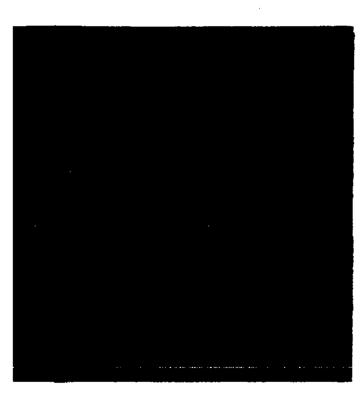
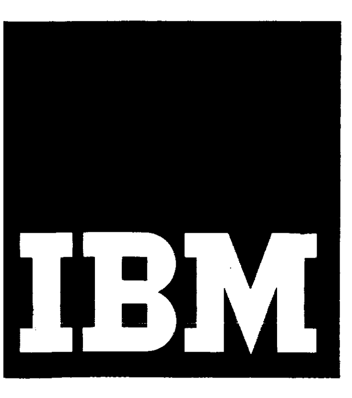




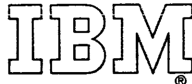
General Information Manual

IBM Commercial Translator

MAJOR REVISION

(June 1960)

This manual, form F28-8043, is a major revision of the original Commercial Translator manual, form F28-8013, which is obsolete and should be destroyed.



General Information Manual
IBM Commercial Translator

Copyright 1960 by International Business Machines Corporation

Contents

CHAPTER 1: GENERAL DESCRIPTION OF THE COMMERCIAL TRANSLATOR SYSTEM

- Introduction and Plan of the Manual 1
- Designing Data Processing Programs 1
- What Is a Data Processing System? 1
  - Communication with a Data Processing Machine 2
- The Commercial Translator System 3
- Examples—1, 2, 3 .... 3, 4, 5, 6, 7
  - Construction of the Commercial Translator Language 7
  - Examples 4, 5 .... 8, 9, 10
- Outline of Manual 10

CHAPTER 2: THE STRUCTURE OF THE LANGUAGE

- Underlying Principles 11
- Character Set 12
- Verbs 12
  - Operators 13
- Names 13
  - Kinds of Names 13
    - Data-Names 13
    - Procedure-Names 14
    - Condition-Names 14
  - Formation of Names 15
  - Compound Names 15
  - Placing Names in the Program 16
- Constants 17
  - Literals 18
  - Named Constants 19
  - Figurative Constants 19
- Expressions 20
  - Arithmetic Expressions 20
  - Conditional Expressions 21
    - Relations 21
    - Condition-Names 22
    - AND, OR, and NOT 23
  - Truth Functions 24
- Clauses 24
  - Imperative Clauses 25
  - Conditional Clauses 25
- Sentences 25
- Sections 26
- Divisions 26
- Punctuation and Spacing 27
- Lists, Tables, and Subscripts 28
- Functions 32
  - Use of Functions in Procedure Statements 34

CHAPTER 3: PROCEDURE DESCRIPTION

- Introduction 35
  - Verbs 35
  - Commands 35
  - Format of Procedure Statements 37
  - Presentation of Commands and Examples 38

Program Commands 38
Input/Output Commands 38
The OPEN Command 39
The GET Command 39
The FILE Command 40
The CLOSE Command 41
Data Transmission Commands 41
The MOVE Command 42
The MOVE CORRESPONDING Command 43
Arithmetic Commands 44
The SET Command 44
The ADD Command 47
The ADD CORRESPONDING Command 47
Control Commands 48
The GO TO Command 48
The DO Command 49
The STOP Command 54
Other Program Commands 54
The LOAD Command 54
The DISPLAY Command 54
Processor Commands 55
The OVERLAP Command 55
The BEGIN SECTION and END Commands 56
The INCLUDE Command 58
The CALL Command 59
The NOTE Command 59
The ENTER Command 59
Relationship Between Program Commands and Processor Commands 60

CHAPTER 4: DATA DESCRIPTION
What a Data Description Is 61
The Purpose of a Data Description 61
Files, Records, and Fields 63
Data Description Format 65
General 65
Ctl. and Serial (Col. 1-6) 65
Data Name (Col. 7-22) 67
Level (Col. 23-24) 68
Type (Col. 25-30) 71
Quantity (Col. 31-35) 77
Mode (Col. 36) 78
Justify (Col. 37) 78
Description (Col. 38-71) 79
Cont. (Col. 72) 84
Identification (Col. 73-80) 84
Storage Areas 84

APPENDIX 1: PROGRAMMING EXAMPLE 87
APPENDIX 2: SUPPLEMENTARY INFORMATION 105
Rules for Forming Conditional Expressions 105
Rules for Forming Arithmetic Expressions 106
List of Commercial Translator Commands 108
List of Commercial Translator Words 110
APPENDIX 3: GLOSSARY 111

INDEX 117

Preface

Background

Persons familiar with the history of programming will realize that the development of the Commercial Translator system is a logical step in the evolution of programming systems. The step from machine language coding to symbolic coding systems such as those for the IBM 702 and 705 was a natural and relatively simple development. With the introduction of "SOAP" for the IBM 650, "SAP" for the 704 and "Autocoder" for the 705, it became possible to write programs using English language words or abbreviations instead of symbolic numbers. A parallel development was the concept of writing one synthetic instruction, i.e., a macro-instruction, to represent several machine operations. Several systems at this level of development are a combination of the "one-for-one" coding of symbolic languages and the "several-for-one" coding of macros. Although these systems greatly simplify programming, they are all essentially machine-oriented; that is, the programmer must think in terms of the operation repertory of the particular machine. A highly significant step occurred with the advent of compilers such as FORTRAN. For the first time, coding systems were designed in terms of the language of the problem to be coded instead of in terms of a specific data processing system. The FORTRAN language is used to formulate mathematical problems for several data processing systems. It naturally preceded the development of the Commercial Translator system since it is directly related to the language of mathematics, the forms of which are accepted universally. Now, with Commercial Translator, a problem-oriented language becomes available for the formulation of commercial problems.

Concerning this Manual

The primary purpose of this manual is to present the Commercial Translator system in sufficient detail to permit the programming of applications in its language. Certain information is omitted: Separate publications will describe the operation of the processors, i.e., the programs for each data processing system which translate the Commercial Translator language into the machine language of that particular system. These publications will also cover the rules for stating environment description, which is the portion of a Commercial Translator program that specifies the available components of the data processing system and the external characteristics of the files to be processed.

Chapter 1: General Description of the Commercial Translator System

Introduction and Plan of the Manual

This manual has been written to be useful to two different groups of readers. Chapters 2 through 4 present a comprehensive description of the Commercial Translator language. The reader with experience in data processing may therefore begin immediately with Chapter 2. The present chapter contains a brief discussion of the essential characteristics of data processing systems (language questions aside), an indication of the motivation that led to the Commercial Translator language, and a number of relatively simple graded examples. These examples introduce the fundamentals of the Commercial Translator language without attempting to be complete.

This chapter then is intended to be of assistance to the new user who has had little or no experience in data processing.

Designing Data Processing Programs

Suppose you have been assigned to a team that is to set up a data processing system for some application in payroll, or inventory control, or utility billing, or insurance, or even in an area of science or engineering. What do you need to know about data processing in order to use the Commercial Translator language on such a job?

To begin with, we can mention a few areas of knowledge that are *not* needed. You need no knowledge of electronics. You need no knowledge of mathematics beyond high school algebra (unless, of course, the problem itself is mathematical). With the Commercial Translator language you do not even need a *detailed* knowledge of how your particular computer system works. However, you do need to know certain facts about data processing, and eventually, if you work with the subject for a while, you will pick up certain detailed facts about your particular computer–but you do not need these now. For now, the general ideas which you should have are discussed below.

What Is a Data Processing System?

A data processing system is composed functionally of five parts, as shown in Figure 1. The *input* section accepts information "from the outside," and converts it into the electronic form in which it is manipulated and stored internally. Externally, information is typically recorded on punched cards, punched paper tape, or magnetic tape. In some applications, printed characters can be read directly. Presently-used business machines cannot recognize handwriting or speech. The *output* section of a computer has the obvious function of converting from the internal representation to some convenient external form, such as printing, punched cards, magnetic tape, punched paper tape, or a variety of specialized media. Though the speeds of all of these devices are much greater than those of manual devices, they are still generally quite slow compared to speeds of internal electronic manipulation. The kind and number of input and output devices naturally depends on the particular machine and its application.

The *storage* section of a computer serves two important purposes. The obvious function is to hold the data on which we wish to operate. A function less obvious to the newcomer is to hold coded *instructions* which we place there to specify the

procedure we wish to follow. A collection of such instructions, about which we shall have more to say later, is called a *program*. There are usually two types of storage. One type, though very fast, is of limited capacity and quite expensive; it is called *main storage*. What is frequently termed *auxiliary storage* can hold much more information, but is substantially slower.

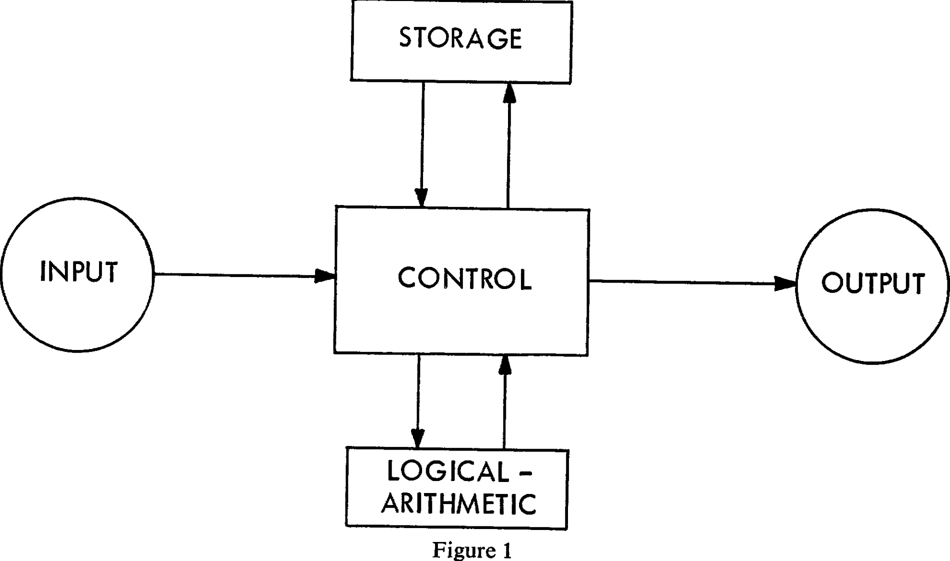


Figure 1

The last two sections of a computer are called the *logical-arithmetic* section and the *control* section. The actual data processing is done in the logical-arithmetic section, and the control section is needed to decode and interpret the instructions in storage.

Communication with a Data Processing Machine

A most important feature of modern data processing machines is the way instructions are held in main storage right along with the data. For this reason we speak of a *stored-program* machine. The machine is therefore able to carry out arithmetic on its own instructions with the result that a program can be designed to modify itself and selectively repeat certain sections.

The instructions which a data processing machine can execute naturally vary from one machine to another, but they can still be grouped into general categories. One group is used for arithmetic operations, another for making the elementary "decisions" of which a data processing machine is capable. Still another group covers input/output operations and a fourth group carries out miscellaneous control functions which are required because of the way the machine operates. Most individual operations are quite elementary, requiring that a large number of them be combined properly in order to carry out a meaningful data processing task. This work, which follows the complete definition of the processing task, is called *programming*.

Data processing requires an extremely precise statement of the problem. We must not say "less than 30" if we mean "less than or equal to 30." There is no way we can say, "make sure the data looks reasonable"; if we want to check the validity of data, we must specify exactly what tests are to be made on it.

With data processing, we are required to detail our procedures in advance to a degree not found in other methods. If we were asking a clerk to do a job, we might end by saying, "and if you run into anything you don't know how to handle, call me and we'll figure out what to do." In order to do a similar thing with a data processing machine, it is necessary first to define precisely what constitutes an exception, and then to write a procedure to handle it.

The Commercial Translator System

The initial definition of a business data processing problem usually involves some type of flow chart, often combined with a written description of the procedure to be followed. In order to be handled by a data processing machine, the procedure must be expressed in the machine's own coded, elementary instructions. In the past, it has been necessary for the programmer to make the translation between these two markedly different languages. Learning to do so is a matter of many months of training and experience.

The Commercial Translator makes it possible to let the machine itself do much of this translation. Now, we need only restate the procedure in a series of formal procedure statements and then let the Commercial Translator *processor* carry out the remainder of the translation to produce actual machine instructions.

Stated in the terms we shall use in this manual, the Commercial Translator processor (which is itself a specialized program) converts from a *source language* (the procedure statements) to an *object language* (the machine instructions).

An important advantage of writing procedure statements in the Commercial Translator language is that they do not depend on the characteristics of the data processing machine on which they will be used. In the terminology of the field, we would say that the procedure statements are *machine-independent.* For each type of machine there must be an associated processor, but the different processors will all accept the same procedure statements. This naturally means that transferring a problem from one machine to another is a far simpler task than before.

Along with the procedure description, we must provide the processor with a data description which describes the problem data. The data description is not entirely machine independent, but rather depends to some extent on the characteristics of the type of machine being used; it probably will need some revision if it is necessary to transfer the problem to a different machine.

It turns out to be a major advantage that the procedure description and the data description are separate because changes are much easier to make. With this independence, if a modification of the procedure is required, we do not have to change the data description; we simply change the affected part and use the processor again. Correspondingly we can change the data description without modifying the procedure.

Examples

The examples which follow introduce the fundamental concepts of Commercial Translator procedure description and data description. It is not intended that these examples should present all of the information about the Commercial Translator language; later parts of this manual do that. This part is intended only to introduce the subject in a manner which may facilitate learning.

Example 1

The central feature of a billing procedure is the multiplication of the unit price by the quantity sold. A sentence in a Commercial Translator program to do this could be:

CTL. 1 3	PROGRAM / PROGRAMMER	
SERIAL 4 6	PROCEDURE NAME 7 12	TEXT 13
01		SET TOTAL.PRICE = UNIT.PRICE *
02		QUANTITY.

This is the equivalent, in Commercial Translator language, of the following procedure statement:

Multiply UNIT.PRICE by QUANTITY to get the TOTAL.PRICE.

In the example SET is a word which has a specific meaning in the Commercial Translator language; namely, it is a command to carry out the arithmetic procedure specified by the rest of the sentence. SET simply states that the calculation following the equal sign is to be performed.

The asterisk (*) in this example is used to indicate multiplication, instead of an X or a centered dot, to avoid possible confusion. For the same reason we use the slash (/) to indicate division and a double asterisk (**) to indicate taking the power of a number (exponentiation). The names of the operations, together with the acceptable symbols which are used in writing arithmetic expressions, are shown in Chapter 2 on page 21.

As can be seen from the example, a Commercial Translator *sentence* is very similar to an ordinary English statement in construction and format. Actually, the parallel extends to some aspects of punctuation. A Commercial Translator sentence always ends in a period. Data is referred to by *name,* such as TOTAL.PRICE. The major exception is that names consisting of more than one English word must be constructed with "imbedded" periods since in the Commercial Translator language a space always means a new name.

A name may be formed by any combination of the 26 letters of the English alphabet and the 10 digits, as long as it begins with a letter. The imbedded period may be used freely, except that it may not be the first or last character. A valid name can be of any length up to thirty characters.

There are, of course, differences between the Commercial Translator language and ordinary English construction. There is a standard format which must be observed. In the Commercial Translator language there is also a greater need for preciseness; for instance, TOTAL.PRICE must always be spelled in exactly the same way. This implies that we don't have the same semantic flexibility as in normal English. A particular name must be used to mean just one thing in a given program.

Although this example, which is used to illustrate the simplest ideas about the Commercial Translator language, is based on a price calculation, it might have been based on computations from many different fields. It might just as well have dealt with payroll, or inventory control, or insurance, or mathematical work in engineering.

Example 2

For a second example, consider a part of an inventory calculation. One sentence of the Commercial Translator procedure for determining whether or not to place an order might be:

SERIAL (4–6)	PROCEDURE NAME (7–12)	TEXT (13–)
01	REORDE	R.ROUTINE. IF QUANTITY.ON.HAND IS
02		LESS THAN MINIMUM THEN MOVE
03		ORDER.QUANTITY TO PURCHASE.AMOUNT.

The first word, REORDER.ROUTINE, is called a *procedure-name*. This is indicated by the fact that it begins in the procedure-name area of the programming form and is followed by a period. A procedure-name provides a way of referring to the sentence which follows.

The sentence illustrates a conditional expression involving a simple relation between two quantities. If the QUANTITY.ON.HAND is less than the MINIMUM (in other words, the reorder point), the action specified following THEN is carried out. If QUANTITY.ON.HAND is not less than the MINIMUM, the action following THEN is *not* carried out.

Figure 2 shows in schematic form the structure of this sentence.

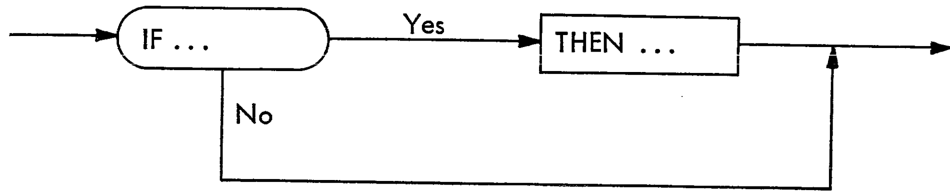


Figure 2

The IF...THEN construction is one of the most important features of the Commercial Translator language. The *conditional clause,* which is introduced by the word IF and concluded by THEN, in effect asks a question to which the answer must be yes or no. We shall speak of each relation involved in a conditional expression as being true or false, or satisfied or not satisfied. This example uses the IS LESS THAN relation. The allowable relations, and the Commercial Translator words which may be used to express them, are shown in Figure 3.

Long Form	Short Form
IS EQUAL TO	=
IS NOT EQUAL TO	NOT =
IS GREATER THAN	GT
IS NOT GREATER THAN	NOT GT
IS LESS THAN	LT
IS NOT LESS THAN	NOT LT

Figure 3

Example 2 also shows a different command, MOVE...TO. The action called for is the copying of information within storage. The information named ORDER.QUANTITY is to be copied and called PURCHASE.AMOUNT.

It is probably apparent by now that certain words have special meanings in the Commercial Translator language: in the present example, the words IF, THEN, MOVE, TO and the phrase IS LESS THAN all have special meaning, and confusion would result if we tried to interpret these words in any other way. Such words are a fixed part of the language. A complete list of the Commercial Translator words appears in Appendix 2 at the end of the manual. Incidentally, the inference one might draw from the statement above is correct: a Commercial Translator program consists of just two kinds of words: fixed words, and names which the programmer selects.

Example 3

To introduce a few more features of the Commercial Translator we use a common payroll example:

SERIAL (4–6)	PROCEDURE NAME (7–12)	TEXT (13–)
01		IF HOURLY AND HOURS.WORKED IS LESS
02		THAN 40 THEN GO TO GROSS.PAY
03		OTHERWISE GO TO NET.PAY.
04	GROSS.PAY.	
19	NET.PAY.	

The conditional clause in this example is different from what we have seen previously. The first part of the clause consists just of the word HOURLY, which is called a *condition-name*. HOURLY is one of the possible values which can be assumed by the implied *data-name* PAYROLL.TYPE, the other values being EXEMPT, SALARIED and TEMPORARY. Since there are only a few of these conditions, it is convenient for the programmer to use his normal terminology. The actual machine instructions are set up to work with a coded representation of these values, e.g., the numbers 1, 2, 3, and 4. Obviously, some way must be provided to correlate the condition-names with the corresponding values. Establishing this correspondence is one of the functions of the *data description*. Shown below is the appropriate part of the data description required for this example.

SERIAL (4–6)	DATA NAME (7–22)	LEVEL (23–24)	TYPE (25–30)	QUANTITY (31–35)	MODE (36)	JUSTIFY (37)	DESCRIPTION (38–)
01	PAYROLL.TYPE	03					9
02	EXEMPT	04	COND				'1'
03	SALARIED	04	COND				'2'
04	HOURLY	04	COND				'3'
05	TEMPORARY	04	COND				'4'

PAYROLL.TYPE is defined as a level 03 entry which indicates its relative importance with respect to other elements of data. The 9 specifies that the data is numeric, and the fact that there is only one 9 means that the field consists of one digit. EXEMPT, SALARIED, HOURLY and TEMPORARY are named as the four conditions by listing them with COND in the Type columns, and the code number used for each is given in quote marks. Thus, in this example, the value of PAYROLL.TYPE is 3 whenever HOURLY is meant.

The second part of the conditional clause is:

HOURS.WORKED IS LESS THAN 40

In this case the value of the data-name HOURS.WORKED is compared with the number 40; 40 is not to be interpreted as a data-name, but literally as the value 40 itself. We speak of 40 as being a *numeric literal*.

The second part of the conditional clause is joined to the first part by the fixed word AND which specifies that both the first part and the second part of the clause must be satisfied before carrying out the operations that follow THEN. This is shown

schematically in Figure 4, which emphasizes that both parts of the conditional clause must be true before carrying out the action specified after THEN. If either (or both) of the parts is false, the action specified after OTHERWISE is executed; if OTHERWISE is absent the program continues with the succeeding sentence.

In the Commercial Translator sentence under consideration, the actions specified after both THEN and OTHERWISE happen to be a new command, GO TO. The GO TO command makes it possible to get out of the one-after-the-other sequential execution of sentences and sections and instead execute next the sentence named by the GO TO.

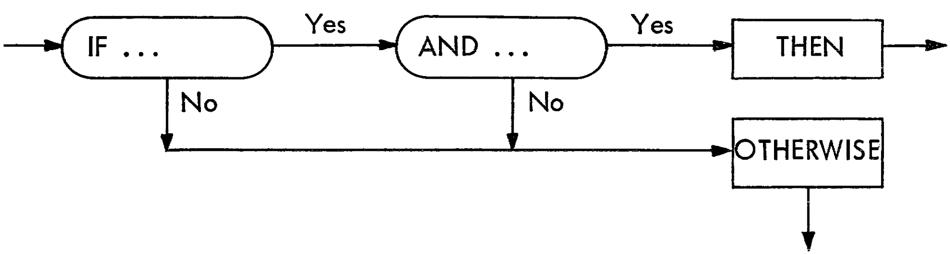


Figure 4

Construction of the Commercial Translator Language

These examples are intended to introduce some of the important ideas of the Commercial Translator language with a minimum of formality, and without attempting to be complete. It may be, however, that the reader will find the remaining examples more meaningful if we pause here to generalize about some of the concepts which have been illustrated.

It may be helpful to think of the Commercial Translator language as existing on two different levels. At the first level, we are concerned with the *elements* of the language: characters, fixed words, punctuation, names, literals, etc. In short, we wish to define precisely the constituent elements of the language. This is done in Chapter 2.

The second level of the language, which may be called the *syntax* of the Commercial Translator language, has to do with how to fit elements together properly, in such a way as to describe the desired procedure. Here we are concerned with the rules for grouping words, provisions for joining expressions, techniques for program organization, etc. These are covered in detail in Chapters 2 and 3.

For instance, one basic structural building block is an *expression;* this is defined as any grouping of elements which always establishes a unique value. An arithmetic expression can take on any numeric value whereas a conditional expression may only be true or false. A true expression is sometimes converted to the value 1 and a false expression to the value 0.

Another concept in the language structure is that of a *clause.* A clause consists of a fixed word or words, together with an associated format or framework into which the programmer inserts names and expressions. We may distinguish two types of clauses: *imperative* clauses, i.e., commands, and *conditional* clauses.

We have seen examples of commands in SET and MOVE operations. We saw that in each case there was associated with the command a format: SET a name = an expression, and MOVE a name TO a name. Commands are independent clauses which may stand alone. When the source program is converted to the object program, commands are converted directly into corresponding machine instructions in the object program.

A conditional clause starts with IF and contains one or more conditional expressions; the actions to be taken if the conditional clause is true are described by one or more commands introduced by THEN. If the conditional clause is false, the commands following THEN are ignored and instead the commands introduced by OTHERWISE are executed. Conditional clauses are dependent clauses and must always be followed by at least one command.

Still another concept of the language is that of *processor* commands. So far, we have been discussing *program* commands, that is, commands which state the data processing steps to be carried out by the object program. The processor commands, on the other hand, tell the processor *how* to organize the object program rather than *what* the object program is to do. One kind of processor command, BEGIN SECTION, simply tells the processor that the following sentences are to be referred to collectively by the designated name; it does not directly cause machine instructions to be created in the object program.

The reader may wish to consider the examples which follow in the light of these general observations about the elements and structure of the language.

Example 4

The fourth example is based on a part of an inventory control calculation. Suppose that we already have in storage in the data processing system a complete inventory record for one part number, and that there is also in storage a transaction record for that part number. Assume that there are just four types of transactions: withdrawals, receipts, returns and stock recounts. The part of the job that we wish to consider is how to take action appropriate to the type of transaction. The program shown below is a little longer than the ones we have seen before, but most of the ideas in it are already familiar.

SERIAL (4–6)	PROCEDURE NAME (7–12)	TEXT (13–)
01		NOTE INVENTORY RECORD MAINTENANCE.
02		GO TO (WITHDRAWAL.ROUTINE, RECEIPT.ROUTINE,
03		RETURN.PROCESS, RECOUNT.PROCEDURE) ON
04		TYPE.OF.TRANSACTION.
05	WITHDRAWAL.ROUTINE.	SET QUANTITY.ON.HAND =
06		QUANTITY.ON.HAND - TRANSACTION.QUANTITY,
07		DO REORDER.CALCULATION, GO TO NEXT.ITEM.

The first line of this program brings into play a processor command:

NOTE INVENTORY RECORD MAINTENANCE.

NOTE indicates that what appears in the rest of the sentence is information for the reader of the program; it is not for the Commercial Translator processor, which ignores it. The programmer is permitted and encouraged to use notes freely, in order to make the program more intelligible to the reader.

The GO TO shown on line 02 is a more powerful form of this command than we have seen before:

GO TO (WITHDRAWAL.ROUTINE, RECEIPT.ROUTINE,
RETURN.PROCESS, RECOUNT.PROCEDURE)
ON TYPE.OF.TRANSACTION.

This is called an assigned GO TO. For any one transaction, only one of the four procedures named in the parentheses will be performed; the one selected will depend on the current value of TYPE.OF.TRANSACTION. Suppose the value of TYPE.OF.

TRANSACTION can vary from 1 to 4. These numbers correspond to the names within the parentheses; if the value is 1 the first name will be selected and so on. This is summarized in Figure 5.

If the current value of TYPE.OF.TRANSACTION is:	Then GO TO:
1	WITHDRAWAL.ROUTINE
2	RECEIPT.ROUTINE
3	RETURN.PROCESS
4	RECOUNT.PROCEDURE

Figure 5

This assigned GO TO provides a multiple branch or switching point. For those with punched card background this may also be thought of as a "digit selection" operation.

Lines 05 and 06 illustrate another use of the SET command. It is evident from examining this sentence that the equal sign (=) in the SET command is used to mean "replace." SET A=B means replace A by the value of B, where B represents any valid expression. Thus, in the example, the difference between QUANTITY.ON.HAND and TRANSACTION.QUANTITY (to the right of the equal sign) replaces the original QUANTITY.ON.HAND.

Line 07 demonstrates another type of transfer of control:

DO REORDER.CALCULATION

The DO command may be thought of as meaning "go to the place named, do whatever it says to, and come back." In our case, it is used to transfer to a procedure named REORDER.CALCULATION and set up a return path so that, after executing that procedure, control will return to the command immediately following the DO command. After performing the REORDER.CALCULATION, control will return to, and execute, the GO TO NEXT.ITEM command on line 07.

Example 5

As another illustration of the use of Commercial Translator, we will use a savings bank procedure: updating the account record to indicate interest payment. The program might look like this:

```
SERIAL  PROCEDURE     TEXT
        NAME
4    6  7          12 13
01      INTEREST.CALCULATION.  IF .03 * PRINCIPAL
02                   IS LESS THAN 1.00 THEN GO TO
03                   END OTHERWISE SET ACCOUNT BALANCE
04                   = 1.03 * PRINCIPAL, MOVE 'INTEREST'
05                   TO ACTION.
```

Line 01 shows again the use of a procedure-name to provide a named point to which program control can be transferred. In this case it precedes a slightly different type of conditional clause. Instead of simply comparing two values as we have before, the programmer has indicated that he wishes to see if the value of an arithmetic expression (.03 * PRINCIPAL) is less than some numeric literal (1.00). It is quite valid to incorporate an arithmetic expression within a conditional expression. For clarity, it may often be desirable to use parentheses to denote the beginning and end of such an arithmetic expression.

The careful reader may have noticed what appears to be an error in line 03: there seems to be an imbedded period missing in ACCOUNT BALANCE. However, the omission is deliberate; this is an example of *name qualification.* Remember that in an ordinary name, a blank indicates the start of a new word. Here, BALANCE is the name of a *field* of data; ACCOUNT is the name of a *record,* which includes a number of fields. The idea of using record names to qualify field names is that certain field names might be fairly common, and that the programmer should not be required to think up unique names for every field in every record. Instead, he can name the fields in any way that seems reasonable and then identify each field by the record in which it appears. Thus, in our example, ACCOUNT BALANCE means the field BALANCE which appears in the record ACCOUNT. Of course, a name qualifier is not required if a particular name is, in fact, unique.

There is one other new point to notice in this example. On lines 04 and 05 we have:

MOVE 'INTEREST' TO ACTION.

The quotation marks indicate that the word INTEREST itself is to be moved to the area named ACTION. Thus INTEREST is identified by the quotation marks as being an *alphameric literal.* An alphameric literal may contain any characters except the quotation mark.

Outline of Manual

The following pages are devoted to a detailed description of the Commercial Translator language and its application. The description proceeds as follows:

Chapter 2—This chapter covers the elements, or components, of the Commercial Translator language and describes the overall structure of the language.

Chapter 3—The procedure-describing part of the language is discussed in this chapter. The various Commercial Translator verbs and the commands in which they appear are explained in detail.

Chapter 4—The rules and conventions for stating the data description are covered in this chapter.

Appendices—

1. A sample payroll program written in the Commercial Translator language is presented in Appendix 1.
2. Appendix 2 contains supplementary information of various kinds for reference purposes.
3. A glossary of terms is included in Appendix 3.

Chapter 2: The Structure of the Language

Underlying Principles

The structure of the Commercial Translator language is very much like that of English. It has, first of all, a basic vocabulary, consisting of words and symbols. Next, it has a set of basic rules of "grammar" or "syntax" by which the words and symbols may be combined to express meanings. Finally, it has punctuation symbols to clarify the groupings of words and symbols so that their meanings will be unmistakably clear.

The Commercial Translator language is much simpler than English, for its requirements are more limited and more precise. Since it is a programming language, it must have the capacity to state facts and give instructions. It must be clear and specific; it does not require the ability to express delicate shades of meaning. The Commercial Translator is a language of action—a language for getting things done—and it therefore consists largely of verbs and nouns. The verbs direct the system to take action of various kinds; the nouns name the data and the procedures on which action is to be taken.

As in English, verbs and nouns may be combined into clauses and sentences so that meanings may be clearly defined. A number of typical Commercial Translator sentences have been examined in Chapter 1 of this manual. The reader will thus have discovered that he can usually tell what a Commercial Translator sentence means merely by reading it, although at this point he may not know the rules for writing such a sentence or the ways in which a sentence can be used in a program. It is the purpose of this chapter to describe the basic components of the language and to show the rules for combining them to express meanings.

Character Set

The words and symbols of the Commercial Translator language are the basic units with which the programmer will work, but he should understand that they in turn are composed of individual letters, numbers, and special characters—in short the basic character set. This set consists of the 26 letters of the alphabet, the ten numerals from 0 to 9, and the special characters shown in the table below.

Special Characters Used in the Commercial Translator Source Language

Name	Character (Set H*)	Card Code
blank		(blank)
plus sign	+	12
minus sign	—	11
multiplication sign	*	11-4-8
division sign	/	0-1
left parenthesis	(	0-4-8
right parenthesis	)	12-4-8
comma	,	0-3-8
period / decimal point	.	12-3-8
dollar sign	$	11-3-8
equal sign	=	3-8
quotation mark	'	4-8

*Set H is one of several character sets available for IBM equipment. All sets use the same card codes, but in the case of special characters, one code may represent one character in one set and another character in another set. For example a "12" punch indicates a plus sign in Set H, while in certain other sets it represents an ampersand. The Commercial Translator system employs the codes of Set H, shown above.

The uses of these characters will be explained subsequently in this manual. The reader should note that the blank is treated as a character, but that a series of blanks will be regarded as a single blank, except in alphameric literals or other constants.

Verbs

In the Commercial Translator language, a verb is a word that prescribes an action. Verbs are not used in a declarative sense in the language, and the programmer should recognize from the start that everything he writes will produce some kind of effect.

The prescribed action may not take place at the time the object program is run. In fact, a number of verbs give instructions which the processor will carry out at the time the object program is assembled. For example, as the programmer analyzes a data processing problem he may recognize that it contains elements which he has already handled in solving previous problems. If so, he may have written program "routines" that can now be used again. If such a routine has been stored in a "library," the programmer may be able to call for it by using the verb INCLUDE, as explained in Chapter 3 of this manual. This verb, therefore, may be used in building the object program, but it will not be used as a part of the object program itself. Verbs which act on the processor when the source program is translated are called *processor verbs*.

However, most of the verbs the programmer will write will cause some action at "object time"—i.e., at the time the object program is run. Thus, the verb ADD will

cause two or more items of data to be added together, the verb DISPLAY will cause specified information to be printed or otherwise displayed, and the verb STOP will cause the machine to halt. Verbs which act at object time are known as *program verbs.*

Operators

Not all words that cause action are verbs. For instance, the reader will learn later in this chapter how to order that a given action be carried out only if a certain condition is met. Consider the sentence IF A = B THEN MOVE A TO OUTPUT. This sentence contains only one verb, the word MOVE, yet it implies two separate operations. The MOVE operation, of course, is one of them; the other is a test to determine if the condition A = B has been met. However, the programmer does not have to write a verb directing the program to test for this condition, for the word IF serves the purpose. The Commercial Translator language contains several words and symbols (such as the arithmetic symbols) which are not verbs but which cause operations. They are called *operators.*

It is necessary to distinguish between verbs and operators, since they are used in different ways. Commercial Translator verbs may be easily recognized by the fact that all of them are words which serve also as verbs in the English language.

For a general indication of how verbs are used, the reader may refer to any of the examples used in Chapter 1. The specific rules governing each verb will be found in Chapter 3.

Names

Most of the words in a typical Commercial Translator program will be nouns. A noun, in the strictest sense, is a *name,* and the programmer will find it very useful to accept that definition and all of its implications. He will write procedures for handling data, but he will refer to the procedures and the data by name. He will rarely work with actual data, and he will never write actual machine instructions.

This is an important concept. As will be seen in Chapter 4, an electronic system has no way of recognizing data and procedures simply by looking at them. They can be identified only by their location within storage, and in the Commercial Translator system these storage locations are referred to by name. For this reason, Commercial Translator nouns will be spoken of hereafter as names.

Kinds of Names

Most of the names the programmer will use will fall into one of three general categories: *data-names, procedure-names,* and *condition-names.*

Data-Names

Data-names are names given to the data used in a program. As the programmer will see, data-names are assigned to *kinds* of data, not (except in the case of constants) to specific values. Thus, such a name as INTEREST.RATE would not refer to a specific interest rate, but to a class of data known as interest rates. Technically (although this need not concern the programmer), it would name an area in storage in which any of several specific interest rates might be placed for some purpose, such as computation.

Data-names may be assigned to single classes of data or to groups of data items. For example, the name PAYROLL.RECORD would probably refer to a group of individual items having such names as EMPLOYEE.NAME, EMPLOYEE.NUMBER, HOURLY.RATE, and so on. It is important to recognize that, in general, the name refers not to any specific value, but only to the kind of data.

Data-names are invented and assigned to data at the discretion of the programmer, following the rules given below which govern the formation of names. All data referred to in the program must be named, but this does not mean that all subdivisions of data must be named. For example, if the programmer wishes to refer to a date, he will have to give it a name, but he does not have to name the component parts, such as the day, the month, or the year, unless he wishes to refer to them individually.

The general category of data-names may be broken down, for convenience, into record-names, field-names, function-names, parameter-names, named constants, and so on. The meanings of these names will be explained later in this manual.

Procedure-Names

Procedure-names are names assigned to individual portions of the program so that one procedure can refer to another. Suppose, for instance, that the programmer has written a routine for computing a discount and that he wishes to use this routine at various times in the program. He could actually copy the routine into the program each time it is required, but this may be inefficient. An alternative method would be to give the routine a name, such as DISCOUNT, or DISCOUNT.CALCULATION; then, when the programmer wished to use this routine he could write such an instruction as DO DISCOUNT.CALCULATION. As the reader will discover, there are many reasons why the programmer may wish to refer from one part of the program to another. Procedure-names provide him with the means of doing this.

Like data-names, procedure-names are invented and assigned by the programmer as he needs them, following the rules for name formation given below. They are placed at the beginning of the portion of the program to which they apply.

Procedure-names may be assigned to any sentence or section of the program. (A section consists of one or more successive sentences.) The reader will find further details of how to use procedure-names in Chapter 3 in the discussion of the commands DO, GO TO, INCLUDE, OVERLAP, BEGIN SECTION, and END. Procedure-names may also be referred to as sentence-names or section-names, as appropriate.

Condition-Names

The concept of condition-names will be more clearly understood after the reader has studied the discussion of conditional expressions beginning on page 21 of this manual. In general, however, a condition-name defines a condition which can be used to control an operation.

For example, suppose that a manufacturer has agreed to pay shipping charges for goods shipped to customers within 50 miles of the factory, but that charges for shipment out of this 50-mile zone will be billed to the customer. Obviously, two different routines are called for in the program, and the decision as to which will be used will depend on the shipping distance.

Suppose that a code is used in the input records to indicate whether the customer is located within the 50-mile zone or not. The programmer will need to refer to this code, and it will usually be convenient to be able to give it a name which can be written in a procedure statement.

In this case, the programmer might wish to use the name OUT.OF.ZONE to indicate that a customer is located more than 50 miles from the factory. He can place this name in the program by writing an entry that instructs the system that the name is to be treated as an equivalent for the code it represents. (The actual method of doing this is explained in Chapter 4.)

Once the condition-name has been defined, the programmer may use it in the procedure statements. For example, he might write such a sentence as IF OUT.OF.ZONE THEN DO BILLING.ROUTINE.A OTHERWISE DO BILLING.ROUTINE.B. The expression IF OUT.OF.ZONE will cause the system to examine the data to determine if the indicated condition has been met. When the reader has studied the discussion of conditional expressions, he will realize that a condition-name of this kind is actually a short way of writing an expression that shows a relationship between two items. Thus, if the "out-of-zone" code were 2, and if the field containing that code were called DISTANCE.CODE, the condition-name OUT.OF.ZONE would actually be equivalent in every way to the expression DISTANCE.CODE = 2.

Condition-names are subject to the general rules for the formation of names, which are given below. They may be assigned by the programmer at his discretion.

Formation of Names

Names may be formed by combining any of the characters from the basic list of alphabetic characters, numerals, and the period, subject to the following rules:

1. Names must not contain blanks.
2. Names must always begin with an alphabetic character.
3. They may contain from 1 to 30 characters.
4. They may neither begin nor end with a period. However, "imbedded" periods may be used within the name for the sake of readability.
5. They may be "qualified" (to make them unique within the program) by the use of other names. This is explained below under the heading "Compound Names."

Compound Names

In many cases a program will contain duplicate names. This often happens when an input file is "updated" to produce an output file, since each file will usually contain the same kinds of records.

Suppose that an input record is named INPUT.MASTER and an output record is called OUTPUT.MASTER. Suppose, further, that each record contains two dates, one called ORDER.DATE, the other called SHIPMENT.DATE.

If the program involves both kinds of records, it would not be possible to distinguish readily between the two ORDER.DATE names and the two SHIPMENT.DATE names. All four names would be defined in the data description (see Chapter 4), which gives the system the information it needs to locate individual items of data. To indicate which of the ORDER.DATE (or SHIPMENT.DATE) names is meant, however, each such name can be "qualified," or "compounded," when used in a procedure statement. That is, the name of a larger data item of which it is a part can be added to the name to identify it. Thus, INPUT.MASTER ORDER.DATE would be clearly distinguishable from OUTPUT.MASTER ORDER.DATE. Names qualified in this manner are referred to as *compound names.*

Compounding of names is not limited to two levels. For example, the various dates mentioned in this example may each have an element called MONTH to which the programmer may wish to refer individually. If, at the time of reference, the program is working with only one record which contains an element called MONTH, there is no ambiguity. But should ambiguity exist, the name of a data item at a higher level may be used as a qualifier. Thus, the programmer may specify, in this case, ORDER.DATE MONTH or SHIPMENT.DATE MONTH. If, as is likely, both input and output records are in use at the same time, three levels of names may be used in a single compound name—for example, INPUT.MASTER ORDER.DATE MONTH.

When names are to be compounded, the following rules apply:

1. Each name must be separated from the next by at least one blank space. (This distinguishes between compound and simple names, since simple names may not contain blanks.)

2. The names must be written in increasing order from the general to the specific. (If the reader is familiar with the concept of level numbers, as discussed in Chapter 4 of this manual, he will note that this means that the names must be listed in order from the lowest to the highest level number.)
3. No qualifying names are required that do not contribute to the uniqueness of the compound name. Thus, in the example given, if there were only one date named in each of the input and output files—e.g., an ORDER.DATE, but no SHIPMENT.DATE—it would not be necessary to use the name ORDER.DATE in forming the compound name; the names INPUT.MASTER MONTH and OUTPUT.MASTER MONTH would suffice.

The organization and structure of data for use in a data processing system is further discussed in Chapter 4, entitled "Data Description." The reader is referred, in particular, to the discussion of level numbers beginning on page 68.

Placing Names in the Program

The reader has seen that the Commercial Translator system uses names as a convenient—in fact, indispensable—means of identifying data, procedures, and conditions. It is now necessary to indicate how each name is placed in the program in a way that permits the system to connect it with the item to which it refers.

A Commercial Translator program consists primarily of *procedure description* and *data description.* The first of these is made up of procedure statements—the actual instructions which govern both the processor and the object program. The second consists of data description statements—entries which instruct the system to reserve a specified amount of storage space for each kind of data and which show the organization and nature of the data so that it can be located and used when needed. These two sections are discussed in Chapters 3 and 4.

All statements for a Commercial Translator program must be written in a specified format, and, as a guide to the programmer, two columnar forms have been prepared for this purpose. One is used for procedure statements, the other for data description statements. The first is described in Chapter 3, the second in Chapter 4.

Procedure-names differ from other names in one important respect. They are used as names for *sentences* and *sections* (which themselves usually contain names of various kinds), whereas data-names and condition-names identify *kinds of information.* A procedure-name identifies a fixed set of procedure statements. A data-name usually identifies a storage area that may contain different values at different times.

Procedure-names are identified in the procedure description. This is a simple matter. It consists merely of writing the procedure-name before the statement or statements to which it refers, in accordance with the rules given in Chapter 3. Once the name has been written, the program will be able to interpret any reference to the name as a reference to the associated procedure statement.

Data-names and condition-names, however, require amplification. As will be seen in Chapter 4, the system must know whether the data is numeric or whether it contains alphabetic characters. It must know where decimal points, if any, are to be placed, where to print dollar signs, and so on. There are a number of such details which must be specified. It would have been possible to set up rules for describing the data in the procedure description, but this would have been inefficient, since the description of each item would have had to be repeated each time its name appeared. Since the description is placed in a separate section, however, each name need be described only once, regardless of the number of times it is used in the program.

It follows that each data-name and each condition-name used in the procedure description must be properly accounted for in the data description, following the rules given in Chapter 4. Once this has been done, the programmer is free to refer to the name repeatedly throughout the procedure description.

Constants

It has been emphasized that data-names used in the Commercial Translator system generally refer to kinds of data, not to specific values. The actual values represented by most data-names are assumed to be variable, and they will either be entered into the system as parts of input files when the object program is run or they will be computed at some point in the program.

However, the programmer will often find it useful to be able to place a specific fixed value into the program instead of having to read it in as data. For example, if a firm allows a discount on its bills, the discount will usually be figured as a fixed percentage. The routine for computing the discount, therefore, does not require any provisions for inserting varying percentage rates. Thus, it would be convenient to be able to write this rate directly into the program.

Any value—or any group of symbols—which is to be used in the program without alteration is called a *constant*. The programmer will find many uses for numeric constants, such as the discount rate mentioned, for alphabetic constants, such as names and titles to be printed out on final reports, and for alphameric constants, which may serve any number of purposes.

In some circumstances, it will be convenient to write the constant directly in a procedure statement. In this case it will be called a *literal*. In other cases, it will be more convenient to give the constant a name and store it within the system so that it can be called for by name when required. In this case it will be called a *named constant*.

As an aid to the programmer, certain standard constants, such as the value 0 and the blank, have been "pre-named." These values are defined in the processor itself, and they have already been given names. Thus, the programmer can write these names in the procedure statements without having to define them in the data description. These special constants, called *figurative constants,* will be discussed later in this section.

Literals and named constants may be used in procedure statements for the same purposes for which data-names are used—that is, as "operands" (i.e., "objects") of Commercial Translator verbs. The essential difference between them is that a literal expresses an actual value—a value to be read "literally" at the point where it is written—whereas a named constant is the *name* of such a value, and it cannot be used, or interpreted, in a procedure statement until it has been defined in the data description.

The following example will show the difference between literals and named constants:

Assume that a discount is to be computed by multiplying the amount of an order by a discount percentage of two per cent. The programmer might write such a statement as

SET DISCOUNT = ORDER.AMOUNT * .02.

This command would instruct the system to take whatever value was in the field called ORDER.AMOUNT, multiply it by the value .02, and place the result in the field called DISCOUNT.

On the other hand, the programmer could place the value .02 in storage, giving it a name such as DISCOUNT.PERCENTAGE, and then write such a statement as

SET DISCOUNT = ORDER.AMOUNT * DISCOUNT.PERCENTAGE.

In this case, the system would take the value in the ORDER.AMOUNT field, look up the value in the field called DISCOUNT.PERCENTAGE, multiply the two values together, and store the result, as before, in the DISCOUNT field.

The same result is obtained in either case, and it may appear at first sight that it is more efficient to write literals than named constants. This may or may not be so. If the constant is short, as in this example, it will usually be more convenient to write it as a literal. If it is long, and if it can be given a short name, it may be more efficient to treat it as a named constant. Furthermore, the technique of naming a constant makes it possible to store large quantities of reference material, such as lists and tables, within the system, and to make use of selected items from such a list or table as required. The reader will see later how this may be done.

It should be noted, however, that in certain special cases the name of a named constant is implied, rather than actually written. In such a case, the constant will be a part of a data item which *is* named, and it will always be possible to identify it by making an appropriate reference to the named item. The reader will see, in the discussion of tables, that each individual item in a table need not be named, but the table itself (and, often, specific kinds of data within the table) will be named. The programmer can then refer to subordinate elements in the table by a kind of indexing known as "subscripting." This is explained later in this chapter.

Literals

Although a literal may be written and used in a procedure statement as if it were a data-name, it differs from data-names (including named constants) in that its value is the value literally stated—it is not used as a name for some other value.

Rules for Forming Literals

Literals may be numeric, alphabetic, or alphameric. Some of the rules for forming numeric literals differ slightly from the rules for alphabetic and alphameric literals. For convenience of reference, the rules governing each type are listed separately.

NUMERIC LITERALS

1. All literals are limited to 50 characters in length, and, when written on the columnar form used for writing procedure statements, they may not be carried over from one line to the next.
2. Numeric literals may contain only numerals, not more than one decimal point, and a plus or minus sign to indicate whether the value of the number is positive or negative. "Floating point" numbers also contain the letter F, as explained in Rule 4 below, and may contain more than one plus or minus sign. The decimal point is required except where it would be the last character of the literal; in that case it must *not* be used. The decimal point will be noted by the system in order to align the number properly for use, but it will not occupy an actual place in storage, and it is not counted in determining the length of the literal.
3. If the literal is to be operated on arithmetically, it must contain not more than 20 digits.
4. Numeric values may be entered as "floating point" numbers by writing the "fraction" (i.e., the number or decimal fraction), then the symbol F, and then the exponent. The fraction and the exponent may each have a plus or minus sign. The symbol F will not occupy a space in storage, and it is not counted in determining the length of the literal. The system will accept floating point numbers using a base of 10 only. (A floating point number is a number expressed as a decimal number or decimal fraction multiplied by some power of 10. For example, the number 1500 might be written as 1.5F3, which is equivalent to 1.5 times 10^3; the same number might also be written as 15F2, .15F4, or in any other similar way that is convenient. The number .002, which is equivalent to 2 times 10^{-3}, might be written 2F-3.)
5. Numeric literals must not be enclosed in quotation marks.

Alphabetic and Alphameric Literals

1. An alphabetic or alphameric literal may contain any of the characters from the basic character set except the quotation mark. A blank is treated as a character and may be included in an alphameric literal.
2. Like numeric literals, alphabetic and alphameric literals are limited to 50 characters in length and may not be carried over from one line to another when written on the columnar form used for writing procedure statements.
3. All non-numeric literals must be enclosed in quotation marks to distinguish them from names. This rule applies even should the literal contain symbols (such as the arithmetic symbols and the blank) which may not be used in names.

Named Constants

Named constants are placed in the system by specifying them in the data description in accordance with the rules given in Chapter 4. Each named constant will usually have its own individual name, but in certain cases it may be part of a group of constants having a group name. As the reader will note from the discussion of tables and lists, individual items in a list are often in this category. However, even when the individual item does not have a name, it is always referred to by a name of some kind, and it may thus be treated in the general category of named constants.

A named constant may be referred to in procedure statements by writing its name as if it were any other data-name (although, of course, it should not be used in a way which will change its value).

Rules for Forming Named Constants

Named constants, like literals, may be wholly numeric, wholly alphabetic, or alphameric. All of the rules specified above for forming literals apply to constants, except as follows:

1. Named constants are not limited as to length, and when written on the columnar form used for data description entries, they may be carried over from one line to the next. In this case they are subdivided and written as a series of complete constants, one on each line, but at a level lower than that of the name given to the total constant. After these parts have been read into the system, the processor will reassemble them to form the original constant.
2. All named constants, including those which are wholly numeric, must be enclosed in quotation marks.
3. Named constants may include any of the characters in the machine's character set, including the quotation mark and the blank. (The presence of a blank in an otherwise numeric constant makes it alphameric.)

Figurative Constants

The figurative constants resemble named constants except that their names are already assigned in the processor itself, so that the programmer need not write data description entries for them.

Figurative constants are names for certain constant quantities which are used frequently in data processing programs. The list includes names which represent zeros, blanks, and the lowest and highest characters in the collating sequence of the machine system being used. Following is a list of the figurative constants:

ZERO or ZEROS
BLANK or BLANKS
LOW.VALUE or LOW.VALUES
HIGH.VALUE or HIGH.VALUES

In general, a figurative constant is used to place the value it names in a given storage area, although it is not limited to this usage. For example, if the programmer

wishes to reduce the value of a data item called COUNTER to zero, he can do so by writing the instruction MOVE ZEROS TO COUNTER. This procedure will replace all previous data in COUNTER by zeros. Similarly, if he wished to erase all data in an area called AMOUNT, he could write MOVE BLANKS TO AMOUNT. In each case, the specified area will be completely filled with characters of the value named.

The names LOW.VALUE and HIGH.VALUE refer, respectively, to the lowest and highest characters in the collating sequence of the system for which the program is written.

Expressions

The words and symbols of the Commercial Translator language are combined into clauses and sentences in order to give instructions to the processor and the object program. Before considering the formation of these larger units of the language, however, it is important to examine two specialized forms of expression: arithmetic expressions and conditional expressions.

Arithmetic Expressions

An arithmetic expression is a combination of data-names, conditional expressions, and/or literals, joined together by one or more arithmetic operators in such a way that the entire expression can be reduced to a single numeric value. (An arithmetic operator is a symbol representing addition, subtraction, etc.; a list of operators is given below.) Both simple and compound names may be used in an arithmetic expression.

Consider the following examples:

```
A + B
GROSS.PAY - DEDUCTIONS
GROSS.SALES * COMMISSION
BEGINNING.ON.HAND + RECEIPTS - SHIPMENTS
A * (B + C) - (D / E)
TOTAL.SALES * .3
```

In these expressions, the symbols +, −, *, and / are arithmetic operators used to express addition, subtraction, multiplication, and division, respectively. They link together a variety of terms, including the values represented by the data-names A, B, C, GROSS.PAY, DEDUCTIONS, and the literal .3. All of these will represent specific values at object time. If the data-name GROSS.PAY, for example, should be 125.50, and if DEDUCTIONS should turn out to be 31.20, the expression GROSS.PAY - DEDUCTIONS would reduce to a value of 94.30. Similarly, if TOTAL.SALES should have a value of 12,000, the expression TOTAL.SALES * .3 would reduce to a value of 3,600.

Arithmetic expressions may be used as components of conditional expressions, clauses, and sentences, as will be shown later in this chapter. Use of an arithmetic expression will cause a computation to be performed in order to obtain the single result to which the expression can be reduced.

It was stated above that conditional expressions can also be used in arithmetic expressions. This is a specialized usage in which a test is made to determine whether or not a particular condition is met. The "truth operator" TR is used to indicate this test. It is generally used to multiply one or more terms in an arithmetic expression. Since it takes on a value of 1 if the conditions of the test are met, and a value of 0 if they are not met, it can be used to cancel a term in an arithmetic expression if a condition exists in which the term is not wanted. The method for doing this will be explained later, in the discussion of "truth functions."

The complete list of arithmetic operators and their meanings is given in the following table:

Operator	Meaning
+	Addition
−	Subtraction or Negation
*	Multiplication
/	Division
**	Exponentiation
ABS	Absolute Value (i.e., the value of a number treated as if the sign were positive)
TR	Truth Value (see the discussion of truth functions)

Conditional Expressions

A conditional expression is an expression which, taken as a whole, may be either true or false, depending on conditions existing when the expression is examined. Generally, a conditional expression contains at least one variable quantity, and the truth or falsity of the expression will depend on the particular value assumed by the variable or variables. For example, the expression A IS GREATER THAN 10 is conditional, since it may or may not be true, depending on the value of the quantity A. Obviously, if A had a value of 12, the expression would be true; if the value were 7, the expression would be false.

A conditional expression may contain data-names, condition-names, arithmetic expressions, and expressions which show relationships between values (such as the expression IS GREATER THAN). Conditional expressions may be joined together by the words AND and OR to form compound conditional expressions.

Conditional expressions may be of either of two principal types: (1) relations, and (2) condition-names.

Relations

A conditional expression may contain an expression that shows a relationship between values. The example given above, A IS GREATER THAN 10, illustrates the general concept. Six basic relational expressions may be used in conditional expressions, each of which may be written in a full form or in an abbreviated form. They are as follows:

IS GREATER THAN	or	GT
IS NOT GREATER THAN	or	NOT GT
IS EQUAL TO	or	=
IS NOT EQUAL TO	or	NOT =
IS LESS THAN	or	LT
IS NOT LESS THAN	or	NOT LT

These expressions may be used to connect data-names, literals, and arithmetic expressions. The following examples indicate typical uses of the relational expressions:

```
BEGINNING.ON.HAND + RECEIPTS − SHIPMENTS IS
  LESS THAN REORDER.POINT
AGE GT 21
A * (B + C) − (D / E) = 500
DEPENDENTS NOT = 0
A GT B OR A = C
```

Condition-Names

In many cases, a record can be processed in one of several ways, depending on certain characteristics of the record or the existence of a particular condition at the time it is processed. For example, assume that a company is using a file of personnel records to prepare a report on employees' dependency status. The records of single employees will probably be processed in a different manner from those of married employees. It might also be essential to distinguish between married and divorced employees. Thus, each record must be examined individually to determine marital status. Marital status would normally be indicated by a code of some kind, and, for the purposes of this illustration, it will be supposed that the initials M, S, and D indicate the classifications (i.e., the "conditions") "married," "single," and "divorced," respectively.

One of these initials will appear in each record, and the programmer must reserve a place in storage for this code by giving a general name to the area in which the code is to be placed. This is done by writing an entry in the data description, as explained in Chapter 4. (See, in particular, the discussion beginning on page 71, in which the reader is shown how a data description for this example may be written.) Assume that the general name used is MARITAL.STATUS. This is a data-name, and it may be used in procedure statements in any of the ways in which data-names may be used. However, in this case, it is desired to name, in addition, the specific values which this data-name represents. Specifically, the programmer will wish to be able to refer to the initials M, S, and D.

If he wishes, he may write a relational expression such as has been shown above. Thus, MARITAL.STATUS = 'M' is a relational expression, and it may be used in such an instruction as IF MARITAL.STATUS = 'M' THEN DO TABULATION.ROUTINE.A. (Note, incidentally, that the initial M is placed in quotation marks to show that it is a literal, for this code will appear literally in the record.)

However, it is often simpler to treat the appearance of a particular code in the assigned area as a *condition*; this condition can then be given a *condition-name* which signifies that the condition is present. The condition-name MARRIED, for example, could then be used to indicate the presence of the initial M. Condition-names are assigned in the data description, and the reader is referred again to Chapter 4 to see how this may be done.

Once a condition-name has been defined, it may be used directly in procedure statements. The condition-name actually serves as an abbreviation of a relational expression. In this example, the condition-name MARRIED is exactly equivalent to MARITAL.STATUS = 'M' and it may be substituted for it wherever the programmer wishes. The instruction mentioned above (IF MARITAL.STATUS = 'M' THEN DO TABULATION.ROUTINE.A) can then be reduced to IF MARRIED THEN DO TABULATION.ROUTINE.A.

Similarly, all of the other conditions which the field MARITAL.STATUS can assume are defined in the data description and given condition-names, assuming the programmer has need of them. (If the programmer needs to refer only to the condition MARRIED, he need not assign condition-names to the other conditions.)

The reader should note that the condition-name itself is a conditional expression in the full meaning of that term. It may be used in clauses and sentences in the same manner as any other conditional expression. The most general use of a condition-name is in a clause of the IF . . . THEN type, such as IF MARRIED THEN, or IF SINGLE THEN. This construction will be explained in the discussion of conditional clauses later in this chapter.

AND, OR, and NOT

The Commercial Translator system is fully capable of interpreting and processing "compound conditions"—i.e., conditional expressions which are themselves composed of two or more conditional expressions.

Suppose the following expressions are being used as conditions:

MARRIED
OVER.21
HOURLY.RATE IS GREATER THAN 3.50
HOURLY.RATE IS LESS THAN 5.00

These expressions may be joined in any combination the programmer might wish, using the words AND and/or OR. In some cases, it will also be necessary to enclose one or more pairs of conditions in parentheses. The negative form of a conditional expression may also be used; this is obtained by placing the word NOT before the expression.

The interpretation of conditional expressions is governed by four rules:

1. The word OR is interpreted as "either or both." In other words, it is used in the "inclusive" sense employed in formal logic.
2. The word AND is interpreted as "both." Simple conditions joined by AND must both be true in order for the compound condition to be true.
3. Each conditional expression must be completely stated. For example, the system would not accept such an expression as HOURLY.RATE IS GREATER THAN 3.50 AND LESS THAN 5.00. To convey the intended meaning, the full expression must be written as HOURLY.RATE IS GREATER THAN 3.50 AND HOURLY.RATE IS LESS THAN 5.00.
4. When a compound conditional expression contains both the word OR and the word AND, it will be interpreted as if the terms connected by AND were enclosed in parentheses. In other words, each pair of conditions joined by the word AND will be considered as a single condition which is true if both of the subordinate conditions have been met. If a contrary sense is intended, parentheses must be used to show it.

The following examples should clarify the interpretation of these rules:

Compound Conditional Expression	**Interpretation**
MARRIED AND OVER.21	Both of the conditions MARRIED and OVER.21 must be met.
MARRIED OR NOT OVER.21	Either the condition MARRIED or the condition NOT OVER.21, or both, must be met.
NOT MARRIED AND HOURLY.RATE IS NOT LESS THAN 5.00	Both of the conditions NOT MARRIED and HOURLY.RATE IS NOT LESS THAN 5.00 must be met.
MARRIED OR OVER.21 AND HOURLY.RATE IS GREATER THAN 3.50	Either the condition MARRIED must be met or the combination OVER.21 AND HOURLY.RATE IS GREATER THAN 3.50 must be met, or both.

Compound Conditional Expression	Interpretation
(MARRIED OR OVER.21) AND HOURLY.RATE IS GREATER THAN 3.50	Either the condition MARRIED or the condition OVER.21 must be met, or both, and, in addition, the condition HOURLY.RATE IS GREATER THAN 3.50 must also be met.
MARRIED AND OVER.21 AND HOURLY.RATE IS GREATER THAN 3.50	All three conditions must be met.
MARRIED OR OVER.21 OR HOURLY.RATE IS GREATER THAN 3.50	The compound condition will be met if any one of the three conditions, or any two, or all three conditions, are met.

Truth Functions

It has been pointed out that conditional expressions may be used in arithmetic expressions in connection with the "truth operator" TR. When the truth operator is used, as explained below, it converts the conditional expression into an arithmetic expression having a value of either 1 or 0. This value can then be used normally in the arithmetic expression.

The conditional expression is placed in parentheses and the operator TR is placed immediately in front of it. The resulting term is called a *truth function*. The truth operator signals to the system that it must make a test to determine the truth or falsity of the conditional expression. If the condition is found to be true, the truth function as a whole is automatically assigned a value of 1; if the condition is false, the truth function is given a value of 0.

Suppose that a manufacturer agrees to give a discount of five per cent on bills for purchases of more than one thousand dollars. If the amount of the purchase is to be found in a field called ORDER.AMOUNT, the programmer could write such a statement as:

```
SET DISCOUNT = ORDER.AMOUNT * .05 * TR
  (ORDER.AMOUNT IS GREATER THAN 1000).
```

The conditional expression ORDER.AMOUNT IS GREATER THAN 1000 will then be examined. If it is found to be true, the whole truth function will be replaced by a value of 1; in this case the discount would be computed as five per cent of the value in the ORDER.AMOUNT field. If the conditional expression is found to be false, the truth function will be given a value of 0, and the net effect would be to cause the discount to be computed as 0.

The truth operator may be used with relational expressions, as in the example given, or with condition-names.

Clauses

The basic components of the Commercial Translator language have now been discussed. The remainder of this chapter will show how these components can be combined to express meanings.

The basic complete unit of meaning in the Commercial Translator language is the *sentence*. Sentences, however, are composed of one or more shorter units, known as *clauses*. There are two kinds of clauses: imperative and conditional.

Imperative Clauses

An imperative clause is a group of words that expresses a complete command. The clause may or may not include symbols. It always begins with a verb and may contain one or more operands of the verb, as appropriate. The operands may include data-names, procedure-names, condition-names, literals, figurative constants, and arithmetic expressions. Following are several examples of imperative clauses:

```
OPEN INPUT.MASTER.FILE
GET PAY.RECORD
ADD 1 TO COUNTER
MOVE ZEROS TO AMOUNT.FIELD
GO TO TAX.CALCULATION
SET NET.PAY = GROSS.PAY - (GROSS.PAY
  - (EXEMPTIONS * 13)) * .18 - OTHER.DEDUCTIONS
```

Conditional Clauses

A conditional clause consists of a conditional expression introduced by the word IF and terminated by the word THEN. The conditional expression may be simple or compound. The word IF is an operator which informs the system that the conditional expression which follows must be tested for truth or falsity. The word THEN defines the limit of the conditional expression to be tested; it will always be followed by an imperative clause.

Following are several examples of conditional clauses:

```
IF OUT.OF.ZONE THEN
IF AMOUNT GT 200 THEN
IF MARRIED AND OVER.21 THEN
IF A = B OR A = C AND A GT D THEN
```

Sentences

A Commercial Translator sentence corresponds to a sentence in English—it expresses a complete and independent thought. It must contain at least one imperative clause and may contain, in addition, one conditional clause and one or more additional imperative clauses. It is terminated by a period, which must be followed by a blank to distinguish it from the "imbedded period" used in names and from the decimal point used in numeric literals. Imperative clauses within the same sentence must be separated by commas.

When a conditional clause is used in a sentence, it must begin the sentence, and it must be followed by one or more imperative clauses to be executed if the prescribed condition is met. If the programmer wishes to prescribe alternative action to be taken if the conditional expression proves false, he may specify it by writing next (without intervening punctuation) the word OTHERWISE, followed by one or more imperative clauses. If the conditional expression should prove false, and if the sentence does not contain the word OTHERWISE, the conditional sentence will cause no action and the system will proceed to the next sentence in the program.

The following examples show how Commercial Translator sentences may be constructed:

```
ADD 1 TO COUNTER.
GO TO TAX.CALCULATION.
STOP.
IF MARRIED THEN MOVE NAME TO MAILING.LIST.M,
  ADD 1 TO COUNTER.
IF GROSS.PAY LT NET.PAY THEN GO TO ERROR.ROUTINE.
IF OUT.OF.ZONE THEN DO BILLING.ROUTINE.A OTHERWISE
  DO BILLING.ROUTINE.B.
```

Sentences may be assigned procedure-names, so that the programmer can make reference to them in other parts of the program. It will be seen that this provision makes it possible to carry out any sequence in the program whenever it is desired, without having to rewrite it each time. Suppose, for instance, that the last sample sentence given above, which begins IF OUT.OF.ZONE . . ., had been given the name DISTANCE.CHECK. Whenever the programmer wishes to make this check and take appropriate action, he can instruct the program to "transfer" to this sentence by writing either DO DISTANCE.CHECK or GO TO DISTANCE.CHECK. (This will be explained more fully in Chapter 3.) The actual method of assigning names will be discussed in Chapter 3 under the description of the columnar form used in writing procedure statements.

Sections

A section consists of one sentence, or a group of successive sentences, which has been given a name for reference purposes in accordance with the rules given below.

At first sight, it may appear that a Commercial Translator section corresponds closely to a paragraph of English. However, there are several distinctions. Sentences are *not* grouped into sections for the purpose of clarifying the logic or the structure of the program for the benefit of the system, whereas paragraphing in English does perform such a service for the reader. Sectioning, in the Commercial Translator, is used for the purpose of naming portions of procedure.

For example, suppose the programmer has written a sequence of sentences to calculate compound interest. Instead of having to write out this sequence each time the calculation is required, he can group the sentences into a section and give it a name by which it can be referred to elsewhere in the program. If this name were INTEREST.ROUTINE, for instance, he could write such a clause as DO INTEREST.ROUTINE at any point in the program, and this instruction would cause, first, a transfer to that routine and, second, a transfer back to the original point when the routine has been carried out. (Details will be found in the discussion of the DO command in Chapter 3.) Grouping of sentences into sections, then, is a device for identifying such a group for further reference.

The beginning of a section must be identified by a procedure-name, followed by a period. The name must then be followed by the words BEGIN SECTION, in accordance with the rules for using that verb, as given in Chapter 3. Following the last sentence in the routine, the programmer must write the word END, followed by the procedure-name used originally to identify the section.

Sections may be "nested," that is, one or more sections may be contained within a larger section, but each such section must be wholly contained—it cannot overlap another section.

Where necessary, the names of sections can be used as parts of compound names.

Divisions

All Commercial Translator programs are composed of three divisions, the *Procedure Description,* the *Data Description,* and the *Environment Description.* The first of these contains the procedure statements of which the program is composed. The second provides the processor with information about the data to be used in the object program. The third is used to make certain technical connections between the program and the machine system on which it will be run; these details are dependent on the particular system and are therefore discussed in the publications covering the processors for the various systems.

It is not necessary that these three divisions appear as separate entities. If appropriate, the programmer may write a portion of one, then a portion of another, and so on. However, each such portion must be properly labeled with a *division header*. Each header consists of the name of the division, preceded by an asterisk. The three division headers are:

*PROCEDURE
*DATA
*ENVIRONMENT

These names are written, where required, in the "name margin" of the form used for the procedure description, as explained in Chapter 3, or in the name columns of the form used for the data description, as explained in Chapter 4. In other words, the asterisk always appears in the left-most name column, followed immediately by the remainder of the header.

All entries following a division header are assumed to be a part of the specified division.

Punctuation and Spacing

The punctuation of Commercial Translator sentences is reasonably simple and, for the most part, self-evident. It may be summarized in the following rules:

1. All words are separated by blanks, or by any character which cannot legally be used in a word, such as an arithmetic operator. (See the rules for forming names, beginning on page 15.)

2. Multiple blanks are treated as single blanks, except within alphameric literals, where each blank will be treated as a separate character. (Numeric literals cannot contain any blanks.)

3. Each sentence must be terminated by a period, followed by a blank. Should the blank be omitted, the period will be treated as an "imbedded period," except where the period is found in Column 72 of the procedure description form.

4. The "imbedded period"—i.e., a period surrounded by non-blank characters—serves one of two functions, depending on the context: (1) If it appears in a "word" consisting *wholly* of numerals, or in a floating point number, it will be treated as a decimal point. (2) In any other context (except within a literal), it will serve as the equivalent of a hyphen—in other words, as the means of connecting separate parts of a simple name.

5. Successive imperative clauses in a sentence must be separated by single commas; for the sake of clarity, the programmer may use blanks in addition, but they will be ignored.

6. Arithmetic operators may be used with or without surrounding blanks. Blanks may be used for the sake of clarity, as in the expression A + B * C, but they will be ignored; this expression could therefore be written A+B*C. However, the programmer must not write two successive arithmetic operators unless the second of them is either the operator TR or the operator ABS. Where the effect of two successive operators is required, one term may be enclosed in parentheses; thus, while the expression A * −B is illegal, the same value may be written A * (−B). The minus sign in this case could have been followed by a blank, but the notation given is customary.

7. The rules of punctuation and spacing do not apply within literals, which may contain any character except the quotation mark, or within constants defined in the data description, which may contain any character. Alphabetic and alphameric literals, and all named constants, must be enclosed in quotation marks. (See the rules governing literals and named constants, beginning on page 18.)
8. Floating point numbers are written as numeric literals. (See the discussion of numeric literals, beginning on page 18, and also the rules for writing floating point numbers in the data description, on page 80.)
9. Parentheses must be used to group together two or more terms which are to be acted on by a single arithmetic operator, in accordance with the rule that all operators act on the next named item, or the next parenthetical expression, following the operator.
10. Subscripts must be enclosed in parentheses, and if more than one subscript is used within the same parentheses, they must be separated by single commas. (See the discussion of lists, tables, and subscripts beginning below.)
11. Parentheses may be used wherever needed in arithmetic expressions and in compound conditional expressions for the sake of clarity; where ambiguity would result from their omission, they *must* be used.
12. Division headers must begin with single asterisks.
13. Punctuation and spacing associated with any particular verb will be found in the general format prescribed for the verb in Chapter 3.
14. When it is necessary to carry over an item from one line to another on either the procedure description or data description forms, selection of the "break point" must follow the rules given for those forms. In general, it should also be understood that a blank is assumed to follow Column 72 of the procedure description form and Column 71 of the data description form.
15. When a function is named as an operand in a procedure statement, the names of the data to be substituted for the parameters must be placed in double parentheses immediately following the function-name. These data-names must be separated by single commas. (See the discussion of functions beginning on page 32.)

Lists, Tables, and Subscripts

Just as a clerk may use a reference table to obtain data, the programmer can direct the program to look up data in a list or table stored within the data processing system. The programmer must therefore know how to place the table in storage initially and how to write instructions for locating data in it.

A list, or table, may be regarded as an ordered grouping of data. The actual data may be either fixed or variable, depending on the need. If it is fixed, it is usually entered into the system as a series of constants; if it is variable, it will be either placed in the system when the input records are read in or produced as a result of some operation performed within the system.

In either case, it is necessary to establish the format of the table in such a way that the system can locate individual items of data within it. Thus, placing a table in the system requires two steps: (1) establishing the structure and format of the table, and (2) making provisions to enter the required data in that structure. When this is done properly, the programmer may call for any item of data in the table as required.

The actual methods of storing a table for use are described in Chapter 4, since data description entries are required. To illustrate the general principles involved in preparing and using a table, however, the following example may be helpful:

Suppose that the programmer wishes to be able to refer to a table of passenger transportation rates and that the general form of this table can be represented by the following excerpt:

City	One-Way	Round Trip	Excursion
. . .	. . .	. . .	. . .
Los Angeles	153.42	285.16	212.87
Miami	78.60	141.63	118.92
. . .	. . .	. . .	. . .

When a table of this kind appears on a printed page, it is seen to have a grid-like structure, consisting of vertical columns intersected by horizontal lines. Such a structure cannot be placed in storage in this form, since data is fed into the system in a continuous stream. However, the table can be read line by line, one line succeeding another, until the entire table has been placed in the system in the form of a long list of data. In fact, several tables of the same type can be entered into the system in succession, as if they made up a single "three-dimensional" table. Thus, in the example given, there might be one rate table for the vacation season, another for the "off season," and so on. The two- or three-dimensional structure of the table is preserved by the use of "level numbers," as explained in Chapter 4, but the total mass of data would appear in storage as one long "string" of data.

It follows that a simple list of items may be thought of as a one-column table, or that a table, no matter how complex, may be thought of as a special form of list. In the example given, an essentially two-dimensional table has been reduced to a list, called RATE.TABLE, consisting of, say, 30 lines, each called RATE, and each containing the four items CITY, ONE.WAY, ROUND.TRIP, and EXCURSION; these data-names correspond to the column titles in the example.

In this form, the data is not usable until some means has been established for locating each individual item. This is done initially by arranging the data so that each line consists of exactly the same number of character spaces, with each item in the line occupying a position that corresponds exactly to the position of the corresponding item in each other line. Each item can then be located by its position in storage, which, as has been pointed out, is the basic principle by which all data in storage is located.

It is then necessary to find a way of identifying each position and each line so that any particular item can be located. A method for doing this is explained in Chapter 4, in connection with the sample table previously mentioned. The actual method used depends on the principle of counting. Continuing with the example given, suppose Miami is the 17th city listed in the table. If the programmer wished to obtain the one-way rate for Miami, he would have to find a way of indicating to the system that it must find the 17th line of the table and then locate the second entry on that line. Obviously, there must be some means of relating the name Miami to the number 17.

This may be done in several ways. In most data processing operations, the normal way would be to show the relation externally by writing a series of code numbers corresponding to the names. Only the code numbers themselves would enter the system. For the counting principle to operate correctly, the numbers would have to be assigned sequentially and in the same order as the corresponding lines of data in the table.

These code numbers would then be used in the input records in lieu of the actual names. Naturally, they would have to be placed in a field reserved for them, so that they could be properly identified. Suppose that in this case the field were called DESTINATION. If the programmer then wrote a procedure statement containing the data-name DESTINATION, the system would look at that field to see what number it contained. This number could then be used as a means of referring to the table.

Subscripts

Numbers or names used to locate items in a list or a table are known as *subscripts.* In order to use them, the programmer must place them within parentheses following the data-name showing the kind of data being sought. Thus, since the table of this example contains a data sequence called RATE, the instruction MOVE RATE (DESTINATION) TO LIST.A would cause the system to obtain the number in the DESTINATION field, use this number to locate the corresponding line of the table, and move the entire contents of the line to the area called LIST.A. In this case, the data moved would include all items contained on that line, including CITY, ONE.WAY, ROUND.TRIP, and EXCURSION.

Often, however, only one of these items would be required. As will be seen in Chapter 4, the data description entries used to place the table in storage permit the programmer to locate individual items on each line. If these entries have been properly made, the programmer can write such a statement as MOVE ONE.WAY (DESTINATION) TO BILL.AMOUNT, and the system would then locate the line having the number indicated in the DESTINATION field, single out the particular part of the line called ONE.WAY, and move that one item to the area called BILL.AMOUNT.

It is conceivable that the programmer might wish to specify that the system obtain a rate for a particular city, rather than the rate for whatever city happened to be indicated in the DESTINATION field. Once again, correspondence between the name and the number would have to be established. This could be achieved by writing a data description entry in which the name of the city was specified as the name of a literal number. Thus, MIAMI might be specified as the name of the constant value '17'. Then, if the programmer wrote MOVE RATE (MIAMI) TO LIST.A or MOVE ONE.WAY (MIAMI) TO BILL.AMOUNT, the system would recognize that the name MIAMI was equivalent to the number 17 and would use that number as a means of referring to the table.

It has been shown how a single subscript may be used to identify an item in a table. Actually, as many as three subscripts may be used within the same pair of parentheses. To take a simple example, suppose that the contents of a book had been stored within the system and that the programmer wished to make reference to a particular word in a particular line on a particular page of the book. Suppose, further, that the data-names WORD, LINE, and PAGE had been properly entered in the data description in such a way that entries called WORD were subordinate to entries called LINE, which, in turn, were subordinate to entries called PAGE. (As Chapter 4 explains, level numbers are used for this purpose.)

To specify the 4th word in line 10 of page 150, the programmer could then write either PAGE (150) LINE (10) WORD (4) or PAGE LINE WORD (150, 10, 4). These two expressions are equivalent. In the first case, each data-name is subscripted individually. In the second case, it is the name WORD which is subscripted, but this name has been compounded by the use of the names of two higher levels, PAGE and LINE, to make it unique. If the name WORD had been unique in the program—i.e., had not been used outside of this particular sequence—the desired item could be indicated simply as WORD (150, 10, 4).

When more than one subscript is included within the same pair of parentheses, they must be separated by single commas. The number of subscripts used must cor-

respond to the number of subdivisions—i.e., the number of "levels"—of the table required to obtain the item. For example, if the programmer wished to obtain *all* of the words on line 10, it would be sufficient to write either PAGE (150) LINE (10) or PAGE LINE (150, 10). If the name LINE were unique, it would suffice to write LINE (150, 10).

A subscript may consist of a single name representing a variable quantity (such as a data-name), or a literal, or it may consist of an arithmetic expression of the form

a * VARIABLE ± b

in which the quantities a and b are literals and the name VARIABLE is the name of a field which may contain a variable quantity. This name may be a compound name, but, whether simple or compound, it must not have a subscript. Condition-names may not be used in subscripts. A name, literal, or arithmetic expression used as a subscript must represent a positive integral value.

Following are examples of subscripted names:

Subscripted Name	**Comment**
PAGE LINE WORD (150, 10, 4)	The subscripts actually apply to the name WORD, but it is assumed that this name is not unique and has been compounded by the use of the names PAGE and LINE.
WORD (150, 10, 4)	The name WORD is assumed to be unique, but it is part of a larger structure having three levels which must be identified by subscripts.
RATE.TABLE ONE.WAY (2, DESTINATION, 4)	It is assumed that there are several different tables called RATE.TABLE, each having the same structure. The 2 indicates the second table of this series, the name DESTINATION is the name of a field in the table, and the code 4 is a literal code which identifies a further subdivision of the table—e.g., "family plan, 4 persons."
ITEM (BASE + 1)	This subscript obtains the value following the one obtained by the notation ITEM (BASE). This construction may be useful in such procedures as interpolating between successive items in a table. Usually (in this particular kind of procedure) both items would be obtained and both would be used as inputs to some interpolating procedure.
ITEM (4 * AMOUNT)	In this case, the value of AMOUNT would be obtained and it would then be multiplied by the factor 4 and used as the subscript. Such a subscript could be used to obtain every fourth value in a table, assuming the value in the AMOUNT field is increased by 1 on each repetition of the routine.

Functions

While the term "function" is often associated with mathematics, the concept has certain general applications that can greatly simplify the writing of programs. Accordingly, the Commercial Translator system has been designed to handle certain kinds of functions as an added convenience to the programmer.

The term *function* is used in the Commercial Translator to mean a *result* obtained as a consequence of some procedure. More precisely, it means a result obtained from a procedure specified by a BEGIN SECTION command, and, in particular, it is a result named in the GIVING clause of that command.

For example, in the command BEGIN SECTION USING A, B, C GIVING MINIMUM, the name MINIMUM is a *function-name*. This name refers to a field in storage which will contain the function after the procedure specified by this command has been carried out. (A hypothetical procedure of this type will be described and explained presently.)

The reader will note that this command also contains a clause beginning with the word USING and that three data-names (A, B, and C) have been specified in that clause. These names represent data which will be used in obtaining the function. The three items of data are called *parameters,* and the three names are *parameter-names.* The term "parameter" is limited, in the Commercial Translator system, to data named in the USING clause of the BEGIN SECTION command.

Each of these names—the function-name and the three parameter-names—identifies a field in storage. At object time the three parameter fields will contain data to be used in the procedure, and the rules governing the DO command (which causes the procedure to be executed) provide a means by which the data will be placed automatically in those fields.

To illustrate the use of functions and parameters, a sample procedure will be examined in detail. Suppose that the programmer wishes to determine which of three values is the lowest, so that he can use it elsewhere in the program. Suppose, further, that he wishes to be able to compare values of different kinds and therefore wishes to use a procedure that can be used with data from a variety of sources. For this purpose he has written a procedure called MINIMUM.ROUTINE, which consists of the following procedure statements:

```
MINIMUM.ROUTINE.  BEGIN SECTION USING A, B, C GIVING MINI-
      MUM.
    IF A IS LESS THAN B THEN MOVE A TO MINIMUM OTHERWISE
      MOVE B TO MINIMUM.
    IF C IS LESS THAN MINIMUM THEN MOVE C TO MINIMUM.
    END MINIMUM.ROUTINE.
```

The name MINIMUM.ROUTINE identifies this procedure so that the programmer can refer to it elsewhere in the program. For purposes of illustration, it will be assumed that the programmer has written the following command later in the program:

```
DO MINIMUM.ROUTINE USING D, E, F GIVING MINIMUM.2.
```

It is important to see precisely what happens in this case. First of all, when the processor encounters the MINIMUM.ROUTINE section as it is originally written, it will note the three parameter-names (A, B and C) and the function-name (MINIMUM). It will examine the entries written for those names in the data description (which will be discussed in Chapter 4) and will obtain information about the amount of storage space to be reserved for each, together with certain other technical details.

The net effect will be that *space* in storage is reserved for the actual data to be used and the actual result to be obtained. The values themselves will be obtained at object time when the program encounters the DO MINIMUM.ROUTINE command.

The reader will have noted that this DO MINIMUM.ROUTINE command, like the BEGIN SECTION command, contains USING and GIVING clauses. These clauses name the values to be used in the MINIMUM.ROUTINE procedure and the result to be produced. When the system encounters this command at object time, it will act as follows:

First, it will examine the field specified by the first data-name (D). It will obtain the value found there and move it to the first field named in the BEGIN SECTION command—i.e., field A. It will then obtain the value in field E and move it to field B. Similarly, it will move the value found in field F to field C. This process of data substitution follows the rule that items of data named in a DO command will be placed in the parameter fields named in the BEGIN SECTION command in the order in which they are named.

Once the actual data has been placed in the parameter fields, the MINIMUM.ROUTINE procedure will be carried out, using the data which has now been placed in the parameter fields. If fields D, E, and F had originally contained values of 11, 7, and 8, respectively, field A will now have a value of 11, field B a value of 7, and field C a value of 8. The procedure will then operate as follows: (1) The system will examine fields A and B, obtain the values they contain, and compare them. Since in this case the value of B is smaller, it will be moved to the field called MINIMUM. (2) The system will then examine field C and the MINIMUM field, note the values contained in them, and compare them. In this case, the value in the MINIMUM field is the lower. Thus, since the condition IF C IS LESS THAN MINIMUM has *not* been met, the instruction MOVE C TO MINIMUM will be ignored. (The lowest of the three values, of course, is already stored in the MINIMUM field.) (3) The system will then note the name of any field specified in the GIVING clause of the DO command. In this case, there is such a name, and it is different from the function-name given in the BEGIN SECTION command. In such a case, the value in the latter will be moved to the former. Thus, the value 7 will now be placed in the MINIMUM.2 field. (4) The command END MINIMUM.ROUTINE defines the end of the procedure, and the system will then return to the command following the DO instruction.

The reader will see, thus, that the effect of the USING and GIVING clauses in the DO command is to cause a series of data movements. Prior to the execution of the procedure, data named in the USING clause of the DO command will be moved to the parameter fields named in the USING clause of the BEGIN SECTION command. After the procedure has been carried out, the results will be moved from the function fields named in the GIVING clause of the BEGIN SECTION command to the data fields named in the GIVING clause of the DO command.

It is not necessary to employ the USING clause each time a DO command is written, if the programmer wishes to use the values already placed in the parameter fields. However, since the procedure will operate with whatever values are found in the parameter fields, the programmer must be sure that no unintended values are left there. In some cases, it may be useful to place "high values," "low values," blanks, or zeros, in a field; the figurative constants, thus, can be used as data-names.

Similarly, the GIVING clause need not be written in a DO command if the programmer intends to refer to the result (or results) as placed in the function field (or fields) named originally in the BEGIN SECTION command.

As the reader has seen, the use of the BEGIN SECTION and DO commands makes it possible to set up basic procedures and to use them with many different kinds of data. If the procedure is properly specified by the use of a BEGIN SECTION command,

the programmer may use this procedure for any number of subsequent operations, using data from different fields. The MINIMUM.ROUTINE procedure used in this example might be used in such different ways as the following:

DO MINIMUM.ROUTINE USING CALCULATED.PRICE, MARKET.PRICE, HIGH.VALUES GIVING PRICE.

DO MINIMUM.ROUTINE USING RAIL.EXPRESS, AIR.FREIGHT, PARCEL.POST GIVING SHIPPING.RATE.

DO MINIMUM.ROUTINE USING FLAT.RATE, QUANTITY.RATE, HIGH.VALUES GIVING RATE.

Note the use of the figurative constant HIGH.VALUES as a data-name in two of these examples. In these cases the figurative constant would cause the highest values in the machine's collating sequence to be placed in the indicated parameter field. Since the programmer is concerned with choosing the lower of two values, this procedure assures that no unwanted lower value is found in the third parameter field.

Use of Functions in Procedure Statements

The reader has now seen how the same procedure may be used to obtain different results by processing different data obtained from different fields. The result (i.e., the function) will always be located in a specified field, and thus it may be obtained for use in other procedures. In other words, the function-names of the BEGIN SECTION command may be used like any other data-names, once the procedure has been performed.

However, a still shorter method is available. Instead of having to write a DO command which orders a procedure to be carried out, the programmer may write the name of a function directly in a procedure statement, together with the names of the data items to be substituted for the parameters, and the system will carry out the BEGIN SECTION procedure just as if the DO command had been written. In order to achieve this result, the programmer must specify the data to be used by placing the data-names in double parentheses immediately after the function-name. These names must be separated by single commas.

When this technique is used, the function-name itself must be specified. Since no DO command is used, there is no way of directing that the function be moved from the function field to another field; thus, it can be obtained only from the former.

Referring once again to the example of the MINIMUM.ROUTINE and to the three examples above, the programmer could write such statements as the following:

MOVE MINIMUM ((CALCULATED.PRICE, MARKET.PRICE. HIGH.VALUES)) TO PRICE.LIST.

SET SHIPPING.COST = MINIMUM ((RAIL.EXPRESS, AIR.FREIGHT, PARCEL.POST)) * QUANTITY.

SET RATE.FACTOR = MINIMUM ((FLAT.RATE, QUANTITY.RATE, HIGH.VALUES)) * 1.15.

All function-names and all parameter-names used in the program must be written in the data description, as explained in Chapter 4. In particular, they must be identified by the type codes FUNCT or PARAM, as appropriate, and they must be described in accordance with the rules governing data description entries.

Chapter 3: Procedure Description

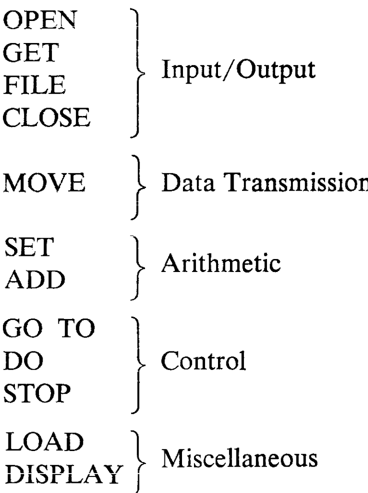
Introduction

This chapter is concerned with the verbs and commands which are presently a part of the procedure vocabulary of the Commercial Translator language. The verbs fall into two main categories. Most of them are used in commands that state the data processing steps to be performed, and thus they are called "program" verbs. The other category includes the "processor" verbs; these verbs are used in the processor-directing commands.

Verbs

The list of verbs is given below. The organization of this chapter is based on the classification shown in the list:

Program Verbs



Processor Verbs

OVERLAP
BEGIN SECTION
END
INCLUDE
CALL
NOTE
ENTER

The Commercial Translator system is designed as an "open-ended" programming system. That is, the list of verbs and associated commands is never closed. The processors are so constructed as to permit the introduction of new verbs and commands when required.

Commands

Most of the discussion in this chapter is concerned with commands. Each verb in the preceding list forms the basis for a particular type of command; accordingly, the various commands can be classified as either program commands or processor commands. A command normally consists of a verb followed by one or more operands and may be written as an imperative clause or as a sentence.

COMMERCIAL TRANSLATOR PROCEDURE DESCRIPTION

CTL. 1 3	PROGRAM	SYSTEM	PAGE OF
	PROGRAMMER	DATE	IDENT. 73 80

SERIAL 4 6	PROCEDURE NAME 7 12	TEXT 13 72

Format of Procedure Statements

The clauses, sentences and sections that constitute the procedural portion of a Commercial Translator source program are written in "free-form" text on a programming form designed for that purpose. The form is shown on page 36; it has just four fields, which are as follows:

Ctl. and Serial (Col. 1-6)

Though separated on the form, the six spaces of "Ctl." and "Serial" are actually one field; the information entered in the first three spaces is the same for all lines on one page. This area of the form is used to indicate card sequence by means of numbers and/or alphabetic characters. The information will be sequence-checked by the processor. (Source programs are read by the processor in the form of a deck of punched cards or their equivalent on tape; each line of the programming form becomes one card in the source program deck.) The right-most space of this field, i.e., Column 6, is usually left blank or else made zero so that subsequent inserts can be numbered sequentially. For example, if the serials on a page are 010, 020, 030, etc., cards numbered 011, 012, . . . 019 can be inserted later between 010 and 020.

Procedure Name (Col. 7-12)

Columns 7-12 of the form are reserved for procedure-names, i.e., the names of sentences or sections. It is convenient to speak of this field as the *name margin* of the "Text" (see below) and to think of the procedure-names as "sticking out" into this margin.

The rules for procedure-names are as follows:

1. The procedure header, *PROCEDURE, is a special name that must appear on a separate line preceding each procedural portion of a source program to distinguish it from data description or environment description.
2. Procedure-names, like data-names, may contain up to thirty characters. Unlike data-names, however, they must be followed by a period and a blank.
3. Procedure-names are written left-justified in the name margin and, if longer than six characters, extend into the "Text" area.

Text (Col. 13-72)

The procedure statements of the program are written in the "Text" area of the programming form. A number of examples in this chapter, as well as the sample program in Appendix 1, illustrate the manner in which the procedure is stated. The clauses, sentences and sections of procedure are written in free form, subject to the following rules:

1. A named sentence follows its name. That is, it begins to the right of the name margin, on the same line as its name.
2. An unnamed sentence must begin on a separate line and to the right of the name margin.
3. Succeeding lines of a sentence must begin to the right of the name margin. (If desired, they may be indented to facilitate reading the text.)
4. When a section of procedure is begun, only the BEGIN SECTION command may appear on the same line as the name of the section.
5. To end a section, the command "END section.name" is written as a sentence, i.e., on a separate line.

Identification (Col. 73-80)

Columns 73-80 can be used to identify the program. The information entered here will be the same for all lines on the page. The use of this area of the form is optional.

Presentation of Commands and Examples

In the following description of the specific elements of the Commercial Translator procedure vocabulary, each command is represented in its "general form." In each case the general form is enclosed in a rectangular frame to distinguish it from text and examples. The verbs and other words which are a fixed part of the language appear in the general forms in boldface capital letters. The words and phrases representing names, clauses and sentences which will be written by the programmer are shown in lower-case italics. For reference purposes, the same general forms are presented in Appendix 2 in a more concise manner, without any discussion. For those commands that may contain a variable number of operands, a construction such as, "data.name.1, data.name.2, . . . data.name.n" is used to indicate that as many as "n" operands may be specified by the programmer. It is important to note that, except for imbedded periods within italicized words, any punctuation shown in the general form of a command is a fixed and necessary part of the command.

In contrast to the general forms, *examples* of the various commands are written completely in capital letters and are not enclosed in frames.

Program Commands

Each of the Commercial Translator program commands causes some event or series of events to take place at object time, that is, the time at which the object program is run. The program commands are discussed, in turn, in the following pages according to the classification shown on the first page of this chapter.

Input/Output Commands

Within the category of program commands it is helpful to consider the input/output commands as a subgroup. The control of data flow through the data processing system is accomplished by means of an input/output control system. This system is called into play when the programmer uses one of the input/output commands in a statement of the procedure to be executed. The four verbs associated with these commands are OPEN, GET, FILE, and CLOSE.

Using the input/output commands, the programmer initiates the movement of data into buffers or internal storage, the checking of the validity of the file itself, the checking of the validity of the input or output operation, the storage of data in internal storage to insure its availability when required, and finally, the making available or filing away of data according to the needs of the program. Thus the input/output control system provides data flow control and, where feasible, a "look ahead" at the data flow.

The input/output control system in the Commercial Translator is a record processing system. That is, the unit of data which is made available by the system and on which attention is focused during each processing cycle is the record. Should the needs of the program require that more than one record from a file be made available for processing at one time, it will be necessary for the programmer to provide working storage into which he will move the additional records as required.

The input/output control system provided in the initial version of the Commercial Translator is intended for the handling of tape and card files. The detection of errors is an implicit part of the tape and card handling system; error correction, where feasible, is handled automatically. More specific information on the use of the data flow control vocabulary is presented in the following paragraphs.

The OPEN Command

The OPEN command initiates the handling of one or more files. It may take either of two general forms:

```
OPEN file.name.1, file.name.2, ... file.name.n
OPEN ALL FILES
```

The first form of the command causes only the named file(s) to be opened. When the second form is used, all files specified in the environment description are opened. A given file must be opened, of course, before it can be addressed by a GET or FILE command.

If the file being opened is an input file, the following series of events occurs:

1. The label record, if any, is read into storage and checked for validity according to the standard label-handling conventions.
2. Subsequent records are brought into the portion of storage governed by the input/output control system, filling the area which has been allocated to the file.
3. Checking is performed, and a record count is initiated.

In the case of an output file the following events occur:

1. If specified by the programmer, a label record is created and written.
2. Preparation is made to file data records in the output file as they become available in processing.
3. Checking is performed, and a record count is initiated.

Some typical OPEN commands are:

```
OPEN INVENTORY.FILE.
OPEN ALL FILES.
OPEN STATISTICS, CUSTOMER.FILE, INVOICE.FILE.
```

The GET Command

The GET command is used to fetch records from an input storage area which is filled automatically from a file stored on tape or cards. The programmer need be concerned only with the use of single records since all auxiliary input operations such as

unblocking
tape alternation
tape identification
error checking
reading ahead

are automatically provided in the object program by the processor, based on information in the environment description.

The GET command may take either of two general forms:

```
GET RECORD FROM file.name
GET record.name
```

The first form of the command assumes that the specified input file contains more than one type of data record. The record obtained may be of any type, and the programmer must arrange to identify the type. The second form implies an input file containing only one type of data record; the record is obtained from the file associated with "record.name" in the environment description.

Either form of the GET causes the next record to be made available so that the entire record or any of its parts may be used in processing. Note that the previous record of the file is no longer addressable after the execution of a GET command.

To provide for the execution of an alternate command conditional upon end of file, the optional phrase,

..., AT END *any imperative clause*

may be appended to either form of the GET. A command thus specified is performed after the last record of a file has been made available for processing and a subsequent GET command has been encountered. The programmer should always use the AT END option if the possibility exists of reaching end of file upon execution of the GET.

When the GET command is executed at object time, the following events take place:

1. The next record of the file is made available for processing.
2. If end of tape is reached, the end-of-tape label is read, and checks are made. The input tape is rewound, and provision is made for an alternate tape unit to be substituted.
3. If end of file is reached, any alternate command specified in the AT END phrase is performed.

Some examples of the use of GET are:

```
GET RECORD FROM INVOICE.FILE.
GET MASTER, AT END GO TO END.OF.MASTERS.
```

The FILE Command

The FILE command is used to place records on tape (for subsequent on-line or off-line processing), on cards, or on the printer. The programmer is concerned only with the placing of the unit record. Other considerations, such as

blocking
tape alternation
file identification
error checking
setting checkpoints

are provided automatically by the data flow control system based upon information supplied through the environment description. There are two forms of the FILE command:

FILE *record.name* FILE *record.name* IN *file.name*

In the first form the record is filed in the output file with which it has been associated through the environment description. In the second form the named record is filed from storage to the output file even though that record may not have been hitherto associated with that file. Creating new records in working storage and then merging them into a master file is an instance of the latter situation.

When the FILE command is executed at object time, the following events take place:

1. The named record is added to the list of those awaiting write-out. When the proper number of records has been accumulated, writing occurs.

2. When the physical end of a reel is sensed, the end-of-reel label is prepared and written, arrangements are made for alternation of tape units, and the tape is rewound.
3. A count of the number of records written is maintained.

It should be noted that a record that has been filed is still available for further processing. It is entirely possible, for example, to file a record in each of several files by means of a succession of FILE commands.

Some examples of the FILE command are:

FILE MASTER.
FILE PAY.RECORD.
FILE DETAIL IN ERROR.FILE.

The CLOSE Command

The CLOSE command terminates the use of one or more data files. It may take either of two general forms:

```
CLOSE file.name.1, file.name.2, ... file.name.n

CLOSE ALL FILES
```

In its first form the CLOSE command causes only the named file(s) to be closed. With the second form, all files defined in the environment description are closed.

In the case of an input file the CLOSE command causes appropriate "housekeeping" operations such as:

1. The record count is compared with the count in the end-of-file label if label records are present and if end of file has been reached. If the count does not agree, notification is given through external display. If the tape is not at end of file the record count is ignored.
2. If the file is on tape, a rewind is initiated.
3. The storage area allocated to the file is released for the use of other files.

If the addressed file is an output file, operations such as the following occur:

1. Any remaining information belonging to that file is written.
2. If specified by the programmer, an end-of-file label containing the record count is written.
3. The tape is rewound (if the file is on tape).
4. The storage area is released for the use of other files.

Each file which has been opened must ultimately be closed.

Examples of the CLOSE command are:

CLOSE INVENTORY.FILE, STATISTICS.
CLOSE ALL FILES.

Data Transmission Commands

The transmission of data from one area of storage to another is implicit in the functioning of a number of the Commercial Translator verbs. For example, the SET verb requires the transmission of result data after the computation has been performed. Except for one verb, however, such transmission is incidental to the main purpose. It is the MOVE verb which has as its primary function the transmission, or movement, of data from one area of storage to one or more other areas. MOVE and its alternate form, MOVE CORRESPONDING, are used in writing the two data transmission commands, which are described in the following paragraphs.

The MOVE Command

The MOVE command calls for the movement of data from one area of storage to one or more other areas. Concurrent editing will occur automatically in certain cases depending on the format of the data as defined in the data description. The general form of the MOVE command is:

MOVE *data.name.1* TO *data.name.2, data.name.3, ... data.name.n*

The data specified by "data.name.1" is moved to the area of storage designated by "data.name.2" and to any other area(s) mentioned in the command (data.name.3, . . . data.name.n). As used in this command "data.name" may represent data at any level defined in the data description (see Chapter 4).

The following rules must be observed in writing MOVE commands:

1. Information from numeric fields may be moved to other numeric fields, to alphameric fields, and to report fields.
2. Information from alphabetic or alphameric fields may be moved only to other alphabetic or alphameric fields.

Some examples of the MOVE command are:

MOVE CURRENT.DATE TO CHECK.DATE.
MOVE ZEROS TO MONTH.TOTAL, YEAR.TOTAL, CUMUL.TOTAL.

EDITING FEATURE

Editing of the data in the sending area to conform to the format of the receiving area is a feature of the MOVE command. Such editing occurs automatically if an explicit format definition, i.e., a field pictorial, is given in the data description for both the "from" and the "to" areas. (The field pictorial is a convenient method of describing formats and is discussed in detail in the data description portion of the manual; see page 79.)

The following conventions are observed in the editing feature of the MOVE command:

Numeric Information

The data from the sending area is aligned with respect to the decimal point (assumed or actual) in the receiving area. Such alignment may involve the dropping of leading digits or low-order digits (or both if the sending field is larger than the receiving one). If the "from" area is smaller than the "to" area, the excess positions of the receiving area will be replaced by zeros.

Alphameric Information

If the sending area is larger than the receiving area, the data being moved will be left-justified and truncated; i.e., low-order characters will be dropped as may be necessary to make the data fit into the receiving area. If the sending area is the smaller, the data will be left-justified in its new location. The low-order positions of the receiving area, i.e., the excess positions, will be filled with blanks.

A few examples are included below to clarify the editing feature. Regarding the pictorials shown in each case, for the purposes of these examples it is sufficient to know that the digit 9 represents any digit, the character A any non-numeric character, the character X any character, and that V shows the position of the assumed decimal point.

Sending Area			Receiving Area		
Pictorial	Data before MOVE	Data after MOVE	Pictorial	Data before MOVE	Data after MOVE
99V99	1234	1234	99V99	8765	1234
99V99	1234	1234	99V9	876	123
9V9	12	12	99V99	8765	0120
AAAAA	JONES	JONES	AAAAA	VWXYZ	JONES
99999	01234	01234	999V9	7777	2340
AAA	RUN	RUN	AAAXX	JOB #2	RUN
AAAAA	BLACK	BLACK	AAA	RED	BLA

Note that in each case the data in the sending area remains unaltered after the MOVE has been executed. Note also, in the sixth example, that the information in the two excess positions of the receiving area is replaced by blanks.

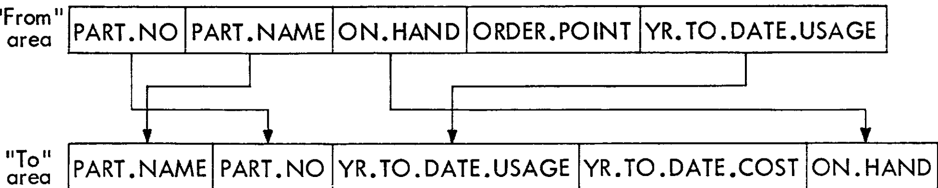
The MOVE CORRESPONDING Command

The MOVE CORRESPONDING command permits the programmer to move groups of data from one area of storage to one or more other areas. The general form of the command is:

MOVE CORRESPONDING *data.name.1* TO *data.name.2, data.name.3,*

... data.name.n

This command can be thought of as a series of MOVE commands, each of which is governed by the rules for MOVE. It is assumed, however, that each "data.name" represents an area of storage which is subdivided into smaller areas such as fields. The effect of MOVE CORRESPONDING is to move each field for which a corresponding field (i.e., a field having the same name) exists in the receiving area(s). The order of the corresponding fields need not be the same in the sending and the receiving areas. For example, the movement of data illustrated in the following diagram can be accomplished using MOVE CORRESPONDING:



Non-corresponding fields in the sending area are *not* moved. Non-corresponding fields in the receiving area remain unaltered.

EDITING FEATURE

Automatic editing of data to conform to the format of the receiving areas is done in essentially the same manner as in the MOVE command. Such editing, however, operates on subdivisions, or fields, of the data specified in the command. Accordingly, in order for editing to occur, the corresponding fields or smaller units must have explicit format definitions in the data description.

Arithmetic Commands

Two of the verbs in the Commercial Translator vocabulary are used for arithmetic. They are SET and ADD. These two verbs form the basis for the three types of arithmetic commands, as follows:

The SET Command

The SET command permits the programmer to specify a computation or a sequence of computations. It can be used for all arithmetic. The general form of this command is:

SET *variable.1, variable.2, ... variable.n = arithmetic expression*

The equal sign in the SET command is used in the sense of replacement. It means, "replace the value of the variable(s) on the left side of the equal sign with the value of the expression on the right." If required, the result of a computation will be edited automatically according to the format of a receiving field as specified in the data description. For example, if a result represents an amount of money, editing appropriate to the defined format of the money field will be performed.

Ordinarily the result of the SET operation will be rounded to the number of places indicated by the format description of the result field or fields. If the dropping of digits (instead of rounding) is desired, the arithmetic expression is followed by

. . . TRUNCATED

In the process of storing the final result of a SET command it is possible for a loss of significant high-order digits to occur. For example, if a result field has been defined as having three places to the left of the decimal point and a result such as 1001 is developed, the high-order "1" will be lost. This situation is known as "overflow." If the SET command specifies just one result field (i.e., if it has only one variable-name to the left of the equal sign), the programmer may anticipate and provide for an overflow by appending

..., ON OVERFLOW *any imperative clause*

at the end of the SET command. In the event of an overflow, the object program will execute the command thus specified instead of storing the erroneous result.

Some examples of the SET command are:

SET GROSS.PAY = BASE.RATE * 40.

SET NET.PAY = GROSS.PAY − (FICA + STATE.TAX + DEDUCTIONS).

SET A= B * (C + D), ON OVERFLOW GO TO ERROR.

SET REORDER = ORDER.AMT * TR (STOCK.LEVEL LT ORDER.POINT).

SET APPROX = 2 * SQUARE.ROOT ((X)) TRUNCATED.

SET A, B, C =D.

The second of these examples shows the way in which subexpressions may be contained in parentheses when required. Likewise, the examples taken together illustrate the use of each element which may appear in an arithmetic expression, viz., arithmetic operators, names of variables and constants, literals, truth functions, and function-names. Each of these elements is considered individually below even though it may be discussed in greater detail elsewhere in the manual. (The formal rules governing the formation of arithmetic expressions are given in Appendix 2.)

Arithmetic Operators

The following arithmetic operators are used to join other elements to form meaningful arithmetic expressions:

Operator	Written as
Addition	+
Subtraction	–
Multiplication	*
Division	/
Exponentiation	**

These operators are sometimes referred to as "binary" operators because each of them relates two quantities. There are three additional arithmetic operators known as "unary" operators. They are:

Operator	Written as
Negation	–
Absolute value	ABS
Truth value	TR

These operators affect only the quantity or expression which follows, hence the term unary. If the operand of a unary operator is an expression it must be enclosed in parentheses, as in these examples:

```
. . . + ABS (A + B)
. . . * ( – (ON.HAND + ON.ORDER) )
. . . TR (A IS LESS THAN B)
```

Variables and Constants

Data-names which represent numeric variables or constants may appear anywhere in arithmetic expressions. Those which represent alphameric variables and constants, however, may appear only within truth functions.

Literals

Literals may be used as needed in arithmetic expressions. Literals used in the above examples of the SET command are 40 in the first example and 2 in the fifth.

Truth Functions

A conditional expression enclosed in parentheses preceded by the truth operator TR is known as a truth function. A truth function always has one of two values, 1 or 0, depending on whether the conditional expression is true or false. Accord-

ingly, a truth function can be manipulated arithmetically in the same manner as any other quantity.

In the fourth of the preceding SET command examples, TR (STOCK.LEVEL LT ORDER.POINT) is a truth function. When the SET command is executed, the relation between STOCK.LEVEL and ORDER.POINT is evaluated. If the relation is true, that is, if STOCK.LEVEL is less than ORDER.POINT, then the truth function takes the value 1. If the relation is false, it becomes 0. The result of this SET command, then, is to cause the field REORDER to contain the order amount if stock level is below order point, or zero if the stock level is at or above order point.

FUNCTION-NAMES

The names of functions are used in arithmetic expressions in much the same way as data-names or literals. That is, a function-name is treated as a quantity, as in the fifth of the preceding examples of the SET command, where the square root of X is multiplied by 2 and truncated to obtain the result. There are some differences, however, between data-names and function-names. Data-names represent values which, at object time, are immediately available for arithmetic operations. Function-names, on the other hand, imply an evaluation process at object time. The evaluation is carried out by procedural statements identified by a BEGIN SECTION command (see page 56). These procedure statements deliver a value to be used in computing the value of the arithmetic expression.

SET Used with Condition Names

The SET command has a second function that is not strictly arithmetic. It provides a convenient way of changing the status of a conditional variable, i.e., a variable which has one or more conditions associated with it. When SET is used for this purpose it takes the general form:

SET *condition.name*

As a result of this command, the variable with which "condition.name" is associated (in the data description) is assigned the status, or value, of the specified condition.

Using the example mentioned in Chapter 2, if the current value of the variable MARITAL.STATUS is SINGLE and the programmer wishes to change it to MARRIED, he simply writes:

SET MARRIED.

This is the equivalent of writing:

SET MARITAL.STATUS = 'M'.

It is important to recognize the distinction between testing and setting the value of a conditional variable. Testing the value provides a basis for a decision but does not change the value; the testing is accomplished by using a conditional clause, IF . . . THEN. Setting a value of a conditional variable, on the other hand, causes the current value to be replaced by another of the possible values and is effected by means of the SET command.

The ADD Command

The ADD command provides a way to add a given quantity to each of one or more variable quantities. The general form is:

ADD *data.name* TO *variable.1, variable.2, ... variable.n*

The effect of the ADD command is to increment the one or more named variables by the quantity represented by "data.name." In this context, "data.name" may be a literal or the name of a variable, constant or function.

The optional phrases TRUNCATED and ON OVERFLOW described above in conjunction with the SET command are equally applicable to the ADD command. As in the case of SET, the ON OVERFLOW option is permitted only if the command specifies a single result field.

Some examples of the use of ADD are:

```
ADD GROSS.PAY TO YR.TO.DATE.GROSS.
ADD 1 TO COUNTER, ON OVERFLOW GO TO BEGIN.
ADD ADJUST.FACTOR TO RATE TRUNCATED.
```

The ADD CORRESPONDING Command

This command permits the programmer to specify a series of additions. Its general form is:

ADD CORRESPONDING *data.name.1* TO *data.name.2, data.name.3,* *... data.name.n*

Each "data.name" in an ADD CORRESPONDING command must represent an area of storage which is composed of smaller units or fields. The effect of the command is to cause each field in "data.name.1" to be added to its corresponding field (i.e., a field with the same name) in the "to" area(s). Non-corresponding fields in "data.name.1" and in the "to" areas are not affected by the command.

To illustrate, suppose that the programmer wishes to accumulate department totals and grand totals from certain fields in each detail record of a payroll. The detail-record fields involved might be GROSS.PAY, FICA, FED.TAX, STATE.TAX and DEDUCTIONS. Having defined DEPT.TOTALS and GRAND.TOTALS as records containing corresponding fields, the programmer could write:

```
ADD CORRESPONDING DETAIL.RECORD TO DEPT.TOTALS,
  GRAND.TOTALS.
```

This one command would cause all of the desired totals to be accumulated; it would have the same effect as writing:

```
ADD DETAIL.RECORD GROSS.PAY TO DEPT.TOTALS
  GROSS.PAY, GRAND.TOTALS GROSS.PAY.
ADD DETAIL.RECORD FICA TO DEPT.TOTALS FICA,
  GRAND.TOTALS FICA.
ADD DETAIL.RECORD FED.TAX TO DEPT.TOTALS FED.TAX,
  GRAND.TOTALS FED.TAX.
ADD DETAIL.RECORD STATE.TAX TO DEPT.TOTALS
  STATE.TAX, GRAND.TOTALS STATE.TAX.
ADD DETAIL.RECORD DEDUCTIONS TO DEPT.TOTALS
  DEDUCTIONS, GRAND.TOTALS DEDUCTIONS.
```

Control Commands

The control commands enable the programmer to state the logical flow of a program. During the execution of the object program the procedure is normally executed in the order in which it appears in the source program. The control commands are used primarily to specify a departure from this sequence in order to execute some other portion of the program. The verbs that form the basis of the Commercial Translator control commands are GO TO, DO, and STOP.

The GO TO Command

The GO TO command is used to specify transfer-type operations. It has three forms:

UNCONDITIONAL

The unconditional GO TO is written:

GO TO *procedure.name*

It provides a transfer of control, or branching, to the item of procedure named in the command. The "procedure.name" may be the name of either a sentence or a section in the procedural part of the source program.

Some examples are:

GO TO MAIN.ROUTINE.
GO TO CALC.ORDER.POINT.

CONDITIONAL

The conditional GO TO command is essentially a multiple branch or switching point; from this point, control may pass to one of several places in the program. As the name implies, the transfer of control depends on the truth or falsity of one or more conditional expressions. The general form of the command is:

GO TO *procedure.name.1* WHEN *conditional expression 1,*
procedure.name.2 WHEN *conditional expression 2,* ...
procedure.name.n WHEN *conditional expression n*

When the conditional GO TO is executed at object time, each conditional expression is evaluated in turn until one is found to be true. Thereupon, control is transferred to the "procedure.name" associated with that conditional expression. Any remaining conditional expressions are left unevaluated. If none of the conditional expressions in the command is found to be true, control passes to the next clause or sentence in sequence.

The following is a typical use of the conditional GO TO:

GO TO ERROR.RTN WHEN DETAIL IS LESS THAN MASTER,
PROCESSING WHEN DETAIL IS EQUAL TO MASTER,
NO.ACTIVITY WHEN DETAIL IS GREATER THAN MASTER.

ASSIGNED

The assigned GO TO command also serves as a multiple branch point in a program. In this case, however, the transfer of control depends upon the prior setting of an

index rather than on the truth or falsity of conditional expressions. The setting of the index may have occurred in one of two ways:

1. Through bringing the index into storage as a constant or file variable.
2. As a result of computation involving the index.

The general form of the command is:

GO TO (*procedure.1, procedure.2, ... procedure.n*) ON *index.name*

A transfer of control will be made according to the value of the index at the time the GO TO is executed. If the value of the index is 1, control will pass to the first sentence or section of procedure named in the command; if 2, the second item named; and so on.

The functioning of the assigned GO TO command assumes that the value of the index will always be an integer in the range 1 to *n*. If the index has any other value, no transfer will occur; instead, control will pass to the next clause or sentence in sequence.

To illustrate, suppose that in a payroll job several different methods are used to compute gross pay depending on the employee's classification. The programmer might use an assigned GO TO command to effect a transfer to the appropriate routine, as follows:

GO TO (PIECE.WORK, INCENTIVE, HOURLY.RATE, SALARY) ON PAY.TYPE.

The index PAY.TYPE in this case would be a field in each employee record which would contain a 1, 2, 3, or 4 depending on the employee's classification.

Note that the parentheses shown in both the general form and the example are a fixed part of the command and must always be included.

The DO Command

The DO command provides a means of departing from the normal sequence of program steps in order to execute some procedure, i.e., some other portion of program, and then return to the original sequence. In other words, the DO is used to execute subroutines which, in the Commercial Translator system, are either named sentences or sections.

The DO command may take one of several forms. In the simplest form of the command the procedure is executed once each time the DO is encountered. Expanded forms of the command permit repetitive execution, or "looping," of the subroutine and control of subscripts. Data substitution, which is an optional feature, is also provided.

The simpler forms of the DO command are:

DO *procedure.name*

DO *procedure.name* EXACTLY *n* TIMES

where "procedure.name" represents the name of a sentence or section. (If a procedure consists of more than one sentence, it must be defined as a section in order to be named. The processor commands BEGIN SECTION and END are used to define sections.)

Any procedure, i.e., any sentence or section, that is referred to by a DO command must be what is known technically as a "closed" or "linked" subroutine. That is, it must be entered only through the use of a DO command, and not by any other means such as transfer of control to a sentence within the procedure or through the normal passage of control to the first sentence of the procedure. It should be noted, however, that this rule permits the addressing of a procedure by more than one DO command.

When the DO command takes the first of the two forms shown above, the "procedure.name" subroutine is executed only once and control passes to the sentence or clause following the DO.

For example, the programmer might write:

```
DO CALC.ORDER.POINT.
IF ORDER.POINT IS LESS THAN 2.5 * MONTHLY.USAGE THEN
  GO TO . . .
```

The DO command in this example will cause the CALC.ORDER.POINT procedure (subroutine) to be executed once, whereupon control will pass to the IF ORDER.POINT IS . . . sentence.

With the second form of the command the "procedure.name" procedure is executed the specified number of times. The number of repetitions, *n*, may be stated as a literal or as a data-name, but in either case the value of *n* must be a positive integer. To illustrate this form of the DO, suppose that in a payroll program a field called BOND.ACCUM is divided by another field, BOND.DENOM, and the result truncated to an integer in order to determine the number of bonds purchasable. The result might be called NO.OF.BONDS. Then the programmer could write:

```
DO BOND.ORDER.RTN EXACTLY NO.OF.BONDS TIMES.
```

This example assumes, of course, that the payroll program contains a procedure called BOND.ORDER.RTN which is used to prepare a file of bond orders.

The DO Command with Indexing

Repetitive execution, or looping, of a procedure based on indexing is provided in the more powerful forms of the DO command. To control a single index and/or provide subscript control of the variables associated with the index, the DO command takes the form:

DO *procedure.name* FOR *index.name* = *p*(*q*)*r*

where "index.name" represents a field which has been defined in the data description as an integer. The effect of this DO command (in the object program) is to set the index to the initial value *p* and transfer control to the first sentence of the named procedure. After the last sentence of the procedure has been reached and executed, control is returned to the DO command which increments the index by the quantity *q* and causes control to return to the "loop." This process is repeated until the value of the index equals *r*; at this point, control is no longer returned to the loop but instead passes to the sentence or clause which follows the DO. To state this another way, the command, "DO rtn FOR i = p(q)r" is the equivalent of:

```
         SET i = p.
START.   DO rtn.
         IF i = r THEN GO TO NEXT.
         ADD q TO i.
         GO TO START.
NEXT.        .
             .
             .
```

Each of the loop control parameters *p, q* and *r* may be either of the following:

1. A literal having an integer value.
2. The name of a field defined as an integer.

The following example shows a DO command of this form:

DO RATE.UPDATE.CALC FOR STATE = 1(1)50.

In this example, each item in a rate table is being updated. The table contains fifty entries—a rate for each state. The procedure RATE.UPDATE.CALC might appear as:

RATE.UPDATE.CALC. SET RATE (STATE) = RATE (STATE) * ADJUST.FACTOR + FLAT.AMT.

This procedure will be executed repeatedly, once for each value of the index STATE (from 1 through 50). The value of STATE (in the DO command) at any given time specifies which table item is to be dealt with in the procedure. Thus, the variable RATE is said to be subscripted by the index STATE.

When it is desired to control two indices during the execution of a DO, thus providing a loop within a loop, the command takes the form:

DO *procedure.name* FOR *index.name.1 = p.1(q.1)r.1, index.name.2 = p.2(q.2)r.2*

In this case, "index.name.1" and "index.name.2" are initially set to p.1 and p.2 respectively. Each time the inner loop is executed, index.name.2 is incremented by q.2 until it equals r.2; thereupon, index.name.2 is reset to p.2 and index.name.1 is set to p.1 + q.1. Control is returned to the inner loop and the process is repeated. When index.name.1 is equal to r.1, control passes to the clause or sentence which follows the DO.

A maximum of three indices may be controlled with the DO command. Again, the rightmost index is the one which varies most rapidly. The following example illustrates the DO controlling three indices:

DO COMPUTATION FOR HOURS = 1(1)12, MINUTES = 1(1)60, SECONDS = 1(1)60.

In this case, the procedure COMPUTATION is executed with the index SECONDS being incremented from 1 through 60 (in increments of 1), at which time MINUTES is increased by 1 and SECONDS again runs to 60. When the index MINUTES reaches 60 the index HOURS is incremented by 1, and so on. After HOURS has reached 12, the loop is completed and control passes to the operation following the DO. The subroutine will have been executed 60x60x12 times.

The DO Command with Data Substitution

Frequently, a procedure which has been written for a particular purpose can be used at some other point in a program only if provision is made to substitute different items of data to be operated on by the procedure. Such substitution could be specified by writing appropriate MOVE commands in conjunction with the second DO command. For example, if a procedure operates on three pieces of data called A, B, and C and produces two results, D and E, the programmer could use the procedure with other items of data by writing:

```
MOVE V TO A.
MOVE W TO B.
MOVE X TO C.
DO PROCEDURE.
MOVE D TO Y.
MOVE E TO Z.
```

Since this method is somewhat laborious, the Commercial Translator language provides a facility for writing procedures in a generalized form in order to accomplish the same result. Procedures which are to be used with data substitution are always defined by the two processor commands "BEGIN SECTION USING . . . GIVING . . ." and "END" (see page 56).

Data substitution can be specified in each of the several forms of the DO command. When this feature is utilized, the general form of the simplest DO becomes:

DO *procedure.name* USING *data.name.1, data.name.2, . . .* *data.name.n* GIVING *result.name.1, result.name.2 . . .* *result.name.n*

Similarly, if data substitution is desired in the other forms of the DO command, the "USING . . . GIVING . . ." option is simply appended to the command.

The "data.names" in the command specify the variable data, constants or literals which, at *object* time, are substituted for the data represented by the corresponding names (parameters) that appear in the "procedure.name" routine. Likewise, the "result.names" are the names which become associated with the outputs of the subroutine.

The data substitution feature of the language being discussed here should not be confused with *name* substitution, which is another feature of Commercial Translator. Name substitution is effected by the INCLUDE command and occurs at *processing* time; it involves the replacement of names in library procedures that are being incorporated in a program (see page 58).

To illustrate data substitution using a more complete example, suppose that the following procedure has already been written into a program. This procedure is designed to determine a minimum value:

```
MIN.ROUT.  BEGIN SECTION USING A, B, C GIVING MIN.
       IF A IS LESS THAN B THEN MOVE A TO MIN OTHERWISE MOVE
         B TO MIN.
       IF C IS LESS THAN MIN THEN MOVE C TO MIN.
       END MIN.ROUT.
```

In one portion of his program the programmer might have used this MIN.ROUT procedure to determine the lowest of three rates called R.RATE, E.RATE, and M.RATE using the command:

```
DO MIN.ROUT USING R.RATE, E.RATE, M.RATE GIVING MIN.RATE.
```

The effect of this DO command is equivalent to the following commands:

```
MOVE R.RATE TO A.
MOVE E.RATE TO B.
MOVE M.RATE TO C.
DO MIN.ROUT.
MOVE MIN TO MIN.RATE.
```

Now, in another part of the program, the programmer wishes to compare two ages and a constant value to determine the lowest of the three values. The MIN.ROUT procedure will serve the purpose in this situation as well. This time, however, the programmer writes:

```
DO MIN.ROUT USING INSURED.AGE, BENEFIC.AGE, 70 GIVING
  LOW.AGE.
```

Again the appropriate data substitutions are made when the DO command is executed. As a result, the two values representing ages will be compared with the literal value 70, and the lowest of the three will become the value of LOW.AGE.

The DO Command with Named END

When a DO command is compiled, the processor places appropriate instructions in the object program to effect transfer of control between the DO and the associated procedure and to perform any indexing operations specified in the DO. For correct functioning of the DO, these control instructions must be executed each time an iteration of the associated procedure is executed. The control instructions are performed following the last program command specified in the procedure. Accordingly, a problem arises in the case of a multi-sentence procedure in which the last program command is executed only under certain conditions, i.e., when the logic of the procedure requires a conditional exit prior to the last program command. The solution to the problem is to name the "END procedure.name" sentence which terminates the procedure and to use this name as an exit point. (Normally the END sentence is not named; this is the only exception to the rule.) This provides the necessary linkage with the control instructions. The following DO command and its associated procedure illustrate this situation:

```
   DO REORDER.RTN FOR PART.NO = 1001(1)1499.
                     .
                     .
                     .
REORDER.RTN.  BEGIN SECTION.
   IF QTY.ON.HAND(PART.NO) IS GREATER THAN ORDER.POINT
     (PART.NO) THEN GO TO EXIT.
   SET REORDER.AMT(PART.NO) = ORDER.AMT(PART.NO) -
     QTY.ON.HAND(PART.NO).
EXIT.  END REORDER.RTN.
```

In this example the logic of the procedure requires that the current iteration be terminated after the first sentence if the conditional clause is true. Assigning the name EXIT to the END sentence makes it possible to bypass the latter part of the procedure and yet maintain the necessary linkage with the control instructions.

The STOP Command

The STOP command is used to specify a halt in the object program. Its general form is simply:

STOP *n*

where *n* is an integer.

The number *n* will be displayed when the command is executed, i.e., when the machine halts. Restarting the machine causes execution of the object program to be resumed beginning with the next command in sequence.

For a "dead-end" halt, an unconditional GO TO command placed immediately following the STOP can be used to effect a transfer back to the STOP command.

Other Program Commands

In addition to the commands described in the foregoing pages there are two program commands that do not fall into any of the four categories discussed. These "miscellaneous" commands, which employ the verbs LOAD and DISPLAY, are described in the following paragraphs:

The LOAD Command

Since an object program may exceed the capacity of internal storage, a facility is needed for keeping a part of the program in "standby status" (in external storage) and for calling it in to replace a portion of the program currently in internal storage. The LOAD command, used in conjunction with the processor command OVERLAP, provides this facility. The general form of the LOAD command is:

LOAD *procedure.name*

where "procedure.name" is the name of a portion of the program which, at object time, will be waiting in external storage. The LOAD command causes this unit of procedure to be brought into the area of storage shared by the procedures mentioned in the associated OVERLAP command. The portion of the object program formerly occupying that area is replaced by the new portion but can be retrieved later by another LOAD command.

Any unit of procedure addressed by a LOAD command must be named in an OVERLAP command.

The use of the LOAD command is illustrated in the example given in the discussion of the OVERLAP command (see page 56).

The DISPLAY Command

At certain times during the running of the object program, the programmer may wish to examine particular values of data in internal storage, or may need to send messages to the operator. The DISPLAY command causes the object program to present such low-volume information on an appropriate output device or display medium. The general form of the DISPLAY command is:

DISPLAY *'any message' data.name 'any message'*

The DISPLAY command displays all the information that follows the word DISPLAY up to, but not including, a comma or period not enclosed in quotation marks.

The quotation marks in the DISPLAY command have precisely the same meaning as in alphameric literals. The information within the quotation marks will be displayed exactly as it appears in the command. Substitution of a value for its name may be specified by writing the name outside the pair(s) of quotation marks. That is, if the programmer wishes the current value of a data-name rather than the data-name itself to appear in a message, he arranges the text so that the data-name is not enclosed within a pair of quotation marks.

To illustrate this point, suppose that the following command is written:

DISPLAY 'VALUE OF WAGES LESS DEDUCTIONS IS NET.PAY'.

The resulting display at object time will be:

VALUE OF WAGES LESS DEDUCTIONS IS NET.PAY

This is obviously not what the programmer intended. Had the command been written,

DISPLAY 'VALUE OF WAGES LESS DEDUCTIONS IS' NET.PAY.

the message at object time would be:

VALUE OF WAGES LESS DEDUCTIONS IS $67.75

assuming that the current value of NET.PAY was $67.75.

The name of the variable outside the parentheses may be subscripted. It must represent a defined field, however, and not an arithmetic combination of fields.

The manual for each Commercial Translator processor will specify the standard display device for the respective machine system.

Processor Commands

The Commercial Translator processor commands are instructions to the processor; they cause the processor to take certain specific action. Some of the processor commands have an indirect effect on the object program. Others, such as NOTE, have no effect whatsoever on the object program. In general, the processor commands do not generate instructions in the object program.

The OVERLAP Command

An object program produced by the Commercial Translator system is organized as one loading of storage unless the programmer specifies otherwise by means of the OVERLAP command. This command designates portions of the program that are to occupy (at different times) the same area in internal storage. The general form is:

OVERLAP *procedure.name.1, procedure.name.2, ...*
procedure.name.n

As mentioned previously, OVERLAP is used in conjunction with the program command LOAD. OVERLAP instructs the *processor* to assign the same area of memory for the procedures mentioned in the command. LOAD, on the other hand, causes one of the procedures to be called in at object time so that it can be executed.

When the processor assigns storage space for the two or more procedures, it sets aside an area large enough to accommodate the longest. Thus when a procedure is

loaded over one which is longer, not all of the earlier procedure will be "erased." Accordingly, all overlapped procedures should have an unconditional GO TO as the last command.

The loading of an overlapped procedure does not cause its execution. The programmer must specify a transfer of control to the procedure by means of a GO TO or a DO command. Also, care should be taken to insure that a LOAD command does not appear within the procedure it obliterates. This would cause the destruction of the subsequent commands which effect transfer out of the procedure.

When a source program includes one or more OVERLAP commands, the object program will be organized as follows:

1. The initial loading of storage will include all parts of the program not mentioned in any OVERLAP, plus the *first* procedure named in each OVERLAP command.
2. Any procedures not included in the initial loading may be called in by a LOAD command.
3. Any procedure that has been obliterated by a LOAD command may be retrieved (in its original form) by another LOAD command.

A simple example of the use of the OVERLAP and LOAD commands is as follows. The "housekeeping" portion of a program is to be executed and then overlapped by the main part of the program. Assuming that both procedures, i.e., both of these parts of the program, consist of more than one sentence, each must be treated as a section in order to apply names. The structure might be represented as follows:

```
HOUSEKEEPING.  BEGIN SECTION.
               ......................................................
               ......................................................
               ......................................................
               ......................., GO TO SUPERVISOR.1.
               END HOUSEKEEPING.

MAIN.ROUTINE.  BEGIN SECTION.
               ......................................................
               ......................................................
               ......................................................
               ......................................................
               END MAIN.ROUTINE.

SUPERVISOR.1.  LOAD MAIN.ROUTINE, GO TO MAIN.ROUTINE.
```

Elsewhere in the source program, probably included with the other processor commands, the programmer would write:

OVERLAP HOUSEKEEPING, MAIN.ROUTINE.

As indicated in the schematic program above, the program command, LOAD MAIN.ROUTINE, is placed outside the housekeeping routine, possibly in a short supervisory routine.

The BEGIN SECTION and END Commands

The two processor commands, BEGIN SECTION and END, perform a simple but important function. They are used to delimit sections of procedure and thus extend the range of a procedure-name. That is, they enable the programmer to give names to units of procedure that consist of more than one sentence. Unless BEGIN SECTION and END are used, a procedure-name applies only to the sentence which follows it.

This pair of commands is used as shown in the following general form:

```
procedure.name. BEGIN SECTION.
        any sentence.
             .
             .
             .
        any sentence.
        END procedure.name.
```

Two of the program commands and one processor command take advantage of the facility provided by BEGIN SECTION and END. They are DO, LOAD and OVERLAP. The usefulness of these commands would be seriously impaired if they could not operate on pieces of procedure larger than sentences.

As indicated in the general form, the name of a section appears at the beginning and also at the end of the procedure. The second occurrence is required because sections of procedure may be "nested," i.e., one section may be contained within a larger one. For example, the situation illustrated below is permitted:

```
NAME.1.  BEGIN SECTION.                    ⎫
     ..........................            ⎪
     ..........................            ⎪
NAME.2.  BEGIN.SECTION.        ⎫           ⎪
     ..........................⎬ Section 2 ⎬ Section 1
     ..........................⎪           ⎪
     END NAME.2.               ⎭           ⎪
     ..........................            ⎪
     ..........................            ⎪
     END NAME.1.                           ⎭
```

In this example, Section 2 is said to be nested within Section 1.

Normally the terminating sentence, "END procedure.name," is not itself named. There is one exception to this rule: The END sentence in a procedure associated with a DO command may be named in order to provide a reference point at the end of the procedure (see page 53).

As mentioned in the discussion of the DO command, a section of procedure that is addressed by a DO becomes a closed subroutine; it can be entered only through the use of one or more DO commands. If a DO command employs the optional data substitution feature, the BEGIN SECTION command for the associated procedure becomes:

```
BEGIN SECTION USING parameter.1, parameter.2, ...
   parameter.n GIVING function.1, function.2, ... function.n
```

The "parameters" are the names of data which, at object time, will be replaced by the data specified in the USING phrase of the DO command. The "functions" are names representing results produced by the procedure; these results become the values of the result-names specified in the GIVING phrase of the DO command. An example of the USING . . . GIVING phrases is included in the discussion of the DO command on page 52.

Another point mentioned elsewhere should be noted in this context: The names of functions, i.e., the names of results produced by a section of procedure, may be used directly in an arithmetic expression instead of writing a DO command. In this case, each function-name is followed by its parameters enclosed in double parentheses. (See the discussion of functions in Chapter 2 and in this chapter on page 46.)

The INCLUDE Command

The INCLUDE command causes the processor to extract a unit of procedure from the library and to insert it in the present program. The basic forms of the command are:

```
INCLUDE library.procedure

INCLUDE HERE library.procedure
```

The "library.procedure" named in the command may be either a sentence or a section of procedure filed in the library under that name. The first form of INCLUDE causes the procedure to be placed at the end of the present program. With INCLUDE HERE, the procedure is inserted wherever the command appears. The first form is normally used to copy closed subroutines (i.e., procedures that are addressed by DO commands) since such procedures must be set off from the main flow of the program. Procedures to be used "in line" are inserted by means of the second form.

Name substitution is an optional feature of the INCLUDE command which enables the programmer to rename the procedure itself and/or certain names within the procedure. The substitution is done by the processor at the time the procedure is included. To specify a new name for the procedure, an additional phrase is appended to the basic forms of the command:

```
... AS procedure.name
```

If this phrase is used, all occurrences of the library procedure-name are replaced by the "procedure.name" indicated. Otherwise, of course, the procedure is referenced by means of its name in the library.

Similarly, if names within the library procedure are to be replaced by new names, another phrase is appended to the command:

```
...WITH new.name.1 FOR old.name.1, new.name.2 FOR
    old.name.2, ... new.name.n FOR old.name.n
```

Again, all occurrences of the "old.names" are replaced by the specified "new. names."

The CALL Command

The CALL command is used to specify alternate names, or synonyms, for previously defined names. The general form of the command is:

CALL (*old.name.1*) *new.name.1*, (*old.name.2*) *new.name.2*, ... (*old.name.n*) *new.name.n*

Synonyms are useful as abbreviations for often used names and as a means of communication between parts of a program, written by different programmers, that must refer to common areas of data. Data, work areas, and constants are capable of being named and thus may be renamed by means of this command. In the general form, the "old.names" are the names already defined, and the "new.names" are the alternate names being assigned.

Some examples of the CALL command are:

CALL (MASTERFILE) M.
CALL (INSURANCE.PREM) INSPREM, (RETIREMENT.PREM) RETPREM.
CALL (DEPARTMENT.TOTAL HOURS) DEPT.HRS.

Synonyms must always be single names rather than compound names. A synonym may be applied to a compound name, however, as in the third example above.

The NOTE Command

The NOTE command enables the programmer to place explanatory information in the listing of the program. Its general form is:

NOTE *any sentence.*

This command affects only the program listing, not the program itself. The sentence introduced by NOTE will not produce instructions in the object program.

Any combination of characters from the allowable character set may be placed after the verb NOTE as explanatory information. Some examples are:

NOTE START OF MERGE 1.
NOTE UPDATE BEGINS HERE IF RATE HAS CHANGED.

The NOTE command is terminated by the first period that is followed by a blank.

The ENTER Command

Although most source programs will be stated entirely in the Commercial Translator language, there may be occasions when the programmer wishes to employ the symbolic "one-for-one" language of the particular machine system. The ENTER command instructs the processor to accept statements in another language. Its general form is:

ENTER *coding.language*

To revert to the Commercial Translator language, another ENTER command is required, specifically:

ENTER COMMERCIAL TRANSLATOR.

Further information regarding the particular symbolic languages will be provided in the publications dealing with the respective processors.

Relationship Between Program Commands and Processor Commands

It will be obvious to the reader at this point that the program commands and the processor commands are essentially quite different. Accordingly, they cannot be intermixed in source program sentences. For example, it is meaningless to write sentences such as:

IF A = B THEN GO TO C OTHERWISE OVERLAP SECTION.1, SECTION.2.

Processor commands, with the exception of BEGIN SECTION and END, should be written as unnamed sentences. This rule makes it impossible for a program command to transfer control to a processor command. (In a sense, BEGIN SECTION and END are not exceptions to the rule since their main purpose is to permit the programmer to apply a name to two or more sentences of *procedure*.) This is not to say, however, that the two types of commands cannot appear in sequence in a source program. It is perfectly logical, for example, to use INCLUDE HERE following a program command as long as the preceding sentence logically leads to the first sentence of the procedure being included.

Chapter 4: Data Description

What a Data Description Is

The preceding chapter has shown how to write instructions that direct the computer to perform various kinds of operations. Up to this point, the discussion has been limited to operations that act directly on the actual data of the problem being solved. It has been assumed that the various items of data had already been placed in the system for this purpose.

In actual practice, however, the programmer must make the arrangements for placing the data into the system in a manner which permits it to be used efficiently. Although this is not difficult, it requires thoughtful planning. It assumes that the data is arranged according to a well defined plan, and it consists of furnishing the system with information about that plan. If the data is not suitably organized, the programmer must work out an appropriate plan and arrange the data accordingly. The principles involved are not unlike those the programmer would follow if he were designing a set of business forms for handling the data manually.

The need for doing this should quickly become apparent. Suppose, for example, that a clerk has jotted down on a scrap of paper the figures "46.30." These figures presumably represent a value that means something to him. But could anyone else tell what kind of a value this is? Is it, for example, a unit selling price? The number of hours worked by an employee? A discount? The percentage of water in a chemical product? Obviously, the number, standing alone, has very little practical significance.

In normal business operations, of course, there are several means of identifying values of this kind. In some cases the number will actually be labeled. In many other cases its identity will be seen at once from the fact that it appears in a particular position on a particular form, or that it is written in a certain column in a certain ledger.

When data is entered into a data processing system, it is rarely feasible to label each number. If "46.30" is a rate of pay, for example, the programmer would *not* write "$46.30 per 40-hour week." Neither would he write "46.30 lbs. per cu. ft." if it were a measure of weight. The value would be entered simply as a number, and the programmer would have to find some other way to show what the number represented. The method of doing this is explained in this chapter.

The Purpose of a Data Description

It can be said, therefore, that a major purpose of writing a "data description" is to furnish the processing system with a means of identifying each item of data which has been named in the procedure statements. This description must include all items on which the system is to operate.

Actually, the system needs more than a means of merely identifying the data. If the data is numeric, for instance, there must be a means of showing where the decimal point is located. If the system is to print out monetary values, it must have information on where to place the dollar sign, and whether or not to print leading zeros. Details of this sort are usually referred to as "editing." Some of them are handled automatically by the processor, so that the programmer need not concern himself with them. In other cases, the programmer can specify one of several methods for handling a particular situation; in such a case, the data description allows him to give the required instructions.

Writing a data description is not difficult, once a few basic principles are understood, and these are explained in the following pages. The reader should realize,

moreover, that this method of describing data in a separate section of the program has a number of important advantages.

In the first place, the typical program will deal repeatedly with data of the same kind. Use of a separate data description permits the programmer to describe each kind of data once, instead of having to describe each individual item as it occurs.

A second consideration is that, once a given set of data has been described, the same description can be used again as part of a new program that works with the same data, or the same kinds of data. This would be a considerable advantage even if it were necessary to copy the data description into the new program by punching new cards. In practice, however, the data description can actually be stored in the library, on tape or in cards, so that it can be called for when needed.

Finally, the Commercial Translator data description is so designed that a program developed for use on one data processing system can be run on a different kind of system (within limits) with relatively few changes in the data description. A basic reason why this is possible is that the data description is tailored to the logical structure of the data files, which do not alter materially from machine to machine, while most of the details that relate to actual equipment are handled by the processors furnished for the various machine systems. The programmer is thus spared many of the problems that can arise when it is necessary to run a program on a system other than the one for which it was originally designed.

A data description for a data processing system contains a number of elements which are already familiar to those who have worked with punched card equipment. As most readers will recognize at once, punched card operations require that data be arranged in the cards in accordance with a definite pattern. In essence, each item of data is identified by its physical position in the card. Thus, if the first six columns of a deck of payroll cards are reserved for employee numbers, the system will treat all values in those columns as employee numbers, unless special coding is used to indicate exceptions. But any such exceptions must be indicated by codes which are themselves identifiable by their positions in the card. In short, the key to identifying a value in a punched card is its position in the card.

The same basic rule applies to electronic systems, but the number of possible positions is greatly increased. Moreover, data can be shifted from one position to another. Yet it is always the position of the data that identifies it, and it is one of the functions of a Commercial Translator processor to keep track of the position of each item.

For this reason, a major part of a data description consists of the details necessary to determine the position of each item. Among other things, it is necessary to give the length of each kind of item (that is, the space it will require) and its relative position in the file. The processor will use this information to assign an initial location in storage for each kind of data. At the same time, it will note the name which the programmer has assigned to the item. By relating this name to the location, the program will be able to find the item when it is referred to by name. Furthermore, it will keep track of all changes in the position of the item; this saves the programmer from having to deal with internal addresses in the system and permits him to refer to data by name.

It follows, of course, that the efficiency of the program will depend in part on the arrangement of the data. The data description provides for describing a data organization already in existence. If the arrangement of the data does not take full advantage of the capacities of the system, the programmer may wish to reorganize the original input data before writing the data description.

Files, Records, and Fields

Before proceeding with the details of the data description it is only necessary to define three terms as they are used in the Commercial Translator system: "file," "record," and "field." The exact implication of these terms varies from person to person and company to company. In this manual they are used to distinguish between kinds of data that can be operated on by different Commercial Translator instructions. Thus, the verbs OPEN and CLOSE can operate only on files, and a file must therefore be regarded as a body of data organized for use within the scope of those commands. Similarly, the verbs GET and FILE can operate only on records, so that a record can be defined as a body of data which can be governed by those commands. Fields, on the other hand, are subordinate units of data and are not readily definable in terms of the commands that affect them. While these definitions may seem somewhat arbitrary, it will be found that in practice they correlate with normal principles of data organization.

Files

A *file* is a body of data stored in some external medium which can be made accessible to the system by the use of the verb OPEN. In this sense, an external medium is any medium which provides input to the system from without. Data in such a medium cannot be used by the system until it has been brought into it, which is the basic purpose of an OPEN command. The usual external medium of electronic data processing systems, of course, is magnetic tape.

The concept of a file as "external" to the system carries certain implications which should be considered. These arise, not from the definition of a file, but rather from certain practical considerations.

For example, a file is usually a relatively large body of data, although there is no implied relationship between the length of a file and the storage capacity of a tape. Thus, there may be more than one file on a tape, and, on the other hand, a file may extend over a number of tapes.

A file is commonly understood to consist of a number of individual records, the records being generally similar to each other in size, content, and format. (Sometimes a series of differing records may be grouped together in a file; in such a case, the programmer must make provisions for distinguishing among them.) The file itself may be unique, or it may closely resemble other files. Thus, a file of actuarial data might be the only one of its kind in a library, while a file of insurance policy data might differ from another chiefly because it covers a different area or a different period of time. In this sense, a file serves a function like that of a ledger, or a filing cabinet containing a series of similar papers. It is usually a rather large body of related information. Thus it is convenient in most cases to treat such large bodies of data as complete and separate units which can be identified externally by direct reference to the physical media in which they are stored. Such media may be controlled only by the OPEN and CLOSE instructions, and this fact accounts for the definition of a file used in the Commercial Translator system.

Many files contain special records which are used primarily to identify the file and its contents. These records are called "labels." If labels are used, one is normally placed at the beginning of the file, and another is placed at the end. These labels correspond in many ways to the labels on file drawers, except that certain technical information is required because of the nature of an electronic system. Many of the details of labeling are handled automatically by the "input-output package" provided for each system. Most labeling procedures are prescribed in the environment description for each machine system, and will be discussed, therefore, in the publication covering the Commercial Translator processor for each particular system. However,

the programmer has the option of prescribing certain details of a label by the use of the LABEL type code, which is discussed later in this chapter.

Records

A *record* is a portion of a file which can be made accessible to the system by the verb GET, assuming the file has previously been "opened." The size and position of a record in storage are determined by the specifications given in the data description. Its contents are referred to by their location in storage, whereas a file, as such, is never actually brought into storage. In other words, a record, unlike a file, is conceived as an "internal" body of data, even though it may be stored externally for long periods of time as part of a file.

Usually, but not necessarily, the records within a file are similar to each other in content, size, and format. For example, there might be a file called PAYROLL RECORD —EASTERN REGION, which would contain hundreds, or thousands, of individual payroll records. Each individual record would normally contain data relating to a particular employee, such as his name, employee number, job title or class, pay rate, dependency classification, deductions, and so on. The separate parts of a record might be quite dissimilar, but they would all relate to a single subject (in this case, an employee), and other records in the same file would carry similar information about other employees. Furthermore, each item would occupy the same relative position in one record that it does in another; it would also have the same format and be of the same size. This makes it possible to describe each *kind* of data instead of each individual item of data.

A record may be obtained for use by the verb GET, and, when processing has been completed, it may be placed in an output file by the verb FILE.

Fields

A *field* is a block of data which can be operated on as a unit by the arithmetic commands and by the commands that control the movement of data within the system (other than the verbs GET and FILE). In many cases, a field will be a subordinate part of a record (such as a PAY.RATE field within a record called PAY.RECORD). However, this relationship does not necessarily hold. A record may contain only one field, for example, and, in fact, several records, or parts of several different records, may be regrouped within storage to form a new block of data which can then be treated as a field. Fields must always be defined in the data description.

In most cases, a field is considered to deal with only one element of data, at least for the purposes of the operation being performed. In a payroll record, for instance, such items of data as employee name, pay rate, employee number, and so on, would each be treated as a field. In practice, a field may be further subdivided. The classic example is the representation of a date. According to one view, the complete date, consisting of day, month, and year, is a single field, and the smaller components are regarded as "subfields." Another view holds that day, month, and year are each fields, and that the complete date is a group of fields.

As far as the Commercial Translator language is concerned, however, distinctions between subfields, fields, and groups of fields are handled by assigning "level" numbers to the various components. This is explained later in this chapter. It is felt that this method eliminates confusion that could result from trying to show relationships among the smaller units of data by the use of only three terms (i.e., field, subfield, and group of fields), and it allows, through the use of level numbers, a greater degree of flexibility in organizing the data. In general, it should be understood that a field is any element of data which can be operated upon by arithmetic and/or data transmission verbs.

Data Description Format

The detailed information which must be supplied in order to describe the data fully can be entered into the system easily by the use of punched cards. A separate card is used for each kind of data item. The format of the card is reflected in the data description form, which is illustrated in Figure 1.

The spaces on this form correspond exactly to the columns of the cards, so that the information written on each line of the form can be punched directly in a card without editing. Since the form provides a field for each of the various kinds of information required, it will serve in this chapter as a convenient check-off list by which to list and discuss each of the items required in the data description.

A "division header" must be placed immediately before the first entry of each consecutive group of data description cards. The name *DATA, placed so that the asterisk is in Column 7, should be the only entry on the line (i.e., card), except for the serial number and identification entries in Columns 1-6 and 73-80.

General

As will be seen from the illustration, the form provides spaces for each column in the card. The spaces representing Columns 1 through 3 and 73 through 80 are shown only once, since the information they will contain is assumed to be repeated for each line of the form. The information to be written in Columns 4 through 72, however, will vary from card to card, and spaces corresponding to these columns are therefore provided on each line.

The programmer should follow these space marks exactly, so that each item of information will be punched in the columns reserved for it. It will be noted that the data description format differs from the procedure description format in several ways. In particular, the former is more rigid; it does not allow "free form" description.

The columnar format of the card is summarized in the following table:

Columns	Number of Columns	Use
1-6	6	Card Serial Number
7-22	16	Name
23-24	2	Level Indication
25-30	6	Type
31-35	5	Quantity
36	1	Mode
37	1	Justification
38-71	34	Description
72	1	Continuation Indication
73-80	8	Identification

Ctl. and Serial (Col. 1-6)

It is essential that each item of the data description be entered into the system in proper sequence, since the sequence controls the internal position of the data. The first six columns are reserved for a serial number, which is used to indicate the sequence of the cards. This number is normally numeric. Its first three digits are written in the box marked "CTL" (for "control"), which corresponds to Columns 1 through 3. It is assumed that the first three digits will be common to all serial numbers written on the same page. Although these digits must be punched in each card, it is sufficient to write them once in the CTL box, instead of repeating them on each line.

The remaining digits are written in the box labeled "SERIAL," which corresponds to Columns 4 through 6. In normal practice, only Columns 4 and 5 are used initially,

IBM

COMMERCIAL TRANSLATOR DATA DESCRIPTION

CTL. 1 3	PROGRAM	SYSTEM	PAGE OF
	PROGRAMMER	DATE	IDENT. 73 80

SERIAL 4 6	DATA NAME 7 22	LEVEL 23 24	TYPE 25 30	QUANTITY 31 35	MODE 36	JUSTIFY 37	DESCRIPTION 38 71	CONT. 72

Figure 1

and Column 6 is left blank. This makes it possible to insert correction cards later if necessary. The blank is the first character in the collating sequence and will therefore be placed in sequence before any other character in the same column. Thus, if the numbers 23 and 24 have been punched in Columns 4 and 5, each will be read as a three-digit number with a blank in the third position, and a correction card with the number 235 would be collated between them.

When the cards are read into the data processing system, their serial numbers will be checked for correctness of sequence. In all IBM machines the numeric collating sequence is, first, the blank, then the numerals from 0 to 9. Alphabetic characters, if any, will be checked for sequence in accordance with the collating sequence of the particular machine system.

Data Name (Col. 7-22)

Columns 7 through 22 are reserved for any name which the programmer may have assigned to the data described in the card. The rules for forming names (see page 15) state that a name may contain as many as 30 characters. The card format provides only 16 columns, however, so that any name having more than 16 characters must be carried over to the "Data Name" columns in a succeeding line. In order for the processor to recognize this situation, the programmer must place a character in the continuation indication column (Column 72). Any non-blank character will serve the purpose. The processor will then be able to combine the two parts of the name into a single name. The programmer need not worry about the point at which the break between lines occurs; the processor will close up any blanks occurring in the "Data Name" columns. This provision allows the programmer to indent the carry-over if he wishes.

Names are usually written with an assumed left margin immediately to the left of Column 7. They may be indented from this assumed margin if the programmer wishes, but the processor will ignore any indentation. If no name is assigned, Columns 7 through 22 should be left blank.

Names may be assigned to any item of data, or to any group of data items stored consecutively within the system. Thus, names may be given not only to groups of items in the input files, but also to groups formed within storage as a result of operations performed by the system. Any name so assigned may be used as an operand in a procedure statement.

All data-names used in the program must be defined in the data description. In actual operation, the system will convert these names to actual machine addresses when the object program is assembled and will use those addresses as the actual means of locating the data. The purpose of writing the names in the data description is to furnish the processor with the information it needs to do this.

Data-names must not overlap. Each field within a record can be given a name, and any group of consecutive fields can also be given a name. Thus, a single field may be operated on individually by reference to its name, or collectively as part of a group called by the group name. However, the same field may not be included as a part of each of two overlapping named groups of fields.

For example, if three successive fields are named A, B, and C, the group name X might be assigned to the pair A and B. If this were done, the name Y could *not* be assigned to the pair B and C, since field B is already part of a named group of fields. If the programmer needs to be able to refer to fields B and C by a single name, however, he can rename the entire group of three fields, using the REDEF type code described later in this chapter. This procedure would not delete the original names; the new names and the names originally assigned would all be available for use thereafter.

Level (Col. 23-24)

Level numbers are used to describe the way in which a body of data is organized. Basically, level numbers are assigned to items of data to show their relationship to other items of data—or, in other words, to show the structure of a record. Any number from 1 to 99 can be used. All data description entries must be assigned level numbers.

In general, each item is considered to be a subdivision of the last item preceding it which has a lower number. Figure 2 shows how a typical series of files, records, and fields might be organized, using the familiar method called outlining. The file structure is shown by the use of indentation, each item being considered a part of the last item above it which is indented to a lesser degree.

The technique of indentation, in other words, is a visual way of showing level. It may be used in the "Data Name" columns, but it will have no effect on the processor itself. However, since it helps to identify the various levels visually, indentation may be useful in clarifying the file structure when the program is listed.

For comparison, two additional columns have been provided at the right in Figure 2. These show the data classification of each item, together with hypothetical level numbers such as might be assigned to a file structure of this kind. It should be pointed out that entries for files and groups of files are not actually used in the data description.

It is obvious from the outline that each item from EMPLOYEE NUMBER through LOCATION is a part of PAY RECORD, and that each item from FICA through HOSPITALIZATION is a part of DEDUCTIONS, YEAR TO DATE. Had the principle of indentation not been used, the reader might still determine these relationships by examining the level numbers in the right hand column, following the rule that each item is part of the next item above with a lower number. The processor, of course, will ignore any indentation and will store the data in accordance with the level numbers.

It is not necessary that level numbers be assigned in consecutive order, although it is done that way in Figure 2. The items at level 02, for example, might have been assigned level 04, or any other convenient number, as long as it was greater than 01. Similarly, the items at level 03 could have been given any other number as long as it was greater than the number of the next higher classification. In fact, it is often useful to skip numbers when they are initially assigned, to allow for possible regroupings or insertions at a later time.

The reader will also note that each item at the record level and below represents a *kind* of data, not a specific item of information. Thus, although there will be only one file called EASTERN REGION SALES FORCE, within that file there will be many individual units called PAY RECORD, and each of these will contain information of the same general character and format, as specified by the names of the fields within it. The purpose of the data description is to give information about each of these *kinds* of data. The data description should be thought of as a "pattern" which the files will follow.

It may be helpful to consider a second method of showing how data may be organized. Figure 3 shows how the same payroll file might be represented in the form of an organization chart.

This chart demonstrates one important fact: Each item of data is, or may be, related to other items above and below it. In other words, the data is organized "vertically"—no item is related directly to any other item at the same level, except through an item at a higher level. One result of this is that if a particular item is of the same kind as other items in the file, it may be identified by reference to the item above it in the organization structure. Thus, the name PAY RECORD does not single out any

Organization of Payroll Files

Standard Outline	Equivalent in Commercial Translator	
	Data Classification	Level Number
*EASTERN REGION	group of files	
*SALES FORCE	file	
PAY RECORD	record	01
EMPLOYEE NUMBER	field	02
EMPLOYEE NAME	field	02
LAST NAME	field	03
FIRST NAME	field	03
JOB TITLE	field	02
COMMISSION RATE	field	02
GROSS PAY, YEAR TO DATE	field	02
DEDUCTIONS, YEAR TO DATE	field	02
FICA	field	03
FEDERAL INCOME TAX	field	03
STATE INCOME TAX	field	03
SAVINGS BONDS	field	03
HOSPITALIZATION	field	03
NET PAY, YEAR TO DATE	field	02
LOCATION	field	02
*PRODUCTION FORCE	file	
PAY RECORD	record	01
EMPLOYEE NUMBER	field	02
EMPLOYEE NAME	field	02
LAST NAME	field	03
FIRST NAME	field	03
JOB TITLE	field	02
HOURLY RATE	field	02
GROSS PAY, YEAR TO DATE	field	02
DEDUCTIONS, YEAR TO DATE	field	02
FICA	field	03
FEDERAL INCOME TAX	field	03
STATE INCOME TAX	field	03
SAVINGS BONDS	field	03
HOSPITALIZATION	field	03
NET PAY, YEAR TO DATE	field	02
LOCATION	field	02
*WESTERN REGION	group of files	
*SALES FORCE	file	
PAY RECORD	record	01
EMPLOYEE NUMBER	field	02
⋮		
LOCATION	field	02
*PRODUCTION FORCE	file	
PAY RECORD	record	01
EMPLOYEE NUMBER	field	02
⋮		
LOCATION	field	02

*Files and groups of files are not actually entered as such in a Commercial Translator data description. Also, none of the names is in Commercial Translator format.

Figure 2. Typical File Structure (theoretical)

ORGANIZATION OF PAYROLL FILES

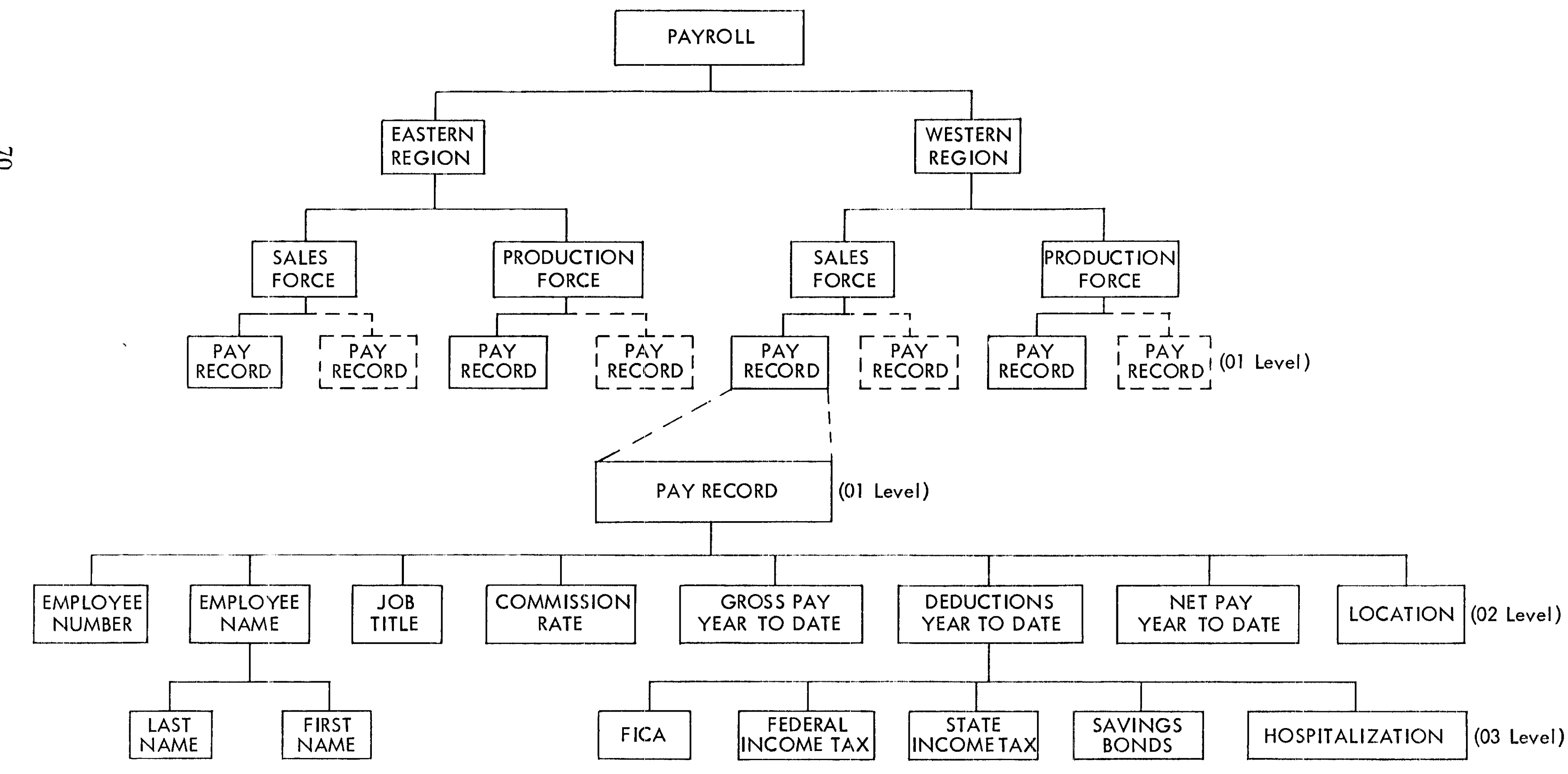


Figure 3. Typical File Organization

one unique kind of item, but SALES FORCE PAY RECORD distinguishes one kind of item from any item designated PRODUCTION FORCE PAY RECORD. Similarly, since in this illustration there are two SALES FORCE files, each can be identified by reference to the next higher level. Thus, a particular kind of item might be known as EASTERN REGION SALES FORCE PAY RECORD.

The use of a higher name to identify a lower one is known as "name qualification," and a name so qualified is known as a "compound name." This is an important method of identifying individual items which do not have names unique in the program. It is true that in this illustration the qualifying names are those of files. However, the principle is applicable internally, within files, whenever it is necessary to distinguish between fields having the same name. (See the discussion of compound names beginning on page 15.)

Level numbers are not actually attached to the data in the sense that an employee number is part of a pay record. They are used to instruct the processor to perform certain technical functions which need not concern the programmer. Essentially, they are used *before* the actual data is read into the system, as a means of preparing the system to receive it. Once the data description has been written, the programmer need no longer concern himself with level numbers unless, owing to changes in the data or the program, a new data description should become necessary.

Type (Col. 25-30)

Columns 25 through 30 are used, when necessary, to show that the data being described is of a certain special type. If these columns are left blank, it will be assumed that the remainder of the particular entry describes a data field or group of fields. The type codes which may be used in these columns are the following:

RECORD
COND
FUNCT
PARAM
REDEF
COPY
LABEL

Each of these is discussed in the following pages.

RECORD

This type code shows that the data being described is a record and is therefore accessible by GET and FILE instructions. This is equivalent to identifying an item of data as an input/output record. Each record named in the data description must also be named in the environment description, as specified in the publication covering the processor for each particular system.

COND

The type code COND is used to show that the data referred to is one of the possible conditions which a conditional variable may assume. In the discussion of conditional expressions in Chapter 2, it was pointed out that a conditional variable is the name of a field which will contain, at different times, any of a number of different values, depending on conditions existing in the data. Each of the values that may be placed in the field is a "condition."

In an example used in Chapter 2, the name MARITAL.STATUS was given as the name of a conditional variable. This name refers to a specific field reserved in storage into which values representing conditions will be entered. Typical conditions for this

field would be "single," "married," and "divorced." While these words could actually be placed in the MARITAL.STATUS field, it is more economical of space, and generally more efficient, to use codes. The initial letters M, S, and D were used as codes in this example. Thus, the field MARITAL.STATUS might contain any one of these letters at a given time.

However, so that the programmer can refer to these codes by their names, he must specify in the data description which code corresponds to each name. This may be done in the following manner:

Suppose that the field MARITAL.STATUS has been given the level number 06. The names of the conditions which may be entered into the field must then be assigned a lower level (i.e., a higher number) and entered in the data description immediately following the name of the field. This means that they will be treated as if they were each a subdivision of the field, in accordance with the rules for assigning level numbers, although, in practice, only one condition will be considered at a time. A portion of the data description might then appear as follows:

SERIAL 4 6	DATA NAME 7 22	LEVEL 23 24	TYPE 25 30	QUANTITY 31 35	MODE 36	JUSTIFY 37	DESCRIPTION 38
	MARITAL.STATUS	06					A
	SINGLE	07	COND				'S'
	MARRIED	07	COND				'M'
	DIVORCED	07	COND				'D'

The entries under "Description" will be explained later in this chapter, but, in summary, the "A" indicates that the field will contain one non-numeric character, while the initials S, M, and D are enclosed in quotation marks to show that they are the actual values to be used in the program.

In this example, the fact that the names SINGLE, MARRIED, and DIVORCED are the names of conditions is shown by the use of the type code COND. The relationship of these conditions to the field MARITAL.STATUS is shown by the fact that the condition-names have a higher level number and follow the name of the conditional variable immediately.

It should always be remembered that the condition-name is the name of the *value* which can be placed in a field; it is not the name of the field itself. As was pointed out in Chapter 2, the condition-name MARRIED, in this case, would be equivalent to MARITAL.STATUS='M'; it follows that such expressions as SET MARRIED or IF MARRIED will be interpreted to mean SET MARITAL.STATUS='M' and IF MARITAL.STATUS='M' respectively. The condition-name, in other words, is a short way of writing an expression that shows the value in the field.

It is not always necessary to list condition-names in the data description in the manner shown above. If the programmer, when he writes the source program, limits himself to the full form of conditional expressions (such as IF MARITAL.STATUS = 'M' THEN . . .), he need not assign names to the conditions. However, if he wishes to use the shorter and more convenient method of referring to conditions by name, he must write a data description entry for each.

FUNCT

A function has been defined in Chapter 2 as a result obtained by following a procedure. In the Commercial Translator system, the term is used only in connection with procedures specified in the BEGIN SECTION command. Any data-name specified in the GIVING clause of the BEGIN SECTION command is a function in this sense, and it must be identified as such in the data description. The function-name is identified by the type code FUNCT, and the function must be fully described in accordance with the provisions of this chapter. The parameters written in the USING clause of the BEGIN SECTION command must be identified in the data description by the type code PARAM, as explained immediately below. (See the discussion of functions in Chapter 2 and the BEGIN SECTION command in Chapter 3.)

PARAM

The type code PARAM is an abbreviation of the term "parameter." It is required, if appropriate, to show that the item of data being described is a parameter for use in a routine used to obtain a function. (See the discussion of functions in Chapter 2, the command BEGIN SECTION in Chapter 3, and the type code FUNCT above.)

Specifically, when the program contains a section introduced by a BEGIN SECTION command, each data-name listed in the USING clause of that command is a parameter, and it must be described as such in the data description.

REDEF

The code REDEF is used whenever it is necessary to redefine an area or an item of data that has previously been defined in some other way. This is usually necessary whenever a portion of the program "overlaps" another—i.e., when it calls for the use, on a "time-sharing" basis, of data or storage space which has previously been defined for some other purpose. It is also used in setting up tables in the system.

For example, it may be necessary to call existing data by a new set of names, or to reorganize it by altering the groupings and/or the subordinate level numbers. Frequently it is necessary to wipe out data to make room for other data. In any such case, a new data description is required for the new items or the new names. However, the name or names of the areas being redefined must first be listed, using the type code REDEF to show that the accompanying data description may also be used to refer to the same area. The REDEF entry must have the same level number as the entry being redefined. An illustration will be found later in this section, in the discussion of tables, on page 75.

Use of the REDEF code does not erase data in storage, unless an attempt is made to place two or more different constants in the same area; however, it does superimpose a new format upon the data already present. If the programmer wishes to change an item in storage, such as a value in a table, he may do so by using a MOVE instruction that specifies the new data and the position in storage where it is to be placed.

Redefinition does not cancel the previous definition. It merely makes it possible to refer to the same area by different names and for different uses. Once an area has been defined, all names associated with the definition may be used at any time, regardless of subsequent redefinitions.

Tables

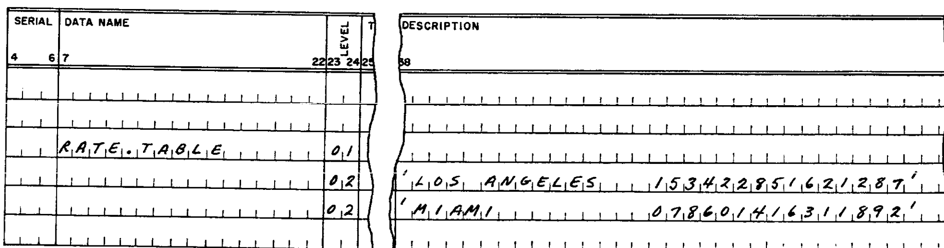
A valuable function of the REDEF code is to make it possible to set up tables in storage and to define the methods of locating individual items in the tables. The following example shows a method of doing this:

Suppose the programmer wishes to place in memory the table described in Chapter 2 (see page 29), which shows passenger transportation rates to each of 30 different cities. Each line in this table lists a city, a one-way rate, a round trip rate, and an excursion rate. If it were printed in a book of rate schedules, it would probably contain four columns, headed "City," "One-Way," "Round Trip," and "Excursion." A portion of this table might have the following form:

City	One-Way	Round Trip	Excursion
. . .	. . .	. . .	. . .
Los Angeles	153.42	285.16	212.87
Miami	78.60	141.63	118.92
. . .	. . .	. . .	. . .

To place such a table into a Commercial Translator program, the programmer must first analyze it to determine the maximum size of each kind of item. He may find, for example, that 14 character spaces will be required for the longest name in the "City" list, and that each of the rate listings can be accommodated if space for five digits is reserved for each rate value.

The actual data may then be entered by copying all of the individual items in sequence, reading across each line and allowing blank spaces (or zeros) as filler in those items that occupy less than the allowed space. Thus the initial data entry might read, in part:



SERIAL	DATA NAME	LEVEL	DESCRIPTION
	RATE.TABLE	01	
		02	'LOS ANGELES 153422851621287'
		02	'MIAMI 078601416311892'

The entries at the 02 level, which could extend over as many lines as necessary to accommodate all of the data, specify a series of constants. These are enclosed in quotation marks in accordance with the rules for writing constants. Note that decimal points are not normally required in constants of this kind, since the location of the decimal point will be specified in a later entry.

The result of this entry is twofold: Space is reserved in memory for the entire table, and the individual items are stored in a sequence which permits them to be located after the table has been more fully described. At this point, however, no means of identifying any single item of data has yet been established. This must be accomplished by superimposing the format and structure of the table on the data already placed in storage. It is done by redefining the format of the data, using a series of entries such as the following:

SERIAL (4–6)	DATA NAME (7–22)	LEVEL (23–24)	TYPE (25–30)	QUANTITY (31–35)	MODE (36)	JUSTIFY (37)	DESCRIPTION (38)
		01	REDEF				RATE.TABLE
	RATE	02		30			
	CITY	03					AAAAAAAAAAAAAA
	ONE.WAY	03					999V99
	ROUND.TRIP	03					999V99
	EXCURSION	03					999V99

In the first of these entries, the data previously defined as RATE.TABLE is redefined (by the type code REDEF) as a new body of data. In this case a name has not been given to the redefined data, but the programmer could assign one if he wished. It will be noted that the level of the new entry (01) is the same as that of the entry being redefined.

The RATE entry and the four data-names which follow it (CITY, ONE.WAY, ROUND.TRIP, and EXCURSION) are on lower levels (02 and 03); therefore, they will be understood to be subordinate elements of the entry at the 01 level, and the code REDEF will apply to them as well.

The number 30 in the "Quantity" columns, as will be explained later in this chapter, shows that space is to be reserved for 30 entries at the 02 level. In other words, there will be 30 items called RATE. It will be seen later that the programmer may single out any one of these items by appending a subscript to the name RATE. This subscript may be a literal, a data-name, or a limited arithmetic expression. The value represented by the subscript must be a positive integer, since it will be used within the system to count individual items until the required one is reached. (See the discussion of lists, tables, and subscripts in Chapter 2.)

It has been noted that the name RATE includes all of the four items immediately following it. Thus, each of the 30 RATE entries will contain one called CITY, one called ONE.WAY, another called ROUND.TRIP, and a fourth called EXCURSION. Since 30 separate RATE entries were specified, the processor will lay out this entire sequence 30 times.

The characters in the "Description" columns show the length and type of data to be expected in each field. This is explained more fully later in this chapter, but the effect of the entries given in the example is as follows: The 14 A's following the name CITY will reserve space in storage for names of up to 14 characters in length. The symbols 999V99 will reserve space for five-digit numeric values, with an assumed decimal point between the third and fourth digits.

Once the processor has this information, it can identify and use any single item of data in the table. It will know, for example, that the first 14 character spaces of each RATE listing contain the name of a destination city, that the next five show the corresponding one-way rate, and so on. In other words, it now has the means of locating any item by its position in storage, which, as has been noted already, is the way in which the system locates *all* items of data in storage.

The programmer may then call for any individual item by its name, using a subscript to specify which one of the 30 items having the same name is wanted. Thus, ONE.WAY (17) would obtain the one-way rate for the 17th city in the table. Usually, however, the subscript would be a variable, such as the name of a field containing a number.

Suppose, for instance, that the input record contains a field called DESTINATION, and that this field will contain, at object time, a numeric code representing one of the cities listed in the table. If the programmer writes MOVE ONE.WAY (DESTINATION) TO BILL.AMOUNT, the system will obtain whatever number has been placed in the DESTINATION field and will use it to determine which item in the table is wanted. It will then find that item and move it to the field called BILL.AMOUNT. The reader will have noted that this number determines the position of data by the simple process of counting lines; the code numbers used, therefore, must be assigned in a sequence corresponding to the lines of the table.

COPY

This type code is used to copy a data description previously defined in the program so that it can be used again elsewhere. This makes it possible to use a data description with new data-names and, if desired, new level numbers.

The COPY type code is used as follows: The new name of the data description entry is written in the "Data Name" columns of the new entry. The code COPY is placed in the "Type" columns. The data description to be copied is specified by writing its original name in the "Description" columns. This description must already have been read into the system for the COPY code to be able to operate on it.

The processor will then obtain the original data description and copy it in its entirety, except for the following modifications: (1) The original name will be replaced by the new name. (2) If a new level number has been specified for the new name, the level numbers of the original data description will be adjusted so that they retain their original relationship to the named entry. Thus, if the original sequence of level numbers had been 01, 03, 04, and if the new name is assigned level 05, the other items would now be placed at levels 07 and 08, respectively.

Suppose the programmer had previously written the following entries in the data description:

SERIAL 4–6	DATA NAME 7–22	LEVEL 23–24	TYPE 25
	PAY.RCD.MASTER	01	
	EMPLOYEE.NAME	02	
	JOB.TITLE	02	
	HOURLY.RATE	02	
	GROSS.PAY	02	
	TAXES	02	
	FICA	03	
	FED.INCOME	03	
	STATE.INCOME	03	
	NET.PAY	02	

Suppose then that he wishes to set up an identical data description for a detail record, except that the new description is to have the name PAY.RCD.DETAIL and it will be placed at level 02. He could write the following entry:

SERIAL 4–6	DATA NAME 7–22	LEVEL 23–24	TYPE 25–30	QUANTITY 31–35	MODE 36	JUSTIFY 37	DESCRIPTION 38
	PAY.RCD.DETAIL	02	COPY				PAY.RCD.MASTER

The effect of this entry would be as though the programmer had written an entirely new set of entries in the following form:

SERIAL 4 6	DATA NAME 7 22	LEVEL 23 24	TYPE 25
	PAY.RCD.DETAIL	02	
	EMPLOYEE.NAME	03	
	JOB.TITLE	03	
	HOURLY.RATE	03	
	GROSS.PAY	03	
	TAXES	03	
	FICA	04	
	FED.INCOME	04	
	STATE.INCOME	04	
	NET.PAY	03	

LABEL

The type code LABEL identifies a data description as that of a label record. This will cause a redefinition of the label area in the input-output control system. The actual use of this code will be explained in the publications pertaining to the various Commercial Translator processors as they apply to individual machine systems.

Quantity (Col. 31-35)

If an item of data will be followed by one or more additional items having the same data description, the programmer can avoid writing the additional data descriptions simply by specifying in Columns 31 through 35 the total number of times the data description is required. This number must refer to data items occurring in sequence.

The "Quantity" columns may be left blank, and, in fact, usually are. In that case, it will be assumed that they contain a value of 1, and the specified data description will be entered only once.

Since quantity numbers are used to specify sequences of data descriptions for use in lists and tables, and since data in tables is referred to by the use of subscripted names, quantity numbers should not be assigned to data items not having names, unless these items include named items at a lower level.

An example of the use of a quantity entry was included in the discussion of the REDEF type code when used to set up a table. (See page 75.) The reader will recall that the field called RATE, together with its subordinate fields, was to be entered 30 times. Accordingly, the value 30 was placed in the "Quantity" columns opposite the name RATE. Since each of the subordinate items (CITY, ONE.WAY, ROUND.TRIP, and EXCURSION) was required only once for each RATE entry, no value was placed in the corresponding "Quantity" columns. If, for some reason, one of the subordinate fields had been needed more than once, a quantity could have been specified.

Quantity numbers may be specified for as many as three levels in a single "nested" group. For example, assume that a program must deal with five items called STATE, that within each of these are four fields named DISTRICT, and that each DISTRICT contains seven smaller fields called CITY. The quantity 5 should then be written for STATE, 4 for DISTRICT, and 7 for CITY. The processor will then reserve storage space for a total of 5 STATE items, 20 DISTRICT items, and 140 CITY items (assuming, of course, that the level numbers show the proper relationships among the three entries).

To call for any one item of data, the programmer must write as many subscripts as are necessary to identify the particular level. Thus, STATE (3) would call for the third item called STATE, DISTRICT (3,2) would call for the second DISTRICT field within the third STATE item, and CITY (3,2,6) would obtain the sixth CITY field in the second DISTRICT of the third STATE. Subscripts are always written in descending order of level.

Mode (Col. 36)

The term "mode" refers to the method by which data is represented, such as the binary mode or the binary coded decimal mode. For the purposes of the Commercial Translator, the mode used in the arithmetic units of the system is considered to be the system's "internal" mode, although the system may be able to read data in other modes.

Column 36 is used to show whether the data being described is prepared in the internal mode for the system being used. In that case, the letter I is punched in this column. Data punched in an "external" mode is represented by the letter E. Specific information on the modes available for each system will be provided in the publications covering the processors for the various machine systems.

Since a body of data may contain information in more than one mode, the mode is specified at the lowest level of data organization—i.e., at the level where the "field pictorial" is shown. (See the "Description" columns.) Thus, if a record at the 01 level contains two fields at the 02 level, one of them in the internal mode and one in the external, there would be no point in specifying a mode for the record itself. The mode (or modes) of a larger unit is therefore a consequence of the mode specified for each of its components.

Justify (Col. 37)

The term "justification" is used in printing and writing to mean the alignment of a margin. Thus, the text of this manual is both "left justified" and "right justified." In data processing, the term has a similar implication, since it is usually necessary to specify that data be justified if the programmer wishes it to be printed out with an aligned margin.

Actually, however, justification means a good deal more than this. Specifically, it refers to the placement of the data in a unit of storage. In many systems, data is stored in machine "words" of fixed length. Very often, therefore, a particular item of data will be shorter than the space allowed for it, and it may be necessary, if alignment is to be preserved, to provide a means of filling the unoccupied spaces with non-significant characters, such as zeros or blanks, depending on the system. On the other hand, it may be desirable to fill all available space with data, so that more than one item—or parts of more than one item—may be stored in a given machine word. This is known as "packing."

It is extremely important to recognize that justification specified for an *input* item of data—i.e., an item to be entered into the system at object time—describes the item as it already exists. It does not change the justification; it is used in informing the system where to look for the incoming data. Justification specified for other items, however, actually controls the placement of the data. It instructs the system how to handle the data when it operates on it internally or prepares it for output.

The effect of specifying justification is as follows: If left justification is specified, the data item is stored, or will be stored, so that its left-hand character is placed in the left-hand position of the next available machine word. This may mean that the right-hand portion of the preceding word is left unoccupied. If right justification is stipulated, the data item is stored, or will be stored, to the right in the next available machine word, leaving unoccupied whatever portion of that word is not required at

the left. If no justification is specified, the data is packed (if it is incoming data) or will be packed—there will be no blank spaces between successive items.

Packing makes efficient use of storage space, but it may make it more difficult for the program to obtain the items independently of each other, since the processor may have to provide for "unpacking." However, packed items can usually be read or written more rapidly, since the system will not have to process non-significant information. In general, if the programmer must be economical of storage space, or if the program calls extensively for reading or writing long sequences of data, it may be more efficient to pack the data. If a great many of the items must be obtained individually, however, and/or if scanning long sequences of data is not a common operation, justification of data may be more efficient. The programmer should evaluate each case on its particular merits.

Alignment of alphameric information, such as lists of names, cannot be specified in the field pictorial. If alignment of such data is required, it can be accomplished by the use of the "Justify" column.

The symbols L and R are used in Column 37 to indicate left and right justification respectively. If neither symbol is used, the data will be packed, as has already been noted.

Description (Col. 38-71)

The "Description" columns are used to show the length of each field of data, together with its format. They may also be used to specify constants and the names of data and procedures, as explained below. Specifically, the following kinds of information may appear in Columns 38 through 71:

1. Format characters. These are shown and described in the table below.
2. Constants.
3. Data names associated with the type codes REDEF and COPY.
4. The word LIBRARY, followed by the name of a data description stored in the library.
5. The words QUANTITY IN, followed by the name of a field which will contain a quantity when the object program is run.

In a number of cases, a complete data description entry will require that more than one of these kinds of information be listed on the same line. For example, it is generally necessary to show both the format and the value of a constant. In such a case, the various items should be written in the order shown above, separated by one or more blanks.

Format Characters

The format characters serve two functions: (1) They show the number of character spaces to be occupied by a field. (2) They show the kind of character that will occupy each space. The resulting data representation is known as a "field pictorial."

If the item of data being described is one which will be brought into the system at object time, the format characters must reflect the format of the data as it already exists; changes in input data cannot be effected by the field pictorial. However, if the item is one produced as a result of the operation of the program—as in moving the data or performing arithmetic on it, for example—the field pictorial has a direct effect on the manner in which the data will be handled.

With certain exceptions, which are explained below, one format character is required for each data character for which storage space is to be reserved. The particular format character chosen for each space prepares the system to receive in that space data of the type shown in the table on the following page.

Format Character	Meaning and Use
A	Any non-numeric character, including the blank.
X	Alphameric character (any character in the machine's character set).
9	Any numeric character.
8	Numeric character, to be replaced automatically by a blank whenever it is a non-significant zero.
*****	Numeric character, to be replaced automatically by an asterisk whenever it is a non-significant zero.
V	Assumed decimal point. This character informs the processor where the decimal point is located, for purposes of calculation and/or alignment of values. The symbol is not required for integers. The symbol V will not reserve an actual space in storage.
.	True decimal point. This character will reserve an actual space in storage.
S	Scale factor. This symbol is used as a "filler" or "spacer" when the input data does not show the position of the decimal point. E.g., a field containing percentages from 1 to 9 would be represented by the notation VS9; this would assure that the values 1 to 9 would be interpreted as .01 to .09. Similarly, if a field contains values that represent thousands, each unspecified digit must be represented by an S; thus, the notation 999SSS would provide for values from 000,000 to 999,000, even though the three right-hand zeros would not appear as input.
$	Dollar sign. An actual dollar sign will be placed in the indicated position, provided it is not followed by the symbol 8. In the latter case, the dollar sign will "float"—i.e., it will be placed immediately to the left of the first significant digit remaining.
,	True comma. This symbol will reserve an actual space in storage, to be occupied by the comma. The comma itself may be replaced by a blank, asterisk, or dollar sign, if the operation of a preceding 8 or * has resulted in the elimination of non-significant zeros to the left.
+	Plus *or* minus sign, one of which will always be placed in the space reserved for it, depending on whether the value is positive or negative. (Compare with use of minus sign, described below.) This sign may be placed in a column by itself, in which case it will reserve an actual space in storage. Alternatively, it may be entered as an "overpunch" with either of the format characters 8 or 9, in either the units or high-order position of a field; in this case, a special space will not be reserved, and the sign of the field will be indicated in accordance with the operating characteristics of the particular system.
—	Minus sign, to be placed in the space reserved for it when the value is negative; when the value is positive, the space will be left blank. If punched in a space by itself, this symbol will reserve a space in memory; otherwise, it may be "overpunched" and will act as described in the rules for the symbol +.
F	Floating point number. This symbol does not reserve an actual space in storage; it informs the processor that the number being described is a floating point number. It is placed between the format characters representing the fraction and those representing the exponent. E.g., +99V9F+99.
(n)	A number placed in parentheses immediately following one of the other format characters instructs the processor to allow for that number of the character specified. E.g., 9(4)A(12) is equivalent to 9999AAAAAAAAAAAA.

EXAMPLES

The following examples indicate the range of characters for which provision is made by the notation shown:

Notation	Range of Characters Provided For
AAA	All characters in the machine's character set except numerals.
88999	All numeric values from 000 to 99999.
****.99	All numeric values from ****.00 to 9999.99.
$888,888.99	All numeric values from $.00 to $999,999.99.

Constants

A constant is a value, or a group of symbols, placed in the program for use without alteration, such as a fixed percentage rate or a fixed name. A constant may be placed in the system by writing a data description entry for it which includes both a statement of its format (using the format characters) and a statement of the actual value or group of symbols. The format is specified by a standard field pictorial entry. The actual value, or the actual symbols, must then be written on the same line, separated from the field pictorial by at least one blank. This value (or this group of symbols) must be enclosed in quotation marks.

While a literal may be thought of as a form of a constant, the reader should note two distinctions: (1) Literals are written in procedure statements, whereas constants are placed in the system by means of data description entries. (2) Constants, unlike literals, must always be enclosed in quotation marks, even though they may be wholly numeric.

To illustrate, if the programmer wished to place the percentage .05 into the data description as a constant, it would be necessary to specify the nature of the constant by a notation such as the following:

SERIAL 4 6	DATA NAME 7 22	LEVEL 23 24	TYPE 25 30	QUANTITY 31 35	MODE 36	JUSTIFY 37	DESCRIPTION 38
	PERCENT.CONST						V99 '.05'

In this example, the notation V99 is, of course, the field pictorial, and it is followed by a statement of the actual value of the constant.

Data Names Associated with REDEF and COPY

When the type codes REDEF and COPY are used, it is necessary to specify the name of the data item or area to be redefined or copied. This name must be entered in the "Description" columns. (See the discussion of those type codes earlier in this chapter.)

Library Names

The word LIBRARY, followed by the name of a data description stored in the library, designates a data description which is to be copied. Availability of this code enables the programmer to store data descriptions which are frequently required, so that he can save the effort of writing new ones when a previous one will serve his purpose. When a library data description is prescribed, the type code COPY must be used.

Quantities Specified in Named Fields

It has been shown that if a value is placed in the "Quantity" columns, the processor will reserve space for repeating the specified data description until it appears the number of times indicated by that value. In certain special cases, the programmer may wish to alter the use of such an area at the time the object program is run. The storage area, of course, will already have been reserved by the action of the processor, and at object time it is no longer possible to set aside new areas. However, for certain specialized purposes, it is possible to regroup the components of an existing area by using the QUANTITY IN option. In this case, a quantity value is placed in a named field and the program is referred to this field by the words QUANTITY IN followed by the name of the field.

This usage is not required in most programs. It represents an advanced programming technique, providing additional facilities which can be used effectively by a skilled programmer. While it is not the purpose of this manual to explain the many refinements and uses of this technique, its general nature is indicated by the following discussion:

Suppose that a certain data description having the field pictorial 999V99 is to be repeated until it appears 100 consecutive times. Each iteration of the data description will reserve 5 character spaces in storage, and therefore a total of 500 spaces will be reserved altogether. Suppose that this area is referred to by the name TABLE.

Suppose, then, that the programmer wishes to set up a number of different tables at different times during the running of the object program, using this same storage area for each. Assume that the first, referred to in this text as Table 1, consists of 10 columns and 10 lines—i.e., there will be 10 data items on each of 10 lines, and each item will have a field pictorial of 999V99. This table could have been set up by the original data description for the area in a series of entries such as the following:

SERIAL 4 6	DATA NAME 7 22	LEVEL 23 24	TYPE 25 30	QUANTITY 31 35	MODE 36	JUSTIFY 37	DES 38
	TABLE	01					
	COLUMN	02		10			
	ITEM	03		10			

Once the table is described, the programmer may refer to individual items in it by the use of subscripts, the first subscript referring to the item at the higher level. Thus, ITEM (4, 2) would refer to the second item of the fourth COLUMN group.

When the system is directed to obtain this item, it relies on a technique of counting. Specifically, it would rely on the fact that ten items are specified for each column. Thus, it would count off the first ten items, then the ten items of the second column, the ten items of the third column, and it would then count to the second item following. In other words, it would obtain the 32nd item in the string of data representing the table.

However, it is supposed that at some other time in the program the programmer wishes to set up another table (Table 2) which uses the same field pictorial for each item but which requires a different grouping of columns and lines. Suppose Table 2 has 20 columns and 5 lines. The programmer could theoretically write the following entries:

SERIAL 4–6	DATA NAME 7–22	LEVEL 23–24	TYPE 25–30	QUANTITY 31–35	MODE 36	JUSTIFY 37	DES 38
	TABLE	01					
	COLUMN	02		20			
	ITEM	03		5			

However, these entries would reserve space in addition to, not in place of, the original table.

Suppose, however, that the programmer, in setting up Table 1, had written the following entries:

SERIAL 4–6	DATA NAME 7–22	LEVEL 23–24	TY 25	...	QUANTITY 35	MODE 36	JUSTIFY 37	DESCRIPTION 38
	TABLE	01						
	COLUMN	02						QUANTITY IN COLUMN.QTY
	ITEM	03			100			QUANTITY IN ITEM.QTY

Suppose, further, that the names COLUMN.QTY and ITEM.QTY are the names of fields into which values may be placed. In this case, the processor would first note the value of 100 in the "Quantity" column opposite the name ITEM and would reserve space for a total of 100 items to be placed in the table. It would cause the object program to look in the fields COLUMN.QTY and ITEM.QTY to determine the actual quantities for the entries COLUMN and ITEM. These values would override any values previously placed in the "Quantity" columns. If the programmer wished the first table to have 10 columns of 10 items each, he would have to be sure that a value of 10 was placed in both the COLUMN.QTY and ITEM.QTY fields. Obviously, if Table 1 were never to be replaced by another table, it would be superfluous to use QUANTITY IN entries, for the quantities could be specified directly in the data description.

However, since the QUANTITY IN entries have been made, it is now an easy matter to construct Table 2. All that is necessary is to place the values 20 and 5 in COLUMN.QTY and ITEM.QTY respectively. If, then, the expression ITEM (4,2) is written, the system will know that instead of counting ten items to the column it must count five. Thus, instead of obtaining the 32nd item, it will obtain the 17th, which is the one required.

If the programmer wished to set up a third table in the same area, with four columns and 25 lines, he could place the values 04 and 25 in the corresponding fields, and, once again, subscripts written in accordance with the new table structure would be correctly calculated by the system for locating individual data items.

General Note: If the description of a data item overflows from the "Description" columns, it may be continued on the next line, following the rules given for the continuation indication column (Column 72). The break at the end of a line must occur between words, since the processor will assume a blank at the end of each line. Multiple blanks, however, are treated as single blanks. If a constant is to be carried over onto a new line, the portion on each line must be treated as a complete constant (i.e., enclosed in quotation marks); the continuation indication is not used in this case, and no blanks will be assumed between successive lines. (Note the example used on page 74 to show how a table of rates might be placed in storage.)

Cont. (Col. 72)

If the description to be entered in either the "Data Name" or "Description" columns (Columns 7 through 22 or Columns 38 through 71) is too large for the space allowed, it may be continued on a following line. So that the processor will recognize this situation, the programmer must enter a character of some sort in Column 72. Any non-blank character will suffice. The processor will then interpret the entry in the succeeding card as a consecutive part of the previous entry.

A continued name must, of course, be placed in the "Data Name" columns, and an overflowing description must be continued in the "Description" columns. The rules for determining where the text should break between lines are given in the discussions of the entries to be made in those columns.

Identification (Col. 73-80)

Columns 73 through 80 are provided for the optional use of the programmer should he wish to place a code on the cards to identify the program of which they are a part. Any characters from the basic character set may be used, since the characters in these columns have no effect on either the processor or the object program.

Storage Areas

It has been pointed out that the internal storage of an electronic data processing system may be used to contain data of various kinds, including the program which governs the processing. As a result, those who work regularly with the technical aspects of programming are accustomed to thinking of different kinds of storage.

For example, when an input record is brought into storage, space must be reserved for the original record before any processing is carried out. This may be thought of as an input area.

Then, after processing begins, it is often necessary to move data from the input area into an area where it can be worked on. This is like moving a ledger card from a file to a writing desk where an entry will be made or a value computed. "Working storage" is therefore required in many cases. Actually, the registers of the system provide space of this kind in many operations, and since in these cases it is provided automatically, it is not thought of as working storage. However, in other cases, it may be necessary to reserve a specific amount of space as working storage. For example, a computation will sometimes produce intermediate results which will be needed later in the program but which are never needed as such in the output record. It may therefore be necessary to reserve an area in which to store such results temporarily.

Another necessary use of storage space is for the assembling of output records. Normally, as each phase of a program is completed, the results will be moved to an output area until the complete record has been assembled. The various output fields may differ in format, size, and sequence from those of the input record, and data may have been added or deleted. Thus, an area of storage must usually be set aside for assembling the output record.

Other storage areas may be used for reference data, such as constants, literals, and tables.

The experienced programmer often finds it convenient to distinguish among these various kinds of storage. Actually, of course, all storage areas are controlled by the same basic techniques—data is always addressed by its location, and data in any area may be governed by any of the system's basic operating instructions. Since the Commercial Translator system eliminates the need for the programmer to keep track of specific storage areas, it also eliminates, for the most part, the need to distinguish between types of storage areas. Storage areas are automatically reserved when the

data description is written, regardless of how the area is to be used. Certain special provisions, especially those governing input and output, are built into the processor for each system, and these are described in the manuals for the various processors.

The programmer, however, must be sure that all data-names required for input and output, as specified in the manual for the system he is using, are properly described in the data description. He should also examine his program to be sure that every item of data used in the program, whether as input, output, or for intermediate operations, is properly described. The processor will then make all necessary provisions for storage space and for identifying the data to be stored.

Appendix 1: Programming Example

Introduction

The Commercial Translator system is so designed that the programmer can analyze a business problem in terms of the problem itself, rather than in terms of the equipment on which it will be processed. He can rely on the processor to convert his source program into a machine program which will take advantage of the operating characteristics of the equipment. Thus, his first approach to a problem will be one in which he determines each of the basic steps needed to solve it, and the sequence in which they must occur.

The basic device for showing the flow of information and the sequence of operations in a simple form is the *flow chart*. In addition, a *process chart* will be useful in determining how data should be grouped for input to the system and what data groupings are to be obtained from it. From these two charts the programmer can proceed easily to writing the source program itself.

To illustrate these three phases, the following pages contain a flow chart, a process chart, and a sample program that could theoretically be used for a simplified payroll operation. This program is intended only to illustrate the general method of using the Commercial Translator language and to show how the data to be used in a program might be defined in a typical data description division. It is in no sense an attempt to demonstrate an ideal solution to a payroll problem.

As the process chart shows, there will be two input files—a master file and a detail file. These will be on magnetic tape. The output files, also on tape, will be the following: (1) An updated master file, which will be used as input in the next running of the program. (2) A payroll report file, which will be used to produce a printed report. (3) A check file, which will control the printing and punching of pay checks. (4) An exception, or error, file, which will contain any master or detail record for which no matching detail or master has been found.

Both the master and the detail files consist of individual pay records for each employee, arranged in ascending sequence by employee number. It is assumed that there will be a master record for each detail record with the same employee number, and a detail record for each master record. Any exceptions will be given an error code and moved to the exception file for subsequent correction.

Each master record contains, among other information, hourly pay rate and year-to-date totals for gross pay, retirement premiums, and insurance. Among the items included in each detail record is a statement of the number of hours worked during the current pay period; this, together with the hourly rate in the master record, is used to compute gross pay. The records also contain other data, such as employee name, department, information to be used in performing withholding tax, FICA, bond deduction and other computations, and so on.

Each of these items of data is named and defined in the data description portion of the program, in accordance with the rules given in Chapter 4 of this manual.

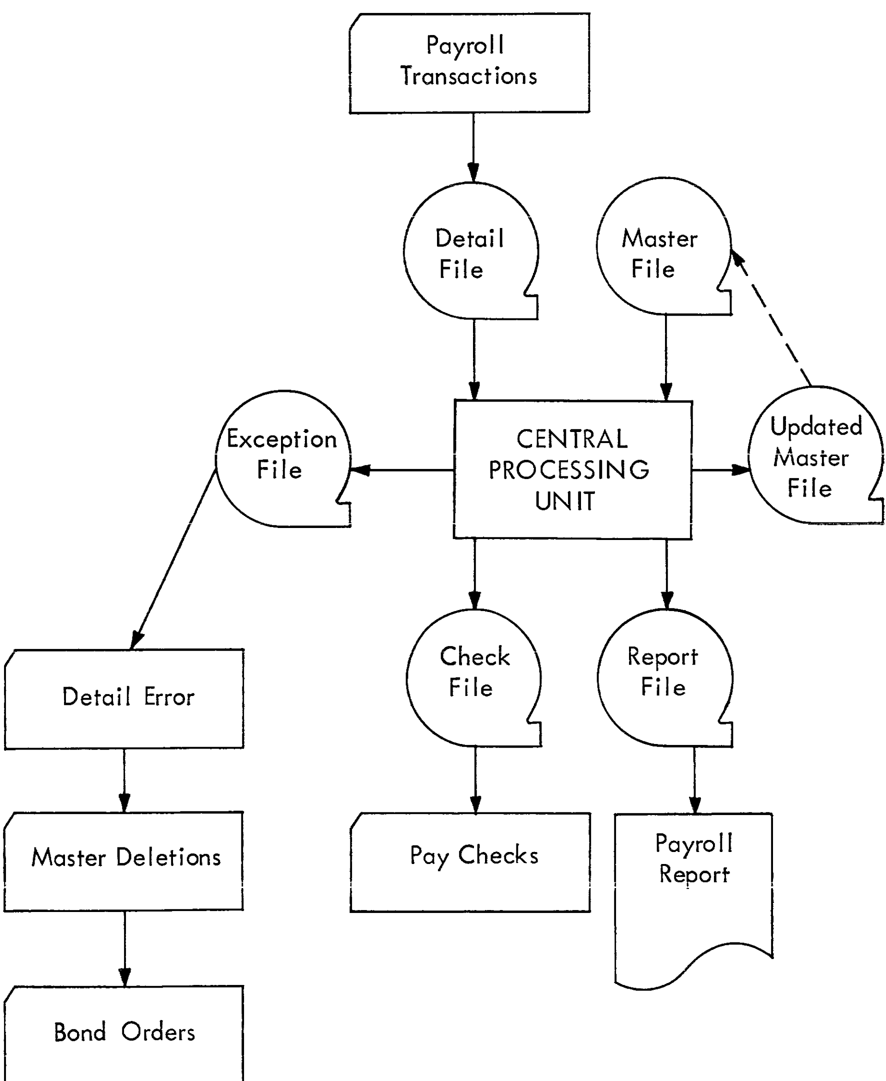
The actual procedure statements, which in this case are written before the data description entries, follow the rules given in Chapter 3. The program begins with a renaming of data items, using the CALL command, so that they can be referred to by abbreviated names. It then proceeds to a comparison of the employee numbers used to control the processing, and the reader will see that provisions are made to transfer

to various routines (using the "conditional" GO TO command) depending on whether the numbers are equal or unequal.

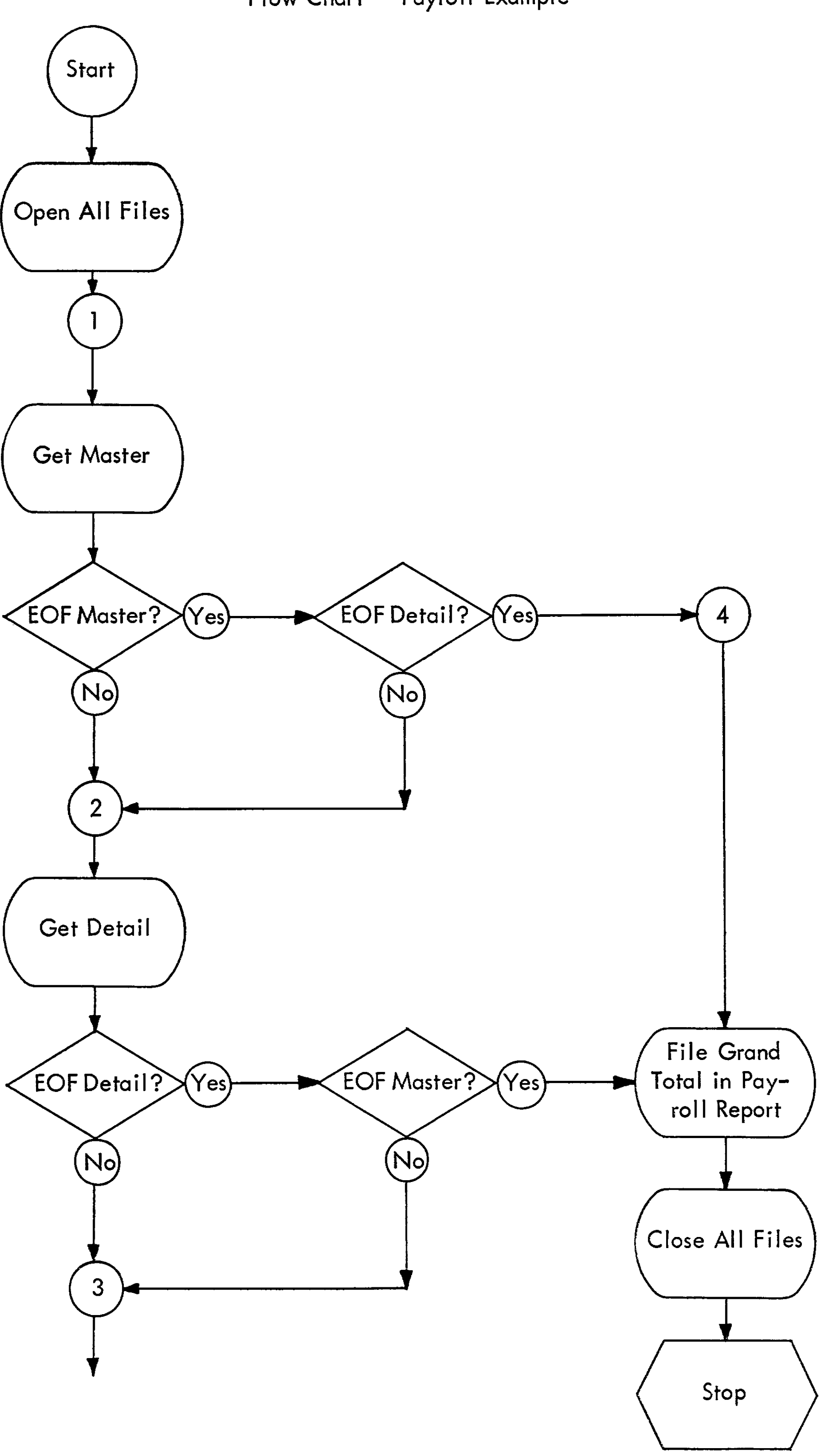
The sequence of procedures from that point on closely parallels that of the flow chart. As the reader will see, the program contains a number of complete routines which are performed when necessary, regardless of the sequence in which they appear in the program. The GO TO, DO, and BEGIN SECTION commands are used to call for them.

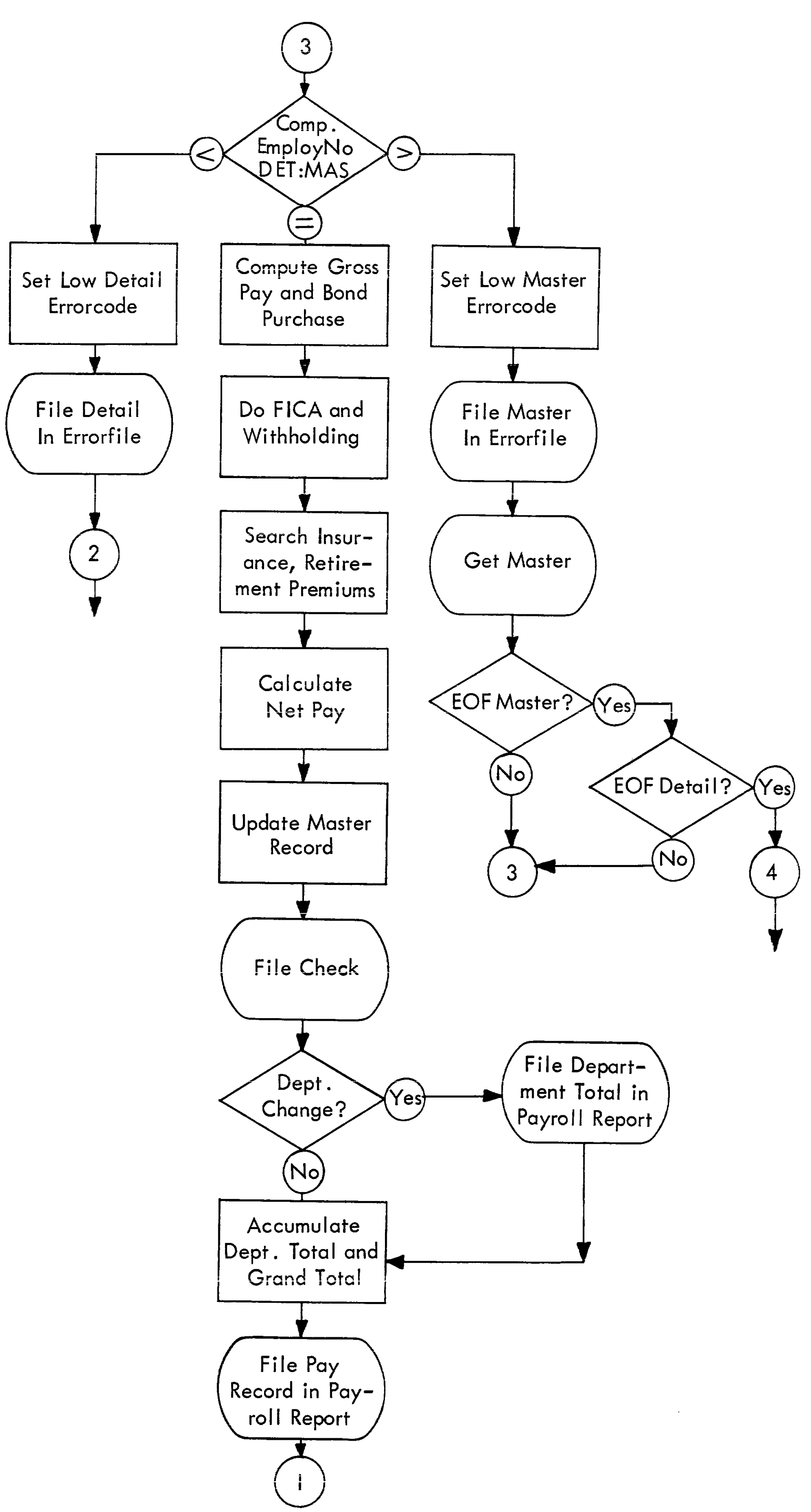
While this program is very much simplified, it does illustrate many of the basic techniques used in programming. The reader will find it helpful to review portions of Chapters 2, 3, and 4 as new concepts are presented in the procedure statements.

Process Chart — Payroll Example



Flow Chart — Payroll Example





3
Comp. EmployNo DET:MAS
<
>
=
Set Low Detail Errorcode
File Detail In Errorfile
2
Compute Gross Pay and Bond Purchase
Do FICA and Withholding
Search Insurance, Retirement Premiums
Calculate Net Pay
Update Master Record
File Check
Dept. Change?
Yes
No
File Department Total in Payroll Report
Accumulate Dept. Total and Grand Total
File Pay Record in Payroll Report
1
Set Low Master Errorcode
File Master In Errorfile
Get Master
EOF Master?
Yes
No
3
EOF Detail?
Yes
No
4

IBM

COMMERCIAL TRANSLATOR PROCEDURE DESCRIPTION

CTL. 1 3	PROGRAM	SYSTEM	PAGE 1 OF 10
010	SAMPLE PAYROLL		
	PROGRAMMER	DATE	IDENT. 73 80

SERIAL 4 6	PROCEDURE NAME 7 12	TEXT 13 72

```
01 *PROCEDURE
02        CALL (EMPLOYEE.NUMBER)  EMPLOYNO,
03             (BONDEDUCTION)     BONDEDUCT,
04             (BONDENOMINATION)  BONDENOM,
05             (BONDACCUMULATION) BONDACCUM,
06             (INSURANCE.PREM)   INSPREM,
07             (RETIREMENT.PREM)  RETPREM,
08             (DEPARTMENT.TOTAL) DPT.
09 START.  OPEN ALL FILES.

10 GET.MASTER.  GET MASTER,  AT END DO END.OF.MASTERS.

11 GET.DETAIL.  GET DETAIL,  AT END GO TO END.OF.DETAILS.

12 COMPARE.EMPLOYEE.NUMBERS.  GO TO COMPUTE.PAY WHEN DETAIL EMPLOYNO
13        IS EQUAL TO MASTER EMPLOYNO,  LOW.DETAIL WHEN DETAIL
14        EMPLOYNO IS LESS THAN MASTER EMPLOYNO.

15 HIGH.DETAIL.  MOVE 'M' TO MASTER ERRORCODE,  FILE MASTER IN
16        ERROR.FILE.
17          GET MASTER,  AT END DO END.OF.MASTERS.
18          GO TO COMPARE.EMPLOYEE.NUMBERS.
```

IBM COMMERCIAL TRANSLATOR PROCEDURE DESCRIPTION

CTL. 1–3	PROGRAM	SYSTEM	PAGE	IDENT. 73–80
020	SAMPLE PAYROLL		2 OF 10	
	PROGRAMMER	DATE		

SERIAL (4–6)	PROCEDURE NAME (7–12)	TEXT (13–72)
01	LOW-DE	TAIL. MOVE 'D' TO DETAIL ERRORCODE, FILE DETAIL IN
02		ERROR-FILE
03		GO TO GET-DETAIL.
04	END-OF	-MASTERS. IF DETAIL EMPLOYNO = HIGH-VALUE THEN GO TO
05		END-OF-RUN OTHERWISE SET MASTER EMPLOYNO = HIGH-VALUE.
06	END-OF	-DETAILS. IF MASTER EMPLOYNO = HIGH-VALUE THEN GO TO
07		END-OF-RUN OTHERWISE SET DETAIL EMPLOYNO = HIGH-VALUE, GO
08		TO COMPARE-EMPLOYEE-NUMBERS.
09	END-OF	-RUN. MOVE CORRESPONDING GRAND-TOTAL TO PAYRECORD, FILE
10		PAYRECORD, CLOSE ALL FILES.
11		STOP 1234.
12	COMPUT	E-PAY. IF DETAIL HOURS IS GREATER THAN 40 THEN SET DETAIL
13		GROSS = (DETAIL HOURS - 40) * MASTER RATE * 1.5.
14		SET DETAIL GROSS = DETAIL GROSS + MASTER RATE * 40, DO
15		FICA-ROUTINE, DO WITHOLDING-TAX-ROUTINE.
16		IF MASTER BONDEDUCT IS NOT EQUAL TO ZERO THEN DO
17		BOND-ROUTINE.
18		DO SEARCH FOR INDEX = 1(1)12.
19	NET.	SET PAYRECORD NETPAY = DETAIL GROSS - DETAIL FICA - DETAIL
20		WHT - DETAIL RETIREMENT - DETAIL INSURANCE - DETAIL
21		BONDEDUCT.

IBM COMMERCIAL TRANSLATOR PROCEDURE DESCRIPTION

CTL. 1–3	PROGRAM	SYSTEM	PAGE 3 OF 10
030	SAMPLE PAYROLL	DATE	IDENT. 73–80
	PROGRAMMER		

SERIAL 4–6	PROCEDURE NAME 7–12	TEXT 13–72
01		ADD CORRESPONDING DETAIL TOTALS TO MASTER TOTALS, MOVE
02		CORRESPONDING DETAIL TO PAYRECORD, CHECK, MOVE PAYRECORD
03		NETPAY TO CHECK AMOUNT.
04		FILE CHECK.
05		IF PAYRECORD DEPARTMENT IS GREATER THAN CURRENT DEPARTMENT
06		THEN MOVE BLANKS TO PAYRECORD EMPLOYNO, PAYRECORD NAME,
07		MOVE CORRESPONDING DEPARTMENT.TOTAL TO PAYRECORD, FILE
08		PAYRECORD, MOVE ZEROS TO DPT HOURS, DPT GROSS, DPT WHT,
09		DPT FICA, DPT BONDEDUCT, DPT INSPREM, DPT RETPREM, DPT
10		NETPAY, DPT BONDPURCHASES.
11		MOVE CORRESPONDING DETAIL TO PAYRECORD, CURRENT, ADD
12		CORRESPONDING DETAIL TO DEPARTMENT.TOTAL, GRAND.TOTAL,
13		FILE PAYRECORD.
14		GO TO GET.MASTER.
15	FICA.ROUTINE.	BEGIN SECTION.
16		IF MASTER FICA + 0.03 * DETAIL GROSS IS LESS THAN 144.00
17		THEN SET DETAIL FICA = 0.03 * DETAIL GROSS OTHERWISE SET
18		DETAIL FICA = 144.00 - MASTER FICA.
19		ADD DETAIL FICA TO MASTER FICA.
20		END FICA.ROUTINE.

IBM

COMMERCIAL TRANSLATOR PROCEDURE DESCRIPTION

CTL. 1 3	PROGRAM	SYSTEM	PAGE 4 OF 10
040	SAMPLE PAYROLL		
	PROGRAMMER	DATE	IDENT. 73 80

SERIAL 4 6	PROCEDURE NAME 7 12	TEXT 13 72
01	WITHOL	DING.TAX.ROUTINE. BEGIN SECTION.
02		IF 13 * MASTER EXEMPTIONS IS LESS THAN DETAIL GROSS THEN
03		SET DETAIL WHT = 0.18 * (DETAIL GROSS - 13 * MASTER
04		EXEMPTIONS) OTHERWISE SET DETAIL WHT = ZEROS.
05		ADD DETAIL WHT TO MASTER WHT.
06		END WITHOLDING.TAX.ROUTINE.
07	BOND.R	OUTINE. BEGIN SECTION.
08		ADD MASTER BONDEDUCT TO MASTER BONDACCUM.
09	BOND.C	ALCULATION. IF MASTER BONDENOM IS NOT GREATER THAN MASTER
10		BONDACCUM THEN SET MASTER BONDACCUM = MASTER BONDACCUM -
11		MASTER BONDENOM OTHERWISE GO TO BOND.END.
12		MOVE CORRESPONDING MASTER TO BONDORDER, ADD BONDORDER
13		BONDENOM TO DPT BONDPURCHASES, MOVE BONDORDER BONDENOM TO
14		PAYRECORD BONDENOM, FILE BONDORDER IN ERROR.FILE.
15		GO TO BOND.CALCULATION.
16	BOND.E	ND. END BOND.ROUTINE.
17	SEARCH	. BEGIN SECTION.
18		IF MASTER RATE GT TABLE.ITEM RATE (INDEX) THEN GO TO
19		SEARCH.END OTHERWISE ADD TABLE.ITEM INSPREM (INDEX) TO
20		DETAIL INSURANCE, ADD TABLE.ITEM RETPREM (INDEX) TO DETAIL
21		RETIREMENT, GO TO NET.
22	SEARCH	.END. END SEARCH.

IBM

COMMERCIAL TRANSLATOR DATA DESCRIPTION

CTL. 1 3	PROGRAM	SYSTEM	PAGE 5 OF 10
050	SAMPLE PAYROLL		
	PROGRAMMER	DATE	IDENT. 73 80

SERIAL 4 6	DATA NAME 7 22	LEVEL 23 24	TYPE 25 30	QUANTITY 31 35	MODE 36	JUSTIFY 37	DESCRIPTION 38 71	CONT. 72
01	*DATA							
02	MASTER	1	RECORD			L		
03	ERRORCODE	2					A	
04	DATA	2						
05	EMPLOYEE.NUMBER	3						
06	DEPARTMENT	4					99	
07	EMPLOYEE	4					9999	
08	NAME	3					A(15)	
09	RATE	3					99V999	
10	DATE	3						
11	MONTH	4					99	
12	DAY	4					99	
13	YEAR	4					99	
14	EXEMPTIONS	3					99	
15	TOTALS	3						
16	GROSS	4					99999V99	
17	RETIREMENT	4					999V99	
18	INSURANCE	4					999V99	
19	FICA	3					999V99	
20	WHT	3					9999V99	
21	BONDEDUCTION	3					99V99	
22	BONDACCUMULATION	2					999V99	
23	BONDENOMINATION	2					999V99	
24		2					AAA	

IBM COMMERCIAL TRANSLATOR DATA DESCRIPTION

CTL. (1–3)	PROGRAM	SYSTEM	PAGE 6 OF 10
060	SAMPLE PAYROLL		
	PROGRAMMER	DATE	IDENT. (73–80)

SERIAL (4–6)	DATA NAME (7–22)	LEVEL (23–24)	TYPE (25–30)	QUANTITY (31–35)	MODE (36)	JUSTIFY (37)	DESCRIPTION (38–71)	CONT. (72)
01	DETAIL	1	RECORD			L		
02	ERRORCODE	2					A	
03	HOURS	2					99V9	
04	DATA	2						
05	EMPLOYEE.NUMBER	3						
06	DEPARTMENT	4					99	
07	EMPLOYEE	4					9999	
08	NAME	3					A(15)	
09	DATE	3						
10	MONTH	4					99	
11	DAY	4					99	
12	YEAR	4					99	
13	EXEMPTIONS	3					99	
14	TOTALS	3						
15	GROSS	4					99999V99	
16	RETIREMENT	4					999V99	
17	INSURANCE	4					999V99	
18	FICA	3					999V99	
19	WHT	3					9999V99	
20	BONDEDUCTION	3					99V99	
21	BONDACCUMULATION	2					999V99	
22	BONDENOMINATION	2					999V99	
23		2					AAAAA	

IBM

COMMERCIAL TRANSLATOR DATA DESCRIPTION

CTL. (1–3)	PROGRAM	SYSTEM	PAGE
070	SAMPLE PAYROLL		7 OF 10
	PROGRAMMER	DATE	IDENT. 73–80

SERIAL (4–6)	DATA NAME (7–22)	LEVEL (23–24)	TYPE (25–30)	QUANTITY (31–35)	MODE (36)	JUSTIFY (37)	DESCRIPTION (38–71)	CONT. (72)
01	BONDORDER	1	RECORD			L		
02	EMPLOYEE.NUMBER	2					9(6)	
03	DATE	2					9(6)	
04	BONDENOMINATION	2					999V99	
05	NAME	2					A(15)	
07	CHECK	1	RECORD			L		
08	EMPLOYEE.NUMBER	2						
09	DEPARTMENT	3					99	
10	EMPLOYEE	3					9999	
11	NAME	2					A(15)	
12	AMOUNT	2					$***9.99	

IBM

COMMERCIAL TRANSLATOR DATA DESCRIPTION

CTL. 1–3	PROGRAM	SYSTEM	PAGE	OF
080	SAMPLE PAYROLL		8	10
	PROGRAMMER	DATE	IDENT. 73–80	

SERIAL 4–6	DATA NAME 7–22	LEVEL 23–24	TYPE 25–30	QUANTITY 31–35	MODE 36	JUSTIFY 37	DESCRIPTION 38–71	CONT. 72
01	PAYRECORD	1	RECORD			L		
02	EMPLOYEE.NUMBER	2						
03	DEPARTMENT	3					99	
04		3					A	
05	EMPLOYEE	3					9999	
06		2					A	
07	NAME	2					A(15)	
08	DATE	2						
09		3					A	
10	MONTH	3					99	
11		3					A	
12	DAY	3					99	
13		3					A	
14	YEAR	3					99	
15	HOURS	2					8889.9-	
16	GROSS	2					$88889.99-	
17	WHT	2					$88889.99-	
18	FICA	2					$8889.99-	
19	BONDEDUCTION	2					$8889.99-	
20	INSURANCE	2					$8889.99-	
21	RETIREMENT	2					$8889.99-	
22	NETPAY	2					$88889.99-	
23	BONDENOMINATION	2					$88899.99-	

IBM

COMMERCIAL TRANSLATOR DATA DESCRIPTION

CTL. (1–3): 090
PROGRAM: SAMPLE PAYROLL
PROGRAMMER:
SYSTEM:
DATE:
PAGE 9 OF 10
IDENT. 73–80:

SERIAL 4–6	DATA NAME 7–22	LEVEL 23–24	TYPE 25–30	QUANTITY 31–35	MODE 36	JUSTIFY 37	DESCRIPTION 38–71	CONT. 72
01	DEPARTMENT.TOTAL	1	RECORD			L		
02	HOURS	2					9999V9	
03	GROSS	2					9(5)V99	
04	WHT	2					9(5)V99	
05	FICA	2					9999V99	
06	BONDEDUCTION	2					9999V99	
07	INSURANCE.PREM	2					9999V99	
08	RETIREMENT.	2						X
09	PREM						9999V99	
10	NETPAY	2					9(5)V99	
11	BONDPURCHASES	2					9(5)V99	
12	GRAND.TOTAL	1	COPY				DEPARTMENT.TOTAL	

IBM

COMMERCIAL TRANSLATOR DATA DESCRIPTION

CTL. (1–3)	PROGRAM	SYSTEM	PAGE 10 OF 10
100	SAMPLE PAYROLL	DATE	IDENT. (73–80)
	PROGRAMMER		

SERIAL (4–6)	DATA NAME (7–22)	LEVEL (23–24)	TYPE (25–30)	QUANTITY (31–35)	MODE (36)	JUSTIFY (37)	DESCRIPTION (38–71)	CONT. (72)
01	TABLE	1				L		
02		2					'0099908006001499100060'	
03		2					'0199912009002499150090'	
04		2					'0299915012003499200120'	
05		2					'0399920015004499250150'	
06		2					'0499930018006499300250'	
07		2					'0799930035099999300500'	
08		1	REDEF				TABLE	
09	TABLE.ITEM	2		12				
10	RATE	3					99V999	
11	INSURANCE.PREM	3					9V99	
12	RETIREMENT.PREM	3					9V99	
13	CURRENT	1	RECORD					
14	DEPARTMENT	2					99	
15	INDEX	2					99	

SAMPLE PAYROLL PROGRAM - MACHINE LISTING

```
SERIAL NAME   TEXT

01001 *PROCEDURE

01002        CALL (EMPLOYEE.NUMBER)  EMPLOYNO,
01003             (BONDEDUCTION)     BONDEDUCT,
01004             (BONDENOMINATION)  BONDENOM,
01005             (BONDACCUMULATION) BONDACCUM,
01006             (INSURANCE.PREM)   INSPREM,
01007             (RETIREMENT.PREM)  RETPREM,
01008             (DEPARTMENT.TOTAL) DPT.

01009 START.  OPEN ALL FILES.

01010 GET.MASTER.  GET MASTER,  AT END DO END.OF.MASTERS.

01011 GET.DETAIL.  GET DETAIL,  AT END GO TO END.OF.DETAILS.

01012 COMPARE.EMPLOYEE.NUMBERS.  GO TO COMPUTE.PAY WHEN DETAIL EMPLOYNO
01013        IS EQUAL TO MASTER EMPLOYNO,  LOW.DETAIL WHEN DETAIL
01014        EMPLOYNO IS LESS THAN MASTER EMPLOYNO.

01015 HIGH.DETAIL.  MOVE 'M' TO MASTER ERRORCODE,  FILE MASTER IN
01016        ERROR.FILE.

01017          GET MASTER,  AT END DO END.OF.MASTERS.

01018          GO TO COMPARE.EMPLOYEE.NUMBERS.

02001 LOW.DETAIL.  MOVE 'D' TO DETAIL ERRORCODE,  FILE DETAIL IN
02002        ERROR.FILE.

02003          GO TO GET.DETAIL.

02004 END.OF.MASTERS.  IF DETAIL EMPLOYNO = HIGH.VALUE THEN GO TO
02005        END.OF.RUN  OTHERWISE SET MASTER EMPLOYNO = HIGH.VALUE.

02006 END.OF.DETAILS.  IF MASTER EMPLOYNO = HIGH.VALUE THEN GO TO
02007        END.OF.RUN  OTHERWISE SET DETAIL EMPLOYNO = HIGH.VALUE,  GO
02008        TO COMPARE.EMPLOYEE.NUMBERS.

02009 END.OF.RUN.  MOVE CORRESPONDING GRAND.TOTAL TO PAYRECORD,  FILE
02010        PAYRECORD,  CLOSE ALL FILES.
02011          STOP 1234.

02012 COMPUTE.PAY.  IF DETAIL HOURS IS GREATER THAN 40 THEN SET DETAIL
02013        GROSS = (DETAIL HOURS - 40) * MASTER RATE * 1.5.

02014          SET DETAIL GROSS = DETAIL GROSS + MASTER RATE * 40,  DO
02015        FICA.ROUTINE,  DO WITHOLDING.TAX.ROUTINE.

02016          IF MASTER BONDEDUCT IS NOT EQUAL TO ZERO THEN DO
02017        BOND.ROUTINE.

02018          DO SEARCH FOR INDEX = 1(1)12.

02019 NET.     SET PAYRECORD NETPAY = DETAIL GROSS - DETAIL FICA - DETAIL
02020        WHT - DETAIL RETIREMENT - DETAIL INSURANCE - DETAIL
02021        BONDEDUCT.
```

```
SERIAL NAME   TEXT

03001           ADD CORRESPONDING DETAIL TOTALS TO MASTER TOTALS, MOVE
03002         CORRESPONDING DETAIL TO PAYRECORD, CHECK, MOVE PAYRECORD
03003         NETPAY TO CHECK AMOUNT.

03004           FILE CHECK.

03005           IF PAYRECORD DEPARTMENT IS GREATER THAN CURRENT DEPARTMENT
03006         THEN MOVE BLANKS TO PAYRECORD EMPLOYNO, PAYRECORD NAME,
03007         MOVE CORRESPONDING DEPARTMENT.TOTAL TO PAYRECORD, FILE
03008         PAYRECORD, MOVE ZEROS TO DPT HOURS, DPT GROSS, DPT WHT,
03009         DPT FICA, DPT BONDEDUCT, DPT INSPREM, DPT RETPREM, DPT
03010         NETPAY, DPT BONDPURCHASES.

03011           MOVE CORRESPONDING DETAIL TO PAYRECORD, CURRENT, ADD
03012         CORRESPONDING DETAIL TO DEPARTMENT.TOTAL, GRAND.TOTAL,
03013         FILE PAYRECORD.

03014           GO TO GET.MASTER.


03015 FICA.ROUTINE.  BEGIN SECTION.

03016           IF MASTER FICA + 0.03 * DETAIL GROSS IS LESS THAN 144.00
03017         THEN SET DETAIL FICA = 0.03 * DETAIL GROSS  OTHERWISE SET
03018         DETAIL FICA = 144.00 - MASTER FICA.

03019           ADD DETAIL FICA TO MASTER FICA.
03020         END FICA.ROUTINE.


04001 WITHOLDING.TAX.ROUTINE.  BEGIN SECTION.

04002           IF 13 * MASTER EXEMPTIONS IS LESS THAN DETAIL GROSS THEN
04003         SET DETAIL WHT = 0.18 * (DETAIL GROSS - 13 * MASTER
04004         EXEMPTIONS) OTHERWISE SET DETAIL WHT = ZEROS.

04005           ADD DETAIL WHT TO MASTER WHT.
04006         END WITHOLDING.TAX.ROUTINE.


04007 BOND.ROUTINE.  BEGIN SECTION.

04008           ADD MASTER BONDEDUCT TO MASTER BONDACCUM.


04009 BOND.CALCULATION.  IF MASTER BONDENOM IS NOT GREATER THAN MASTER
04010         BONDACCUM THEN SET MASTER BONDACCUM = MASTER BONDACCUM -
04011         MASTER BONDENOM OTHERWISE GO TO BOND.END.

04012           MOVE CORRESPONDING MASTER TO BONDORDER, ADD BONDORDER
04013         BONDENOM TO DPT BONDPURCHASES, MOVE BONDORDER BONDENOM TO
04014         PAYRECORD BONDENOM, FILE BONDORDER IN ERROR.FILE.

04015           GO TO BOND.CALCULATION.


04016 BOND.END.  END BOND.ROUTINE.


04017 SEARCH.  BEGIN SECTION.

04018           IF MASTER RATE GT TABLE.ITEM RATE (INDEX) THEN GO TO
04019         SEARCH.END OTHERWISE ADD TABLE.ITEM INSPREM (INDEX) TO
04020         DETAIL INSURANCE, ADD TABLE.ITEM RETPREM (INDEX) TO DETAIL
04021         RETIREMENT, GO TO NET.


04022 SEARCH.END.  END SEARCH.
```

```
SERIAL  NAME             LV TYPE QUAN MJ  DESCRIPTION

05001 *DATA

05002 MASTER             1RECORD        L

05003  ERRORCODE         2               A

05004 DATA               2

05005  EMPLOYEE.NUMBER   3
05006   DEPARTMENT       4              99
05007   EMPLOYEE         4              9999

05008  NAME              3              A(15)
05009  RATE              3              99V999

05010  DATE              3
05011   MONTH            4              99
05012   DAY              4              99
05013   YEAR             4              99

05014  EXEMPTIONS        3              99

05015  TOTALS            3
05016   GROSS            4              99999V99
05017   RETIREMENT       4              999V99
05018   INSURANCE        4              999V99

05019  FICA              3              999V99
05020  WHT               3              9999V99
05021 BONDEDUCTION       3              99V99

05022 BONDACCUMULATION   2              999V99
05023 BONDENOMINATION    2              999V99
05024                    2              AAA

06001 DETAIL             1RECORD        L

06002  ERRORCODE         2              A
06003  HOURS             2              99V9

06004 DATA               2

06005  EMPLOYEE.NUMBER   3
06006   DEPARTMENT       4              99
06007   EMPLOYEE         4              9999

06008  NAME              3              A(15)

06009  DATE              3
06010   MONTH            4              99
06011   DAY              4              99
06012   YEAR             4              99

06013  EXEMPTIONS        3              99

06014  TOTALS            3
06015   GROSS            4              99999V99
06016   RETIREMENT       4              999V99
06017   INSURANCE        4              999V99

06018  FICA              3              999V99
06019  WHT               3              9999V99
06020 BONDEDUCTION       3              99V99

06021 BONDACCUMULATION   2              999V99
06022 BONDENOMINATION    2              999V99
06023                    2              AAAAA
```

```
SERIAL  NAME          LV TYPE QUAN MJ  DESCRIPTION                  CONT

07001 BONDORDER        1RECORD       L

07002  EMPLOYEE.NUMBER 2              9(6)
07003   DATE           2              9(6)
07004  BONDENOMINATION 2              999V99
07005   NAME           2              A(15)


07007 CHECK            1RECORD       L

07008  EMPLOYEE.NUMBER 2
07009   DEPARTMENT     3              99
07010   EMPLOYEE       3              9999

07011  NAME            2              A(15)
07012   AMOUNT         2              $***9.99


08001 PAYRECORD        1RECORD       L

08002 EMPLOYEE.NUMBER  2
08003  DEPARTMENT      3              99
08004                  3              A
08005  EMPLOYEE        3              9999

08006                  2              A
08007 NAME             2              A(15)

08008 DATE             2
08009                  3              A
08010  MONTH           3              99
08011                  3              A
08012  DAY             3              99
08013                  3              A
08014  YEAR            3              99

08015 HOURS            2              8889.9-
08016 GROSS            2              $88889.99-
08017  WHT             2              $88889.99-
08018  FICA            2              $8889.99-
08019  BONDEDUCTION    2              $8889.99-
08020  INSURANCE       2              $8889.99-
08021  RETIREMENT      2              $8889.99-
08022 NETPAY           2              $88889.99-
08023  BONDENOMINATION 2              $88899.99-


09001 DEPARTMENT.TOTAL 1RECORD       L

09002   HOURS          2              9999V9
09003  GROSS           2              9(5)V99
09004   WHT            2              9(5)V99
09005   FICA           2              9999V99
09006   BONDEDUCTION   2              9999V99
09007   INSURANCE.PREM 2              9999V99
09008   RETIREMENT.    2                                             X
09009            PREM                 9999V99
09010  NETPAY          2              9(5)V99
09011   BONDPURCHASES  2              9(5)V99


09012 GRAND.TOTAL      1COPY          DEPARTMENT.TOTAL


10001 TABLE            1             L
10002                  2              '00999080060014991000 60'
10003                  2              '019991200900249915009 0'
10004                  2              '029991501200349920012 0'
10005                  2              '039992001500449925015 0'
10006                  2              '049993001800649930025 0'
10007                  2              '079993003509999930050 0'


10008                  1REDEF         TABLE

10009 TABLE.ITEM       2          12
10010  RATE            3              99V999
10011  INSURANCE.PREM  3              9V99
10012  RETIREMENT.PREM 3              9V99


10013 CURRENT          1RECORD

10014 DEPARTMENT       2              99
10015 INDEX            2              99
```

Appendix 2: Supplementary Information

Rules for Forming Conditional Expressions

Conditional expressions may contain the names of conditions, variables, constants and functions, as well as literals, arithmetic operators, relations of equality and relative magnitude and the operators NOT, AND and OR. Subexpressions may be contained in parentheses as required.

If a conditional expression (such as MARRIED or PAY IS GREATER THAN 2 * X + Y) is designated by the symbol Ci the following rules may be stated concerning the formation of conditional expressions involving Ci, NOT, AND and OR.

1.

The Conditional Expression	Is True If
C1	C1 is true
NOT C1	C1 is false
C1 AND C2	Both C1 and C2 are true
C1 OR C2	Either C1 is true, C2 is true, or both are true
NOT (C1 AND C2)	C1 is false, C2 is false, or both are false
NOT (C1 OR C2)	C1 and C2 are both false

2. If C1 and C2 are conditional expressions, then "C1 AND C2" and "C1 OR C2" are conditional expressions, as are similar expressions formed with the use of NOT. Thus, an expression of the form

   C1 AND (C2 OR NOT (C3 OR C4))

   may be successively reduced by substituting as follows:

   Let C5 equal "C3 OR C4" ⟶ C1 AND (C2 OR NOT C5)

   Let C6 equal "C2 OR NOT C5" ⟶ C1 AND C6

   Let C7 equal "C1 AND C6" ⟶ C7

   This rule indicates how conditional expressions may be formed from conditional expressions.

3. The conditional expression "C1 OR C2 AND C3" is identical with "C1 OR (C2 AND C3)" but is not the same as "(C1 OR C2) AND C3." In other words, conditional expressions are grouped first according to AND and subsequently by OR. However, the programmer's use of parentheses will affect the order of grouping.

4. The rules for formation of symbol pairs are contained in the following table:

		Second Symbol					
		C	OR	AND	NOT	(	)
First Symbol	C	0	1	1	0	0	1
	OR	1	0	0	1	1	0
	AND	1	0	0	1	1	0
	NOT	1	0	0	0	1	0
	(	1	0	0	1	1	0
	)	0	1	1	0	0	1

where the "1" indicates that the pair is permissible, and the "0" indicates a symbol pair that is not permissible. Thus, the pair "OR NOT" is permissible, while "NOT OR" is not permissible.

Rules for Forming Arithmetic Expressions

Arithmetic expressions may contain the names of variables, constants and functions, also literals and conditional expressions, joined by arithmetic operators. Subexpressions may be contained in parentheses as required. The rules for forming arithmetic expressions are as follows:

1. The basic operators are:

Binary Operator	Written as
Addition	+
Subtraction	−
Multiplication	*
Division	/
Exponentiation	**
Unary Operator	**Written as**
Negation	−
Absolute Value	ABS
Truth Value	TR

2. The ways in which symbol pairs may be formed are summarized in the following table:

		Second Symbol					
		Variable	+ − * / **	ABS, TR	Negation −	(	)
First Symbol	Variable	0	1	0	0	0	1
	+ − * / **	1	0	1	0	1	0
	ABS, TR	1	0	0	0	1	0
	Negation −	1	0	0	0	1	0
	(	1	0	1	1	1	0
	)	0	1	0	0	0	1

where "1" indicates a permissible symbol pair, and "0" indicates a pair which is not permitted. Thus, "* (" is permissible, while "(*" is not.

3. When the hierarchy of operations in an expression is not completely specified by parentheses, the order of operations (working from inside to outside) is assumed to be exponentiation, then multiplication and division, and finally addition and subtraction. Thus the expression A + B/C + D**E*F − G will be taken to mean $A + (B/C) + (D^E \cdot F) - G$.

4. When the sequence of consecutive operations of the same hierarchal level (i.e., consecutive multiplications and divisions or consecutive additions and subtractions) is not completely specified by parentheses, the order of operations is assumed to be from left to right. Thus expressions ordinarily considered ambiguous, e.g., $A/B \cdot C$ and A/B/C, are permitted in Commercial Translator statements. For instance, the expression A*B/C*D is taken to mean ((A*B)/C)*D.

5. The expression A^{B^C}, which is sometimes considered meaningful, cannot be written as A**B**C; it should be written as (A**B)**C or A**(B**C), whichever is intended.

List of Commercial Translator Commands

The general forms of all of the Commercial Translator commands are presented below in alphabetic order for reference purposes. In order to make the list as concise as possible, optional words and phrases are shown enclosed in brackets, — [ ]; the brace, {, is used to indicate a choice of one of two or more variant forms.

ADD [CORRESPONDING] *data.name.1* TO *data.name.2, data.name.3,* ... *data.name.n*

BEGIN SECTION [USING *parameter.1, parameter.2,* ... *parameter.n*] [GIVING *function.1, function.2,* ... *function.n*]

CALL (*old.name.1*) *new.name.1,* (*old.name.2*) *new.name.2,* ... (*old.name.n*) *new.name.n*

CLOSE { *file.name.1, file.name.2,* ... *file.name.n*
ALL FILES

DISPLAY { *'any message'*
data.name
any combination of the above

DO *procedure.name* [EXACTLY *n* TIMES] [USING *data.name.1, data.name.2,* .. *data.name.n*] [GIVING *result.name.1, result.name.2,* ... *result.name.n*]

DO *procedure.name* FOR *index.name.1 = p.1(q.1)r.1* [, *index.name.2 = p.2(q.2)r.2, index.name.3 = p.3(q.3)r.3*] [USING *data.name.1, data.name.2,* ... *data.name.n*] [GIVING *result.name.1, result.name.2,* ... *result.name.n*]

END *procedure.name*

ENTER *coding.language*

FILE *record.name* [IN *file.name*]

GET { RECORD FROM *file.name* / *record.name* } AT END *any imperative clause*

GO TO *procedure.name*

GO TO *procedure.name.1* WHEN *conditional expression 1,*
procedure.name.2 WHEN *conditional expression 2, ...*
procedure.name.n WHEN *conditional expression n*

GO TO (*procedure.1, procedure.2, ... procedure.n*) ON *index.name*

INCLUDE [HERE] *library.procedure* [AS *procedure.name*] [WITH *new.name.1* FOR
old.name.1, new.name.2 FOR *old.name.2, ... new.name.n* FOR *old.name.n*]

LOAD *procedure.name*

MOVE [CORRESPONDING] *data.name.1* TO *data.name.2, data.name.3,*
... data.name.n

NOTE *any sentence.*

OPEN { *file.name.1, file.name.2, ... file.name.n* / ALL FILES }

OVERLAP *procedure.name.1, procedure.name.2, ... procedure.name.n*

SET *variable.1, variable.2, ... variable.n = arithmetic expression*
[TRUNCATED] [, ON OVERFLOW *any imperative clause*]

SET *condition.name*

STOP *n*

List of Commercial Translator Words

All those words which are a fixed part of the Commercial Translator vocabulary are listed below. The words are presented here to assist the programmer in avoiding the use of any of them when choosing data-names and procedure-names.

ABS
ADD
ALL
AND
AS
AT

BEGIN
BLANK
BLANKS

CALL
CLOSE
COMMERCIAL
†COND
†COPY
CORRESPONDING

DISPLAY
DO

END
ENTER
EQUAL
EXACTLY

FILE
FILES
FOR
FROM
†FUNCT

GET
GIVING
GO
GREATER
GT

HERE
HIGH.VALUE
HIGH.VALUES

IF
IN
INCLUDE
IS

†LABEL
LESS
†LIBRARY
LOAD
LOW.VALUE
LOW.VALUES
LT

MOVE

NOT
NOTE

ON
OPEN
OR

OTHERWISE
OVERFLOW
OVERLAP

†PARAM

†QUANTITY

RECORD
†REDEF

SECTION
SET
STOP

THAN
THEN
TIMES
TO
TR
TRANSLATOR
TRUNCATED

USING

WHEN
WITH

ZERO
ZEROS

---

†These words have a restricted usage only in data description; they may be used freely in procedure description.

Appendix 3: Glossary

Following is a list of important terms which are used frequently in this manual, including a number which have certain specialized meanings when used with respect to the Commercial Translator system. The definitions given in this glossary are not exhaustive, and the reader is referred to the text of the manual for amplification when required.

ABSOLUTE VALUE
: The value, or magnitude, of a number, regardless of sign. When used arithmetically, the absolute value is treated as having a plus sign.

ADDRESS
: In data processing, a specific location within storage, or a specific item or feature of data processing equipment. In the Commercial Translator language, addresses are represented by names. *Actual address, absolute address,* or *machine address*—an address defined in terms of the operating specifications of a data processing system; such addresses are not used by the programmer in writing Commercial Translator programs.

ALPHABETIC
: With respect to data, consisting of one or more non-numeric characters, including the blank. As used in the Commercial Translator system, the term is not limited to the letters of the alphabet.

ALPHAMERIC
: In the Commercial Translator system, consisting of any of the characters of a machine's character set, except that an alphameric literal may not contain a quotation mark. (See the rules governing literals in Chapter 2.)

ARITHMETIC EXPRESSION
: An expression containing any combination of data-names, literals, constants, and truth functions, joined together by one or more arithmetic operators in such a way that the expression as a whole can be reduced to a single numeric value. (See the discussion of arithmetic expressions in Chapter 2 and the commands SET and ADD in Chapter 3.)

CHARACTER
: One of a set of elementary symbols which may be arranged in ordered groups to express information. These symbols may include the decimal digits 0 through 9, the letters A through Z, punctuation symbols, special input and output symbols, and any other symbols which may be accepted by a data processing system.

CLAUSE
: In the Commercial Translator language, a group of words (including symbols where appropriate) which either expresses a complete command or defines a condition to be tested. (See also IMPERATIVE CLAUSE and CONDITIONAL CLAUSE.)

CLOSED SUBROUTINE
: In the Commercial Translator system, a section of procedure which is executed by means of a DO command. Also called a linked subroutine. (See also OPEN SUBROUTINE.)

COMMAND
: In the Commercial Translator language, a verb and its associated operand(s).

COMPOUND NAME
: In the Commercial Translator language, a name consisting of two or more simple names combined for the purpose of making the right-most one unique, in accordance with the rules given in Chapter 2 of this manual.

CONDITION
: In the Commercial Translator language, the presence of a specified value in a variable field. (See CONDITIONAL EXPRESSION.)

CONDITIONAL CLAUSE
: A clause containing a conditional expression introduced by the word IF and terminated by the word THEN.

CONDITIONAL EXPRESSION — In the Commercial Translator language, an expression which has the particular characteristic that, taken as a whole, it may be either true or false, in accordance with the rules given in Chapter 2 of this manual.

CONDITION-NAME — A name assigned by the programmer to a value, representing one of several conditions, which may be present in a data field, in accordance with the rules given in Chapter 2 of this manual.

CONSTANT — A value which is to be used in a program without alteration, in accordance with the rules given in Chapter 2 of this manual. While a literal may be thought of as a special form of constant, most constants have names which are specified by data description entries, as explained in Chapter 4.

DATA DESCRIPTION — That portion of a Commercial Translator program which consists of entries defining the nature and characteristics of the data to be used in the program. The data description is one of the three main divisions of a Commercial Translator program, the others being the procedure description and the environment description. (See Chapter 4.)

DATA-NAME — A name assigned by the programmer to an item of data for use in a Commercial Translator program, in accordance with the rules given in Chapter 2 of this manual.

DIVISION HEADER — One of three special words used to identify the beginning of a main portion of a Commercial Translator source program. The three words are *DATA, *ENVIRONMENT, and *PROCEDURE, identifying, respectively, portions of the data description, environment description, and procedure description divisions.

ENVIRONMENT DESCRIPTION — That portion of a Commercial Translator program in which the programmer specifies the equipment and equipment features to be used in a program, such as the nature of the input and output equipment, the size and nature of the storage area available, and so on. This subject is discussed in the manuals covering the processors for the various data processing systems.

FIELD — In the Commercial Translator system, an item of data which can be operated upon by the arithmetic and/or data transmission verbs. It is usually thought of as a small unit or element of data, as opposed to a *record,* which is usually composed of a number of fields.

FIELD PICTORIAL — The representation of the nature and length of an item of data by means of the Commercial Translator format characters, in accordance with the rules given in Chapter 4 of this manual.

FIGURATIVE CONSTANT — One of several constants which have been "pre-named" and "pre-defined" in a Commercial Translator processor so that they can be written in the procedure statements without data description entries. The figurative constants are BLANK(S), ZERO(S), HIGH.VALUE(S), and LOW.VALUE(S).

FILE (noun) — In the Commercial Translator system, a body of data, stored in some external medium, which can be made accessible to the system by the verb OPEN.

FIXED WORD — One of a selected list of words having special meanings in the Commercial Translator system and not available to the programmer except in accordance with the rules specified in this manual. A list of fixed words appears in Appendix 2 of this manual.

FLOATING POINT — A form of representing a number X by a pair of numbers Y and Z in the form $X = Y \times B^Z$, where B is the number base used. In the decimal system, $B = 10$; in the binary system, $B = 2$. The decimal system is used in the Commercial Translator source language. To illustrate: the number 127.6 would normally be represented as $.1276 \times 10^3$ (or, in Commercial Translator notation, .1276F3). The quantity Y is called the fraction or mantissa, and in the best notation lies between 0 and 1. The quantity Z is called the exponent or

power. This form is used mainly in scientific calculation where the size of computed quantities is difficult to predict and allow for.

FORMAT CHARACTER — One of the characters specified in the table of format characters in Chapter 4 of this manual for indicating the characteristics of data in the data description portion of a Commercial Translator program.

FUNCTION — In the Commercial Translator system, a result obtained as a consequence of a procedure; specifically, a result named in the GIVING clause of a BEGIN SECTION command. (See the discussion of functions in Chapter 2 of this manual, and the BEGIN SECTION command in Chapter 3.)

FUNCTION-NAME — A name assigned to a function by the programmer. (See FUNCTION.)

HEADER — (See DIVISION HEADER.)

IMBEDDED PERIOD — A period contained within a Commercial Translator name in such a way that it is not adjacent to a blank.

IMPERATIVE CLAUSE — A clause expressing a complete command; it consists of a verb and its operand(s).

INSTRUCTION — A code, symbol, or group of symbols, which causes a data processing system to perform an operation of some kind. An instruction usually consists of a code specifying a kind of operation, and an address which indicates the data and/or item of equipment on which the operation is to be performed.

INTEGER — A whole number. E.g., 54 is an integer, while 54.6 is not.

JUSTIFICATION — 1. In printing or listing, the alignment of a margin. 2. In the internal storage of data within a processing system, the alignment of data with respect to the left or right boundaries of machine words, as explained in Chapter 4 of this manual.

LEVEL — In the Commercial Translator system, the status of one item of data relative to another, showing whether one is to be treated as a part of the other or whether they are unrelated, as specified in the rules governing level numbers in Chapter 4 of this manual.

LITERAL — A value expressed literally in a procedure statement, as opposed to a value represented by a name. Thus, 2 is a literal, whereas the name TWO could be one of a number of possible names used to represent the value 2. (See the rules governing literals in Chapter 2 of this manual.)

LOOP — A coding technique whereby a group of instructions is repeated, usually with modification of at least one of the instructions in the group and/or with modification of the data being operated upon.

MACHINE LANGUAGE — The system of codes by which instructions and data are represented internally within a particular data processing system.

MACHINE WORD — (See WORD.)

MACHINE-INDEPENDENT — An adjective used to indicate that an instruction or a program is conceived, organized, or oriented without specific reference to the technical characteristics of any one data processing system. Use of this adjective usually implies that the instruction or program is oriented or organized in terms of the logical nature of a problem, rather than in terms of the technical means of solving it.

MEMORY	Main storage. (See STORAGE.)
MODE	A system of data representation used within the storage section of a data processing system.
NUMERIC	1. With respect to data, having a numeric value. 2. With respect to literals, consisting wholly of numerals and their included decimal points, plus and minus signs, and floating point symbols, if any.
OBJECT PROGRAM	A program in machine language resulting from the translation of a source program by a processor.
OBJECT TIME	The time at which an object program is executed.
OPEN SUBROUTINE	In the Commercial Translator system, a section of procedure which is executed by a means other than the DO command. Also called an "in-line" subroutine. (See also CLOSED SUBROUTINE.)
OPERAND	In the Commercial Translator language, the "object" of a verb—i.e., the data or equipment governed, or operated on, by a verb.
OPERATOR	In the Commercial Translator language, a word or symbol, other than a verb, which directs the data processing system to take some action. E.g., the arithmetic operator + instructs the system to perform an addition, and the conditional operator IF directs it to test a conditional expression.
PARAMETER	In the Commercial Translator system, a value used in a procedure to obtain a result; specifically, an item of data named in the USING clause of a BEGIN SECTION command. (See the discussion of functions in Chapter 2 of this manual, and the BEGIN SECTION command in Chapter 3.)
PARAMETER-NAME	A name assigned to a parameter by the programmer. (See PARAMETER.)
PROCEDURE	In the Commercial Translator system, a sequence of one or more instructions used to perform some operation. A routine or subroutine.
PROCEDURE DESCRIPTION	That portion of a Commercial Translator program which consists of instructions directing the data processing system to take specified actions. The procedure description is one of the three main divisions of a Commercial Translator program, the others being the data description and the environment description.
PROCEDURE-NAME	A name assigned by the programmer to a procedure for use in a Commercial Translator program, in accordance with the rules given in Chapter 2 of this manual.
PROCESS TIME	The time at which a source program is translated into an object program.
PROCESSOR	A specialized program used to translate a source program into an object program.
PROCESSOR VERBS	Verbs which give instructions to the processor to be acted upon at the time the source program is being translated into the object program. Such verbs do not act at object time.
PROGRAM	A complete sequence of instructions directing a data processing system to perform an operation. The term implies an extended sequence incorporating all of the detailed steps and subroutines required to complete a job. (See also OBJECT PROGRAM and SOURCE PROGRAM.)

PROGRAM VERBS — Verbs which cause the processor to generate machine instructions which will be part of the object program.

QUALIFICATION — With reference to Commercial Translator names, the technique of modifying a name by the addition of another name in order to make it unique, in accordance with the rules given in Chapter 2 of this manual. A qualified name is generally known as a *compound name.*

RECORD — In the Commercial Translator system, a sequence of data which can be made accessible to the system by the verb GET. A record usually consists of a number of fields.

RELATIONAL EXPRESSION — In the Commercial Translator language, an expression that describes a relationship between two terms. E.g., A IS GREATER THAN B, or X IS EQUAL TO 10.

ROUND — To shorten a number by applying some rule to adjust the least significant remaining digit. In the Commercial Translator system, this digit is increased by 1 when the part removed is greater than or equal to one-half. Thus, in the decimal system the number 126.5027 would be rounded to 127, because the last number removed is 5 or greater. Rounding need not occur just at decimal points. For a 6-position field the number would be rounded to 126.503; for a 2-position field it would be rounded to 13, which would be understood to be 130.

ROUTINE — In the Commercial Translator system, a sequence of one or more instructions used to perform some operation. A procedure or subroutine.

SECTION — In the Commercial Translator language, one or more consecutive sentences defined in accordance with the instructions governing the BEGIN SECTION and END commands, as explained in Chapter 3 of this manual.

SENTENCE — In the Commercial Translator language, a complete statement specifying one or more operations, in accordance with the rules given in Chapter 2 of this manual. It is always terminated by a period.

SOURCE LANGUAGE — As used in this manual, the Commercial Translator language.

SOURCE PROGRAM — As used in this manual, a program written in the Commercial Translator language.

STATEMENT — As used in this manual, a clause or a sentence.

STORAGE — A medium in which data may be retained. Storage may be internal or external. *Main storage*—the principal internal area in which data is retained for active use within a data processing system. *Auxiliary storage*—a supplementary storage medium, less active in use than main storage, in which data may be retained; data in auxiliary storage can be addressed directly by the system, but access is generally slower than to main storage.

STORED PROGRAM — A data processing program which is stored internally, within a data processing system. The program itself occupies storage in the same manner as the data used in the program and may be treated as if it were data.

SUBROUTINE — In the Commercial Translator system, a sequence of one or more instructions used to perform some operation. A procedure or routine.

SUBSCRIPT — An integer used to identify a particular item in a list or table, in accordance with the rules specified in Chapter 2 of this manual. It may be written in a Commercial Translator program as a numeric literal, a data-name, or a limited form of arithmetic expression.

TRUNCATED — Shortened by dropping the less significant digits of a number, as opposed to *rounded*. (See ROUND.) E.g., the number 2063.78 becomes 2063.7 when truncated, 2063.8 when rounded.

TRUTH FUNCTION — An expression consisting of the truth operator TR, followed by a conditional expression, in accordance with the rules given in Chapter 2 of this manual. A truth function acquires a value of 1 if the conditional expression is true, a value of 0 if the expression is false.

UNARY OPERATOR — An operator which refers to a single data-name or parenthetical expression; e.g., ABS is a unary operator.

VARIABLE — In the Commercial Translator system, a field in storage which may contain different values at different times during the running of the object program.

VERB — In the Commercial Translator language, one of a selected list of words that cause a data processing system to take an action. (See Chapter 3 of this manual.)

WORD — In the Commercial Translator language, a basic unit of the language, serving the same general purposes as words in other languages. *Machine word*—a subdivision of storage having a fixed size.

Index

ABS 45, 106
Absolute Values 45, 106
ADD 47
ADD CORRESPONDING 47
Alignment (See Justification)
Alphabetic Literals 19, 45
Alphameric Literals 19
AND 23-24, 105-106
Arithmetic Expressions 20-21, 44-46, 106-107
Arithmetic Operators 21, 45, 106
Assigned GO TO 48-49
BEGIN SECTION 26, 32-34, 56-58
(See also DO)
Binary Operators 45, 106
Blanks 27
CALL 59
Character Set 12
Clauses 24-25
(See also Conditional Clauses, Imperative Clauses)
CLOSE 41
Commands 35, 108-109
(See also Verbs, Program Verbs, Processor Verbs)
Compound Conditions 23-24, 105-106
Compound Names 15, 59, 68, 71
COND 71-72
Conditional Clauses 25
Conditional Expressions 21-24, 48, 105-106
Conditional GO TO 48
Condition-Names 13, 14, 16, 22, 46, 71-72, 105
Conditions 22, 71-72
Conditions, Compound 23-24, 105-106
Constants 17-20, 45, 81
Control Commands 48-54
COPY 76-77, 81
Data Description 26-27, 61-85
Data Description, Purpose of 61-62
Data Organization 68-71
Data Processing System 1, 2
Data Substitution 52
Data Transmission Commands 41-43
Data-Names 13, 16, 67
Decimal Point, Assumed 80
Decimal Point, True 80
DISPLAY 54-55
Division Headers 27, 37, 65
Divisions 26-27
DO 32-34, 49-54
DO, with Data Substitution 52-53
(See also Functions, Parameters)
DO, with Indexing 50-51
DO, with Named END 53
Editing 42, 43, 61
END 53, 56-58
ENTER 59
Environment Description . . . . . Preface, 26-27
Expressions 20-24
(See also Arithmetic Expressions, Conditional Expressions)
Field Pictorial 42, 43, 79-81
Fields 64
Figurative Constants 17, 19-20
FILE 40-41
Files 38, 39, 40, 41, 63-64
Floating Point Numbers 18, 28, 80
Form, Columnar, for Data Description 66
Form, Columnar, for Procedure Description 36
Format, Data Description 65
Format, Procedure Statements 37
Format Characters 79-81
FUNCT 34, 73
Function-Names 28, 32-34, 46, 73
Functions 28, 32-34, 46, 58
GET 39
GIVING (See BEGIN SECTION, DO, and Functions)
GO TO 48-49
Headers, Division 27, 37, 65
Imbedded Period 27
Imperative Clauses 25
(See also Commands)
INCLUDE 58
Indentation 37, 67, 68
Index 49, 50-52
Indexing 50-52
Input/Output Commands 38-41
Justification 78-79
LABEL 77
Labels 63, 77
Languages, Lower-Level (See ENTER)
Level 68-71
Level Numbers (See Level)
Library 58, 81
Lists 28-31
Literals 17, 18-19, 45
Literals, Formation of 18
LOAD 54
Loops 50-51
Machine Words 78-79
Memory Partitioning (See LOAD and OVERLAP)
Mode 78
MOVE 42-43
MOVE CORRESPONDING 43
Name Substitution 58
Named Constants 17-18, 19, 81
Named Constants, Formation of 19
Names 13-16
(See also Compound Names, Condition-Names, Constants, Data-Names, Function-Names, Named Constants, Parameter-Names, Procedure-Names)
Names, Formation of 15

Names, Placing in Program . . . 16, 67
NOT . . . 23-24, 105-106
NOTE . . . 59
Nouns (See Names)
Numeric Literals . . . 18
OPEN . . . 39
Operands . . . 17
Operators . . . 13, 21, 45, 106
Operators, Arithmetic . . . 21, 45, 106
OR . . . 23-24, 105-106
OVERFLOW . . . 44, 47
OVERLAP . . . 55-56
Packing . . . 78-79
PARAM . . . 34, 73
Parameter-Names . . . 32-34, 73
Parameters . . . 28, 32-34, 58, 73
Partitioning of Memory (See LOAD and OVERLAP)
Pictorial, Field . . . 42, 43, 79-81
Procedure Description . . . 26-27, 35-60
Procedure-Names . . . 13, 14, 16, 37
Processor Commands . . . 55-59, 60
Processor Verbs . . . 12, 35
(See also Processor Commands)
Program Commands . . . 38-55, 60
Program Verbs . . . 13, 35
(See also Program Commands)
Punctuation . . . 27-28
Quantity . . . 75, 77-78, 82-83
QUANTITY IN . . . 82-83
RECORD . . . 71
Records . . . 39, 40-41, 64, 71
REDEF . . . 67, 73-76, 81
Relational Expressions . . . 21
Relations . . . 21
Renaming (See CALL)
Rounding . . . 44
Scale Factor . . . 80
Sections . . . 26, 49, 56-58
Sentences . . . 25-26
SET . . . 44-46
SET, with Condition-Names . . . 46
Spacing . . . 27-28
STOP . . . 54
Storage Areas . . . 84-85
Subscripts . . . 28, 30-31, 50-51, 75-76
Synonyms (See CALL)
Tables . . . 28-31, 74-76
Transfer Operations (See GO TO, DO)
TRUNCATED . . . 44, 47
Truth Functions . . . 20, 24, 45-46
Truth Operator . . . 20, 21, 24, 45-46, 106
Type Codes . . . 71-78
Unary Operators . . . 45, 106
Unconditional GO TO . . . 48
USING (See BEGIN SECTION, DO, and Parameters)
Variables . . . 45
Verbs . . . 12, 35, 108

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York

Printed in U.S.A. F28-8043