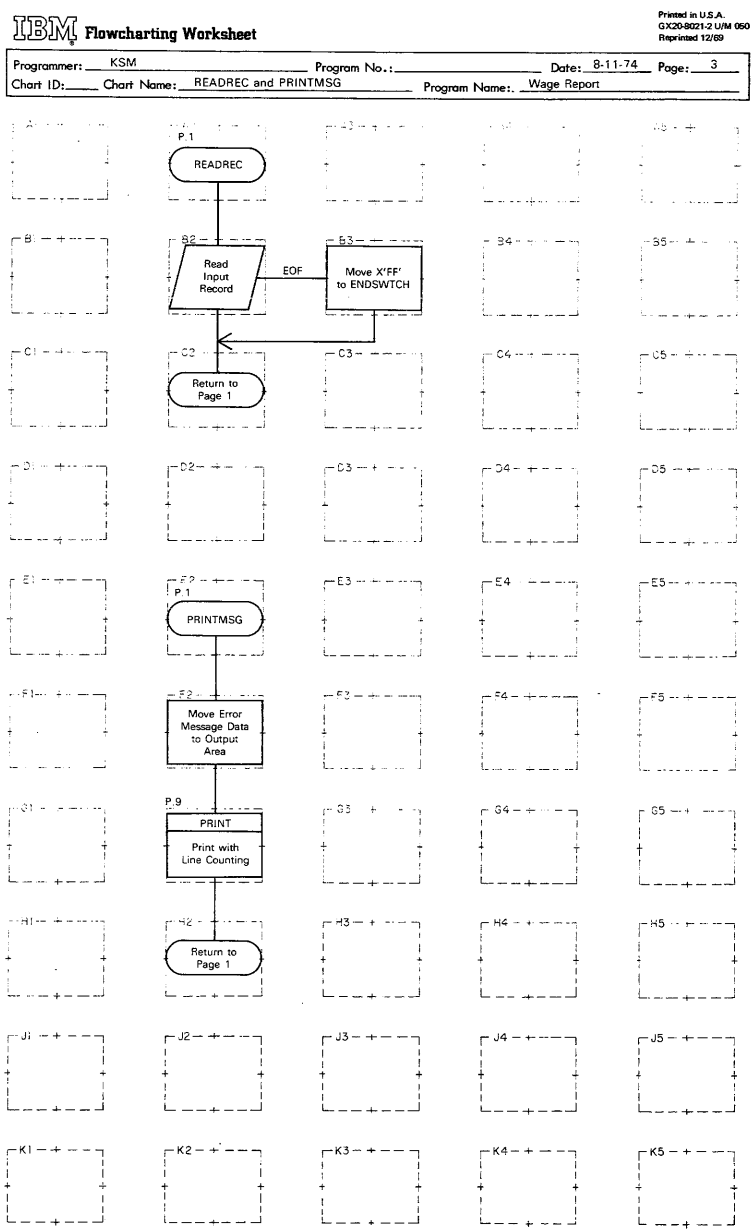


FIGURE 9-4 Wage-Report Subroutine Modules

```
* THE WAGE REPORT MAINLINE MODULE
WAGERPT  START 0
BEGIN    BALR  3,0
         USING *,3
         BAL   11,HOUSEKPG
         BAL   11,READREC
NEWCARD  MVC   WEMPNO,CEMPNO
         MVC   DEMPNO,CEMPNO
         MVC   DEMPNAME,CEMPNAME
SAMEEMP  BAL   11,RATESUB
         BAL   11,PROCESS
         BAL   11,PRINTDET
READCARD BAL   11,READREC
         CLC   WEMPNO,CEMPNO
         BH    ERROR
         BE    SAMEEMP
         BAL   11,PRINTTOT
         CLI   ENDSWTCH,X'FF'
         BNE   NEWCARD
         BAL   11,ENDOJOB
         EOJ
ERROR    BAL   11,PRINTMSG
         B     READCARD
* THE SUBROUTINES FOLLOW
         .
         .
         .
```

FIGURE 9-5 Wage-Report Mainline Module

module flowcharts and the mainline flowchart.

Within the PRINTMSG subroutine module, another subroutine module, named PRINT, is indicated. This subroutine contains the code for the line-counting and printing routine described earlier. The page number above the symbol indicates that the flowchart for the PRINT subroutine can be found on page 9 of this set of flowcharts. This flowchart will in turn refer back to page 3.

One of the critical points to bear in mind when developing a modular program is that each module should be kept independent of the other modules. In other words, each subroutine should only branch back to the next block in the mainline routine, not to other subroutines. Furthermore, no subroutine module should contain any processing that affects the flow of any other subroutine module. For example, a subroutine should never set a switch that will be tested in another subroutine. Note, then, that the ENDSWTCH set in the READREC subroutine in figure 9-4 is tested in the mainline routine only, not in a subroutine.

Although the logic of the wage-report program is simple enough that it doesn't really require modular program design, I hope you can see the value of modularity nonetheless. If you experiment with modular design, I think you will discover how it can simplify programming logic. You may even find that simple programs are more manageable when broken into at least five modules: (1) a mainline module, (2) a housekeeping module, (3) at least one input module, (4) at least one output module, and (5) an end-of-job module.

At any rate, the linkage described earlier is used when linking between mainline and subroutine modules. For instance, the code in figure 9-5 represents the mainline module in figure 9-3 as it appears when written in assembler language. Here, the BAL instruction is used to branch to the various subroutines with the return address always in register 11.

GENERALIZED SUBROUTINES

In most cases, when you write a subroutine you will also write the mainline routine that uses or calls the subroutine. As a result, at the time that you write a subroutine, you will know the names of the fields being operated upon by that subroutine. In figure 9-2, for example, the programmer knew that the names LINECNT, HDGLINE1, HDGLINE2, HDGLINE3, and WORKAREA were defined in the *calling routine*.

In some cases, however, a *generalized subroutine* is valuable. Since this type of subroutine doesn't use specific names for fields, it can be used in many different programs by many different programmers. To *pass data* from the

calling routine to the generalized subroutine, three techniques are commonly used.

Address-in-Register Technique To illustrate the *address-in-register technique*, suppose the following specific subroutine is supposed to be converted to a more general form:

```
PRINTSUB PUT   PRTOUT,WORKAREA
         AP    LINECNT,=P'1'
         MVI   WORKAREA,X'40'
         MVC   WORKAREA+1,WORKAREA
         BR    11
```

It might then be coded like this:

```
PRINTSUB PUT   PRTOUT,(9)
         AP    0(2,10),=P'1'
         MVI   0(9),X'40'
         MVC   1(132,9),0(9)
         BR    11
```

Here, the data is passed from the calling routine to the subroutine by placing the addresses of the two fields to be operated upon in registers 9 and 10. (Notice that a register in parentheses can be substituted for the work-area name in the PUT macro.) Before the subroutine can be linked to by using this technique, the calling routine must place the appropriate data addresses in registers 9 and 10 in the following way:

```
         LA    9,WORKAREA
         LA    10,LINECNT
         BAL   11,PRINTSUB
```

By using registers rather than labels, the subroutine can more easily be used in a wide variety of programs.

The limitation of this technique is the number of registers available for addressing passed data. For example, if you are using nine of the 12 registers normally available (0, 1, 14, and 15 are reserved for I/O operations), you can pass a maximum of only three addresses to the subroutine.

Address-Constant Technique The second data-passing technique, the *address-constant technique*, avoids this register usage problem. Instead of storing each data-field address in a register, the addresses are stored in consecutive fullwords of storage called *address constants*, or *adcons*. Then, a single register is loaded with the address of this list of data addresses and is passed to the subroutine.

To illustrate, suppose you want to pass five data fields to a subroutine. You would then construct an address list like this:

```
ADDRLIST DC    A(FIELD1)
         DC    A(FIELD2)
         DC    A(FIELD3)
         DC    A(FIELD4)
         DC    A(FIELD5)
```

These DC statements define address constants by using the type code A. Each constant generates a fullword that contains the address of the label enclosed in parentheses. Before linking to the subroutine, the calling routine loads a register with the address of the adcon list (register 1 is commonly used). The instructions to do this could be:

```
         LA    1,ADDRLIST
         BAL   11,SUBROUT
```

When control is passed to the subroutine, it locates the address list by using the pointer in register 1. A frequently used method of making these field addresses available for processing in the subroutine is illustrated here:

```
SUBROUT  STM   5,9,SAVE
         LM    5,9,0(1)
         .
         .
         .
         LM    5,9,SAVE
         BR    11
SAVE     DS    5F
```

First, a group of registers is stored in a save area. The address list is then loaded into the registers with a load-multiple

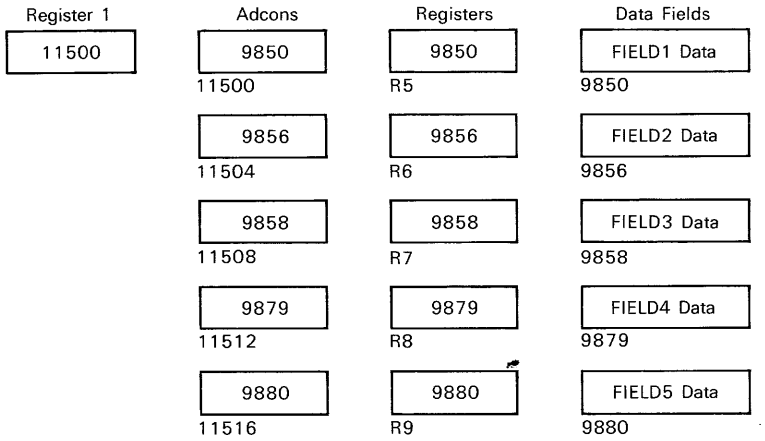


FIGURE 9-6 Adcons, Registers, and Data Fields

instruction using the pointer in register 1. When this has been done, each field can be referred to by using the appropriate register in an explicit operand. When the subroutine function has been completed, the registers are reloaded with their original contents (from the save area) and control is returned to the calling routine.

The thing you must remember when using adcons is that they contain addresses, not data. After the adcon addresses are loaded into registers, the registers are used as base registers to address the actual data fields. This process of going from register 1, to the adcons, to registers, to the data fields is illustrated schematically in figure 9-6.

To see this adcon technique used in a more complete example, look at figure 9-7. This is a table-lookup routine that could be used for the RATESUB module in the flowchart in figure 9-3. This routine is explained in detail in chapter 7 since it is a modification of figure 7-7. The important thing here, however, is to understand the linkage between the calling routine and the subroutine.

The calling routine passes two fields to the subroutine: CRDPAYCL (a field containing the pay class for a job) and PAYRATE (a field that should receive the corresponding pay rate from the pay table). To do this, two adcons are used and the address of the first adcon in the list is placed in register 1 before the calling routine branches to the subroutine.

In the subroutine, the adcon addresses are loaded into registers 6 and 7. These registers are then used as base registers when addressing the pay-class and pay-rate fields. Register 8 is used to address the table entries. If the pay class can't be found in the table, the subroutine moves packed 9s into the PAYRATE field and returns to the calling routine. Coded in this way, the pay-rate-lookup subroutine can be inserted into any program that needs this lookup function. As a result, if three different payroll programs use the lookup function, it still only needs to be coded once.

Calling-Sequence Techniques In the *calling-sequence technique*, which is a modification of the address-list technique, the address list is defined immediately following the instruction that links to the subroutine. As a result, the link register serves two functions. First, it gives a return address to the subroutine, although this return address must be adjusted by the subroutine. Second, it points to the list of adcons.

Figure 9-8 uses the pay-rate-table lookup again to illustrate this technique. As you can see, the address list is coded right after the BAL instruction in the calling routine. Because register 11 is used for the return address in the BAL instruction, the address of the adcon list is in register 11 instead of register 1. Therefore, the load-multiple instruction that loads the field addresses into registers 6 and 7 has been changed to:

```
LM      6,7,0(11)
```

When the subroutine is ready to return to the calling routine, the return address must be adjusted to skip over the adcons to the next valid instruction. In this case, since the list is two adcons long, the return address is eight bytes beyond the address passed in register 11. As a result, this unconditional-branch instruction with an explicit operand is used to return to the calling program:

```
B       8(11)
```

```
          .
          .
          .
          LA     1,ADDRLIST
          BAL    11,RATESUB
          CP     PAYRATE,=P'999'
          BE     CLASSERR
          .
          .
          .
ADDRLIST  DC     A(CRDPAYCL)
          DC     A(PAYRATE)
          .
          .
          .
CRDPAYCL  DS     CL2
          .
          .
          .
PAYRATE   DS     PL2
```

Calling Routine

```
RATESUB   STM    6,8,SAVE
          LM     6,7,0(1)
          LA     8,PAYTABLE
CMPCLASS  CLC    0(2,6),0(8)
          BE     PAYFOUND
          CLI    0(8),X'FF'
          BE     NOTFOUND
          LA     8,4(8)
          B      CMPCLASS
PAYFOUND  ZAP    0(2,7),2(8)
          B      RETURN
NOTFOUND  ZAP    0(2,7),=P'999'
RETURN    LM     6,8,SAVE
          BR     11
SAVE      DS     3F
PAYTABLE  DS     0CL40
          DC     X'F0F1221C'
          DC     X'F0F2239C'
          .
          .
          .
          DC     X'FFFFFFFF'
```

Table-Lookup Subroutine

FIGURE 9-7 Address-Constant Data-Passing Technique

Because the A type code in a DC instruction forces the address constant to be aligned on a fullword boundary, there is an alignment problem when this technique is used. During assembly, if the location counter is not an even multiple of four (at a fullword boundary), the assembler will skip bytes until a fullword boundary is reached. To illustrate, suppose the location counter value is hex 0C82 when the assembler encounters the BAL instruction in the calling routine. After the four-byte BAL instruction has been assembled, the location counter is hex 0C86. Since the A type DC statement forces alignment at a fullword boundary, the assembler skips the location counter to hex 0C88 for the first position of the address constant. The result is a gap of two undefined bytes between the BAL instruction and the first adcon. This gap will cause a serious problem during execution, because register 11 will be loaded with hex 0C86 (plus the base address) when the BAL instruction has been executed. Since hex 0C88 is the actual address of the first adcon, the subroutine will load the wrong addresses into registers 6 and 7.

The solution to this problem is to force the BAL instruction to be aligned at a fullword boundary. Then, since the BAL is a four-byte instruction, the adcon will also be at a fullword boundary. To do this, the conditional-no-operation (CNOP) instruction is used. It generates from zero to three instructions, each two-bytes long, that do nothing. They

```
          .
          .
          .
          CNOP  0,4
          BAL   11,RATESUB
          DC    A(CRDPAYCL)
          DC    A(PAYRATE)
          CP    PAYRATE,=P'999'
          BE    CLASSERR
          .
          .
          .
CRDPAYCL  DS    CL2
          .
          .
          .
PAYRATE   DS    PL2
```

Calling Routine

```
RATESUB   STM   6,8,SAVE
          LM    6,7,0(11)
          LA    8,PAYTABLE
CMPCLASS  CLC   0(2,6),0(8)
          BE    PAYFOUND
          CLI   0(8),X'FF'
          BE    NOTFOUND
          LA    8,4(8)
          B     CMPCLASS
PAYFOUND  ZAP   0(2,7),2(8)
          B     RETURN
NOTFOUND  ZAP   0(2,7),=P'999'
RETURN    LM    6,8,SAVE
          B     8(11)
SAVE      DS    3F
PAYTABLE  DS    0CL40
          DC    X'F0F1221C'
          DC    X'F0F2239C'
          .
          .
          .
          DC    X'FFFFFFFF'
```

Table-Lookup Subroutine

FIGURE 9-8 Calling-Sequence Data-Passing Technique

are conditional-branch instructions coded so that the branch is never effective. They branch on condition zero to the address in register zero.

You can use CNOP instructions to force alignment at any halfword, fullword, or doubleword boundary. The calling routine in figure 9-8, for example, contains this CNOP instruction:

```
CNOP  0,4
```

This causes the instruction that follows to be located at a fullword boundary. If the location counter isn't a multiple of four when the CNOP is encountered by the assembler program, one two-byte zero-condition branch (BCR) is generated. If the location counter is already at an even multiple of four, no zero-condition branch is generated. Thus, you can be sure that there will be no gap between the BAL instruction and the adcon that follows it.

Figure 9-9 is a chart of the valid CNOP operands and the resulting alignment. The first operand tells which byte of a fullword or doubleword you want the location counter to be adjusted to (0, 2, 4, or 8); the second operand tells whether a fullword (4) or a doubleword (8) is being referred to. Since you will want fullword alignment in this case, the operands should be 0,4.

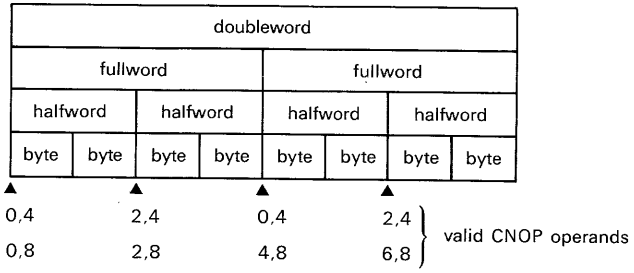


FIGURE 9-9 The CNOP Assembler Command

NESTED SUBROUTINES

As mentioned earlier, it is sometimes desirable to have subroutines within subroutines when designing programs. These can be referred to as *nested subroutines*. Many inventory-transaction-processing programs, for example, process the various types of transactions (receipts, issues, adjustments, and the like) in subroutines. Each of these processing subroutines in turn uses two lower-level subroutines—one to retrieve the appropriate master record from a master file, and the other to write the master record back on the file. Figure 9-10 is a diagram of this program structure. This three-level structure consisting of mainline routine, processing subroutines, and I/O subroutines is used in many different kinds of update programs.

The link from subroutine to subroutine in these programs is coded by using the same techniques as before. Generally, it is best to select one technique, usually address-list or calling-sequence, and use it for all of your subroutine links. Also, it is a common practice to use the same register for all links, thereby saving as many registers as possible for other uses. Each subroutine must then save the return address sent to it in the link register so that the register can be used to link to lower-level subroutines. Figure 9-11 illustrates this linkage between nested subroutines.

This type of standard subroutine linkage has several advantages for a programming staff. First, if necessary, different parts of a program can be written by different

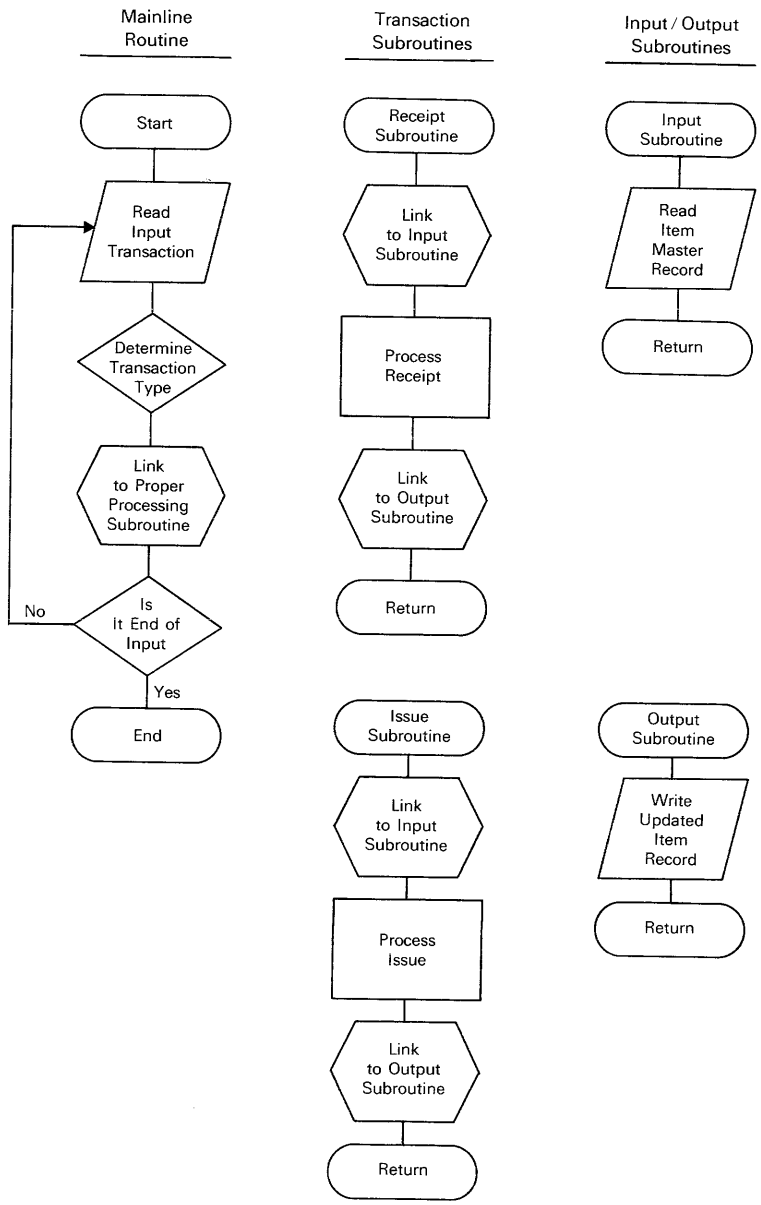


FIGURE 9-10 Nested Subroutines

Mainline Processing

```
         BAL   11,RECEIPT

         BAL   11,ISSUE
```

Transaction-Processing Subroutines

```
RECEIPT  ST    11,RECSAV11

         BAL   11,INPUT

         BAL   11,OUTPUT

         L     11,RECSAV11
         BR    11
RECSAV11 DS    F
ISSUE    ST    11,ISSSAV11

         BAL   11,INPUT

         BAL   11,OUTPUT

         L     11,ISSSAV11
         BR    11
ISSSAV11 DS    F
```

Input/Output Subroutines

```
INPUT    ST    11,INPSAV11

         L     11,INPSAV11
         BR    11
INPSAV11 DS    F

OUTPUT   ST    11,OUTSAV11

         L     11,OUTSAV11
         BR    11
OUTSAV11 DS    F
```

FIGURE 9-11 Nested Subroutine Linkage

programmers with a minimum of confusion about linking the routines together. Second, subroutines can easily be added to the program later on. Third, the save area for the link register in each subroutine provides a trace back to the calling routine. This makes the debugging task easier if the program should fail during execution and dump storage.

THE COPY STATEMENT

For subroutines that are used in only a few programs, it is common to pass subroutine source decks from one programmer to another. The programmer inserts this subroutine deck into his source deck and assembles it along with the rest of his program. When a subroutine is used in many programs, however, it is more efficient to store the subroutine in the *source-statement library* of the system-residence device. Then, the subroutine can be copied into a source program without bothering with card decks.

To copy a subroutine from the library, the BAL COPY statement is used. This assembler command has COPY as the operation code and the name of the subroutine as the operand, as in this example:

```
COPY  RATESUB
```

This command will cause the source statements for the subroutine named RATESUB to be copied from the source-statement library into the source program. The subroutine will then be assembled with the rest of the program.

Chapter 16 explains how to place a subroutine in the source-statement library. The only reason that the library can't be used for all subroutines is that it has a limited amount of storage space. Because of this, a subroutine must have sufficient use to justify placing it in the library. If a subroutine is stored in the library and is only used in two programs, it must nevertheless be kept in the library on the chance that the programs might need to be changed and thus reassembled. If the subroutine were removed from the library, the program would not be able to be reassembled without first inserting the subroutine source cards.

Terminology

subroutine
linkage
return address
entry point
exit point
modular program design
module
mainline module
subroutine module
striping
calling routine
generalized subroutine
to pass data
address-in-register technique
address-constant technique
address constant
adcon
calling-sequence technique
nested subroutines
source-statement library

Objectives

1 Give two reasons for using subroutines.
2 Given a programming problem, code appropriate routines as subroutines and code the proper linkage statements in the calling routine.
3 Given a description of a generalized subroutine, code the subroutine using any one of the three techniques for passing data described in this topic.

Problems

1 (Objective 2) You are writing a payroll program. A subroutine named GROSCALC is available for calculating the gross pay for an employee when the hourly pay rate and the number of hours worked are known. To use the subroutine, you must use the calling-sequence technique with the fields passed in this order: pay-rate, hours-worked, and gross-pay. Register 11 is to be used for the return address and the field names used in your routine are PAYRATE, HRSWRKED, and GROSSPAY.
   a. Code the linkage statements required for using this subroutine.
   b. Code a COPY statement for inserting the subroutine source statements into your program.

2 (Objective 3) Code the generalized subroutine linked to by problem 1. Each of the fields passed to your subroutine is three bytes long in packed format with two decimal positions (XXX.XX). To calculate the gross pay, multiply rate times hours worked and round the result.

Solutions

1 a. The following is acceptable code:

```
         CNOP  0,4
         BAL   11,GROSCALC
         DC    A(PAYRATE)
         DC    A(HRSWRKED)
         DC    A(GROSSPAY)
```

   b.
```
         COPY  GROSCALC
```

2 The following is acceptable code:

```
GROSCALC STM   5,7,SAVE        SAVE REGS 5,6,&7
         LM    5,7,0(11)       LOAD ADDRESSES
         ZAP   WRKPAY,0(3,5)   PUT PAYRATE IN WRKPAY
         MP    WRKPAY,0(3,6)   MULTIPLY BY HOURS
         AP    WRKPAY,RNDFCTR  ROUND PAY
         MVN   WRKPAY+4(1),WRKPAY+5 TRUNCATE 2 DIGITS
         ZAP   0(3,7),WRKPAY(5) MOVE TO GROSSPAY
         B     12(11)          EXIT
WRKPAY   DS    PL6
RNDFCTR  DC    P'50'
```

TOPIC TWO Subprograms Suppose an organization needs a certain inventory program ready for use within one month. However, the estimated programming time required for producing the program is three months. How can the deadline be met?

The usual solution is to divide the program into a main program and one or more subprograms. Then, a different programmer can work on each of the programming modules. All the programmers can use the same language, or they can use different languages. For example, the main program could be written in COBOL, subprogram A could be written in assembler language, and subprogram B in COBOL. When a programmer finishes coding a subprogram, the source language is assembled into an *object module*, which is then stored on the system-residence device. When all the modules have been assembled and tested, the main program and the subprograms are combined and the entire program is tested. By dividing the inventory program into subprograms, the time required to prepare the entire program can be reduced from three months to one month or less, depending on the number of modules used and the number of programmers assigned to the job.

In addition to reducing preparation time, there are other reasons for using subprograms. Certain routines are repeated in many different types of programs. By treating them as subprograms, they need be written and assembled only once, thus eliminating duplication of programming effort. Second, subprograms can be used to make the best use of available programming personnel: the most difficult segment of a program can be assigned to the most experienced programmer, the least difficult to the least experienced programmer, and so on. Finally, some routines can be

written more efficiently in one language than in another, and some routines can't be written at all in certain high-level languages. That's why many high-level-language programs make use of one or more assembler-language subprograms.

What is the difference between a subroutine and a subprogram? In general, a subroutine is combined with the main program at the source-language level. Then the entire source program is assembled into object language. In contrast, subprograms are converted to object language before they are combined with the main program. The object modules are combined by the linkage-editor program in a separate job step.

SUBPROGRAM LINKAGE

This topic presents three levels of sophistication in subprogram linkage and structure. The first level shows the basic aspects of linking to a subprogram, passing data to it, and the coding of the subprogram itself. The second level shows these basic aspects in a standard form that is widely used in the data processing industry. The third level shows how standard macros can be used to construct most of the standard linkage between the main program and the subprogram.

I'll use the pay-rate-table lookup routine of topic 1 to illustrate these three levels of subprogram structure and linkage. Each of these three versions of the pay-rate-lookup subprogram is based on the routine as it is shown in figure 9-7. In fact, the coding is identical except for the first few statements that make up the entry-point portion of the subprogram and the few statements that make up the exit coding.

Basic Linkage The *basic* subprogram *linkage* technique is illustrated in figure 9-12. This illustration uses the calling-sequence technique for data passing and the BALR instruction for linkage. Since the entry point of the subprogram (RATESUBP) is not defined in the calling program, it can't be used as an operand in a BAL instruction that links to the subprogram. Otherwise, an assembly error would occur. As a result, the entry point must appear as an address constant:

```
ADDRSUB  DC    A(RATESUBP)
```

This address constant must be preceded by an EXTRN instruction in which the entry-point label appears as the operand:

```
         EXTRN RATESUBP
```

The EXTRN instruction is a message to the assembler that the symbol appearing as an operand is external to this program. That is, it isn't defined within this program. The EXTRN causes the assembler to allow the address constant to be assembled as zeros (X'00000000'). It also causes the adcon to be put into a list for resolution by the linkage-editor program. This procedure, called *adcon resolution*, means that the linkage-editor program will fill the adcon area with the proper address as part of link-editing the main program and subprogram together before execution.

EXTRN instructions can have many symbols as operands. Also, you can code as many EXTRN statements as you need to in a single program. For instance, a program that calls five subprogram entry points might have these EXTRN statements:

```
         EXTRN EP1,EP2,EP3
         EXTRN SUBA,SUBB
```

Since EXTRN statements don't generate any object code and therefore occupy no storage, they can be placed anywhere in a program. However, they must precede the use of the symbols that they define as external ones.

The transfer of control to the subprogram is accomplished by loading the address constant into a register. Register 12 is used in figure 9-12. Then a branch-and-link-register instruction provides the actual link. Here are the instructions from figure 9-12:

```
L      12,ADDRSUB
CNOP   2,4
BALR   11,12
```

```
            .
            .
            .
            EXTRN RATESUBP
            .
            .
            .
            L     12,ADDRSUB
            CNOP  2,4
            BALR  11,12
            DC    A(CRDPAYCL)
            DC    A(PAYRATE)
            CP    PAYRATE,=P'999'
            BE    CLASSERR
            .
            .
            .
ADDRSUB     DC    A(RATESUBP)
            .
            .
            .
```

Calling Program

```
SUBPROG     START 0
            ENTRY RATESUBP
RATESUBP    BALR  10,0
            USING *,10
            STM   6,8,SAVE
            LM    6,7,0(11)
            LA    8,PAYTABLE
CMPCLASS    CLC   0(2,6),0(8)
            BE    PAYFOUND
            CLI   0(8),X'FF'
            LA    8,4(8)
            B     CMPCLASS
PAYFOUND    ZAP   0(2,7),2(8)
            B     RETURN
NOTFOUND    ZAP   0(2,7),P'999'
RETURN      LM    6,8,SAVE
            B     8(11)
SAVE        DS    3F
PAYTABLE    DS    0CL40
            DC    X'F0F1221C'
            .
            .
            .
            DC    X'FFFFFFFF'
            END
```

Subprogram

FIGURE 9-12 Basic Subprogram Linkage

To pass the data-field addresses, the calling program in figure 9-12 uses the calling-sequence technique. As a result, a CNOP instruction is required to align the two-byte BALR instruction at a halfword boundary so that no undefined bytes can exist between the BALR instruction and the following address constants. Since the CNOP operands are 2,4, the BALR instruction will be aligned two bytes before a fullword boundary and the adcon that follows will be on the fullword boundary. (See figure 9-9 for the CNOP operand summary.)

Since the subprogram in figure 9-12 is assembled separately from the main program, several points must be taken into consideration; these considerations are illustrated by the structure of the program in the figure. First, an ENTRY statement tells the assembler program to make the entry-point address available for resolution by the linkage editor. You can consider the ENTRY statement to be a counterpart to the EXTRN statement. The ENTRY statement also follows the same coding form as the EXTRN: both multiple symbols as operands and multiple ENTRY

```
          .
          .
          .
          LA     1,ADDRLIST
          L      15,ADDRSUB
          BALR   14,15
          CP     PAYRATE,=P'999'
          BE     CLASSERR
          .
          .
          .
ADDRSUB   DC     V(RATESUBP)
ADDRLIST  DC     A(CRDPAYCL)
          DC     A(PAYRATE)
          .
          .
          .
```

NOTES:
1 Register 1 points to address list.
2 Register 15 contains subprogram entry point.
3 Register 14 has return address.
4 Subprogram entry point is defined with a Vcon.

Calling Program

```
RATESUBP  START  0
          USING  *,15
          STM    6,8,SAVE
          LM     6,7,0(1)
          .
          .
          .
          .
          .
          .
RETURN    LM     6,8,SAVE
          BR     14
SAVE      DS     3F
PAYTABLE  DS     0CL40
          .
          .
          .
          END
```

NOTES:
1 Entry point is program name.
2 Register 15, loaded by calling program, is used as base register.

Subprogram

FIGURE 9-13 Standard Subprogram Linkage

statements in a program are valid.

Base-register loading and assignment is the second point that must be considered in the subprogram structure. Since subprograms are assembled separately, they are not addressed by the main program's base register. In the subprogram in figure 9-12, register 10 is loaded and assigned as the base register with the standard BALR and USING technique. The previous contents of register 10 are thus destroyed.

The rest of the pay-rate-lookup routine—the compare loop, the exit, and the table definition—is exactly the same as the subroutine version in figure 9-7. It is important to notice that the END statement for the subprogram is coded with no operand. This is true for all subprograms.

Standard Linkage Figure 9-13 illustrates *standard linkage*, the second level of subprogram linkage and structure. The figure illustrates a version of the basic linkage technique that is considered an informal standard within most IBM-supplied programs.

There are three important points to be noted in the calling routine in figure 9-13. First, in place of registers 11 and 12 which are used in the basic technique, registers 14 and 15 are used for linkage. Register 15 is loaded with the

```
            .
            .
            .
            LA      13,SAVE
            CALL    RATESUBP,(CRDPAYCL,PAYRATE)
            CP      PAYRATE,=P'999'
            BE      CLASSERR
            .
            .
            .
SAVE        DS      18F
            .
            .
            .
```

NOTES:
1 Register 13 contains address of 18-word save area.
2 CALL macro does four things:
   a. it constructs the adcon list and places list address in register 1;
   b. it loads register 15 with Vcon literal containing address of subprogram entry point;
   c. it loads register 14 with the return address; and
   d. it generates a BALR to the subprogram.

Calling Program

```
RATESUBP START 0
         SAVE  (14,12)
         BALR  3,0
         USING *,3
         ST    13,SUBSAVE+4
         LA    13,SUBSAVE
         .
         .
         .
RETURN   L     13,SUBSAVE+4
         RETURN (14,12)
SUBSAVE  DS    18F
PAYTABLE DS    0CL40
         .
         .
         .
         END
```

NOTES:
1 SAVE macro stores registers 14 through 12 in calling-program save area.
2 Standard base register assignment.
3 Register 13 stored in second word of subprogram save area.
4 Register 13 loaded with address of subprogram save area.
5 Register 13 reloaded with address of calling-program save area prior to return.
6 RETURN macro reloads registers 14 through 12 and returns to calling program.

Subprogram

FIGURE 9-14 Standard Subprogram Linkage with Macro Use

subprogram entry point and register 14 holds the return address. Second, register 1 holds the address of the data-address list that is passed to the subprogram. Third, a V-type address constant is coded for the subprogram entry point. The effect of a Vcon is equal to the combination of an A-type adcon and an EXTRN for the symbol. A four-byte address constant (initially filled with hex zeros) is defined, and the adcon is placed in the external symbol list for resolution by the linkage editor.

The subprogram in figure 9-13 is, of course, coded to be compatible with the calling routine. It is basically the same subprogram as that in figure 9-12, but the few statements that are part of the linkage to the calling program are changed to reflect the standard-linkage conventions. Register

1 is used in the load-multiple instruction to load registers 6 and 7 with the adcons and the BR instruction branches to register 14.

You should also notice that no ENTRY statement is used in the subprogram in figure 9-13. The ENTRY statement can be omitted because the subprogram entry point, RATESUBP, appears as the program name on the START instruction. Since a program name is automatically considered an external symbol by the assembler, its address is available for address-constant resolution during link-editing. If a subprogram has multiple entry points, ENTRY statements can be used along with the name on the START instruction.

The assignment of register 15 as the base register in the subprogram is a fairly common technique. It avoids the possibility of destroying the contents of some other register that may be used by the calling program. This is especially important when the calling program is written in a higher level language such as COBOL or FORTRAN in which the programmer has no control over the use of registers. A BALR instruction to load register 15 isn't necessary because the calling routine has already loaded the subprogram start address into it.

Standard Macro Linkage The third technique for subprogram linkage, (*standard macro linkage*) uses macro instructions. This technique is illustrated in figure 9-14. Here, the calling routine uses a CALL macro to generate instructions that form a link to the subprogram:

```
CALL  RATESUBP,(CRDPAYCL,PAYRATE)
```

The code generated by this CALL macro is much like the standard linkage code used in figure 9-13. The data names that appear in the second operand of the CALL macro (enclosed in parentheses) are used to construct an address list. The address of this list is then placed in register 1. The first operand of the CALL macro is the subprogram-entry-point name, which is used in the definition of a V-type address constant whose address is loaded into register 15. Register 14 is loaded with the proper return address. Finally, a BALR instruction is generated to perform the actual branch to the subprogram.

One additional aspect of this more sophisticated subprogram linkage is that register 13 is loaded with the address of an 18-word save area. This area is referred to by the subprogram and is used to save the contents of all the registers. Of course, the registers are reloaded from this area just before the subprogram branches back to the calling program. This save/restore technique protects the calling program's use of the registers and allows the subprogram to use any or all of the registers.

In the subprogram in figure 9-14, RATESUBP is again used as the program name, so no ENTRY instruction is needed. The first executable code in this third version of the subprogram is generated by the SAVE macro:

```
SAVE  (14,12)
```

This macro generates a store-multiple instruction that saves the contents of the register group specified within parentheses. It stores these registers in the 18-word save area of the calling program. (The address of this area must have been placed in register 13 by the calling program.) In figure 9-14 registers 14 through 12 are stored.

The 18-word save area has a fixed format which is illustrated in figure 9-15. When the SAVE macro is assembled, it generates code so the registers will always be stored in the same locations within the save area. For instance, if register 3 has been included in the range specified in the SAVE-macro operand, it is always stored in the ninth word. Although the programmer can use any register range in the SAVE macro, the range 14 through 12 is the most common way to code the macro.

Once the SAVE macro has been executed, the subprogram can use any register except 13 without disrupting the calling program operations after its return. The subprogram in figure 9-14 uses a normal BALR/USING combination to load and assign register 3 as the base register for the subprogram. The two instructions that follow the USING statement prepare this subprogram so it can CALL other subprograms if required. First, the address of the calling program's save area (register 13) is stored in the

WORD	DISPLACEMENT	CONTENTS
1	0	used by PL/I
2	4	address of previous save area (calling program)
3	8	address of next save area
4	12	contents of register 14
5	16	contents of register 15
6	20	contents of register 0
7	24	contents of register 1
8	28	contents of register 2
9	32	contents of register 3
10	36	contents of register 4
11	40	contents of register 5
12	44	contents of register 6
13	48	contents of register 7
14	52	contents of register 8
15	56	contents of register 9
16	60	contents of register 10
17	64	contents of register 11
18	68	contents of register 12

FIGURE 9-15 Assignments for Standard Save Area

subprogram's own 18-word save area, SUBSAVE. Register 13 is then loaded with the address of SUBSAVE. When this has been done, the subprogram can link to another subprogram in which a SAVE macro is also coded. This "chaining" of save areas is explained in more detail later in this topic.

The pay-rate-lookup function would appear in the subprogram at this point. Since all the registers have already been saved, registers 6 and 7 can be loaded immediately with the address constants pointed to by register 1. Similarly, they don't have to be reloaded at the end of the subprogram. The compare loop, table definition, and other parts of the lookup routine are again unchanged from previous versions.

Before returning to the calling program, the subprogram must restore the original contents of the registers. First, the address of the calling-program save area is reloaded into register 13 from the second word of the subprogram's own save area. A RETURN macro is then coded to reload registers 14 through 12 and generate the branch back to the calling program. The operand of any RETURN macro should be identical to the operand of the SAVE macro with which it is paired so the proper registers are restored. (As a sidelight, you can see that the RETURN macro has a six-character operation code. To use it, you must leave at least one blank between the operation code and the operand.)

NESTED SUBPROGRAMS

Subprograms are often nested in the same way that subroutines are nested. Figure 9-16 shows a main program that calls a first subprogram, which then calls a second subprogram. If you follow the flow from the main program CALL to the first subprogram, you can see that the coding in this subprogram is the same as that illustrated in figure 9-14. Notice that after the address of SAVEA has been stored, register 13 is loaded with the address of SAVEB. Then, when the first subprogram calls the second subprogram, the logic is repeated for SAVEB and SAVEC. Because the relationship between the calling program and the called program is the same at any level, this nesting of subprograms can extend to any number of levels.

The saving and restoring of the registers when using the SAVE and RETURN macros offers a special debugging benefit. Because the second word of each program's save area contains the address of the calling program's save area, a chain through each level of the structure exists. As a result, if the program has a failure and dumps, the storage dump can be used to trace through each save area and figure out the path that was followed just before the failure.

Main Program

```
          .
          .
          .
          LA     13,SAVEA
          .
          .
          CALL   SUB1,(.....)
          .
          .
          .
          .
          .
          .
          .
SAVEA     DS     18F
```

word 1: not used
word 2: not used
word 3: not used
word 4: register 14, return address
word 5: register 15, SUB1 address
words 6–18: registers 0–12

First Subprogram

```
SUB1      SAVE   (14,12)
          BALR   ......
          USING  ......
          ST     13,SAVEB+4
          LA     13,SAVEB
          .
          .
          .
          CALL   SUB2,(.....)
          .
          .
          .
          L      13,SAVEB+4
          RETURN (14,12)
SAVEB     18F
```

word 1: not used
word 2: SAVEA address
word 3: not used
word 4: register 14, return address
word 5: register 15, SUB2 address
words 6–18: registers 0–12

Second Subprogram

```
SUB2      SAVE   (14,12)
          BALR   .....
          USING  .....
          ST     13,SAVEC+4
          LA     13,SAVEC
          .
          .
          .
          .
          .
          .
          .
          L      13,SAVEC+4
          RETURN (14,12)
          .
SAVEC     DS     18F
```

word 1: not used
word 2: SAVEB address
words 3–18: not used

FIGURE 9-16 Nested Subprogram Linkage

TESTING SUBPROGRAMS

To test a subprogram when the main program has been completed, it is common to assemble, link-edit, and test both modules in the same job. To do this, the following job-control-card setup can be used:

```
// JOB TEST
// OPTION LINK,DUMP
// EXEC ASSEMBLY
   (main program source deck)
/*
// EXEC ASSEMBLY
   (subprogram source deck)
/*
// EXEC LNKEDT
// EXEC
   (data deck, if any)
/*
/&
```

Here, the first two job steps assemble the main program and subprogram into object modules. Then, the linkage editor resolves the address constants and combines the modules so they can be executed as one program. Finally, the link-edited modules are executed.

If the main program isn't complete when the subprogram is ready to be tested, it is sometimes necessary to write a simple main program that can be used to test the subprogram. Later on, when the main program is complete, a more complete test run can be made.

In some cases, the main program has been tested and stored in the *relocatable library* of the systems-residence device. The subprogram is then tested using job-control cards like this:

```
// JOB TEST
// OPTION LINK,DUMP
   INCLUDE program-name
// EXEC ASSEMBLY
   (subprogram source deck)
/*
// EXEC LNKEDT
// EXEC
   (test data, if any)
/*
/&
```

In this case, the name of the main module is specified in the INCLUDE card. Then, the main object module from the relocatable library is link-edited with the subprogram object module just assembled.

DISCUSSION

Although this topic describes assembler-language calling programs and subprograms, remember that a high-level language can also be used for a calling program or subprogram. If a high-level language is used, the standard-linkage conventions are used: register 1 is used to point to the adcon list, register 15 contains the subprogram entry point, and register 14 contains the return address. If, for example, the rate-lookup subprogram in figure 9-13 was called by a COBOL main program, this COBOL statement would provide the same linkage as the assembler-language calling program:

```
CALL 'RATESUBP' USING CARD-PAY-CLASS, PAY-RATE
```

The first adcon pointed to by register 1 would then contain the address of the CARD-PAY-CLASS field; the second adcon would contain the address of the PAY-RATE field. Similarly, the address of RATESUBP would be loaded into register 15 by this statement, and the address of the next COBOL statement would be put in register 14. Finally, an 18-word save area is defined in the calling program and its address is placed in register 13. Thus, you can see that the assembler-

language subprogram will work regardless of the language used for the calling program.

Another point to note is that there are many cases in which a subprogram will have multiple entry and exit points. An assembler-language I/O subprogram, for example, might have three or more entry points. The first entry point will be used for opening the file, the second for reading and writing records in the file, and the third for closing the file. For each entry point, there will be a corresponding exit point.

Finally, you may want to consider what happens when two object modules are link-edited. Because base registers are used for addressing, only the address constants represent absolute addresses. This means that an object module can be loaded into any storage positions, and that all operand addresses will address the appropriate fields or instructions. That's one major reason for using base-plus-displacement addressing—the object modules are easily relocated. During link-editing, then, the primary job is to place the proper addresses in the adcons of a program (resolve them) after assigning storage locations to each object module to be linked.

Terminology

object module
basic linkage
adcon resolution
standard linkage
standard macro linkage
relocatable library

Objectives

1 Given the description of a subprogram and a programming problem, write a BAL calling program that uses the subprogram.

2 Given the description of a subprogram and its calling program, write the subprogram in assembler language.

Problems

1 (Objective 1) Assume that the gross-pay-calculation module described in problem 1 of topic 1 is to be a subprogram rather than a subroutine. Code the appropriate linkage for the calling program assuming the subprogram name is GROSCALC. Use the standard macro technique.
2 (Objective 2) Convert the gross pay subroutine of problem 2 in topic 1 to a subprogram using the standard macro technique.

Solutions

1
```
         LA    13,MAINSAVE
         CALL  GROSCALC,(PAYRATE,HRSWRKED,
               GROSSPAY)
```

2 The following is an acceptable solution:

```
GROSCALC SAVE  (14,12)
         BALR  3,0
         USING *,3
         LM    5,7,0(1)
         ZAP   WRKPAY,0(3,5)
         MP    WRKPAY,0(3,6)
         AP    WRKPAY,RNDFCTR
         MVN   WRKPAY+4(1),WRKPAY+5
         ZAP   0(3,7),WRKPAY(5)
         RETURN (14,12)
WRKPAY   DS    PL6
RNDFCTR  DC    P'50'
```

10

Writing Macro Definitions

The use of standard macros is a basic part of assembler-language programming. For instance, GET and PUT macros, in combination with DTF macros, are used to perform most I/O operations. In addition, standard macros such as SAVE and RETURN for subprogram linkage and COMRG for supervisor communication provide special processing capabilities.

During assembly, each macro statement you have coded is replaced by the statements that make up the *macro definition*. This is referred to as *macro expansion*. For example, the GET macro is expanded into two load instructions and a branch-and-link instruction that branches to the I/O module.

Usually, the macro definition is stored in the DOS *source-statement library*, which is one of the three libraries found on the system-residence device. Each time the assembler finds a macro statement in the BAL source program, it looks up the macro definition in the source-statement library. It then inserts

MACRO DEFINITION:

```
Header Statement              MACRO
Prototype Statement  &LABEL   SUMWDS &R1,&W1,&W2,&W3,&CON
Model Statements     &LABEL   SR   &R1,&R1
                              A    &R1,&W1
                              A    &R1,&W2
                              A    &R1,&W3
                              S    &R1,&CON
                              A    &R1,=F'50'
Trailer Statement             MEND
```

SAMPLE MACRO STATEMENT:

```
                     ROUT1    SUMWDS 5,WORDA,WORDB,WORDC,BCON
```

MACRO EXPANSION:

```
                     ROUT1    SR   5,5
                              A    5,WORDA
                              A    5,WORDB
                              A    5,WORDC
                              S    5,BCON
                              A    5,=F'50'
```

FIGURE 10-1 The SUMWDS Macro Definition

the statements from the macro definition into the assembler source program immediately following the macro statement.

As an assembler-language programmer, you can write macro definitions of your own. Why would you want to? For the same reason that the standard macros were written: to provide an easy way to code a frequently used series of assembler-language statements. Once it has been written and stored in the source-statement library, you can use your own macro statement just as you use one of the standard macros.

One reason some programmers don't write macro definitions is the difficulty involved in writing them. In truth, writing a macro like the DTFCD (which consists of 429 macro-definition statements) can be an extremely complex task. On the other hand, writing simple macros *is* a manageable task, and the resulting macros can improve the efficiency of a company's programming efforts. In topic 1 of this chapter the more straightforward forms of macro writing are presented. These techniques are relevant to the needs of the business programmer. In topic 2, advanced macro-writing techniques are presented—techniques that are more relevant to the needs of the software specialist. Because of the complexity of the subject, you are probably better off if you study this chapter only after you have become quite proficient in BAL coding.

TOPIC ONE Basic Macro Writing Figure 10-1 presents a simple macro definition. This macro adds three binary fields to a register, subtracts one field from it, and adds the literal value 50 to it. The figure shows the macro definition, an example of its use in a macro statement, and the expansion of the sample statement.

There are four parts in every macro definition. The *header statement* always has MACRO as the operation code; the *trailer statement* always has MEND as the operation code. Following the header statement is the *prototype statement* that gives the form in which the macro statement must be written. After the prototype statement are the *model statements* that indicate the code that is to be inserted into the assembler-language source program when the macro statement is processed.

Depending on the function of the macro, the prototype statement can consist of just a macro operation code or it can consist of a macro operation code plus one or more *variable symbols*. A variable symbol is made up by the & sign followed by from one to seven letters or numbers, the first of which must be a letter. Thus, in figure 10-1, &LABEL, &R1, and so on, are variable symbols. The macro operation code can be from one to eight letters or numbers in length starting with a letter, but it cannot duplicate a BAL operation code or another macro name.

The variable symbols in the prototype statement indicate which operands can be used when coding the macro statement. Thus, the SUMWDS macro has places for five operands and, in this case, requires all five. The SUMWDS macro also provides for a label, &LABEL, which isn't required,

but which can be used if the program must branch to the SUMWDS macro statement.

The model statements represent the instructions to be used in the expansion of the macro. During expansion, the operands given in the macro statement are substituted for the variable symbols that have the same position in the prototype statement as the operands do in the macro statement. Thus, 5 is substituted for &R1 wherever &R1 appears in the model statements. Similarly, WORDC is substituted for &W3 wherever it appears in the model statements. The expanded instructions are then placed in the source program and the assembly continues.

When operands are coded in the prototype statement as in figure 10-1, they are referred to as *positional operands*. This means that the position of each operand indicates its use in the macro expansion. If, for example, the SUMWDS macro statement were coded

```
SUMWDS WORDA,5,WORDB,WORDC,BCON
```

WORDA would be substituted for &R1, which would result in faulty source code.

If a positional operand is not to be used for a macro, it is omitted when the macro statement is written. If, for example, operands (in this case, also called *parameters*) 2 and 5 in the SUMWDS operand were to be omitted, you would code the macro statement as:

```
SUMWDS 5,,WORDB,WORDC
```

Because all of the SUMWDS operands are required, however, this statement would result in faulty code; the second model statement, for instance, would be expanded to

```
A       5,
```

which couldn't be assembled into object code.

The other way of specifying operands is to use *keyword operands*. These are used in the DTF statements. If, for example, SUMWDS had been written using keyword operands, it would be written as shown in figure 10-2. Here, the parameters in the prototype statement, minus the leading & sign, are the keywords that must be used in the

Macro definition:

```
        MACRO
&NAME   SUMWDS &REG1=,&WORD1=,&WORD2=,&WORD3=,&CON=
        SR      &REG1,&REG1
        A       &REG1,&WORD1
        A       &REG1,&WORD2
        A       &REG1,&WORD3
        S       &REG1,&CON
        A       &REG1,=F'50'
        MEND
```

Sample macro statement:

```
        SUMWDS WORD1=W1,WORD2=W2,WORD3=W3,REG1=7,CON=C1
```

Macro expansion:

```
        SR      7,7
        A       7,W1
        A       7,W2
        A       7,W3
        S       7,C1
        A       7,=F'50'
```

FIGURE 10-2 SUMWDS Macro Using Keyword Operands

macro statement. Unlike positional operands, keyword operands can be coded in any sequence.

If a keyword operand is to have a default value, the value is coded in the prototype statement after the equals sign. If, for example, the keyword &REG1 had a default value of 5 in the SUMWDS macro, it would be coded as &REG1=5 in the prototype statement. Then, should this parameter be omitted, register 5 will be substituted in the body statements. If a keyword operand is omitted but doesn't have a default value, no value is assigned to that parameter in the macro expansion.

It is legal, though rare, to define a combination of positional and keyword operands. In such a case, the positional operands must be defined first in the prototype statement and coded first in the program macro statement. Here, then, is a hypothetical example of a prototype

```
         MACRO
         ITMSTR
ITMRCD   DS    0CL70     INV MSTR RCD LAYOUT
IITEM    DS    CL8       ITEM NUMBER
IDESC    DS    CL20      ITEM DESCRIPTION
IUM      DS    CL4       UNIT OF MEASURE
IOPOL    DS    CL2       ORDER POLICY CODE
IOQTY    DS    PL4       ORDER QTY
IOPNT    DS    PL4       ORDER POINT
ISS      DS    PL4       SAFETY STOCK
IBOH     DS    PL4       BALANCE ON HAND
IOOQTY   DS    PL4       ON ORDER QTY
IALLOC   DS    PL4       ALLOCATED QTY
         DS    CL12      AVAIL FOR EXPANSION
         MEND
```

FIGURE 10-3 Simple ITMSTR Macro Definition

statement that defines both positional and keyword operands:

```
&NAME    SMPLE &P1,&P2,&P3,&KEY1=,&KEY2=NO
```

When coding the SMPLE macro, the first three operands must be treated as positional operands, the last two are treated as keywords.

Although the SUMWDS macro uses variable symbols in the model statements, *ordinary symbols* can also be used. Ordinary symbols are those you normally code in a program. For instance, the model statement

```
MVC   FLD1,FLD2
```

will be generated just as it appears whenever the macro statement is used. The ordinary symbols can be defined within the model statements or in the BAL program itself.

Figure 10-3 gives an example of a macro that uses only ordinary symbols. It indicates an easy way to code the record layout of an inventory master record that is used in several programs. To get the record definitions inserted into the program, the programmer codes:

```
ITMSTR
```

Since the macro expansion defines the master fields, the macro can be coded only once in a program; otherwise, the fields would be defined more than once and would result in diagnostics.

The symbols used for labels and operands in model statements can also be combinations of ordinary and variable symbols. If a MVC instruction in the macro definition were coded

```
MVC   FLD&A,FLD&B
```

the operands of the generated MVC instruction would be the characters FLD plus whatever values were assigned to &A and &B. For instance, if &A = 1 and &B = 2, the generated instruction would be:

```
MVC   FLD1,FLD2
```

Ordinary and variable symbols can also be combined in reverse. For example,

```
MVC   &A.FLD,&B.FLD
```

will generate

```
MVC   OUTFLD,INFLD
```

if &A = OUT and &B = IN. In this case, the period (.) in &A.FLD and &B.FLD is called a *concatenation character.* It is used to separate a variable symbol from an ordinary symbol. This is necessary so the assembler program doesn't look for a variable symbol &AFLD. There are many ways in which variable and ordinary symbols can be combined in the model statements; some of them are indicated by the examples in figure 10-4. Note that whenever the assembler might be confused by two operand parts in succession, a concatenation character is used to separate the parts.

Figure 10-5 is a version of the ITMSTR macro that illustrates symbol combinations. This time the prototype statement shows that a label may be coded and that a

single operand is expected. You can see that the label of the macro statement in the source program will be assigned to the first model statement through the variable symbol &LABEL and that the operand is used as a prefix for the field labels. If a programmer coded the macro

```
MSTRWORK ITMSTR IM
```

it would be expanded to:

```
MSTRWORK DS    0CL70    INV MSTR RCD LAYOUT
IMITEM   DS    CL8      ITEM NUMBER
IMDESC   DS    CL20     ITEM DESCRIPTION
IMUM     DS    CL4      UNIT OF MEASURE
IMOPOL   DS    CL2      ORDER POLICY CODE
IMOQTY   DS    PL4      ORDER QTY
IMOPNT   DS    PL4      ORDER POINT
IMSS     DS    PL4      SAFETY STOCK
IMBOH    DS    PL4      BALANCE ON HAND
IMOOQTY  DS    PL4      ON ORDER QTY
IMALLOC  DS    PL4      ALLOCATED QTY
         DS    CL12     AVAIL FOR EXPANSION
```

Notice that the operand for the macro should always start with a letter and should not be more than three characters long, otherwise invalid labels for the fields will result. (Topic 2 shows how conditional assembly instructions can be used to check a macro statement for valid operands during expansion.)

symbol coded in model statement	values assigned to variable symbols	generated symbol
&FLD.A	&FLD=SUM	SUMA
FIELD&A	&A=1	FIELD1
NAME.&Z	&Z=ZZZ	NAMEZZZ
&D1.X.&L1	&D1=B4,&L1=32	B4X32
&DISP.(&BASE)	&DISP=84 &BASE=9	84(9)
&F1+5*&F2	&F1=6,&F2=FACT	6+5*FACT

FIGURE 10-4 Model-Statement Symbol Combinations

```
         MACRO
&LABEL   ITMSTR &PF
&LABEL   DS    0CL70    INV MSTR RCD LAYOUT
&PF.ITEM DS    CL8      ITEM NUMBER
&PF.DESC DS    CL20     ITEM DESCRIPTION
&PF.UM   DS    CL4      UNIT OF MEASURE
&PF.OPOL DS    CL2      ORDER POLICY CODE
&PF.OQTY DS    PL4      ORDER QTY
&PF.OPNT DS    PL4      ORDER POINT
&PF.SS   DS    PL4      SAFETY STOCK
&PF.BOH  DS    PL4      BALANCE ON HAND
&PF.OOQTY DS   PL4      ON ORDER QTY
&PF.ALLOC DS   PL4      ALLOCATED QTY
         DS    CL12     AVAIL FOR EXPANSION
         MEND
```

FIGURE 10-5 ITMSTR Macro with Combined Symbols

TWO USEFUL MACROS

With this as background, you should be able to write some useful macros. Remember that you can use any assembler instructions in the model statements with literals, ordinary symbols, variable symbols, or combined symbols as operands. Although you may use keyword parameters (operands) in the prototype statement, positional parameters are more commonly used for simple macros.

Since getting the date from the supervisor is common to many types of programs, the GETDATE macro in figure 10-6 is one that can be useful. This macro illustrates the use of a macro (COMRG) within a macro definition. The COMRG macro, which is described in chapter 11, places the address of the current date in the supervisor-communication region in register 1. As a result, the GETDATE macro moves the date into the operand named in the GETDATE macro. As the comment in the model statements indicates, this field should be at least eight bytes long (the date is in this form: MM/DD/YY). Once this macro has been stored in the

```
         MACRO
&LABEL   GETDATE &FLD
* THE OPERAND FIELD SHOULD BE AT LEAST 8 BYTES
&LABEL   COMRG
         MVC   &FLD.(8),0(1)
         MEND
```

FIGURE 10-6 A GETDATE Macro

```
         MACRO
&LABEL   FRSTSW &BRCH
&LABEL   BC    0,&BRCH
         MVZ   *-3,X'F0'
         MEND
```

FIGURE 10-7 First-Time-Switch Macro

source-statement library, a programmer can store the date in a field named DATEFLD by using this statement:

```
GETDATE DATEFLD
```

In this case, the macro expansion will generate the following move statement:

```
MVC   DATEFLD(8),0(1)
```

When a macro is used within a macro, the inner macro doesn't appear as one of the generated source statements on the source listing. As a result, the COMRG macro will not be shown on the source listing when the GETDATE macro is expanded, but its generated statements will.

Another useful macro I have seen created in several forms is a first-time-switch macro. The purpose of this type of macro is to let a series of statements be executed the first time through a routine, but to branch around those statements on subsequent passes through the routine. Figure 10-7 shows one version of the first-time-switch macro definition.

To use this macro, the programmer gives the label of the instruction to be branched to after the first time through the program. For instance, he might code the following in order to clear a print area the first time through a program and leave it untouched on successive uses of the print routine:

```
PRINT    FRSTSW OTHER
         MVI   PRTAREA,X'40'
         MVC   PRTAREA+1(132),PRTAREA
OTHER    .
         .
         .
```

The resulting source statements will then look like this:

```
   PRINT    FRSTSW OTHER
+  PRINT    BC    0,OTHER
+           MVZ   *-3,X'F0'
            MVI   PRTAREA,X'40'
            MVC   PRTAREA+1(132),PRTAREA
   OTHER    .
            .
            .
```

Because the branch-on-condition (BC) instruction has a mask of zero, the branch to OTHER won't take place the first time through the program. (Remember from chapter 2 that a mask of hex 0 means "never branch"; a mask of hex F means "always branch.") However, the MVZ instruction following the branch modifies its mask. Since the BC instruction is four bytes long, *−3 in the MVZ instruction refers to the second byte of the branch instruction. (Remember that * indicates the present location counter value.) Then, when the MVZ is executed, the mask in the BC instruction is changed from hex zero to hex F, and the second time through the routine, the BC instruction will branch to OTHER.

MAKING THE MACRO DEFINITIONS AVAILABLE

Once you have defined a macro, you must make the definition available to your programs in order to test it. The

normal way of doing this is to put the macro definitions at the beginning of a source deck that uses the macros. All macro definitions must precede the first source card (the START instruction) of the program to be assembled. The macro can then be used in the program.

After you have successfully tested a macro definition, you will probably want to put it in the source-statement library. To do this, you use one of the programs supplied with the Disk Operating System, a library-maintenance program named MAINT. Figure 10-8 shows the job-control cards required to add a macro definition to the source-statement library. The macro definition goes between the BKEND cards; the macro name goes in both the CATALS card and the first BKEND card. Additional information on the use of this maintenance program is given in chapter 16.

```
// JOB jobname
// EXEC MAINT
   CATALS A.macro
   BKEND A.macro
         MACRO
         .
         .
         .
         .
         .
         .
         MEND
   BKEND
/*
/&
```

FIGURE 10-8 Job-control Statements for Adding Macro to Source-Statement Library

Terminology

macro definition
macro expansion
source-statement library
header statement
trailer statement
prototype statement
model statement
variable symbol
positional operand
parameter
keyword operand
ordinary symbol
concatenation character

Objective

Design and code macro definitions using the macro-writing elements presented in this topic.

Problems

1 Code a macro definition, named LCNT, that allows a programmer to have the following line-count constants generated in his program by coding only one macro statement:

```
PCON0=P'0'
PCON1=P'1'
PCON50=P'50'
LCNT=P'50'
```

The macro should have one operand that can be used to assign a two-character prefix to the labels of the constants. If the operand is omitted, the labels given above should be generated.

2 Code a macro definition named LCNTRT that generates statements to perform a line-count check. This is to be coordinated with the line-count-constant macro written in problem 1 so use the fields from problem 1 in the model statements. The macro is to use positional operands with the first operand representing the two-character-constant prefix (if any), the second representing the label to be branched to if the LCNT field is less than 50, and the third representing the label to be branched to if the LCNT field is 50 or greater. If the LCNT field is 50 or greater, the macro should reset the LCNT field before branching.

Solutions

1 The following is an acceptable solution:

```
         MACRO
         LCNT  &P
&P.PCON0 DC    P'0'
&P.PCON1 DC    P'1'
&P.PCON50 DC   P'50'
&P.LCNT  DC    P'50'
         MEND
```

2 The following is an acceptable solution:

```
         MACRO
&LABEL   LCNTRT &PRE,&PRTDET,&PRTHDG
&LABEL   CP    &PRE.LCNT,&PRE.PCON50
         BL    &PRTDET
         ZAP   &PRE.LCNT,&PRE.PCON0
         B     &PRTHDG
         MEND
```

TOPIC TWO Advanced Macro Writing The macros in topic 1 accomplish two types of macro expansion. The first, called *text insertion*, simply inserts the model statements into the source program. For example, the first ITMSTR macro and the standard COMRG macro accomplish text insertion only.

The second level of macro expansion is called *text insertion with modification*. The macros with operands in topic 1 are examples of this level since they cause the model statements to be modified based on the operands given. Nevertheless, these macros still involve a fixed series of model statements that are to be inserted in the program.

The highest level of macro writing involves *text manipulation*. This means that the operands of the macro determine which statements are inserted into the source program as well as the form those statements are to take. A GET macro, for example, can be coded with filename as its only operand or it can have a work-area name as its second operand. When the macro is expanded, the statements generated vary depending on whether the work-area operand is present.

In order to write text-manipulation macros, you need to know how to define and use *SET symbols*. These symbols, which are used in macro definitions, can have their values changed during macro expansion. In addition to this, you need to know how to write *conditional assembly instructions*—assembly instructions that can alter the sequence in which the assembler expands a macro. Such instructions allow the assembler to loop through the model statements in a macro definition, generating several source statements from a single model statement.

Quite frankly, writing macro definitions for text manipulation is a skill that cannot be mastered by all programmers. On the programming staff of a large company, for instance, perhaps only one programmer will be capable of writing an I/O macro for a specialized I/O function, perhaps none will be able to do it. On the other hand, the macro-writing facilities of assembler language do represent a language in itself and they do allow a programmer to create a macro language within Basic Assembler Language. Because of this, macro writing is of interest to the computer scientist or software specialist.

Because SET symbols and conditional assembly instructions work together, the information relating to both of them must be covered before an example of a text-manipulation macro can be presented. Therefore, this topic presents the macro-writing facilities in this sequence: (1) SET symbols, (2) symbol attributes, (3) assigning values to SET symbols, and (4) conditional assembly instructions. Following presentation of the macro-writing elements, several text-manipulation macros are illustrated.

SET SYMBOLS

There are three types of SET symbols. They are called arithmetic (A), binary (B), and character (C) types. An arithmetic SET symbol can be assigned any numeric value between -2^{31} and $+2^{31}-1$. This is the same range that a binary fullword has. In contrast, a binary SET symbol can be assigned only two values: 0 and 1. A character SET symbol can be assigned a string of up to eight characters.

A SET symbol is *local* or *global* depending on its range

within a program. If the value assigned to the SET symbol is effective within only one macro expansion, it is called local. In this case, the SET symbol can have different values assigned to it if the macro is used elsewhere in the program. In contrast, a global SET symbol can be defined only once in a program. It can be used in other macros or in multiple appearances of the same macro, but it cannot be defined more than once. The global SET symbol, then, is common to an entire assembly and is available for use by other macros. In most cases, however, you will use local SET symbols.

All SET symbols used in a macro definition must be declared (defined) immediately following the prototype statement—globals first, then locals. There are six operation codes for *symbol declaration*; they are shown in these examples:

```
GBLA   &NBR1
GBLB   &B1,&B2,&SWTCH
GBLC   &STR,&X37
LCLA   &VAR1,&TIP
LCLB   &OFF
LCLC   &NAME,&FIELD
```

The label field in a symbol declaration must be blank, and the operation code is GBL (global) or LCL (local) with the type code of the SET symbols being defined completing the operation code. Since SET symbols are a type of macro variable symbol, the first character of each SET symbol name must be &. Notice that multiple SET symbols can be declared in one statement by separating the names with commas.

When a SET symbol is declared, it is assigned an initial value of zero for A and B types and null for C types. A null value means literally nothing—zero length and no data. As an example of SET symbol declarations, here is the prototype statement from the standard CALL macro, followed by its declarations:

```
&NAME    CALL   &P1,&P2
         LCLA   &N
         LCLA   &KAP1,&KAP2
```

&N, &KAP1, and &KAP2 are all assigned an initial value of zero when the macro is expanded.

Type codes for symbols defined in DS and DC statements:	
A	A-TYPE ADDRESS CONSTANT
B	BINARY
C	CHARACTER
D	LONG FLOATING-POINT
E	SHORT FLOATING-POINT
F	FULLWORD FIXED-POINT
H	HALFWORD FIXED-POINT
P	PACKED DECIMAL
V	V-TYPE ADDRESS CONSTANT
Z	ZONED DECIMAL
Type codes for symbols defined as instruction labels:	
I	MACHINE INSTRUCTION
M	MACRO INSTRUCTION
Type codes for symbols defined as macro operands:	
N	A SELF-DEFINING TERM (NUMBER)
O	AN OMITTED TERM

FIGURE 10-9 Type Codes for Symbols

SYMBOL ATTRIBUTES

All symbols—ordinary, variable, and SET symbols—have attributes. Two attributes you are familiar with are *length* and *type*. For instance, an ordinary symbol defined as PL3 has a length attribute of 3 and a type code of P. In addition to the normal type codes used in DS and DC statements, there are others to indicate that a symbol is the label of an instruction or the name of a macro operand. Figure 10-9 summarizes the common type codes for symbols. If, for example, a macro operand is omitted, its type attribute is O. If it is a number, its type attribute is N (a number is a self-defining value).

A third attribute, *number*, applies only to symbolic parameters that have *sublists*. The term sublist refers to the fact that one symbolic parameter can have several values. The parameter must then have a *subscript* added to identify an individual value in the sublist. The subscript is a number in parentheses following the symbol name. For instance, &P(2) would refer to the second item in the sublist for the symbol &P; &P(5) would refer to the fifth item in the sublist.

For an operand with a sublist, the prototype statement can indicate the maximum number of values within the sublist in the following way:

```
&LABEL    SAMP  &P1,&P2(5)
```

Here, the prototype statement indicates that the symbolic parameter &P2 may have five sublist values. It isn't necessary to indicate the maximum number of values for a sublist parameter in a prototype statement, but it can make the macro definition easier to understand.

When a macro with one or more sublist parameters is coded in a program, the sublist operands must be separated by commas and enclosed in parentheses. For example, when the SAMP macro shown above is coded, the individual values might be written as:

```
          SAMP  SUM,(X,Y,,AZ)
```

Parameter &P2, then, will have these values assigned to it:

PARAMETER	VALUE
&P2(1)	X
&P2(2)	Y
&P2(3)	undefined
&P2(4)	AZ

Since the third name in the sublist has been omitted, it is considered to be undefined.

At any rate, the number attribute of a symbol is equal to the number of sublist positions indicated in the macro statement—that is, one more than the number of commas in the parentheses. For &P2 in the macro statement above, the number attribute is 4 because three commas are used. When the sublist operand is omitted, the number attribute is 0. For most parameters (those with no sublist), the number attribute is 1.

A fourth attribute that is useful is *count*. It is equal to the number of characters in the operand of a macro statement. To illustrate, suppose a prototype statement is coded as:

```
&LABEL    GSPLX &OP1,&OP2
```

Then, if the macro statement is coded as

```
PICT      GSPLX SCHMALTZ,X
```

the count attribute of &LABEL is 4. Similarly, the count attribute of &OP1 is 8 and of &OP2 is 1. When the macro is coded with different operands, the count attributes are likely to be different.

In addition to these four attributes, two others exist: scaling and integer attributes. Since they are rarely used, they aren't covered in this book.

ASSIGNING VALUES TO SET SYMBOLS

Values are assigned to SET symbols during macro expansion using the SETA (arithmetic), SETB (binary), and SETC (character) statements.

SETA The label of the SETA statement must be the variable symbol to which a value is to be assigned. The operand of the statement is an *arithmetic expression* that represents the value to be assigned. For instance, the following SETA statement assigns a value of 46 to the SET symbol named &A1:

```
&A1       SETA  46
```

In this case, the arithmetic expression is a *self-defining term*, the number 46.

The arithmetic expression can range from a self-defining term to a complex expression involving many variables and the *arithmetic operators*: + for plus, − for minus, * for multiply, and / for divide. Here's a more involved arithmetic expression used as the operand of a SETA statement:

```
&DELTA   SETA  &X1+10/&X2
```

The evaluation of the expression proceeds from left to right with multiplication and division done first, followed by addition and subtraction. If, for example, &X1 has a value of 10 and &X2 has a value of two, &DELTA will be assigned a value of 15.

If parentheses are used in an arithmetic expression, the expressions within the innermost sets of parentheses are evaluated first. If the statement above is coded

```
&DELTA   SETA  (&X1+10)/&X2
```

&DELTA is assigned a value of 10 if &X1 equals 10 and &X2 equals 2. Within parentheses, evaluation proceeds as before, with multiplication and division first.

An attribute can also be assigned to a SET symbol by using the SETA statement, provided the attribute is numeric. For example, the following code assigns a value equal to the length attribute of &A1 to &LA1:

```
&LA1     SETA  L'&A1
```

If the length of the operand that is substituted for &A1 is eight, &LA1 is assigned a value of eight. To indicate an attribute, L' (length), T' (type), K' (count), or N' (number) is used. Since L', N', and K' are numeric, they can be used in SETA expressions.

SETB The SETB statement can assign a value to a binary SET symbol in much the same way as a value is assigned to a SETA statement, as can be seen in this example:

```
&SW1     SETB  0
```

Here, a value of zero is assigned to &SW1.

The operand of a SETB statement can also be a *logical expression* that is evaluated by the assembler as true or false. If the expression is true, the SET symbol is assigned the value 1. If the expression is false, zero is assigned. Logical expressions are composed of two arithmetic expressions or two character expressions connected by one of the following *logical operators*:

OPERATOR	MEANING
EQ	Equal to
NE	Not equal to
LT	Less than
LE	Less than or equal to
GT	Greater than
GE	Greater than or equal to

The expressions compared by the logical operators can be self-defining terms, arithmetic expressions composed of arithmetic operators and variable names, or symbol attributes. In the following examples, the logical expressions are all true, so the value assigned in each case is 1:

```
          STATEMENT                                VARIABLE
                                                   VALUES
&X3       SETB  (8 GT 3)
&LIMIT    SETB  (&INDEX LE &HIGH)                  &INDEX=19,
                                                   &HIGH=20
&SWITCH   SETB  (&PARM1+5 NE &PARM2*3)             &PARM1=4,
                                                   &PARM2=6
&DONE     SETB  (4 EQ L'&KEY2)                     Length of
                                                   &KEY2 is 4
```

In this next group of SETB statements, the logical expression is false so the value assigned to the binary SET symbol in each case is zero:

```
          STATEMENT                                VARIABLE
                                                   VALUES
&B1       SETB  (&NAME EQ 'FIRST')                 &NAME='BRK'
&B2       SETB  (T'&P1 NE 'C')                     Type code of
                                                   &P1='C'
&SYMB     SETB  (&KEY4 EQ &END)                    &KEY4='NXT',
                                                   &END='END'
```

SETC Character SET symbols are assigned values with a SETC statement. The operand can be a self-defining character string, another variable symbol, an attribute of a variable symbol, or any of these combined or concatenated. The operand must be enclosed in single quotes as in these examples:

Statement	Symbol values	Assigned value
SETA EXAMPLES:		
`&A72 SETA 72`		72
`&NBR SETA &INDEX+10`	&INDEX=3	13
`&LIMIT SETA &BASE*5-&ORG`	&BASE=100,&ORG=37	463
`&LGTH SETA L'&P1-1`	Length of &P1=7	6
SETB EXAMPLES:		
`&SW1 SETB 0`		0
`&YES SETB (&PARM2 EQ 'YES')`	&PARM2='NO'	0
`&TOOLONG SETB (L'&FLD LE 256)`	Length of &FLD=180	1
SETC EXAMPLES:		
`&STRING SETC 'AB C'`		'AB C'
`&NAME SETC '&KEY1'`	&KEY1=JONES	'JONES'
`&NAME SETC 'MR .&KEY1'`	&KEY1=JONES	'MR JONES'
`&PRT1 SETC 'MR.. &KEY1'`	&KEY1=BOLTZ	'MR. BOLTZ'
`&TNAME SETC 'T'&NAME'`	Type Attribute of &NAME=C	'C'

FIGURE 10-10 SET Statement Examples

	STATEMENT		REMARK
`&STRING`	`SETC`	`'KEY'`	Operand is a self-defining string
`&NAME`	`SETC`	`'&KEY1'`	Operand is a variable symbol
`&NAME`	`SETC`	`'MR &KEY1'`	Operand is a string plus variable symbol
`&TYPE`	`SETC`	`'T'&PARM1'`	Operand is the type attribute

Figure 10-10 gives several examples of each type of SET statement and shows the resulting value assigned to the symbol. Note that, in the third SETC statement, two periods in a character string are required for one period to show in the resulting value because a period is the concatenation character. Similarly, two consecutive apostrophes are required for one apostrophe to show in the resulting value.

CONDITIONAL ASSEMBLY STATEMENTS

The primary conditional assembly statements are the AGO and AIF statements. They are approximately equal to an unconditional-branch instruction and a conditional-branch instruction, but they direct the assembly of the source program rather than the execution of the object program.

Normally, the assembler program processes the source-program statements sequentially. Conditional assembly statements allow you to control the sequence in which the source statements are processed. You can cause the assembler to skip some of the statements and jump ahead in the source-statement sequence or to branch backward in the sequence to have some of the source code processed again—just as if it had been coded in the source program more than once. To illustrate, let's consider this code from a macro definition:

```
          .
          .
          .
          AGO    .END
          LA     &R1,256(&R1)
          LA     &R2,256(&R2)
          MVC    0(256,&R2),0(&R1)
          .
          .
.END      MVC    0(&LGTH,&R2),0(&R1)
          .
          .
```

Here, the AGO instruction causes the assembler to skip ahead to the last MVC instruction. As a result, the two LA instructions, the first MVC instruction, and any other instructions before .END aren't generated in the resulting source code.

The label .END is called a *sequence symbol*. Sequence symbols are labels that can be referred to in conditional assembly instructions in order to direct the assembly sequence, but they aren't generated with the rest of the statement that they label. A period is the first character of a sequence symbol.

The operand of the AIF statement must be a logical expression followed by a sequence symbol. The logical expression must be formed in the same way as those used in SETB statements. If the expression is true, the assembler branches to the sequence symbol that follows the logical expression. If the expression is false, the assembler continues processing the statements sequentially.

In the example that follows, a variable symbol, &LGTH, is compared to a value of 256:

```
AIF    (&LGTH LE 256).SHORT
```

If &LGTH is less than or equal to 256, the expression is true and the assembler interrupts its sequential processing and goes to the statement starting with the sequence symbol .SHORT. If the expression is false, the assembler will process the statement following the AIF statement.

Logical expressions can also include the operators OR, AND, and NOT in various combinations to form complex logical expressions. For example, the logical expression

```
(&A GT 14 AND &B LT 12)
```

is true only if both logical terms are true. In contrast,

```
(&A GT 14 OR &B LT 12)
```

is true if either one or both of the logical terms are true.

Sometimes it is necessary to use multiple sets of parentheses to group the various logical expressions so they will be evaluated properly:

```
((&A GT 2 AND &A LT 13) OR
(&A GT &B AND NOT &A GT 15))
```

In this example, if either of the internal expressions is true, the overall expression is true. Since the internal expressions are made up of two expressions in AND relationships, both expressions in the AND relationships must be true to make the internal expression true. Although the AND, OR, and NOT operators can be used in logical expressions for both SETB and AIF instructions, they are used most often in AIF statements.

THE MOVE-LONG MACRO

The System/360 MVC instruction can move a maximum length of 256 bytes. To increase this maximum, the move-long macro shown in figure 10-11 has been written. It allows a move of any number of bytes to be coded in a single macro instruction. The first two operands name the receiving and sending fields and the third operand specifies the number of bytes to be moved. During macro expansion, the macro can generate one or more MVC instructions that move the data in 256-byte chunks plus a final MVC instruction that moves 256 bytes or less. The first MVC generated is assigned the label that is coded on the source-program macro statement. If more than one MVC

MACRO DEFINITION:

```
         MACRO
&LABEL   MOVEL &RFLD,&SFLD,&LGTH
         LCLA  &DISP,&ALGTH
         LCLC  &FIRST
&ALGTH   SETA  &LGTH
&FIRST   SETC  '&LABEL'
.LOOP    AIF   (&ALGTH LE 256).LAST
&FIRST   MVC   &RFLD+&DISP.(256),&SFLD+&DISP
&DISP    SETA  &DISP+256
&ALGTH   SETA  &ALGTH-256
&FIRST   SETC  ''
         AGO   .LOOP
.LAST    ANOP
&FIRST   MVC   &RFLD+&DISP.(&ALGTH),&SFLD+&DISP
         MEND
```

SAMPLE MACRO STATEMENT:

```
MOVERCD MOVEL OUTA,RCD,600
```

GENERATED SOURCE STATEMENTS:

```
MOVERCD MVC   OUTA+0(256),RCD+0
        MVC   OUTA+256(256),RCD+256
        MVC   OUTA+512(88),RCD+512
```

FIGURE 10-11 Move-Long Macro

instruction is generated (the third operand is more than 256), each successive MVC addresses an area 256 bytes farther into the fields than the previous MVC.

To see how this macro definition is expanded, assume the sample statement

```
MOVERCD  MOVEL OUTA,RCD,600
```

in figure 10-11 has been coded in a program. When the assembler finds this statement in the source program, it goes to the source-statement library to find the macro definition. After the definition has been found, the prototype symbols are assigned the values coded in the source macro statement: &LABEL=MOVERCD, &RFLD=OUTA, &SFLD=RCD, &LGTH=600. Then, the symbols &DISP, &ALGTH, and &FIRST are declared so the first two are given initial values of zero and &FIRST is given an initial value of a null character string. At this point, the assembler reaches these SET statements:

```
&ALGTH   SETA  &LGTH
&FIRST   SETC  '&LABEL'
```

The first statement assigns the value 600 to &ALGTH; the second assigns the string MOVERCD to &FIRST.

Next, the assembler encounters this AIF statement:

```
.LOOP    AIF   (&ALGTH LE 256).LAST
```

Since &ALGTH has a value of 600, the statement is false, no branch takes place, and this statement is reached:

```
&FIRST   MVC   &RFLD+&DISP.(256),&SFLD+&DISP
```

When the current variable values are substituted in the statement, the following source statement is generated:

```
MOVERCD  MVC   OUTA+0(256),RCD+0
```

Then, the assembler reaches these statements

```
&DISP    SETA  &DISP+256
&ALGTH   SETA  &ALGTH-256
&FIRST   SETC  ''
         AGO   .LOOP
```

so the value of &DISP is increased by 256, the value of &ALGTH is decreased by 256, and the value of &FIRST is set to a null string. The AGO statement then branches the assembler back to the AIF statement.

The second time through the definition, the value of &ALGTH is 344 so the AIF still doesn't branch. The following source statement is then generated:

```
         MVC   OUTA+256(256),RCD+256
```

and the loop repeats. Note here that no label is given to the instruction since &FIRST has a null value.

The third time through the definition, &ALGTH has a value of 88 so the branch to .LAST takes place. Since the

operation code of the .LAST statement is ANOP (no operation), the assembler goes on to the MVC model statement. That causes this statement, the final source statement, to be generated:

```
        MVC   OUTA+512(88),RCD+512
```

When the assembler reaches the MEND statement, it returns to the source code following the MOVEL macro statement.

To illustrate the macro expansion if the length is less than 256, consider this macro statement:

```
MOVITM  MOVEL PITEM,MITEM,8
```

This time the AIF statement will branch to .LAST the first time through the definition. This source statement is then generated:

```
MOVITM  MVC   PITEM+0(8),MITEM+0
```

To write this move-long macro definition, I used two techniques that you should note. First, I used the character symbol &FIRST so the label on the source macro statement can be generated on the first MVC, but not on succeeding ones. I accomplished this by assigning &FIRST the value of &LABEL before entering the main loop and by reassigning a null value after the first MVC has been generated.

Second, the ANOP assembly instruction provides a no-operation instruction. It is used to provide an instruction on which to place a sequence symbol when the sequence symbol can't be coded directly on the model statement. In this case, I needed to have &FIRST on the MVC instruction, so the sequence symbol couldn't go on it.

One other thing you should note about the move-long macro is that there are only two model statements in the definition—the two MVC statements. All the other statements are assembler statements that are part of the macro-writing facility. As you can imagine, then, most of the statements in a complex macro definition are assembler statements as opposed to model statements.

CHECKING MACRO SOURCE STATEMENTS

Macro definitions often include some checking or editing of the source-program macro statement. If checking or editing is required, conditional assembly statements are used to check that all the necessary operands are present, that certain operands are numeric, and so on. If any improper conditions are found, the MNOTE instruction can be used to cause an error message to be printed on both the source listing and the diagnostic listing.

To illustrate, suppose the MOVEL macro is modified so it starts as follows:

```
        MACRO
&LABEL  MOVEL &RFLD,&SFLD,&LGTH
        LCLA  &DISP,&ALGTH
        LCLC  &FIRST
        AIF   (T'&LGTH EQ 'N').LOK
        MNOTE 2,'LENGTH OPERAND NOT NUMERIC'
        MEXIT
.LOK    ANOP
        .
        .
        .
```

Here, the AIF statement tests the type attribute of the length operand to see if it is equal to 'N.' If the operand is numeric, the expansion continues with the sequence symbol .LOK. If the operand isn't numeric, the MNOTE instruction causes the message LENGTH OPERAND NOT NUMERIC to be printed on the assembly listing.

The first operand of the MNOTE statement is the *severity code*. If a severity code is present, the message will be printed in the error diagnostic listing at the end of the assembly listing as well as in the source listing immediately following the macro. If the severity-code operand is omitted,

```
MNOTE 'LENGTH OPERAND NOT NUMERIC'
```

the MNOTE message is only printed in the source listing. Other than causing the MNOTE message to be printed in the diagnostic listing, the severity code has no function in DOS.

```
          BKEND   A.GET
                                                                                S4530001
          MACRO                                                                 S4530002
&LABEL    GET     &FILEN,&PARAM                                                 S4530003
.* IBM SYSTEM/360 DISK OPERATING SYSTEM                                        S4530004
* 360N-CL-453 GET       CHANGE LEVEL 3-0                                       S4530005
          AIF   (T'&FILEN NE '0').ONE                                           S4530006
          MNOTE 0,'NO FILENAME SPECIFIED.SET TO '*''                            S4530007
&LABEL    L     1,=A(*)                  *****ERROR-PATCH DTF TABLE ADDRESS   S4530008
          AGO   .THREE                                                          S4530009
.ONE      AIF   ('&FILEN'(1,1) NE '(').TWO                                      S4530010
          AIF   ('&FILEN(1)' EQ '1').FOUR                                       S4530011
&LABEL    LR    1,&FILEN(1)                GET DTF TABLE ADDRESS              S4530012
          AGO   .THREE                                                          S4530013
.TWO      ANOP                                                                  S4530014
&LABEL    L     1,=A(&FILEN)               GET DTF TABLE ADDRESS              S4530015
.THREE    AIF   (T'&PARAM EQ '0').EIGHT                                         S4530016
          AIF   ('&PARAM'(1,1) EQ '(').SIX                                      S4530017
          L     0,=A(&PARAM)               GET WORK AREA ADDRESS              S4530018
          AGO   .EIGHT                                                          S4530019
.SIX      AIF   ('&PARAM(1)' NE '1').SEVEN                                      S4530020
          MNOTE 0,'INVALID REGISTER SPECIFICATION FOR WORKAREA'                 S4530021
          LR    0,0                        *****ERROR-PATCH REGISTER NUMBER   S4530022
          AGO   .EIGHT                                                          S4530023
.SEVEN    AIF   ('&PARAM(1)' EQ '0').EIGHT                                      S4530024
          LR    0,&PARAM(1)                GET WORK AREA ADDRESS              S4530025
.EIGHT    L     15,16(1)                   GET LOGIC MODULE ADDRESS           S4530026
          BAL   14,8(15)                   BRANCH TO GET ROUTINE              S4530027
          MEXIT                                                                 S4530028
.FOUR     ANOP                                                                  S4530029
&LABEL    DC    0H'0'                                                           S4530030
          AGO   .THREE                                                          S4530031
          MEND

          BKEND
```

FIGURE 10-12 The GET Macro

The example above also introduces the MEXIT (pronounced M EXIT) statement. It causes the assembler to discontinue expanding the macro and go back to the source program. An MEXIT can have a sequence symbol, but no operands.

IBM-SUPPLIED MACROS

Figures 10-12, 10-13, and 10-14 are the source-statement library listings of the IBM GET and SAVE macros and the first part of the DTFCD macro. They illustrate extensive use of conditional assembly statements. By studying them, you can get a better appreciation of what writing a complex macro definition involves. Because proficiency in macro writing can only be achieved by studying definitions like these and by writing your own definitions, I won't explain them in detail. However, I will point out a few highlights.

First, *internal comments* are used in all three definitions. An internal comment has .* in columns 1 and 2 and any characters in the remaining columns. Unlike a regular comment, it isn't printed when the macro is expanded on the source listing. However, it prints when the macro definition itself is listed. The internal comments shown in the macro definition listing indicate the operating system being used.

Second, the statement labeled .ONE in the GET macro

```
         BKEND    A.SAVE

         MACRO                                                          D4530001
&NAME    SAVE  &REGS                                                    D4530002
         LCLA  &R1,&R2                                                  D4530003
         LCLA  &KAREGS                                                  D4530004
.* IBM SYSTEM/360 DISK OPERATING SYSTEM                                 D4530005
* 360N-CL-453 SAVE      CHANGE LEVEL 3-8                          3-8  D4530006
         AIF   (T'&REGS(1) EQ 'N').E                                    D4530007
         MNOTE 1,'FIRST REGISTER NOT A SELF-DEFINING VALUE. 14 ASSUMED.' D4530008
         AGO   .F                                                       D4530009
.E       ANOP                                                           D4530010
&KAREGS  SETA  K'&REGS                                                  D4530011
         AIF   ('&REGS'(1,1) EQ '(' AND '&REGS'(&KAREGS,1) EQ ')').C    D4530012
           MNOTE 0,'OPERAND NOT ENCLOSED IN PARENTHESES - IN ERROR IF   *D4530013
                  MORE THAN ONE OPERAND'                                D4530014
&R1      SETA  &REGS                                                    D4530015
         AGO   .D                                                       D4530016
.C       ANOP                                                           D4530017
&R1      SETA  &REGS(1)                                                 D4530018
.D       AIF   (&R1 NE 13 AND &R1 LE 15).A                              D4530019
       MNOTE 3,'1ST REG MUST NOT BE 13 OR GREATER THAN 15. 14 ASSUMED.' D4530020
.F       ANOP                                                           D4530021
&R1      SETA  14                                                       D4530022
.A       AIF   (N'&REGS EQ 1).ST                                        D4530023
         AIF   (T'&REGS(2) EQ 'N').H                                    D4530024
     MNOTE 1,'2ND REGISTER NOT A SELF-DEFINING VALUE.12 ASSUMED.'  3-8  D4530025
           AGO   .G                                                     D4530026
.H       ANOP                                                           D4530027
&R2      SETA  &REGS(2)                                                 D4530028
           AIF   (&R1 GE 14 AND (&R2 GE &R1 AND &R2 LE 15 OR &R2 LE 12)  -D4530029
                 OR &R1 LE &R2 AND &R2 LE 12).B                         D4530030
       MNOTE 3,'IMPROPER RANGE OF REGISTERS. 2ND REG = 12 ASSUMED.'     D4530031
.G       ANOP                                                           D4530032
&R2      SETA  12                                                       D4530033
.B       ANOP                                                           D4530034
&NAME      STM   &R1,&R2,12+4*(&R1+2-(&R1+2)/16*16)(13)                 D4530035
         MEXIT                                                          D4530036
.ST      ANOP                                                           D4530037
&NAME      ST          &R1,12+4*(&R1+2-(&R1+2)/16*16)(13)               D4530038
         MEND                                                           D4530039

         BKEND
```

FIGURE 10-13 The SAVE Macro

definition illustrates selection of characters from a character string:

```
.ONE     AIF   ('&FILEN'(1,1) NE '(').TWO
```

The first part of the logical expression—'&FILEN'(1,1)—means: select a character string beginning in position 1 of &FILEN and continue for a length of 1. In this case, the first character of the filename operand is compared to a left parenthesis. (The presentation of the GET macro in chapter 3 didn't mention it, but the filename operand of GET can be the number of a register enclosed in parentheses. If used, the register is expected to contain the address of the DTF. This AIF is thus checking to see if the register notation is used.)

Third, the same type of character selection is made in statement number D4530012 of the SAVE macro definition. It's also an AIF instruction that checks to see that parentheses surround the registers specified as the operand of the macro, which is a sublist operand. It uses a SET symbol, &KAREGS,

```
          BKEND   A.DTFCD
                                                                                E4530001
          MACRO
&NAME     DTFCD &BLKSIZE=80,&CONTROL=,&CRDERR=,&CTLCHR=,&DEVADDR=,&DEVIC1E4530002
                E=2540,&EOFADDR=,&IOAREA1=,&IOAREA2=,&IOREG=,&OUBLKSZ=,&2E4530003
                RECFORM=FIXUNB,&RECSIZE=,&SSELECT=,&TYPEFLE=INPUT,&#ORKA3E4530004
                =,&MODNAME=,&SEPASMB=,&RDONLY=                          3-3 E4530005
          LCLA  &AL1,&AL2,&AL3,&AL4,&AL8,&AL9,&BLKS,&OUBL                   E4530006
          LCLB  &BL1                                                        E4530007
    LCLC &CG1,&CG2,&SFX,&CNTL,&WKIO,&IOA2,&RECF,&IOA1,&RECZ,&IOG      3-9 E4530008
.* IBM SYSTEM/360 DISK OPERATING SYSTEM.                                    E4530009
* 360N-CL-453  DTFCD   CHANGE LEVEL 3-10                             3-10 E4530010
&CG1      SETC  '&DEVICE'(1,4)                                              E4530011
&SFX      SETC  '&SYSNDX'                                                   E4530012
          AIF   (K'&NAME LT 8).ADREM                                        E4530013
          MNOTE 9,'FILE NAME EXCEEDS SEVEN CHAR LIMIT, MACRO GENERATION *E4530014
                TERMINATED'                                                 E4530015
          MEXIT                                                             E4530016
.ADREM     AIF   ('&TYPEFLE' EQ 'OUTPUT').D1                                E4530017
           AIF   (T'&CTLCHR EQ 'O').D3                                      E4530018
           MNOTE 0,'CTLCHR INVALID PARAMETER. IGNORED'                      E4530019
.D3        AIF   (T'&RECFORM EQ 'O' OR '&RECFORM' EQ 'FIXUNB').D4           E4530020
           MNOTE 0,'IMPROPER RECFORM. ''FIXUNB'' ASSUMED'                   E4530021
.D4        AIF   (T'&RECSIZE EQ 'O').D5                                     E4530022
           MNOTE 0,'RECSIZE INVALID PARAMETER. IGNORED'                     E4530023
           AGO   .D5                                                        E4530024
.D1        AIF   (T'&EOFADDR EQ 'O').D5                                     E4530025
           MNOTE 0,'EOFADDR INVALID PARAMETER. IGNORED'                     E4530026
.D5        ANOP                                                             E4530027
          AIF   (T'&BLKSIZE EQ 'N').E00                                     E4530028
          MNOTE 0,'IMPROPER BLKSIZE. SET TO 80'                             E4530029
&BLKS     SETA  80                                                          E4530030
          AGO   .E2                                                         E4530031
.E00      ANOP                                                              E4530032
&BLKS     SETA  &BLKSIZE                                                    E4530033
          AIF   ('&RECFORM' NE 'FIXUNB').E2                                 E4530034
          AIF   ('&CTLCHR' NE '' AND '&TYPEFLE' EQ 'OUTPUT').Q1             E4530035
          AIF   (&BLKS LE 80).E2                                            E4530036
&BLKS     SETA  80                                                          E4530037
          AGO   .Q15                                                        E4530038
```

FIGURE 10-14 The DTFCD Macro—The First 72 of 429 Statements

to specify the character position to check for the right parenthesis. A value is assigned to &KAREGS by the preceding statement, which is a SETA statement that assigns the count attribute of the SAVE operand to &KAREGS. The operand &REGS is considered to be one operand since it is enclosed in parentheses which indicates a sublist.

Finally, the beginning portion of the DTFCD definition illustrates extensive checking for operand omissions by comparing the type attribute to O. In some cases, an omitted operand merely allows a default value to be used. In other cases, an omission means that the macro cannot be generated properly and must be corrected. This type of check is used throughout the standard macros and should be used in your own macros to protect against omitted operands.

CONCLUSION

This chapter presents the major macro-writing facilities of Basic Assembler Language. However, there are additional facilities for which you may want to research IBM manuals

```
.Q1      AIF   (&BLKS LE 81).E2                                          E4530039
&BLKS    SETA  81                                                        E4530040
.Q15     MNOTE 0,'BLKSIZE GREATER THAN &BLKS . &BLKS ASSUMED'            E4530041
.E2      AIF   (T'&IOAREA1 NE '0').E025                                  E4530042
         MNOTE 0,'NO IOAREA1 SPECIFIED, SET TO 0'                3-8     E4530043
&IOA1    SETC  '0'                                               3-8     E4530044
         AGO   .EG1                                                      E4530045
.E025    ANOP                                                            E4530046
&IOA1    SETC  '&IOAREA1'                                                E4530047
.EG1      AIF   ('&CG1' NE '2501').REL                                   E4530048
          AIF   ('&CONTROL' EQ '').UVW                                   E4530049
          MNOTE 0,'CONTROL INVALID PARAMETER. IGNORED'                   E4530050
.UVW      AIF   ('&CTLCHR' EQ '' OR '&TYPEFLE' NE 'OUTPUT').EC92        E4530051
          MNOTE 0,'CTLCHR INVALID PARAMETER. IGNORED'                    E4530052
          AGO   .EC92                                                    E4530053
.REL      ANOP                                                           E4530054
&CNTL    SETC  'C'                                                       E4530055
         AIF   ('&CONTROL' EQ 'YES').EC93                                E4530056
         AIF   (T'&CONTROL EQ '0').EC91                                  E4530057
          MNOTE 0,'IMPROPER CONTROL. ''YES'' ASSUMED'                    E4530058
.EC93    AIF   (T'&CTLCHR EQ '0').EC9                                    E4530059
          AIF   ('&TYPEFLE' NE 'OUTPUT').EC9                             E4530060
          MNOTE 0,'BOTH CTLCHR AND CONTROL SPECIFIED.ONLY CONTROL ASSUM*E4530061
                ED'                                                      E4530062
         AGO   .EC9                                                      E4530063
.EC91    AIF   ('&TYPEFLE' NE 'OUTPUT').EC92                             E4530064
&CNTL    SETC  'A'                                                       E4530065
         AIF   ('&CTLCHR' EQ  'ASA').EC9                                 E4530066
&CNTL    SETC  'Y'                                                       E4530067
         AIF   ('&CTLCHR'  EQ 'YES').EC9                                 E4530068
         AIF   ('&CTLCHR'  EQ '').EC92                                   E4530069
          MNOTE 0,'IMPROPER CTLCHR. ''YES'' ASSUMED'                     E4530070
         AGO   .EC9                                                      E4530071
.EC92    ANOP                                                            E4530072
```

FIGURE 10-14 The DTFCD Macro—The First 72 of 429 Statements (Continued)

GC24-3414, *DOS Assembler Language*, and GC33-4010, *OS/VS and DOS/VS Assembler Language*. Although this topic is intended to familiarize you with advanced macro-writing techniques, it is only a start. To develop the ability to write macros such as the GET or DTFCD will require many hours of study on your own.

Terminology

text insertion
text insertion with modification
text manipulation
SET symbol
conditional assembly instruction
local SET symbol
global SET symbol
symbol declaration
symbol attributes:
- length
- type
- number
- count

sublist
subscript
arithmetic expression
self-defining term
arithmetic operator
logical expression
logical operator
sequence symbol
severity code
internal comment

Objective

Design and code macro definitions at the third level of difficulty: text manipulation. These definitions will require the use of SET symbols and conditional assembly instructions.

Problems

1 Add conditional assembly statements to the definition of the first-time-switch macro in figure 10-7 so that if no operand is coded a branch is made to the first instruction that follows the macro. If the operand is missing, the macro definition should also cause the assembler to print an appropriate message in the diagnostic listing.

2 This problem is designed to get you to study the GET macro in figure 10-12.

   a. List the sequence numbers of the statements within the GET macro definition that will be processed if a program macro is coded as follows:

```
CRDIN    GET   CRDFILE
```

   b. List the statements that will be generated by the macro expansion.

Solutions

1 The following code is an acceptable solution:

```
         MACRO
&LABEL   FRSTSW &BRCH
         LCLC  &ADDR
&ADDR    SETC  '&BRCH'
         AIF   (T'&BRCH NE 'O').GOOD
         MNOTE 1,'NO OPERAND PRESENT'
&ADDR    SETC  '*+4'
.GOOD    ANOP
&LABEL   BC    0,&ADDR
         MVZ   *-3,X'F0'
         MEND
```

2 a. S4530001, −0002, −0003, −0004, −0005, −0009, −0013, −0014, −0015, −0025, −0026

   b.
```
CRDIN    L     1,=A(CRDFILE)     GET DTF TABLE ADDRESS
         L     15,16(1)          GET LOGIC MODULE ADDRESS
         BAL   14,8(15)          BRANCH TO GET ROUTINE
```

11

Useful Standard Macros and Assembler Commands

This chapter presents a number of macro instructions and assembler commands that are in common use by professional programmers. Topic 1 presents the macros; topic 2 presents the assembler commands. The topics aren't related so it doesn't matter which you read first.

TOPIC ONE Standard Macros There are several different kinds of standard macros that are provided with the Disk Operating System. One kind you are already familiar with is the *I/O macro*—for example, the DTF, OPEN, CLOSE, GET, and PUT macros. If you have read chapter 9, you are also familiar with *program-linkage macros* such as the CALL and RETURN macros. In addition, there are *system-generation macros* that are used for assembling the supervisor and *multitasking macros* that are used for special operations when two or more programs are

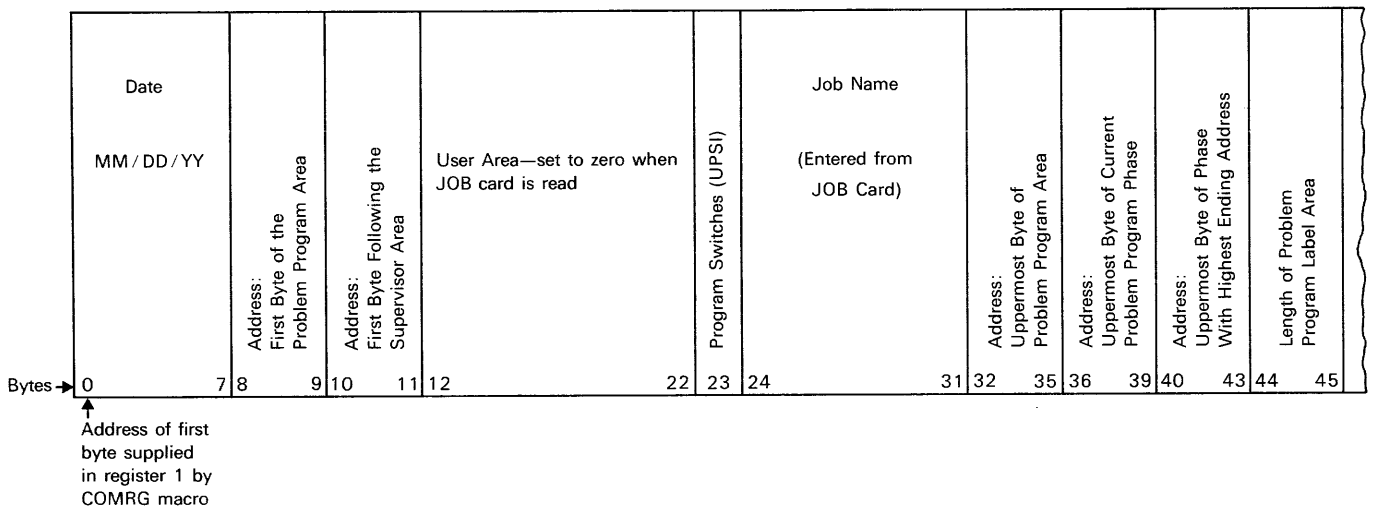


FIGURE 11-1 Communication Region of the Supervisor

being executed by one CPU. These types of macros are beyond the scope of assembler-language programming.

One type of macro that almost all professional BAL programmers use is the *supervisor-communication macro.* This type allows you to get information from the supervisor, give information to it, or request a service from it. Some of the most widely used macros in this group are: COMRG, GETIME, CANCEL, DUMP, and PDUMP.

COMRG Within the DOS supervisor there is a special area called the *communication region.* In this area, the date and various other items of information are stored. Figure 11-1 is a map of the 46 bytes of the communication region that are of interest to the assembler-language programmer. For instance, bytes 24-31 contain the jobname of the job being run, and bytes 12-22 can be used to pass data from a program in one job step to a program in a job step that follows.

The COMRG macro can be coded to make the data in the communication region available to your program. The COMRG macro has no operands, but it can have a label if you need to refer to it. When this macro is executed, the address of the communication region is placed in register 1 and control is returned to your program. You must then code instructions to extract the data and save it in your program area.

In actual practice, the date is the only data in the communication region that is commonly used by the assembler-language programmer. The following instructions illustrate the beginning of a program that gets the date from the communication region:

```
PROG36   START 0
         BALR  3,0
         USING *,3
         COMRG
         MVC   DATE,0(1)
         .
         .
DATE     DS    CL8
```

The date can then be printed at the top of a report or other document.

Most high-level languages do not have this ability to get

```
DATA DIVISION.
FILE SECTION.
    ...
    ...
WORKING-STORAGE SECTION.
77  CDATE    PIC X(8).
    ...
    ...
    ...
PROCEDURE DIVISION.
BEGIN.
    CALL 'GETDATE' USING CDATE.
```

Cobol Main Program

```
GETDATE  START 0
         USING *,15
         L     1,0(1)
         ST    1,FLDADDR
         COMRG
         MVC   BDATE,0(1)
         L     1,FLDADDR
         MVC   0(8,1),BDATE
         BR    14
BDATE    DS    CL8
FLDADDR  DS    F
         END
```

Assembler Subprogram

FIGURE 11-2 Subprogram for Getting Date

the current date from the communication region. Because of this, it is common to have an assembler subprogram that gets the date for COBOL main programs. Figure 11-2 illustrates one version of this type of BAL subprogram with a COBOL main program.

If you haven't read chapter 9 and aren't familiar with COBOL, you may not understand the BAL subprogram. To put it simply, a COBOL main program branches to the address in register 15 to get to the subprogram and gives the address to be returned to in register 14. Thus, the BAL subprogram in figure 11-2 uses register 15 as its base register, and it branches to the address in register 14 when it is finished.

The COBOL main program also uses register 1 to indicate where those of its fields needed by the subprogram can be found. This isn't done directly, though. Instead, register 1 contains an address that refers to one or more fullwords containing the addresses of the fields needed by the subprogram. When the COBOL program passes control to the BAL subprogram in figure 11-1, then, register 1 contains the address of a fullword that contains the address of the COBOL field named CDATE.

Because register 1 is used for program linkage as well as by the COMRG macro, the subprogram first stores the address of the COBOL field CDATE using this code:

```
L     1,0(1)
ST    1,FLDADDR
```

The load instruction moves the contents of the fullword addressed by register 1 into register 1. Thus, register 1 contains the address of CDATE after this load instruction has been executed. Then, the address of CDATE is stored in the fullword named FLDADDR. After the COMRG macro has been used to get the date and the date has been stored in BDATE, the BAL date field is moved to the COBOL date field by these instructions:

```
L     1,FLDADDR
MVC   0(8,1),BDATE
```

GETIME The DOS supervisor also maintains the time of day in an internal clock. (This is true only if the CPU has the timer feature and the supervisor is generated to include the clock, but most System/360-370s do include this feature.)

```
          .
          .
          GETIME STANDARD
          ST     1,TIME
          MVC    PRTTIME,PATTERN
          ED     PRTTIME,TIME
          .
          .
          .
TIME      DS     F
PATTERN   DS     X'402021206120206120 20'
          .
          .
PRTTIME   DS     CL10
          .
          .
```

NOTES:
1 Hex 61 is the slash
2 Edited time appears as HH/MM/SS

FIGURE 11-3 Printing the Time

You can retrieve the time of day from the supervisor by using the GETIME macro.

When the GETIME macro has been executed, the time value is returned in register 1. The format of this value is determined by the operand of the macro. The three valid forms of the operand are:

```
GETIME STANDARD
GETIME BINARY
GETIME TU
```

If the operand is omitted, STANDARD is assumed.

When STANDARD is coded or the operand omitted, the time is returned as a packed-decimal number in the format HHMMSS where H is hours, M is minutes, and S is seconds. If the time is 08:32:48, for instance, the four bytes of register 1 will contain hex 0083248C. Here, the rightmost half-byte contains a valid positive sign. Figure 11-3 illustrates a routine that gets the time in packed-decimal format and edits it into a print-line field.

When BINARY is coded as the operand, the value placed in register 1 is a binary number. It represents the number of seconds that have elapsed since 12:00 A.M. of the previous day. The operand TU also causes a binary value to be placed in register 1, but this value is in units of 1/300 of a second. As a result, it is 300 times the value returned when the operand is BINARY.

CANCEL, DUMP, and PDUMP As you write more complex programs and the number of possible error situations increases, you will find many cases in which you'll want a storage dump to help you figure out what caused the problem. DOS has three standard macros that provide this facility.

For the most catastrophic error conditions, you can code a CANCEL macro. This macro can have a label but no operands. It causes the program to be canceled (abnormally terminated). A dump of storage will occur if the dump option was specified when the supervisor was generated or if a // OPTION DUMP card has been included in the job-control cards for the program. The storage dump includes the general registers, the supervisor area, and the rest of main storage.

The DUMP macro provides a similar capability. The dump will always occur, however, and the program will be terminated. In a multiprogramming system (more than one program in storage at one time), DUMP will print the supervisor, the registers, and its own partition (program area), but not the other partitions of main storage.

For less critical errors, you may want to use the PDUMP (partial dump) macro. A PDUMP provides a dump of the general registers and the program area between two addresses that you supply as operands. For example:

```
PDUMP SYMBOL1,SYMBOL2
PDUMP (6),(8)
```

The first example dumps the area between the two labels

```
DATA DIVISION.
FILE SECTION.
    ...
    ...
    ...
WORKING-STORAGE SECTION.
77  WS-BC-COUNT     PIC S9(5),  VALUE ZEROS.
    ...
    ...
    ...
    02  END-DUMP    PIC X(59),  VALUE SPACES.
PROCEDURE DIVISION.
...
    ...
    ...
    CALL 'PDUMP' USING WS-BC-COUNT, END-DUMP.
```

Cobol Main Program

```
PDUMP    START 0
         USING *,15
         STM   6,7,SAVE
         LM    6,7,0(1)
         PDUMP (6),(7)
         LM    6,7,SAVE
         BR    14
SAVE     DS    2F
         END
```

Assembler Subprogram

FIGURE 11-4 Subprogram for Snapshot Dump

SYMBOL1 and SYMBOL2, wherever they are defined in the program. (Of course, SYMBOL1 should precede SYMBOL2.) The second example dumps the area from the address in register 6 to the address in register 8. When the registers and the selected program area have been dumped, processing continues with the next instruction following the PDUMP macro.

PDUMP thus provides a "snapshot" dump capability. You can code it in certain parts of a new program to dump record areas, counter fields, total fields, and so on, in order to track the processing of a program. Then, when the program has been debugged, you can remove the PDUMPS and reassemble.

You may also want to include PDUMP as a permanent part of some error routines. This will allow you to get a snapshot dump of the troublesome records and then continue to process the rest of the data.

A PDUMP function is another useful assembler-language subprogram for higher level language programs. Figure 11-4, for example, is a PDUMP subprogram that provides this capability. The calling main program—in this case, a COBOL main program—provides the starting and ending addresses for the dump by referring to its own labels or data names in its link to the subprogram. When control passes to the BAL subprogram, register 1 contains the address of the first of two fullwords containing the starting and ending addresses for PDUMP.

Terminology

I/O macro
program-linkage macro
system-generation macro
multitasking macro
supervisor-communication macro
communication region

Objective

1 Use any of the macros in this chapter in an appropriate program.

Problems

1 Suppose the inventory-listing program in figure 3-15 has a serious bug in it. Although the program runs to completion and the output prints, the output makes no sense at all. As a result, you would like the contents of all data fields printed (from CRDIPTA to the end of the program) each time a card is processed. How would you use PDUMP for this purpose?

Solution

1 Insert the following code as line 215:

```
PDUMP CRDINPA,TOTVALUE+5
```

TOPIC TWO Assembler Commands You are already familiar with many assembler commands: START, USING, END, DSECT, and ORG are all assembler commands. In addition, if you have read chapter 9, you have seen some commands for writing subroutines and subprograms. If you have read chapter 10, you have seen some commands for writing macro definitions. This topic first describes the USING command as it is used for specifying multiple base registers. Next, it presents two commands for controlling the assembly process—LTORG and EQU. Finally, it presents a few commands that control the appearance of both the assembly listing and the punched-card output of the assembly.

MULTIPLE BASE REGISTERS

Do you remember from chapter 3 that a single base register can serve a maximum program segment of 4096 bytes? This means that another base register must be assigned and loaded for each additional program segment of 4096 bytes. For example, if a program requires three base registers, the base registers can be assigned and loaded as shown in figure 11-5. After the BALR instruction loads the first base register as usual, the USING statement assigns register 3 as the first base register, and then assigns registers 4 and 5 as succeeding base registers. The assembler program will use register 3 as the base register until 4096 bytes have been defined. Then, for references to any bytes beyond the 4096th from the beginning of the program, it will use register 4 or 5 as the base register. Register 4 serves as the base for the program area from 4097 to 8192 bytes from the start of the program. Register 5 is used as the base register for the last segment of the program, from 8193 to 12288 bytes from the start of the program.

```
PROGC11  START 0
BEGIN    BALR  3,0
         USING *,3,4,5
         LM    4,5,BASEADR
         B     EXEC
BASEADR  DC    A(BEGIN+4096,BEGIN+8192)
EXEC     .
         .
         .
```

FIGURE 11-5 Using Multiple Base Registers

To load the proper base addresses in registers 4 and 5, the load-multiple (LM) instruction is used with *address constants*. An address constant causes the address of one or more locations in storage to be stored in one or more fullwords. In figure 11-5, this address constant is used:

```
BASEADR  DC    A(BEGIN+4096,BEGIN+8192)
```

A is the type code, and the addresses of the fields given in parentheses following the type code are placed in as many fullwords of storage as are needed. Thus, the address of BEGIN+4096 is placed in the first fullword; the address of BEGIN+8192 is placed in the second fullword. Then, when the LM instruction is executed, the appropriate base addresses are loaded in registers 4 and 5.

Notice that the unconditional branch to EXEC in figure 11-5 branches over the address constants to the next instruction of the program. Naturally, a program check would occur if this wasn't done since the address constant would

not be a valid instruction. Although the address constants (or *adcons*) can be defined anywhere within the first 4096-byte segment of the program, it is most common to find them in the housekeeping routine as shown in the figure.

To determine the number of base registers required by your program, you must estimate the number of bytes that the program will require. To do this, use 100 bytes for each DTF statement and 5 bytes for each instruction. Then, add the DTF bytes, the instruction bytes, and the number of bytes for I/O areas. This rough estimate should be close enough for determining the number of registers needed. If you assign too few, a diagnostic will call attention to your error.

```
          .
          .
          AP     QTYWORK,=P'1'
          .
          .
QTYWORK   DC     PL4'0'
          LTORG
PRTWK1    DS     CL133
INAREA1   DS     CL400
          END    BEGIN
```

FIGURE 11-6 Using LTORG

LTORG AND EQU

The assembler program normally places literal constants coded as instruction operands at the end of your program. In this position they are likely to be overlaid by input records or blocks that are longer than expected. By using the LTORG instruction, you can force placement of these literal constants at some other position in your program.

In figure 11-6, the LTORG instruction causes the literal constant =P'1' to be defined following the field QTYWORK. In other words, it tells the assembler where literals should start. If the LTORG instruction had not been included, the literal constant would have been defined following the input area INAREA1. You can code any number of LTORG statements in a program. Each time the assembler encounters a LTORG instruction, it places all the literals defined in operands since the last LTORG statement immediately following the new LTORG.

The EQU instruction allows you to assign the attributes of one label to another label as:

```
FOLLY     DS     CL8
MIRTH     EQU    FOLLY
```

Here, the symbol MIRTH is assigned the address and length attribute of FOLLY. The two symbols can then be used interchangeably in instruction operands.

One of the most common uses of the EQU instruction assigns symbols to register numbers. Since symbols are included in the cross-reference list of the assembly listing but registers aren't, using the EQU instruction allows you to get a trace of register usage that would normally not be available. This is how the registers are normally coded:

```
R0        EQU    0
R1        EQU    1
R2        EQU    2
          .
          .
          .
R14       EQU    14
R15       EQU    15
```

Then, whenever a register number is coded in an instruction, you substitute the equal symbol as in these examples:

```
          LA     R15,SUBADDR
          BALR   R14,R15
```

The statement numbers of these two instructions will be included in the cross-reference entry for the symbol R15. This coding technique can be especially helpful in debugging large programs.

```
PROG618   START 0
BEGIN     BALR  3,0
          USING *,3
          B     PROCESS
          PRINT NOGEN
CARDIN    DTFCD DEVADDR=SYSIPT,IOAREA1=C1,....
PRTOUT    DTFPR DEVADDR=SYSLST,IOAREA1=P1,....
          PRINT GEN
PROCESS   OPEN  CARDIN,PRINTOUT
```

FIGURE 11-7 Use of PRINT Command to Suppress Generated Instructions

ASSEMBLY LISTING CONTROL

There are several assembler instructions that control the presence and appearance of the assembly listing: PRINT, TITLE, EJECT, and SPACE.

PRINT The PRINT instruction has three operands that can be in any order and in any combination. The major operand is coded ON or OFF and determines whether or not the listing will be printed at all, as:

```
PRINT ON
PRINT OFF
```

If ON is coded, or if the operand is omitted, the listing is printed as usual. If OFF is coded, no assembly listing is printed. When OFF is coded, the other operands and any other listing-control assembler instructions are ignored. The OFF operand is rarely used, but it can be used if you have made no changes to a source program since the last assembly and you need to assemble and test again.

A second operand controls the printing of macro-generated statements. If NOGEN has been coded, none of the statements generated by the expansion of a macro instruction are printed. If the operand has been omitted or coded GEN, all of these generated statements are printed on the assembly listing and denoted with a plus sign to the right of the statement number. Figure 11-7 depicts a typical use of the PRINT instruction with the GEN/NOGEN operand. Here, printing of macro-generated statements is turned off so the large number of statements generated by the DTFs will not be listed. Following the DTFs, printing of macro-generated statements is turned back on so the code generated by macros such as OPEN will appear on the listing.

The last PRINT operand controls the printing of the data generated by DC instructions and literals. If the operand DATA is coded, each byte of generated data is printed in the object code column of the assembly listing, eight bytes per line. If the operand is omitted, or coded NODATA, only the first eight bytes of the defined data appear on the listing.

The DATA operand is only occasionally used because programmers generally don't need to see all the data generated by DC instructions and literals. For some special cases, however, the programmer might want the DATA option on so that the data is visible. Figure 11-8 illustrates the use of the DATA option to print table definitions. The first PRINT instruction causes all the data in the table to print. The last PRINT instruction turns the option off again. The programmer can then use the assembly listing to proofread the table definition. The listing can also serve as documentation of the table contents.

TITLE A TITLE instruction causes a heading to be printed at the top of each page of an assembly listing as in this example:

```
TITLE 'PROG86 BILLING PROGRAM'
```

The TITLE instruction commonly appears as the first statement of a program, and the heading that prints is taken from its operand (the data enclosed in single quotation marks). The maximum length of the title is 100 characters.

If the TITLE instruction has a label, the first four characters of the label are punched into columns 73 to 76 of the object deck. In the example that follows, PG27 would be punched in the object deck, if one were produced:

```
PG27A     TITLE 'PROGRAM27, REVISION A'
```

EJECT The EJECT assembler instruction causes the assembler program to skip to the top of a new page before printing the next line of the assembly listing. The EJECT instruction has no operand, and labels are ignored.

SPACE Another assembler instruction, SPACE, allows you to cause one or more lines to be skipped in the assembly listing. The operand must be a decimal number that specifies the number of lines to be skipped. If no value is coded, one line is skipped. If the number of lines to be skipped is more than the number of lines remaining on the page, the effect of the SPACE is equal to EJECT.

OBJECT DECK CONTROL

The main use of the PUNCH and REPRO assembler commands is to create linkage-editor control cards. Normally, you would punch these linkage-editor cards separately on the keypunch and manually combine then with the object deck for input to the linkage-editor program. By using PUNCH and REPRO, however, these control cards can be punched, along with the object deck, by the assembler. In some cases, PUNCH and REPRO are used to punch a complete set of job-control cards so that no manual intervention is required before using the object deck for input.

The PUNCH instruction punches whatever appears as its operand (the data enclosed in single quotes). Column 1 of the resulting output card will contain the data in the first position of the operand that follows the opening single quote. For example:

```
PUNCH '    INCLUDE'
```

This statement would punch INCLUDE in columns 4 through 10 of the output card. (INCLUDE is a linkage-editor control card that goes before the object deck when it is being link-edited.)

A REPRO instruction uses the next statement in the program as its operand. It punches a duplicate of this statement as part of the object deck. For example:

```
      REPRO
INCLUDE
```

Here, the effect is the same as the PUNCH example—a card with INCLUDE punched in columns 4 through 10 precedes the object deck. The PUNCH or REPRO cards would usually be placed at the start of your source deck.

```
          .
          .
          PRINT DATA
TABLE     DS    0CL176
          DC    18CL4'10AA'
          DC    C'11AB'
          DC    C'12AC'
          DC    4C'16DD'
          DC    20C'20XX'
          PRINT NODATA
```

FIGURE 11-8 Use of DATA Option to Print Table Definitions

Terminology

address constant
adcon

Objectives

1 When needed, specify and load multiple base registers with the USING instruction, the LM instruction, and adcons.
2 Use any of the assembler commands in this topic in appropriate problem programs.

Problem

(Objective 1) Give the opening statements of a program that requires two base registers.

Solution

```
PROG1     START 0
BEGIN     BALR  3,0
          USING *,3,4
          L     4,BASEADR
          B     EXEC
BASEADR   DC    A(BEGIN+4096)
EXEC      .
          .
          .
```

Part 4

Part 4

Magnetic Tape Programming

This part consists of two chapters that can be studied any time after you complete part 2. Chapter 12 presents the magnetic tape concepts related to assembler-language programming; chapter 13 presents the assembler-language code and techniques for using tape files. If you have written programs with tape input and output in another language, or if you have taken a course that emphasized tape concepts, chapter 12 may be largely review for you.

12

Magnetic Tape Concepts

If you have had an introductory course or if you have coded tape operations in another language, you may already be familiar with most of the material in this chapter. If you think this might be the case, you may want to check the terminology and objective lists at the end of each topic to help you decide what material you need to cover. Topic 1 describes the characteristics of magnetic tape and tape equipment. Topic 2 describes some programming considerations related to tape operations.

TOPIC ONE Tape Characteristics The *magnetic tape* used in computer operations is a big brother of the tape used for home tape recorders. Magnetic tape is a continuous strip of plastic that is coated on one side with a metal oxide. For the System/360-370, this tape is one-half inch wide and usually comes in 250, 600, 1200, and 2400 foot lengths on plastic reels of varying size. Figure 12-1 pictures a reel of tape.

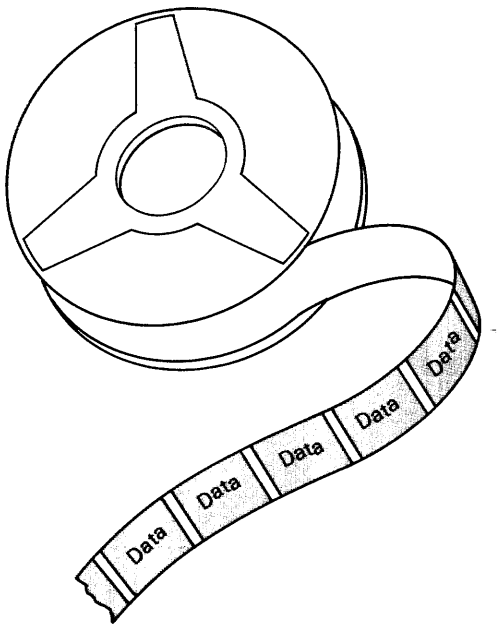


FIGURE 12-1 A Magnetic Tape

Data is recorded on the coated surface of the tape as patterns of magnetized spots. These spots correspond to the bits of the computer's internal storage. A succession of bits that are strung together form a *record*—that is, a collection of related data fields. For example, the data from a file of punched cards could be recorded on tape so that each tape record corresponded to an individual card. In such a case, each tape record would be 80 bytes long. If the last 15 columns of the cards weren't used, however, you might want to make the tape records only 65 bytes long. In general, tape records can be as short or as long as needed.

Each of the tape records is separated by a space on which no data is recorded. This space is called an *interrecord gap* (frequently abbreviated *IRG*). The gap is usually six-tenths of an inch long.

Figure 12-2 illustrates the normal recording mode used on System/360-370 tapes. There are nine vertical positions on a tape; each position represents a bit. The nine bits correspond to a byte of System/360-370 storage—eight data bits plus a parity bit. Because each vertical position contains nine bit positions, this magnetic tape is called a *nine-track-tape.*

Most System/360-370s use nine-track tapes, but some also have *seven-track-tape* capabilities. This means they can process tapes from older computer systems. Seven-track tapes are recorded in a code called *Binary Coded Decimal*, or *BCD*. In this code, seven bits are used to represent one character of data—six data bits plus one parity bit.

To ensure the accuracy of tape operations, two types of checks are used. First, the parity bit is used to make the number of on-bits in each byte of data an odd number. If a byte is read with an even number of on-bits, an error is detected. This type of checking can be called *vertical parity checking.*

The second type of check is performed in the horizontal direction. At the end of each tape record—the last byte before the interrecord gap—is a *horizontal*, or *longitudinal, check character.* The bits of this byte are set on or off to form a parity check for each of the horizontal tracks. In the case of a computer using odd parity, the bits of the check character would be set so the number of on-bits in each track is odd. If a tape is read that contains an even number of on-bits in one or more of the tracks, an input error is detected. Since this checking, called *horizontal parity checking*, is done automatically, the programmer generally does not concern himself with it.

Often in tape operations, more than one tape record is recorded between IRGs. This is called *blocking* records. Figure 12-3, for instance, shows how a *block* of five records would look on tape. Here, the *blocking factor* of the file is five, since five records (often called *logical records*) are stored in each tape block (often called a *physical record*). Because blocking is such a common practice, the IRG is often called an *interblock gap*, or *IBG*. Blocking is common because it increases the storage capacity of a reel of tape in addition to increasing the speed at which the records on the tape can be read or written.

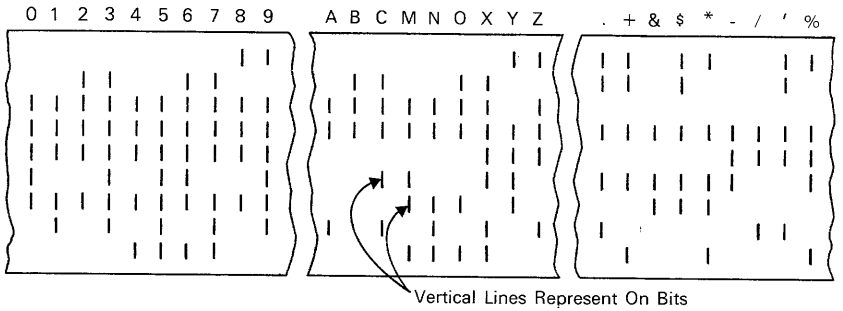


FIGURE 12-2 Coding on Tape

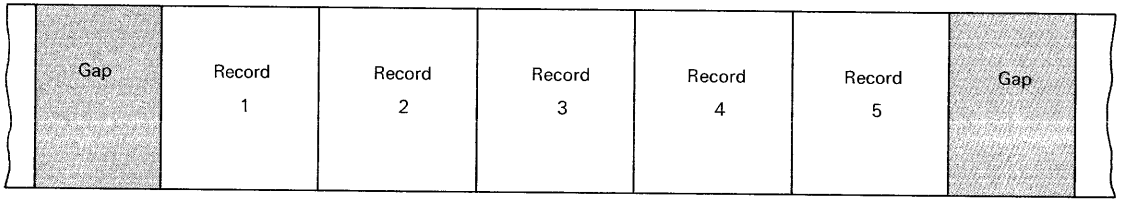


FIGURE 12-3 Blocked Records

THE TAPE DRIVE

Figure 12-4 illustrates one of IBM's 2400 series *tape drives*. Tape drives are the hardware units that read and write data on tape. To mount a tape, the computer operator threads the tape through a read/write mechanism in the center of the unit and then onto an empty takeup reel as shown in figure 12-5. The process is similar to mounting a tape on a home tape recorder.

Once the tape is mounted, the operator pushes the start button on the tape drive and the tape drive locates the first record on the file by searching for a *load-point marker* which is a reflective spot on the surface of the tape. Once the load-point marker has been located, tape records can be read or written under control of a stored program. When data is read from a tape, the data that is on the tape remains unaltered so it can be read many times. When data is written on the tape, it replaces (and thus destroys) any data that was on the tape. Before a tape is removed from the tape drive, the tape is rewound onto the original reel, so it is ready to be read or written on again.

Although the basic programmable functions of a tape drive are reading and writing records, there are a number of others. For example, most tape drives can be programmed to rewind the tape, to backspace the tape one block of records, and to skip ahead over faulty sections of tape. In addition, some tape drives can be programmed to read tape backwards, which in some applications can increase the speed of tape operations.

During reading operations, input records are checked for vertical and horizontal parity as discussed earlier. As a check on writing operations, output records can be checked immediately after being written because the reading

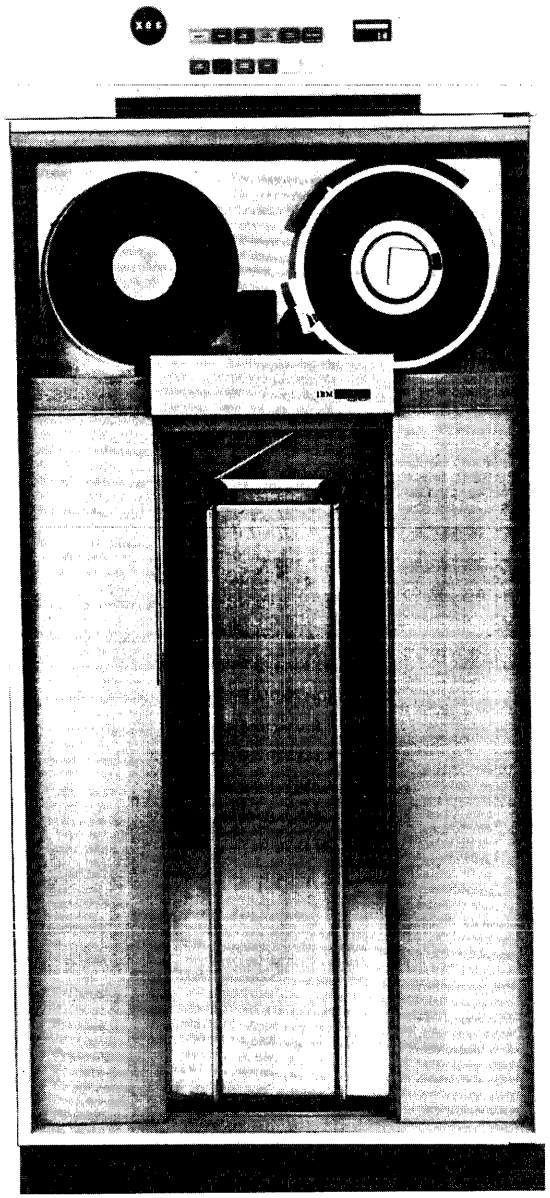


FIGURE 12-4 The Tape Drive

mechanism is located just after the writing mechanism. As soon as a character or block of records is written, it can be checked for vertical and horizontal parity by the reading mechanism.

Many System/360-370 systems use tape drives extensively, even though the primary I/O form in these systems is the disk. A medium sized system frequently includes two to six tape drives while larger systems may have a dozen or more.

TAPE SPEED AND CAPACITY

One measure of the speed of tape operations is the *transfer rate* or *transfer speed* of a tape drive. Transfer rate is measured in bytes per second and it measures the amount of time it takes to transfer data from the tape drive to storage or vice versa. For example, one common model of the 2400 series tape drive has a transfer rate of 60,000 bytes per second. (This speed is often referred to as 60 *KB*, where KB means thousands of bytes per second.) Other tape drives that are found on the System/360-370 have speeds ranging from 15 KB all the way up to 320 KB. To appreciate tape speeds, consider that a transfer rate of 80,000 bytes per second (80 KB) is the equivalent of reading 1000 80-column cards per second, or 60,000 cards—a stack 35 feet high—in one minute.

Transfer rate is somewhat misleading, however, because a tape drive actually stops and starts every time that it comes to an IBG. Transfer rate does not reflect this *start/stop time*. For several models of tape drives used on System/360-370, this start/stop time is 8/1000 of a second, or 8 milliseconds. To appreciate the effect of this starting and stopping between each physical record, suppose that a file of 6000 records, each 100 bytes long, were stored on a tape with a blocking factor of one. At 60 KB, it would take 10 seconds to read the data in the file (600,000 bytes at 60,000 bytes per second). However, the tape would also have to stop and start 6000 times. At 8/1000 of a second for each stop/start, this would take an additional 48 seconds. In other words, the tape drive spends 10 seconds reading the data and 48 seconds starting

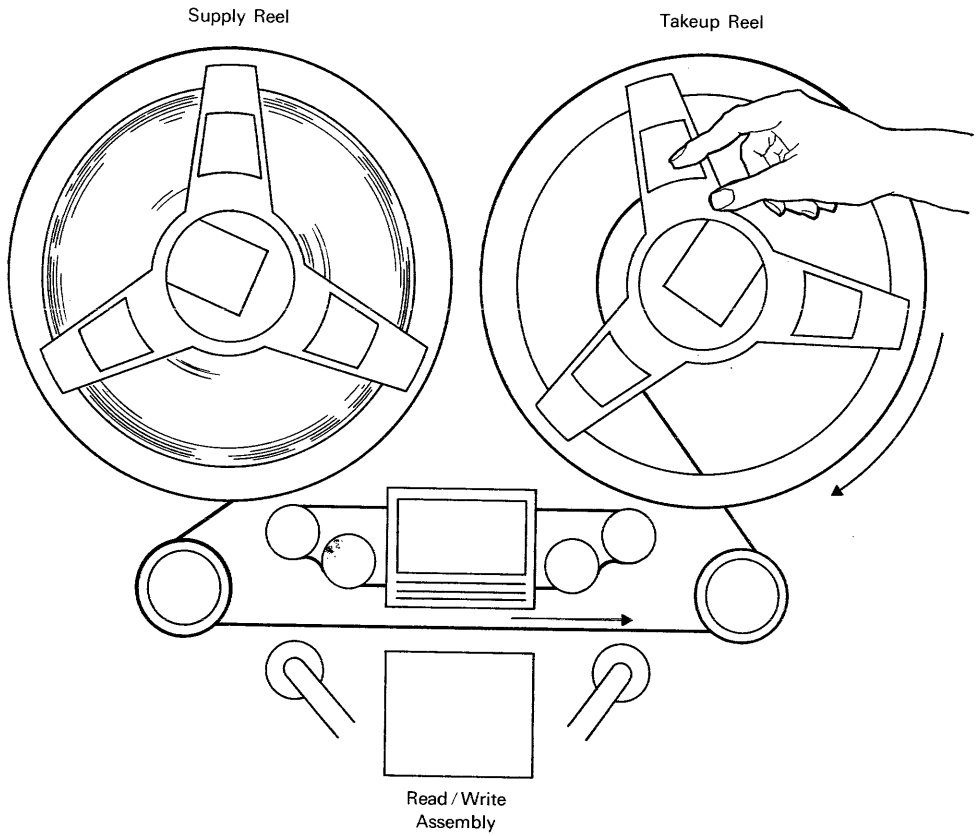


FIGURE 12-5 Mounting a Tape

and stopping. The effective processing rate is therefore much less than 60 KB.

Now, suppose the records are blocked with a blocking factor of ten. Ten seconds are still required for reading the 600,000 bytes of data, but only 4.8 seconds are required for starting and stopping. Since the total time for reading the file is reduced from 58 seconds to 14.8 seconds, you can see the effect of blocking on the speed of tape operations.

The capacity of a reel of tape depends on the length of the tape and the *density* of the tape. Density is a measure of the number of bytes of data that can be recorded on one inch of tape. The most common models of tape drives used on the System/360 use a density of 800 *bpi* (bytes per inch). At this density, an 80-byte record requires one-tenth of an inch of tape. Some System/360 and most System/370 installations use 1600 bpi tape drives. On these drives an 80-byte record takes only one-twentieth of an inch.

Blocking also affects the capacity of a tape. I'll use a sample file of 8000 records, 100 bytes each, to illustrate the effect of blocking. At 800 bpi, the 800,000 bytes require 1000

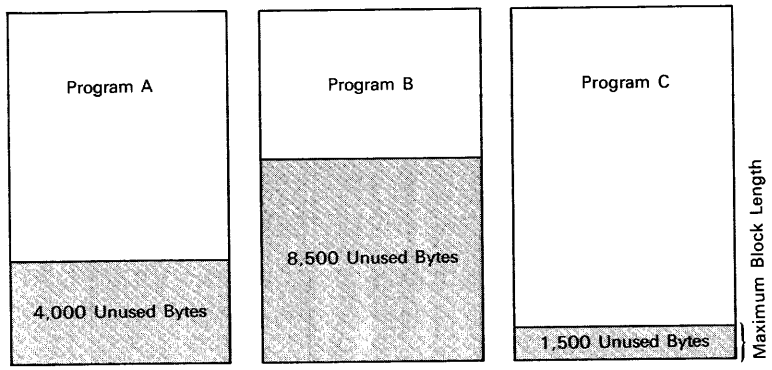


FIGURE 12-6 Determining the Blocking Factor

inches of tape. With a blocking factor of one, the 8000 IRGs (at 6/10 inch) take 4800 inches of tape. The total file then uses 5800 inches of tape. If the blocking factor is increased to ten, however, only 800 IRGs are required. Then, only 480 inches of tape are required for IRGs, and the file is reduced from 5800 inches to 1480 inches.

How large can a blocking factor be? It depends on the storage capacity of the computer. When a block of data is read, all of the records between the two IBGs is transferred into storage. As a result, if a block of records consists of 4000 bytes of data, the program-input area must be 4000 bytes. The blocking factor of a tape file is usually set by a system designer after he has considered all of the programs for which the file will be input or output. The block length can only be as large as the input or output area of the longest program that uses the tape.

To illustrate, suppose a tape file is going to be used by three programs and 16,000 storage positions are available for the programs. Disregarding the input area for the tape file, program A requires 12,000 bytes, program B requires 7,500 bytes, and program C, 14,500 bytes. The maximum block length is therefore 1500 bytes. This concept is illustrated in figure 12-6.

Terminology

magnetic tape
record
interrecord gap
IRG
nine-track tape
seven-track tape
Binary Coded Decimal
BCD
vertical parity checking
longitudinal check character
horizontal check character
horizontal parity checking
blocking
block
blocking factor
logical record
physical record
interblock gap
IBG
tape drive
load-point marker
transfer rate
transfer speed
KB
start/stop time
density
bpi

Objective

Describe how the blocking factor for a tape file is determined. Specifically, name the factors that must be considered.

TOPIC TWO Programming Considerations Tape is a heavily used form of I/O in System/360-370 installations because tape is an inexpensive way to store large volumes of data. Since the data on tape is always processed sequentially and a tape file can be either input or output, but not both, the program logic used to process tape files is very much like that for card input and output files. However, tape processing presents some complications that

must be handled by a program. Some of the most important of these complications are error recovery, record blocking and deblocking, and label checking. Fortunately, the I/O macros and associated I/O modules that IBM supplies with System/360–370 perform these functions for the assembler-language programmer. You don't have to write code for these functions, but you must understand them in order to be able to code I/O macros properly.

ERROR-RECOVERY ROUTINES

When an error is detected during a tape-reading operation, the error may often be recovered by using an *error-recovery routine*. If, for example, a piece of dust or dirt on the surface of the tape caused the error, it may be brushed off as the tape passes through the reading mechanism. If the tape is backspaced and the record is reread, the data can be transferred to storage without error. In a typical tape routine like the one represented by the flowchart in figure 12-7, a tape is backspaced and reread 99 times before the program stops trying to recover the error. If the error still exists, a message is printed on the console typewriter and the program is canceled.

The same type of routine is used for a writing operation. If a writing error is detected, the tape is backspaced and the write instruction is tried again. After a number of retries, the program may skip a certain amount of tape—the equivalent of a long IRG—and try again. If the error persists, the routine may end the job.

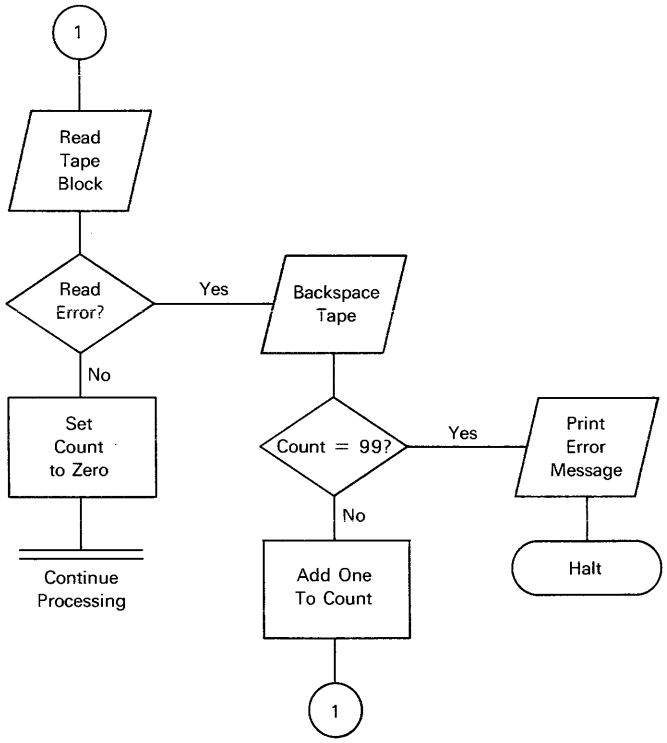


FIGURE 12-7 Error-Recovery Routine

BLOCKING AND DEBLOCKING ROUTINES

When a tape drive executes a read command, it reads an entire block of records into storage. The program, however, is usually written to process only one record at a time. *Deblocking routines* keep track of which record in a block is being processed and they issue read commands to the tape drive only when a new block of records is required.

For an example, consider a program written to process a tape of sales-transaction records that are blocked five records per block. The first time an input record is requested for processing, the deblocking routine issues a read command to the tape drive and a block of five records is read into storage. The first record is then passed to the processing routine. The next four times that the processing routine requests a transaction record, the deblocking routine merely passes the next one from the block in storage. Finally, when the sixth request is executed in the main program, the deblocking routine sees that it has processed all the records in the current block and reads the next block from the tape.

A similar situation exists when blocked output files are created. Each time the processing portion of the main program PUTs an individual record, a *blocking routine* moves

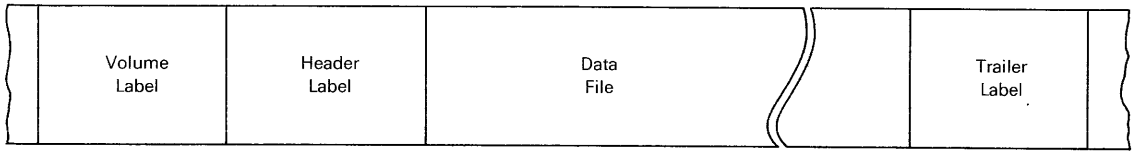


FIGURE 12-8 Labels for a Tape File

the record to the output area. Only when the entire block has been filled is an actual write command issued to the tape drive by the blocking routine.

LABEL-CHECKING ROUTINES

When a computer operator is given the instructions for running a job, he is told which reel of tape to mount on which tape drive. The reels of tape are identified by external labels that are attached to the outside of each reel. Suppose, however, that the operator makes a mistake. Suppose he mounts a tape containing current accounts-receivable records on a tape drive that is going to write a file of updated inventory records. If this mistake isn't caught, the accounts-receivable records will be destroyed.

To prevent this type of error, *internal labels*—labels that are actually records on the tape itself—are used. For example, System/360-370 tape files with *standard labels* contain the three label records shown in figure 12-8 (IBGs aren't shown in this illustration). The *volume label*, located immediately after the load-point marker, identifies the reel of tape. The *header label*, which contains information such as the filename, the date the file was created, and the date after which it is okay to destroy the file, identifies the file. The *trailer label*, located after the data records of the file, contains the same data as the header label plus a block count that indicates the number of blocks of data that the file contains.

These labels are processed by using a *label-checking routine* that compares the data the labels contain with data supplied by job-control cards when the tape file is opened for processing by the program. The job-control cards indicate which file should be mounted on which tape drive and what the volume and header labels for each file should be. You can find examples of header labels and related job-control cards in chapter 16.

To appreciate the value of label checking, consider the label processing that is done before writing an output file. First, the volume and header labels are read and analyzed. If the volume number agrees with the volume number given in the job-control cards and the expiration date is past, the routine backspaces the tape, writes a new header label for the output file, and begins processing. Otherwise, a message is printed to the operator indicating that he has mounted the wrong file, thus avoiding what might have been a costly error.

For input files, the header label is checked to make sure that the identifying information agrees with the information given in the job-control cards. At the end of the file, the block count in the trailer label is compared with a block count accumulated during processing. If the counts are the same, all the input blocks have been processed. If they are different, the routine prints a message to the operator to tell him that an error has occurred.

While standard labels are normally used in System/360-370 tape operations, it is also possible to process tape files that have either no labels or non-standard labels. If no labels are present, the checking process is skipped. If non-standard labels are used, the programmer must supply a routine to process them.

In some cases, a file of records will require more than one reel (volume) of tape. This is referred to as a

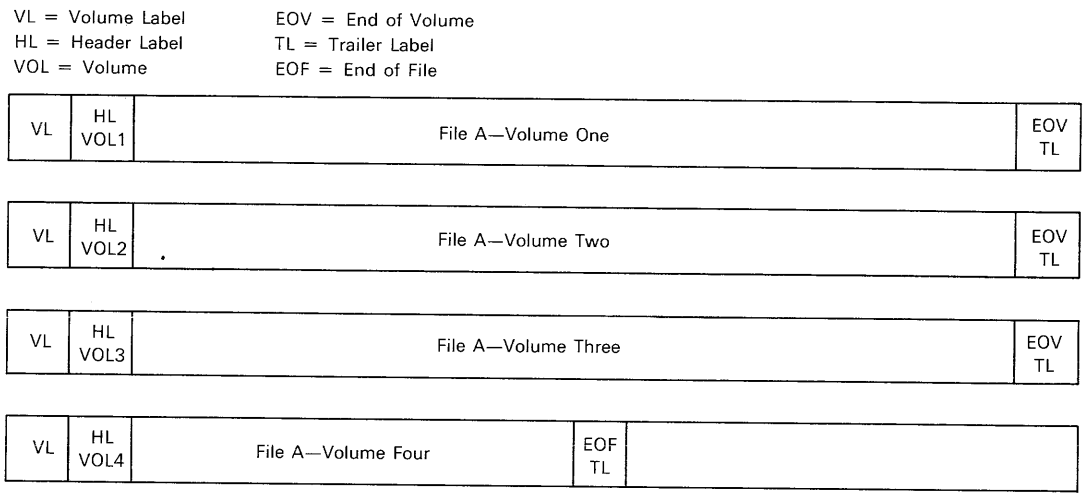


FIGURE 12-9 A Multivolume File

multivolume file and requires additional label-checking routines. When writing a multivolume file, the tape drive must check for a reflective spot near the end of each tape known as the *end-of-reel marker*. When this marker is encountered, the label-checking routines write a trailer label (called an *end-of-volume label*), that includes the block count for that reel of tape. The routines then check the labels on the next reel of tape. If the next reel is accepted for processing, a header label is written on that tape. This label contains a volume-sequence number indicating the order in which the reels of tape should be read. On the last reel of the multivolume file, the program writes a file trailer label (called an *end-of-file label*), that contains the block count for the reel. This switching from one reel of tape to another is called *tape switching*. A four-volume multivolume file and the associated labels are illustrated in figure 12-9.

For multivolume input files, the label-checking routines check the header label of each reel to see that the right file is being processed and that the reels are being processed in order. Thus, the first reel in the file must have a volume-sequence-number 1, the second must have sequence-number 2, and so on. If the wrong file has been mounted, a message is printed and the program halts. At the end of each reel, prior to tape switching, the program checks the block count in the trailer label against a block count accumulated by the program; this is a check to see that all blocks have been read.

One final aspect of label-checking routines concerns *multifile volumes* (*multifile reels*). These are reels of tape that have more than one file stored on them—for example, an inventory file, a billing file, and a sales-reporting file. In this case, each file is preceded by a header label and followed by a trailer label as shown in figure 12-10. When reading a file from a multifile volume, the label-checking routines must be able to scan the tape until the correct label is located. When writing a file in any but the first position of a multifile volume, the label-checking routines must be able to locate the correct position for the output file.

Even with label-checking procedures, it is possible to destroy a current file by writing other records on the same tape. It can happen when the warning message on the computer console is ignored by the computer operator. To

VL = Volume Label
HL = Header Label
TL = Trailer Label

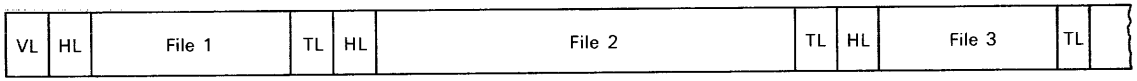


FIGURE 12-10 A Multifile Volume

provide further protection against this type of error, *file-protection rings* must be mounted along with a tape before the tape drive can write on the tape. The file-protection ring, illustrated in figure 12-11, pushes in a pin that is part of the tape drive. If the ring is not present, the tape cannot be written on and its data cannot be destroyed no matter what other errors take place.

WHAT THE PROGRAMMER MUST KNOW

Even though the IBM-supplied I/O macros and program modules contain the code necessary to perform all of these tape functions, the assembler-language programmer must supply several important items of information when he codes the DTF statement. You must specify the length of the tape records and their format. You must also supply the blocking characteristics and the label format—standard, nonstandard, or none.

If all the records in a tape file have the same number of bytes, the records, called *fixed-length records*, have a *fixed format*. These records can be blocked or unblocked. In addition, however, the records in a tape file can be of variable lengths. *Variable-length records* can also be blocked, resulting in variable-length blocks.

The format of a variable-length tape file is shown in figure 12-12. As you can see, each of the variable-length records is preceded by a four-byte record-length field. The number of bytes contained in the record, including the four-byte length field, is recorded as a binary value in the first two bytes. The second two bytes are reserved for use by the Disk Operating System and usually contain EBCDIC blanks. Another four-byte field at the start of the block specifies the total number of bytes in the entire block. This

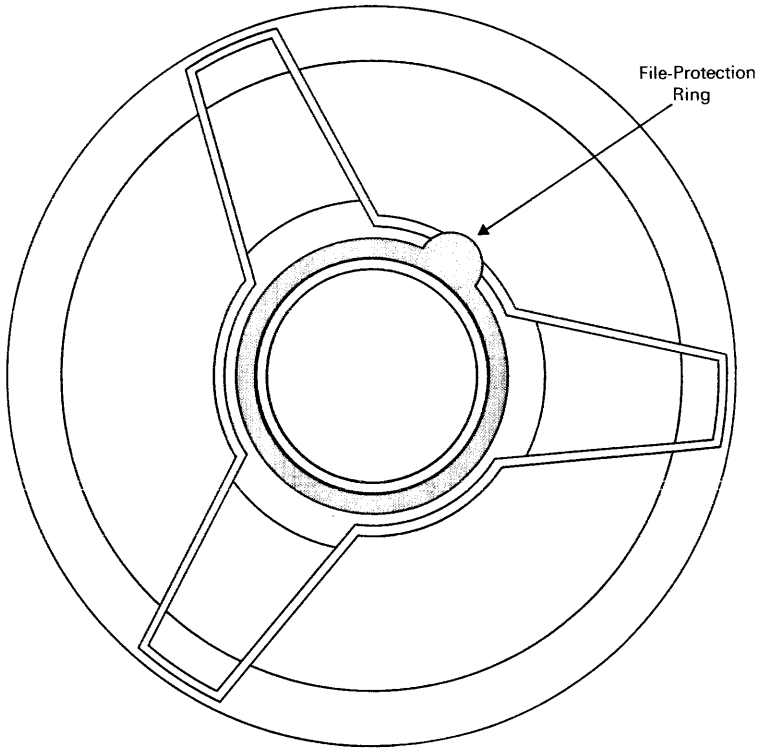


FIGURE 12-11 The File-Protection Ring

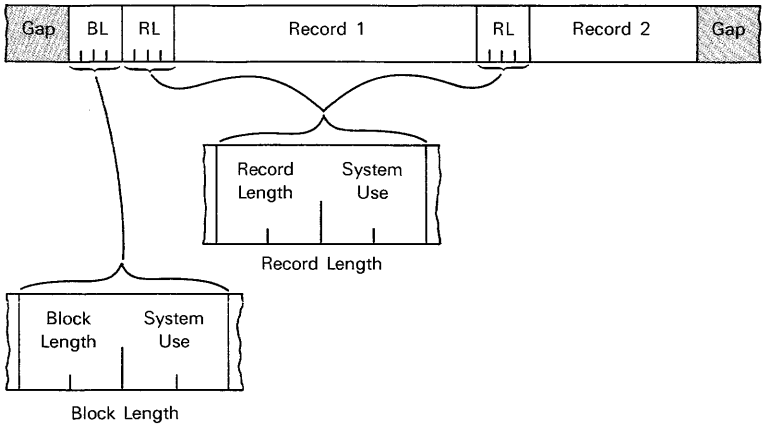


FIGURE 12-12 Variable-Length Format

field is in the same format as the record-length field: Bytes 1 and 2 carry the total block length—including the four-byte block-length field—as a binary value, and bytes 3 and 4 are reserved for system use. As a BAL programmer, you must know the exact formats of the variable-length records and blocks used in order to define their characteristics properly.

Terminology

error-recovery routine
deblocking routine
blocking routine
internal label
standard label
volume label
header label
trailer label
label-checking routine
multivolume file
end-of-reel marker
end-of-volume label
end-of-file label
tape switching
multifile volume
multifile reel
file-protection ring
fixed-length records
fixed format
variable-length records

Objectives

1 Describe the purpose of an error-recovery, blocking, deblocking, or label-checking routine.
2 Distinguish between the terms in each of the following pairs:
    fixed-length vs. variable-length records
    multifile volume vs. multivolume file

13

BAL For Tape Processing

Processing the most common types of tape files in assembler language is no more difficult than processing cards. You even use the same basic macros, GET and PUT, to perform the input and output of records. Instead of defining the file with a DTFCD, however, you code a DTFMT (magnetic tape) macro. Topic 1 of this chapter discusses the coding for fixed-length records; topic 2 deals with coding for variable-length records. Before starting this chapter, you should be familiar with the concepts and terms presented in chapter 12.

TOPIC ONE Fixed-Length Records There are 28 different keyword operands that can be used in a DTFMT macro. However, most of these operands are rarely used. In fact, for about 95 percent of all the tape programs I've seen or written, a subset of eleven operands does the job. This subset is summarized in

Priority	Keyword	Programmer Code	Default	Remarks
Required	BLKSIZE	Number of bytes in block		Must be same as length of I/O area.
Required	DEVADDR	SYSnnn		Logical unit for the tape file.
Required	FILABL	STD NSTD NO	NO	Specifies that the tape volume and file have standard (STD). non-standard (NSTD), or no (NO) labels.
Required	IOAREA1	Name of first or only I/O area		Length of I/O area must be large enough to hold entire tape block.
Optional	IOAREA2	Name of second I/O area		If IOAREA2 is coded, IOREG or WORKA must be coded.
Optional	EOFADDR	Label of first instruction of EOF routine		Required for input files.
Optional	TYPEFLE	INPUT OUTPUT	INPUT	Usually omitted for input files.
Optional	RECFORM	FIXUNB FIXBLK VARUNB VARBLK	FIXUNB	Specifies tape block format. Most common is fixed-length, blocked (FIXBLK).
Optional	RECSIZE	Record length in number of bytes		Used only when RECFORM=FIXBLK.
Optional	IOREG	Register number (nn)		Coded if records are processed in I/O area instead of work area. Address of record is placed in I/O register by I/O module. Omit WORKA operand.
Optional	WORKA	YES		Records processed in work area named in GET or PUT.

FIGURE 13-1 DTFMT Operand Summary

figure 13-1. If you find a need for some special processing later in your programming experience, you can consult IBM's manual, *DOS Supervisor and I/O Macros*, GC24-5037, for the complete operand list.

In general, the operands in figure 13-1 are used in the same manner as they are in DTFCD or DTFPR statements. BLKSIZE gives the length of the I/O area, IOAREA1 gives the name of the first I/O area, and EOFADDR gives the name of the instruction to be branched to when all of the records in a file have been read.

For the keyword DEVADDR, the programmer codes SYS followed by a three-digit number like SYS015. The number can be from 000 through 221 when using DOS, but usually there are standard assignments for each SYS number. Some examples of standard assignments are:

```
Tape drive 1=SYS008
Tape drive 2=SYS009
Tape drive 3=SYS010
Tape drive 4=SYS011
```

As a result, the programmer will use SYS008 for the first tape drive required by the program, SYS009 for the second,

```
TAPEPRT  START 0                                                          TPRT0010
TAPEIN   DTFMT BLKSIZE=500,DEVADDR=SYS008,EOFADDR=EOFTAPE,FILABL=STD,     XTPRT0020
               IOAREA1=T1,IOAREA2=T2,IOREG=(4),RECFORM=FIXBLK,            XTPRT0030
               RECSIZE=50                                                 TPRT0040
PRTOUT   DTFPR DEVADDR=SYSLST,BLKSIZE=132,IOAREA1=PRTAREA,PRINTOV=YES     TPRT0050
BEGIN    BALR  3,0                                                        TPRT0060
         USING *,3                                                        TPRT0070
         USING TAPEREC,4                                                  TPRT0080
         OPEN  TAPEIN,PRTOUT                                              TPRT0090
READTAPE GET   TAPEIN                                                     TPRT0100
         MVC   PITNUM,TITNUM                                              TPRT0110
         MVC   PITDES,TITDES                                              TPRT0120
         MVC   PONHAND,TONHAND                                            TPRT0130
         PUT   PRTOUT                                                     TPRT0140
         PRTOV PRTOUT,12                                                  TPRT0150
         B     READTAPE                                                   TPRT0160
EOFTAPE  CLOSE TAPEIN,PRTOUT                                              TPRT0170
         EOJ                                                              TPRT0180
T1       DS    CL500                                                      TPRT0190
T2       DS    CL500                                                      TPRT0200
PRTAREA  DS    0CL132                                                     TPRT0210
PITNUM   DS    CL5                                                        TPRT0220
         DC    5C' '                                                      TPRT0230
PITDES   DS    CL20                                                       TPRT0240
         DC    5C' '                                                      TPRT0250
PONHAND  DS    CL5                                                        TPRT0260
         DC    97C' '                                                     TPRT0270
TAPEREC  DSECT                                                            TPRT0280
TITNUM   DS    CL5                                                        TPRT0290
TITDES   DS    CL20                                                       TPRT0300
         DS    CL15
TONHAND  DS    CL5
         DS    CL5
         END   BEGIN                                                      TPRT0330
```

FIGURE 13-2 Tape-to-Printer Program

and so on. Until you learn the standard assignments for your system, begin assigning your SYS numbers with SYS008.

The FILABL operand tells the I/O modules what type of labels are found on the tape. You will normally use standard labels so STD should be coded. If nonstandard labels are used (NSTD), the programmer must use the LABADDR operand to give the name of the label-processing routine used in the program. This routine is usually written by the lead programmer and inserted into all programs that use the nonstandard labels.

The RECFORM operand gives the format of the tape records. Tape records can be blocked fixed-length (FIXBLK), unblocked fixed-length (FIXUNB), blocked variable-length (VARBLK), or unblocked variable-length (VARUNB). If the RECFORM operand is omitted, FIXUNB is assumed.

When processing fixed-length blocked records, either a work area (WORKA=YES) or an *I/O register* can be used. If a work area is used, the GET macro moves the next record to be processed into the work area. In contrast, if an I/O register is specified—for instance, IOREG=(4)—the records are processed in the I/O areas. In this case, the contents of the I/O register specified (one of the general-purpose registers) are continually adjusted by the blocking or deblocking routine so that the register addresses the record being processed. This technique is illustrated by the program in figure 13-2, which is explained below.

One of the first tape programs you are likely to write is

a *tape-to-printer program*. A tape-to-printer program reads an input tape and prints its contents in one form or another. Figure 13-2, for example, is a program that reads 50-byte tape records with this format:

BYTES	FIELD NAME
1-5	Item Number
6-25	Item Description
26-30	Unit Cost
31-35	Unit Price
36-40	Reorder Point
41-45	On-Hand
46-50	On-Order

These records are blocked with a blocking factor of ten, and the tape reading is to be overlapped with processing. Because the purpose of the program is to illustrate tape coding, the program simply lists the item-number, item-description, and on-hand fields.

Because register 4 is specified as the I/O register in the DTFMT statement, register 4 will always address the record being processed. The USING statement and the dummy section in the program make use of this fact:

```
          USING TAPEREC,4
          .
          .
          .
TAPEREC   DSECT
TITNUM    DS    CL5
TITDES    DS    CL20
          DS    CL15
TONHAND   DS    CL5
```

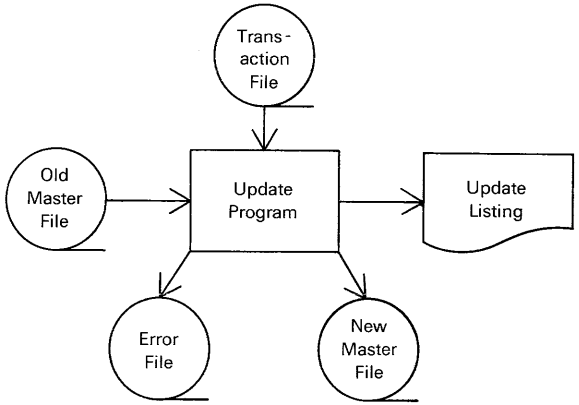
If register 4 contains the address of the first byte of the record, the names TITNUM, TITDES, and TONHAND can be used to address the fields in the record being processed.

To illustrate this more specifically, suppose T1 is located at address 8000 when the program is loaded and T2 is at 8500. When the first GET is executed, the contents of register 4 will contain address 8000. Then, the displacements taken from the dummy section will properly address the fields of the 1st input record. When the second GET is executed, the contents of register 4 will be increased to 8050 so the 2d record is addressed. This continues until the 11th record is being processed and register 4 contains address 8500, the address of T2. When the 21st record is being processed, register 4 contains 8000 again, the address of the first byte in T1. Thus, by adjusting the I/O register, the GET macro and the associated DTFMT automatically take care of the blocking and deblocking and the switching from one I/O area to the next.

What about label-checking routines? They are taken care of by the I/O macros too. For instance, the OPEN statement checks the volume and header labels to make sure the correct tape has been mounted on the correct tape drive. Similarly, the CLOSE macro takes care of checking and creating trailer labels, and rewinding files. If an input or output file requires more than one volume, the GET and PUT macros also process the end-of-volume labels at the end of one file and the volume and header labels at the start of the next.

AN UPDATE PROGRAM

Other than the use of an I/O register (which can be avoided by using a work area), tape input or output should present no problems. However, the logic of a tape program can be more complex than card or printer programs, because several tape files can be processed by a single program. To illustrate, consider a *tape-update program* such as the one presented in figure 13-3. As you can see by the system flowchart at the top of the figure, there are two input tapes—a transaction tape and a customer master tape—and two output tapes—an updated, or new, master tape and an error tape of unmatched transactions. (An *unmatched transaction* is one that doesn't have a master record with the same *control-field* number—in this case, customer number.) The program is also supposed to print an update listing of all matched transaction records as indicated by the print chart in figure 13-4. You might notice that a few of the tape



Processing Specifications:

1. Use transaction records to update master records. There may be none,one, or several transactions for each master, and both files have been sorted into customer-number sequence.
2. Print an update report with one line for each transaction record showing customer number, customer name, transaction date, and transaction amount.
3. Write a record on the error tape if an unmatched transaction is detected.
4. Print a total line at the end of the report showing the number of transactions processed, the number of unmatched transactions, and the sales total represented by the transaction records.

Customer Master Record Format (5 Records Per Block):

Position	Field	Length	Format
1–4	Customer number	4	C
5–24	Customer name	20	C
25–39	Street address	15	C
40–54	City	15	C
55–56	State	2	C
57–61	Zip Code	5	C
62–69	Date of last sale (MM/DD/YY)	8	C
70–74	Year-to-date sales (2 decimal places)	5	P
75–77	Number of transactions this year	3	P
78–90	Reserved for future use	13	

Transaction and Error Record Format (20 Records Per Block):

Position	Field	Length	Format
1–2	Transaction code (C'50')	2	C
3–6	Customer number	4	C
7–14	Transaction date (MM/DD/YY)	8	C
15–18	Sales amount (2 decimal positions)	4	P

FIGURE 13-3 Tape Update Problem

fields have packed-decimal format; this is legal for tape or disk records.

Figure 13-5 is a program flowchart for this update program. It represents a logic flow common to all tape-update programs. The flowchart starts by reading one record from the master file and one from the transaction file and relies on the fact that both files are in ascending control-field sequence. Perhaps the key to the flowchart is the decision in block 4. Here, the control field (customer number) of the master record is compared with the control field of the transaction record. One of three possible conditions will result from this comparison. (1.) If the control fields are equal (T = M), the transaction is used to update the old master record. (2.) If the transaction is

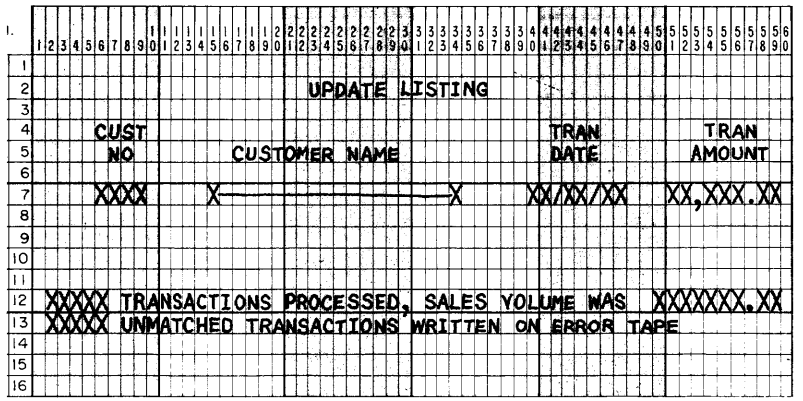


FIGURE 13-4 Print Chart for Update Listing

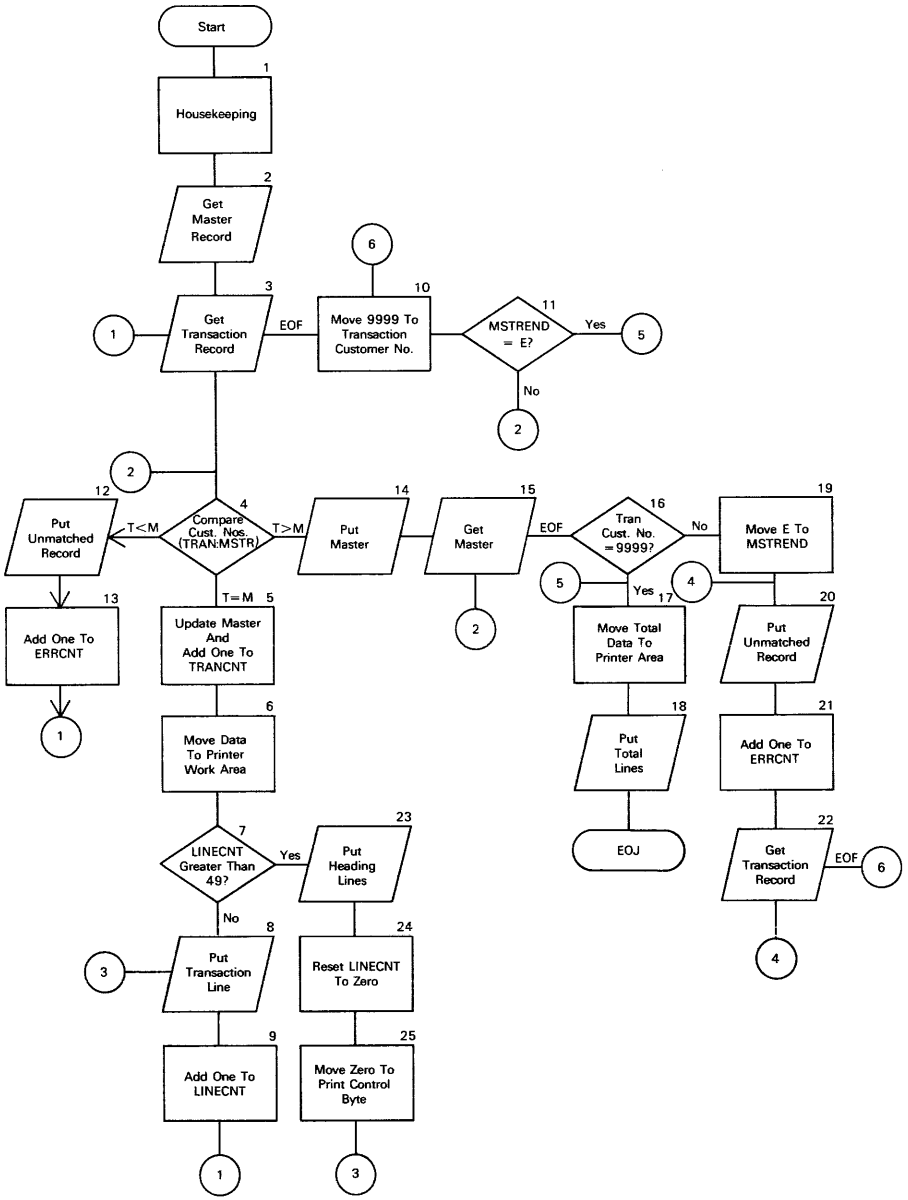


FIGURE 13-5 Tape Update Flowchart

```
TPUPDTE  START 0                                                                UPDT0010
TRANS    DTFMT BLKSIZE=360,DEVADDR=SYS008,EOFADDR=TRANEOF,FILABL=STD,          XUPDT0020
               IOAREA1=T1,IOAREA2=T2,IOREG=(5),RECFORM=FIXBLK,                  XUPDT0030
               RECSIZE=18                                                       UPDT0040
OLDMSTR  DTFMT BLKSIZE=450,DEVADDR=SYS009,EOFADDR=MSTREOF,FILABL=STD,          XUPDT0050
               IOAREA1=OM1,IOAREA2=OM2,RECFORM=FIXBLK,RECSIZE=90,               XUPDT0060
               WORKA=YES                                                        UPDT0070
NEWMSTR  DTFMT BLKSIZE=450,DEVADDR=SYS010,FILABL=STD,IOAREA1=NM1,              XUPDT0080
               IOAREA2=NM2,RECFORM=FIXBLK,RECSIZE=90,TYPEFLE=OUTPUT,            XUPDT0090
               WORKA=YES                                                        UPDT0100
ERRORS   DTFMT BLKSIZE=360,DEVADDR=SYS011,FILABL=STD,IOAREA1=E1,               XUPDT0110
               IOREG=(6),RECFORM=FIXBLK,RECSIZE=18,TYPEFLE=OUTPUT               UPDT0120
UPDLIST  DTFPR DEVADDR=SYSLST,IOAREA1=PRTAREA,BLKSIZE=133,CTLCHR=ASA,           XUPDT0130
               WORKA=YES                                                        UPDT0140
BEGIN    BALR  3,0                                                              UPDT0150
         USING *,3,4                                                            UPDT0160
         L     4,BASE2                                                          UPDT0161
         B     PROCESS                                                          UPDT0162
BASE2    DC    A(BEGIN+4096)                                                    UPDT0163
         USING TRANMAP,5                                                        UPDT0170
PROCESS  OPEN  TRANS,OLDMSTR,NEWMSTR,ERRORS,UPDLIST                             UPDT0180
         GET   OLDMSTR,MSTRWORK                                                 UPDT0190
READTRAN GET   TRANS                                                            UPDT0200
MATCH    CLC   TCSTNO,MCSTNO                                                    UPDT0210
         BE    UPDATE                                                           UPDT0220
         BH    TAPEOUT                                                          UPDT0230
         LA    7,READTRAN      LOAD RETURN ADDRESS FOR UNMATCHED ROUTINE        UPDT0240
         B     UNMATCHD                                                         UPDT0250
UPDATE   AP    TRANCNT,=P'1'                                                    UPDT0260
         AP    SLSTOTAL,TAMOUNT                                                 UPDT0270
         MVC   MLSTDATE,TDATE                                                   UPDT0280
         AP    MYTDSLS,TAMOUNT                                                  UPDT0290
         AP    MYTDTRNS,=P'1'                                                   UPDT0300
         MVI   DETLINE,X'40'                                                    UPDT0310
         MVC   DETLINE+1(132),DETLINE          BLANK PRINTER WORK AREA          UPDT0320
         PACK  TCSTNO,TCSTNO                                                    UPDT0330
         MVC   DCSTNO,PATTERN2                                                  UPDT0340
         ED    DCSTNO,TCSTNO+1                                                  UPDT0350
         MVC   DCSTNAME,MCSTNAME                                                UPDT0360
         MVC   DDATE,TDATE                                                      UPDT0370
         MVC   DAMOUNT,PATTERN1                                                 UPDT0380
         ED    DAMOUNT,TAMOUNT                                                  UPDT0390
         CP    LINECNT,=P'49'                                                   UPDT0400
         BNH   PRTDETL                                                          UPDT0410
         PUT   UPDLIST,HDGLINE1                                                 UPDT0420
         PUT   UPDLIST,HDGLINE2                                                 UPDT0430
         PUT   UPDLIST,HDGLINE3                                                 UPDT0440
         ZAP   LINECNT,=P'0'                                                    UPDT0450
         MVI   DCTL,C'0'                                                        UPDT0460
PRTDETL  PUT   UPDLIST,DETLINE                                                  UPDT0470
         AP    LINECNT,=P'1'                                                    UPDT0480
         B     READTRAN                                                         UPDT0490
TAPEOUT  PUT   NEWMSTR,MSTRWORK                                                 UPDT0500
         GET   OLDMSTR,MSTRWORK                                                 UPDT0510
         B     MATCH                                                            UPDT0520
```

FIGURE 13-6 Tape Update Program

```
* THE END-OF-FILE ROUTINE FOR THE TRANSACTION FILE FOLLOWS                    UPDT0530
                                                                             UPDT0540
TRANEOF  MVC   TCSTNO,=C'9999'                                               UPDT0550
         CLI   MSTREND,C'E'          HAS MASTER EOF BEEN REACHED             UPDT0560
         BE    PRTTOTAL              IF YES, PRINT TOTAL LINES               UPDT0570
         B     MATCH                 IF NO, MATCH TRANSACTION AND MASTER     UPDT0580
* THE END-OF-FILE ROUTINE FOR THE OLD MASTER FILE FOLLOWS                     UPDT0590
                                                                             UPDT0600
MSTREOF  CLC   TCSTNO,=C'9999'                                               UPDT0610
         BE    PRTTOTAL                                                      UPDT0620
MORETRNS BAL   7,UNMATCHD      BRANCH AND LINK TO UNMATCHD ROUTINE           UPDT0630
         GET   TRANS                                                         UPDT0640
         B     MORETRNS                                                      UPDT0650
* A SUBROUTINE FOR PROCESSING UNMATCHED TRANSACTIONS FOLLOWS                  UPDT0660
UNMATCHD MVC   0(18,5),TRECORD   MOVE TRAN RECORD TO ERROR OUTPUT AREA       UPDT0670
         PUT   ERRORS                                                        UPDT0680
         AP    ERRCNT,=P'1'                                                  UPDT0690
         BR    7                       BRANCH TO ADDRESS IN REGISTER 7       UPDT0700
* THE ROUTINE FOR PRINTING THE TOTAL LINES FOLLOWS                            UPDT0710
PRTTOTAL MVC   TL1TRNCT,PATTERN2                                             UPDT0720
         ED    TL1TRNCT,TRANCNT                                              UPDT0730
         MVC   TL1AMT,PATTERN3                                               UPDT0740
         ED    TL1AMT,SLSTOTAL                                               UPDT0750
         PUT   UPDLIST,TOTLINE1                                              UPDT0760
         MVC   TL2ERRCT,PATTERN2                                             UPDT0770
         ED    TL2ERRCT,ERRCNT                                               UPDT0780
         PUT   UPDLIST,TOTLINE2                                              UPDT0790
         CLOSE TRANS,OLDMSTR,NEWMSTR,ERRORS,UPDLIST                          UPDT0800
         EOJ                                                                 UPDT0810
* WORK FIELD DEFINITIONS FOLLOW                                               UPDT0820
PATTERN1 DC    X'4020206B20202143 2020'                                      UPDT0830
PATTERN2 DC    X'402020202021'                                               UPDT0840
PATTERN3 DC    X'40202020202020214B2020'                                     UPDT0850
LINECNT  DC    P'50'                                                         UPDT0860
TRANCNT  DC    PL3'0'                                                        UPDT0870
ERRCNT   DC    PL3'0'                                                        UPDT0880
SLSTOTAL DC    PL5'0'                                                        UPDT0885
MSTREND  DC    C'N'                                                          UPDT0890
* I/O AND I/O WORK AREA DEFINITIONS FOLLOW                                    UPDT0900
T1       DS    CL360                                                         UPDT0910
T2       DS    CL360                                                         UPDT0920
OM1      DS    CL500                                                         UPDT0930
OM2      DS    CL500                                                         UPDT0940
NM1      DS    CL500                                                         UPDT0950
NM2      DS    CL500                                                         UPDT0960
E1       DS    CL360                                                         UPDT0970
PRTAREA  DS    CL133                                                         UPDT0980
HDGLINE1 DS    0CL133                                                        UPDT0990
         DC    CL23'1 '                                                      UPDT1000
         DC    C'UPDATE LISTING'
         DC    96C' '
```

FIGURE 13-6 Tape Update Program (Continued)

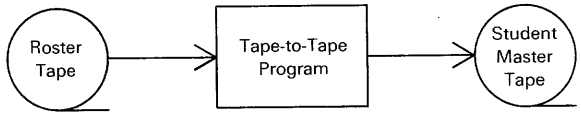
greater than the master (T > M), the master record is written on the new tape file and another master record is read. (3.) If the transaction is less than the master (T < M), an unmatched transaction is indicated and a record is written on the unmatched tape.

When an end-of-file (EOF) condition is detected for

```
HDGLINE2 DS    0CL133                                                  UPDT1010
         DC    C'0     CUST'                                           UPDT1020
         DC    32C' '                                                  UPDT1030
         DC    C'TRAN          TRAN'                                   UPDT1040
         DC    75C' '                                                  UPDT1050
HDGLINE3 DS    0CL133                                                  UPDT1060
         DC    C'       NO          CUSTOMER NAME'                     UPDT1070
         DC    12C' '                                                  UPDT1080
         DC    C'DATE        AMOUNT'                                   UPDT1090
         DC    74C' '                                                  UPDT1100
DETLINE  DS    0CL133                                                  UPDT1110
DCTL     DS    CL1                                                     UPDT1120
         DS    CL4                                                     UPDT1130
DCSTNO   DS    CL6                                                     UPDT1140
         DS    CL5                                                     UPDT1150
DCSTNAME DS    CL20                                                    UPDT1160
         DS    CL5                                                     UPDT1170
DDATE    DS    CL8                                                     UPDT1180
         DS    CL2                                                     UPDT1190
DAMOUNT  DS    CL10                                                    UPDT1200
         DS    CL73                                                    UPDT1210
TOTLINE1 DS    0CL133                                                  UPDT1220
         DC    C'-'                                                    UPDT1230
TL1TRNCT DS    CL6                                                     UPDT1240
         DC    C' TRANSACTIONS PROCESSED, SALES VOLUME WAS '           UPDT1250
TL1AMT   DS    CL11                                                    UPDT1260
         DC    73C' '                                                  UPDT1270
TOTLINE2 DS    0CL133                                                  UPDT1280
         DC    C' '                                                    UPDT1290
TL2ERRCT DS    CL6                                                     UPDT1300
         DC    C' UNMATCHED TRANSACTIONS WRITTEN ON ERROR TAPE'        UPDT1310
         DC    81C' '                                                  UPDT1320
MSTRWORK DS    0CL90                                                   UPDT1330
MCSTNO   DS    CL4                                                     UPDT1340
MCSTNAME DS    CL20                                                    UPDT1350
         DS    CL37                                                    UPDT1360
MLSTDATE DS    CL8                                                     UPDT1370
MYTDSLS  DS    PL5                                                     UPDT1380
MYTDTRNS DS    PL3                                                     UPDT1390
         DS    CL13                                                    UPDT1400
* THE DSECT FOR THE TRANSACTION RECORD FOLLOWS                         UPDT1410
TRANMAP  DSECT                                                         UPDT1420
TRECORD  DS    0CL18                                                   UPDT1430
         DS    CL2                                                     UPDT1440
TCSTNO   DS    CL4                                                     UPDT1450
TDATE    DS    CL8                                                     UPDT1460
TAMOUNT  DS    PL4                                                     UPDT1470
         END   BEGIN                                                   UPDT1480
```

FIGURE 13-6 Tape Update Program (Continued)

the old master file, the program tests to see whether the last transaction has also been read and processed. If it hasn't (the control field isn't equal to 9999), one or more unmatched transactions are indicated. As a result, the program writes the unmatched transaction on the error tape and reads another transaction. If the end-of-file for



Course Roster Tape Format (10 Records Per Block):

Positions	Field	Length	Format
1–4	Course Code	4	C
5–29	Course Title	25	C
30–31	Course Credit Hours	2	C
32–39	unused	8	
40–45	Student ID Number	6	C
46–70	Student Name	25	C

Student Master Tape Format (1 to 10 Course Segments):

	Positions	Field	Length	Format
ROOT	1–6	Student ID Number	6	C
SEGMENT	7–31	Student Name	25	C
	32–33	Total Credit Hours	2	P
COURSE	1–4	Course Code	4	C
SEGMENT	5–29	Course Title	25	C
	30–31	Course Credit Hours	2	P

Processing Specifications:

Create a variable-length student master tape from a fixed-length course-roster tape sorted in student-ID-number sequence. If an input record is out of sequence, print a message to the operator and end the program.

FIGURE 13-8 Variable-Length Tape-Creation Problem

read/write time can be saved by use of the variable-length format.

This type of record design—root segment plus a variable number of fixed-length segments—is also found in files that store a variable number of transactions along with some master data. A checking account record, for instance, might contain the account number, name, and other master data in the root which would be followed by one segment for each check written during the month. Each segment might contain the check number, amount, and transaction date.

To illustrate variable-length processing, figure 13-8 presents a typical problem. This problem is to create a student master file for a college. Each student master record has a root segment containing the student's ID number, name, and total number of credit hours being taken. Following the root is a variable number of segments that list the courses the student is currently taking. Each of these course segments contains the course number, title, and the credit hours for the course. Naturally, the total credit hours in the root segment should be the sum of the hours in the course segments. A student may be enrolled in only one course, or he may take up to ten courses.

This student master file is to be created from another tape file—a course-roster file. This roster file contains fixed-length records that were created from the enrollment cards turned in to the instructors at the start of the school year. These records are stored in course-code order and each contains the course code, title, credit hours for the course, student ID number, and name. For a course with 28 students enrolled there would be 28 records in the file.

Before the roster file is processed by the file-creation program, it is sorted into student ID sequence. Then, all of the course records for an individual student will be grouped together. First all the courses for the student with the lowest

ID number, then all the courses for the next student, and so on.

The flowchart in figure 13-9 illustrates the logic of this program. After the program reads the first course record from the roster file, it assembles the root segment and the first course segment for the first student master record. Then, as long as the following records are for the same student, additional course segments are added to that student master record. As each of these segments is added, the total-credit-hours field in the root segment is updated and the master-record length is increased by the course-segment length. When the first input record for the next student is read, the master record for the first student is written on the tape and the record assembly process starts over for the second student. If an out-of-sequence course record is read (which should never occur since the file has just been sorted), a message is printed for the operator and the program ends.

Figure 13-10 illustrates the file creation-program. There is little in this program that is really new to you, but let me point out a few things.

To begin with, the block size is determined by analyzing both the maximum record size and the most common record size. In this case, the maximum record size is 347 bytes:

```
  4 bytes—record length
 33 bytes—root segment
310 bytes—10 course segments
347 bytes
```

The most common record size (assuming an average of six courses per student) is 223 bytes:

```
  4 bytes—record length
 33 bytes—root segment
186 bytes—six course segments
223 bytes
```

I chose a block size of 704 bytes, because (1) it holds two maximum record lengths and (2) it holds three most common record lengths plus one additional course segment. Although the only BAL requirement is that the block size

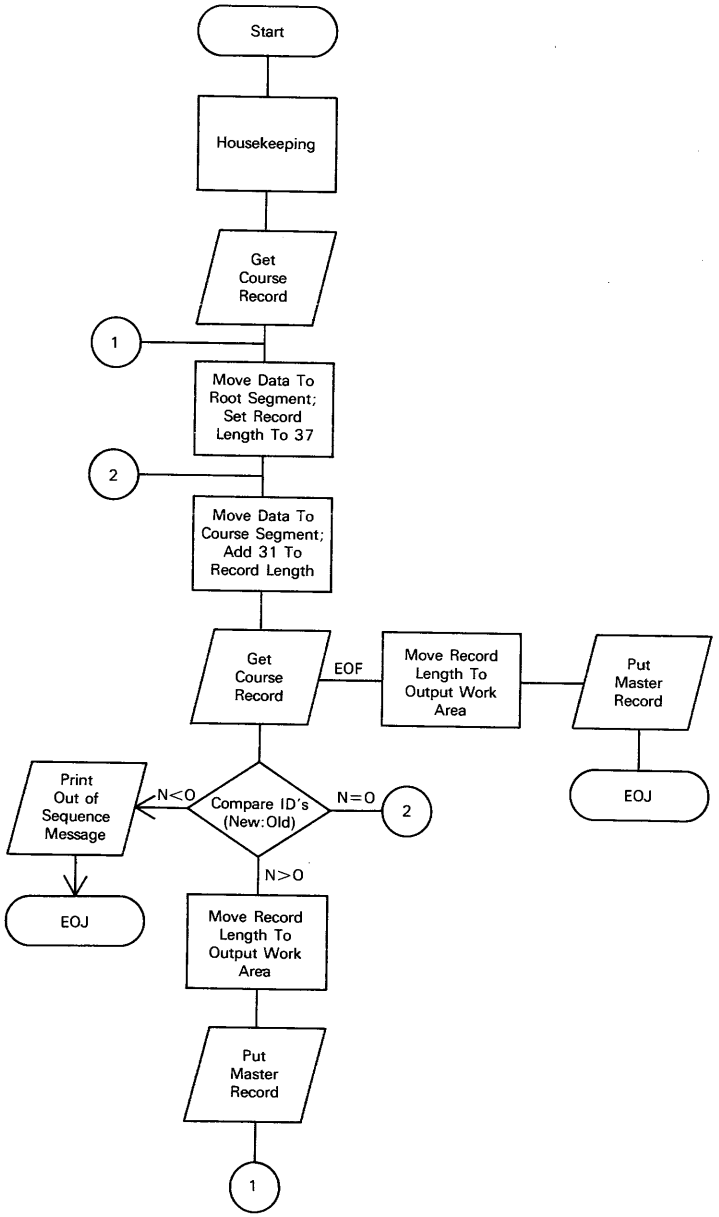


FIGURE 13-9 Program Flowchart—Variable-Length Tape Creation

```
TCREATE  START 0                                                          0010
COURSE   DTFMT BLKSIZE=700,DEVADDR=SYS006,EOFADDR=EOFCRSE,FILABL=STD,  X  0020
               IOAREA1=CIN1,IOAREA2=CIN2,IOREG=(6),RECFORM=FIXBLK,     X  0030
               RECSIZE=70,TYPEFLE=INPUT                                   0040
STUDENT  DTFMT BLKSIZE=704,DEVADDR=SYS007,FILABL=STD,IOAREA1=TOUT1,    X  0050
               IOAREA2=TOUT2,RECFORM=VARBLK,TYPEFLE=OUTPUT,WORKA=YES      0060
ERRMSG   DTFPR DEVADDR=SYSLST,IOAREA1=PRTOUT,BLKSIZE=132                  0070
BEGIN    BALR  3,0                                                        0080
         USING *,3                                                        0090
         USING CWORK,6                                                    0100
         USING SCOURSE,5                                                  0110
         OPEN  COURSE,STUDENT,ERRMSG                                      0120
         GET   COURSE                                                     0130
ROOT     MVC   SID,CID                                                    0140
         MVC   SNAME,CNAME                                                0150
         ZAP   SHOURS,=P'0'                                               0160
         LA    7,37            LOAD 37 IN REGISTER 7                      0170
         LA    5,SSEGAREA      LOAD THE ADDRESS OF THE 1ST SEGMENT IN R5   0180
SEGMENT  MVC   SCNBR,CCNBR                                                0190
         MVC   SCNAME,CCNAME                                              0200
         PACK  SCHRS,CCHRS                                                0210
         AP    SHOURS,SCHRS                                               0220
         LA    5,31(5)         INCREASE REGISTER 5 BY 31                  0230
         LA    7,31(7)         INCREASE REGISTER 7 BY 31                  0240
GETCRSE  GET   COURSE                                                     0250
         CLC   SID,CID                                                    0260
         BE    SEGMENT                                                    0270
         BH    SEQERR                                                     0280
         STH   7,SRECLEN              STORE LENGTH AND MOVE BLANKS INTO   0290
         MVC   SRECLEN+2(2),=X'4040' RECORD LENGTH BYTES OF MASTER AREA   0300
         PUT   STUDENT,SWORK                                              0310
         B     ROOT                                                       0320
SEQERR   PUT   ERRMSG                                                     0330
         B     CLOSING                                                    0340
EOFCRSE  STH   7,SRECLEN                                                  0350
         MVC   SRECLEN+2(2),=X'4040'                                      0360
         PUT   STUDENT,SWORK                                              0370
CLOSING  CLOSE COURSE,STUDENT,ERRMSG                                      0380
         EOJ                                                              0390
```

FIGURE 13-10 Variable-Length Tape-Creation Program

must be large enough for one maximum-length record plus the four bytes for block length, I chose a larger block size than necessary in order to get the speed and storage improvements that result from blocked data.

Actually, the block-size specification only represents the maximum size block that can be created. The actual sizes of the blocks will vary. As the master records are assembled in student ID sequence, the number of course segments per student varies, and thus the record size varies. As each record is PUT, the tape I/O module checks the space remaining in the block that it is currently building. If the new record fits the block without causing the 704-byte maximum to be exceeded, it is added to the block. If the new record makes the block greater than 704 bytes, the I/O module first writes out the current block at whatever length it may be, and then begins building the next block with the new record.

Suppose, for example, that each of the first four students in the input file is taking five courses. Since each of their master records would be 192 bytes long, the first three

```
PRTOUT    DS     0CL132                                                  0400
          DC     C'OUT OF SEQUENCE INPUT RECORD--PROGRAM ENDS'           0410
          DC     90C' '                                                  0420
          DS     0F        THIS FORCES BOUNDARY ALIGNMENT                0430
SWORK     DS     0CL347                                                  0440
SRECLEN   DS     CL4                                                     0450
SROOT     DS     0CL33                                                   0460
SID       DS     CL6                                                     0470
SNAME     DS     CL25                                                    0480
SHOURS    DS     PL2                                                     0490
SSEGAREA  DS     CL310                                                   0500
SCOURSE   DSECT                                                          0510
SCNBR     DS     CL4                                                     0520
SCNAME    DS     CL25                                                    0530
SCHRS     DS     PL2                                                     0540
CWORK     DSECT                                                         0550
CCNBR     DS     CL4                                                     0560
CCNAME    DS     CL25                                                    0570
CCHRS     DS     CL2                                                     0580
          DS     CL8                                                     0590
CID       DS     CL6                                                     0600
CNAME     DS     CL25                                                    0610
TCREATE   CSECT                                                          0620
CIN1      DS     CL700                                                   0630
CIN2      DS     CL700                                                   0640
TOUT1     DS     CL704                                                   0650
TOUT2     DS     CL704                                                   0660
          END    BEGIN                                                   0670
```

FIGURE 13-10 Variable-Length Tape-Creation Program (Continued)

records would be moved into the I/O area as they were PUT. At this point, the block would be 580 bytes long:

```
  4 bytes—block length
576 bytes—three master records
580 bytes
```

If the fourth 192-byte record were added to the block, the block would exceed 704 bytes. Instead, the I/O module would write out the 580-byte block and start a new block with the fourth record. You can see, then, that the actual length of the blocks would vary considerably. By analyzing the most common record size along with the maximum size, you can choose a block size that will allow larger actual blocks which will result in improvement of both processing speed and storage efficiency.

The second point to note is that I used a work area for the student master file rather than an I/O register. I did this because it's the simplest way to handle variable-length records. If I build the records in a work area, the I/O module will keep track of the space available in the block as each record is PUT, write out the block when required, and then begin building a new block. If I had elected to build the records directly in the I/O area, I would have had to code my own routine to provide that control. In general, then, use a work area when creating a file of blocked, variable-length records.

The definition of the variable-length work area itself is also of interest. The root segment of the record is defined in typical fashion. For the course segments, however, I decided to use a dummy section. Space for the maximum number of the 31-byte segments is reserved as part of the work area. Then, a DSECT is coded as a mask for that area. The base register for the DSECT is loaded with the beginning address of the course segment area when a root

section for a record is assembled. Then, after each course segment is assembled, the register is increased by 31 bytes for the next possible segment.

You should also examine the technique used to calculate the record length. Since the length must be stored in binary format, I have calculated it in binary to avoid unnecessary conversion. When a new root segment is assembled, I initialize register 7 to a binary value of 37 (33 data bytes plus the 4-byte record length). As each course segment is added to the record, the register is increased by 31 to reflect the length of the course segment. Then, just before each record is PUT, this accumulated length is stored in the first two bytes of the record-length field by a STH (store-halfword) instruction. The STH instruction works like a store instruction, but only the rightmost two bytes of the register are stored.

Because the store halfword instruction requires boundary alignment, I have preceded the variable-length work-area definition with this DS statement:

```
DS    0F
```

This causes the location counter to be moved to the next fullword boundary, although no storage is reserved because the duplication factor is zero. Thus, the STH instruction will work properly when the length is placed in the work area. The last two bytes of the record-length field are filled with blanks to conform to the standard variable-length format. Since block length is calculated and stored in the output block by the I/O module, the programmer does not have to worry about it.

The final point to note is the use of the CSECT statement in line 62. Because the roster-record DSECT follows immediately after the course-segment DSECT, there is no need to restart the location counter between them. After the roster-record DSECT, however, the CSECT is necessary to reestablish the location counter so the tape I/O areas are assigned proper storage locations.

Terminology

root segment

Objective

Given a problem involving variable-length tape records that are blocked or unblocked, code the BAL solution. The problem may involve one or more input or output tapes such as a tape-update program with an output error tape.

Problem

Suppose the student master tape created by the program in figure 13-10 is input to a tape-to-printer program. Write a program to print a listing of students and courses as indicated by the print chart in figure 13-11. Because the purpose of this program is to get you to use variable-length logic, don't bother to print the headings or use forms overflow.

Solution

Figure 13-12 is an acceptable solution.

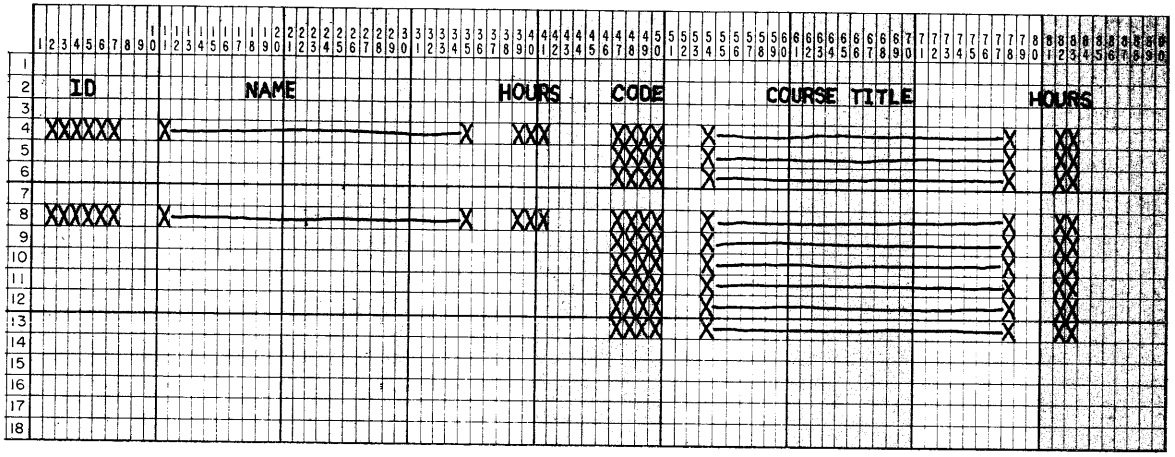


FIGURE 13-11 Print Chart for Variable-Length Tape-to-Printer Program

```
TAPTOPRT  START  0
BEGIN     BALR   3,0
          USING  *,3
          USING  SCOURSE,4              ASSIGN R4 AS BASE FOR DSELT
          OPEN   TAPE,LIST
READTAPE  GET    TAPE,STUDENT
          MVC    LSID,SID
          MVC    LSNAME,SNAME
          MVC    LSHOURS,HRSEDIT
          ED     LSHOURS,SHOURS
          LA     4,SSEGAREA             LOAD FIRST SGMT ADDR
          LH     5,SLGTH                LOAD R5 WITH RCD LGTH
          S      5,BCON37               SUBTRACT ROOT SGMT LGTH
NXTCRSE   MVC    LCNBR,SCNBR
          MVC    LCNAME,SCNAME
          MVC    LCHRS,HRSEDIT
          ED     LCHRS,SCHRS
          PUT    LIST,LLINE
          MVI    LLINE,X'40'
          MVC    LLINE+1(131),LLINE
          S      5,BCON31               SUBTRACT SGMT LGTH
          C      5,BCON0                IF RCD LGTH IS NOW ZERO, ALL
          BNH    READTAPE               SGMTS PROCESSED, GET NEW RCD.
          LA     4,31(4)                ELSE GO ON TO NEXT COURSE SGMT
          B      NXTCRSE
EOFTAP    CLOSE  TAPE,LIST
          EOJ
```

FIGURE 13-12 Variable-Length Tape-to-Printer Program

```
TAPE      DTFMT DEVADDR=SYS007,BLKSIZE=704,EOFADDR=EOFTAP,FILABL=STD,    X
                IOAREA1=TAPIN1,IOAREA2=TAPIN2,RECFORM=VARBLK,WORKA=YES
LIST      DTFPR DEVADDR=SYSLST,BLKSIZE=132,IOAREA1=PRTOUT1,              X
                IOAREA2=PRTOUT2,WORKA=YES
          DS    0H                        ALIGN WORK AREA ON H BOUNDARY
STUDENT   DS    0CL347
SLGTH     DS    H                         DEFINE RCD LGTH BYTES FOR CALCS
          DS    CL2
SID       DS    CL6
SNAME     DS    CL25
SHOURS    DS    PL2
SSEGAREA  DS    CL310
SCOURSE   DSECT
SCNBR     DS    CL4
SCNAME    DS    CL25
SCHRS     DS    PL2
TAPTOPRT  CSECT
LLINE     DS    0CL132
          DS    C' '
LSID      DS    CL6
          DC    3C' '
LSNAME    DS    CL25
          DC    C' '
LSHOURS   DS    CL5
          DC    3C' '
LCNBR     DS    CL4
          DC    3C' '
LCNAME    DS    CL25
LCHRS     DS    CL5
          DC    51C' '
HRSEDIT   DC    X'4020204B21'
BCON0     DC    F'0'
BCON31    DC    F'31'
BCON37    DC    F'37'
PRTOUT1   DS    CL132
PRTOUT2   DS    CL132
TAPIN1    DS    CL704
TAPIN2    DS    CL704
          END   BEGIN
```

FIGURE 13-12 Variable-Length Tape-to-Printer Program (Continued)

Part 5

Part 5

Direct-Access Programming

This part consists of two chapters that can be studied any time after you complete part 2. Chapter 14 presents the direct-access concepts related to assembler-language programming; chapter 15 presents the assembler-language code and techniques for using disk files. If you have a strong background in direct-access concepts for the IBM 2314 and 3330, chapter 14 may be largely review for you. Because the material in this part is difficult, you should be prepared to put a lot of effort into mastering it. Quite honestly, there are many professional programmers who have never mastered disk file handling at the level illustrated in chapter 15.

14

Direct-Access Concepts

A substantial majority of today's data processing systems are direct-access systems. Direct-access devices provide large amounts of storage with fast access to any of the records stored. Unlike sequential devices such as a card reader or tape drive, any record on a direct-access file can be read or written without having to read or write the records that precede it. To process the 500th record in a direct-access file, only the 500th record has to be read.

This chapter is divided into three topics. The hardware characteristics of two of the most common direct-access devices are presented in the first topic, and several ways in which records can be stored on and accessed from direct-access devices are described in the second topic. Then, topic 3 presents some programming considerations peculiar to direct-access devices. Because you may be familiar with some of the material in this chapter if you have written direct-

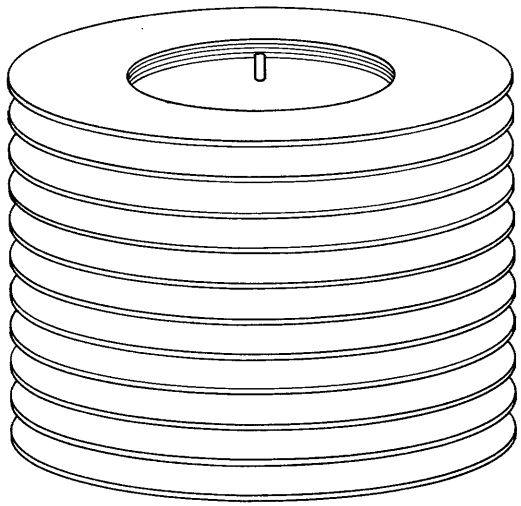


FIGURE 14-1 The Disk Pack

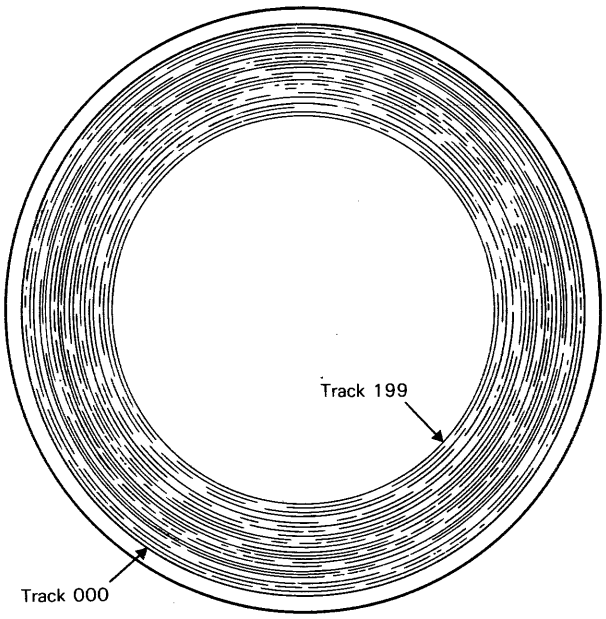


FIGURE 14-2 Tracks on a Disk Surface

access programs in some other language, you may want to look at the terminology and objective lists for each topic in order to decide which topics you need to read and which you can skip.

TOPIC ONE Direct-Access Devices An extremely high percentage of all System/360s are disk systems using the Disk Operating System or the full Operating System. While many of these systems do have tape drives, their primary storage units are disk devices, in particular, the IBM 2314 Direct-Access Storage Facility. The 2314 is by far the most widely used direct-access device.

Most System/370 installations are also disk systems. Many use 2314 devices, but the 3330 Direct-Access Storage Facility, another disk device, is the predominant device used on 370 systems. The 3330 is very similar to the 2314, but it is designed for the System/370 and offers greater storage capacity and faster access than the 2314.

THE 2314 DIRECT-ACCESS STORAGE FACILITY

The *disk pack* is the device on which data is recorded; the *disk drive* is the I/O unit that writes data on and reads data from a disk pack. The disk pack used with the 2314 disk facility (called the 2316 pack) is schematically illustrated in figure 14-1. It consists of eleven metal disks—14 inches in diameter— that are permanently stacked on a central spindle. When the disk pack is mounted on a 2314, it rotates at a constant speed of 40 revolutions per second while data is read from or written on it. When the disk pack is removed from the disk drive, a protective plastic cover is placed over it so it can be stored for later use.

Except for the top surface of the top disk and the bottom surface of the bottom disk, data can be recorded on both sides of the eleven disks that make up the 2316 pack, just as sound can be recorded on both sides of a phonograph record. As a result, this disk pack has a total of

20 usable surfaces, each of which has a magnetic surface coating on which data can be recorded.

On each of the twenty recording surfaces there are 200 concentric circles called *tracks*; they are illustrated in figure 14-2. These tracks are numbered from 000 through 199. Since there are 20 surfaces and 200 tracks per surface, the 2316 pack has a total of 4000 tracks on which data can be recorded. Although the tracks get smaller as they near the center of the disk, each track can hold the same amount of data, a maximum of 7294 bytes.

Data is recorded on a track in the form of magnetized spots, called *bits*. These bits, which correspond to internal storage bits, are strung together on a track so that eight bits make up one byte of data. To illustrate, suppose that figure 14-3 represents a portion of one track on one recording surface. If 0 represents an off-bit and 1 represents an on-bit, this portion of track contains three bytes of data. In EBCDIC code, the first byte, 11000001, represents the letter A; the second byte, 11110010, represents the digit 2; and the third byte, 01011011, represents the special character $.

The actual number of records on any track of a disk pack for the 2314 varies depending on the size of the records being stored. For example, one track can hold one 7294-byte record, two 3520-byte records, or three 2298-byte records. You can see from this that the capacity of a track decreases as the number of records on the track increases. If records are stored 1 per track, the track capacity is 7294 bytes; if 2 per track, the capacity is 7040 bytes (two times 3520); if 3 per track, the capacity is 6894 bytes; and so on. By the time you get to records that are 100 bytes long, the track capacity is only 36 records, or 3600 bytes.

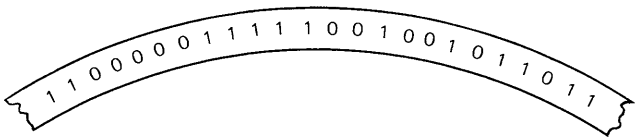


FIGURE 14-3 Coding on One Section of a Track

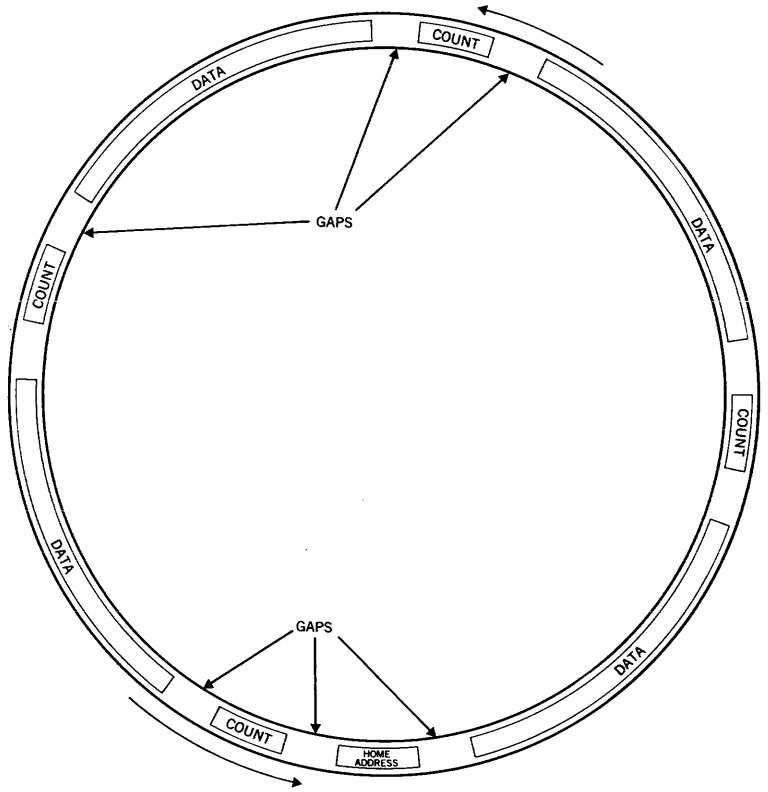


FIGURE 14-4 Count-Data Format

When records are stored on a disk pack, they may be stored in either of two track formats. The first, called the *count-data format*, is illustrated in figure 14-4. In this format, each record (*data area*) on a track is preceded by a *count area*. Since the illustration, which represents only one track, has four data areas, there are four count areas on the track. These count areas contain the *disk addresses* and lengths of the records following them. Just as a storage address identifies one and only one storage position, so a disk address identifies one, and only one, data area on a disk pack. By using the count area, each of the records on a disk pack can be directly accessed and read or written.

In addition to count areas and data areas, each track

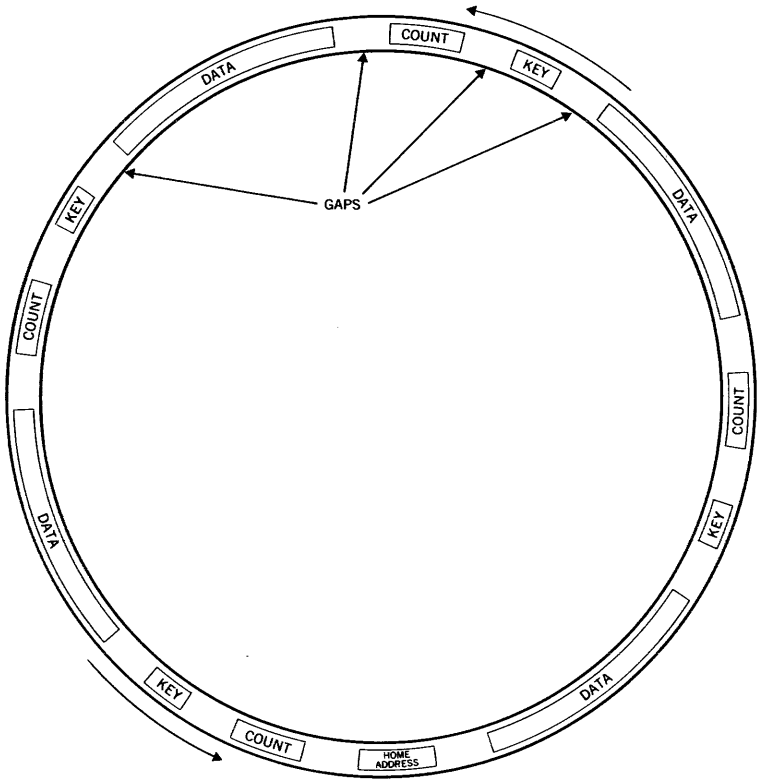


FIGURE 14-5 Count-Key-Data Format

in the count-data format has a *home address*. The home address, which comes immediately before the first count area on a track, uniquely identifies each of the tracks on a disk pack. On the 2316 disk pack, there are 4000 different home addresses, one for each of the 4000 tracks.

The second track format that may be used is called the *count-key-data format*. Like the count-data format, there is a home address at the start of each track. Unlike the count-data format, there is a *key area* between each count and data area. This is shown in figure 14-5. The key area, which may be from 1 through 256 bytes in length, contains the control data that uniquely identifies a record in a file. For example, in a file of inventory master records, the part

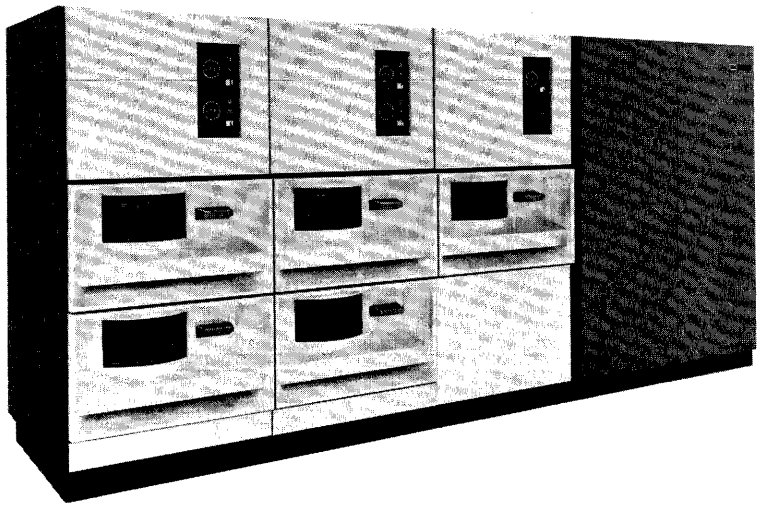


FIGURE 14-6 The 2314 Direct-Access Storage Facility with Five Spindles

number would logically be recorded in the key area. In a file of payroll master records, the employee number would be recorded in the key area. The difference between count and key, then, is that the count area contains a disk address that uniquely identifies a record location on the disk pack, the key area contains a control field that uniquely identifies a record in a file. As you will see later, both count and key areas can be used to locate records that are being directly accessed.

Because the count-key-data format has gaps separating the key from the count and data areas, the track capacity of this format is less than that of the count-data format. For example, with 1 record per track (one home address, one count, one key, and one data area), the track capacity is 7249 bytes. This 7249 bytes includes both key and data areas—say a 10-byte key and a 7239-byte data area—and contrasts the 7294-byte capacity of a track in count-data format. Similarly, with a 5-byte key and a 95-byte data area (a total of 100 bytes) only 29 records can be recorded per

track in contrast to the 36 records that are possible with the count-data format.

The 2314 disk facility contains from one to nine independent disk drives, called *modules*. In figure 14-6, for example, a facility with five disk drives, or modules, is illustrated. To mount a disk pack on one of the five drives, the operator pulls out the selected disk drive, places the disk pack on the drive's spindle, and pushes the module back into the facility. In a multi-drive device like this, a drive is often referred to as a *spindle*. Thus, the device in figure 14-6 can be called a 2314 with five spindles.

When the operator pushes the start button of the unit, the disk pack begins rotating until it reaches a speed of 40 revolutions per second. At this speed, the drive can read data from or write data on the recording surfaces. When it reads data, the data on the disk pack remains unchanged; when it writes data, the data that is written replaces any data previously stored there.

The mechanism that is used to read and write data on the 2314 is called the *access mechanism*. This mechanism, which is illustrated in side view in figure 14-7, consists of 20 *read/write heads*, one for each of the 20 recording surfaces. These heads are numbered from 0 through 19. Only one of the 20 heads can be turned on at any one time so that only one track can be operated upon at one time. Each of the heads can both read and write data but can only perform one operation at a time.

In order to operate on all 200 tracks of each recording surface, the access mechanism moves to the track that is supposed to be operated upon. When the access mechanism moves, all 20 heads move in unison. This means that 20 tracks can be operated upon in any one setting of the access mechanism. These 20 tracks are said to make up one *cylinder* of data. In other words, if the access mechanism is positioned at the 75th cylinder, the 75th track on each recording surface can be read or written. Since there are 200 tracks on each surface of the 2316 pack, there are 200 different settings of the access mechanism and 200 cylinders. In figure 14-7, the access mechanism is positioned at approximately the 65th cylinder.

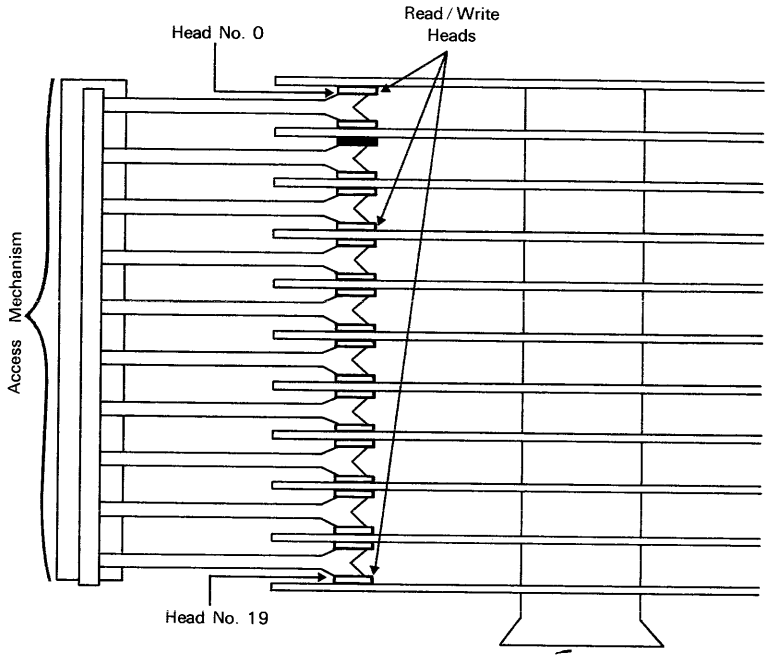


FIGURE 14-7 The Access Mechanism in Sideview

Any programming references to data records on disk are eventually reduced to cylinder number, head number, and either record number or key. The cylinders are numbered 0 through 199, the heads 0 through 19, and record number starts at 0 and goes through the maximum number of records that can be stored on the track. Because record-number 0 is used by the Disk Operating System, data records always start with record number 1.

When directly accessing and reading a record on a disk, there are four phases that the disk drive goes through. During the first phase, called *access-mechanism movement*, the access mechanism moves to the cylinder that is going to be operated upon. The time required for this movement depends on the number of cylinders moved. If the movement is just one cylinder—for instance, a move from the 25th to the 26th cylinder-25 milliseconds (25-thousandths of a second) are required. On the other hand, if the movement

is 80 cylinders—say, from the 10th to the 190th cylinder—the time is from 115 to 120 milliseconds, depending on the model used. In any event, the more cylinders that are moved, the more time access-mechanism movement takes. The average time required by access-mechanism movement when processing a file that uses all 200 cylinders of the disk pack is 75 milliseconds on one model of the 2314, 60 milliseconds on another.

Once the heads have been moved to the correct cylinder, the appropriate read/write head must be turned on. This is called *head switching*. If the track on the third recording surface is supposed to be read, head number 2 is turned on. In figure 14-7, head number 2 is blacked-in to indicate that this head is on (white heads are off). Since head switching takes place at electronic speeds, it has a negligible effect on the total amount of time required to read or write a record.

After the head has been turned on, there is a delay while the appropriate record rotates around to the head. This phase is called *rotational delay* (or *latency*). Since one complete rotation on the 2314 takes 25 milliseconds, the maximum time that rotational delay could take is 25 milliseconds. On the other hand, the appropriate record might just be reaching the head as the head is switched on. In this case, there is no rotational delay. Since rotational delay will vary between 0 and 25 milliseconds, the average delay is about 12.5 milliseconds.

The last phase in the process of accessing and reading a record is called *data transfer*. Here, data from the disk is transferred to storage in the CPU. On the 2314, data transfer takes place at 312,000 bytes per second, or 312 KB (*KB* means thousand bytes per second). At this rate, a 312-byte record requires 1 millisecond for data transfer.

When accessing and writing a record, the same four phases are completed. First, the access mechanism is moved; second, the appropriate head is turned on; third rotational delay takes place; and, fourth, the data is transferred from storage to the disk. In a reading or writing operation, access-mechanism movement and rotational delay are by far the most time-consuming phases.

Like other I/O devices on a computer system, the disk drive checks to make sure that reading and writing take place without error. Although I haven't mentioned it before, there are actually two cyclic check characters at the end of each count area, key area, and data area. These characters are used as a check on accuracy. They are calculated based on the combinations of bits used in the count, key, or data area during a writing operation. Then, when a record is read, the cyclic check characters are recalculated and compared with those that are read. If they don't agree, an input error is indicated.

A writing operation may be checked by using the write-verify instruction. When the write-verify instruction is executed following a write instruction, the data that has just been written is read and the cyclic check characters are checked as in a read operation. If there is a discrepancy, it indicates that the writing operation did not take place correctly in the first place. The write-verify is time-consuming, since one full rotation of the disk must be made before the record that is written can be read. Nevertheless, write-verification is often used when permanent files are being recorded.

The actual *I/O commands* (I/O instructions on the System/360-370 are called commands) that a 2314 disk can be programmed to execute are many. These commands can be broken down into five types: Seek, Search, Read, Write, and Write-Verify.

The *Seek* command causes the access mechanism to be moved to a specified cylinder and a specified head to be turned on. A typical *Search* command searches a track until it finds a count or a key equal to the one specified in the command. If the search doesn't find the specified key or count, the search may be continued on successive tracks in the cylinder.

Once the Seek and Search have been executed, a *Read* or *Write* may take place. This is the data-transfer phase of the operation. In a typical business program, either just data or data plus key is transferred during a Read or Write command. Following the Write command, a *Write-Verify* may be executed.

To illustrate the use of the commands, suppose the fifth record on the seventh track of the 120th cylinder must be accessed and read. The Seek command would specify that the access mechanism be moved to the 120th cylinder and the seventh head (head number 6) be turned on. Next, a Search command would compare the counts on the track with the count specified in the command. Since the count for a record indicates the cylinder number, head number, and record number on the track, the count for this record would indicate that it is the fifth record on the track. When the count in the command and the count on the track are equal, the Read command would be issued, which would cause the data area following the count to be read.

When using the count-key-data format, a slightly different set of instructions can be used. First, the Seek finds the selected cylinder and turns on the selected head. Second, the Search looks for a key on the track that is equal to the one specified in the command. When they are equal, a Read is issued thus transferring the data area following the selected key into storage.

THE 3330 DIRECT-ACCESS STORAGE FACILITY

The physical characteristics of the 3330 are very similar to those of the 2314. The 3336 disk pack, like the 2316 pack, consists of 11 disks with 20 recording surfaces. The 3330 drive, which consists of up to nine modules, spins this removable pack and reads or writes data through an access mechanism that carries the 20 read/write heads to the same position on each recording surface.

The 3330, however, offers significantly faster access-mechanism movement, faster data-transfer rate, and nearly four times the storage capacity of the 2314. Part of the increase in storage capacity is due to the doubled number of cylinders. Each recording surface on the 3330 is divided into 404 tracks instead of the 200 on the 2314. The other part of the increased storage capacity is due to the nearly doubled capacity of each data track—13,030 bytes on a fully used 3330 track as compared to 7294 bytes on a fully used 2314 track. Only 19 of the 20 tracks in each cylinder of a 3330 are used for storing data, however. The 20th track is used as a synchronizing track to help guide the 20 read/write heads.

Because the pack spins at 60 revolutions per second on the 3330 instead of the 40 per second on the 2314, rotational delay averages only 8.4 milliseconds on the 3330 compared to 12.5 on the 2314. Since the access mechanism also moves faster on the 3330, average access-mechanism-movement time is only 30 milliseconds on the 3330 versus 60 or 75 milliseconds on the 2314. Finally, since more data is passing under the heads at a faster rotational speed on the 3330, the data-transfer rate is increased to 806,000 bytes per second, or 806 KB.

A second model of the 3330 has recently been announced. The new model has twice as many cylinders as the basic model. This, of course, means the storage capacity of each pack is doubled. Otherwise, the operational features of the device are the same as the basic model 3330.

From a programmer's point of view, the 3330 is handled in the same manner as the 2314. The data formats are the same—count-data and count-key-data—and the I/O commands are nearly identical. However, the 3330 does support one additional function beyond the capabilities of the 2314. This function is *rotational-position sensing*. This feature allows the device to sense the position of the disk (relative to the beginning of the track) as it spins under the read/write head. This information can be used to alter the sequence of the I/O commands to improve the performance of disk operations.

To illustrate, suppose two commands are waiting to be serviced. The first requests that record number 8 on track 10 be read, the second requests that record number 2 on track 6 be written. The RPS (Rotational-Position Sensing) feature meanwhile tells the control unit for the disk drive that record number 11 is currently passing under head 10. This means that the disk must make almost one complete revolution before the first I/O command can be serviced. Instead, the control unit can decide to service the second command before the first because record number 2 on track 6 will be under the read/write heads before record 8 on track 10. Thus, both I/O operations can be completed in one revolution of

the disk instead of two. This feature can significantly improve the overall performance of disk operations.

At this time, the RPS feature can only be used with the full Operating System, not with DOS. Under OS, this feature is implemented by using a BAL macro. In the future, however, I would expect the feature to be made available under DOS and I would expect it to be handled automatically by the I/O modules so the BAL programmer won't have to worry about writing code to make use of it.

CONCLUSION

Besides the 2314 and 3330, there are other IBM direct-access devices. These include the 2311 disk, the 2301 drum, and the 2321 data cell. Although drums and data cells differ significantly from the 2314 and 3330 in physical characteristics, many of the same principles apply. For example, all the direct-access devices for the System/360-370 use the notion of cylinder and track in forming a direct-access address. In addition, the track formats used on drums and data cells can be either count-data or count-key-data—the same as on the 2314. From a programmer's point of view, then, the concepts are much the same, although the hardware may differ.

Terminology

disk pack
disk drive
track
bit
count-data format
data area
count area
disk address
home address
count-key-data format
key area
module
spindle
access mechanism
read/write head
cylinder
access-mechanism movement
head switching
rotational delay
latency
data transfer
KB
I/O commands:
- Seek
- Search
- Read
- Write
- Write-Verify

rotational-position sensing

Objectives

1 List and describe the physical operations required by the 2314 or 3330 in order to access and read or write a record from a file (1) in count-data format and (2) in count-key-data format.
2 List the I/O commands that must be executed in order to access and read or write a record on the 2314 or 3330.

TOPIC TWO File Organization Card readers and tape drives allow you to process records in only one way —sequentially. On a direct-access device, however, there are several possible ways of organizing a file. The three organizations supported by the Disk Operating System are: (1) sequential, (2) direct, and (3) indexed-sequential.

SEQUENTIAL FILE ORGANIZATION

Although direct-access devices were designed for directly accessing records, they may also store and process records sequentially. In fact, for some files, *sequential file organization* is the most efficient method. When writing records sequentially on a 2314, the first record of the file is stored in the first record position on the first track of the first cylinder of the file, the second record is stored in the second record position of the first track of the first cylinder, the third record in the third record position of the first track, and so on. When the first track of the first cylinder has been filled, the records begin to fill the second track of the first cylinder; this continues through the last track of that first cylinder. Then, the records start to fill the first track of the second cylinder of the file. The last record in a sequential file is always an *end-of-file record*, which is simply a count area indicating a data area with a length of zero. Figure 14-8 represents a sequential file that has been loaded on cylinders 20 through 25 on a 2314 disk. The records are 1332 bytes without keys so only five of them fit on each track. Notice that the end-of-file record follows the last data record even though not all of the disk space allocated for the file is used.

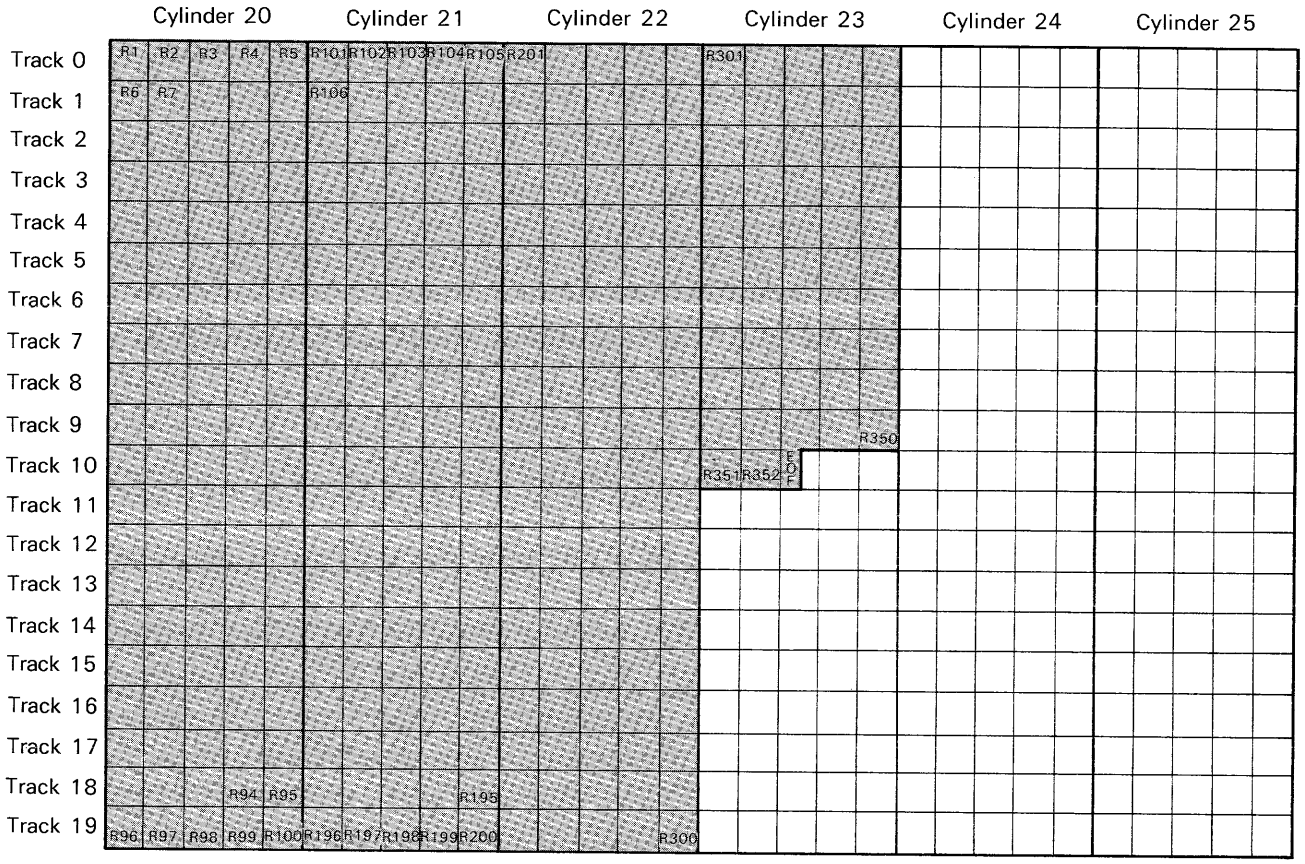


FIGURE 14-8 A Sequential File on the 2314

When reading records in a sequential file, they are read beginning with the first physical location of the file and continuing consecutively until the end-of-file record is read. Because the records in a sequential file are almost always processed sequentially, keys aren't needed and the count-data format is used.

To make efficient use of direct-access storage the records in a sequential file are usually *blocked*. This means that more than one record is read or written in a single Read or Write command. To illustrate, suppose a block consists of five 120-byte records. There will be 600 bytes in the data area following each count and five records will be read by one Read command.

Blocking is important because it reduces the time required to read or write a sequential file. With unblocked records, one Search, and therefore one rotational delay, is required for each record to be accessed. If the records are blocked, only one rotational delay is required for each block. When one Read command is executed, an entire block of records is read into storage. By eliminating rotational

delay, blocking can significantly reduce the time required to read the records in a sequential file.

Blocking can also affect the storage capacity of a disk pack. On a 2314 pack, for example, 7200 100-byte records take 200 tracks, or ten cylinders, if the records are unblocked. If they are blocked nine to a block, seven such blocks can be recorded on each track, a total of 63 records per track. Then, the entire file requires only 115 tracks—less than six cylinders.

DIRECT FILE ORGANIZATION

In *direct file organization*, the records on the direct-access device are in no particular sequence. When a program is ready to read or write a record, it must supply the information required to locate the record on the device. For direct files on the 2314 or 3330, the program must supply the cylinder and head number of the desired record before the Seek can be executed. And before the Search can be executed, either the record number or key of the desired record must be supplied. The trick in processing records in direct-file organization, then, is determining the disk address for each record that is going to be processed. This information is normally developed in a programming routine called a *randomizing routine*.

To illustrate the use of randomizing routines, suppose that 9600 inventory master records are going to be stored on a 2314. These records are going to be in count-key-data format with a key length of 7 bytes and a data area of 149 bytes. As you would guess, the key area for each record will contain the part number of the master inventory item.

Since 24 149-byte records with 7-byte keys can be stored on one 2314 track, this file requires 400 tracks, or 20 cylinders. However, because of the difficulty of assigning these records to a specific track and because additional master records may be added to the file later, 25 cylinders are allotted to the file. The file will be placed on one of the packs in the area from cylinder 60 through cylinder 84.

The problem of the randomizing routine for this file is to convert a record's part number, which may range from 100,000 through 9,000,000 to a specific cylinder and head number somewhere within cylinders 060–084. This data will be used in the Seek command when accessing a record. After the access mechanism has been moved to the selected cylinder and the selected head has been turned on, the Search command will locate the record on the track by searching for a key equal to the record's part number.

One common method used in randomizing routines is called the *division/remainder method*. In this method, the part number is first divided by the prime number closest to and less than the number of tracks allotted to the file (a prime number is a number such as 1, 2, 3, 7, and 11 that can only be divided evenly by one and itself). Since there are 500 tracks assigned to the file, the prime number 499 is used. If the part number is 254932, the quotient is 510 with a remainder of 442. In the division/remainder method, though, only the remainder of this first division is significant. This number, 442, gives the relative location of the track on which the record should be stored. Because the divisor is always 499 for this file, the remainder will always be from 000 through 498.

Once the relative-track location has been determined, the randomizing routine must convert it to a cylinder and head number. In the example, if the relative track, 442, is divided by 20 (the number of tracks in a cylinder), the remainder is a head number between 0 and 19. Then, if 60 is added to the quotient, the result is a cylinder number between 060 and 084. For the part number 254932, the cylinder number becomes 82 (442 divided by 20 equals 22; 22 plus 60 equals 82) and the head number becomes 2 (442 divided by 20 leaves a remainder of 2). Figure 14-9 summarizes the characteristics of the file and the randomizing routine used in this example.

When initially loading the inventory records on the 2314, each master part number is converted to a cylinder and head number as described. Then, the record is written in the first available location on the track indicated. If the track is filled (if it already has 24 records), additional programming routines are required. Two of the most widely used alternatives are (1) to write the record on the next track in

the cylinder, assuming that it has an available record location, and (2) to write the record in an *overflow area* somewhere else on the disk.

An overflow area is an area used for records that cannot be stored in the locations assigned to them by the randomizing routine. In the example of the inventory file, cylinder 085 could be used as an overflow area. Then, if there is no room for record number 254932 on cylinder 82, head 2, the record could be written in the first available location in cylinder 085.

When accessing and reading an inventory record from a 2314 file, the part number of the desired record is first converted to the cylinder and head number by the randomizing routine. Then, after the Seek locates the selected cylinder and head, the Search looks for a record with a key equal to the part number of the desired record. If it finds the specified key, the record is read into storage. If it does not find the key, programming routines corresponding to the routines used in loading the file are required. If, for example, an overflow area has been used, the program must seek the overflow cylinder and search for the specified record there.

Although the randomizing routine summarized in figure 14-9 creates a valid cylinder and head number for each record in the master file, it may not be the best randomizing method available. One major problem is that between 100,000 and 9,000,000 there are 17,836 possible part numbers that will randomize to the same cylinder and track as part number 254932—but only 24 records will fit on a track. If a large number of the part numbers actually randomize to the same cylinder and track as part number 254932, a complicated and probably inefficient overflow-control routine must handle the situation. Although the division/remainder technique may work well for files in which the keys are spread evenly through their range, it usually takes some experimentation with real data to be sure the technique works satisfactorily. Most often, a system analyst or programmer will develop a randomizing routine that is tuned to the actual keys found in a file and the actual disk addresses to be used for the file.

When records in a direct file are blocked, additional programming routines are required. To load a blocked file, the records have to be randomized to a block of records rather than to an individual storage location. Overflow records for each block might be stored in the next available block or in overflow areas. Because blocking is likely to improve the use of the disk storage but decrease the speed

A Direct-Access File on The 2314

Characteristics of the file:

1. A file of 9600 inventory records is supposed to be stored in cylinders 060 through 084.
2. Each track can hold a total of 24 inventory records, which consist of 149-byte data areas and 7-byte keys.
3. The part numbers, which are the keys, range from 100,000 to 9,000,000.

Problem of randomizing routine:

To convert the part numbers of the 9600 master records to cylinder and head numbers within the 25 cylinders assigned to the file.

The division/remainder method:

1. Divide part number by 499—the remainder is the relative track, always between 0 and 498.
2. Divide the remainder by 20—the remainder is the head number.
3. Add 60 to the quotient in the second division giving cylinder number.

Examples of randomizing:

Part No.	*Cylinder No.*	*Head No.*
100,000	70	0
254,932	82	2
794,210	75	1
1,048,342	82	2
9,000,000	61	16

Loading the records on the 2314:

1. Convert the part number to cylinder and head number.
2. Seek cylinder and head number.
3. Search for next available location on the track.
4. Write the record in the first available location.

(If the entire track is filled, the record must be written in successive tracks of the cylinder or in an overflow area of the disk.)

FIGURE 14-9 Direct File Organization

at which the records are accessed, the use of blocking for direct files depends on such considerations as the total number of records to be stored, the available storage, and how frequently the records will be processed.

There are two reasons why direct files aren't popular for actual use. First, it is usually difficult to develop an efficient randomizing routine. Second, the programmer must code file maintenance routines that keep track of available record spaces when records are added to or deleted from a file. These routines can be very complex. Together these two difficulties make direct files tough to use.

INDEXED-SEQUENTIAL FILE ORGANIZATION

Although sequential and direct file organizations have their advantages, they also have their limitations. For example, while a blocked sequential file may make maximum use of the storage capacity of a direct-access device, it has many of the limitations of a tape file. To update a sequential file, all of the records in the file must be read rather than just those affected by transactions and the entire file has to be rewritten in order to add a record to the file. On the other hand, while direct-file organization allows a record to be rapidly accessed, it wastes storage capacity. (Remember the example where 25 cylinders are assigned to a file consisting of 20 cylinders of data.) In addition, a direct file must usually be sorted into sequential order before a sequential report can be prepared from it.

Indexed-sequential file organization is designed to allow both sequential and direct (or random) processing. (*Random processing* refers to the fact that records are not processed in any particular sequence.) Using indexed sequential organization, the records in a file are stored on the direct-access device so that they can be read sequentially, but *indexes* are kept so that any record can be read randomly by looking up its location in the indexes. If records are added to the file, an additional disk area called an overflow area is used; this makes it unnecessary to rewrite the entire file which would have to be done in sequential organization. When a record is stored in the overflow area, the indexes are changed so that the records can still be processed in sequence.

The indexes for an indexed sequential file are kept in three levels: a *master index*, a *cylinder index*, and *track indexes*. The master index is optional and is normally used only for very large files. Let me skip it for now and I'll come back to it.

The cylinder index is used to determine what cylinder a record is located in. To illustrate, suppose a file of master customer records is stored in cylinders 11–15 of a disk pack and the cylinder index is kept on the first track of cylinder 10. Since there are five cylinders used in the file, there will be five records in the cylinder index, always in the count-key-data format. The key in each index record contains the highest customer number stored in each cylinder of the file and the data area indicates what cylinder the records are in. For example, a cylinder index might contain the following data:

KEY	DATA
1949	C11
3241	C12
5972	C13
7566	C14
9840	C15

To find the record for customer 6500, a program would search the cylinder index until a higher or equal key is found. It's much like looking a value up in a table except that disk records are being read instead of a table in storage. By searching the cylinder index above, the program would find that the record with key 6500 is located on cylinder 14. (C is used here to indicate that the number in the data area is a cylinder number.)

Here's where the optional master index comes in. When a file is large and occupies many cylinders, the cylinder index may get to the point where it takes up more than one or two tracks. Since it is searched sequentially, the amount of time required to find a higher or equal key might be excessive. In such cases, you can elect to use a master index. By searching the master index first, you can go

directly to the proper track of the cylinder index instead of searching it from the beginning. The need for a master index depends on the amount of disk space that a file will occupy. This, of course, is influenced by the key length, record length, number of records in the file, and blocking factor.

In general, a master index is rarely used. If one is used, it is likely to be found in the same cylinder as the cylinder index. For instance, track zero of cylinder 10 might contain the master index and tracks 1-3, the cylinder index.

Once the cylinder index has directed the program searching for record 6500 to the proper cylinder of the file—in this example, cylinder 14—it must find the proper track. The same search technique is used to search the track index of cylinder 14 for a high or equal key. A track index is found on the first track of each cylinder of the file. These index records indicate the highest customer record contained on each track of the cylinder. For example, the track index for cylinder 14 might contain the following data:

KEY	DATA
6198	T2
6258	T3
6322	T4
6398	T5
6449	T6
6570	T7
6609	T8
6701	T9
6813	T10
6893	T11
6979	T12
7053	T13
7119	T14
7200	T15
7303	T16
7471	T17
7566	T18

By searching this index, the program can determine that record 6500 is on track seven of the cylinder (T is used to indicate that the number in the data area is a track number).

Once the program has determined the track number, it can find a record by searching for a key equal to the control number in the record—in this case, customer number 6500. Because the search is always for key rather than count, indexed sequential files must always be in the count-key-data format.

If you wonder why the track index in the above example only indicates 17 tracks (tracks 2-18), remember that the first track is used for the track index. If there is room left over on this first track, it can be used for storing data records, in which case, there will be an index for track 1 also. In this example, it is assumed that the index covers the entire first track.

As for the 19th and 20th tracks of the cylinder, they are used when records are added to the file. They are called a *cylinder-overflow area*. (Tracks 2 through 18, make up the *prime data area* of the cylinder.) The number of tracks assigned to the overflow area is based on the number of records likely to be added to a cylinder.

An indexed sequential file can also have a *general overflow area*. This area—usually located at the end of a file area—consists of one or more cylinders reserved for additions to the file. For instance, cylinder 16 might be the general overflow area for the file just described.

To determine whether a file should have cylinder-overflow areas, a general overflow area, or both, the system analyst or programmer must analyze how additions to a file are likely to be grouped. If no additions will ever be made to the file, both cylinder and general overflow areas can be omitted. If the additions are likely to be spread evenly throughout the file, it is probably best to assign one, two, or three tracks per cylinder for overflow and allocate a small general overflow area. On the other hand, if the additions will be bunched in small ranges of the record keys, it is more efficient to have no cylinder-overflow area and have all the additions go into the general overflow area.

An indexed sequential file, then, is made up of several different disk areas: the optional master index, the cylinder index, the prime data area including track indexes, prime

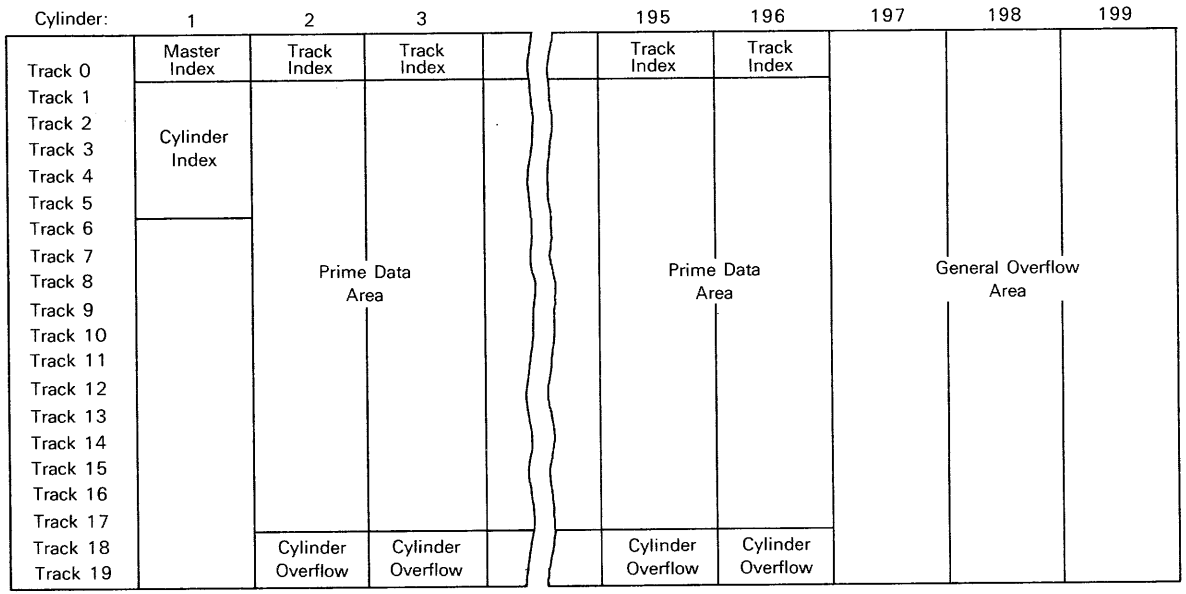


FIGURE 14-10 Indexed Sequential File on 2314 Pack

data tracks, and cylinder-overflow tracks, and the general overflow area. Figure 14-10 illustrates a disk area that houses an indexed sequential file with all six types of areas.

When an indexed sequential file is created, the records are written in sequence in the prime data area. As the records are written, the indexes are created to allow the records in the file to be accessed directly. When all of the records have been stored on the file, an end-of-file record is written just as if the file had sequential organization.

After the file has been loaded, track 1 of each cylinder contains the track index records. Any area left on track 1 and all of tracks 2 through 18 (assuming two tracks for cylinder overflow) contain the records of the file in sequence. The tracks that make up the cylinder overflow area contain no data at all. The entire area of all cylinders of the general overflow area is also blank.

When a record is added to an indexed sequential file, it is placed in its sequential location in the prime data area. All the records on that track with higher keys are moved up one record location and the record that is moved off the high end of the track is placed in the cylinder overflow area, if there is one. Otherwise, the bumped record goes into the general overflow area. To allow records in the overflow areas to be processed in sequence as well as directly, overflow index records are kept, along with normal index records, in the track index area. When a record is placed in the overflow area, the overflow index is changed so that it locates the next record in sequence following the records on the normal track in the prime data area. If more than one record is moved from one normal track to the cylinder overflow area, a chaining field (called the *sequence-link field*) in each overflow record is used to point by direct address to the next record in sequence. Thus, the sequence of records is maintained.

To illustrate the use of a cylinder-overflow area, overflow index records, and chaining, consider the example

TRACK 1 (INDEX RECORDS)				TRACKS 2–18 (PRIME DATA AREA)							TRACKS 19–20 (CYLINDER OVERFLOW AREA)		
Normal		Overflow											
Key	Data	Key	Data	*Track*	*Keys on Track*						*Count*	*Key*	*Sequence-Link Field*
187	T2	187	T2	2	012	041	049	094	101	187	C11,T19,R1	339	R255
284	T3	284	T3	3	188	210	218	247	250	284	C11,T19,R2	1397	T19,R6
322	T4	339	T19,R4	4	287	291	294	301	307	322	C11,T19,R3	580	R255
397	T5	397	T5	5	341	348	354	363	370	397	C11,T19,R4	331	T20,R2
513	T6	580	T20,R3	6	410	415	420	434	470	513	C11,T19,R5	1742	R255
641	T7	641	T7	7	585	592	601	615	621	641	C11,T19,R6	1404	R255
787	T8	787	T8	8	660	680	685	710	740	787	C11,T20,R1	949	R255
940	T9	949	T20,R1	9	812	819	901	914	927	940	C11,T20,R2	333	T19,R1
991	T10	991	T10	10	951	957	967	984	985	991	C11,T20,R3	555	T19,R3
1205	T11	1205	T11	11	1032	1105	1117	1121	1187	1205			
1297	T12	1297	T12	12	1207	1208	1231	1239	1250	1297			
1391	T13	1404	T19,R2	13	1330	1337	1341	1355	1366	1391			
1522	T14	1522	T14	14	1410	1415	1423	1480	1481	1522			
1639	T15	1639	T15	15	1523	1530	1537	1539	1599	1639			
1740	T16	1742	T19,R5	16	1641	1645	1691	1701	1703	1740			
1833	T17	1833	T17	17	1748	1780	1788	1790	1805	1833			
1949	T18	1949	T18	18	1838	1847	1897	1901	1930	1949			

FIGURE 14-11 One Cylinder of an Indexed Sequential File

in figure 14-11. It illustrates both the normal and overflow index records in the track index of one cylinder (cylinder 11), and nine overflow records in the overflow area (tracks 19 and 20). (C, T, and R are used to indicate cylinder, track, and record numbers.) Since the normal and overflow index records for track 2 are the same, it means that there are no overflow records for this track. The same applies for track 3. For track 4, however, the key of the normal index entry is 322, the key of the overflow index record is 339. That means that there are one or more overflow records from this track. Both the normal and overflow records can be read sequentially, however, in this manner:

1 Read the records on the normal track in sequence. (These records are always in sequence.)

2 Read the overflow record indicated in the overflow index record, namely, record 4 on track 19. This will be the next record in sequence.

3 Read the record indicated in the sequence-link field of the overflow record. Since this field points to the second record on track 20, that record is next in sequence.

4 Continue reading the chain of records until the sequence-link field indicates record number 255; then continue with the first record on the next normal track—in this case, track 5.

The same logic holds for processing the other overflow records in sequence. As a result, the next records in sequence

would be those from track 5, then those from track 6, then record 3 on track 20, then record 3 on track 19, then the records from track 7, and so forth. If you wonder why the records in the overflow area aren't in sequence, it is because the records are placed there in the order in which additions are made to the file. The only sequential linkage between these records comes from the sequence-link field.

To access records directly when there are overflow records, a somewhat different logic is used. First, the track indexes are searched as usual. The order of these indexes is normal track-1 index, overflow track-1 index, normal track-2 index, overflow track-2 index, etc. If the desired record is in the prime data area, the track is searched for the selected key as if there were no overflow records. If the desired key falls in the overflow area, however, the record indicated by the overflow index record is searched for and read. If this isn't the desired record, the record indicated in the sequence-link field is searched for and read. This search is continued until the desired record is found or the end of the chain is reached.

To illustrate, consider a search for record number 555 in figure 14-11. Since 555 is greater than 513 but less than 580, the track indexes indicate that the record is in the cylinder-overflow area. As a result, the search begins with the record indicated in the overflow index record, record 3 of track 20. Since this record is the desired record, the search ends after reading one overflow record. If the record was not record 555, however, the search would continue with the record indicated in the sequence-link field. Because of the extra searching (rotational delay) required by records in the overflow area, indexed sequential files should be reorganized periodically so that all records are returned to the prime tracks.

Indexed sequential files, then, offer the major advantages of both sequential and random processing. Of course, this flexibility is paid for in processing speed. When processed sequentially, an indexed sequential file is likely to be slower than a sequential file because of the extra searches for records in the overflow areas. Similarly, when processed randomly, an indexed sequential file will be slower than a direct file because of the extra seeks and searches for the index records and overflow records. Because each of the three types of file organizations has its advantages and limitations, you can decide which type of organization to use only after you have analyzed a file's characteristics and all of the uses to which it will be put.

Terminology

sequential file organization
end-of-file record
to block records
direct file organization
randomizing routine
division/remainder method
overflow area
indexed-sequential file organization
random processing
index
master index
cylinder index
track index
cylinder-overflow area
prime data area
general overflow area
sequence-link field

Objectives

1 Explain how blocking can (1) increase the number of records that can be stored on a disk and (2) decrease the time required to read or write a sequential file on disk.
2 Describe the function of a randomizing routine as applied to direct files.
3 Explain the significance of the following as they relate to indexed-sequential file organization:
   master index
   cylinder index
   track index
   prime data area
   cylinder-overflow area
   general overflow area
   sequence-link field

TOPIC THREE Programming Considerations As in other types of input/output processing, much of the code required to perform disk operations is contained in I/O modules that are supplied as part of the Disk Operating System. These subprograms not only issue the basic I/O commands like Seek, Search, and Read, they also perform error checking and recovery, blocking and deblocking, label checking, and file handling. Even though these routines are parts of the I/O modules associated with a file organization, you must understand them to be able to code properly the I/O macros through which you use the I/O modules.

ERROR-RECOVERY ROUTINES

When a record on a direct-access device is read, the cyclic check characters are checked to make sure that the read operation took place without error. If an error is detected, it may be recovered. Sometimes, for example, the piece of dust that caused the error is brushed off as the recording surface passes under the read/write head. If the program rereads the record by waiting one complete revolution while the record rotates under the read/write head again, the record may be read without error. In a typical *error-recovery routine* for a sequential-disk file, the record is reread ten times. If the error still exists, a message is printed on the console typewriter and the system is halted.

If an error is detected during a writing operation, it too may be corrected. In some cases, the record is written on an *alternate track*, and if this attempt to write the record takes place without error the program continues. (Both the 2314 and 3330 have several cylinders that can be used for alternate tracks. The 2314, for instance, actually has 203 cylinders with cylinders 0 through 199 used for normal processing and 200 through 202 for alternate-track assignment.)

BLOCKING AND DEBLOCKING ROUTINES

When a disk drive executes a read command, it reads an entire block of data into storage. The program, however, is usually looking for only one record at a time. *Deblocking routines* pick the proper record out of the block for a program.

To illustrate, consider a program written to process a sequential file of accounts-receivable payment records that are blocked five records per block. The first time an input record is requested for processing, the deblocking routine issues to the disk drive the proper commands, ending with a Read command. After the first block of five records is read into storage, the first record is passed to the program. The next four times the processing routine requests a payment record, the deblocking routine merely passes the next record from the block in storage. Finally, when the sixth record is requested, the deblocking routine sees that it has processed all the records in the current block and reads the next block from the disk.

A similar situation exists when creating blocked output files. Each time the processing portion of the main program PUTs an individual record, a *blocking routine* moves the record to the output area and inserts it in the next available record position in the block. Only when the entire block has been filled are the commands to write the block on the disk executed. Blocked records in the three file organizations are handled somewhat differently in detail, but the basic concept is the same.

LABEL-CHECKING ROUTINES

A single disk pack normally contains many files. Each file is assigned to a particular area on the pack. For example, a pack might have an accounts-receivable file in cylinders 1 through 30, a payroll master file in cylinders 36 through 55, and other files on the rest of the pack. A second pack could have cylinders 1 through 10 allocated as a temporary work area for record sorting, and a customer master file in cylinders 11 through 20. These file allocations are illustrated in figure 14-12.

I'll use these sample disk packs to illustrate some problems that labels protect against. First, suppose the operator mistakenly mounts disk pack 1 for a sort operation. Unless the error is caught, the sort program will use cylinders

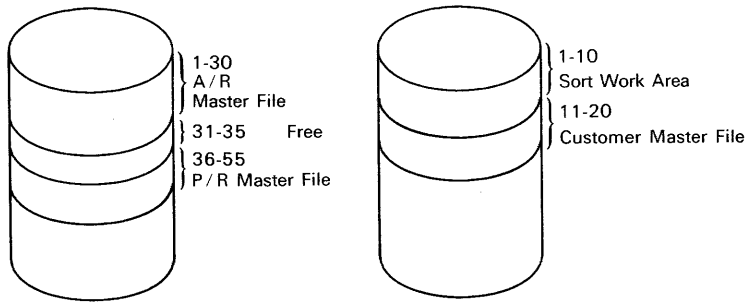


FIGURE 14-12 Disk Pack Allocations

1 through 10 and destroy some of the accounts-receivable master file.

To prevent this type of error, internal labels are used for each file that is stored on a disk. These labels are actually records on the device itself; they are usually found in cylinder zero. The entire label area is called the *Volume Table of Contents*, or *VTOC* (pronounced vee-tock). The first record in the VTOC is a *volume label* containing a six-character serial number assigned to the pack. Then, for each file on the disk, there are one or more records called *file labels*. These labels give the name of the file, its organization, its location on the disk, and its expiration date. The expiration date tells when a file is no longer needed so that its area can be assigned to another file. If a file is found in more than one area of the disk—cylinders 11-20 and 41-49, for example—this information is given in the file label also. Each area of a file is referred to as an *extent* of the file; this is expressed as lower and upper limits of the area in terms of cylinder and head numbers.

These labels are processed by label-checking routines —routines that compare the data the labels contain with data supplied in job-control cards at the time the program is executed. These job-control cards indicate which program should be run, which pack should be mounted on which disk drive, and what information the volume and file labels should contain. They also indicate the extents of each file that is going to be processed. Chapter 16 explains in detail the contents and formats of these cards.

When a file is opened for processing, the volume label is compared to the one supplied in the job-control cards. This is called *volume-label checking*. If the six-character volume codes don't match, the operator is notified by a console message and given the opportunity to mount the proper pack. This would prevent the accounts-receivable file from being destroyed when the sort program is executed.

Once the volume label check has been completed, *file-label checking* is performed. This level of label processing actually does two types of checking. First, the filename is checked. For an input file, the name of the file as supplied in the job-control cards is compared to the list of filenames on the pack. If an equal name is found, the file is opened for processing. Otherwise, a message is issued to the operator and processing stops. For output files, the list of filenames is checked to see that no file with the same name already exists on the pack. If a duplicate is found, a message is printed on the console typewriter.

If no duplicate name exists for an output file, the second part of file-label checking is performed. The disk-extents in the job-control cards are compared with the extents given in the file labels for all other files on the disk pack. As long as the new file does not overlap any active file, the program continues. If, however, the new file will overlap an area already assigned to some other file, processing is halted and a message is issued to the operator.

Suppose a programmer wanted to use the free area in cylinders 31-35 on disk 1 in figure 14-12. By mistake, he assigns cylinders 30-35 for his new file. Extent checking will prevent the last cylinder of the accounts-receivable file from being destroyed. When his program tries to open the output file, the conflict over cylinder 30 will be found and the operator will be alerted to the problem.

FILE-HANDLING ROUTINES

Sequential files are always processed beginning with the first record, then proceeding sequentially through all the records to the last one. The I/O modules therefore provide

little file-handling logic. About all that is necessary is a routine to calculate the proper disk address for the Seek and Search commands as each record or block is processed.

Direct files also require little file-handling support. Since it is the programmer's responsibility to develop the proper disk address in a randomizing routine, the I/O module has little to do except issue the I/O commands and handle read or write errors.

Indexed sequential files, however, demand a great deal of file-handling support. Consider what the I/O module must do in response to a programmer's request to read a record randomly by key. First, it must issue the I/O commands to read each record in the cylinder index (I'm assuming there is no master index) until a high or equal key is found. Then, a new series of I/O commands must be constructed to read the track-index records sequentially using the cylinder specified in the cylinder index. If the desired record is on one of the prime data tracks of the cylinder, only one more set of I/O commands is needed to read the data record. If the record is in an overflow area, however, it might require several more sets of I/O commands to follow the sequence chain. All this, just to read a record randomly by key. You can imagine that adding a record to a file including finding the proper place for the record, moving records up on the track, bumping a record off the track, rewriting the bumped record in an overflow area, and updating the proper index records requires even more extensive and complex file-handling routines.

WHAT THE PROGRAMMER MUST KNOW

Because of the many I/O routines needed for disk files, it is fortunate that assembler language in combination with the I/O modules and supervisor provides most of the code needed. In general, a programmer need not be concerned about error-recovery, label-checking, blocking, or deblocking routines. Nor does he need to worry about the individual Seek, Search, Read, Write, and Write-Verify commands, or about file-handling routines for indexed sequential files.

What, then, must a programmer know? He must be able to specify the characteristics of the file: block size, record size, key length, record format, and so on. When writing programs for direct files, he is also responsible for the randomizing routine.

If variable-length records are used, the programmer must also know the exact format of these records and blocks. Because variable-length disk records are rarely used, they are not described in this chapter. If you ever find a need for processing these records, however, the record formats and concepts are the same as for variable-length tape records, which are presented in chapter 12.

Determining Record and Block Size

Another thing the programmer should be able to do is determine the best record length and block length for a disk file. He must also be able to calculate the number of cylinders required for his file in order to determine the extents of his file. For this job, a programmer uses disk capacity charts like those for the 2314 and 3330 given in figures 14-13 and 14-14. The sections labeled WITHOUT KEYS apply to records in the count-data format; the sections labeled WITH KEYS apply to records in the count-key-data format.

To look up how many records will fit on a track or cylinder, you look in the BYTES PER RECORD column for the minimum and maximum values that bracket your record or block length. To illustrate, suppose a customer master file consists of 15,000 records containing 160 bytes of data. Assuming the file will be stored on a 2314, I'll use figure 14-13 for determining record and block lengths.

For the first case, suppose the file will be in sequential organization and the records will be unblocked. In the WITHOUT KEYS section, record length 160 falls between 158 and 166, so 27 records can be stored on each track and 540 on each cylinder. It will then take almost 24 cylinders to hold the 15,000 records. Because the same number of records can be stored per track whether the records are 160 or 166 bytes long, I would use a record length of 166 bytes. Then, 6 bytes will be available in each record for future use.

Bytes Per Record				Records Per	
Without Keys		With Keys			
Min	Max	Min	Max	Track	Cylinder
3521	7294	3477	7249	1	20
2299	3520	2255	3476	2	40
1694	2293	1650	2254	3	60
1333	1693	1289	1649	4	80
1093	1332	1050	1288	5	100
922	1092	878	1049	6	120
794	921	751	877	7	140
695	793	651	750	8	160
616	694	572	650	9	180
551	615	507	571	10	200
497	550	453	506	11	220
451	496	408	452	12	240
412	450	369	407	13	260
375	411	334	368	14	280
348	377	305	333	15	300
322	347	278	304	16	320
299	321	255	277	17	340
277	295	234	254	18	360
259	276	216	233	19	380
242	255	199	215	20	400
227	241	184	198	21	420
212	226	169	183	22	440
200	211	157	168	23	460
188	199	145	156	24	480
177	187	134	144	25	500
167	176	124	133	26	520
158	166	115	123	27	540
149	157	106	114	28	560
140	148	97	105	29	580
133	139	90	96	30	600
126	132	83	89	31	620
119	125	76	82	32	640
113	118	70	75	33	660
107	112	64	69	34	680
101	106	58	63	35	700
95	100	52	57	36	720
91	94	47	51	37	740
86	90	43	46	38	760
81	85	38	42	39	780
77	80	34	37	40	800
73	76	30	33	41	820
70	72	26	29	42	840
66	69	23	25	43	860
62	65	19	22	44	880
58	61	15	16	45	900
55	57	12	14	46	920
52	54	9	11	47	940
48	51	5	8	48	960
46	47			49	980
44	45			50	1000
41	43			51	1020
38	40			52	1040
35	37			53	1060
33	34			54	1080
31	32			55	1100
28	30			56	1120
26	27			57	1140
24	25			58	1160
23	23			59	1180
21	22			60	1200
19	20			61	1220
17	18			62	1240
15	16			63	1260
13	14			64	1280
12	12			65	1300
10	11			66	1320
8	9			67	1340
7	7			68	1360
5	6			69	1380

FIGURE 14-13 2314 Disk Capacity Chart

Bytes Per Record				Records Per	
Without Keys		With Keys			
Min	Max	Min	Max	Track	Cylinder
6448	13030	6392	12974	1	19
4254	6447	4198	6391	2	38
3157	4253	3101	4197	3	57
2499	3156	2443	3100	4	76
2060	2498	2004	2442	5	95
1746	2059	1690	2003	6	114
1511	1745	1455	1689	7	133
1328	1510	1272	1454	8	152
1182	1327	1126	1271	9	171
1062	1181	1006	1125	10	190
963	1061	907	1005	11	209
878	962	822	906	12	228
806	877	750	821	13	247
743	805	687	749	14	266
688	742	632	686	15	285
640	687	584	631	16	304
597	639	541	583	17	323
558	596	502	540	18	342
524	557	468	501	19	361
492	523	436	467	20	380
464	491	408	435	21	399
438	463	382	407	22	418
414	437	358	381	23	437
392	413	336	357	24	456
372	391	316	335	25	475
353	371	297	315	26	494
336	352	280	296	27	513
319	335	263	279	28	532
304	318	248	262	29	551
290	303	234	247	30	570
277	289	221	233	31	589
264	276	208	220	32	608
253	263	197	207	33	627
242	252	186	196	34	646
231	241	175	185	35	665
221	230	165	174	36	684
212	220	156	164	37	703
203	211	147	155	38	722
195	202	139	146	39	741
187	194	131	138	40	760
179	186	123	130	41	779
172	178	116	122	42	798
165	171	109	115	43	817
158	164	102	108	44	836
152	157	96	101	45	855
146	151	90	95	46	874
140	145	84	89	47	893
134	139	78	83	48	912
129	133	73	77	49	931
124	128	68	72	50	950

Bytes Per Record				Records Per	
Without Keys		With Keys			
Min	Max	Min	Max	Track	Cylinder
119	123	63	67	51	969
114	118	58	62	52	988
109	113	53	57	53	1007
105	108	49	52	54	1026
101	104	45	48	55	1045
96	100	40	44	56	1064
92	95	36	39	57	1083
89	91	33	35	58	1102
85	88	29	32	59	1121
81	84	25	28	60	1140
78	80	22	24	61	1159
74	77	18	21	62	1178
71	73	15	17	63	1197
68	70	12	14	64	1216
65	67	9	11	65	1235
62	64	6	8	66	1254
59	61	3	5	67	1273
56	58	2	2	68	1292
54	55			69	1311
51	53			70	1330
48	50			71	1349
46	47			72	1368
43	45			73	1387
41	42			74	1406
39	40			75	1425
36	38			76	1444
34	35			77	1463
32	33			78	1482
30	31			79	1501
28	29			80	1520
26	27			81	1539
24	25			82	1558
22	23			83	1577
20	21			84	1596
19	19			85	1615
17	18			86	1634
15	16			87	1653
13	14			88	1672
12	12			89	1691
10	11			90	1710
9	9			91	1729
7	8			92	1748
6	6			93	1767
4	5			94	1786
3	3			95	1805
1	2			96	1824

FIGURE 14-14 3330 Disk Capacity Chart

Normally, of course, you would block a sequential file to improve both processing efficiency and disk use. What blocking factor should you use? To get maximum disk use, pick some multiple of 160 that is close or equal to one of the maximum BYTES PER RECORD values on the capacity chart. Generally, the higher the blocking factor, the better level of disk use and the least overall rotational delay. However, you must also keep in mind how much storage is available for I/O areas in the programs that must process the file.

By comparing the multiples of 160 to the maximum values in the capacity chart, I picked a blocking factor of 14. That means that three 2240-byte blocks of 14 records each will fit on each track—a total of 42 records per track and 840 per cylinder. The 15,000 records then require a little less than 18 cylinders.

Even though this blocking significantly improves the use of the disk space, 58 bytes (2298 minus 2240) are wasted in each block. That means I can add 4 bytes to each record and still stay within the maximum limit. (A record length of 164 bytes times 14 records per block equals 2296 bytes.) Now each record has an additional 4 bytes that can be used for expansion of the records and I still get the same number of records per track and cylinder.

If the file is to be in indexed sequential organization, you must use the WITH KEYS section. Take the case of 160-byte unblocked records. The BYTES PER RECORD values on the chart represent the sum of the key length and the block length. I'll assume that the key to the customer master records is to be the customer number and that it is 4 bytes long. The BYTES PER RECORD value is then 164. Since 164 falls between 157 and 168, 23 records can be stored on each track and record size can be increased to 164 bytes.

For indexed sequential files, though, data is stored only on the prime tracks when loading the file. The first track of each cylinder holds the track index (usually, it occupies about one-half of a 2314 track or about one-third of a 3330 track) and any tracks reserved for overflow on each cylinder hold no records at loading time. For the customer master file, I'll assume that one track per cylinder is reserved for overflow, so I've approximately 18½ tracks available on each cylinder for loading the file. At 23 records per track (I'll guess that 11 records will fit on the first track along with the track index), 425 records will be stored on each cylinder (18 tracks times 23 records plus 11 records equals 425). The file will then require more than 35 cylinders.

Indexed sequential files are usually blocked to improve disk use. This time you look for a blocking factor such that the sum of one key length and the block length approaches or equals a maximum BYTES PER RECORD value. Again, a blocking factor of 14 would be a good choice for this file. The sum of block length, 2240, and key length, 4, is 2244, only 10 bytes less than the maximum shown, 2254 bytes.

This blocking factor permits 42 records per track. Since the block is about one-third of a track, only one block will fit on the first track with the track index. The total number of records per cylinder, then, is 770 (18 times 42 plus 14). The 15,000 record file can now be stored on 20 cylinders.

Terminology

error-recovery routine
alternate track
deblocking routine
blocking routine
label-checking routine
Volume Table of Contents
VTOC
volume label
file label
extent
volume-label checking
file-label checking

Objectives

1 Explain the significance of each of the following in relation to disk processing:
   error-recovery routine
   label-checking routine
   blocking or deblocking routine
   file-handling routine
   extents
2 Given track capacity charts and the characteristics of a file, choose an appropriate blocking factor, block size, and record length for the file. Also, determine how many cylinders are required for the file.

Problem

(Objective 2) An indexed sequential file of 19,000 records, 218-bytes per record with eight-byte keys, is to be stored on a 3330.

a. What is the most logical blocking factor if the block size must be under 2000 bytes and what is the best record length?
b. How many cylinders are required for the prime area of the file if two tracks are required in each cylinder for cylinder overflow?

Solution

a. A blocking factor of 9 with 221-byte records.
b. 21 cylinders.

15

BAL for Direct-Access Devices

This chapter is divided into three topics. The first presents BAL code for sequential files, the second for indexed sequential files, and the third for direct files. Because the vast majority of System/360–370 direct-access files are in sequential or indexed sequential organization, you may want to skip topic 3. This chapter assumes you are familiar with the concepts presented in chapter 14.

TOPIC ONE **Sequential Files** Sequential files on a direct-access device are handled very much like card or tape files. The only significant difference occurs in writing update programs. On a direct-access device, the updated master record can be written in the same location as the one from which it was read. In contrast, an update program for tape usually reads the old master records from one tape and writes the updated master records on another.

Priority	Keyword	Programmer Code	Default	Remarks
Required	BLKSIZE	Length of I/O area		Block length, but for output files must include 8 bytes for count.
Required	IOAREA1	Name of I/O area		Length equals block length. For output files, must include 8 bytes at front for count field.
Optional	IOAREA2	Name of second I/O area		Must be same length as IOAREA1.
Optional	EOFADDR	Label of first instruction of EOF routine		Must be coded for input files.
Optional	DEVICE	2311 2314 3330	2311	
Optional	RECFORM	FIXUNB FIXBLK VARUNB VARBLK	FIXUNB	FIXBLK is most common.
Optional	RECSIZE	Record length		Used only if RECFORM=FIXBLK
Optional	TYPEFLE	INPUT OUTPUT	INPUT	Usually omitted for input files.
Optional	UPDATE	YES		Used only when records are to be updated in place. PUT used to rewrite record just read and updated. TYPEFLE must be INPUT.
Optional	IOREG	Register number (nn)		Included if blocked records to be processed in I/O area. Omit WORKA.
Optional	WORKA	YES		Records are to be processed in work area named in GET or PUT. Omit IOREG operand.
Optional	VERIFY	YES		Output records are to be write verified.

FIGURE 15-1 DTFSD Operand Summary

Figure 15-1 summarizes the most widely used operands for the DTF for sequential disk files called the DTFSD. Although most of these operands should be familiar to you, there are three points that should be made. First, the I/O area for an output file must be as large as the block size plus eight bytes for the count area. As a result, the BLKSIZE operand and the I/O area definition must reflect these additional bytes. However, the program is not responsible for moving any data into the eight-byte count area. This is done automatically by the I/O macros in conjunction with the I/O modules.

Second, there are four forms that can be expressed in the RECFORM operand. Because variable-length disk records are rarely used, however, their coding is not illustrated in this topic. Since the variable-length coding for sequential disk files is analogous to that for tape files, you can refer to chapter 13 for illustrations of variable-length coding.

Third, a register can be specified in the IOREG operand

Shipment Record Format (10 Records Per Block):

Position	Field	Length	Format
1–2	Transaction Code (C'82')	2	C
3–8	Product Code	6	C
9–12	Quantity Shipped	4	C
13–16	Order Number	4	C
17–24	Shipment Date (MM/DD/YY)	8	C

Order Record Format (5 Records Per Block):

Position	Field	Length	Format
1–4	Order Number	4	C
5–8	Customer Number	4	C
9–28	Customer Name	20	C
29–34	Product Code Ordered	6	C
35–38	Quantity Ordered	4	C
39–46	Date Ordered (MM/DD/YY)	8	C
47–52	Product Code Shipped	6	C
53–56	Quantity Shipped	4	C
57–64	Date Shipped (MM/DD/YY)	8	C
65–80	Not Used	16	

Report Record Format (Unblocked):

Same as updated order file record.

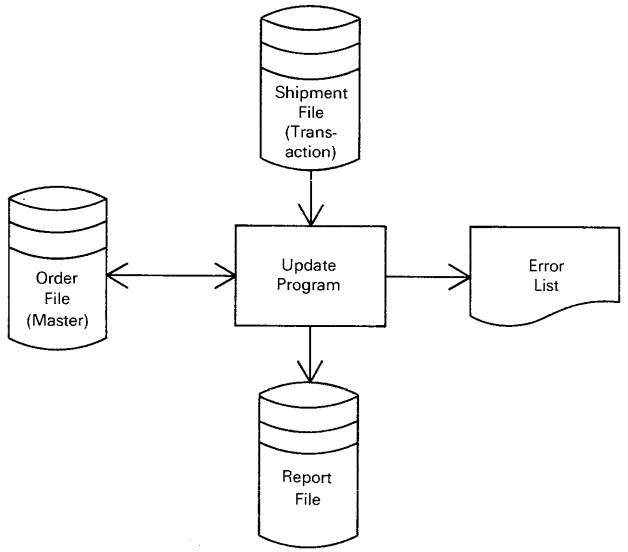


FIGURE 15-2 Sequential Disk Update Problem

for use in the automatic blocking and deblocking routines. These routines will then place the address of the next record to be processed in the *I/O register* specified, regardless of whether one or two I/O areas are used for the file. This technique is illustrated by the program in figure 15-4 and is described more fully later on. When an I/O register is used, a work area cannot be used.

AN UPDATE PROGRAM

To illustrate coding for sequential files, figure 15-2 presents an update problem. A shipment file representing shipments made is used to update an open order file representing unfilled customer orders. Both files are stored on a 2314 in order-number sequence, and there can be only one shipment record for each order record. To update the order record, the product code, quantity shipped, and shipment date from the shipment record are moved into the corresponding fields of the order record. After an order record has been updated, the update record is written in its original location on the order file and a report record is written on a 2314 report file. This report file is used later as input to a tape-to-printer program. If an unmatched transaction (a shipment record that has no corresponding order record) is read, a line is

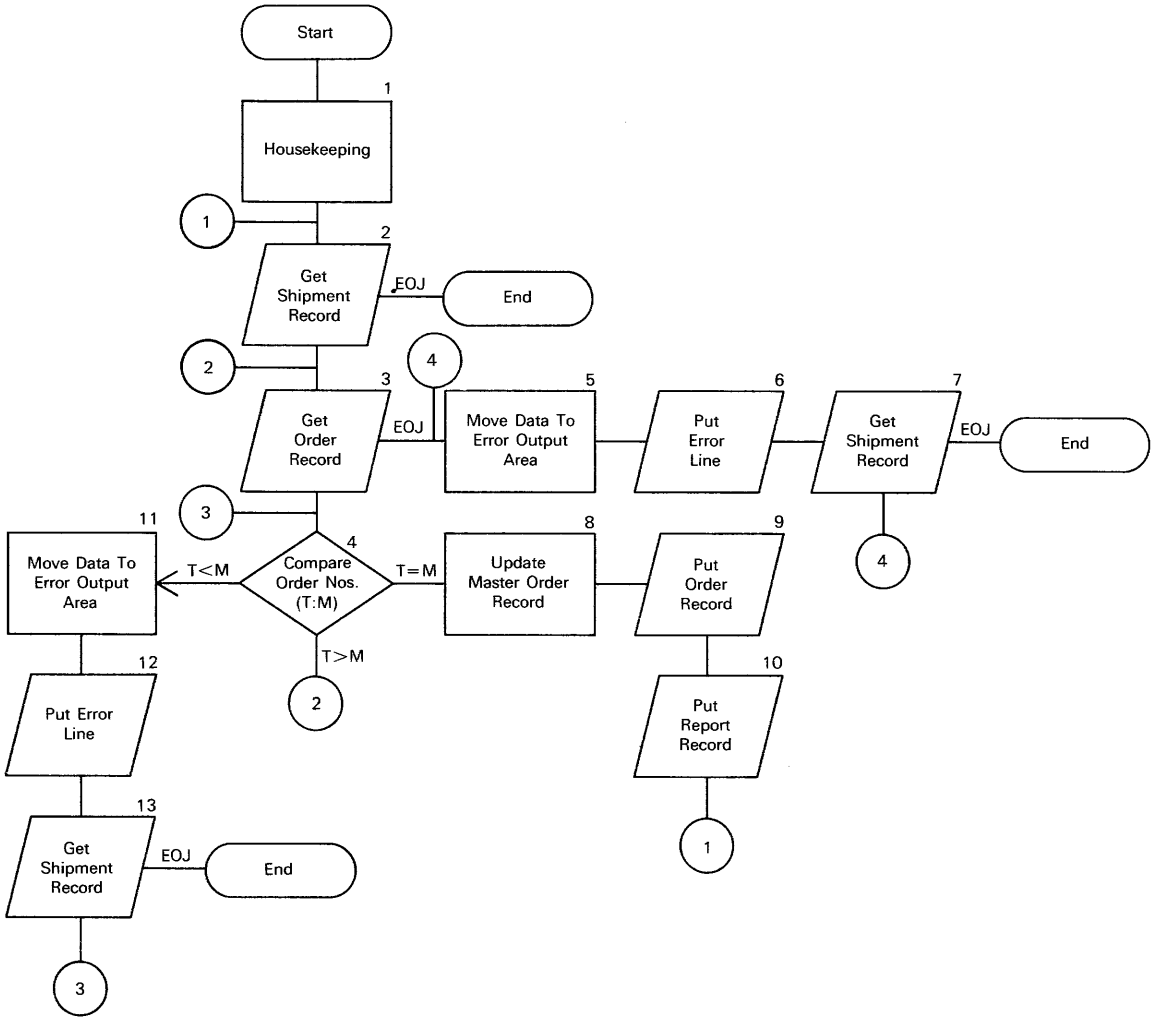


FIGURE 15-3 Flowchart for Sequential Update Program

printed on the error list.

Figure 15-3 is a flowchart for this problem and figure 15-4 is its BAL solution. Block 4 on the flowchart represents the key to the logic of the program. Here, the order number in the shipment record (the transaction) is compared with the order number in the order record (the master). If they are equal (T=M), the order record is updated and written back onto the file and a report record is written on the report file. If the transaction order number is greater than the master order number (T>M), another master record is

```
ORDUPD   START 0
BEGIN    BALR  3,0
         USING *,3
         USING SHIPMASK,4
         OPEN  SHIPTR,ORDERS,RPTTR,ERRS
READSHIP GET   SHIPTR
READORD  GET   ORDERS,ORDWORK
TEST     CLC   ORDNBR,SHORDNBR
         BL    READORD
         BE    MATCH
         MVC   ERRTR,SHIPMASK
         PUT   ERRS
         GET   SHIPTR
         B     TEST
MATCH    MVC   ORPRODSH,SHPROD
         MVC   ORQTYSH,SHQTY
         MVC   ORDATESH,SHDATE
         PUT   ORDERS,ORDWORK
         PUT   RPTTR,ORDWORK
         B     READSHIP
EOFSHIP  CLOSE SHIPTR,ORDERS,RPTTR,ERRS
         EOJ
EOFORD   MVC   ERRTR,SHIPMASK
         PUT   ERRS
         GET   SHIPTR
         B     EOFORD
SHIPTR   DTFSD BLKSIZE=240,EOFADDR=EOFSHIP,DEVICE=2314,IOAREA1=SHIPIN1,X
               IOAREA2=SHIPIN2,IOREG=(4),RECFORM=FIXBLK,RECSIZE=24
ORDERS   DTFSD BLKSIZE=400,EOFADDR=EOFORD,DEVICE=2314,IOAREA1=ORDIO,    X
               RECFORM=FIXBLK,RECSIZE=80,UPDATE=YES,WORKA=YES
RPTTR    DTFSD BLKSIZE=80,DEVICE=2314,IOAREA1=RPTIO,TYPEFLE=OUTPUT,     X
               WORKA=YES
ERRS     DTFPR DEVADDR=SYSLST,BLKSIZE=132,IOAREA1=ERRLINE
SHIPMASK DSECT
SHTRCODE DS    CL2
SHPROD   DS    CL6
SHQTY    DS    CL4
SHORDNBR DS    CL4
SHDATE   DS    CL8
ORDUPD   CSECT
ORDWORK  DS    0CL80
ORDNBR   DS    CL4
         DS    CL24
ORPRODOR DS    CL6
ORQTYOR  DS    CL4
ORDATEOR DS    CL8
ORPRODSH DS    CL6
ORQTYSH  DS    CL4
ORDATESH DS    CL8
         DS    CL16
ERRLINE  DS    0CL132
         DC    20C' '
ERRTR    DS    CL24
         DC    CL88'      UNMATCHED SHIPMENT TRANSACTION'
SHIPIN1  DS    CL240
SHIPIN2  DS    CL240
ORDIO    DS    CL400
RPTIO    DS    CL80
         END   BEGIN
```

FIGURE 15-4 Sequential Update Program

```
         USING SHIPMASK,4
         .
         .
         .
READSHIP GET   SHIPTR
         .
         .
         .
MATCH    MVC   ORPRODSH,SHPROD
         MVC   ORQTYSH,SHQTY
         MVC   ORDATESH,SHDATE
         .
         .
         .
SHIPTR   DTFSD BLKSIZE=240,EOFADDR=EOFSHIP,DEVICE=2314,IOAREA1=SHIPIN1,X
               IOAREA2=SHIPIN2,IOREG=(4),RECFORM=FIXBLK,RECSIZE=24
         .
         .
         .
SHIPMASK DSECT
SHTRCODE DS    CL2
SHPROD   DS    CL6
SHQTY    DS    CL4
SHORDNBR DS    CL4
SHDATE   DS    CL8
ORDUPD   CSECT
         .
         .
         .
```

FIGURE 15-5 Using an I/O Register

read and the loop repeats. If the transaction is less than the master order number (T<M), it indicates an unmatched transaction, in which case, an error line is printed, another transaction is read, and processing continues.

The only other point of interest in the flowchart is the end-of-file processing for the master file. If this end-of-file is reached before the end of the shipment file, it indicates one or more unmatched shipment records. In this case, the program prints an error line on the error list, reads another shipment record, and repeats the master end-of-file loop. When the shipment end-of-file is reached within this loop, the program ends.

If you look at the DTFSD for the shipment file, you can see that it uses two I/O areas along with an I/O register, register 4. So that the same names can be used for each shipment record processed, a dummy section and a USING statement are coded as indicated in figure 15-5. This is the code from figure 15-4 that relates to the use of the I/O register. Since the I/O register will always contain the address of the first byte of the record being processed and the names in the DSECT are represented by displacements starting from zero, the names such as SHPROD, SHQTY, and SHORDNBR will always represent the appropriate fields in the shipment record being processed. The CSECT statement at the end of the dummy section resets the location counter to the next available storage location so the assembly can continue.

In the DTFSD for the master order file, UPDATE=YES is specified. This means that when a PUT is issued for the file, the record will be written in the location from which the previous GET read a record. As a result, you cannot issue a PUT for an update file without first issuing a GET.

Otherwise, you should have little difficulty following this program. Label-checking routines are taken care of primarily by the OPEN and CLOSE macros. If a file requires multiple packs or multiple areas on one pack, the switching from one extent to the next is taken care of by the GET or PUT macro. Since all disk labels must be in standard format, there isn't even a label-type operand in the DTF (which there must be for tape). Since the symbolic device name such as SYS015 is normally specified in a job-control card rather than in the DTF for a file, the DEVADDR operand generally is not used.

Terminology

I/O register

Objective

Given a problem involving disk files in sequential organization, code a BAL solution. The problem may require any of the following: creating the file, updating the file, retrieving records from the file, and adding records to or deleting records from the file. The records may be blocked or unblocked, the file may require one or more disk packs or extents, and the program may require more than one input or output file.

Problem

Suppose an input deck of cards has this format:

CARD COLUMNS	FIELD
1-6	Product Code
7-10	Quantity Shipped
11-14	Order Number
12-22	Shipment Date (MM/DD/YY)

Write a program to create from this deck the shipment file specified in figure 15-2. Use two I/O areas and I/O register 5 for the shipment file.

Solution

Figure 15-6 is an acceptable solution. The key point is the use of the eight-byte count area in the I/O area.

```
CRDDISK  START 0
BEGIN    BALR  3,0
         USING *,3
         USING DMASK,5
         OPEN  CRDFILE,DSKFILE
READCARD GET   CRDFILE
         MVC   DCODE,=C'82'
         MVC   DDATA,CDATA
         PUT   DSKFILE
         B     READCARD
CRDEOF   CLOSE CRDFILE,DSKFILE
         EOJ
CRDFILE  DTFCD DEVADDR=SYSIPT,EOFADDR=CRDEOF,IOAREA1=CAREA
DSKFILE  DTFSD BLKSIZE=248,IOAREA1=D1,IOAREA2=D2,IOREG=(5),DEVICE=2314,X
               RECFORM=FIXBLK,RECSIZE=24,TYPEFLE=OUTPUT
DMASK    DSECT
DCODE    DS    CL2
DDATA    DS    CL22
CRDDISK  CSECT
CAREA    DS    0CL80
CDATA    DS    CL22
         DS    CL58
D1       DS    CL248
D2       DS    CL248
         END   BEGIN
```

FIGURE 15-6 Card-to-Disk Program

TOPIC TWO Indexed Sequential Files BAL programmers commonly refer to indexed sequential files as ISAM files (related to the Indexed Sequential Access Method). When an ISAM file is created, the processing is done sequentially. The records are written on the disk in sequence by control number in successive record locations on the prime data tracks. Once an ISAM file has been stored on a disk, it may be processed in sequence or in random order, whichever is more efficient for the job being done.

Figure 15-7 summarizes the most widely used operands for a DTF statement for an ISAM file, called a DTFIS. Although this summary is self-explanatory, there are a number of related points that you should be aware of.

First, there is no DEVADDR operand in the DTFIS since the symbolic-device name (such as SYS015) is given in a job-control card prior to program execution. In addition, there is no file-label operand (which there is for tape files), since all disk labels must be in standard format.

Second, there are always at least two extents for an indexed sequential file: one for the file area and one for the cylinder index. In addition, there will be one extent for a master index if one is used, one for a general overflow area if such an area is used, and extra extents for the file area if it is a multipack file. Thus, the DSKXTNT operand will indicate two or more extents.

Third, it is possible to have the cylinder index and master index on a device other than the device for the prime area. For instance, the master and cylinder index might be on a 2314, while the prime data area is on a 3330. If this is the case, the operand HINDEX specifies the device for the highest index used. However, in actual practice, the indexes and the prime area are usually on the same device.

Fourth, it is possible to have the cylinder index read into storage at the start of a program. Then, the searches through the index are done at CPU speeds rather than at the slower disk speeds. This can cut down on a great deal of

access-mechanism movement and rotational delay. If this feature is used, the INDAREA operand gives the name of the index area in storage and the INDSIZE operand gives the number of bytes in the area. Although the formula in the summary will calculate the number of bytes required for the entire cylinder index (the number of prime cylinders plus four, multiplied by the key length plus six), it is possible to specify less than this maximum number. If less than the maximum number of cylinders is specified, only a portion of the cylinder index will be read into storage at one time with a resulting decrease in efficiency.

Fifth, when ISAM records are blocked, only the highest control field number in the block is stored in the physical key for the block. That means *embedded keys*—keys within the actual records—must be used for finding individual records. In this case, the KEYLOC operand gives the starting byte of the key field within each record. If, for example, the control field is found in bytes 11-15 of a record, the operand would be KEYLOC=11. By combining this operand with the required KEYLEN operand, the assembler can determine which bytes in a record are the key field.

Finally, the I/O area length depends on whether the program creates (*loads*) a file, processes a file sequentially, processes it randomly, or adds records to it. The length varies because different items are read into storage or are required for output depending on the function being performed. Sometimes only the data area is involved; sometimes count, key, or sequence-link fields are involved. The I/O area length is summarized in figure 15-8. For example, the I/O area for adding unblocked records to a file must be the eight bytes for the count area plus the key length plus ten bytes for the sequence-link field plus the record length. When adding blocked records to a file, the I/O area is usually the length of the count area plus the key length plus the data area length; however, if the data area is very small, the minimum I/O area size is count plus key plus sequence-link field plus one record length.

When processing ISAM files, different elements may be read into an input area or be required by an output area. It depends on whether the data involves a prime track or an overflow area. If, for example, an unblocked record is read randomly from an overflow area, the ten-byte sequence-link field will precede the data. If the record is read from a prime track, it will be followed by ten unused bytes in the I/O area. For this reason, a work area (rather than an I/O register) is used with indexed sequential files. The work areas will contain either just one record or one record plus key. To define a work area, use the righthand column in figure 15-8 to determine what the contents of the work area will be.

FILE CREATION

Loading (creating) an ISAM file is much like creating a sequential-disk file. Usually, an input file of card, tape, or sequential-disk records provides the data that goes into each indexed sequential record. Since the indexed sequential records must be created in key sequence, the input file records must also be in key-field sequence.

Figure 15-9 shows a sample program that loads an ISAM file of basic inventory records from a deck of input cards. The records are 70 bytes long and contain these fields in sequence: item number (the key field), item description, unit of measure, reorder-policy code, reorder quantity, reorder point, safety stock quantity, on-hand quantity, on-order quantity, allocated quantity, and 12 unused bytes containing hex zeros. The basic processing loop of the program reads a card, moves and packs its data into the ISAM record, and writes an output record.

A programmer normally has to make several decisions about the indexed sequential file before he can code the load program. He must decide what blocking factor to use, whether a master index is necessary, and how much and what type of overflow area should be reserved. For this example, I'll assume that the file will be a maximum of 4500 records and that I'll block them 10 per block. I'll also assume that the number of additions to this file will be fairly low and that a single track per cylinder will be enough overflow area to last a year.

At the blocking factor I've chosen, I'll be able to get 80 records (eight blocks per track at 10 records per block) on

Priority	Keyword	Programmer Code	Default	Remarks
Required	DSKXTNT	Maximum number of extents for this file		
Required	IOROUT	LOAD ADD RETRVE ADDRTR		Specify type of processing: loading file, adding records, retrieving (also updating in place), or add/retrieve.
Required	RECFORM	FIXUNB FIXBLK		
Required	RECSIZE	Number of bytes in record		
Required	KEYLEN	Key length		
Optional	NRCDS	Number of records per block	1	Required for RECFORM=FIXBLK
Optional	DEVICE	2311 2314 3330	2311	
Optional	HINDEX	2311 2314 3330	2311	Specify the type of DASD device that holds the highest level index. Usually same as DEVICE.
Optional	MSTIND	YES		Indicates that master index is used for this file. If omitted, no master index is created or expected in add or retrieve.
Optional	INDAREA	Name of user area reserved to hold cylinder index in storage		Cylinder index can be held in storage to process records faster. INDAREA names the storage area; INDSIZE specifies number of bytes.
Optional	INDSIZE	Number of bytes reserved to store cylinder index in INDAREA field.		Must be included if INDAREA is coded. Most effective size is large enough to hold entire index: (Nbr of prime cyls+4)×(keylgth+6)
Optional	CYLOFL	Number of tracks per cylinder for overflow records		Usually 1 or 2 for 2311; 2, 3, or 4 for 2314 and 3330. Depends on addition activity.
Optional	KEYLOC	Starting position of key field in blocked records		For blocked records, key is embedded in data of each record. Required if RECFORM=FIXBLK.
Optional	KEYARG	Name of user key field		In random retrieval, user supplies key of record to be read or written in this field. Also used to specify starting key for skip sequential.

FIGURE 15-7 DTFIS Operand Summary

Priority	Keyword	Programmer Code	Default	Remarks
Optional	TYPEFLE	RANDOM SEQNTL RANSEQ		Coded only when IOROUT=RETRVE or ADDRTR. Indicates random, sequential, or combination retrieval.
Optional	IOAREAL	Name of I/O area for load function		Must be coded if IOROUT=LOAD, ADD, or ADDRTR. For LOAD, length must be count (8)+key+block. Same for ADD of blocked records. For unblocked records, must also allow 10 bytes for sequence-link field.
Optional	IOAREAR	Name of I/O area for random processing		Must be coded if IOROUT=RETRVE or ADDRTR and TYPEFLE=RANDOM or RANSEQ. For unblocked records, length must be sequence link (10)+record length. For blocked records length is block length or record length+10, whichever is more.
Optional	IOAREAS	Name of I/O area for sequential processing		Must be coded if IOROUT=RETRVE or ADDRTR and TYPEFLE=SEQNTL or RANSEQ. For unblocked records, length must be key length+sequence link (10)+record length. For blocked records, length is same as for IOAREAR.
Optional	IOAREA2	Name of second I/O area for load or sequential retrieve		Two I/O areas can be used for loading or sequential processing. Length must be the same as IOAREAL or IOAREAS.
Optional	WORKL	Name of work area for load or add		For unblocked records, work area must hold key and record. For blocked records, only record. Omit IOREG.
Optional	WORKR	Name of work area for random processing		Work-area length must be one record length. Omit IOREG.
Optional	WORKS	YES		Indicates that GET or PUT will specify work area. Work-area length same as for WORKL. Omit IOREG.
Optional	IOREG	Register number (n)		For load or sequential processing, records can be processed in I/O area. I/O module puts record address in I/O register. Omit WORKL or WORKS.
Optional	VERIFY	YES		Request write verification of output records.

FIGURE 15-7 DTFIS Operand Summary (Continued)

Function	Record Format	I/O Area Keyword	I/O Area Length	Work Area Keyword	Work Area Contents
Load	Unblocked	IOAREAL IOAREA2	C+K+RL	WORKL	Key + Record
Load	Blocked	IOAREAL IOAREA2	C+K+D	WORKL	Record
Add	Unblocked	IOAREAL	C+K+SL+RL	WORKL	Key + Record
Add	Blocked	IOAREAL	C+K+D or C+K+SL+RL whichever is larger	WORKL	Record
Retrieve Sequentially	Unblocked	IOAREAS IOAREA2	K+SL+RL	WORKS	Key + Record
Retrieve Sequentially	Blocked	IOAREAS IOAREA2	Data Area or SL+RL, whichever is larger	WORKS	Record
Retrieve Randomly	Unblocked	IOAREAR	SL+RL	WORKR	Record
Retrieve Randomly	Blocked	IOAREAR	Data Area or SL+RL, whichever is larger	WORKR	Record

CODES: C=Count Area=8 bytes
K=Key Area
RL=Record Length
D=Data Area
SL=Sequence-Link Field=10 bytes

FIGURE 15-8 I/O and Work Area Summary for Indexed Sequential Files

each 2314 track. The entire file will then fit on three cylinders. The resulting cylinder index will be so small that it won't fill a whole track and a master index will be unnecessary. The file is then described in two disk extents: one for the cylinder-index track and the other for the three-cylinder, prime data area.

When you look at figure 15-9, first examine the DTFIS for the inventory master file. The file will reside on a 2314 so both DEVICE and HINDEX specify 2314. Reflecting the file-planning decisions, DSKXTNT is coded 2, CYLOFL specifies 1 track per cylinder, and the MSTIND operand is omitted. The INDAREA and INDSIZE operands don't apply to a load function so they too are omitted. KEYLEN is 8 bytes, and because this is a blocked file, KEYLOC must be included. The key, item number, begins in the first position of the record so KEYLOC=1. TYPEFLE and KEYARG are omitted since they apply only to retrieval programs. IOAREAL names the output area (MSTRIO) that I have reserved, and the length of MSTRIO is 716 bytes—8 bytes for count, 8 for key, and 700 for the block of ten records. WORKL names the work area for the master file (MSTRRCD) in which each new record will be constructed. Since the records will be blocked, MSTRRCD reserves enough space for the record, but no extra space for the key.

Prior to the basic processing loop (the last instruction of the housekeeping routine), the program uses the SETFL macro as follows:

```
SETFL INVMSTR
```

This required macro has the filename as its only operand.

```
INVLOAD  START 0
BEGIN    BALR  3,0
         USING *,3
         OPEN  INPUT,INVMSTR
         SETFL INVMSTR                 PREPARE DISK FOR LOADING
READCRD  GET   INPUT
         MVC   MSTRRCD(34),CARD        MOVE 34 BYTES TO OUTPUT AREA
         PACK  MORDQTY,CORDQTY         PACK INPUT FIELDS
         PACK  MORDPT,CORDPT
         PACK  MSAFSTK,CSAFSTK
         PACK  MONHAND,CONHAND
         PACK  MONORD,CONORD
         PACK  MALLOC,CALLOC
         WRITE INVMSTR,NEWKEY          WRITE OUTPUT RECORD
         CLI   INVMSTRC,X'00'          COMPARE STATUS-BYTE TO HEX ZEROS
         BE    READCRD
         DUMP                          PRINT STORAGE CONTENTS
EOFCRD   ENDFL INVMSTR                 FINISH LOAD ROUTINE
         CLOSE INPUT,INVMSTR
         EOJ
INPUT    DTFCD DEVADDR=SYSIPT,IOAREA1=CARD,EOFADDR=EOFCRD
INVMSTR  DTFIS DEVICE=2314,DSKXTNT=2,CYLOFL=1,HINDEX=2314,                X
               RECFORM=FIXBLK,NRECDS=10,RECSIZE=70,KEYLEN=8,KEYLOC=1,     X
               IOROUT=LOAD,IOAREAL=MSTRIO,WORKL=MSTRRCD
CARD     DS    0CL80
CITEM    DS    CL8
CDESC    DS    CL20
CUM      DS    CL4
CORDPOL  DS    CL2
CORDQTY  DS    CL5
CORDPT   DS    CL5
CSAFSTK  DS    CL5
CONHAND  DS    CL5
CONORD   DS    CL5
CALLOC   DS    CL5
         DS    CL16
MSTRRCD  DS    0CL70
MITEM    DS    CL8
MDESC    DS    CL20
MUM      DS    CL4
MORDPOL  DS    CL2
MORDQTY  DS    PL4
MORDPT   DS    PL4
MSAFSTK  DS    PL4
MONHAND  DS    PL4
MONORD   DS    PL4
MALLOC   DS    PL4
         DC    12X'00'
MSTRIO   DS    CL716
         END   BEGIN
```

FIGURE 15-9 ISAM File Creation Program

Bit	Cause	Explanation
0	DASD error	Any uncorrectable DASD error has occurred (except wrong length record).
1	Wrong length record	A wrong length record has been detected during an I/O operation.
2	Prime data area full	The next to the last track of the prime data area has been filled during the load or extension of the data file. The problem program should issue the ENDFL macro and a load extend should be done on the file with new extents given.
3	Cylinder Index area full	The Cylinder Index area is not large enough to contain all the entries needed to index each cylinder specified for the prime data area. This condition can occur during the execution of the SETFL. The user must extend the upper limit of the cylinder index by using a new extent card.
4	Master Index full	The Master Index area is not large enough to contain all the entries needed to index each track of the Cylinder Index. This condition can occur during SETFL. The user must extend the upper limit, if he is creating the file, by using an extent card. Or, he must reorganize the data file and assign a larger area.
5	Duplicate record	The record being loaded is a duplicate of the previous record.
6	Sequence check	The record being loaded is not in the sequential order required for loading.
7	Prime data area overflow	There is not enough space in the prime data area to write an EOF record. This condition can occur during the execution of the ENDFL macro.

FIGURE 15-10 ISAM Error-Status Byte—Bit Meanings for IOROUT=LOAD

When it is executed, it prepares, or formats, portions of the disk for the indexes that will be stored on it.

The main processing loop begins with the GET of the input card. The new master record is then constructed in the work area. To write the record on the ISAM file, the WRITE macro is used as follows:

```
WRITE INVMSTR,NEWKEY
```

Here, operand 1 is the filename and operand 2 is the word NEWKEY. The I/O module then writes the new record on the file. (For blocked files, the I/O module may only add the new record to the block being built in the I/O area until the block is filled. When the block has been filled, it is written on the disk.)

When control is returned to the program following the WRITE macro, the status of the WRITE operation is indicated in a special byte generated by the DTFIS statement. The name of this status byte is the filename followed by the character C—in this case, INVMSTRC. The program checks this byte to find out if the WRITE operation was successful. Figure 15-10 shows how each bit in the status byte is used to indicate a different type of error.

In the sample program, I check only to see if any of the bits are on. If none are on, the new record is assumed to have been written successfully and the program continues by reading the next input card. If any of the bits in INVMSTRC are on, however, the program issues the DUMP macro (described in detail in chapter 11), which causes a storage printout and program termination.

If all the records are successfully loaded and end-of-file is reached on the card-input file, the program issues another

special macro for an ISAM load function: ENDFL. This macro requires only the filename as an operand:

```
ENDFL INVMSTR
```

When executed, this macro causes the last block to be written on the disk (even if it isn't completely filled) and follows that block with an end-of-file record. The ENDFL macro also performs some final operations on the index records.

The status-byte summary in figure 15-10 shows the several types of errors that might take place during execution of a load operation. *DASD* error refers to a Direct-Access Storage Device error—for instance, the device is unable to write the block of data because of a hardware malfunction. If bits 2, 3, 4, or 7 are on, it indicates poor planning because one or more file areas are full. In this case, the program can be run again with larger areas assigned in job-control cards, or the file can be extended (additional records written in an enlarged prime area) by using a modified job-control card and starting the load program where it left off. The duplicate-record and sequence-check bits represent input errors that make it impossible for the load program to store a record properly.

In actual practice, a program would check the error-status byte in more detail. For instance, it might check for duplicate records or sequence errors, print those records on an error list, and then try to continue to load the file. Or it might check for the prime area being full, and if it is, print an error message, issue the ENDFL macro, and end the program. Examples of more extensive status-byte checking are illustrated in the other programs in this topic.

RANDOM RETRIEVAL

Figure 15-11 is a source listing of a simple program that illustrates random retrieval of records from an ISAM file. The program prints the on-hand inventory balance for items in the inventory-master file that was loaded by the load program in figure 15-9. The inventory items to be listed on the report are determined by input cards that have the item number punched in the first eight columns.

Because the cylinder index is going to be processed in storage, INDAREA=INDEX is specified in the DTFIS. Then, INDSIZE=98 is given for the size of this index area (3 cylinders plus 4 equals 7; key length of 8 plus 6 equals 14; 7 times 14 equals 98 bytes for the cylinder index). In the data definitions, the area named INDEX is defined as a 98-byte field to correspond to these DTF operands.

Because the records are blocked, the I/O area is 700 bytes—the length of the data area. Similarly, the work area is the length of one record. Because random processing is used, the DTF keywords for the I/O area and the work area are IOAREAR and WORKR.

The processing flow of the program should be easy for you to follow. After an input card has been read and some portions of the print line have been constructed, the requested item number is moved to the KEYARG field named ITEMKEY. The READ macro then requests the I/O module to read the record randomly by key:

```
READ  INVMSTR,KEY
```

Here, the first operand is the filename, the second operand is the word KEY.

Following the READ macro is the WAITF macro with the filename as its only operand. This macro must follow any ISAM READ or WRITE macro except when loading or extending a file. The WAITF macro causes the program to wait until the I/O operation is completed before continuing.

After reading the record, the program tests the status byte to see if any errors occurred. Figure 15-12 gives the meaning of each of the bits in the status byte, which again has its name made up of the filename followed by the letter C. In this program, I want to print an error message if the desired record can't be found (bit 3 is on). If bit 7 is on, I can ignore it since this means that the record just read came from an overflow area rather than a prime track, and that has no effect on the program. If any of the other bits are on, I'll use the DUMP macro to get a storage printout and end the program.

To test the actual bits, I have used the test-under-mask (TM) instruction. This instruction, which is covered in detail

```
INVRPT   START 0
BEGIN    BALR  3,0
         USING *,3
         OPEN  INPUT,REPORT,INVMSTR
READINP  GET   INPUT
         MVI   LINE,X'40'
         MVC   LINE+1(132),LINE
         MVC   ITEMKEY,IITEM
         MVC   LITEM,IITEM
         READ  INVMSTR,KEY
         WAITF INVMSTR
         TM    INVMSTRC,X'FE'       TEST BITS 0-6
         BZ    FOUND                BRANCH IF BITS 0-6 ARE ALL ZERO
         TM    INVMSTRC,X'EE'       TEST BITS 0-2 AND 4-6
         BZ    NOTFOUND             BRANCH IF THEY ARE ALL ZERO
         DUMP
NOTFOUND MVC   LERR(15),=C'NO RECORD FOUND'
         B     PRINT
FOUND    MVC   LDESC,MDESC
         MVC   LONHAND,PATTRN
         ED    LONHAND,MONHAND
PRINT    CP    LCOUNT,=P'50'
         BL    PRTDET
         PUT   REPORT,HEAD
         MVI   LCTL,C'0'
         ZAP   LCOUNT,=P'0'
PRTDET   PUT   REPORT,LINE
         AP    LCOUNT,=P'1'
         B     READINP
EOFINP   CLOSE INPUT,REPORT,INVMSTR
         EOJ
INPUT    DTFCD DEVADDR=SYSIPT,IOAREA1=INPCRD,EOFADDR=EOFINP
REPORT   DTFPR DEVADDR=SYSLST,IOAREA1=PRTOUT,BLKSIZE=133,CTLCHR=ASA,    X
               WORKA=YES
INVMSTR  DTFIS DEVICE=2314,HINDEX=2314,DSKXTNT=2,CYLOFL=1,              X
               INDAREA=INDEX,INDSIZE=98,RECFORM=FIXBLK,NRECDS=10,       X
               RECSIZE=70,KEYLEN=8,KEYLOC=1,IOROUT=RETRVE,              X
               TYPEFLE=RANDOM,KEYARG=ITEMKEY,IOAREAR=MSTRIO,            X
               WORKR=MSTRRCD
INDEX    DS    CL98
LCOUNT   DC    P'50'
ITEMKEY  DS    CL8
INPCRD   DS    0CL80
IITEM    DS    CL8
         DS    CL72
MSTRRCD  DS    0CL70
MITEM    DS    CL8
MDESC    DS    CL20
         DS    CL18
MONHAND  DS    PL4
         DS    CL20
```

FIGURE 15-11 ISAM Random Retrieval Program

```
LINE     DS    0CL133
LCTL     DS    CL1
         DS    CL5
LITEM    DS    CL8
         DS    CL12
LDESC    DS    CL20
         DS    CL10
LONHAND  DS    CL8
         DS    CL2
LERR     DS    CL15
         DS    CL62
HEAD     DC    CL133'1      ITEM                DESCRIPTION              X
                    ON HAND'
PATTRN   DC    XL8'4020202020202021'
PRTOUT   DS    CL133
MSTRIO   DS    CL700
         END   BEGIN
```

FIGURE 15-11 ISAM Random Retrieval System (Continued)

Bit	Cause	Explanation
0	DASD error	Any uncorrectable DASD error has occurred (except wrong length record).
1	Wrong length record	A wrong length record has been detected during an I/O operation.
2	End of file	The EOF condition has been encountered during execution of the sequential retrieval function.
3	No record found	The record to be retrieved has not been found in the data file. This applies to Random (RANSEQ) and to SETL in SEQNTL (RANSEQ) when KEY is specified, or after GKEY.
4	Illegal ID specified	The ID specified to the SETL in SEQNTL (RANSEQ) is outside the prime data file limits.
5	Duplicate record	The record to be added to the file has a duplicate record key of another record in the file.
6	Overflow area full	An overflow area in a cylinder is full, and no independent overflow area has been specified, or an independent overflow area is full, and the addition cannot be made. The user should assign an independent overflow area or extend the limit.
7	Overflow	The record being processed in one of the retrieval functions (RANDOM/SEQNTL) is an overflow record.

FIGURE 15-12 ISAM Error-Status Byte—Bit Meanings for IOROUT=ADD, RETRVE, or ADDRTR

in chapter 8, lets you test the condition of any bit within a byte. The branch instruction that follows it can then branch according to the resulting condition code.

To illustrate, consider the first two instructions that test the status byte in figure 15-11:

```
TM      INVMSTRC,X'FE'
BZ      FOUND
```

The TM instruction tests the status bits indicated by the immediate operand, X'FE', to see whether they are 1s or 0s. Since hex FE is binary 1111 1110, the first 7 bits (bits 0-6) of INVMSTRC will be tested. Then, if they are all 0s (meaning the record has been read without error), the program will branch to FOUND since the operation code BZ means branch if zeros. If at least one of the bits is on, however, the branch will not take place. Although there are three other operation codes for branching after the TM instruction, the only other one you are likely to need for ISAM processing is BO for branch if all bits tested are ones.

If at least one of the first seven bits is on, the program does not branch and these instructions are reached:

```
TM      INVMSTRC,X'EE'
BZ      NOTFOUND
```

Since hex EE is binary 1110 1110, all bits but the "not found" and "overflow record" bits will be tested. If they are all 0s it means the previous BZ instruction didn't branch because the "not found" bit was on. Then, the program branches to the routine named NOTFOUND. If this second BZ operation doesn't branch, it indicates one of the other errors and the program executes the DUMP macro.

If you keep the above points in mind, this program should present no problems for you. The NOTFOUND routine moves a message to the output line and prints it. The PRINT routine uses line-counting logic, and the EOF routine for the card input file closes the files and ends the job.

To update records after retrieving them randomly presents no additional problems. Usually, the record has just been read so the KEYARG field still contains the record key. A WRITE macro, with KEY as the second operand, can then be coded to write the record on the disk in its original location. For instance,

```
WRITE INVMSTR,KEY
```

will write an updated inventory record back on the disk. After the WRITE macro, a WAITF macro must be issued to make sure that the operation has been completed before processing continues.

SEQUENTIAL RETRIEVAL

To illustrate sequential processing, I've written a program that reads the inventory-master file created in figure 15-9 and prints an inventory-status report from it. This program is illustrated in figure 15-13. Since the logic of the program is straightforward, you should have no difficulty following it once you learn the new elements it presents.

Since the attributes of the master file are the same as those for random retrieval, many of the DTFIS operands are the same. For sequential processing, however, the cylinder index is not brought into storage because it isn't accessed often. Therefore, the INDAREA and INDSIZE operands are omitted. IOROUT still specifies RETRVE, but TYPEFLE must now be changed to SEQNTL, and KEYARG can be omitted. Also, the keyword IOAREAS replaces the keyword IOAREAR, and WORKS=YES replaces WORKR=MSTRRCD.

The I/O macros used to perform the sequential processing are the familiar GET and PUT. They are coded in the same way as for card, tape, or sequential disk files. In this program, only the GET is used since no update is performed. If the records were written back on the disk, however, PUT would be used. Since I specified WORKS=YES in the DTFIS, the GET includes a second operand that names the record work area.

Before a program can begin sequential processing, the SETL macro must be issued. This macro can initialize sequential processing in four ways. The second operand of the macro (the first operand is filename) specifies which of the four ways you want to use as follows:

```
SETL  filename,BOF
SETL  filename,KEY
SETL  filename,GKEY
SETL  filename,idname
```

The most common starting point (and the one used in figure 15-13) is at the first record of the file. This is specified by coding BOF as the second operand.

You may also want to start processing at some particular key in the file. The KEYARG operand must then be coded. Before the SETL is issued with KEY as the second operand, the starting key must be placed in the KEYARG field.

Rather than use a specific key, you can also specify a *generic key*. For example, if a file contains account master records keyed by account number, you can specify that processing is to start with the first account in the 1000 series without knowing if the first key is 1002, 1009, or whatever. The KEYARG field is then set to 1000 at the start of the program, and the SETL macro is coded with GKEY as the second operand.

The last form of the SETL macro specifies the starting point by giving the name of the field that contains the disk address of the first record to be processed. In actual practice, this form of SETL is almost never used, so you don't have to worry about it.

To end the processing of a sequential file, the ESETL macro is used. This macro, which has the filename as its only operand, is used in combination with the SETL macro. In figure 15-13, the ESETL macro is coded in the EOF routine for the sequential file to indicate that all records in the file have been processed.

In some programs, SETL and ESETL are coded several times in order to start and end processing at several different points in the file. First one section of the file is processed, then another section—perhaps located ahead of the section first read, perhaps located farther into the file—is also processed sequentially. This type of processing is called *skip-sequential processing*. For example, the records from key 1000-1500 might be processed, followed by the records from keys 500-750, followed by keys 2800-2850. The starting and ending keys for each section can be given by input cards that are read at the start of the program.

To find the end-of-file for an ISAM file, the programmer must check the status byte after each GET to see if bit number 2 has been turned on (see the bit summary in figure 15-12). If it is on, the program must branch to an end-of-file routine. In figure 15-13, this status-byte testing takes place after the GET is issued:

```
TM    INVMSTRC,X'FE'
BZ    PROCESS
TM    INVMSTRC,X'20'
BO    EOFMSTR
DUMP
```

The first TM instruction tests whether any of bits 0-6 are on. If all are 0, the branch to PROCESS takes place and the mainline routine continues. If not, the second TM instruction tests bit 2. If it is on indicating end-of-file, the program branches to the EOF routine. Otherwise, some error is indicated, the DUMP macro is issued, and the program ends.

The rest of the program should be clear to you. The program moves data to the printer work area and prints a line using a line-counting technique for forms overflow. One point of interest is the editing of the six output fields starting with RORDQTY. Since the edit pattern uses field separators (hex 22s) and since all six packed fields are adjacent in storage, only one edit instruction is needed. This use of field separators is described in detail in chapter 8.

FILE ADDITIONS

Adding records to an indexed sequential file is almost like loading them. Figure 15-14 illustrates a program that adds records to the inventory-master file created in figure 15-9. By comparing the load and add programs, you can see their many similarities. In fact, there are only four significant differences between them.

First, the DTFIS is slightly different in the add program. The IOROUT operand specifies ADD instead of LOAD, but the rest of the operands including IOAREAL and WORKL are

```
INVSTAT  START 0
BEGIN    BALR  3,0
         USING *,3
         OPEN  STATUS,INVMSTR
         SETL  INVMSTR,BOF          START PROCESSING WITH FIRST RECORD
NXTRCD   GET   INVMSTR,MSTRRCD
         TM    INVMSTRC,X'FE'       TEST BITS 0-6
         BZ    PROCESS              IF BITS ARE ZERO, CONTINUE
         TM    INVMSTRC,X'20'       TEST BIT 5
         BO    EOFMSTR              IF BIT 5 IS ON, GO TO EOF ROUTINE
         DUMP                       DUMP STORAGE AND END
PROCESS  MVI   RLINE,X'40'
         MVC   RLINE+1(132),RLINE
         MVC   RITEM,MITEM
         MVC   RDESC,MDESC
         MVC   RUM,MUM
         MVC   RORDPOL,MORDPOL
         MVC   RORDQTY(61),PATTRN
         ED    RORDQTY(61),MORDQTY  EDIT SIX FIELDS
         ZAP   AVAILWK,MONHAND
         AP    AVAILWK,MONORD
         SP    AVAILWK,MALLOC
         MVC   RAVAIL,PATTRN
         ED    RAVAIL,AVAILWK
         CP    LCOUNT,=P'50'
         BL    PRTDET
         PUT   STATUS,RHEAD
         ZAP   LCOUNT,=P'0'
         MVI   RCTL,C'0'
PRTDET   PUT   STATUS,RLINE
         AP    LCOUNT,=P'1'
         B     NXTRCD
EOFMSTR  ESETL INVMSTR
         CLOSE STATUS,INVMSTR
         EOJ
STATUS   DTFPR DEVADDR=SYSLST,IOAREA1=PRTOUT,BLKSIZE=133,CTLCHR=ASA,    X
               WORKA=YES
INVMSTR  DTFIS DEVICE=2314,HINDEX=2314,DSKXTNT=2,CYLOFL=1,              X
               RECFORM=FIXBLK,NRECDS=10,RECSIZE=70,KEYLEN=8,KEYLOC=1,   X
               IOROUT=RETRVE,TYPEFLE=SEQNTL,IOAREAS=MSTRIO,WORKS=YES
LCOUNT   DC    P'50'
PATTRN   DS    0CL61
         DC    X'40'
         DC    6X'2020202020202122222'
```

FIGURE 15-13 ISAM Sequential Retrieval Program

the same. (INDAREA and INDSIZE aren't used because they are illegal for file additions.) Second, SETFL and ENDFL aren't required for the add program. Third, although the NEWKEY form of the WRITE macro is used to add records, the bits in the status byte for the add program have the meanings given in figure 15-12, not those given in figure 15-10. Fourth, for file additions, the WAITF macro must follow the WRITE macro, but the WAITF macro isn't required for file loading.

In figure 15-14, the status-byte checking routine proceeds

```
AVAILWK  DS    PL4
RHEAD    DC    CL105'1
                OP          ITEM      DESCRIPTION                U/M   X
                      ORDQTY    ORDPNT     SAFSTK     ONHAND     ONORDR'
         DC    CL28'      ALLOC     AVAIL'
MSTRRCD  DS    0CL70
MITEM    DS    CL8
MDESC    DS    CL20
MUM      DS    CL4
MORDPOL  DS    CL2
MORDQTY  DS    PL4
MORDPT   DS    PL4
MSAFSTX  DS    PL4
MONHAND  DS    PL4
MONORD   DS    PL4
MALLOC   DS    PL4
         DS    CL12
RLINE    DS    0CL133
RCTL     DS    CL1
         DS    CL10
RITEM    DS    CL8
         DS    CL2
RDESC    DS    CL20
         DS    CL4
RUM      DS    CL4
         DS    CL2
RORDPOL  DS    CL2
         DS    CL4
RORDQTY  DS    CL8
         DS    CL2
RORDPT   DS    CL8
         DS    CL2
RSAFSTK  DS    CL8
         DS    CL2
RONHAND  DS    CL8
         DS    CL2
RONORD   DS    CL8
         DS    CL2
RALLOC   DS    CL8
         DS    CL2
RAVAIL   DS    CL8
         DS    CL8
PRTOUT   DS    CL133
MSTRIO   DS    CL700
         END   BEGIN
```

FIGURE 15-13 ISAM Sequential Retrieval Program (Continued)

in several stages. First, filenameC (INVMSTRC) is compared to binary zero. If none of the bits in filenameC are on, the record has been added successfully, and the program goes on to the next input record. In the second stage of error checking, a TM instruction with a mask of hex F8 tests to see if any bits other than bits 5, 6, or 7 are on. If any are on, an I/O error has occurred and a DUMP macro is executed. In the third stage of error checking, the TM instruction uses a mask of hex 02 to test to see if bit number 6 is on. If it is on, it indicates that the record to be added has a key that is

```
INVADD   START 0
BEGIN    BALR  3,0
         USING *,3
         OPEN  CARD,LOG,INVMSTR
READCRD  GET   CARD
         MVI   LINE,X'40'
         MVC   LINE+1(133),LINE
         MVC   LITEM,CITEM
         MVC   LDESC,CDESC
         MVC   MSTRRCD(34),CRD
         PACK  MORDQTY,CORDQTY
         PACK  MORDPT,CORDPT
         PACK  MSAFSTK,CSAFSTK
         PACK  MONHAND,CONHAND
         PACK  MONORD,CONORD
         PACK  MALLOC,CALLOC
         WRITE INVMSTR,NEWKEY      ADD RECORD TO FILE
         WAITF INVMSTR
         CLI   INVMSTRC,X'00'      TEST STATUS BYTE
         BNE   ERRCHK1             IF ANY BIT IS ON, GO TO ERRCHK1
         MVC   LERR(5),=C'ADDED'   MOVE MESSAGE TO PRINTER WORK AREA
         B     PRINT
ERRCHK1  TM    INVMSTRC,X'F8'      TEST BITS 0-4
         BZ    ERRCHK2             IF ALL ARE ZERO, GO TO ERRCHK2
         DUMP
ERRCHK2  TM    INVMSTRC,X'02'            TEST BIT 6
         BO    FILEFULL                  IF ON, GO TO FILEFULL
         MVC   LERR(13),=C'DUPLICATE RCD' IF OFF, MOVE MESSAGE
         B     PRINT                        TO WORK AREA AND CONTINUE
FILEFULL MVC   LERR(9),=C'FILE FULL'     MOVE ERROR MESSAGE TO WORK
         PUT   LOG,LINE                     AREA, PRINT IT, AND EOJ
         B     EOFCRD
PRINT    CP    LCOUNT,=P'50'
         BL    PRTDET
         PUT   LOG,HEAD
         ZAP   LCOUNT,=P'0'
         MVI   LCTL,C'0'
PRTDET   PUT   LOG,LINE
         AP    LCOUNT,=P'1'
         B     READCRD
EOFCRD   CLOSE CARD,LOG,INVMSTR
         EOJ
```

FIGURE 15-14 ISAM File Addition Program

a duplicate of one already on the file. If this is the case, an error message is added to the print line, the line is printed, and the program goes on to the next input record. If bit 6 is on, it indicates that the record could not be added because there was no room for it. Depending on how the overflow areas were allocated originally, the cylinder-overflow tracks for the cylinder involved could be full, the general overflow area could be full, or both. In any case, if bit 6 is on an error message is printed and the program ends. Before the program is ended, however, the files are closed so all indexes are updated and the file can be used as input to create the same file with adequate disk space allocated.

```
CARD     DTFCD DEVADDR=SYSIPT,IOAREA1=CRD,EOFADDR=EOFCRD
LCG      DTFPR DEVADDR=SYSLST,IOAREA1=PRTOUT,BLKSIZE=133,CTLCHR=ASA,    X
               WORKA=YES
INVMSTR  DTFIS DEVICE=2314,HINDEX=2314,DSKXTNT=2,CYLOFL=1,              X
               RECFORM=FIXBLK,NRECDS=10,RECSIZE=70,KEYLEN=8,KEYLOC=1,   X
               IOROUT=ADD,IOAREAL=MSTRIO,WORKL=MSTRRCD
LCOUNT   DC    P'50'
MSTRRCD  DS    0CL70
MITEM    DS    CL8
MDESC    DS    CL20
MUM      DS    CL4
MORDPOL  DS    CL2
MORDQTY  DS    PL4
MORDPT   DS    PL4
MSAFSTK  DS    PL4
MONHAND  DS    PL4
MONORD   DS    PL4
MALLOC   DS    PL4
         DC    12X'00'
CRD      DS    0CL80
CITEM    DS    CL8
CDESC    DS    CL20
CUM      DS    CL4
CORDPOL  DS    CL2
CORDQTY  DS    CL5
CORDPT   DS    CL5
CSAFSTK  DS    CL5
CONHAND  DS    CL5
CONORD   DS    CL5
CALLOC   DS    CL5
         DS    CL16
LINE     DS    0CL133
LCTL     DS    CL1
         DS    CL10
LITEM    DS    CL8
         DS    CL2
LDESC    DS    CL20
         DS    CL2
LERR     DS    CL5
         DS    CL86
HEAD     DC    CL133'1          ITEM     DESCRIPTION          ACTION'
PRTOUT   DS    CL133
MSTRIO   DS    CL716
         END   BEGIN
```

FIGURE 15-14 ISAM File Addition Program (Continued)

FILE DELETIONS

In the I/O module there is no direct provision for deleting records from an ISAM file. However, a widely used standard technique has been developed to simulate deletion. One byte in the record (usually the first byte of the record unless it is part of the key) is reserved for a delete code. To delete a record, then, a program fills the delete-code byte with some particular code—hex FF is most commonly used—thus indicating that the record has been deleted. All of the programs that process the file must ignore any records that have the delete code. Note, however, that the deleted

Loading	Additions	Sequential Retrieval and Update	Random Retrieval and Update	Random Additions and Retrieval
SETFL filename WRITE filename,NEWKEY ENDFL filename	WRITE filename,NEWKEY WAITF filename	SETL filename,BOF SETL filename,KEY SETL filename,GKEY GET filename GET filename,workarea PUT filename PUT filename,workarea ESETL filename	READ filename,KEY WAITF filename WRITE filename,KEY WAITF filename	READ filename,KEY WAITF filename WRITE filename,KEY WAITF filename WRITE filename,NEWKEY WAITF filename

FIGURE 15-15 ISAM I/O Macro Summary

records actually remain on the file until it is reorganized. To reorganize the file, a reload program reads the old file ignoring records with the delete code, and loads a new file. Thus, deleted records are dropped from the file.

CONCLUSION

As you can see from this presentation, the trick to processing ISAM files in assembler language is coding the DTF and other I/O macros properly. To this end, the ISAM I/O macros are summarized in figure 15-15. If you check this list, you can avoid omitting a required macro. Notice that the WAITF macro is required following all READ and WRITE macros except when loading a file. It is not required when using GET or PUT.

If you are familiar with ISAM coding in COBOL, you may wonder why anybody would bother with BAL code for these files. In truth, the BAL code is more complex and leaves more chance for error than COBOL. Nevertheless, there are two reasons why BAL code will continue to be used. First, the BAL routines are likely to require much less storage than comparable COBOL routines. Second, BAL lets you do things with ISAM files that can't be done in COBOL. For instance, the IBM COBOL compilers don't provide for skip-sequential processing. As a result, many BAL subprograms for this function have been linked to COBOL main programs. Similarly, a COBOL program can't process an ISAM file both sequentially and randomly in the same program. To get this added control of ISAM files, a BAL program or subprogram is likely to be used.

Terminology

embedded key
to load a file
DASD
generic key
skip-sequential processing

Objective

Given a problem involving indexed sequential files, code a BAL solution. The problem may require any of the following: loading a file, adding records to a file, deleting records from a file, sequentially or randomly updating a file, and sequentially or randomly retrieving records from a file. The problem may require that cylinder overflow, general overflow, or the cylinder index in storage option be used.

```
ISREORG  START 0
BEGIN    BALR  3,0
         USING *,3
         OPEN  OLDMSTR,NEWMSTR
         SETL  OLDMSTR,BOF
         SETFL NEWMSTR
GETOLD   GET   OLDMSTR,NEWRCD          READ OLD MSTR FILE SEQUENTIALLY
         TM    OLDMSTRC,X'FE'          CHECK FOR ANY ERROR BITS
         BZ    GOODREAD                IF NONE, READ WAS GOOD
         TM    OLDMSTRC,X'20'          IF END OF FILE BIT IS ON
         BO    EOFOLDM                 GO TO EOF ROUTINE
         DUMP                          ELSE DUMP TO SHOW ERROR
GOODREAD CLI   DELBYTE,X'FF'           IF RECORD IS TO BE DELETED,
         BE    GETOLD                  BYPASS AND READ NEXT MSTR RCD
         WRITE NEWMSTR,NEWKEY          WRITE NEW MSTR RCD
         TM    NEWMSTRC,X'FF'          IF NO I/O ERRORS,
         BZ    GETOLD                  GO READ NXT OLD MSTR RCD,
         DUMP                          ELSE  DUMP
EOFOLDM  ESETL OLDMSTR
         ENDFL NEWMSTR
         CLOSE OLDMSTR,NEWMSTR
         ECJ
OLDMSTR  DTFIS DEVICE=2314,DSKXTNT=3,CYLOFL=1,HINDEX=2314,                X
               RECFORM=FIXBLK,NRECDS=10,RECSIZE=70,KEYLEN=8,KEYLOC=1,     X
               IOROUT=RETRVE,TYPEFLE=SEQNTL,IOAREAS=OLDINP1,              X
               IOAREA2=OLDINP2,WORKS=YES
NEWMSTR  DTFIS DEVICE=2314,DSKXTNT=3,CYLOFL=1,HINDEX=2314,                X
               RECFORM=FIXBLK,NRECDS=10,RECSIZE=70,KEYLEN=8,KEYLOC=1,     X
               IOROUT=LOAD,IOAREAL=NEWOUT1,IOAREA2=NEWOUT2,WORKL=NEWRCD
NEWRCD   DS    0CL70
         DS    CL69
DELBYTE  DS    CL1
OLDINP1  DS    CL700
OLDINP2  DS    CL700
NEWOUT1  DS    CL716
NEWOUT2  DS    CL716
         END   BEGIN
```

FIGURE 15-16 ISAM File Reorganization Program

Problem

Assume that many additions and deletions have been made to the inventory master file that has been created and processed by the programs in this topic. To indicate a deletion, hex FF has been placed in the last byte of a record. Write a program that reorganizes this file, dropping deleted records and returning all active records to the prime data area. The input is the old master file; the output is the new master file.

Solution

Figure 15-16 is an acceptable solution.

TOPIC THREE Direct Files Because of the complexities of direct-file processing—both in coding and in logic—direct-file organization is used infrequently. As a result, there are many BAL programmers who have never written

processing routines for direct files. Nevertheless, there are certain file-handling problems for which direct-file organization is well suited. When this is the case, BAL is generally used because it allows the programmer to process direct files in ways that are impossible in any other language.

Because direct files can be processed in many different ways, I am making no attempt to illustrate the most common processing patterns for direct files as I did with indexed sequential files. Instead, this topic presents all of the coding elements related to direct-file processing with the hope that the programmer will then be able to apply the elements to any direct-file problem he may face. First, the required formats for disk addresses are presented; second, the I/O macros for direct files are presented; third, the DTFDA (DTF for Direct-Access file) operands are presented. After this presentation of the coding elements, two sample programs are presented illustrating the creation and processing of a typical direct file.

DISK ADDRESSES

The records in a direct file have no sequence as far as the I/O module is concerned. They can be accessed only by referring to their physical disk locations—their disk addresses. The programmer must therefore supply the proper disk address to the I/O module each time a record is read or written.

You can pass the disk address to the I/O module in any one of three formats. These formats are illustrated in figure 15-17. The first format is cylinder and head number, plus a record number or record key. These values are supplied as binary values in an eight-byte format. The first three bytes (numbered 0, 1, and 2) are always zero for 2314 and 3330 devices. The fourth and fifth bytes (numbers 3 and 4) must specify the cylinder number. For a 2314, the cylinder value ranges from hex 0000 to hex 00C7 (decimal 0 through 199). For 3330 files, the range is hex 0000 through hex 0193 (decimal 0 through 403) for the basic model. Byte numbers 5 and 6 must contain the head number. The valid range for head number is hex 0000 through hex 0013 (decimal 0 through 19) for the 2314 and hex 0000 through hex 0012 for the 3330. (Only 19 tracks are available for data on the 3330 because one track is used as the synchronizing track.) The last byte of the eight-byte disk address, byte 7, is for record number, which must also be specified as a hex value. If the records are to be referenced by keys, record number must be set to zero.

The second disk-address format is *relative-track number* plus record number, both as zoned-decimal values. The first eight bytes of this ten-byte format specify a track number relative to the start of the file. For example, if you wanted to read a record located on track zero of cylinder 62 within a file that began on track zero of cylinder 60 on a 2314, the value in the relative-track-number field should be 40. For a record on track 15 of cylinder 63, the relative-track number should be 75. The last two bytes of this format must contain the record number as a decimal value. Again, if the records are to be referred to by key, the record number should be zero.

The relative-track format offers an advantage over the cylinder-and-track format because you don't have to be concerned with the actual cylinder number. As a result, you don't have to know the actual disk area assigned to the file. The reference to relative track 40, for instance, will search the proper track whether the file begins on cylinder 60 or cylinder 128.

The third address format shown in figure 15-17 is also a relative-track number. In this format, however, the values are in hex rather than zoned decimal. Bytes 0-2 must contain the relative track number in hex, and byte 3 must hold the record number.

Relative-track addressing has been supported by DOS for only a few years. Most new programs that use direct files should use relative-track addressing because it is easier, but most older programs that use direct files (including ones supplied by IBM) were coded to use the format-1 disk addresses (cylinder and head). Use relative-track addresses for all new programs, then, but be sure you understand the format-1 addresses in case you have to deal with an older program.

I/O MACROS

Direct files are processed through READ and WRITE macros. If you want to refer to a record by record number (also called reference by ID), you code:

```
   READ  filename,ID
or WRITE filename,ID
```

In this case, the program must supply the proper record ID (track address and record number) in the format being used.

If you want to refer to the records by key (the file must be in count-key-data format), you code:

```
   READ  filename,KEY
or WRITE filename,KEY
```

Again, the disk-address field must contain the proper track address, but the record number must be set to zero. Also, another field, called the KEYARG field, must contain the key of the record to be read or written.

Two other formats of the WRITE macro are also used in processing direct files. Before creating a direct file, you may need to clear the disk area of previous data. In such a case, this special form of the WRITE macro is used:

```
WRITE filename,RZERO
```

With RZERO as the second operand, the WRITE macro causes the track specified in the disk-address field to be erased and made available for new data. This includes resetting the fields in record zero of the track to indicate that no records are stored on the track. (Remember that record zero is never used for storing data; it is used by the DOS modules.) If you don't use this RZERO method of clearing a disk area before creating a file, it must be done by using one of the programs of the Disk Operating System called the *clear-disk utility*. As you will learn later, there are differences in these two methods of clearing a disk area that will affect the logic of the file-creation program.

The other special form of the WRITE macro allows you to write a record in the next available space on a track without knowing which record number it will be. This macro is coded:

Format 1: Physical Disk Address, 8 Bytes (MBBCCHHR), Hex

Bytes	Content in Hex	Information
0	M	Module number (Usually X'00' for disk)
1,2	BB	Bin number (Always X'0000' for disk)
3,4	CC	Cylinder number (X'0000' to X'00C7' for 2314, X'0000' to X'0193' for 3330)
5,6	HH	Head number (Track number) (X'0000' to X'0013' for 2314, X'0000' to X'0012' for 3330)
7	R	Record number (Sequential number of record on the track, X'00' to X'FF' Zero for reference by key)

Format 2: Relative Track Address, 10 Bytes (TTTTTTTTRR), Zoned Decimal

Bytes	Content in ZD	Information
0–7	TTTTTTTT	Relative Track number (Track number relative to the first track of the file.)
8,9	RR	Record number (Sequential number of record on the track. Zero for reference by key.)

Format 3: Relative Track Address, 4 Bytes (TTTR), Hex

Bytes	Content in Hex	Information
0–2	TTT	Relative track number (Track number relative to the first one of the file.)
3	R	Record number (Sequential number of record on the track. Zero for key reference.)

FIGURE 15-17 Disk Address Formats

Priority	Keyword	Programmer Code	Default	Remarks
Required	BLKSIZE	Length of I/O area		See IOAREA1 for length.
Required	DEVICE	2311 2314 3330	2311	
Required	TYPEFLE	INPUT OUTPUT	INPUT	
Required	IOAREA1	Name of I/O area		Length of I/O area must include 8 bytes for count if AFTER=YES is coded; must include bytes for key if KEYLEN is coded.
Required	ERRBYTE	Name of two-byte error-code field reserved by user		User must define two-byte field for I/O module to post error indications. See figure 15-20 for description of codes.
Required	SEEKADR	Name of user disk-address field		Length of address field depends on address format: 8 bytes for physical address, 10 for decimal relative-track address, 4 for hex relative-track address.
Optional	IDLOC	Name of disk-address field for address returned after READ or WRITE		Length of IDLOC must be same as SEEKADR field. Address of record read or written will be returned in same format as SEEKADR.
Optional	DSKXTNT	Max number of extents for this file		Indicates that relative-track-address format is used. RELTYPE specifies hex or decimal format.
Optional	RELTYPE	HEX DEC		Specifies format of relative-track address as hex (4 bytes) or decimal (10 bytes).
Optional	RECFORM	FIXUNB VARUNB UNDEF		If undefined (UNDEF) is coded, RECSIZE must be coded.

FIGURE 15-18 DFTDA Operand Summary

```
WRITE filename,AFTER
```

When it has a third operand of EOF, it causes an end-of-file record to be written in the next available record position:

```
WRITE filename,AFTER,EOF
```

The end-of-file record is used if the file is processed sequentially by some other program.

The last macro required when processing direct files is the WAITF macro. This macro must follow each READ or WRITE macro that you use. Its format is:

```
WAITF filename
```

This macro is necessary because the I/O modules used for direct processing do not wait for the I/O operation to be completed before returning control to a program. For example, if you issue a READ macro and follow it with

Priority	Keyword	Programmer Code	Default	Remarks
Optional	RECSIZE	Register number (nn)		Required if RECFORM=UNDEF. Register must contain binary record length before WRITE, will hold length of record after READ.
Optional	KEYARG	Name of user key field		Used only if KEYLEN is coded. Length of KEYARG field should be equal to key length. KEY must be placed here before READ or WRITE by key.
Optional	KEYLEN	Length of Key		Used only if file in count-key-data format.
Optional	XTNTXIT	Label of first instruction in extent processing routine.		Allows user to code routine to process or save extent information. See Chapter 15 for details.
Optional	READID	YES		READ filename,ID will be used.
Optional	READKEY	YES		READ filename,KEY will be used.
Optional	SRCHM	YES		Request search of multiple tracks to end of cylinder for READ or WRITE by KEY.
Optional	WRITEID	YES		WRITE filename,ID will be used.
Optional	WRITEKY	YES		WRITE filename,KEY will be used.
Optional	AFTER	YES		WRITE filename,AFTER or RZERO will be used.
Optional	VERIFY	YES		Request write verification of output records.

FIGURE 15-18 DTFDA Operand Summary (Continued)

instructions that are to process the data just read, you will find that the data has not yet arrived in storage when the instructions are executed. The WAITF macro stops execution of your program until the previous I/O operation has been completed.

DTFDA OPERANDS

The most widely used DTFDA operands are summarized in figure 15-18. Although this summary is largely self-explanatory, explanation of some of the operands follows.

IOAREA1 The label of the I/O area is coded in this operand. To determine the number of bytes required in the I/O area for the file, use the table in figure 15-19. The contents of the I/O area are determined by the disk format (count-data or count-key-data) and by the I/O macro used. If more than one macro is used for one direct file, the I/O area must provide for the largest possible I/O content. If, for example, a program uses the WRITE AFTER and WRITE KEY macros for one file, the I/O area must provide for count, key, and data areas. The program must then process the varying contents of the I/O area by using appropriate dummy sections or work areas.

Macro Instruction	I/O Area Contents	
	With KEYLEN	Without KEYLEN
READ filename,KEY	Data	
READ filename,ID	Key and Data	Data
WRITE filename,KEY	Data	
WRITE filename,ID	Key and Data	Data
WRITE filename,RZERO	Not Used	Not Used
WRITE filename,AFTER	Count, Key, and Data	Count and Data

FIGURE 15-19 I/O Area Contents for Direct Files

RECFORM Direct files are always unblocked. (You may wish to do some blocking and deblocking of your own, but the I/O module regards all records as unblocked.) The records can be fixed-length, variable-length, or undefined. If VARUNB is specified, the programmer is responsible for calculating the record and block lengths and placing them, in the appropriate format, in the first eight bytes of the I/O area. This variable-length format, which is the same as that described for tape files, is rarely used.

When UNDEF is specified, it means the programmer does not know what type of records will be processed. This code is normally used in programs that are designed to work with a variety of files. If UNDEF is coded, the RECSIZE operand must also be included. In the majority of cases, FIXUNB is used.

RECSIZE If RECFORM=UNDEF, the RECSIZE operand must be included and must specify a register number. When an undefined record is read, the I/O module puts the length of the record into this register as a binary value. Similarly, before an undefined record is written, the length of the record must be put into this register.

KEYLEN The KEYLEN operand must specify the length of the keys if the file is in count-key-data format. It snould be omitted if the file is in count-data format.

ERRBYTE The ERRBYTE operand gives the name of a two-byte field (defined by your program) that can be used by the I/O module to indicate processing errors for a direct file. Figure 15-20 lists all of the error conditions that can be posted. The sample programs in this topic will help you understand which type of error is applicable to which type of READ or WRITE macro.

SEEKADR The SEEKADR operand specifies the label of the disk address field. The field must be of proper length for the type of reference that you selected: eight bytes for cylinder and head addresses, ten bytes for relative track in decimal, and four bytes for relative track in hex.

KEYARG The KEYARG operand should be coded for files in count-key-data format that are processed using the KEY

Byte	Bit	Meaning
FIRST BYTE	0	(Not used)
	1	Wrong Length Record
	2	Nondata transfer (hardware I/O error)
	3	(Not Used)
	4	No room found (WRITE AFTER only)
	5	(Not used)
	6	(Not used)
	7	Reference outside file extents
SECOND BYTE	0	Data check in count area
	1	Track overrun
	2	End of cylinder (SRCHM)
	3	Data check in reading key or data
	4	No record found
	5	End-of-file record read
	6	End-of-volume reached (multi-volume files)
	7	(Not used)

FIGURE 15-20 ERRBYTE Bit Settings

form of READ or WRITE. It names a field in which the program must place the key of the desired record before issuing an I/O macro. If a count-key-data file is read by ID, the key of the record that is read will be placed in this field by the I/O module.

RELTYPE If relative-track addresses are used, this entry is coded either DEC—to indicate that the 10-byte decimal format is used—or HEX—to indicate the four-byte hex format.

DSKXTNT The DSKXTNT operand must be included when relative-track addresses are used. It should be omitted when format-1 addresses are used. The entry specifies the number of extents assigned for a file—usually, one.

SRCHM The SRCHM operand stands for *search multiple tracks*. If the file is in count-key-data format, this operand will cause a READ or WRITE by KEY to search beyond the track indicated in the disk address. If the specified key is not found on the indicated track, the search will continue to subsequent tracks until the record is found or the end of the cylinder is reached.

IDLOC If the IDLOC operand is coded, the I/O module will place the ID (disk address) of a record in the field specified here after each READ or WRITE operation. The returned ID will be in the same format as that used by the SEEKADR field. The ID will be the address of the next record in the file except when the KEY form of the READ or WRITE is used with the multiple-track search option. In this case, the address of the record just read or written is placed in the IDLOC field. These returned addresses are summarized by the table in figure 15-21. If a record isn't found, an error indication will be posted to the ERRBYTE field and the returned ID is unpredictable.

XTNTXIT If the XTNTXIT operand is included, control is given to the instruction named by the label coded in this operand during the OPEN process. At this time, register 1 will have the address of a 14-byte field that contains the

Macro Instruction	ID Supplied	
	With SRCHM	Without SRCHM
`READ filename,KEY`	Same record	Next record
`READ filename,ID`	Next record	Next record
`WRITE filename,KEY`	Same record	Next record
`WRITE filename,ID`	Next record	Next record

FIGURE 15-21 ID Supplied in IDLOC after READ or WRITE

extents of the file in the form shown in figure 15-22. This extent information can then be saved for use in file processing routines. In general, this extent processing is useful only when format-1 disk addresses are used. The lower extent captured by the XTNTXIT routine can be used as a basis for the randomizing routine. To end the XTNTXIT routine and return to the OPEN routine, the LBRET macro must be used. It is coded with no operands:

```
LBRET
```

SAMPLE PROGRAMS

As you can begin to appreciate from seeing the I/O macros and DTFDA statement, direct files are more difficult to use

Bytes	Contents
0	Extent type code as specified in the EXTENT job-control card
1	Extent sequence number
2–5	Lower limit of the extent in binary (CCHH)
6–9	Upper limit of the extent in binary (CCHH)
10–11	Symbolic unit number in binary
12–13	Not used

NOTE: At entry to user's XTNTXIT routine, register 1 contains the address of the 14-byte area shown above.

FIGURE 15-22 Format of Extent Area for XTNTXIT Routine

than sequential files or indexed sequential files. The DTF is more complex, the I/O macros are a little more complicated, and the programmer himself must develop the proper disk address. In addition, he must provide all of the file-handling logic. Because of these difficulties, direct files aren't popular. In some cases, however, a direct file can offer a good solution to a particular programming need. The two sample programs that follow illustrate just such a case. In fact, they are based on a real situation.

A certain manufacturing company had all of its factory labor reported by the manufacturing department in which it was done. At one point in the processing of this data, however, it was necessary to assign the labor costs to general ledger accounts. Unfortunately, the table that related the department number to the account number was too big—about 125 entries and 1000 bytes—to be loaded into core storage with some of the programs that used it. On the other hand, an indexed sequential file to allow random lookup of account number using department number as key would have been very inefficient due to the multiple searches and reads of index records for such a small number of data records.

A direct file in count-key-data format is a good solution for such a problem. The key can be department number (four bytes) and the data can be the account number (also four bytes). Since a 2314 track can hold 48 of these short records, the entire file will fit on three tracks with room for up to 144 departments. Then, by specifying the multiple-track search option and setting the disk address to the first track of the file, all three tracks of the file can be searched for a particular key by using just one READ macro.

The first sample program, shown in figure 15-23, creates this direct Department/Account file. It creates each of the records from a file of cards that relates each department number to one general ledger account number.

In the DTFDA, the KEYLEN operand must be included to indicate count-key-data format; it is coded with the key length of four bytes. Since I will use the AFTER form of WRITE in this program, the I/O area must include eight bytes for the count field, four for the key (department number), and four for the data (account number). The BLKSIZE entry is therefore 16. DAERRS is coded in the ERRBYTE operand to indicate the name of my two-byte error-indication field. The name of my area for the disk address is TRKADDR, which is coded in the SEEKADR operand. Since I have no need in this program for a returned disk address, IDLOC is omitted. The KEYARG operand is also omitted because I will be using only the RZERO and AFTER forms of WRITE. Since I decided to use the hex form of relative-track addressing, HEX is used in the RELTYPE operand and the XTNTXIT operand is omitted (because I don't need to know the file extents). The DSKXTNT operand must be included for relative-track addressing; it is coded 1. Of the remaining possible operands, only AFTER=YES is coded. This is done so I can issue the RZERO and AFTER forms of the WRITE macro.

Notice that the definition of the I/O area, DAOUT, corresponds to the BLKSIZE operand. Space has been reserved for the count, key, and data portions of the record as was indicated by the summary in figure 15-19. Similarly, the disk address field, TRKADDR, is coordinated with the RELTYPE operand. This field is four bytes long as required for relative track addresses in hex format. In this program, I reserved those four bytes as a fullword, so the field will be aligned on a fullword boundary. This allows me to manipulate the track address in a register and then store it in the TRKADDR field before issuing a WRITE command.

Now look at the program logic. After the file has been opened, a small loop is initialized and executed thus clearing the three tracks of the file so the new records can be written:

```
            LA    4,3
            LA    5,0
CLEARTRK    ST    5,TRKADDR
            WRITE DAFILE,RZERO
            WAITF DAFILE
            LA    5,256(5)
            BCT   4,CLEARTRK
```

First, register 4 is loaded with a binary value of three so it can be used as a counter for the loop. Then, register 5 is set to zero and stored in the track-address field to set the

```
         START 0
BEGIN    BALR  3,0
         USING *,3
         OPEN  CRD,DAFILE
         LA    4,3
         LA    5,0
CLEARTRK ST    5,TRKADDR
         WRITE DAFILE,RZERO
         WAITF DAFILE
         LA    5,256(5)
         BCT   4,CLEARTRK
         LA    4,143
         LA    5,0
         ST    5,TRKADDR
NXTCRD   GET   CRD
         MVC   DKEY,CDEPT
         MVC   DACCT,CACCT
WRITE    WRITE DAFILE,AFTER
         WAITF DAFILE
         CLC   DAERRS,=X'0000'
         BE    COUNT
         CLC   DAERRS,=X'0800'
         BE    NXTTRK
         DUMP
NXTTRK   LA    5,256(5)
         ST    5,TRKADDR
         B     WRITE
COUNT    BCT   4,NXTCRD
CRDEOF   WRITE DAFILE,AFTER,EOF
         CLOSE CRD,DAFILE
         EOJ
DAFILE   DTFDA BLKSIZE=16,DEVICE=2314,TYPEFLE=OUTPUT,RECFORM=FIXUNB,    X
               KEYLEN=4,IOAREA1=DAOUT,ERRBYTE=DAERRS,SEEKADR=TRKADDR,   X
               DSKXTNT=1,RELTYPE=HEX,AFTER=YES
CRD      DTFCD DEVADDR=SYSIPT,IOAREA1=CARDIN,EOFADDR=CRDEOF
CARDIN   DS    0CL80
CDEPT    DS    CL4
         DS    CL6
CACCT    DS    CL4
         DS    CL66
DAOUT    DS    0CL16
DCOUNT   DS    CL8
DKEY     DS    CL4
DACCT    DS    CL4
TRKADDR  DS    F
DAERRS   DS    CL2
         END   BEGIN
```

FIGURE 15-23 Direct File Creation Program

relative track to zero. When the RZERO form of the WRITE is executed the first time, it clears the first track. The WAITF macro then makes sure that the WRITE macro execution has been completed before processing continues.

When the WRITE RZERO macro has been completed for the first track, the LA instruction is used to increase the contents of register 5 to the next relative-track address. Remember that the hex format of the relative-track address is TT TT TT RR. Thus, the value 256 in the LA instruction changes the contents of register 5 from X'00000000' to X'00000100'. The branch-on-count (BCT) instruction then causes the program to repeat the loop twice more thus clearing tracks 2 and 3. After the third track has been cleared, register 5 is reset to zero and the program continues.

The second part of the program reads the cards containing the department numbers and account numbers. Register 4 is used again for loop control so that a maximum of 143 records will be written on the file. The value 143 is used because the three tracks can hold 144 records (3 times 48); but I want to reserve the last record position for an end-of-file record. In this main routine, register 5 is used for manipulating the relative track address.

After a card has been read and the record assembled, the AFTER form of the WRITE is issued with the relative-track address set to zero. This causes the first record to be written on the first track of the file. While the WRITE is being executed, the WAITF halts processing. When it continues, the program checks to see if the error bytes are all zero. If so, no errors have occurred and the next record can be written on the file by repeating the loop.

If the error bytes aren't zero, at least one of the bits must be on. One particular error that I'm expecting will occur when the WRITE for the 49th record is issued. Since only 48 records can be stored per track, the "no room found" bit will be turned on at this time. When this happens, I must bump the relative-track address to the address of the next track and issue the WRITE for that record again. The same thing should happen when the 97th record is to be written. If any other bits are on, an unexpected I/O error has occurred, in which case a DUMP macro will cause a storage dump and program cancellation.

When all the input cards have been read (or when 143 records have been written), I issue an AFTER EOF form of the WRITE macro. This causes an end-of-file record to be written. The Department/Account file is now created.

Although this program uses WRITE RZERO to clear the disk for file creation, a disk can also be cleared by using the clear-disk utility I mentioned earlier. When this utility is used, however, the tracks are not only cleared, they are also formatted. If, for example, the clear-disk utility were used for this file, the tracks would be given 48 count areas, key areas, and data areas corresponding to the design of the file. Furthermore, the count areas would contain the disk address including cylinder number, head number, and record number. In such a case, the WRITE AFTER macro could not be used to store the actual data on the disk. Instead, the program would have to supply the record number for each record to be written on the disk and use the WRITE ID form of the macro. It is important to understand this difference in methods of clearing disk areas for direct files because each method has a significant effect on the file-creation logic.

Using the Department / Account File

The second program illustrated is actually a subprogram. It can be used by a main program written in any language to look up the general ledger account number for a manufacturing department. This program is listed in figure 15-24.

Before getting into the main part of the program, let me discuss the DTFDA that describes the Department/Account file. This, by the way, is a good opportunity to see the same file described in two different DTFs. This time the file will be processed by using only the KEY form of the READ macro. As a result, the I/O area and the BLKSIZE operand need to account for only the data portion of the record as indicated by the table in figure 15-19. Since I omitted the TYPEFLE operand, the default value, INPUT, is assumed. The required SEEKADR operand names the disk-address field, TRACK, and since I have no need for the returned disk address, I have

```
DEPTACCT START 0
         SAVE  (14,12)
         BALR  3,0
         USING *,3
         LM    4,6,0(1)
         CLI   OPENSW,X'FF'
         BE    SEARCH
         OPEN  DEPTFLE
         MVI   OPENSW,X'FF'
SEARCH   MVC   DEPTKEY,0(4)
         READ  DEPTFLE,KEY
         WAITF DEPTFLE
         CLC   ERRS,=X'0000'
         BNE   CHKERRS1
         MVC   0(4,5),DACCT
         MVI   0(6),C'0'
         B     RETURN
CHKERRS1 TM    ERRS,X'FF'
         BZ    CHKERRS2
DISASTER MVC   0(6),C'9'
         PDUMP DEPTFLE,OPENSW
         B     RETURN
CHKERRS2 TM    ERRS+1,X'D7'
         BZ    NORECFND
         B     DISASTER
NORECFND MVC   0(4,5),=C'0000'
         MVI   0(6),C'1'
RETURN   RETURN  (14,12)
DEPTFLE  DTFDA BLKSIZE=4,DEVICE=2314,RECFORM=FIXUNB,KEYLEN=4,          X
               IOAREA1=DARCD,ERRBYTE=ERRS,SEEKADR=TRACK,KEYARG=DEPTKEY,X
               DSKXTNT=1,RELTYPE=HEX,READKEY=YES,SRCHM=YES,            X
               TYPEFLE=INPUT
DEPTKEY  DS    CL4
ERRS     DS    CL2
TRACK    DC    X'00000000'
OPENSW   DC    X'00'
DARCD    DS    0CL4
DACCT    DS    CL4
         END
```

FIGURE 15-24 Direct File Processing Program

omitted the IDLOC operand. The KEYARG operand gives the name of the field in which I will place the key of the record to be read (DEPTKEY) before reading the records by key. I again elected to use the hex format of relative-track addressing so the RELTYPE operand specifies HEX. Of the operands that indicate the type of I/O macros to be issued, I coded READKEY=YES and SRCHM=YES to indicate a search of all three tracks in the file, by key, with a single READ macro.

Notice in the data definitions following the DTFDA that the attributes of the fields correspond to the DTF entries. The length of the KEYARG field, DEPTKEY, is four bytes; the ERRBYTE field, ERRS, is two bytes; and the disk-address field, TRACK, is four bytes. For this program, I defined the relative-track address field as all hex zeros so it will address relative track zero. I will leave this field set to zero throughout the program, so each time a READ is issued for a key a search for that key will begin on the first track of the file. Then, since SRCHM=YES is specified, the two succeeding tracks will also be searched until the key is found or the end of the

cylinder has been reached. (To prevent unnecessary searching, I will use appropriate job-control cards to allocate this file to the last three tracks of a cylinder.)

Now look at the instructions to see how the subprogram works. The main program passes three data fields to the subprogram before transferring control to it. The first field is department number for which the main program needs the assigned account number. The second field is the account-number field in which the subprogram should place the account number. The third field is a one-byte field in which the subprogram can place a code that tells the main program how the lookup operation turned out. For this program, I chose these codes:

1 If the lookup is successful, the code is C'0' and the main program can expect to find the proper account number in the account number field.
2 If no record can be found for the requested department number, the code C'1' is passed back to the main program.
3 If some kind of major error occurs in reading the Department/Account file, C'9' is passed back to the main program and the main program should close its files and halt.

If you have studied chapter 9, Subroutines and Subprograms, you already understand how the link between a main program and a subprogram works. If you haven't, all you need to know is that register 1 points to a list of the addresses of the passed fields when the main program passes control to the subprogram. In this case, the first fullword at the storage area pointed to by register 1 will contain the address of the department number field; the second fullword will contain the address of the field in which account number should be placed; and the third fullword will contain the address of the error-code field. The LM instruction, the fifth instruction of the subprogram, loads these addresses into registers 4, 5, and 6 so the fields that are actually in the main program can be referred to by the subprogram.

The first routine in the subprogram opens the Department/Account file the first time the subprogram is entered, but it branches around the OPEN statement on subsequent uses of the subprogram:

```
CLI   OPENSW,X'FF'
BE    SEARCH
OPEN  DEPTFLE
MVI   OPENSW,X'FF'
```

Since OPENSW contains hex 00 the first time the routine is executed, the file will be opened. Thereafter, OPENSW will contain hex FF and the OPEN statement will be skipped.

To perform the lookup operation, the department number from the main program (its address has been loaded into register 4) is moved to the KEYARG field, DEPTKEY:

```
MVC   DEPTKEY,0(4)
```

Then, the KEY form of the READ is issued and a WAITF macro follows. This causes the I/O module to search the disk file starting with the first track. When the I/O operation has been completed, the program checks the ERRBYTE field to find out the results.

Three types of conditions can be the result of the READ operation. In one possible result, if the key is found and the record successfully read, the ERRBYTE field will have all bits off. If this is the case, I move the account number from the file I/O area to the main program field whose address is now in register 5. Also, I move C'0' to the main program error-code field whose address is in register 6. Then, I return control to the main program by issuing the RETURN macro.

The second possible result is that all three tracks of the file are searched, but the record isn't found. In this case, two bits should be on in the ERRBYTE field: (1) the bit indicating no-record-found and (2) the bit for end-of-cylinder. The first test-under-mask instruction checks all the bits in the first byte of the ERRBYTE field. If any of these are on, some sort of I/O error has occurred. The second TM instruction tests all the bits of the second error byte except the ones for no-record-found and end-of-cylinder. (Since hex D7 is binary 1101 0111, bits 2 and 4 in this second byte aren't checked.) If any bits other than these two are on, an I/O error has

occurred and the program reaches the instruction labeled DISASTER. If only these two bits are on, it means that the file was searched without error, but the key wasn't found. Then, the subprogram moves zeros to the main program's account-number field, and C'1' is moved to the main program error-code byte. Thus, the main program can determine that the requested department number wasn't in the file.

If an I/O error has occurred (the third possible resulting condition) the program moves C'9' to the main program error byte to indicate a major error. Before returning to the main program, however, I cause a partial storage dump by using the PDUMP macro, which is described in chapter 11. This macro will dump storage between the DTFDA macro and the field named OPENSW. This partial dump will help me figure out why the error took place.

You might notice that this subprogram does not provide for closing the Department/Account file. This is all right since a closing routine for an input direct file doesn't affect the labels or file in any way.

Conclusion

These two programs represent one of the simplest and most useful ways to use direct files—using a direct file for storing and searching a large table. From this example, you can imagine how much more complex it might be to manipulate and maintain a direct file for broader uses. Although you probably won't deal with direct files often (if ever), I hope this exposure will at least give you a deeper understanding of disk operations.

Terminology

relative-track address
relative-track addressing
clear-disk utility
search multiple tracks

Objective

Given a problem involving direct disk files, code a BAL solution.

Problem

Assume that the three tracks for the Account/Department file were cleared by the clear-disk utility before executing the file-creation program. Since this utility places 48 records with blank key and data areas on the disk, the file-creation program will not be the same as the one in figure 15-23. Write this file-creation program.

Solution

Figure 15-25 is an acceptable solution. In this program, the I/O module returns the address of the next record in the IDLOC field after a READ ID or WRITE ID has been executed. If the record read or written is the last one on a cylinder, the address returned will be that of the first record of the next cylinder and the end-of-cylinder error bit is turned on. As a result, my error-check routines allow the end-of-cylinder bit to be on, but any other error bit on indicates an error. After writing a new record, I move the address returned in the IDLOC field to the SEEKADR field so it contains the address of the next record to be written.

```
CACREATE START 0
BEGIN    BALR  3,0
         USING *,3
         OPEN  CRD,DEPTACT
READCARD GET   CRD
         MVC   RCDKEY,CDEPT          MOVE DEPT NBR TO KEY
         MVC   RCDDATA,CACCT         MOVE ACCT NBR TO DATA
         WRITE DEPTACT,ID
         WAITF DEPTACT
         CLC   ERRS,=X'0000'         IF NO =RR BITS ON,
         BE    SETUPNXT              GO SET UP FOR NXT RCD
         TM    ERRS,X'FF'            ELSE, TEST 1ST ERR BYTE BITS
         BZ    TEST22                IF NONE ON, GO TEST 2ND BYTE
         DUMP
TEST22   TM    ERRS,X'DF'            CHECK ALL BITS BUT END OF CYL
         BZ    SETUPNXT              GO SET UP FOR NXT RCD
         DUMP                        ELSE, DUMP
SETUPNXT MVC   RCDADDR,NXTADDR       MOVE NXT RCD ADDR FROM IDLOC
         B     READCARD              GO READ NXT CRD
CRDEOF   CLOSE CRD,DEPTACT
         EOJ
DEPTACT  DTFDA BLKSIZE=8,DEVICE=2314,TYPEFLE=OUTPUT,RECFORM=FIXUNB,     X
               KEYLEN=4,IOAREA1=RCDIO,ERRBYTE=ERRS,SEEKADR=RCDADDR,     X
               IDLOC=NXTADDR,DSKXTNT=1,RELTYPE=HEX,WRITEID=YES
CRD      DTFCD DEVADDR=SYSIPT,IOAREA1=CARDIN,EOFADDR=CRDEOF
CARDIN   DS    0CL80
CDEPT    DS    CL4
         DS    CL6
CACCT    DS    CL4
         DS    CL66
RCDIO    DS    0CL8
RCDKEY   DS    CL4
RCDDATA  DS    CL4
ERRS     DS    CL2
RCDADDR  DC    F'1'
NXTADDR  DS    F
         END   BEGIN
```

FIGURE 15-25 Direct File Creation Using Preformatted Tracks

Part 6

Part 6

The Operating System

This part consists of just one chapter. It shows how the Disk Operating System relates to assembler-language programming. It can be studied any time after part 2 has been completed.

16

The Disk Operating System and its Job-Control Language

Being able to write a program is not enough. A good programmer must know how to coordinate his program with the computer's operating system so the program will execute properly. To do this, he must have a basic understanding of the facilities of the operating system and he must know how to code job-control statements. Topic 1 describes the facilities of the Disk Operating System and indicates how they work together. Topic 2 builds on the brief introduction to job-control language that was presented in chapter 1. It describes how to code the job-control language (JCL) statements that an accomplished programmer uses. The material covered includes the JCL for tape and disk file processing, for expanded control of link-editing (for example, combining main and subprograms), and for some of the other DOS facilities.

TOPIC ONE DOS Facilities The Disk Operating System is simply a collection of programs stored in library files on a disk pack. These programs are designed to improve operating efficiency and programming efficiency. A description of some of the basic DOS programs—those used on all DOS systems—follows.

Supervisor The most used program of the Disk Operating System is the *supervisor* program. It is loaded into storage at the start of a day's operations, stays in storage throughout the day, and controls the overall operation of the computer system. Two of the supervisor's most important functions are (1) loading all other programs and (2) starting all I/O operations.

Job-Control Program Job-to-job transition begins when the supervisor loads the *job-control program* into storage. The job-control program both processes all the job-control statements that are used to request the execution of a program and checks the availability of the required input and output devices. If a job-control statement is invalid or if a specified device isn't available, the job-control program will print an error message on the console typewriter so the operator can correct the job deck or make the device available. When all the job-control statements have been processed with no errors found, control returns to the supervisor so the requested program can be loaded and executed.

Linkage Editor When an object program is assembled or compiled, it is in relocatable form. That means that it is usually relocated—assigned to storage locations other than those given during assembly or compilation—before being loaded and executed. This relocation is done by the *linkage editor.*

When a main program is combined with one or more subprograms, this, too, is done by the linkage editor. In this case, all of the separate segments of object code (called *object modules*) must be relocated before they can be combined. Furthermore, the linkage editor must resolve all references between the object modules.

Language Translators *Language translators* are the assemblers and compilers that are supplied with an operating system. With DOS, an assembler plus COBOL, FORTRAN, PL/I, and RPG compilers are provided. The intent of the translators is to reduce the programming time required to prepare a working object program. To fulfill this purpose, all of the translators print diagnostic listings to help in correcting clerical errors in the source deck and often they provide debugging statements that aid in testing the object program.

Sort/Merge Programs In any computer installation, much of the processing requires that records be in certain sequences. Therefore, two or more tape or direct-access files often have to be merged into one file, or one tape or direct-access file has to be sorted into a sequence. Although this sorting and merging may take as much as 40 percent of the total running time of a computer system, the *sort/merge programs* themselves are very much alike. They differ primarily in the number of files to be merged, the length of the records in the files, the blocking factors used, the length and location of the field on which the file is to be sequenced, and the number of I/O devices to be used.

DOS provides one or more sort/merge programs, depending on the I/O devices of the system. These are generalized programs that can be used for many different jobs. To use them, the user supplies coded specifications and the sort/merge program adjusts accordingly. This eliminates the need for sort programs written by the computer user.

Utility Programs Many of the programs of a typical computer installation are relatively simple programs that convert data from one I/O form to another—for example, printing the contents of a file of tape records, or converting a disk file to a tape file. So that a computer user need not write such programs, DOS provides *utility programs.* When the user specifies in coded form the characteristics of the files involved, the utility programs adjust to these

characteristics and do the desired processing. Thus routine card-to-printer, card-to-tape, card-to-disk, tape-to-printer, tape-to-tape, and many other such programs are supplied with DOS.

Library-Maintenance Programs To reduce duplication of programming effort within a company, DOS provides for segments of both source and object code to be stored in libraries on the system-residence device. This means that, if a programmer writes a payroll routine for finding a job rate when the job class is known, it can be stored either as source code or as object code in one of the libraries of DOS. Any other program that involves this routine can then retrieve the routine from the appropriate library. If the routine is stored as source code, the statements can be placed in the program and assembled along with the rest of the source code. If the routine is stored as object code, the object code of the main program and the object code of the payroll routine can be combined before execution by the linkage editor.

To add new routines to and delete old routines from a library, *library-maintenance programs* are required. These, are supplied as part of the Disk Operating System. Library-maintenance programs also provide routines for printing the names of the segments stored within a library or for printing the specified segments themselves.

I/O Modules Throughout this book there are references to the IBM-supplied *I/O modules*. These are object modules (subprograms) that help programs written in any language perform I/O operations. These modules are part of the Disk Operating System.

Macros In various chapters of this book there are references to standard *macros*—for instance, the DTFCD and GET macros. These are segments of assembler-language source code that get assembled into appropriate object code. As a result, they are critical to the assembly and execution of your source programs.

THE DOS LIBRARIES

There are three DOS libraries that are stored on the system-residence device: the *core-image library*, the *relocatable library*, and the *source-statement library*. To find programs or segments of source or object code in these libraries, one *directory* is kept for each library. These directories keep the current names and locations of the contents of each library.

Core-Image Library Before object code can be executed, it must be stored in the core-image library. This library has two sections: a permanent section in which the commonly used programs are kept, and a temporary section in which programs are stored for one-time use. If a program is stored in the permanent section, it is called a *phase* and is given a *phase name* that is used to locate the program in the file. For example, the phase name for the assembler is ASSEMBLY; the phase name for the linkage editor is LNKEDT. When a program is stored in the temporary section of the core-image library, it isn't given a phase name because it isn't catalogued in the library's directory. Since a program in the core-image library can be executed without link-editing, most production programs and many of the programs of the operating system—such as the job-control program, the linkage editor, and the assembler—are stored in this library.

Relocatable Library The programs in the relocatable library must be link-edited before they can be executed. As a result, subprograms are normally stored in this library. Prior to execution, the subprogram object *modules*, referred to by *module names*, are combined with a main module by the linkage editor. The primary DOS programs stored in this library are the I/O modules.

Source-Statement Library The segments of source code stored in the source-statement library are called *books* and the names used to refer to the books are *book names*. *Sublibraries* within this library are used to store each type of source-language book. For example, there is one sublibrary

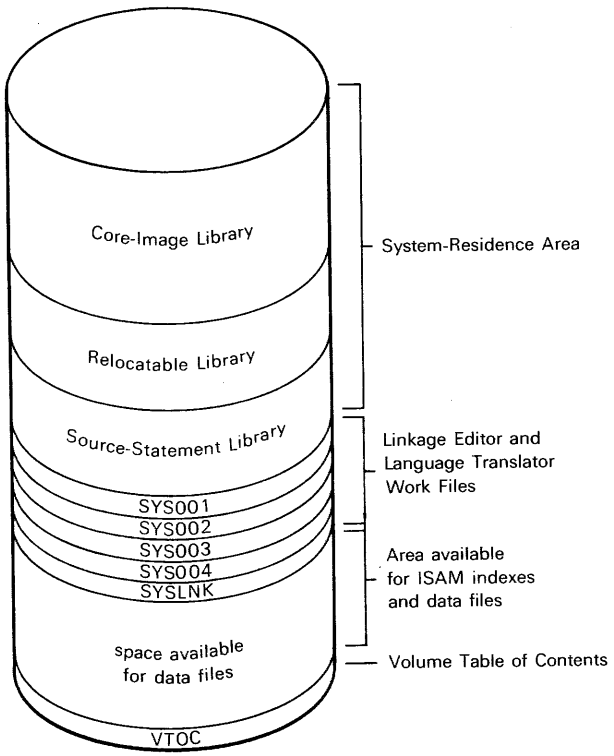


FIGURE 16-1 DOS System Pack Allocation

for COBOL, another for assembler. The standard assembler macros are stored in the assembler-language sublibrary.

SYSTEM PACK ALLOCATION

The creation of a system-residence pack for an installation is called *system generation*. This is done by selecting DOS programs from the complete DOS system pack supplied by IBM and writing them on the installation system pack. Since any one installation is not likely to use all of the DOS facilities, only those that will be used are stored on the system pack.

During system generation, each of the three libraries of DOS is allocated some area on the system pack. Figure 16-1 illustrates a typical system pack for a small DOS installation. The libraries commonly occupy one-half or more of a 2314 pack with the core-image library taking up more than half of this library space.

During system generation, the work file areas required by the language translators and linkage editor are also allocated space on the system pack. These areas are also illustrated in figure 16-1. During a COBOL compilation, for example, the areas labeled SYS001, SYS002, SYS003, and SYS004 are used as temporary storage areas. Similarly, the area labeled SYSLNK is used as a temporary storage area during link-editing. Any space left on the pack can be used for data files.

DOS IN OPERATION

To show how the DOS programs work together, let me trace through the steps involved in executing a typical job—assembling one of your source programs, link-editing it, and testing it. I'll assume that the program to be tested reads cards and a tape-input file and writes out a disk file and a printed report.

1 Sometime earlier (this morning, last night, several days ago), the supervisor program was loaded into the low-address portion of storage when the operator performed an *IPL* (Initial Program Load). This is done by pushing a particular button on the computer console. The DOS supervisor is then resident in storage and ready to begin processing jobs.

2 When the supervisor has finished its IPL processing (it usually takes one or two seconds), it loads the job-control program from the core-image library into the problem program storage area.

3 The job-control program reads the JCL statements for the assembly, which is the first job step of the job. As it does this, it notes that you have requested that the object form of your program be saved for input to the linkage editor (// OPTION LINK). When it finds the // EXEC ASSEMBLY statement, the job-control program branches

back to the supervisor.

4 The supervisor loads the assembler from the core-image library into storage, overlaying the job-control program. Then it branches to the first instruction of the assembler.

5 The assembler processes your source program, looking up macro definitions in the source-statement library, and writing and reading the program in various stages of translation to and from the work files on the system pack. When the assembler has finished, it has placed the object code of your program, plus some notes that tell which I/O modules are needed to support your program, on the linkage-editor disk area (SYSLNK). The assembler then branches back to the supervisor.

6 The supervisor reloads the job-control program from the core-image library.

7 The job-control program processes the JCL statements that request execution of the linkage editor and then it branches back to the supervisor.

8 The supervisor loads the linkage editor and gives it control of the computer.

9 The linkage editor reads the input prepared by the assembler program. It accepts your object program and sees that four I/O modules—one each for the card file, the printer file, the tape file, and the disk file—must be link-edited with your program. The linkage editor retrieves the I/O modules from the relocatable library and combines them with your object program to form a phase, which it stores in the temporary area of the core-image library. It then branches to the supervisor.

10 The supervisor again loads the job-control program.

11 Job control processes the JCL statements that request execution of the program that has just been link-edited. If any particular input or output units are requested, job control checks to make sure they are available. In this case, it checks to make sure the card reader and the printer are available for your program, and it assigns one of the tape drives and one of the disk drives to your program. When it is done, job control branches to the supervisor.

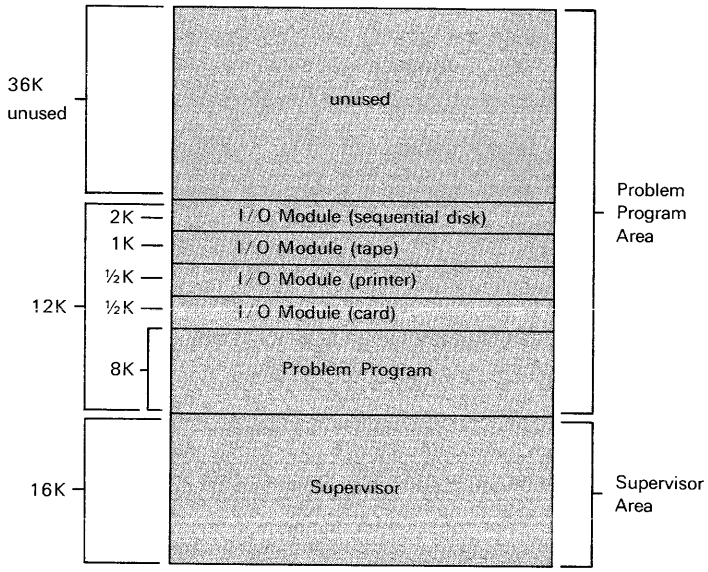


FIGURE 16-2 Storage Allocation of 12K Program in 64K CPU

12 The supervisor loads the phase made up of your program and the four I/O modules, then branches to the first instruction of your program. The contents of storage now look something like figure 16-2. The problem program area can of course be used for any system or user program other than the supervisor.

13 Your program instructions execute until you OPEN the input and output files. Since the OPEN macro has generated instructions that cause a branch to the supervisor, control passes to the supervisor.

14 The supervisor loads the OPEN routine into a small area reserved within the supervisor area. This area, called a *transient area*, is used for infrequently used routines (*transient routines*) such as the OPEN and CLOSE routines. The OPEN routine checks the status of the reader and printer, reads and processes the tape file label, performs the label checking necessary for the

output disk file, and then branches back to your program.

15 Again your program is executed until the instructions generated by a GET, PUT, READ, or WRITE are encountered. Any of these cause a branch to the proper I/O module.

16 The I/O module constructs and passes to the supervisor an I/O request based on the operands coded in your DTF.

17 The supervisor starts the requested I/O operation, then branches back to the I/O module.

18 The I/O module now waits until the I/O operation is completed, checks for errors after completion, performs any blocking or deblocking that may be necessary, and branches back to your program.

19 Your program continues to execute. Each time another translated GET, PUT, or other I/O macro is reached, steps 15 through 18 are repeated. Near the end of your program, a CLOSE of the files is requested. This causes a branch to the supervisor and the CLOSE routine is executed in the transient area. Finally, your program executes the instructions that were generated by the EOJ macro, and which cause a branch back to the supervisor.

20 The supervisor performs some end-of-job housekeeping, mostly in the nature of releasing I/O units that have been assigned to your program; then it again loads the job-control program to start the execution of the next job.

I hope this description gives you a better idea of the interrelationships between the programs of an operating system. As you can imagine, many man-years of programming time, and close coordination of the programming effort are involved in the development of an operating system like DOS. Fortunately, from the user's standpoint, much that happens during the execution of a job doesn't have anything to do with him. At the operational level, running a job consists primarily of preparing I/O devices—putting forms in printers, mounting tapes on tape drives, and so on—and putting the appropriate job-control cards in the card reader. Similarly, from a programmer's point of view, preparing a program consists primarily of creating a source deck, the related test data, and the appropriate job-control cards. Detailed knowledge of the relationships between the programs of the operating system is only important when the system fails—an infrequent occurence.

ADVANCED DOS FACILITIES

All DOS systems use the facilities described so far. However, DOS also provides some advanced facilities. These facilities, which are being implemented on more and more systems each year, require greater expertise on the part of the user. Once implemented, however, they can significantly improve the productivity of a system.

Multiprogramming Multiprogramming means the simultaneous execution of more than one program. Actually, this is misleading because what really happens is that multiple programs are present in separate parts of storage simultaneously, but only one program actually executes at any given time. The others wait for input or output operations to complete, or just sit and wait to be given control by the supervisor.

Multiprogramming is valuable because it can increase the overall productivity of a computer system. As you learned in chapter 1, internal operations like data movement and arithmetic operations can be executed thousands of times faster than input or output operations. Since it is the nature of business programs to have little internal processing between I/O operations, the CPU is idle a great deal of time, with the program waiting for an I/O operation to be completed. In an effort to make productive use of this idle time, multiprogramming systems allow additional programs to be present in storage so their instructions can be executed while the first program waits for its I/O operation to be completed.

In standard DOS, up to three programs may be

multiprogrammed at one time. Under supervisor control, the storage area above the supervisor is divided into three *partitions* called *background* (BG), *foreground 1* (F1), and *foreground 2* (F2). Although they can be altered by special operator commands, the sizes of these partitions are assigned standard values at the time the system is generated. Figure 16-3 shows the storage allocations for the three partitions in a typical DOS multiprogramming system. The supervisor then has the added responsibility of deciding which partition gets to process after each time the supervisor has started an I/O operation. To solve this problem, a priority system is used with the supervisor always giving control first to the F1 program. If the F1 program is waiting for an I/O operation to finish, it then gives control to the F2 program. If F2 is waiting also, control is given to the BG program.

Spooling Even with three programs running together, there is still usually more CPU time available than can be used because of the difference between internal processing speed and the speed of the I/O units. In a further effort to make better use of the available CPU time, spooling programs serve as an interface between the very slow I/O devices—the card readers, punches, and printers—and the processing programs. When the processing program attempts to print a line on the printer, for example, the spooling program causes the line to be written instead as a record on a tape or disk file. Since these devices are much faster than the printer, they allow the problem program to resume processing much sooner. Later on, when little is being done by the computer, the spooling program prints out the print lines that were temporarily stored on tape or disk. Because spooling is used on most large systems, line counting is used for forms-control on these systems.

The same spooling procedure is used in reverse for card-input files. The spooling program first places the cards on tape or disk. Then, when the processing program tries to read a card from the card reader, it is actually read from the tape or disk and the program waits a much shorter time for I/O completion. Although there are several spooling programs used in DOS installations, the most common one is IBM's POWER program. It is usually run in the F1 partition as shown in figure 16-3.

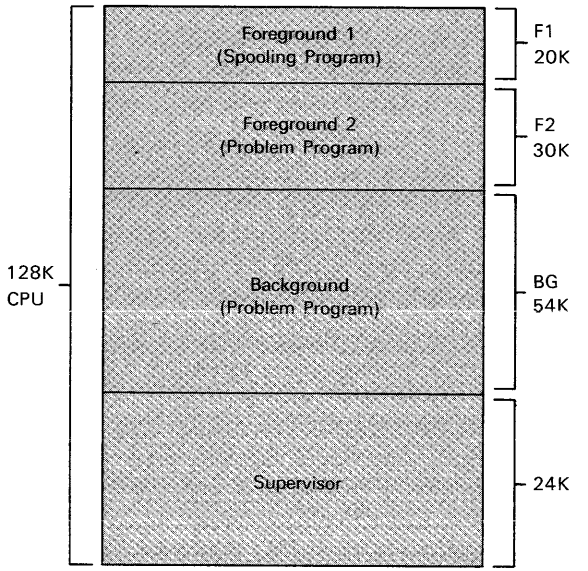


FIGURE 16-3 Storage Allocation in a Multiprogramming DOS System

Virtual Storage An advanced level of DOS is available with the System/370. It is called *DOS/VS*, where VS stands for Virtual Storage. It allows simulation of a very large CPU on a much smaller one by using disk storage as an extension of internal storage. In a typical DOS/VS system, for example, a 512K CPU might be simulated on a 196K actual CPU. During execution of a program, small sections of the program (*pages*) are moved into storage from the disk, one section overlaying a previous section.

Since the CPU speed of a System/370 is even faster than the System/360, the difference between the speed of internal operations and that of I/O operations is even greater. This offers more multiprogramming opportunity so DOS/VS allows five partitions instead of the three with standard DOS. Because of page swapping, the actual internal operations of DOS/VS are more complex than (though similar to) those

of standard DOS. Both operating systems work the same from a programmer's point of view, however.

Terminology

supervisor
job-control program
linkage editor
object module
language translator
sort/merge program
utility program
library-maintenance program
I/O module
macro
core-image library
relocatable library
source-statement library
directory
phase
phase name
module
module name
book
book name
sublibrary
system generation
IPL
transient area
transient routine
multiprogramming
partition
background
foreground 1
foreground 2
spooling
virtual storage
DOS/VS
pages

Objectives

1 Describe the purpose of each of the following DOS programs:
   supervisor
   job-control program
   linkage editor
   language translators
   sort/merge programs
   utility programs
   library-maintenance programs
   I/O modules
   macros
2 Name the three DOS libraries and describe the typical contents of each library.

TOPIC TWO DOS Job-Control Language

Chapter 1 presented the basic JCL statements for assembling and testing a program—the JOB statement, the EXEC statement, the end-of-data statement (/*), and the end-of-job statement (/&). It also introduced the OPTION statement.

This topic expands on that base of knowledge. It shows how to assign files to I/O devices, how to run programs with tape and disk files, how to combine two or more object modules, how to use the library-maintenance programs, and even how to run simple utility programs. Because each of these functions requires different forms of JCL cards, this topic is broken down by function. Depending on what you have read up to now, only selected functions may be of interest to you.

BASIC JCL EXPANDED

OPTION Statement When a DOS system is generated, certain system parameters are set. The purpose of the OPTION statement is to reset one or more of these parameters, thus affecting assemblies, link-edit runs, test runs, and compilations of languages like COBOL and FORTRAN. The OPTION statement that you use in your assemble, link, and test runs (shown in figure 16-4) illustrates two of the possible OPTION operands: the LINK option indicating that the resulting object program will be link-edited and the DUMP option meaning that program failure during the test run should cause a storage dump.

A partial list of the *valid* OPTION operands is given in figure 16-5. A typical DOS system might have the following *standard options* set at system generation time:

```
NOLOG,NODUMP,NOLINK,NODECK,LIST,LISTX,SYM,XREF,ERRS
```

The standard options are in effect for all jobs unless they are overridden by an OPTION statement. If they are overridden, the altered settings from the OPTION statement are in effect until the end of the job or until they are reset by another OPTION statement. At the end of the job, all the options revert to their standard settings. Since an OPTION card is only required when you want to change a standard option,

```
// JOB TEST
// OPTION LINK,DUMP
// EXEC ASSEMBLY
   (source program)
/*
// EXEC LNKEDT
// EXEC
   (test data)
/*
/&
```

FIGURE 16-4 Basic JCL Setup

you must know the standard options of your own system to be able to decide when an OPTION statement is necessary.

For the standard options listed above, you need a card like this to get an object deck:

```
// OPTION DECK
```

If you want the JCL statements to print on the printer, and you also want to link-edit the resulting object program, you need a card like this:

```
// OPTION LOG,LINK,DUMP
```

Here, the DUMP operand is used to request a storage dump if the test run fails. The operands in the OPTION card can be given in any sequence, but they must be separated by commas with no intervening blanks. The CATAL operand is explained later in the library-maintenance section of this topic.

ASSGN Statement The ASSGN statement—another basic JCL statement—is used to assign the files defined in your program to physical I/O devices. To help you understand how I/O device assignments work, consider the typical system shown in figure 16-6. The schematic represents a Model 40 System/360 with 128K storage and a 2540 reader/punch, a 1403 printer, four 2401 tape drives, and a five-spindle 2314 facility.

Operand	Meaning
LOG **NOLOG**	Print (log) all control statements on SYSLST.
DUMP **NODUMP**	Print a storage dump if an abnormal program end occurs.
LINK **NOLINK**	Save the object program for immediate input to linkage editor program.
DECK **NODECK**	Punch the object program in cards.
LIST **NOLIST**	Print a source-program listing.
LISTX **NOLISTX**	Print a procedure-division map during a COBOL compilation. No effect for assemblies.
SYM **NOSYM**	Punch a symbol table in the object deck for assemblies or print a data-division map for COBOL compilations. Since neither option is of much value to the BAL programmer, this option is usually NOSYM.
XREF **NOXREF**	Print a symbol cross-reference list.
ERRS **NOERRS**	Print a diagnostic listing.
CATAL	Permanently catalog any programs that are link-edited into the core-image library.

FIGURE 16-5 OPTION Statement Operands

Each of the physical I/O units on a System/360-370 is assigned a *device address* that is expressed as three hex digits. The 2540 in figure 16-6, for example, is assigned device addresses hex 00C for the reader and hex 00D for the punch. (The 2540 is considered to be two devices, even though both are housed in one physical unit.) Each tape drive is assigned an individual address (hex 180 to 183) as is each individual disk spindle on the 2314 (hex 130 to 134).

The programmer, however, doesn't refer to I/O units directly by their physical addresses. Instead, he refers to *logical units* (also called *symbolic units*). In the DTFCD, DTFPR, and DTFMT macros, for example, the DEVADDR operand is coded with a logical unit such as SYSIPT, SYSLST, or SYS009.

When a system is generated, the physical addresses of the I/O units are assigned to various logical units. These

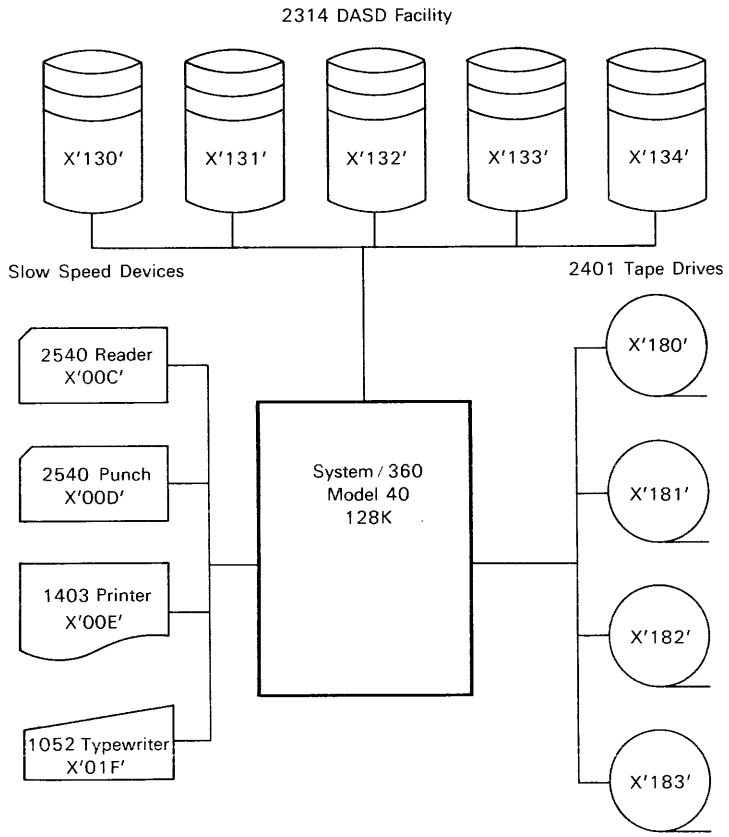


FIGURE 16-6 A Typical System/360 Configuration

assignments are referred to as *standard assignments*. Figure 16-7, for example, shows the standard assignments that might be generated for the System/360 configuration in figure 16-6. The *system logical units* are referred to by the DOS control programs, but they can also be referred to by user programs. The *programmer logical units* are to be used by user programs.

Since the system-residence device is usually the disk spindle with the lowest hex address (hex 130 in figure 16-6), in figure 16-7 SYSLNK and SYSRES are assigned to the system-residence device. Furthermore, SYS001 through SYS004 are assigned to the system-residence device. These logical units represent work areas used by the language translators. Although they could be on another disk spindle, they are usually on the system-residence device.

From SYS005 up to the highest numbered logical unit specified in the supervisor (it can go as high as SYS221), assignments are made that will best serve the programming requirements of the particular installation. An arrangement of these assignments similar to those shown in figure 16-7 is very common. Notice that some of the logical units are left unassigned to accommodate special situations or to provide for later addition of more I/O units.

The ASSGN card is used to temporarily override the standard assignments as in this example:

```
// ASSGN SYS005,X'182'
```

The first operand names the logical unit and the second specifies the physical address of the device to which the logical unit is to be assigned. This assignment is in effect from the time the ASSGN card is read until the end of the job or until another ASSGN card changes the assignment of this logical unit. When the job ends, the logical unit reverts to its standard assignment.

To illustrate the need for the ASSGN card; suppose you want to run a program named REPRO that was originally written for another system on the sample configuration of figure 16-6 running under the standard assignments of figure 16-7. REPRO reproduces card decks, reading input cards from SYS006, which it expects to be a 2540 card reader, and

	Logical Unit	Standard Assignment	Use
System Logical Units	SYSRDR	X'00C'	JCL input
	SYSIPT	X'00C'	Other input
	SYSPCH	X'00D'	Punched output
	SYSLST	X'00E'	Printed system output
	SYSLOG	X'01F'	Operator messages
	SYSLNK	X'130'	Link-editor input area
	SYSRES	X'130'	System residence area
Programmer Logical Units	SYS000	unassigned	
	SYS001–SYS004	X'130'	Work areas for compilers
	SYS005	X'00C'	
	SYS006	X'00D'	
	SYS007	X'00E'	
	SYS008–SYS009	unassigned	
	SYS010–SYS011	X'180'	
	SYS012–SYS013	X'181'	
	SYS014–SYS015	X'182'	
	SYS016–SYS017	X'183'	
	SYS018–SYS019	unassigned	
	SYS020–SYS024	X'131'	
	SYS025–SYS029	X'132'	
	SYS030–SYS034	X'133'	
	SYS035–SYS039	X'134'	
	SYS040–SYS050	unassigned	

FIGURE 16-7 Sample Standard Assignments

punching duplicate cards on SYS008, which it expects to be a 2540 card punch. To run REPRO on the sample system, these two ASSGN cards would be necessary:

```
// ASSGN SYS006,X'00C'
// ASSGN SYS008,X'00D'
```

These ASSGN cards can appear anywhere in the job step as long as they precede the EXEC card for the program that requires the changed assignments.

In most installations, a programmer codes the DEVADDR of his DTF macros with a logical unit whose standard assignment refers to the proper device. For example, a programmer uses SYSLST for the printer because the printer normally has SYSLST as its standard assignment. When the appropriate logical units are used, as given in the standard assignments, no ASSGN cards are needed.

JCL FOR TAPE FILES

To prevent accidental destruction of tape files, most DOS installations use the standard tape-label facility. Because of this, label information must be supplied to the operating

TLBL Statement Format

```
// TLBL filename,
        ['file-ID'],
        [date],
        [volume serial number],
        [volume sequence number],
        [file sequence number],
        [generation number],
        [version number]
```

filename	The 1 to 7 character DTFMT label from program
'file-ID'	From 1 to 17 characters enclosed in single quotes. Will be checked if specified for input. If omitted for output, filename is used.
date	The creation date for an input file or the expiration date for an output file. It may be in the form YY/DDD where YY indicates the year and DDD indicates the number of the day in the year. For an output file, it may also be written as DDDD indicating the number of days the file should be retained. If omitted for output, current date is used for both creation and expiration dates.
volume serial number	The six-character volume-serial number of the tape volume (reel). If coded, it is checked during OPEN. If omitted, mounted tape is assumed to be correct volume.
volume sequence number	One to four characters indicating the number of the volume in a multivolume file. If omitted for output, 0001 is assumed.
file sequence number	From 1 to 4 digits indicating the position of a file on a multifile volume. If omitted, 0001 is assumed.
generation number	From 1 to 4 digits indicating the number of a particular edition of the file. If omitted for output, 0001 is assumed.
version number	One or two digits indicating the number of a particular version of an edition of the file.

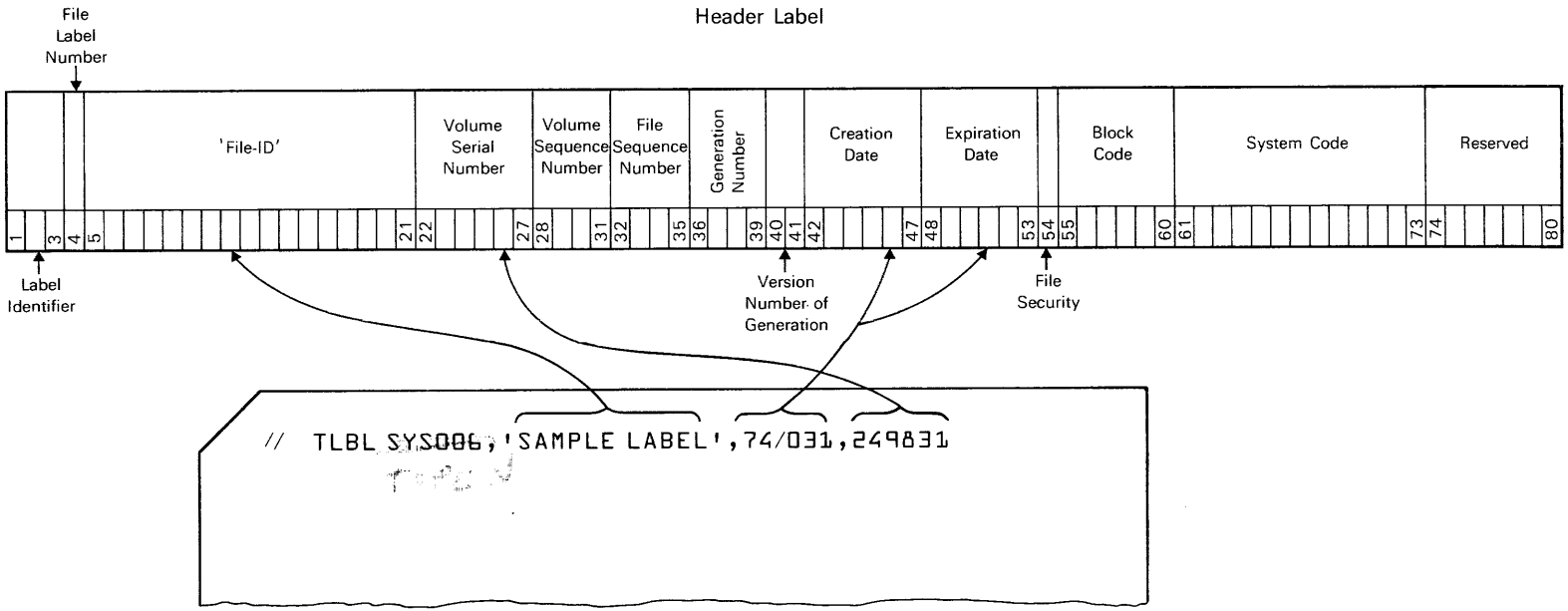


FIGURE 16-8 Standard Tape Label and TLBL Statement Format

system at the time a program involving tape files is executed. For this purpose, a TLBL card is required. To specify that extra storage is required for label information, the LBLTYP card is also required prior to the linkage-editor step. For example, the following JCL cards could be used to assemble, link, and execute the tape-to-printer program in figure 13-2.

```
// JOB TAPE TEST
// OPTION LINK,DUMP
// EXEC ASSEMBLY
   (source deck)
/*
// LBLTYP TAPE
// EXEC LNKEDT
// ASSGN SYS008,X'180'
// TLBL TAPEIN,'SAMPLE LABEL',74/031,24893
// EXEC
/&
```

Here, the LBLTYP card specifies TAPE indicating that the linkage editor must provide an 80-byte tape label area at the start of the user's program. Then, when the tape file is opened, the label information from the TLBL card is compared with the label record on the tape to make sure that the proper tape is mounted. Since no cards are to be processed, no data deck follows the last EXEC card.

Figure 16-8 shows the format of the tape-label record, the format of the TLBL card, and an explanation of each of the TLBL operands. Notice the relationships between the TLBL operands and the fields of the tape label. When the tape file is opened, comparisons are made between the TLBL operands and the tape label fields. For input files, as in the example above, the operands in the TLBL statement are compared to the corresponding fields of the tape label. If they are equal, the file is opened and the processing can begin. If they are not equal, a message is issued to the console typewriter to tell the operator that the wrong tape has been mounted. Since you can selectively code the optional operands of the TLBL card for an input file, only certain fields need be checked. In the example above, only the file identifier, the date, and the serial number are checked.

For output files, the label processing is different. First, the volume serial number, if coded in the TLBL statement, is compared to the tape volume label. (Often, this TLBL operand is omitted for output files to allow any available tape to be used.) Then, if the tape already holds a file, the *expiration date* in its label is compared to the current date stored in the supervisor. If the current date is greater than the expiration date, the file has expired and the tape can be used for the new file. Otherwise, the operator is told by way of the console typewriter that the file hasn't expired and should be saved.

If a tape is accepted for output, the file-identifier operand in the TLBL card becomes the file-identifier field in the label, and the current date becomes the creation date. The expiration date for an output file comes from the TLBL date operand in one of two ways. The date operand either specifies an absolute expiration date in the format YY/DDD, or it specifies a retention period as a number of days, DDDD. This period is added to the current date to calculate an expiration date.

Since multivolume files and multifile volumes are rare, the volume-sequence number and file-sequence number operands are rarely used. Similarly, the generation-number and version-number operands are seldom used. These two fields are intended to allow multiple versions of the same file to be created and identified without changing the TLBL job-control statement every time. A typical situation where this facility might be useful would be if a transaction file were created every day, five days a week, with each of the daily tapes saved for five days. There would then be five versions or generations of the file at all times, and the generation or version numbers could be used to distinguish between them. In actual practice, however, only a few operands are used in the TLBL card so DOS label-checking routines process only a few of the several fields they are capable of processing.

Figure 16-9 illustrates the JCL to assemble, link, and test the tape update program of figure 13-6. This program reads the old tape file, updates it with data from a transaction tape,

```
// JOB TAPE UPDATE TEST
// OPTION LINK,DUMP
// EXEC ASSEMBLY
   (source deck)
/*
// LBLTYP TAPE
// EXEC LNKEDT
// ASSGN SYS008,X'181'
// ASSGN SYS009,X'182'
// ASSGN SYS010,X'183'
// TLBL TRANS,'TRANSACTIONS',74/102,003414
// TLBL OLDMSTR,'CUSTOMERS',,008432
// TLBL NEWMSTR,'CUSTOMERS',30
// TLBL ERRORS,'ERROR TAPE',10
// EXEC
/&
```

FIGURE 16-9 JCL for a Tape Processing Program

writes the updated records onto a new tape, and writes an error tape. Before the EXEC LNKEDT card, the LBLTYP card indicates that the 80-byte label area must be reserved at the start of the user's program. Assuming the standard device assignments of figure 16-7, three ASSGN cards are needed to assign logical units SYS008, SYS009, and SYS010 to drives X'181', X'182,' and X'183'. Since logical unit SYS011 has a standard assignment of X'180', it doesn't require an assign card. The file-ID, the date, and the volume serial number are specified in the TLBL cards for the transaction file, so these fields will be checked; for the input master file, only file ID and the serial number will be checked. In the TLBL cards for the output files, the volume serial numbers are omitted, so any available tapes can be used. The expiration date for the new master file will be 30 days from the current date; for the error tape, it will be 10 days from the current date.

JCL FOR DISK FILES

All disk files in the standard file organizations have standard disk labels. This means that DLBL and EXTENT job-control statements must be included in the JCL for programs that use disk files so these labels can be checked. Figure 16-10 shows the format of the file label stored on the disk and the format of the DLBL and EXTENT statements. It also shows the relationship between the statement operands and the label fields. The file-ID and date operands of the DLBL card are similar to those of TLBL except that the file-ID operand must always be present for disk input files so the proper disk label for the file can be found in the Volume Table of Contents. If a file has sequential organization, the code operand in the DLBL card can be omitted; otherwise, the code operand must indicate the file organization.

The EXTENT statement (or statements) can be considered an extension of the DLBL information and must always be placed directly after the DLBL statement. The first EXTENT operand names the logical unit on which the file is to be found. Although the volume serial number is optional, it is almost always coded so that the OPEN routine can check to see that the proper pack has been mounted. The type operand is coded 1 (or omitted, since the default value is 1) for all *extents* (disk areas) except indexed-sequential index areas or general overflow areas. The extent-sequence-number operand is frequently omitted, allowing the default value of zero to be used. The relative-track operand identifies the starting track of the file as a track number relative to the first track of the disk pack, track zero of cylinder zero. On a 2314, for example, a file starting in the 30th cylinder would have a relative-track value of 600. The last operand of the EXTENT statement specifies the number of tracks allocated to this particular extent. For most sequential and direct files, only one EXTENT statement is used so the number of tracks specified represents the entire file. Except for indexed sequential index-area extents, the number of tracks is usually given in even cylinder multiples.

To illustrate the JCL for a sequential file, suppose a

program is to create a sequential file of accounts-receivable transaction records. The programmer has decided to put the file on disk volume CMP060 and has found that cylinders 60-64 are free to hold it. These are the DLBL and EXTENT statements for the file:

```
// DLBL ARTRANS,'ACCTS RCV TRANS'
// EXTENT SYS023,CMP060,,0,1200,100
```

The DLBL statement gives the filename (ARTRANS) and the file identifier that is to become the label of the file. Since the date operand is omitted, the file will be assigned a retention period of seven days. Because the code operand is also omitted, SD is assumed. The EXTENT statement indicates that the file is to be found on logical unit SYS023 (which you would expect to be assigned to a disk device), and that the volume serial number of the disk to be used is CMP060. When the type operand is omitted this way, the value for a data area (type 1) is assumed, so it is correct for this file. The sequence number is zero, which is usual for a sequential file. Since the extents of the file are given as relative track 1200 and 100 tracks in length, the five cylinders from 60 through 64 are used.

Now suppose a different program reads this same sequential file. Because the file label is stored on the disk this time, some of the entries in the JCL cards can be omitted as shown in the following example.

```
// DLBL INPUTAR,'ACCTS RCV TRANS'
// EXTENT SYS025,CMP060
```

The filename and file-ID operands must both be present in the DLBL card so that the file label can be found in the Volume Table of Contents. On the EXTENT statement only the logical unit is actually required. but I also coded the volume serial number so it would be checked. When the file is opened, the volume serial number of the disk found on SYS025 will be compared to CMP060. If it's the same, processing will continue. If it's different, processing will halt and a message will be issued on the console to tell the operator that the wrong pack has been mounted. If the serial number is omitted, this check isn't performed. If the type, sequence-number, relative-track, and number-of-track operands are omitted for an input file, the OPEN routine retrieves these values from the file label on the disk, and the retrieved values are used without checking.

Indexed sequential files (ISAM files) have several special requirements. When ISAM files are created, a minimum of two EXTENT statements is required—one EXTENT for the cylinder index area and one for the prime data area. If a master index is used, or if a general overflow area is to be reserved, an additional EXTENT statement must be included for each of them. As a result, the number of EXTENT statements for an ISAM file can be two, three, or four. If a master index is used, its EXTENT card must be sequence-number zero. If no master index is used, the first sequence number is one. Although the sequence order of the other EXTENT statements for an ISAM file isn't fixed, it is common practice to make the cylinder index EXTENT sequence number 1, the prime data EXTENT sequence number 2, and the general overflow area EXTENT sequence number 3 (if there is one).

Here is a sample of the use of DLBL and EXTENT cards to create an indexed sequential file:

```
// DLBL NEWMSTR,'CUSTOMER MASTER',99/365,ISC
// EXTENT SYS025,MST003,4,1,2590,2
// EXTENT ,,1,2,2600,160
// EXTENT ,,2,3,2760,40
```

In this case, the file is to be placed on a disk pack whose serial number is MST003. The cylinder index (no master index) is allocated to tracks 10 and 11 of cylinder 129; the prime data area is allocated to cylinders 130 through 137; and the general overflow area is allocated to cylinders 138 and 139. The DLBL has the maximum expiration date (99/365) so the file can never be accidentally destroyed by writing over it. Notice also that ISC is the code operand, which is proper for an ISAM creation function.

The first EXTENT statement describes the cylinder index area—type code 4 and sequence number 1. The relative-track operand is calculated by multiplying 129 by 20 and adding 10. Although ISAM indexes cannot reside on a prime data

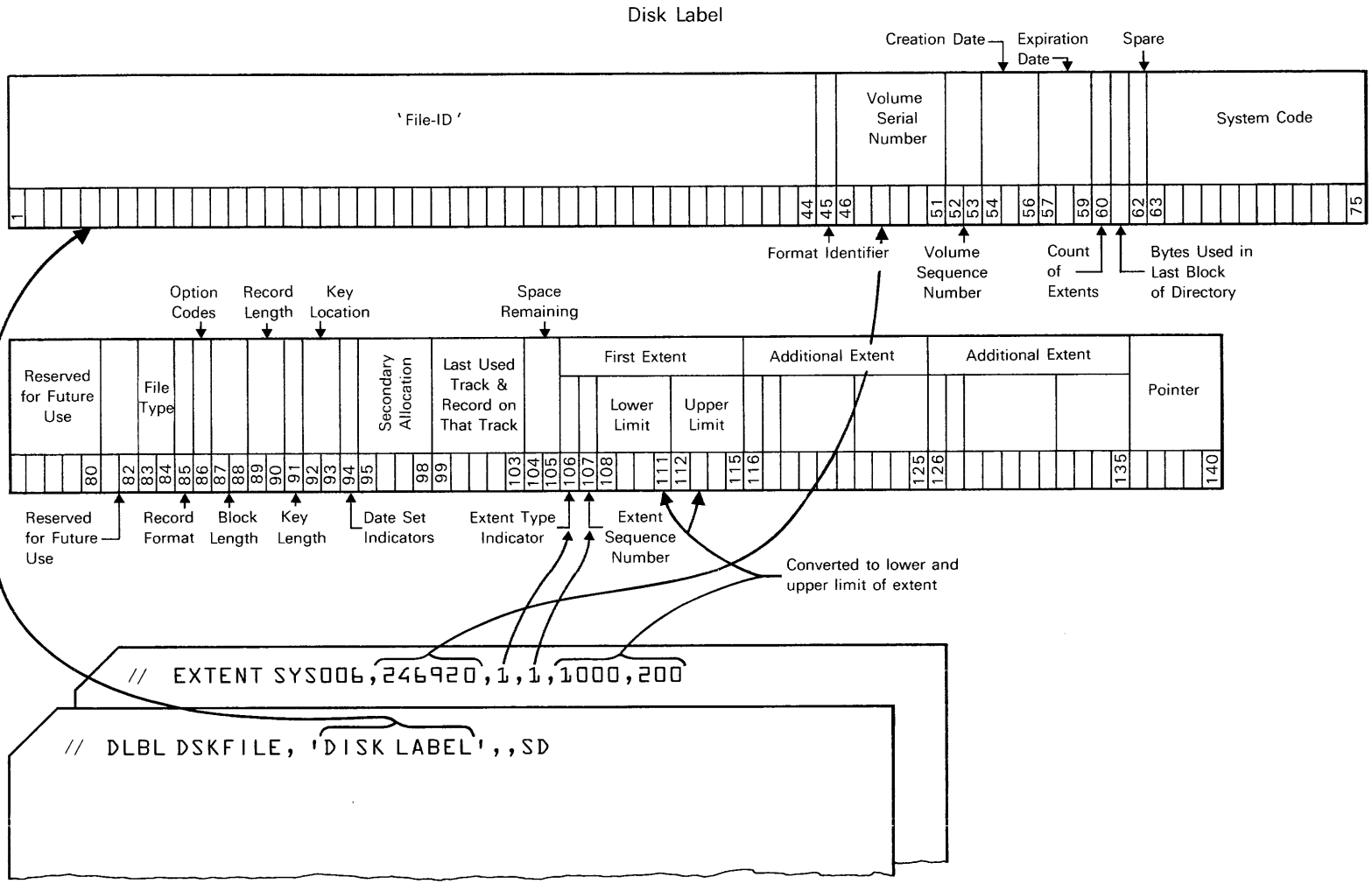


FIGURE 16-10 Standard Disk Label and DLBL and EXTENT Statement Formats

cylinder, they are often stored on some cylinder close to the prime data area or, sometimes, on a different disk pack. In the latter case, the disk pack is always mounted at the same time as the pack containing the prime data and overflow areas. It is common to find the indexes for many of the indexed sequential files that are used in an installation stored on some free cylinder of the upper portion of the systems pack.

The second EXTENT statement in the ISAM-create example describes the prime data area. When the logical-unit

DLBL Statement Format

```
// DLBL filename,
        'file-ID',
        [date],
        [code]
```

filename	The 1- to 7-character label of the program DTF
'file-ID'	From 1 to 44 characters enclosed in single quotes. If omitted for output, filename is used.
date	The creation date for an input file or the expiration date for an output file. It may be in the form YY/DDD where YY indicates the year and DDD indicates the number of the day in the year. Or, for an output file, it may also be written as DDD indicating the number of days the file should be retained. If omitted for output, the current date is used for the creation date and the expiration date is set at seven days later.
code	file organization code: SD=Sequential; ISC=ISAM load; ISE=other ISAM; DA=Direct. SD is assumed if omitted.

EXTENT Statement Format

```
// EXTENT logical unit,
          [volume serial number],
          [type],
          [sequence number],
          [relative track],
          [number of tracks]
```

logical unit	The logical-unit number (SYSxxx) on which the file is located.
volume serial number	The 6-character serial number of the disk pack
type	One digit indicating the extent type: 1=data area; 2=ISAM general overflow area; 4=ISAM index area. If omitted, 1 is assumed.
sequence number	From 1 to 3 digits indicating the sequence of this extent. Usually begins with sequence zero. For ISAM, sequence-number zero is reserved for the master index extent. If no master index is used, ISAM sequence starts with 1. Zero is assumed if sequence number is omitted.
relative track	From 1 to 5 digits indicating the starting track, relative to zero, of this extent area. Example: for a 2314 (20 tracks per cylinder), relative track 600 is first track of cylinder 30.
number of tracks	From 1 to 5 digits indicating the number of tracks allocated to this extent area. Usually specified in even cylinder increments for sequential and direct files. For ISAM, prime data area must be in even cylinder increments.

FIGURE 16-10 Standard Disk Label and DLBL and EXTENT Statement Formats (Continued)

and volume-serial-number operands are omitted as they are here, the values from the previous EXTENT card are assumed. The relative-track operand of this EXTENT starts the prime data area at track zero of cylinder 130 and the number-of-tracks operand reserves eight cylinders. (Incidentally, the number-of-tracks operand for an ISAM prime data EXTENT must always be in even cylinders.) The last EXTENT card for this file allocates cylinders 138 and 139 as a general overflow area.

For any program that processes nonsequential disk files, a LBLTYP statement must be included prior to the EXEC statement for the linkage editor. The LBLTYP statement reserves a label-processing area at the beginning of the user's program area. For nonsequential disk labels, the LBLTYP operand is NSD(nn) where nn specifies the maximum number of extents that any nonsequential file may have:

```
// LBLTYP NSD(04)
// EXEC LNKEDT
```

If a program is to process both nonsequential disk files and

```
// JOB TEST
// ASSGN SYS009,X'00C'
// ASSGN SYS024,X'132'
// DLBL TSTFLE,'TEST ISAM FILE',0,ISC
// EXTENT SYS024,TSTPAK,4,1,3600,1
// EXTENT ,,1,2,3620,40
// EXEC TSTCRT
   (input test file data)
/*
// OPTION LINK
// EXEC ASSEMBLY
   (update program source deck)
/*
// LBLTYP NSD(02)
// EXEC LNKEDT
// OPTION DUMP
// ASSGN SYS005,X'00C'
// DLBL MSTFLE,'TEST ISAM FILE',,ISE
// EXTENT SYS024
// EXEC
   (test input transactions)
/*
// ASSGN SYS009,X'00E'
// DLBL INPFLE,'TEST ISAM FILE',,ISE
// EXTENT SYS024
// EXEC TSTPRT
/&
```

FIGURE 16-11 JCL for a Disk Test Run

tape files, the NSD operand must be used.

Figure 16-11 shows a five-step job that will test a program that updates an ISAM master file. The first step creates the ISAM master file from card data using a program named TSTCRT that is stored in the core-image library. Only two EXTENT statements are necessary for the test file; one EXTENT is for the cylinder index and the other for the prime data area. The second, third, and fourth steps are the assembly, link-edit, and test of the update program. Just one EXTENT statement follows the DLBL in the test step and it has only the logical unit operand coded. I've coded it this way because I can assume that the disk volume TSTPAK is still mounted on SYS024 and all the rest of the extent information can be taken from the file label on the disk. The last job step prints out the updated ISAM records using a program named TSTPRT so the programmer can see if the test executed properly.

The same principles apply for files in direct organization. The LBLTYP card is used to indicate that the labels of a nonsequential file will be processed. The DLBL and EXTENT cards give the label information of the file, which is usually a one-extent file.

JCL FOR LINK EDITING

Your primary exposure to link editing so far has been restricted to the assemble, link, and test situation. In such a situation, the assembler program places the object program to be link-edited on the linkage-editor disk area. The program then becomes input to the linkage editor without any additional JCL direction from you. The output of the linkage-editor step is an object program located in the temporary area of the core-image library.

But what about the I/O modules that get link-edited with your program to form the phase that is actually executed? To get these modules link-edited with the rest of the program, the assembler places some linkage-editor control statements in the SYSLNK disk area. These tell the linkage editor to retrieve the appropriate modules from the relocatable library and to include them in the phase.

To get subprograms link edited with other object modules, you too can use linkage-editor control statements. Although there are four of these—PHASE, INCLUDE, ENTRY, and ACTION—only the PHASE and INCLUDE statements will be useful to you until you become more sophisticated in the use of the operating system. You can find detailed explanations of all the linkage-editor control statements in IBM manual GC24-5036, *DOS System Control and Services*.

Figure 16-12 illustrates the JCL for a link-edit and test job and shows both the PHASE and INCLUDE statements in their most common forms. Linkage-editor control statements always follow immediately after the OPTION statement and they must have at least column 1 blank. In figure 16-12, the PHASE statement assigns the phase name REPORT to the executable phase that will be made up of the modules MAINPROG, SUBPROGA, the object deck in the job stream, and SUBPROGX. If a PHASE statement isn't included in a link-edit step, the program name of the first module becomes the phase name. In most cases, the second operand of the PHASE statement, called the *origin*, should be an asterisk (*) although there are several other ways to code this operand. The asterisk causes the phase to be assigned to the first storage location in the problem program area.

When the INCLUDE statement has an operand, it means the module named as the operand is to be retrieved from the relocatable library and included in the phase. Examples of such an INCLUDE statement are the INCLUDE statements for MAINPROG, SUBPROGA, and SUBPROGX in figure 16-12. If no operand is coded, it means that a module in object-deck form follows. The end of this object deck must be indicated with an end-of-data card (/*). Because the executable phase in figure 16-12 processes a tape file, the LBLTYP and TLBL cards are also included.

```
// JOB LINK AND TEST
// OPTION LINK
   PHASE REPORT,*
   INCLUDE MAINPROG
   INCLUDE SUBPROGA
   INCLUDE
   (object deck of another subprogram)
/*
   INCLUDE SUBPROGX
// LBLTYP TAPE
// EXEC LNKEDT
// ASSGN SYS008,X'00E'
// ASSGN SYS016,X'182'
// TLBL RPTTAP,'REPORT TAPE FILE'
// EXEC
/&
```

FIGURE 16-12 Link-Edit and Test Example

JCL FOR LIBRARY MAINTENANCE

Several programs are supplied as part of DOS to allow you to maintain the three system libraries. The phases, modules, and books of the core-image, relocatable, and source-statement libraries can be catalogued (added), deleted, renamed, displayed, and punched out as needed. Figure 16-13 shows the JCL for maintaining these libraries. Notice that MAINT is the primary program for cataloging and deleting entries from the libraries—except for cataloging in the core-image library, which is done by the linkage editor program. MAINT also allows the libraries to be consolidated. This involves moving active entries together so that all deleted entries are actually removed from the library and the reclaimed disk space can be reused. The program DSERV will display the directory of any or all of the libraries. Displaying or punching elements from the libraries is done through CSERV, RSERV, or SSERV depending on which library is to be used.

To catalogue a phase in the core-image library, the CATAL option is used with or without a PHASE card. To assemble and catalog the reorder-listing program of figure 3-4, for example, these JCL cards could be used:

```
// JOB CATALOG
// OPTION CATAL
// EXEC ASSEMBLY
   (source deck)
/*
// EXEC LNKEDT
/&
```

Since the PHASE card has been omitted, the phase name (REORDLST) is taken from the START card. Once catalogued,

Function	Core-Image Library	Relocatable Library	Source-Statement Library

catalog	// OPTION CATAL
	PHASE .... (optional)
	INCLUDE ....
	// LBLTYP .... (if necessary)
	// EXEC LNKEDT
		// EXEC MAINT
		CATALR modulename
		(object deck of module)
			// EXEC MAINT
			CATALS sublib.bookname...

delete	// EXEC MAINT DELETC phasename...	// EXEC MAINT DELETR modulename... DELETR modulename...	// EXEC MAINT DELETS sublib.bookname...
rename	// EXEC MAINT RENAMC oldname,newname...	// EXEC MAINT RENAMR oldname,newname...	
condense	// EXEC MAINT CONDS CL	// EXEC MAINT CONDS RL	// EXEC MAINT CONDS SL
display directory	// EXEC DSERV DSPLY CD	// EXEC DSERV DSPLY RD	// EXEC DSERV DSPLY SD
display macro			// EXEC SSERV DSPLY sublib.bookname...
punch out	// EXEC CSERV PUNCH phasename	// EXEC RSERV PUNCH modulename...	// EXEC SSERV PUNCH sublib.bookname...

FIGURE 16-13 JCL for Library Maintenance

the program can be executed with this EXEC card:

```
// EXEC REORDLST
```

If a PHASE card is included, it should be after the OPTION card in this form:

```
PHASE phasename,*
```

where the phase name is made up of eight or less letters, or letters and numbers, always beginning with a letter. It can be the same as the program name in the START statement or it can be different.

Figure 16-14 shows the JCL for a job that illustrates several other library-maintenance procedures. The first job step executes DSERV to print the directories of all three libraries. Step 2 catalogues the object form of PROGA in the relocatable library. If PROGA was already catalogued in the relocatable library, the new version would replace the old; it isn't necessary to delete the old one first. This replacement function is the same for all three libraries. Step 3 punches two object modules named PROGB and PROGC, and step 4 displays the assembler-language books named GET and PUT. Within the source-statement library, all code in the assembler-language sublibrary has the sublib prefix of the letter A followed by a period as in A.GET and A.PUT.

```
// JOB LIB MAINT
// EXEC DSERV
   DSPLY ALL
/*
// EXEC MAINT
   CATALR PROGA
   (object deck of PROGA)
/*
// EXEC RSERV
   PUNCH PROGB,PROGC
/*
// EXEC SSERV
   DSPLY A.GET,A.PUT
/*
/&
```

FIGURE 16-14 Sample Library-Maintenance Job

JCL FOR UTILITY PROGRAMS

The DOS system includes several utility programs that perform either file-to-file copy functions such as card-to-printer or tape-to-disk copies, or special functions such as clearing a disk area or displaying the contents of the Volume Table of Contents to show all the file labels of a pack. Figure 16-15 is a list of the utility programs commonly used in a DOS system that has both tape and disk devices. IBM manual GC24-3465, *DOS Utility Program Specifications*, contains the detailed descriptions of the JCL and control cards required for each of the programs. In general, however, the programs all use JCL similar to that shown in figures 16-16 and 16-17. First, specific logical units are assigned for the input and output files—usually SYS004 for the input file and SYS005 for the output file. Second, any necessary TLBL or DLBL and EXTENT statements are included with UIN as the input filename and UOUT as the output filename. Third, the EXEC statement names the utility program by its phase name as shown in figure 16-15. Finally, a *utility modifier statement* follows the EXEC statement. Utility modifier statements are unique to each program; but they

Program Name	Program Function
File-to-File Programs	
CDPP	Card-to-Printer or Card-to-Punch
CDTP	Card-to-Tape
CDDK	Card-to-Disk
TPTP	Tape-to-Tape
TPCD	Tape-to-Card
TPDK	Tape-to-Disk
TPPR	Tape-to-Printer
DKPR	Disk-to-Printer
DKCD	Disk-to-Card
DKTP	Disk-to-Tape
DKDK	Disk-to-Disk
Special Purpose Programs	
INTD	Initialize Disk
INTP	Initialize Tape
CLRDSK	Clear Disk
CRDD	Copy/Restore Disk to Disk
CRDT	Copy Disk to Tape
CRTD	Restore Tape to Disk
TPCP	Tape Compare
LISTVTOC	VTOC Display

FIGURE 16-15 DOS Utility Programs

follow a general format that specifies the type of function desired, gives the record and block lengths of the input and output files, and selects some optional functions of the individual utility program. In cases in which field manipulation is required, one or more *field-select statements* may follow the utility modifier statement. Field-select statements have operands that specify the starting position and length of the input fields that are to be moved, packed, or unpacked from an input field to an output field. The output fields are also identified by starting position and length. The last utility control statement is always // END.

As examples, the JCL for a tape-to-printer and card-to-disk run are presented in figures 16-16 and 16-17.

```
// JOB TAPE TO PRINTER
// ASSGN SYS004,X'181'
// ASSGN SYS005,X'00E'
// TLBL UIN,'TAPE TEST'
// EXEC TPPR
// UTP TL,FF,A=(80,400),B=(132),IU,OC,S2,PY
// END
/&
```

FIGURE 16-16 JCL for Tape-to-Printer Utility

These utilities may also be useful in your testing operations for tape and disk problems. The utility modifier statement in the tape-to-printer example indicates that the fixed-length (FF) input tape records of 80 bytes within 400-byte blocks (A=(80,400)) are to be listed (TL) on the printer in 132 byte (B=(132)) lines. The tape record characters are to be printed in EBCDIC form (OC) with double spacing (S2) and page numbering (PY). When the whole input tape has been processed, it is to be rewound and unloaded (IU). You can use this tape-to-printer JCL to print your own test tapes by altering the TLBL statement and the input record and block length (A=(80,400)) to suit your tape file. If the *DOS Utility Program Specifications* manual is available, you may want to find out what other options this utility offers.

In the card-to-disk example, the card data is to be blocked and field-selected (TRF) as it is transferred to the fixed length (FF) disk records. The cards are 80-bytes long and unblocked (A=(80,80)) and are punched in EBCDIC format (I1). The disk output file will be written as 40-byte records, blocked 10 per block (B=(40,400)) with no write-checking performed (ON) on the 2314 (E=(2314)) device. Since field-selection has been indicated, a field-select statement (// FS) follows the utility modifier statement. It says that columns 1 through 29 of the input card are to be transferred to the 29 bytes of the disk record beginning with byte 1 (1,29,1). Then a 5-byte field starting in column 30 of the card is to be packed into a 3-byte field that begins in byte 30 of the disk record (30,(P,5,3),30). Finally, a second 5-byte field, beginning in column 35 of the card, is to be packed into a 3-byte field that begins in byte 33 of the disk record (35,(P,5,3),33).

You may want to use this type of JCL in your testing in order to create a test file. To perform a simple card-to-disk copy, change the first operand of the utility modifier statement from TRF to TC (but don't leave any blank columns) and change the output record length and block size (B=(40,400)) to those you need—perhaps B=(80,400). Finally, omit the field-select statement and change the DLBL and EXTENT statements to reflect your own disk file area.

DISCUSSION

You may have noticed that nowhere in this introduction to JCL is there any mention of multiprogramming. The reason is that the same job-control setups are used regardless of the partition the job will eventually be run in. In fact, if your programs are run in batches, you will never know which partition they were actually executed in or even if they were run on a multiprogramming system. As a result, although multiprogramming, spooling, and virtual storage can improve the productivity of a computer system, they don't affect the tasks of an assembler-language programmer. It is at the operational level that concerns with partition sizes, the partition to be used for a program, and the like become important.

```
// JOB CARD TO DISK
// ASSGN SYS004,X'00C'
// ASSGN SYS005,X'131'
// DLBL UOUT,'TEST TIME CARDS'
// EXTENT SYS005,WRK002,1,0,200,20
// EXEC CDDK
// UCD TRF,FF,A=(80,80),B=(40,400),I1,ON,E=(2314)
// FS 1,29,1/30,(P,5,3),30/35,(P,5,3),33
// END
/&
```

FIGURE 16-17 JCL for Card-to-Disk Utility

Terminology

standard options
device address
logical unit
symbolic unit
standard assignments
system logical units
programmer logical units
expiration date
extents
origin
utility modifier statement
field-select statement

Objective

Given the description of a job that fits one of the job-control-card patterns described in this chapter, make up job-control cards that will cause the intended sequence of programs to be executed.

Problems

1 Suppose the program in figure 15-4 is stored in the core-image library under the name ORDUPD. The transaction file, which is to be mounted on 2314 spindle X'131', has the file identifier ORDER TRANS and can be found in cylinders 11-20 of the pack. The pack has volume serial number 219109. The master file, which is to be mounted on spindle X'132', has the file identifier ORDER MASTER and is in cylinders 10-55 on the pack. This pack has volume serial number 103111. The report file, which is to be written on a pack mounted on spindle X'133', has cylinders 151-160 as its extents. Any pack can be used for this temporary file, since the next program will print the report from it. The standard assignments are those given in figure 16-7. Write the JCL statements that are necessary to run this program.

2 Suppose the report file of problem 1 is to be printed immediately after creation by a disk-to-printer utility program with the following utility modifier and field select cards:

```
// UDP TLF,FF,A=(80,80),B=(132),OC,PY,S2,E=(2314)
// FS 1,4,1/5,4,8/9,20,15/29,6,38/35,4,47
// FS 39,8,54/47,6,65/53,4,74/57,8,81
```

What JCL cards would cause proper execution of this utility?

3 Suppose the programs of figures 15-9 (ISAM creation) and 15-13 (ISAM sequential retrieval) are to be assembled and tested in successive job steps. The file is to be stored in cylinders 65-70 of a pack that has volume serial number TST001 and is mounted on spindle X'134'. The cylinder index is to be stored in cylinder 71. Use the name TEST FILE for the file identifier of the master file and make the expiration date the same as the creation date. The standard assignments are those of figure 16-7. Write the required JCL.

4 Suppose you have written a payroll subprogram named GROSCALC and a main program named PAYROLL5.

a. Write the JCL for assembling GROSCALC and punching an object deck.

b. Write the JLC for using the object deck to catalog GROSCALC in the relocatable library.

c. Assuming that only card input and printer output are involved, write the JCL for assembling the main program, linking it with the subprogram, and testing the complete phase.

d. Assume that the test run worked but you decide to change the name of the subprogram from GROSCALC to PRSUB1. Write the JCL for the name change.

Solutions

1
```
// JOB TEST UPDATE PROGRAM
// OPTION DUMP
// DLBL SHIPTR,'ORDER TRANS'
// EXTENT SYS020,219109
// DLBL ORDERS,'ORDER MASTER'
// EXTENT SYS025,103111
// DLBL RPTTR,'REPORT FILE',0
// EXTENT SYS030,,,,3020,200
// EXEC ORDUPD
/&
```

Notice that no ASSGN cards are needed because the standard assignments are used in the EXTENT cards. (Remember that in an assembler-language program a logical unit is generally not given to a disk file in its DTF. Instead, the logical unit is assigned in the EXTENT card at the time of program execution.)

2
```
// JOB DSKTOPRT
// ASSGN SYS004,X'133'
// ASSGN SYS005,X'00E'
// DLBL UIN,'REPORT FILE'
// EXTENT SYS004
// EXEC DKPR
// UDP TLF,FF,A=(80,80),B=(132),OC,PY,S2,E=(2314)
// FS 1,4,1/5,4,8/9,20,15/29,6,38/35,4,47
// FS 39,8,54/47,6,65/53,4,74/57,8,81
// END
/&
```

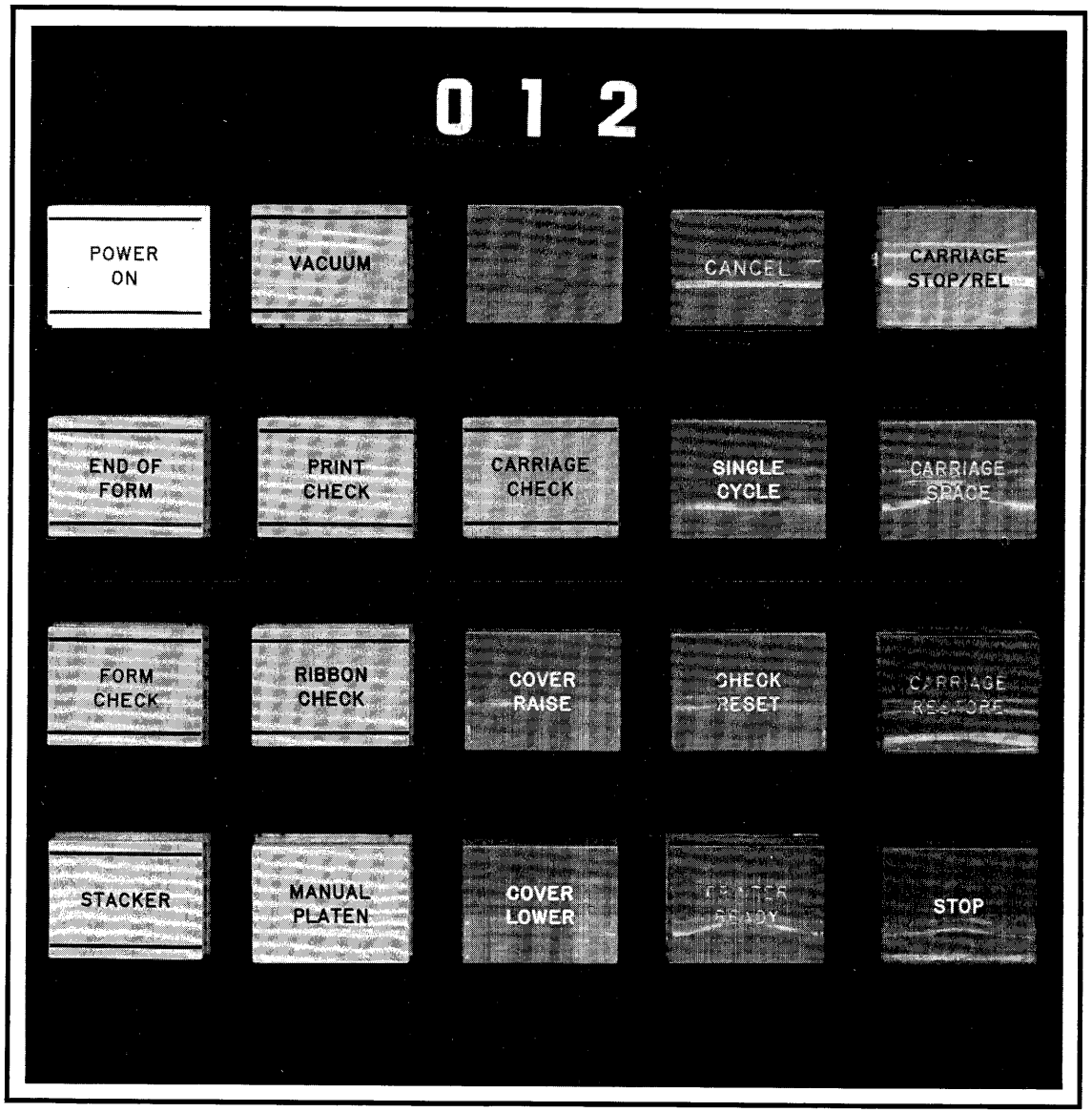
3
```
// JOB ASSEMBLE AND TEST
// OPTION LINK,DUMP
// EXEC ASSEMBLY
   (source deck for ISAM creation)
/*
// LBLTYP NSD(02)
// EXEC LNKEDT
// DLBL INVMSTR,'TEST FILE',0,ISC
// EXTENT SYS035,TST001,4,1,1420,20
// EXTENT ,,1,2,1300,120
// EXEC
   (data deck)
/*
// EXEC ASSEMBLY
   (source deck for ISAM retrieval)
/*
// LBLTYP NSD(02)
// EXEC LNKEDT
// DLBL INVMSTR,'TEST FILE',,ISE
// EXTENT SYS035
// EXEC
/&
```

4 a.
```
// JOB PUNCH
// OPTION DECK
// EXEC ASSEMBLY
   (source deck)
/*
/&
```

b.
```
// JOB CATALOG
// EXEC MAINT
   CATALR GROSCALC
   (object deck)
/*
/&
```

c.
```
// JOB TEST
// OPTION LINK,DUMP
   INCLUDE GROSCALC
// EXEC ASSEMBLY
   (source deck for main program)
/*
// EXEC LNKEDT
// EXEC
   (data deck)
/*
/&
```

d.
```
// JOB CHANGE
// EXEC MAINT
   RENAMR GROSCALC,PRSUB1
/&
```


0 1 2
POWER ON
VACUUM
CANCEL
CARRIAGE STOP/REL
END OF FORM
PRINT CHECK
CARRIAGE CHECK
SINGLE CYCLE
CARRIAGE SPACE
FORM CHECK
RIBBON CHECK
COVER RAISE
CHECK RESET
STACKER
MANUAL PLATEN
COVER LOWER
STOP

Part 7

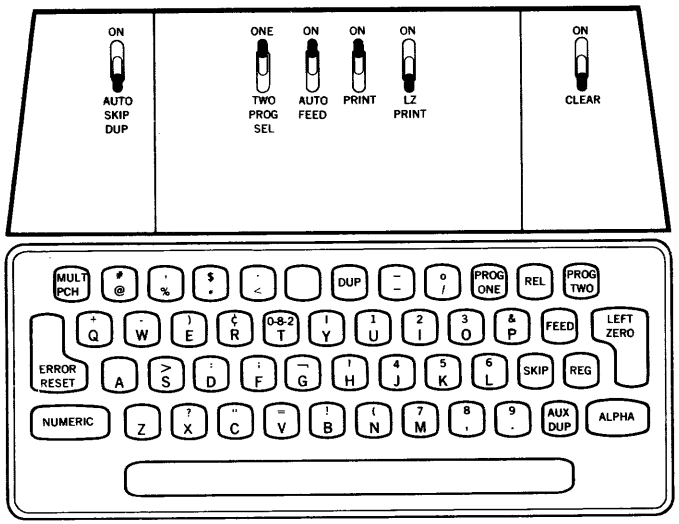


FIGURE A-1 Keyboard of the 029 Keypunch

Format of a Program Card for Punching Assembler-Language Source Deck

Columns	Punches	Comments
1-9	1AAAAAAAA	Label field, alphabetic
10-15	1AAAAA	Operation Code field, alphabetic
16-71	1AAA..(55 As)..AA	Operand field, alphabetic
72	1	Continuation column, alphabetic
73-80	1AAAAAAA	Sequence field, alphabetic

If sequence field is to be all numbers, punch:

73-80	b&&&&&&& (b = blank)	Sequence field, numeric

If sequence numbers are not required, fill columns 73-80 with As. If remarks are coded, you might want to break the operand and remark fields:

16-49	1AAA..(33 As)..AA	Operand field, alphabetic
50-71	1AAA..(21 As)..AA	Remarks field, alphabetic

Setup

1 Raise the starwheels in the program unit by pushing the program-control level to the right.

2 Pull the cover of the program unit forward.

3 Lift the program drum from its spindle in the program unit.

4 Remove the old program card, if there is one, and mount the new program card on the program drum. (If you don't know how to mount a program card, ask someone to show you.)

5 Put the drum back on its spindle in the program unit making sure that the alignment pin on the bottom of the drum is in the alignment hole in the program unit.

6 Close the cover of the program unit and lower the starwheels by pushing the program-control lever to the left. Then, press the REL key. This causes the program drum to rotate once, thus engaging the starwheels in the holes in the program card.

7 Place blank cards in the input hopper.

8 Depress the FEED key twice to move a card to the punching station.

9 Set the switches located above the keyboard so that AUTO FEED, PRINT, and AUTO SKIP DUP are on and LZ PRINT is off.

Operating Procedures

1 Skipping and shifting of the keyboard are now under program control as dictated by the program card. To punch alphabetic data in a numeric field, use the ALPHA key. To punch numeric data in an alphabetic field, use the NUMERIC key. To skip to the end of a field, press the SKIP key.

2 If you make an error, this is how you can correct it. First, push the REL key; this releases the error card to the reading station. Second, turn off the AUTO SKIP DUP switch. Third, press the DUP key until the keypunch reaches the column that contains the error. This

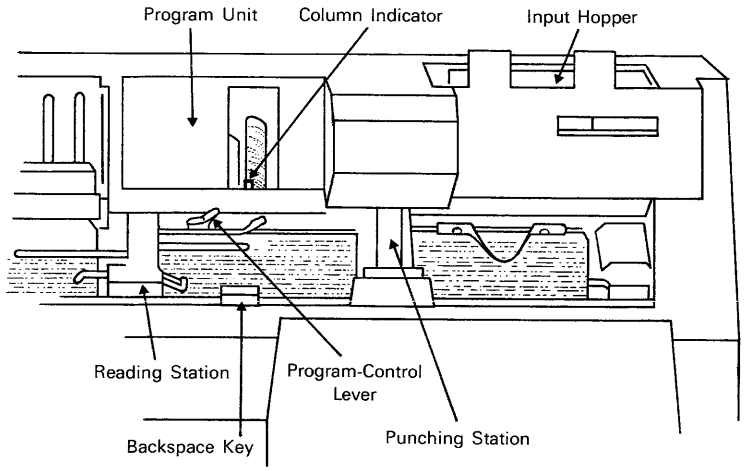


FIGURE A-2 Features of the 029 Keypunch

procedure duplicates all data that was punched correctly before the error was made. Now, correct the error, turn AUTO SKIP DUP back on, and continue punching as usual. Be sure, however, that you remove the incorrect card from the source deck.

VARIATIONS WHEN USING THE 026 KEYPUNCH

1 The 026 requires a warmup period of about half a minute after the mainline switch has been turned on. Because of this, it is customary to press the REL key after turning the machine on. When the warmup period is over, the program drum rotates once because of the release key, and this indicates that the machine is ready for operation.

2 There is no CLEAR switch on the 026. To move cards to the output stacker, the AUTO FEED switch must be turned off. Then the release and register (REG) keys are pressed alternatively—REL, REG, REL, REG, REL, REG—until all cards have been moved to the stacker.

3 The keyboard of the 026 (illustrated in figure A-3) does not have all of the special characters that the 029 has. For example, the semicolon (;), the quotation mark ('), and the equals sign (=) are not included. To punch these characters on the 026, the MULT PCH key must be used. When this key is depressed, the keyboard is in numeric shift and the card does not advance until the MULT PCH key is released. To punch a quotation mark, which consists of a 5- and an 8-punch, the MULT PCH key is pressed and both the 5 and the 8 keys are struck. When the MULT PCH key is released, the card moves to the next card column. The special characters for which MULT PCH must be used are

Character	Card Code
(	12,5,8
+	12,6,8
)	11,5,8
;	11,6,8
– (minus)	0,5,8
?	0,7,8
'	5,8
=	6,8

SPECIAL PROCEDURES

Resetting a Locked Keyboard

In some cases, the keyboard will lock because of an error—if, for example, some nonnumeric character such as the letter A

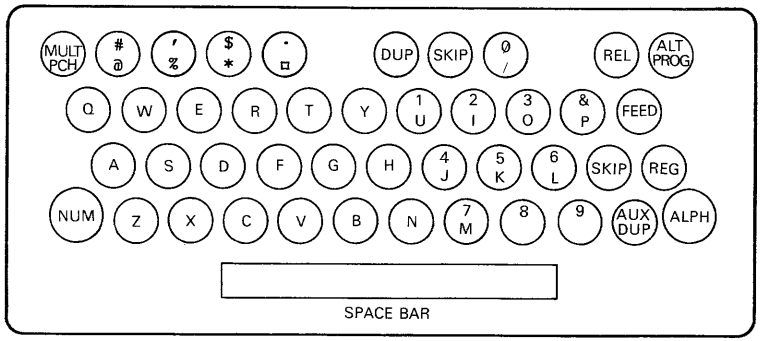


FIGURE A-3 Keyboard of the 026 Keypunch

is keyed in a numeric field. On the 026 the keyboard can be unlocked by pressing either the REL or the backspace key; on the 029 it can be released by pressing the ERROR RESET key.

Duplicating a Damaged Card

1 Turn off AUTO FEED and AUTO SKIP DUP. Clear the machine of cards. (Cards may be left in the stacker, however.)

2 Smooth the damaged card and insert it through the plastic guides at the reading station. Slide the card forward as far as it will go, and then pull it back about one-quarter of an inch.

3 Place a blank card through the guides at the punching station. Again, slide the card forward until it stops, and then pull it back about one-quarter inch.

4 Press the REG key to register the cards at the reading and punching station.

5 Press the DUP key until all of the data in the damaged card has been punched into the new card. Then, press the REL key to release both cards.

6 Clear the cards from the machine.

Correcting a Few Columns of an Incorrect Card

1 Do steps 2-4 (described above) for duplicating a damaged card.

2 Push the DUP key for all correct card columns; rekey the error columns.

Adding Data to a Card

1 Insert the card through the guides at the punching station. Slide the card forward as far as it will go; then pull it back about one-quarter inch.

2 Press the REG key to register the card at the punching station.

3 Using the space bar, space over the punched columns to the column(s) that need to be punched.

4 Key the additional data in the appropriate column(s).

Appendix B.

DTF Summary

Priority	Keyword	Programmer Code	Default	Remarks
Required	DEVADDR	SYSnnn		SYSIPT is the usual code for a card reader, SYSPCH for a card punch.
Required	IOAREA1	Name of I/O area		
Optional	IOAREA2	Name of second I/O area		Use for overlap; WORKA must also be coded.
Optional	WORKA	YES		YES if second I/O area is specified; a work area can also be used with a single I/O area. GET or PUT names work area.
Optional	DEVICE	1442 2501 2520 2540	2540	Usually omitted if 2540.
Optional	EOFADDR	Label of first instruction of EOF routine		Required for input file.
Optional	TYPEFLE	INPUT OUTPUT	INPUT	Usually omitted for input. Must be OUTPUT for punching.

DTFCD Operand Summary

Priority	Keyword	Programmer Code	Default	Remarks
Required	DEVADDR	SYSnnn		SYSLST is most common.
Required	IOAREA1	Name of I/O area for print lines		
Optional	BLKSIZE	Length of output lines	121	For 1403, usually 132 or 133 for control character use.
Optional	CONTROL	YES		Required if CNTRL macro is used; otherwise omit. If used, omit CTLCHR.
Optional	CTLCHR	ASA YES		Required if line control characters are used. Usual code is ASA; requires ASA line control character as first in print line. YES requires 360 line control character. If used, omit CONTROL.
Optional	DEVICE	1403 1443 3211	1403	Usually omitted for 1403.
Optional	IOAREA2	Name of second I/O area		Use for overlap; WORKA must also be coded.
Optional	WORKA	YES		Required if second I/O area used. PUT names work area. Can also be used for single I/O area.
Optional	PRINTOV	YES		Required if PRTOV macro will be used. Otherwise omit.

DTFPR Operand Summary

Priority	Keyword	Programmer Code	Default	Remarks
Required	BLKSIZE	Number of bytes in block		Must be same as length of I/O area.
Required	DEVADDR	SYSnnn		Logical unit for the tape file.
Required	FILABL	STD NSTD NO	NO	Specifies that the tape volume and file have standard (STD). non-standard (NSTD), or no (NO) labels.
Required	IOAREA1	Name of first or only I/O area		Length of I/O area must be large enough to hold entire tape block.
Optional	IOAREA2	Name of second I/O area		If IOAREA2 is coded, IOREG or WORKA must be coded.
Optional	EOFADDR	Label of first instruction of EOF routine		Required for input files.
Optional	TYPEFLE	INPUT OUTPUT	INPUT	Usually omitted for input files.
Optional	RECFORM	FIXUNB FIXBLK VARUNB VARBLK	FIXUNB	Specifies tape block format. Most common is fixed-length, blocked (FIXBLK).
Optional	RECSIZE	Record length in number of bytes		Used only when RECFORM=FIXBLK.
Optional	IOREG	Register number (nn)		Coded if records are processed in I/O area instead of work area. Address of record is placed in I/O register by I/O module. Omit WORKA operand.
Optional	WORKA	YES		Records processed in work area named in GET or PUT.

DTFMT Operand Summary

Priority	Keyword	Programmer Code	Default	Remarks
Required	BLKSIZE	Length of I/O area		Block length, but for output files must include 8 bytes for count.
Required	IOAREA1	Name of I/O area		Length equals block length. For output files, must include 8 bytes at front for count field.
Optional	IOAREA2	Name of second I/O area		Must be same length as IOAREA1.
Optional	EOFADDR	Label of first instruction of EOF routine		Must be coded for input files.
Optional	DEVICE	2311 2314 3330	2311	
Optional	RECFORM	FIXUNB FIXBLK VARUNB VARBLK	FIXUNB	FIXBLK is most common.
Optional	RECSIZE	Record length		Used only if RECFORM=FIXBLK
Optional	TYPEFLE	INPUT OUTPUT	INPUT	Usually omitted for input files.
Optional	UPDATE	YES		Used only when records are to be updated in place. PUT used to rewrite record just read and updated. TYPEFLE must be INPUT.
Optional	IOREG	Register number (nn)		Included if blocked records to be processed in I/O area. Omit WORKA.
Optional	WORKA	YES		Records are to be processed in work area named in GET or PUT. Omit IOREG operand.
Optional	VERIFY	YES		Output records are to be write verified.

DTFSD Operand Summary

Priority	Keyword	Programmer Code	Default	Remarks
Required	DSKXTNT	Maximum number of extents for this file		
Required	IOROUT	LOAD ADD RETRVE ADDRTR		Specify type of processing: loading file, adding records, retrieving (also updating in place), or add/retrieve.
Required	RECFORM	FIXUNB FIXBLK		
Required	RECSIZE	Number of bytes in record		
Required	KEYLEN	Key length		
Optional	NRCDS	Number of records per block	1	Required for RECFORM=FIXBLK
Optional	DEVICE	2311 2314 3330	2311	
Optional	HINDEX	2311 2314 3330	2311	Specify the type of DASD device that holds the highest level index. Usually same as DEVICE.
Optional	MSTIND	YES		Indicates that master index is used for this file. If omitted, no master index is created or expected in add or retrieve.
Optional	INDAREA	Name of user area reserved to hold cylinder index in storage		Cylinder index can be held in storage to process records faster. INDAREA names the storage area; INDSIZE specifies number of bytes.
Optional	INDSIZE	Number of bytes reserved to store cylinder index in INDAREA field.		Must be included if INDAREA is coded. Most effective size is large enough to hold entire index: (Nbr of prime cyls+4)×(keylgth+6)
Optional	CYLOFL	Number of tracks per cylinder for overflow records		Usually 1 or 2 for 2311; 2, 3, or 4 for 2314 and 3330. Depends on addition activity.
Optional	KEYLOC	Starting position of key field in blocked records		For blocked records, key is embedded in data of each record. Required if RECFORM=FIXBLK.
Optional	KEYARG	Name of user key field		In random retrieval, user supplies key of record to be read or written in this field. Also used to specify starting key for skip sequential.

DTFIS Operand Summary

Priority	Keyword	Programmer Code	Default	Remarks
Optional	TYPEFLE	RANDOM SEQNTL RANSEQ		Coded only when IOROUT=RETRVE or ADDRTR. Indicates random, sequential, or combination retrieval.
Optional	IOAREAL	Name of I/O area for load function		Must be coded if IOROUT=LOAD, ADD, or ADDRTR. For LOAD, length must be count (8)+key+block. Same for ADD of blocked records. For unblocked records, must also allow 10 bytes for sequence-link field.
Optional	IOAREAR	Name of I/O area for random processing		Must be coded if IOROUT=RETRVE or ADDRTR and TYPEFLE=RANDOM or RANSEQ. For unblocked records, length must be sequence link (10)+record length. For blocked records length is block length or record length+10, whichever is more.
Optional	IOAREAS	Name of I/O area for sequential processing		Must be coded if IOROUT=RETRVE or ADDRTR and TYPEFLE=SEQNTL or RANSEQ. For unblocked records, length must be key length+sequence link (10)+record length. For blocked records, length is same as for IOAREAR.
Optional	IOAREA2	Name of second I/O area for load or sequential retrieve		Two I/O areas can be used for loading or sequential processing. Length must be the same as IOAREAL or IOAREAS.
Optional	WORKL	Name of work area for load or add		For unblocked records, work area must hold key and record. For blocked records, only record. Omit IOREG.
Optional	WORKR	Name of work area for random processing		Work-area length must be one record length. Omit IOREG.
Optional	WORKS	YES		Indicates that GET or PUT will specify work area. Work-area length same as for WORKL. Omit IOREG.
Optional	IOREG	Register number (n)		For load or sequential processing, records can be processed in I/O area. I/O module puts record address in I/O register. Omit WORKL or WORKS.
Optional	VERIFY	YES		Request write verification of output records.

DTFIS Operand Summary (Continued)

Priority	Keyword	Programmer Code	Default	Remarks
Required	BLKSIZE	Length of I/O area		See IOAREA1 for length.
Required	DEVICE	2311 2314 3330	2311	
Required	TYPEFLE	INPUT OUTPUT	INPUT	
Required	IOAREA1	Name of I/O area		Length of I/O area must include 8 bytes for count if AFTER=YES is coded; must include bytes for key if KEYLEN is coded.
Required	ERRBYTE	Name of two-byte error-code field reserved by user		User must define two-byte field for I/O module to post error indications. See figure 15-20 for description of codes.
Required	SEEKADR	Name of user disk-address field		Length of address field depends on address format: 8 bytes for physical address, 10 for decimal relative-track address, 4 for hex relative-track address.
Optional	IDLOC	Name of disk-address field for address returned after READ or WRITE		Length of IDLOC must be same as SEEKADR field. Address of record read or written will be returned in same format as SEEKADR.
Optional	DSKXTNT	Max number of extents for this file		Indicates that relative-track-address format is used. RELTYPE specifies hex or decimal format.
Optional	RELTYPE	HEX DEC		Specifies format of relative-track address as hex (4 bytes) or decimal (10 bytes).
Optional	RECFORM	FIXUNB VARUNB UNDEF		If undefined (UNDEF) is coded, RECSIZE must be coded.

DTFDA Operand Summary

Priority	Keyword	Programmer Code	Default	Remarks
Optional	RECSIZE	Register number (nn)		Required if RECFORM=UNDEF. Register must contain binary record length before WRITE, will hold length of record after READ.
Optional	KEYARG	Name of user key field		Used only if KEYLEN is coded. Length of KEYARG field should be equal to key length. KEY must be placed here before READ or WRITE by key.
Optional	KEYLEN	Length of Key		Used only if file in count-key-data format.
Optional	XTNTXIT	Label of first instruction in extent processing routine.		Allows user to code routine to process or save extent information. See Chapter 15 for details.
Optional	READID	YES		READ filename,ID will be used.
Optional	READKEY	YES		READ filename,KEY will be used.
Optional	SRCHM	YES		Request search of multiple tracks to end of cylinder for READ or WRITE by KEY.
Optional	WRITEID	YES		WRITE filename,ID will be used.
Optional	WRITEKY	YES		WRITE filename,KEY will be used.
Optional	AFTER	YES		WRITE filename,AFTER or RZERO will be used.
Optional	VERIFY	YES		Request write verification of output records.

DTFDA Operand Summary (Continued)

Decimal Arithmetic Instructions

Instruction	Mnemonic Op Code	Machine Op Code	Operand Format	
			Explicit	Implicit
Add Decimal	AP	FA	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
Compare Decimal	CP	F9	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
Divide Decimal	DP	FD	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
Multiply Decimal	MP	FC	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
Subtract Decimal	SP	FB	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
Zero and Add Decimal	ZAP	F8	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2

Fixed-Point Arithmetic Instructions

Instruction	Mnemonic Op Code	Machine Op Code	Operand Format	
			Explicit	Implicit
Add	A	5A	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Add	AR	1A	R1,R2	
Add Halfword	AH	4A	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Compare	C	59	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Compare	CR	19	R1,R2	
Compare Halfword	CH	49	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Divide	D	5D	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Divide	DR	1D	R1,R2	
Load	L	58	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load	LR	18	R1,R2	
Load Halfword	LH	48	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load Multiple	LM	98	R1,R3,D2(B2)	R1,R3,S2
Multiply	M	5C	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Multiply	MR	1C	R1,R2	
Multiply Halfword	MH	4C	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Store	ST	50	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Store Halfword	STH	40	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Store Multiple	STM	90	R1,R3,D2(B2)	R1,R3,S2
Subtract	S	5B	R1,D2(X2)	R1,S2(X2) or R1,S2
Subtract	SR	1B	R1,R2	
Subtract Halfword	SH	4B	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2

Floating-Point Arithmetic Instructions

Instruction	Mnemonic Op Code	Machine Op Code	Operand Format	
			Explicit	Implicit
Add Long	AD	6A	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Add Long	ADR	2A	R1,R2	
Add Short	AE	7A	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Add Short	AER	3A	R1,R2	
Compare Long	CD	69	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Compare Long	CDR	29	R1,R2	
Compare Short	CE	79	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Compare Short	CER	39	R1,R2	
Divide Long	DD	6D	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Divide Long	DDR	2D	R1,R2	
Divide Short	DE	7D	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Divide Short	DER	3D	R1,R2	
Load Complement Long	LCDR	23	R1,R2	
Load Complement Short	LCER	33	R1,R2	
Load Long	LD	68	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load Long	LDR	28	R1,R2	
Load Negative Long	LNDR	21	R1,R2	
Load Negative Short	LNER	31	R1,R2	
Load Positive Long	LPDR	20	R1,R2	
Load Positive Short	LPER	30	R1,R2	
Load Short	LE	78	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Load Short	LER	38	R1,R2	
Multiply Long	MD	6C	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Multiply Long	MDR	2C	R1,R2	
Multiply Short	ME	7C	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Multiply Short	MER	3C	R1,R2	
Store Long	STD	60	R1,D2(X2,B2)	R1,S2(X2) or R1,S2
Store Short	STE	70	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Long	SD	6B	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Long	SDR	2B	R1,R2	
Subtract Short	SE	7B	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Short	SER	3B	R1,R2	

System/370 Instructions

Instruction	Mnemonic Op Code	Machine Op Code	Operand Format	
			Explicit	Implicit
Compare Logical Long	CLCL	0F	R1,R2	
Insert Characters Under Mask	ICM	BF	R1,M3,D2(B2)	R1,M3,S2
Move Long	MVCL	0E	R1,R2	
Shift and Round Decimal	SRP	F0	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
Store Characters Under Mask	STCM	BE	R1,M3,D2(B2)	R1,M3,S2

Appendix D.

Assembler Command Summary

Commands for controlling the assembly listing and object deck:

Assembler Statement	Comments
`EJECT`	Causes skip to new page.
`PRINT ON,NOGEN,NODATA`	Operands cause source statements to print (ON or OFF), macro generated statements to be printed (GEN) or skipped (NOGEN), and only 8 bytes (NODATA) or all (DATA) of defined data to print.
`PUNCH '  INCLUDE'`	Data coded between apostrophes as operand will be punched in card, beginning in column 1, as part of object deck.
`REPRO`	Punches duplicate of following statement in object deck.
`SPACE 3`	Spaces assembly listing the number of lines coded as operand.

Commands that affect the assembly of the program:

Assembler statement			Comments
	COPY	BOOKNAME	Causes source code book from source-statement library to be inserted in program. The operand gives the book name.
	CNOP	2,4	Used for instruction alignment. Operand can be 0,4; 2,4; 0,8; 2,8; 4,8; 6,8.
PROGNAME	CSECT		Restores location counter after DSECT definition. Label gives program name as given in START instruction.
DSNAME	DSECT		Signals start of dummy section.
	END	BEGIN	Signals end of source program. Operand names program entry point, usually BALR to load base register.
	ENTRY	ENAME	Identifies entry point. Operand is label of instruction that is to be an entry point.
NAME1	EQU	NAME2	Symbol NAME1 is assigned same address as length as NAME2.
	EXTRN	EXNAME	Tells assembler that EXNAME is defined outside this program. Address should be resolved by linkage editor.
	LTORG		Causes literals to be defined at current location.
	ORG	*-80	Alters location counter during assembly. Operand can be symbol or expression.
PROGNAME	START	0	Defines start of program. Operand becomes initial value of location counter.
	USING	*,3	Defines second operand as base register. First operand can be DSECT name to assign base register for DSECT.

Commands for macro definition and conditional assembly:

Assembler Statement			Comments
.SEQSYM	AGO	.SYM	Unconditional branch to sequence symbol coded as operand.
.SEQ2	AIF	(logical expression).S3	Conditional branch. If the logical expression is true, branches to sequence symbol that follows.
.SYM2	ANOP		A no-operation; allows definition of sequence symbol.
	GBLA	SYM1,SYM2,...	Declare arithmetic global SET symbols.
	GBLB	SYM3,SYM4,...	Declare binary global SET symbols.
	GBLC	SYMA,SYMB,...	Declare character global SET symbols.
	LCLA	SYM5,SYM6,...	Declare arithmetic local SET symbols.
	LCLB	SYM7,SYM8,...	Declare binary local SET symbols.
	LCLC	SYMC,SYMD,...	Declare character local SET symbols.
	MACRO		Signal start of macro definition.
	MEND		Signal end of macro definition.
	MEXIT		Terminate macro expansion.
	MNOTE	2,'MESSAGE'	Define error message within macro definition.
SYM1	SETA	20	Assign operand value to arithmetic SET symbol coded as label. Operand can be any arithmetic expression.
SYM3	SETB	1	Assign value (0 or 1) to binary SET symbol coded as label. Operand can also be a logical expression. If the expression is true, value 1 is assigned; if false, value 0 is assigned.
SYMC	SETC	'NO'	Assign value to character SET symbol. Operand can be any character expression.

Appendix E.

System/370 Instructions

The System/370 offers 29 more instructions than the System/360. Although most of them are used only in operating system programs to control the System/370's added features, these 5 can be useful in business programming:

OP CODE	INSTRUCTION
MVCL	Move Long
CLCL	Compare Logical Long
SRP	Shift and Round Decimal
ICM	Insert Characters Under Mask
STCM	Store Characters Under Mask

MVCL The move-long instruction overcomes the 256-byte limit of the normal move-character (MVC) instruction by allowing any number of bytes to be moved. It is coded in register-to-register format as in this example:

```
MVCL  6,10
```

Each operand--which must be an even-numbered register--represents an even-odd register pair. The even register of the operand-1 pair (register 6 in the example above) must contain the address of the first byte of the receiving field. The odd register of the operand-1 pair (register 7 in the example) must contain the length of the receiving field as a binary value. The even register of the operand-2 pair (register 10 above) must contain the address of the sending field. The odd register of the operand-2 pair must contain two values: (1) the length of the sending field which must be stored in the second, third, and fourth bytes and (2) a pad character which must be stored in the first, or leftmost, byte. When the MVCL is executed, the data from the sending field is moved to the receiving field--byte by byte from left to right. If the receiving field is shorter than or as long as the sending field, it executes just like an MVC instruction without the 256-byte limit. If the receiving field is longer than the sending field, the pad character is used to fill the extra bytes of the receiving field.

Figure E-1 is an example of the simple situation in which the receiving field and sending field are the same length.

```
.
.
LA    8,RECVFLD
LA    9,800
LA    4,SENDFLD
LA    5,800
MVCL  8,4
.
.
```

FIGURE E-1 Sample Use of the Move-Long Instruction

The load-address instruction, as used in figure E-1, offers a handy way to get the field addresses into the even registers and, by coding an explicit address, to get the length values into the odd registers. The pad character in this case would be hex 00 since it will be loaded into the leftmost byte of register 5 by the LA instruction. Because, in this case, the fields are the same length the pad character won't be used.

By intentionally setting the length of the sending operand to zero, you can completely fill an area of storage with the pad character. As shown in figure E-2 for example, you could initialize a 2000-byte table area to blanks before placing other data into it. When the sending-field length is zero like this, the sending-field address is not used; this means any value can be in the register. The pad character, hex 40, is used to fill all of the bytes of the receiving field.

CLCL The compare-logical-long instruction is also coded in register-to-register format and the register content requirements are the same as for the move-long. The shorter of the two fields is considered to be extended with the pad character stored in the leftmost byte of the odd register of the operand-2 pair. The instruction compares the bytes of the fields one at a time, from left to right, bumping the address registers by one and decrementing the length values by one as each byte is compared. If all the bytes are equal, the condition code is set to indicate an equal compare. If an unequal pair of bytes is found, the condition code is set to indicate operand A is low or high, and the contents of the address and length registers are left at their current settings. At this time the address registers contain the addresses of the mismatched bytes, and the length registers contain the number of bytes remaining to be compared. These addresses can then be used to locate the unequal bytes.

```
         .
         .
         LA    10,TABLE
         LA    11,2000
         LM    6,7,BLANK
         MVCL  10,6
         .
         .
BLANK    DC    X'00000000'
         DC    X'40000000'
         .
         .
```

FIGURE E-2 Clearing a Table Area to Blanks with the Move-Long Instruction

```
         .
         .
         LA    4,INTAP1
         LA    5,2000
         LA    8,INTAP2
         LA    9,2000
COMP     CLCL  4,8
         BE    NEXT
         MVC   ERRBYTE1(1),0(4)
         MVC   ERRBYTE2(1),0(8)
         BAL   11,PRTSUB
         LA    4,1(4)
         LA    8,1(8)
         S     5,=F'1'
         S     9,=F'1'
         B     COMP
         .
         .
```

FIGURE E-3 Sample Use of the Compare-Logical-Long Instruction

In figure E-3, two 2000-byte tape blocks are compared. If they are equal, a branch is made to read two new blocks. If a mismatched pair of bytes is found, each of the unequal bytes is moved to a print area using the addresses left in the registers by the compare instruction. After a branch-and-link to a print subroutine, the addresses in the even registers are bumped by one so that they point to the next byte beyond the unequal one in each field. Also, the remaining length of each field is decremented by one. Then the compare-long

is executed again and comparison of the two blocks continues. If another unequal pair of bytes is found, they are printed, the addresses and lengths are adjusted, and the CLCL is executed again. This procedure continues until all of the bytes of the two blocks have been compared and the branch to EQUAL is taken.

If the lengths of the fields to be compared are not equal, the shorter of the two is considered to be extended with the pad character. Figure E-4 shows how this feature of CLCL can be used to find the first nonblank character in an area. The odd register of the operand-2 pair (register 11 in figure E-4) is loaded with the pad character and a zero length value. Each byte of the first operand is then compared with the pad character. When the first character that doesn't match the pad character is found, the instruction will stop and the address of the nonblank character will be left in the even register of operand 1.

SRP In many business programming problems, you must deal with packed-decimal fields. Shifting and rounding these fields in System/360 programs requires use of the add-decimal (AP), move-numeric (MVN), and move-with-offset (MVO) instructions. At best, this rounding process is cumbersome. To make this operation simpler, the System/370 offers the shift-and-round-decimal instruction. With it, you can shift left or right any number of digit positions and round when required as can be seen in this example:

```
SRP    FIELD,3,0
```

The first operand specifies the packed-decimal field, operand two indicates the shift direction and the number of digit positions, and operand three specifies the rounding factor.

The specification of operand two is the tricky part of using this instruction. It is actually interpreted as a storage address and translated into a 32-bit binary number. When the instruction is executed, bits 0 through 25 are ignored and bits 26 through 31 are treated as a signed binary value. Bit 26 becomes the sign bit and bits 27-31 are assigned binary place values: 16, 8, 4, 2, and 1. Positive values indicate left shifts and negative values indicate right shifts. The easiest way to generate the proper bit pattern for operand two is to code one of the absolute address values shown in figure E-5.

```
          .
          .
NEWCARD   LM     8,11,START
          CLCL   8,10
          BE     NODATA
          ST     8,ADDR1
          .
          .
          .
ADDR1     DS     F
START     DC     A(CARD)
          DC     A(80)
          DC     A(0)
          DC     X'40000000'
          .
          .
```

FIGURE E-4 Searching for Blanks with the Compare-Logical-Long Instruction

Number of Shift-Left Positions	Operand-2 Code
0	0
1	1
2	2
3	3
4	4
5	5
Number of Shift-Right Positions	**Operand-2 Code**
0	64
1	63
2	62
3	61
4	60
5	59

FIGURE E-5 Operand-2 Codes for Shift-and-Round-Decimal Instruction

Instruction	Mask	Reg Before	Storage	Reg After
ICM 4,4,STOR	0100	00000000	FFFFFFFF	00FF0000
ICM 7,15,STOR	1111	00000000	804B36AD	804B36AD
ICM 10,10,STOR	1010	FFA33629	8000FFFF	80A30029
ICM 12,3,STOR	0011	00000000	F1F2F3F4	0000F1F2
ICM 11,8,PAD	1000	00007B36	40404040	40007B36

FIGURE E-6 Examples of Insert-Characters-Under-Mask Instruction

For example, to shift left 2 digit positions, you would code:

```
SRP    FIELD,2,0
```

To shift right 3 digit positions, you would code:

```
SRP    FIELD,61,5
```

This example also illustrates the most common rounding factor, 5, which is added to the digit position to the right of the one being rounded to. The rounding factor is applied only to right shifts and must be in the range 0 through 9. When a field is shifted left, the rightmost digit positions are padded with zeros, so rounding doesn't apply.

To illustrate, suppose you have multiplied an account-balance field with two decimal places by an interest-rate field that also has two decimal places. The resulting interest amount has four decimal places that you would probably want to round to two places. If the result field is five bytes long and is named INTAMT, this instruction would shift the result right two places and round;

```
SRP    INTAMT,62,5
```

If INTAMT contains hex 000362067C prior to rounding, the instruction execution can be shown as follows:

```
INTAMT Before        INTAMT After
  000362067C           000003621C
```

The rounding factor (5) is added to the last digit shifted out (6) to round the rightmost remaining digit from 0 to 1.

ICM The insert-characters-under-mask instruction operates much like a load instruction except that it allows you to insert data bytes into selected bytes of the register instead of always filling all four bytes:

```
ICM    6,12,SOURCE
```

When executed, bytes from the third operand are loaded into the register specified as the first operand. The number specified as operand two is translated into a four bit binary value to form a mask that governs which of the register bytes are filled. Each bit of the mask corresponds to a byte of the register: the first mask bit to the first byte of the register, the second mask bit to the second register byte, and so on. If a mask bit is on, the corresponding byte of the register is loaded. If a mask bit is off, the corresponding register byte is left unchanged. The selected register bytes are loaded from consecutive bytes of storage, beginning with the one addressed by operand three. Figure E-6 shows several examples of ICM and illustrates the before and after contents of the register. The last example shows how ICM might be used to insert a pad character into the leftmost byte of the odd register of the second operand for a move-long instruction.

STCM Store-characters-under-mask is the converse of insert-characters-under-mask. Bytes selected from the register coded as the first operand are stored in consecutive bytes of storage beginning at the address of the third operand. The bytes to be stored are selected based on the mask given as the second operand. Figure E-7 shows several examples.

Instruction	Mask	Reg Contents	Stored Bytes
STCM 6,15,FLD	1111	00006A4C	00006A4C
STCM 12,4,FLD	0100	00367FA2	36
STCM 4,10,FLD	1010	80003D68	803D

FIGURE E-7 Examples of Store-Characters-Under-Mask Instruction

Appendix F.

Problems for Computer Lab

Subset Problems

1. 80-80 Listing with Totals (Chapter 3)
2. Sales Listing (Chapter 3)
3. Grade Reporting (Chapter 3)
4. Multiple Card Definitions (Chapter 5)
5. Explicit Operand and DSECT Use (Chapter 5)

Part 3 Problems

6. Binary Arithmetic (Chapter 6)
7. Single-Level Table (Chapter 7, Topic 1)
8. Multilevel Table (Chapter 7, Topic 2)
9. Translation (Chapter 8)
10. OI, NI, and EDMK (Chapter 8)
11. Free-Form Input (Chapter 8)
12. Writing a Subprogram (Chapter 9)
13. Writing a Macro Definition (Chapter 10)

Tape and Disk Problems

14. Tape Creation (Chapter 13, Topic 1)
15. Tape Update (Chapter 13, Topic 1)
16. Sequential Disk Creation (Chapter 15, Topic 1)
17. Sequential Disk Update (Chapter 15, Topic 1)
18. ISAM Creation (Chapter 15, Topic 2)
19. ISAM Sequential Processing (Chapter 15, Topic 2)
20. ISAM Random Update (Chapter 15, Topic 2)
21. ISAM File Maintenance (Chapter 15, Topic 2)

NOTE: The minimum text requirements for each problem are given in parentheses following the problem name. For problems 6-21, the subset (chapters 1-5) plus the chapter or topic in the parentheses are the minimum requirements.

SUBSET PROBLEMS

1. 80-80 Listing with Totals

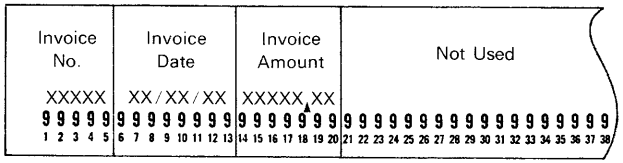
Write a program to perform an 80-80 listing of any input deck. Print 50 cards per page, count the number of cards listed, and print a count line after spacing two lines below the last card listed. The count line should look like this:

XXXX INPUT CARDS LISTED

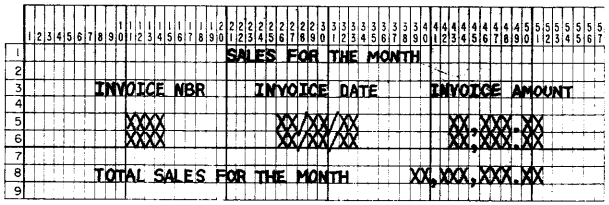
2. Sales Listing

Write a program that lists the total sales for a month. For each sale an invoice card has been punched. List the individual invoices with headings and 50 lines on each page. Accumulate the total sales for the month and at the end of the report print a final total line showing the total sales for the month.

Input



Output



Test Data

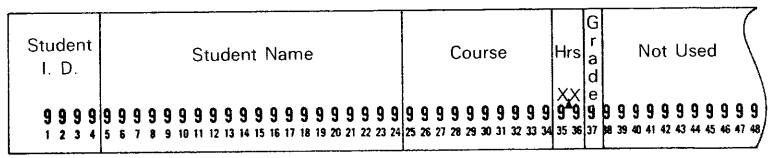
Invoice no. (Cols. 1–5)	Invoice Date (Cols. 6–13)	Invoice Amount (Cols. 14–20)	Invoice no. (Cols. 1–5)	Invoice Date (Cols. 6–13)	Invoice Amount (Cols. 14–20)
30564	04/02/74	0034600	30586	04/08/74	0234818
30565	04/02/74	0076892	30587	04/08/74	0185260
30566	04/02/74	0230680	30588	04/09/74	0094682
30567	04/03/74	0009243	30589	04/09/74	0021492
30568	04/03/74	0173086	30590	04/09/74	0071809
30570	04/03/74	0068224	30591	04/09/74	0018394
30571	04/03/74	0008713	30592	04/09/74	0065332
30572	04/03/74	0200816	30593	04/10/74	0193483
30573	04/04/74	0011433	30594	04/10/74	0041226
30575	04/04/74	0080718	30595	04/10/74	0068219
30576	04/04/74	0004429	30597	04/10/74	0034370
30577	04/04/74	0340900	30599	04/11/74	0113620
30578	04/04/74	0081312	30600	04/11/74	0037384
30579	04/05/74	0066245	30601	04/11/74	0071941
30580	04/05/74	0001312	30602	04/11/74	0018611
30581	04/05/74	0041283	30603	04/12/74	0031693
30583	04/05/74	0076277	30604	04/12/74	0092716
30585	04/08/74	0063340			

3. Grade Reporting

Write a program to calculate and print the Grade Point Averages of students. Each input card has the credit hours and grade for one course that a student is taking. The course cards for each student are grouped together, but there can be a variable number per student. Compare on student I.D. to find the break between students.

Unless you want to use your own school's Grade Point Average system, use a 4.0 system in which A=4.0, B=3.0, C=2.0, D=1.0, and F=0.0. The G.P.A. is calculated by multiplying the number of credit hours for each course by the value of the grade received, summing the totals, and dividing the sum by the total number of credit hours taken by the student. Round the G.P.A. to one decimal place for printing on the report and skip to a new page after each group of four students.

Input



Output

This is a sample of the desired output. Before writing the program, make a print chart that approximates this format.

```
STUDENT I.D.    STUDENT NAME              COURSE          CREDIT HOURS    GRADE

   5362         WILLIAMS, JOS.            ECON 101            3.0           B
                                          PSYCH 101           3.0           C
                                          MATH 201            3.0           B
                                          PHYSICS 100         2.0           D
                                          ENG 151             3.0           B
                                          PHYS ED 100         2.0           A

                                          GRADE POINT AVERAGE       2.6

   5378         WILSON, BETH              PSYCH 301           3.0           A
                                          SOC SCI 320         2.0           A
```

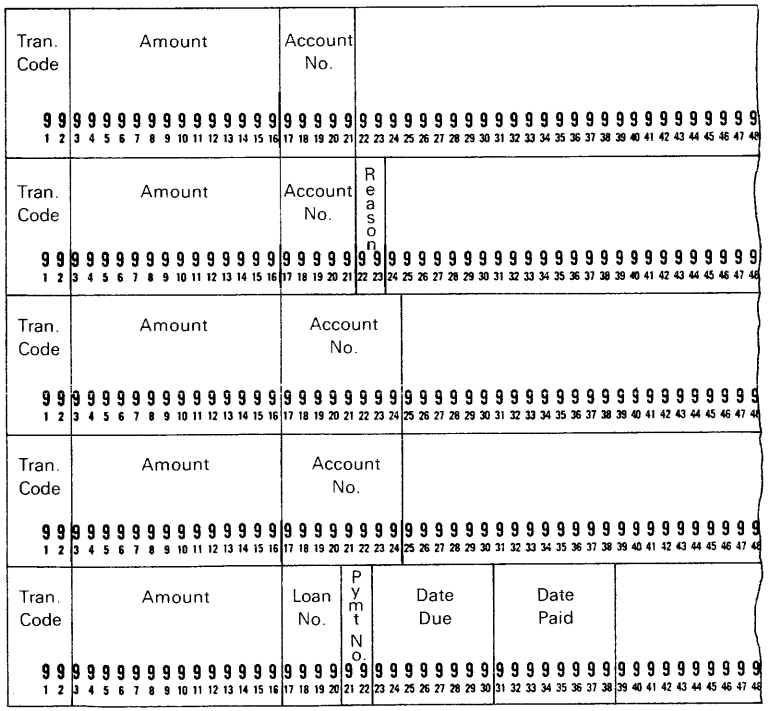
Test Data

Student I.D. (1–4)	Student Name (5–24)	Course (25–34)	Credit Hours (35–36)	Grade (37)	Student I.D. (1–4)	Student Name (5–24)	Course (25–34)	Credit Hours (35–36)	Grade (37)
1342	APPLEBOTTOM, JACK	ACCT 331	30	B	8374	MENDES, RAFAEL	EE 312	20	A
1342	APPLEBOTTOM, JACK	FRCH 103	30	C	8374	MENDES, RAFAEL	PSYCH 101	20	C
1342	APPLEBOTTOM, JACK	MATH 320	20	B	6204	TUFER, ALFRED	ENG 381	30	C
1342	APPLEBOTTOM, JACK	MATH 400	30	A	6204	TUFER, ALFRED	GOVT 340	20	C
1342	APPLEBOTTOM, JACK	SOC SCI 140	20	C	6204	TUFER, ALFRED	HIST 200	20	D
3287	FRANKLIN, MARGARET	ART 230	10	B	6204	TUFER, ALFRED	SOC SCI 120	20	B
3287	FRANKLIN, MARGARET	ENG 381	30	A	6204	TUFER, ALFRED	POL SCI 301	30	A
3287	FRANKLIN, MARGARET	ENG 460	10	B	6204	TUFER, ALFRED	SPAN 101	30	C
3287	FRANKLIN, MARGARET	GOVT 361	30	C	4044	ZWICKI, FRANCIS	ART 260	10	C
3287	FRANKLIN, MARGARET	HIST 240	30	A	4044	ZWICKI, FRANCIS	ENG 320	30	B
3287	FRANKLIN, MARGARET	POL SCI 210	30	A	4044	ZWICKI, FRANCIS	ENG 460	10	A
8374	MENDES, RAFAEL	ENG 143	30	C	4044	ZWICKI, FRANCIS	ITAL 201	30	C
8374	MENDES, RAFAEL	EE 101	40	A	4044	ZWICKI, FRANCIS	PSYCH 101	30	D
8374	MENDES, RAFAEL	MATH 120	30	B	4044	ZWICKI, FRANCIS	POL SCI 301	30	C

4. Multiple-Card Definitions

The card layouts that follow represent several transaction cards that might be used in a bank or savings and loan company. Write a program that validates the various transactions by checking the rightmost column of each numeric field for a blank. Print a simple 80-80 listing of each transaction with the message ERROR appearing to the right of transactions that have one or more fields blank. Use the ORG statement to align the multiple definitions of the card area. You can use the transaction code to determine which checking routine of your program to use. If a transaction code is invalid, print INVALID CODE to the right of the data line. (It will probably be easiest to use an explicit operand to address the rightmost byte of each field to be checked.)

Input



Test Data

```
06000000000020000541 39
06000000000156785414
07000000000024145426802
07              83195482301
07000000000007816      03
16000000000250005600428
16                5600428
  000000000250005600428
17000000000057600560018 2
22000000000008519134612 05/01/7405/02/74
22000000000005960     1005/01/7404/30/74
22                 20660405/01/7404/18/74
```

5. Explicit Operands and DSECT

Write a program to print an inventory report from a deck of inventory cards. As indicated by the card layout, each item may be stored in one to ten locations. To refer to the ten card and print line quantity fields, code a processing loop using either explicit operands with substitute base registers or dummy section labels.

Input

Card Columns	Data
1–6	Item Number
7–20	Description
21–26	Location #1, 21–22 = Loc. I.D., 23–26 = Qty
27–32	Location #2, 27–28 = Loc. I.D., 29–32 = Qty
33–38	Location #3, 33–34 = Loc. I.D., 35–38 = Qty

and so on, with the I.D. and quantity for location #10 in columns 75–80

Output

Print a report having a format similar to the one below. Use a print chart form to plan the exact spacing of each field.

```
 PART                        LOC #1      LOC #2      LOC #3           LOC #10      TOTAL
  NO.    DESCRIPTION         ID  QTY     ID  QTY     ID  QTY ..... ID  QTY      ON HAND
111111   BOLT                Z4   40     Z5   63     H8  700       E6  275        2378
222222   WIDGET              L2   57     L3   43                                   100
```

Test Data

1 6	7 20	21 26	27 32	33 38	39 44	45 50	51 56	57 62	63 68	69 74	75 80
003044	HANDLE	B 70040	D 40028	M60126							
003810	HANDLE PIN	L 91386	F 20624								
004462	BOLT, 1/4	C 70320	X 40213	L 30040	K 20418	N 80083	E 60092	T 30026	R 70118	H 10202	J 40070
005071	AXLE PIN	T 60846	A 70120	L 40083	G 10078						
005302	WHEEL	P 31000	S 21000	Y 61000							
006657	BLADE	Q 80500	V 90500	V 80500	Z 20500						
008129	REAR BRACE	D 30126	D 50150	D 60150	M10300	M20300	M30300				
008844	NUT, 1/4	F 61200	F 71200								
011263	MOWER BODY	N 20060	N 30060	U 20024	W40036	A 20016	C 60048	C 80060	C 90060		
012768	ROLLER	H 70100	H 80100	H 90100	G 20100	G 30100					

PROBLEMS FOR PART 3

6. Binary Arithmetic

Write a program to calculate the win/loss and point statistics for football teams. Use fixed-point binary arithmetic to calculate the total number of games won, lost, and tied, the win percentage, the total and average points scored per game, and the total and average points allowed. The input cards give the game scores for each team.

Input

Card Columns	Data
1–20	Team Name
21–25	Year Season (XX–XX)
26–27	Game 1 Points Scored
28–29	Game 1 Points Allowed
30–31	Game 2 Points Scored
32–33	Game 2 Points Allowed
.....	.....
.....	.....
70–71	Game 12 Points Scored
72–73	Game 12 Points Allowed

Output

Before writing this program, design a print chart with this approximate format:

TEAM	YEAR	WON	LOST	TIED	WIN PCNT	TOTAL POINTS SCORED	AVG GAME SCORED	TOTAL POINTS ALLOWED	AVG GAME ALLOWED
PANTHERS	72-73	08	04	00	66	276	23	204	17
TIGERS	72-73	05	05	02	50	216	18	228	19

Test Data

```
Col.                Col.
1                   21
TIGERS              73-741614301612121821142312181310112010141210281215l2
PANTHERS            73-741416120709001024213406132406161914101714171432091
PHANTOMS            73-742310071212120818182607171815182316201012301420l0
BANSHEES            73-741023171212122118101811192107101220161417122814l7
ASTRALS             73-743621121620102410231419111518231807073208141712l5
ATOMICS             73-742136270822160818342118120721121007072610143009l2
DEMONS              73-741000082710200620261813061013191709000832122110l0
CHEETAS             73-740010163016222006181017070624201100091026211207l4
```

7. Table Handling

Write a program that will print a sales report from sales transaction input cards that contain the item number and the quantity sold. These input data cards are preceded by five cards that contain the prices for each valid item. Your program must use the item numbers and prices from the first five cards to build a table in storage that is then used to look up the item price when the sales transaction cards are being processed. If an item from a sales transaction card can't be found in the table, print an error message that indicates the invalid item number.

Input

Item-Price Cards:

Columns	Data
1–3	Item Number
4–7	Unit Price (XX XX)
8–10	Item Number
11–14	Unit Price
15–17	Item Number
18–21	Unit Price
.....	....
.....	....
64–66	Item Number
67–70	Unit Price

Each card contains the number and price of ten items.

Sales Transaction Cards:

Columns	Data
1–3	Item Number
4–7	Quantity Sold

Output

The report should have this general format:

```
ITEM      UNIT      QTY       SALES
NO.       PRICE     SOLD      AMOUNT
245       16.20        5        81.00
738        0.75     2000      1500.00
```

Test Data

First five cards:

```
1                                                                    70
0120125017025002300390240050026100003012500311725070125007113390721442
0770005080125010750001110077158055516162501922500201004021306662770111
2970777305001034115003425750343650034722503510035356088835902223800999
3821000423008847100155281750530200054700305901200612033374314007471600
7910099793200079422007960020797002580224008090444812260082228008433000
```

Item cards:

Cols. 1–3	4–7	Cols. 1–3	4–7
026	0010	815	0001
343	0008	080	0012
427	0006	111	0123
071	0012	351	1234
796	0014	351	2000
843	1000	613	4000
012	0100	812	6000
809	0010	077	8000

8. Multilevel Table Handling

Write a program to print a report of the estimated shipping costs for orders identified by shipment cards. The shipping rate per pound depends on the total weight of the order and on the district to which the order is to be shipped. The table that follows shows these cost-per-pound rates as dollar figures: for instance, .030 equals three cents per pound. You must define this table in your program and use it to look up the proper shipping rate to use in calculating the estimated shipping cost for each order.

Order Weight	Shipping District				
	1	2	3	4	5
0–50	.030	.032	.034	.036	.038
51–100	.028	.029	.029	.030	.031
101–499	.024	.025	.026	.027	.028
500–999	.020	.022	.023	.025	.026
1000–1999	.014	.015	.016	.017	.018
2000 & over	.010	.011	.012	.013	.014

Shipping Rate Per Pound

Input

Card Columns	Data
1–4	Order Number
5–9	Order Weight
10	Shipping District

Output

Print a report similar to this one:

```
ORDR      ORDR      SHIP      SHIP      ESTIMATED
NO.       WGT       DIST      RATE        COST

3445       632         3      .023         14.54
3452      1236         1      .014         17.30
```

Test Data

```
4728000363
4729012161
4730001005
4731027341
4732004991
4733000514
4734340832
```

9. Translation

Write a program to translate some System/360 zoned-decimal data to a format for another computer. Construct a translate table so that the EBCDIC digits 0 through 9 are translated to the bit configurations shown in the table below. Translate the twelve-byte field in columns 1–12 of each input card and print the new data in a simple list.

EBCDIC DIGIT	TRANSLATED CODE
0	X'40'
1	X'C7'
2	X'E3'
3	X'C1'
4	X'D7'
5	X'E8'
6	X'C8'
7	X'E4'
8	X'C9'
9	X'D9'

Test Data

```
Col.
1
000644935000
000547014200
000820981620
```

10. OI, NI, and EDMK Instructions

Write a program that will print a trial payroll. The input cards contain the name, pay rate, and hours worked by day for each man. In addition to the normal pay of hours times pay rate, any man who worked on day 1 (Sunday) or day 7 (Saturday) should be paid a bonus of $10.00. If he worked both days, he should be paid a bonus of $25.00. Use the or-immediate instruction to set program switch bits to indicate either or both days worked so that the bonus can be added to the pay after multiplying hours times rate. Use the and-immediate instruction to turn the switches off. When the pay has been calculated, print a trial check register using the edit-and-mark instruction to put a dollar sign right next to the first digit of the amount.

Input

Card Columns	Data
1–20	Name
21–24	Pay Rate (XX XX)
25–27	Hours Worked Day 1 (XX X)
28–30	Hours Worked Day 2
31–33	Hours Worked Day 3
34–36	Hours Worked Day 4
37–39	Hours Worked Day 5
40–41	Hours Worked Day 6
42–44	Hours Worked Day 7

Output

```
S.K. GRATZ                    $328.20
D.O. FARNSWORTH                $84.92
G.K. MASMON                   $237.64
```

Test Data

Cols. 1	21
D.T. ALLSTON	044000008008008008008000
K.J. CHAMPION	028000008008008008504010
A.S. DISTENT	031204408008008008008000
L.T. MOONEY	029312210600008008008003
B.U. TRUMPT	040800000006300003700000

11. Free-Form Input

In some application areas (banking, for example) data is often entered directly into the computer through video display units or other types of terminals. This input is usually entered in *free form*—that is, the lengths of the data fields are not fixed. Instead, a field-separator character, like a comma, is used to indicate the end of one field and the start of the next. Write a program that will print free-form data cards in fixed-length-field format. Of the several ways to do this, one is to move a single byte at a time into the output area, comparing each byte with the separator character to determine when a field ends. Another way is to use the translate-and-test instruction to scan the input area for the separator character, calculate the length of the data by subtracting the address of the beginning of the field from the address of the separator, and insert the calculated length in the proper byte of a move instruction.

Input

(Name,Account Number,Amount)

HANSEN,4632,5632
MURPHY,70520,7460
BRIZOWSKI,824,21873
JAX,3658,422
CJONACKI,46,150000

Note: The amount field is assumed to have two decimals.

Output

Print Position	Data
1–20	Name
30–34	Account Number
40–47	Amount (XXXXX.XX)

12. Writing a Subprogram

Addresses are part of many of the records that are maintained in data processing installations. Customer records often have both billing and shipping addresses; vendor (supplier) records may have separate ordering and payment addresses; and employee records may have a next-of-kin address as well as the employee address. To conserve tape or disk space, the state portion of these addresses is often stored as a two-character code. Then, in all of the programs that print these addresses, the two character code must be expanded to the full state name.

Write a subprogram that converts a 2-character state code to a 20-character state name. Make the entry point name CONVST. This subprogram should accept two addresses from the calling program: (1) the address of the 2-byte state code field, and (2) the address of the 20-byte field in which the subprogram should place the state name. Assume that the standard postal codes shown in the table below are used.

Also shown below is a main program that will CALL the subprogram. It prints a mailing-address list from input cards. To test the subprogram, place the subprogram source code in the indicated position and run the job.

Test Data

1	21	41	61	63
HOMEWOOD LUMBER	6238 HIGH PASS RD.	CRESTED BUTTE	MT	59902
WESTON HARDWARE	46 ASPEN WAY	KETCHUM	ID	83340
REDWOOD SUPPLIES	847 HIWAY 88	JACKSON	CA	95642
QUADE LUMBER	2776 PIONEER DR.	EUGENE	OR	97405
THOMPSON TREE FARM	87 OLYMPIC LANE	ACME	WA	98220
TETON LUMBER	8724 YELLSTONE WAY	JACKSON	WY	83025
WESTERN SUPPLY	16404 PIKE AVE.	DURANGO	CO	81301
MULLEN LUMBER	63 WHEELER AVE.	RATON	NM	87740
MINING SUPPLIES	44 GOLD RUN	PARK CITY	UT	84064
SUNRISE LUMBER	1001 EAST RIDGE	KINGMAN	AZ	86401

State Table

CODE	STATE NAME	CODE	STATE NAME
AL	Alabama	MT	Montana
AK	Alaska	NE	Nebraska
AZ	Arizona	NV	Nevada
AR	Arkansas	NH	New Hampshire
CA	California	NJ	New Jersey
CO	Colorado	NM	New Mexico
CT	Connecticut	NY	New York
DE	Delaware	NC	North Carolina
FL	Florida	ND	North Dakota
GA	Georgia	OH	Ohio
HI	Hawaii	OK	Oklahoma
ID	Idaho	OR	Oregon
IL	Illinois	PA	Pennsylvania
IN	Indiana	RI	Rhode Island
IA	Iowa	SC	South Carolina
KS	Kansas	SD	South Dakota
KY	Kentucky	TE	Tennessee
LA	Louisiana	TX	Texas
ME	Maine	UT	Utah
MD	Maryland	VT	Vermont
MA	Massachusetts	VA	Virginia
MI	Michigan	WA	Washington
MN	Minnesota	WV	West Virginia
MS	Mississippi	WI	Wisconsin
MO	Missouri	WY	Wyoming

BAL Main Program and JCL

```
// JOB (INSERT YOUR OWN JOBNAME)
// OPTION LINK,DUMP
// EXEC ASSEMBLY
MAIN      START 0
CARD      DTFCD DEVADDR=SYSIPT,IOAREA1=CRDIO,EOFADDR=EOFCRD
PRINT     DTFPR DEVADDR=SYSLST,IOAREA1=PRTIO,BLKSIZE=132
BEGIN     EALR  3,0
          USING *,3
          OPEN  CARD,PRINT
          LA    13,SAVAREA
READCARD  GET   CARD
          MVC   PRTNAME,CRDNAME
          MVC   PRTSTRT,CRDSTRT
          MVC   PRTCITY,CRDCITY
          CALL  CONVST,(CRDSTATE,PRTSTATE)
          MVC   PRTZIP,CRDZIP
          PUT   PRINT
          B     READCARD
EOFCRD    CLOSE CARD,PRINT
          EOJ
CRDIO     DS    0CL80
CRDNAME   DS    CL20
CRDSTRT   DS    CL20
CRDCITY   DS    CL20
CRDSTATE  DS    CL2
CRDZIP    DS    CL5
          DS    CL13
PRTIO     DS    0CL132
PRTNAME   DS    CL20
          DC    10C' '
PRTSTRT   DS    CL20
          DC    10C' '
PRTCITY   DS    CL20
          DC    10C' '
PRTSTATE  DS    CL20
          DC    10C' '
PRTZIP    DS    CL5
          DC    7C' '
SAVAREA   DS    18F
          END   BEGIN
/*
// EXEC ASSEMBLY
   (INSERT YOUR SUBPROGRAM SOURCE CODE HERE)
/*
// EXEC LNKEDT.
// EXEC
   (TEST DATA)
/*
/&
```

13. Writing a Macro Definition

Write a definition for a macro that will generate instructions to check a numeric field for valid data. The generated instructions should check the rightmost byte of a field and change it to an EBCDIC zero if it is other than a valid zoned-decimal digit. To use your macro, a programmer should code an operation code NUMED and specify two operands: (1) the name of the field to be edited, and (2) its length. Here's an example:

```
EDIT     NUMED FIELDA,6
```

In your first attempts to define this macro, assume that the macro operands are always coded correctly. Then, as a second phase, add some checks for valid operands to your definition. To test your macro definition, alter the program you coded for problem 4 so that it uses this macro to validate the various transaction cards. At first, just assemble the problem 4 program and look at the generated statements to check your macro definition. Then, when you think the generated statements are correct, execute the program.

PROBLEMS FOR TAPE AND DISK

These problems require you to write a set of programs to create and update a file of customer master records. Problems 14 and 15 deal with the file on tape, problems 16 and 17 create and update the file as a sequential disk file, and problems 18, 19, 20, and 21 deal with the records as an indexed sequential file. All of the file creation programs (problems 14, 16, and 18) require the input cards listed in figure F-1 and should create the master records in the format shown in figure F-2. The update programs (problems 15, 17, and 19) each require as input the sales transaction cards shown in figure F-3, and in addition to updating the master file, should print a sales-activity report similar to the one shown in figure F-4. You can design your own print chart for this report so you will determine the exact spacing of the report. For any update or file maintenance program, the report should give the processing date, which can be obtained by using the COMRG macro described in chapter 11.

14. Tape Creation

Write a program to create a tape file of customer master records from the input cards shown in figure F-1. The format for the output tape records is shown in figure F-2. Don't forget to initialize the numeric fields for month-to-date and year-to-date sales to zero. You should also set the bytes reserved for expansion to blanks. To check the operation of your program, modify the tape-to-printer utility shown in chapter 16 to agree with your record-length and blocking factors, and execute it as the last step of your test job. You can then check the tape records to see if your creation program executed properly.

15. Tape Update

Use the sales transaction cards of figure F-3 as input to update the month-to-date and year-to-date sales fields in the tape that you created in problem 14. During the update, a sales activity report similar to the one in figure F-4 should be printed. If you find an invalid sales transaction (for example, a customer number does not match one in the file), print a line with blank name, month-to-date, and year-to-date fields followed by the message NO CUSTOMER RCD. Again, as the last step of the test run use the tape-to-printer utility program to print the updated tape. Make sure that all the records are present in the new file and that the proper records were updated.

16. Sequential-Disk-File Creation

Write a program that creates the customer master records shown in figure F-2 as a sequential-disk file. Use the cards of figure F-1 as input. Shown below are the job-control statements for a disk-to-printer utility program that you can use to print your disk records and to check the operation of your program. Make the execution of the disk-to-printer

utility a job step that follows the execution of your program. Be sure that the assignment of the input file, SYS004, and the DLBL information agree with the information used during file creation. If you blocked the sequential-disk records, change the blocksize parameter in the utility-modifier statement to agree:

```
A=(recsize,blksize)
```

Similarly, if you use a 3330, change the E parameter to agree.

Disk-to-Printer Utility

```
// ASSGN SYS004,X'???'
// ASSGN SYS005,X'00E'
// DLBL UIN,'file-ID'
// EXTENT SYS004
// EXEC DKPR
// UDP TL,FF,A=(100,100),B=(132),S1,E=(2314)
// END
/&
```

17. Sequential Disk Update

Write a program that will update, in place, the sequential-disk file of customer master records that you created in problem 16. Use the sales transaction cards of figure F-3 and print a sales-activity report as shown in figure F-4. If you find an invalid sales transaction (customer number does not match one in the master file), print a line with blank name, month-to-date, and year-to-date fields and print the message NO CUSTOMER RCD to the right of the line. As a final job step, you can again execute the disk-to-print utility program to print your updated file. You can then check the updated records.

18. ISAM File Creation

Write a program to create the customer master records in figure F-2 as an indexed sequential file. The input cards of figure F-1 contain the master file data. Customer number should be used as the key for the records and the last byte of the record should be reserved for a delete flag. The file will be very small so use only a cylinder index and allocate only two cylinders for prime data. Don't reserve any general overflow area, but allocate one track per cylinder for additions. If an input record is out of sequence or contains a duplicate key (one already written on the file), print an appropriate error message and ignore the record. If more than five duplicate records or out-of-sequence records are read, print an appropriate error message and cancel the program. Since the program for problem 19 prints the contents of this master file, you can use it to make sure this program created the file properly.

19. ISAM Sequential Processing

Write a program that will read the ISAM customer master file sequentially and print a report showing the month-to-date and year-to-date sales figures. The report should be similar to the sales-activity report of figure F-4, but without the invoice number, date, and amount fields. Assume that X'FF' will be used as the delete flag and omit deleted records from the report.

20. ISAM Random Update

Write a program to randomly update the indexed sequential customer master file. Use the sales transaction cards of figure F-3 as input, and print the sales activity report of figure F-4 as an output register. If no master record exists for an input transaction, put the transaction data into the print-line fields (put blanks in the name, month-to-date, and year-to-date fields) and print the line with the message NO CUSTOMER RCD to the right of the data fields. You can use the program that you wrote for problem 19 to print the file and check the updated records.

21. ISAM File Maintenance

Write a program to maintain (add and delete records) the ISAM customer master file. The input transaction cards are just like the input cards for the original file-creation program, with the exception of a transaction code that is added in columns 79 and 80: AD for add and DL for delete. Print a register of the maintenance transactions including an indication of the result of each transaction. An example is shown below. Your program must include tests for duplicate key in the case of an add attempt, and no-record-found in the case of a delete attempt. If either condition is detected, an appropriate message should be printed and the card should be ignored. Again, execute the program you wrote for problem 19 as the final step of your test job. Use the report to make sure that the proper records were added and deleted.

Output

The maintenance register should be something like this:

```
CUST NO.     CUSTOMER NAME            REQUESTED MAINT      ACTION

 1024        NESBITT MFG.                  ADD             ADDED
 1832        WESTCO                        ADD             ADDED
 2472        ZUGG & CO.                    DEL             DELETED
 2630        B.A. KREBS CO.                DEL             NO RCD FND
 3114        CASSADY AND CODY              ADD             DUPLICATE
```

Test Data

Cust. No. (1–4)	Customer Name (5–24)	Street Address (25–44)	City (45–64)	State (65–66)	Zip (67–71)	Tran. Code (79–80)
1024	NESBITT MFG.	2730 BELMONT DRIVE	NEW HAVEN	CT	06516	AD
1832	WESTCO	8677 COTTON CANYON RD.	PROVO	UT	84118	AD
2472	ZUGG & CO.	1326 VAN NESS	IOWA CITY	IA	52240	DL
2630	B.A. KREBS CO.	17643 COLFAX	DENVER	CO	80228	DL
3114	CASSADY AND CODY	8776 SNOW VALLEY RD.	WHITEFISH	MT	59937	AD
3916	STONE, KEMP & CO.	867 MELLON WAY	PITTSBURGH	PA	15241	DL
4515	PERE & SONS	36 BALLARD AVE.	SEATTLE	WA	98166	AD

Index

The text of this book is set in 10 point Zenith with heads set in Galaxy Medium by Applied Typographic Systems of Mountain View, California.

The book was printed by Webcrafters, Inc. of Madison, Wisconsin.

Pat Rogondino, of Palo Alto, California, prepared the illustrations. Tom Tracy, of Oakland, California, took the photographs for the part openers and cover.

Michael Rogondino of Rogondino & Associates, Palo Alto, California, did the book design and the cover design. He also guided the book's production.