ICL

ICL Training

PROGRAMMING

THE 1900 SERIES

IN COBOL

A. S. RYE - FACE

PW5

ICL Education and Training

Course training manual

Programming the 1900 Series in COBOL

Training Publications PN5, issue 4
Issued by ICL Training
ICL Beaumont, Old Windsor, Berks.
TP No. 4177

Preface

Programming the 1900 Series in COBOL is a training manual in COBOL programming on the ICL 1900 Series computers. It is intended for use on ICL COBOL training courses and is therefore not a reference manual and is not intended to be used long after the termination of the course.

The manual should meet the needs of all trainee programmers in full and Compact COBOL. It does not assume any previous knowledge of the 1900 Series or of programming. All the major full COBOL facilities are included. Direct access programming is also dealt with, but the manual provides only a starting-point and is not comprehensive in this field.

The structure of the manual is as follows, *Chapters 1 to 3* provide a basic introduction to the 1900 Series and to flowcharting. *Chapter 4* introduces the COBOL language and its main features. *Chapters 5 to 7* deal with the four divisions of a COBOL program, and the main body of programming. *Chapter 8* describes steering lines and the compilation process, *Chapter 9* makes some points about efficient program writing and testing and *Chapter 10* introduces subroutines. *Chapter 11* is a description of direct access. *Chapter 12* introduces SORTing, *Chapter 13* describes segmentation and lastly, *Chapter 14* deals with some less common features of COBOL.

It is stressed that this manual is essentially a training document and is not a complete description of COBOL facilities. For authorative definitions, reference should be made to the reference manual *COBOL*.

The following acknowledgement is reprinted from the COBOL 1965 Report issued by the U.S. Department of Defence.

Any organization interested in reproducing the COBOL report and specification in whole or in part, using ideas taken from this report as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce this section as part of the introduction to the document. Those using a short passage, as in a book review, are requested to mention COBOL in acknowledgement of the source, but need not quote this entire section.

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the COBOL Committee as to the accuracy and functioning of the

programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

Procedures have been established for the maintenance of COBOL. Enquiries concerning the procedures for proposing changes should be directed to the Executive Committee of the Conference on Data Systems Languages.

The authors and copyright holders of the copyrighted material used herein

> FLOW-MATIC (Trademark for Sperry Rand Corporation), Programming for the Univac (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Contents

Illustrations

Chapter 1 Basic 1900 concepts

INTRODUCTION

A computer may be regarded as consisting of a *central processor* and a number of *peripherals*. Within the central processor are performed the tasks of manipulating data (*data processing*) for which the whole machine is designed; these tasks are controlled by a precise and logical set of instructions (or *program*) which has been prepared by a *programmer*. The function of the peripherals is to transfer the data to and from the central processor.

The most important part of the 1900 central processor is the *store*, in which the data and program instructions are represented in binary form. The store is divided into units known as *store locations* or *words* and the size or capacity of the store is expressed in terms of the number of locations it contains; this is calculated in multiples of 1,024, for example, 8 × 1,024 = 8,192 locations. However, it is more usual to abbreviate the capacity term by the use of K, for example, 8K = 8,192.

Before data can be processed it must be converted into a form acceptable to the central processor (this also applies to the program instructions which operate on the data). Similarly when the results of the processing are extracted from the machine these must be converted back into a form which human beings can read. The handling of these processes (known respectively as *input* and *output*) is performed by machines called *peripherals*.

The main input peripherals on the 1900 are the *card reader* and the *paper tape reader*, which transfer to the central processor data and program instructions punched in coded form in cards or paper tape. The most important output peripheral is the *line printer*.

In addition to the above machines, often referred to as *slow peripherals*, there are also magnetic devices to and from which transfers can be made at greater speed. These devices are most often used as a form of *backing store* to contain data, such as records, which may be required on more than one computer run. They include *magnetic tape*, *cassette tape*, *magnetic discs* and the *magnetic drum*; the two latter are known as *direct access* devices. All these machines are called *fast peripherals*.

The tangible machinery of the computer, that is, the central processor and the peripherals, is known as *hardware*. *Software* refers to any intangible system for controlling the operations of this machinery; thus, strictly speaking, any program can be called 'software', although in practice the name is usually reserved for programs or parts of programs prepared and supplied by the manufacturers of the computer.

A special case of such software is *Executive*, the program which must be permanently in the store of all 1900 central processors while they are switched on, and which controls certain of their activities. Executive (which will be more fully described in Chapter 2 of this manual) is unusual in that certain facilities are available to it; from the point of view of the programmer, in fact, its operation can be regarded as equivalent to the operations of hardware.

THE BINARY SYSTEM

Within a 1900 Series computer both the data to be processed and the program instructions which perform the processing take the form of numbers which are represented electronically. The methods used, however, mean that such numbers must be represented in *binary form*.

Whereas the notation we use for arithmetic in everyday life (the decimal system of notation) involves the use of ten different symbols (the figures 0 to 9) the binary system requires only two. In writing these are represented by the symbols 0 and 1 and within a computer they may be represented, for instance, by a circuit which can only be in one of two distinct states; one state would then be allocated to the same function as the symbol 0 and the other state to the symbol 1.

The difference between the decimal system and the binary system can be expressed in mathematical terms by saying that the decimal system has a 'radix' of *ten*, whereas the binary system has a 'radix' of *two*.

A decimal number, for example 689, consists of a combination of digits: reading from right to left each digit indicates a successively higher power of ten. Hence 689 could also be regarded as:

six hundreds + eight tens + nine ones

or in mathematical notion as

$(6 \times 10^2) + (8 \times 10^1) + (9 \times 10^0)$

In a binary number, however, although the digits are arranged in the same order, each binary digit indicates not a power of *ten* but a power of *two*. In binary notation the symbol 1 indicates that a particular power of two is present and the symbol 0 indicates that it is absent. Hence the binary number

1101

could be regarded as

one eight + one four + no twos + one

and could be written in mathematical notation as

$(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$

or more simply

$2^3 + 2^2 + 2^0$

In decimal notation this could be written

$8 + 4 + 1 (= 13)$.

Each 0 or 1 is known as a binary digit or *bit*. A group of a fixed number of such bits makes up one computer *word*. In storage, each 1900 word is contained in a location accommodating 24 bits and identified by an *address*, one of a consecutive series of numbers, which may be expressed in decimal or in binary. The contents of a word may be *moved* (or *copied*) to another word by means of appropriate program instructions.

Although, when written, a binary number contains more (often many more) digits than a decimal number there are two main advantages in employing the binary notation inside the computer: it is more economical of storage than any 'coded' representation of decimal numbers and simpler circuitry is required to carry out the arithmetic operations described below.

Binary arithmetic

Although there is no need for the programmer to have detailed knowledge of the physical means by which arithmetic is performed inside the computer, an appreciation of the basis of such operations can be gained by considering more examples demonstrating binary arithmetic, using the notation already defined.

Addition of binary numbers can be performed by the same method as addition of decimal numbers, by adding together the individual digits, starting from the right, and effecting a 'carry' when necessary. In the case of binary numbers, however, since each column of bits indicates only a power of two, this 'carry' will occur whenever two 1's are added together. Hence the rules for binary addition can be summarized as follows:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 and carry 1
1 + 1 + carry = 1 and carry 1

An example of such a binary addition can be most clearly demonstrated by showing first the 'partial sum' and the 'carries' separately, and then the result of adding in the carries:

1101	13
+1001	+ 9
0100	12
1 1	1
10110	22

Although it would be physically possible to perform *direct subtraction* within the computer, the design of the machine is greatly simplified if a method known as *complementary subtraction* is used. Under this method one number can be subtracted from another by adding to the first number the *complement* of the second.

The complement of any particular number may be defined as the number which needs to be added to it in order to produce a zero answer (discounting the extra digit, which such an addition will produce). Hence, in decimal notation, the complement of 33 is 67, since the two numbers together add up to 100 and the new digit produced is discounted. If therefore we wish to subtract 33 from, for example, 80, one way of doing this would be to add to 80 the complement of 33, that is, 67. This would result in 147, or, if the extra digit is discounted 47.

The complement of a binary number can be formed by changing every 0 in the number to 1 and every 1 to 0, and then adding 1. (In effect, this amounts to copying down the number, starting from the right-hand end and, after the first 1, reversing 0's and 1's.)

In order to subtract 1010 (decimal 10) from 1101 (decimal 13) the complement of 1010 is first formed and then this is added to 1101.

1101
+0110 (complement of 1010)
10011

If we disregard the extra digit at the left-hand end of the number and the two non-significant zeros the answer is binary 11 (decimal 3).

Octal

Although the *programmer* (as distinct from the machine for which he writes his program) does not have to work in binary he occasionally needs to write down a number in binary notation. Since a large number in binary contains a great many bits this is usually done more concisely, using a shorthand method known as *octal*.

In octal representation (which implies counting in eights) the bits of a binary number are considered in groups of three bits. Hence the highest number which can be represented in one group is binary 111, or 7 in decimal and octal notation, and a 24-bit computer word in which every bit was a 1 would be represented in octal as 77777777 (or eight groups of 7).

Decimal symbols for numbers below 7 can also be used to represent other binary patterns which appear in the table which follows. This table assumes, for the sake of clarity, a computer word of 12 bits; the three columns show:

1 The binary number as a simple string of bits.

2 The binary number with the bits grouped in threes.

3 The octal equivalent.

The table also demonstrates that octal numbers are preceded by a hash mark (#), and that leading (that is, non-significant) zeros are usually omitted, as in the second and fourth examples.

Binary	*Binary (grouped)*				*Octal*
111111111000	111	111	111	000	#7770
000000111000	000	000	111	000	#70
111111001011	111	111	001	011	#7713
000000110100	000	000	110	100	#64
100100110101	100	100	110	101	#4465

THE 1900 WORD AND THE PROGRAM AREA

The unit in which a 1900 computer holds data and program instructions in binary form is the *word* of 24 bits. Each word or location (on the 1900 Series the two terms mean the same thing) in store can be referred to (*addressed*) in one of three different ways:

1 By an *absolute number* or *absolute address* which is used by Executive and by hardware but never by the programmer. This number indicates the position of the location in the store as a whole (that is, including the area occupied by Executive).

2 By a number which is assigned to it for the purposes of a particular program. Under this system the first location used in a particular program is assigned the *relative address* 0 and all succeeding locations are numbered consecutively from this starting point.

The absolute address of the starting point is known as the *Datum* of that program. During the running of the program Executive establishes the absolute address of each location used in it by adding its relative address to the Datum; the absolute address can then be used by hardware for its operations.

3 By a name composed by the programmer himself under the rules of the COBOL language. In COBOL these names refer to fields within a record, and the field boundaries do not necessarily coincide with word boundaries.

Note: All the above names and numbers are ways of *referring to* a location; they should be carefully distinguished from the *contents* of that location, that is, a binary pattern which may be an item of data or a coded program instruction.

Format of the word

The 24 bit-positions in each word are also numbered from B0 to B23. In a graphical representation B0 is the bit-position at the extreme left; it will be the most significant bit of the number. B23, the least significant bit of the number will be at the extreme right, as for example

B0, B1, B2 . . . . . B21, B22, B23

(Similarly, when two or more words are represented on paper the word with the lowest address in the program is given the left-hand position.)

Other aspects of the format of each word depend upon the category into which the word falls; the two main categories are words which contain data (*data words*) and *program instruction words*.

Data words

BINARY INTEGERS

When the data contained in a word is held in the form of a binary integer that is, a whole number, (as distinct from a fraction), the least significant bit of the integer occupies B23 of that word.

When the integer is positive, B0 of the word is always zero. (Hence the highest positive number which can be accommodated in one word is $2^{23}-1$, or in decimal, 8,388,607, since an addition of 1 to this would involve overspill into B0.)

When the integer is negative, B0 is always 1. This is because every negative number is held inside the machine as the complement of the corresponding positive number and, as described in the previous section, a binary complement is formed by inverting every 1 and every 0 after the least significant 1. Hence B0 (always 0 in a positive number) becomes 1. The addition of a negative number and the subtraction of the corresponding positive number by means of the complement are essentially the same operation, as they are in conventional decimal arithmetic.

The following table shows the format of some positive and negative numbers within a 24-bit word (B0 is shown, detached, on the far left).

Decimal	*Binary*				
+1	0	00000	000000	000000	000001
+10	0	00000	000000	000000	001010
−1	1	11111	111111	111111	111111
−10	1	11111	111111	111111	110110

BINARY FRACTIONS

A binary fraction is usually held in one word. A *binary point* is conventionally considered to stand between B0 and B1 of the word although it is not physically represented in the machine; bit positions to the right of the binary point represent 2^{-1}, 2^{-2}, 2^{-3}, that is, ½, ¼, ⅛ and so on, reading from the left.

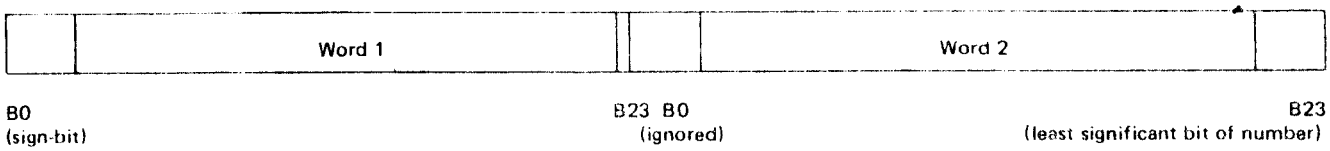
As with integers, B0 is 0 if the fraction is positive and 1 if the fraction is negative; the following table shows the representation of some binary fractions.

Fraction	*Decimal form*	*Binary form*
+ ½	+ .5	0 10000 000000 000000 000000
+	+ .125	0 00100 000000 000000 000000
− ½	− .5	1 10000 000000 000000 000000
−	− .125	1 11100 000000 000000 000000

The fact that there is no physical difference in the representation of fractions and integers means that the programmer is responsible for ensuring the validity of the arithmetic operations for which he is writing instructions; for example, he should never add a word containing an integer to a word containing a fraction.

DOUBLE-LENGTH WORKING

A binary number which is too large to be contained in one word of store can be placed in two (or even more) adjacent words. The least significant bit of the number will then occupy B23 of the word with the higher number in the program (the right-hand word in a graphical representation). The *sign-bit* (indicating whether the number is positive or negative) will be B0 of the left-hand word while B0 in the other word (or words) is set to zero and has no significance in the number.



MIXED NUMBERS

A *mixed number* (a combination of an integer and a fraction) can be held in binary either as separate binary patterns for the integer and the fraction, or as a scaled-up integer. Thus 8.75 can be held as 8 and .75 or as 875.

CHARACTERS

A data word may also be regarded as four units of six bits. These are conventionally numbered *n*0 to *n*3, from left to right.

Within each of these units the six bits may assume one of the 64 different binary configurations (from 000000 to 111111). It is therefore possible to use such units to represent 64 numeric, alphabetic and special (for example, full stop) *characters*, the numbers 0 to 9 being represented by their usual binary form and other characters being represented by an agreed code. See Appendix 2 for the 1900 Series Internal Character Code.

It is in this format of four characters to each word that data is usually stored in the machine immediately after the input process and immediately before it is output. The hardware of the machine cannot perform arithmetic directly on numeric characters, so that character fields must be converted to binary before arithmetic can take place, while answers in binary must be converted to character form before they can be output. However, any program produced by a COBOL compiler will include steps to perform such conversions where necessary.

Note: Character form is used mainly with slow peripherals; data used for arithmetic on tape or disc should be held in binary form. In the interests of program efficiency, conversion to binary of character fields used in arithmetic should be kept to a minimum. For details of techniques to achieve this, see Chapter 9.

Program instruction words

The words which contain the instructions of a 1900 program also consist of configurations of 24 bits.

These words have two main types of format, depending on whether the instruction is a *normal instruction* or a *branch instruction*.

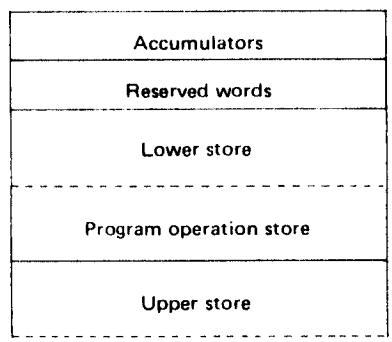
A *normal instruction* consists of:

Bits	*Function*
B0 to B2	A binary number referring usually to an accumulator.
B3 to B9	A *function-code* which indicates the particular operation to which the instruction refers (for example, *addition or subtraction*).
B10 and B11	A number which refers to a *modifier* if one is required.
B12 to B23	The relative address of a word in core store containing the data on which the instruction is intended to operate.

A *branch instruction* differs in that no modifier is ever specified; the function code occupies B3 to B8 and B9 to B23 refer to the address of a word containing the instruction to which the branch is to be made.

The layout of store

The area of store with which the programmer is concerned (that is, the locations that follow the area occupied by Executive) can be represented by a diagram such as that below:



In the following description, the limits of each of these areas are defined in terms of the relative addresses and not the absolute addresses of the locations concerned.

ACCUMULATORS (LOCATIONS 0 TO 7 INCLUSIVE)

The format of a program instruction word allows for the addresses of two locations containing data to be quoted. One address can be accommodated in the last twelve bits (B12 to B23) of the instruction word; this address may be any number up to 4,095, which is the largest number which can be held in binary in twelve bits.

The other address must be one which can be contained within the first three bits of the instruction word; it must, therefore, be one of the locations numbered 0 to 7, since the number 8 in binary (1000) occupies four bits. These first eight locations are known as *accumulators* ('accumulator' is conventionally abbreviated to X); when any operation involves two sets of data, one of these sets must first be placed in an accumulator so that the instruction word may make reference to it.

Some operations (for example, those which shift individual bits within a word) can only be performed on data held in an accumulator.

RESERVED LOCATIONS (LOCATIONS 8 TO 44 INCLUSIVE)

This part of store consists of locations used by Executive or by various software packages which facilitate input and output processes. These locations must not therefore be used by the programmer except when employing such software.

LOWER STORE

Lower Store, usually abbreviated to Lower, can be defined as the remaining locations in store which can be addressed directly by a program instruction word. This may therefore include all the locations up to and including the location with the relative address 4,095; the actual size of Lower will, however, depend on the requirements of the particular program concerned.

Data contained in Lower falls into two categories:

1 *Lower presets*. These are constant (or 'preset') values which are required (or likely to be required) each time the program is run and are therefore specified by the programmer as the contents of certain addresses when the program is written. An example might be a table of tax rates in a program written to calculate employees' pay.

2 *Lower variables*. This is data which varies during the running of the program and cannot therefore be directly specified by the programmer who merely allocates and names sufficient locations to contain the required data. In a payroll program such variables might include the number of hours worked by the employees concerned.

THE PROGRAM OPERATION STORE

The remaining locations beyond those occupied by Lower variables may be used to contain program instructions. Referencing these locations is not a problem since a *branch instruction word* which performs the addressing of other instruction words is provided with fifteen bits for this purpose and can contain numbers up to 32,767 in binary.

Data, as well as program instructions, can be held in any store locations which are available beyond location 4,095; this is known as *Upper Store* or *Upper*. The referencing of data in Upper is described in Chapter 9.

UPPER STORE

Data stored in Upper is also divided into *Upper presets* and *Upper variables*.

THE HARDWARE OF THE 1900 SERIES

The 1900 Series of computers is provided with the facility known as *standard interface*. This takes the form of a multi-pin plug and socket, and is the standard connection between the central processor and the peripheral devices. The control requirements of each kind of peripheral are designed to fit within this pattern, allowing great flexibility in the attachment of peripherals to a central processor; and the processor itself may be changed, within certain limitations, for a more powerful model, without disrupting the entire system.

Central processors

Central processors differ mainly according to:

1 The size of core store (expressed in multiples of 1,024 words).

2 The speed at which program steps are performed.

3 The extent to which certain functions are carried out by hardware operations or by software.

(For programming purposes, however, the differences included under 3 are not of great importance.)

The current 1900 range of central processors have sizes of core store varying from 4K to 256K words. The *store cycle times* of these machines, in other words the time which the processor requires to access a location in store, is usually measured in nanoseconds.

All these processors except the 1901 and 1901A have facilities for the running of more than one program at a time.

Peripherals

Peripheral devices, as explained earlier in this chapter, transfer data to and from the central processor.

The 1900 Series has a wide range of peripherals, which fall into two classes. Slow peripherals include card readers and punches, paper tape readers and punches, and line printers; fast peripherals include magnetic tape and various direct access devices.

Among the speeds available on slow peripherals are the following:

Card readers	:	300, 900 and 1600 cards a minute
Paper tape readers	:	300 and 1,000 characters a second
Card punches	:	33, 100 and 300 cards a minute
Paper tape punches	:	110 characters a second
Line printers	:	300, 600 and 1,350 lines a minute.

This section provides a brief introduction to each type of peripheral, the form in which data is transferred and certain characteristics and facilities which are of interest to programmers. Direct access devices, however, will only be given a brief mention. A separate chapter, Chapter 11, is devoted to direct access with COBOL.

For full details of 1900 peripherals, reference should be made to the following ICL 1900 Series manuals: *Basic Peripherals* (for slow peripherals), *Magnetic Tape*, and *Direct Access*.

BASIC DEFINITIONS

Certain basic terms relating to peripheral transfers in COBOL are defined here; further information on their particular application in COBOL programming is given in Chapter 6.

file

Data on an input or output peripheral unit is held as a continuous stream of material known as a file. For example, a pack of cards or a set of printed lines could be regarded as a file. The term signifies not the medium itself (for example, cards or magnetic tape) but a set of related data, as it is held collectively on the device.

buffer

Each file has its own buffer, an area of store in Upper or Lower data into which input data is read from a peripheral. Similarly, output data is written from the buffer to the peripheral unit. There will normally be one buffer per file, although two can be specified if required. This technique is known as *double buffering* and can increase the transfer rate, since while one buffer is being processed, the other can be filled or emptied.

A buffer holds one unit of input or output. This may be a *record* (see below) or, as in the case of magnetic devices, a number of records. The programmer does not deal directly with the buffers, but instead he considers each file to have a *record area*; this will contain the next record after the previous one is read, or it is where the next record is built up before it is written. The record area may or may not be the actual buffer for the file. This varies according to the type of peripheral, and as to whether double buffering is being used. At any event, it need not concern the programmer.

record

Files of data held on peripherals are divided logically into subdivisions known as records. For example, a punched card or a line of print might constitute a record. This is the basic unit of input or output for the programmer, in other words, the unit in which data is transferred for processing; (on magnetic peripherals, the physical unit of input or output is a block or bucket which may contain many records, although each record is still processed separately).

field

Records are further divided into smaller sections called fields, such as different items on one card, or within a line of print. In a record giving details of an article to be sold, the fields might consist of colour, weight, size and price.

SLOW PERIPHERALS

Data is represented on peripheral devices in a form which varies according to the type of peripheral. Slow peripherals are sometimes known as *character peripherals*, because their input and output involves data in character rather than in binary form. The nature of card peripherals, paper tape peripherals and the line printer is considered below.

Card peripherals

Data can be input or output represented on cards, which are punched by a card punch and read by a card reader. Each card contains 80 columns, each of which corresponds to a single character. A card can therefore take up to 80 characters. A card column consists of 12 punching positions, some of which are punched with holes in a pattern which represents a character. Each character has a unique combination of holes punched in different positions, according to the 64-character 1900 Card Code. This code is given in full in Appendix 3.

Card reading starts at column 1 and can be stopped at any character if no further data is required from that card. Card punching must always be exactly 80 characters.

To indicate to the program that a pack of cards is complete and has been read, an extra card is placed at the end of the pack, punched with four asterisks in columns 1 to 4. (This must be followed by a blank card if double buffering is used.)

Paper tape peripherals

Another medium for input and output is paper tape; this is punched with data by a paper tape punch, and the data is read by a paper tape reader. Standard paper tape is known as 8-track tape. Although 5-track, and other kinds of tape, can be handled by paper tape peripherals, COBOL programs will usually only be concerned with 8-track tape. This sort of tape is so called because it has eight possible punching positions in each frame on the tape. One of these is used for parity checking, an automatic check on the validity of the punched pattern, and the remaining seven tracks are reserved for data.

Each character is represented by one frame of the tape punched with a combination of holes in the seven positions. Characters recorded on 8-track paper tape must be chosen from the 1900 Paper Tape Codes, which are given in Appendix 4. This code, in full has 128 punching patterns, but because the 1900 character has only six bits, only 64 combinations can appear in internal machine code. All the alphabetic and numeric characters and common symbols can be represented but it is wise to avoid the use of the symbols $]↑and←in data input to the object program.

As described above, each character in the 8-track paper tape code consists of seven data bits plus a parity bit. On the other hand, in the 1900 internal machine code, each character is represented by six bits. To be able to accept data into store from paper tape, the tape reader drops each parity bit (after checking to ensure that parity is correct) and then converts each seven-bit character into a six-bit character in store.

Since only 64 combinations are available in a character position in store, and 128 combinations are possible in a paper tape frame, certain characters which are different on the tape become the same pattern in store.

Where ambiguity could arise, shift symbols are introduced to distinguish between the characters which have identical bit combinations. The paper tape reader inserts a *shift character* in front of the data character in core store. Thus some single characters on the paper tape, with the addition of the shift symbol, become two characters in the computer store.

COBOL is normally limited to use of the alpha shift which contains all the normal 64-character set. Since these characters can be assumed to be in alpha shift, no shift character is necessary. Therefore the programmer need not usually concern himself with the other characters requiring shift symbols. An exception to this, however, is the *newline* character, which is used to mark the end of variable length paper tape records. The newline has to be held in store as two characters, because it belongs to delta shift, and will therefore be read into store as #76 32, which would print as †*. The shift character #76 is inserted by the tape reader (or by the program on output) to distinguish between newline and the asterisk character, which is also read into store as #32. The important point to note is that *two* character positions must always be allowed for newline in store.

Both input and output records on paper tape may have a maximum of 4095 characters. Records may be either fixed or variable length. Fixed length records must all have the same number of characters, unfilled positions being punched with space characters. For variable length records, the actual number of characters in each record is punched, followed by a newline, by which the tape reader recognises the end of the record. In this case, records must not exceed 4093 characters and the newline (two characters).

The end of the paper tape file will be indicated by a special character punching of four asterisks followed by a newline (two newlines if double buffering is used).

Line printers

Data in character form is usually output on the line printer. 1900 line printers are able to print either 96, 120 or 160 characters per line, but in COBOL, the normal print record should be 120 characters long. 96 or 160 characters are however acceptable if the printer used is of this specification. Each character in store is printed as a separate character, and will belong to the normal 64-character set. It is usual to fill out each line in store with space characters (#20) at the right hand end, to equal the length of line on the printer; however, this is only essential on some models of printer.

Paper tape control loop

The spacing of the paper must be controlled when output is made on the line printer. The paper used for printing is divided into *forms* by horizontal perforations; forms can vary in depth, but the most common is 45cms deep. Sprocket holes are punched down the sides of the paper.

In order to plan the spacing of printed documents, it is essential for the programmer to know the position on the form at any time and to determine exactly where items are to be printed. Spacing can be controlled by a paper throw of one or two lines as specified in the program and performed by a software loop or by a hardware facility known as a paper tape loop.
This loop is an 8-channel tape into which holes are punched to control the vertical format of printing, each channel indicating a position on the print page. The effect of this is that the printer can be instructed to space lines until a punching is detected in a specified channel of the loop. At this point, spacing stops and printing will begin.

The printer is equipped with a reading head, which senses the holes in the loop and transmits signals to the control electronics in the printer. When the paper is fed into the printer, the appropriate loop is selected by the operator and put on the printer. The two are then lined up so that a punching is sensed in the first channel of the loop at the position of the first line of a printing form. Therefore, every time the paper is moved one or more lines, the loop moves one or more sprocket holes. Whenever a paper throw of more than two lines is required, the program can initiate a throw which continues until halted by a hole sensed in the specified channel of the loop.

On the loop, seven channels are available to COBOL users (or five on the 2401 printer).

Channel 1 is head of form, which represents the first print line of a document. When the printing is set up, the control loop is automatically positioned so that the hole punched in channel 1 is under the sensing head. Channels 2 to 7 are used to represent stopping positions on the paper; as a result of a paper throw, movement continues until a hole is detected in the specified channel. As far as COBOL is concerned, it is best if punchings to halt paper throw in the body of the form are all in the same channel chosen from channels 2 to 7. Channel 8 is not used in COBOL.

The programmer keeps a note of the point he has reached on the form by means of a *line count*, which he sets to the appropriate number of lines on the form. The count is then reduced by one each time the printer spaces a line.

The use of the paper tape loop and the line count in COBOL is explained under *WRITE* in Chapter 7.

FAST PERIPHERALS

It has already been shown that fast peripherals are used to store data on magnetic media, at a much greater input/output rate than slow peripherals. Magnetic devices preserve data stored by a program; unlike slow peripherals, they are not used to transfer data directly between the human user and the central processor, but are an important storage medium in addition to the computer store.

Fast peripherals consist of magnetic tape and direct access devices. The latter are described in Chapter 11.

Data is written from store to magnetic peripherals by what is basically a simple copying process. On 7-track magnetic tape, for example, each frame has six data bits, and one word can therefore be held on the tape in four frames. This facilitates the copying of character or binary fields, or a mixture of both, on to the magnetic medium as a series of words.

Magnetic tape

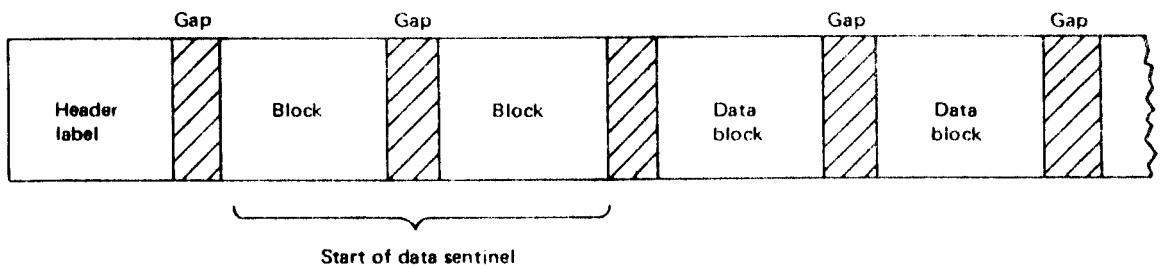
Magnetic tape input and output is handled by a set of subroutines called the *Magnetic Tape Housekeeping System* (MTH), which minimises the operations carried out by the programmer. These subroutines are incorporated in any COBOL program which requires magnetic tape during compilation.

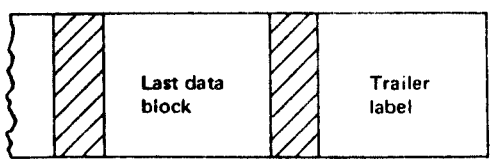
MTH calls on Executive to open magnetic tape files for input, and to close them after output. It also organises output records into blocks, and unpacks blocks of records on input. This process is described below. MTH also deals automatically with continuation reels.

Data is held on peripheral devices in the form of units called records. However, when magnetic tape is input or output, data is presented to or received from the computer in manageable sections called *blocks*. A block normally holds a number of records, the quantity depending on the length of the block and the record size. The main limitation on the size of a block is that a complete block must be held in store at one time. When a block has been read into an input buffer, each record in the block must be unpacked and processed in turn. As described on page 11, each peripheral file has its own buffer, a medium of storage for records before and after processing. Similarly, when a block is written to magnetic tape, the program must pack the required number of records together in the buffer, before writing the block. In COBOL, packing and unpacking are handled automatically.

Although there is no specific limit on the size of a block, it is inadvisable for blocks to be greater than 512 words long, as 1900 utility programs, for example, sort programs, can only handle blocks up to this size.

The standard layout for magnetic tape must be used for any tape files handled by COBOL programs. This layout is illustrated below.





A magnetic tape file, divided into blocks, must also contain a header label, a start of data sentinel and a trailer label as shown above.

The *header label* is used at the beginning of a file to provide essential information about the file. It contains three items of interest to the programmer:

1 The name of the file.

2 The *generation number*, a number given to distinguish one version of the file from another, as for example, an input from an identically named output file.

3 The *retention period*, the number of days for which the file is to be regarded as valid data. After this period, the data on the file may be erased and the tape used for something else. This information is only required for output files.

All this information must be provided for MTH in order that it may identify input files, or write a new header label for output files.

The *start of data sentinel* is a marker on the tape, with a length of two blocks, which shows where the data starts. User sentinels are a similar sort of marker that can be used to divide sets of blocks.

The *trailer label* marks the end of data on the tape, and indicates whether there is a continuation reel, that is, another tape which follows on continuously from the current tape. It also contains a count of the number of data blocks on the tape, which is written and checked by MTH when the tape is read.

Magnetic tape standards as described above also apply to cassette tape. This comprises magnetic tape in *cassettes*, normally for use with small 1900 machines. In this case, the Cassette Tape Housekeeping System is used.

Direct access devices

Direct access devices are so called because they enable data to be accessed readily at any position on the file without running sequentially through the whole file to find the required material.

The main types of direct access device are Exchangeable Disc Store (EDS), Fixed Disc Store (FDS) and Fixed Exchangeable Disc Store (FEDS) on the 2903 and 2904.

1900 direct access is controlled by a similar package to the Magnetic Tape Housekeeping, called the *Direct Access Housekeeping System* (DAH). This provides the same kind of facilities as MTH, and will be added to a COBOL program during compilation if required. More information on DAH is given in Chapter 11.

The word count word

A block of data on magnetic tape or direct access devices contains one or more records. Each record must start with a *word count word* which contains a count in binary of the size of the record in words, including the word count word itself.

On output from a magnetic device, there is no need for the COBOL programmer to place a value in the word count word, as the object program will contain steps which automatically insert the correct value so long as the word count word contains zero. After the record has been moved to the output buffer, the word in the record area will be zeroised, again automatically. However, there may be cases where a given record can vary in size. The programmer should then calculate the size of the particular record in words, and insert the correct value before outputting the record.

THE 1900 PROGRAMMING SYSTEM

Program instructions for 1900 computers are written not in the form of binary numbers (by which they are eventually represented inside the machine) but in one of a number of *programming languages*. Among the languages available are:

PLAN	General purpose language
COBOL, RPG (2903/4)	Commercial languages
FORTRAN, ALGOL	Mathematical/scientific languages

Such languages enable the programmer to state his requirements in a symbolic manner using alphabetic names and characters instead of numeric references only. In some languages (for example, COBOL) this use of alphabetic names enables the program to be written entirely in a restricted form of English.

A program written in accordance with the programming language is known as a *source program*; a special purpose program, known as a compiler, assembler or translator is employed to convert the source program into the entirely numerical 'machine code' or *object program*. Programming languages offer these advantages:

1 Programs can be written and tested in a shorter time.

2 Documentation of the program will be available in a readable form.

3 The maintenance and revision of programs is simplified.

4 The computer procedure is more easily understood by people not trained in machine code.

5 Some languages can be used on different types of computers and thus permit the exchanges of ideas and programs among computer users.

When writing in a programming language the programmer does not usually have the same detailed control over his program as when using machine code, but the advantages listed above considerably outweigh this fact. Some languages are intended for use on one type of machine only, and these usually give the programmer a much closer control over the final program. In the 1900 programming system this function is performed by PLAN, which is the basic programming language of the 1900 series.

A language such as PLAN, which is written for a particular type of machine, is known as a *low-level language*, whereas languages that may be used on more than one type of machine are said to be *high-level*. The main point of distinction is that in a low-level language each instruction in the source program usually corresponds to one instruction in the object program, whereas in high-level languages one instruction in the source program may generate a number of object program instructions. It is possible however for program segments written in PLAN to be combined with segments written in other languages, and in particular PLAN is used in association with COBOL and FORTRAN.

Commercial languages

COBOL and RPG have separate compilers each of which is capable of generating a number of machine instructions from one statement in the source program. They therefore represent higher level languages than PLAN.

COBOL is a common business language which is used in association with many types of computer throughout the world and which enables the programmer to state his requirements in basic English.

Mathematical scientific languages

For programming problems of a mathematical or scientific nature, several languages are available, including FORTRAN. In such languages, arithmetic operations to be performed on data are written as a series of source program statements using a notation similar to standard mathematical formulae. Apart from the basic operations of addition, subtraction, multiplication, division and exponentiation, there are also facilities for performing trigonometric and logarithmic functions.

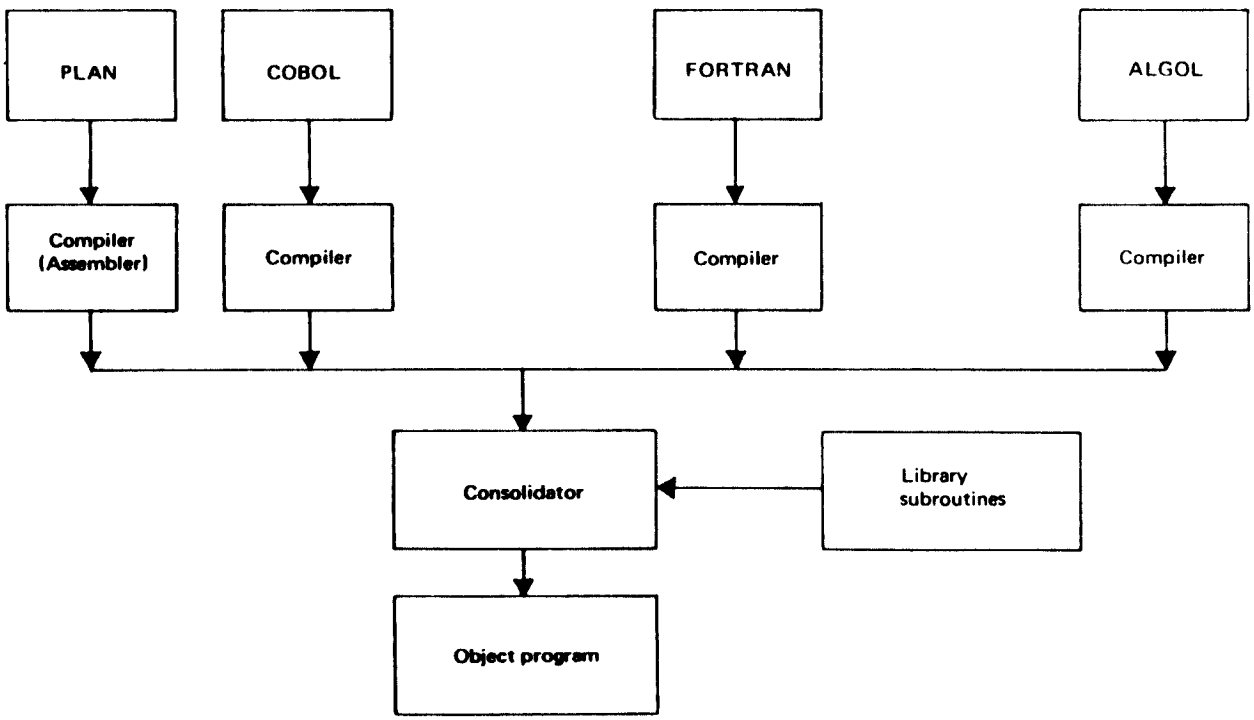
The compilation procedure

Each language in the 1900 programming system has its own compiler; however, conversion is not made directly into the machine code, but into a *semi-compiled* form. See Chapter 8. The full procedure is as follows:

Phase 1. A program can be written as a number of parts or segments, each of which will be compiled separately using the appropriate compiler.

Phase 2. The semi-compiled segments are combined by means of a special purpose program known as a *consolidator* (alternatively this function may be performed by the compiler itself). COBOL segments may be combined with segments written in other languages, and also at this stage any required *library subroutines* (see below) are incorporated. The output from the consolidator is the final object program, with all cross references between segments consolidated and 'leader' information prepared for use when the program is loaded to store for running.

The diagram below demonstrates the sequence:



Subroutines

In the source program there are facilities for automatically incorporating *library subroutines*; these are software routines produced and supplied by ICL which save the programmer the trouble of writing his own instructions for certain very commonly required operations.

Such subroutines are generally held on magnetic tape or discs in semi-compiled form with appropriate 'leader' information. When a subroutine is called for in the source program, the consolidator arranges for the subroutine to be included in the object program during Phase 2 of compilation.

In addition, complete *packages of subroutines* are available to handle input and output from both magnetic devices and slower peripherals. These are MTH and DAH as defined earlier in this chapter.

Chapter 2 Executive and the central processor

All computers in the 1900 Series use an Executive program as part of the control of the machine. It is in many ways irrelevant to the user that this control is performed by program and not by hardware. This is evident from the fact that programs are interchangeable between all processors in the series, even though the division of functions between hardware and software is different in the various cases.

The Executive program may in fact be regarded simply as a permanent part of the machine, a means whereby the designer has been able to make a machine with more convenient facilities than if hardware alone had been used.

It is used in four main fields of activity:

1 Control of peripheral devices and execution of peripheral transfer requests.

2 Provision of *extracode* facilities.

3 Control of multiprogramming (except on 1901 and 1901A processors).

4 Communication with the operator and execution of operator directives, including loading programs and deleting them.

The processor is capable of working in two modes, *Executive* and *Normal*. This need not concern the user at all, as Executive mode will occur only when Executive is being obeyed.

Entry into Executive may arise either as the result of a program carrying out an instruction which involves Executive action in its implementation (a *voluntary* entry by the program), or because of some situation normally outside the central processor, of which Executive should be aware (an *involuntary* entry). The commonest examples of the two types are program instructions which call for peripheral transfers, and signals from peripheral units that transfers have been completed. In either case Executive is entered at the appropriate one of two fixed entry points and indicators are set which allow the cause of the entry to be determined. The return link to the current program is stored within the program's area while the interruption is in progress.

(These interruptions should not be confused with the *hesitations* during which data is transferred between the store and peripheral devices. Hesitations are serviced entirely by hardware, and may occur while the machine is in Normal or Executive mode.)

CONTROL OF PERIPHERALS

When the computer is running in Executive mode, certain orders are allowed to address the peripheral electronics, both for the purpose of determining their state at any time, and also for setting them in order to initiate some action. In the majority of cases the addressing is done by sending control codes across the standard interface to the device; the device then either transmits back a reply, or takes action appropriate to the codes.

In some circumstances, the device itself may initiate the transmission of information to Executive. If the computer is running in Normal mode, this will immediately result in an interruption which forces an entry to Executive. If at the time of the event the computer is already 'interrupted' the interruption is delayed until the current Executive process is completed. The device number of the unit causing the interruption is set up as a bit in a special word. Executive scans the word to obtain the number, and then by means of a request code causes the device to transmit further details as to the cause of the interruption. Commonly the cause will be the completion of a transfer; other possibilities vary with the type of peripheral, but include such things as parity failure on reading paper tape, magnetic tape, or direct access device, reading error or hopper empty or stacker full on card reader, paper low on printer, and so on.

Peripheral transfers

An instruction requesting a peripheral transfer is written into a program in the same way as a normal instruction. When it is detected during program running, the function code indicates that Executive action is required, and a voluntary entry is made in the manner described above.

Executive then analyzes the instruction, first to see that it is a transfer request (since all voluntary entries are dealt with in this way), and then to determine the type and number of the peripheral involved.

The instruction indicates a word or group of words in the program's area which defines the peripheral unit and the core store addresses to be used in the transfer.

Before the transfer is initiated certain checks are performed:

1 Is the transfer order valid? Is the peripheral allocated to this program, or could the transfer overwrite Executive (or, on a multiprogramming machine, another program)?

2 Is the peripheral unit in question available to carry out a transfer? (It may, for instance, still be busy carrying out a previous one.)

3 If this transfer were started, would the input/output hesitation system be liable to overloading, resulting in lost data? Executive checks for possible overloading of the hesitation system on the smaller processors.

4 If the peripheral unit is one that shares a data channel with other units (for example, magnetic tape), is the channel available?

So long as these checks are all successful the transfer can be initiated, and control returned to the program which made the request.

If the data area does not lie within the store limits of the program or the peripheral addressed is not allocated to the program, the instruction is illegal. The program is suspended awaiting operator action, and a message is printed on the console typewriter.

If the peripheral unit is busy the program is suspended awaiting availability of the unit. On all machines except the 1901 and 1901A another program may be entered. If the transfer would be liable to overload the input/output system, the program is suspended until one of the current transfers finishes, when a further attempt is made.

In the case of no channel being available, the procedure is similar to that for the device itself being busy, except that in the multiprogramming case Executive must make a note of the first request refused for this reason at any one time. This avoids a high-priority program completely shutting out one with lower priority.

When a peripheral transfer finishes, the unit gives rise to an involuntary Executive entry (as described above). If a program is suspended awaiting completion of the transfer, the suspension is removed.

EXTRACODES

Internal operations, as well as those affecting peripherals, may actually be performed by Executive subroutines (which are known as *Extracodes*). The function codes of such instructions are detected by the hardware, and a voluntary entry is made. As described above, the instruction is made available to Executive, which analyses it and carries out the appropriate actions. Examples of Extracodes include multiply and divide (on some smaller processors) and floating point operations (on non-scientific processors).

MULTIPROGRAMMING AND DUAL PROGRAMMING

Many programs use the central processor for only a small proportion of the time during which they are being run; the rest of the time is occupied by peripheral transfers to or from the machine. In order to avoid the central processor standing idle, facilities are available on all 1900 machines except the 1901 and 1901A to enable two or more programs to be run at the same time; thus while one program is engaged in peripheral transfers another can use the processor. This system is controlled by Executive.

The sequence of events

Under the multiprogramming system each program is assigned a priority number by the programmer. At the start of a run the program with the highest priority is entered. When that program is suspended the program with the next highest priority is entered, and so on. This procedure, which is supervised by Executive, can best be illustrated by a simple example, in which only two programs, A and B, are involved.

Assuming that program A has the higher priority this program will always use the central processor until it requires a peripheral transfer to take place. When this happens it issues an instruction which transfers control to Executive and Executive initiates the transfer. Program A then continues until it needs to use either the data which is being read in by the peripheral (in the case of transfer from input peripherals) or the area of store from which the data is being transferred (in the case of transfer to an output peripheral).

If, however, the transfer is still taking place, then program A cannot proceed and control is passed to Executive which ascertains the program with the highest priority which can proceed (in this case, since there are only two programs, this will of course be program B). Program B is then allowed to use the central processor until either:

1 The peripheral transfer is completed, in which case program A is 're-activated' since it has higher priority, or

2 Program B itself requires to make a peripheral transfer. In this case use of the central processor will be passed to whichever program finishes its transfer first.

The sequence of events is shown in the diagram below:

PROGRAM A ACTIVE	PERIPHERAL TRANSFER	PROGRAM A ACTIVE
	PROGRAM B ACTIVE	

Assigning priority numbers

The combination of programs which will make most efficient use of computer time is sometimes termed the *optimum program mix*. Such a mix can be obtained if the programmer assigns priority numbers to the programs involved according to some well-defined system.

The only important factor to be considered in such a system is the proportion of time for which each program involved is occupied in peripheral transfers or in using the *mill* (the arithmetic unit in the central processor); a program which needs to spend a great deal of time in peripheral transfers is said to be *peripheral-limited*, whereas one in which a high proportion of actual processing takes place is said to be *mill-limited*.

If, in the example described above, program A were a mill-limited program there would be few *interrupts* (or periods during which A was suspended because of a peripheral transfer) and hence program B would rarely be entered. If, however, program A were peripheral-limited then it is possible that this program would be suspended frequently so that program B could be entered and obeyed. The extent to which one program in a multiprogramming system prevents others being entered is said to be the degree of *lockout* exercised by that program.

Since the degree of lockout is determined by the number of times a program is suspended, and a program is mostly suspended as a result of waiting for a peripheral transfer to be completed, then the degree of lockout may be shown as the ratio between program mill time and total run time. The higher the degree of lockout exercised by a program, the lower the priority which should be assigned to it (so that it will not prevent other programs from being entered); hence the priority number of a particular program may be calculated from the simple formula:

$$100 \left(1 - \frac{\text{mill time}}{\text{estimated mill time total run time}}\right)$$

If the multiprogramming example were expanded to include three programs (A, B and C) with the characteristics given below then the priority number of each program could be calculated as follows:

Program A
Mill time = 50 seconds
Estimated run time = 10 minutes (600 seconds)

$$100 \left(1 - \frac{50}{600}\right) = 92$$

Program B
Mill time = 2 seconds
Estimated run time = 15 minutes (900 seconds)

$$100\left(1-\frac{2}{900}\right) = 99$$

Program C
Mill time = 100 seconds
Estimated run time = 5 minutes (300 seconds)

$$100\left(1-\frac{100}{300}\right) = 67$$

Program B would therefore have the highest priority, program A would have a lower priority and program C would have the lowest priority of all.

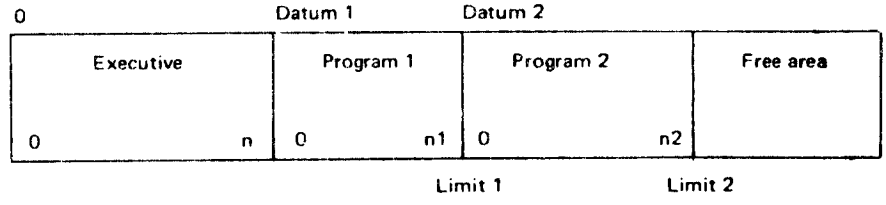
Allocation of store

Apart from the initial assignation of priority numbers by the programmer the procedure of multiprogramming is wholly supervised by Executive. One important aspect of this supervision is *store allocation*.

As has been described in Chapter 1, each location in store used in a particular program is addressed relative to the first word of that program (the *Datum* of the program). However, when the program is loaded each word used in it will occupy a specific location in the store which will bear its own *absolute address*; the latter number will be relative to the store as a whole (including that section of it occupied by Executive) and will not be known to the programmer. Since, however, the absolute address must be available to hardware, Executive arranges for the hardware to 'calculate' it for each word of the program, by adding to the relative address of that word the absolute address of the Datum for that program.

The same process is followed when, during multiprogramming, two or more programs are loaded one after the other. As each program is loaded Executive checks that sufficient store space is available for it; it then 'allocates' an area in store to that program. Executive then sets into a hardware register the absolute address corresponding to word 0 (the Datum of the program); this number is then added to the relative address of each word used in the program. The last address used in the program (called the *Limit*) is also set by Executive and hardware checks that no reference is made by that program to a store location with an absolute number higher than the one borne by Limit.

In this way when a group of programs are loaded together each one is 'protected'; there is no fear that, for example, instructions in one program might be used to operate on data stored for use of another program. The layout of two programs stored together may be illustrated by the diagram below:



When one of a group of programs is 'deleted' then the area of store which it occupies is released; this may leave a gap between two remaining programs. It is, however, convenient that all unused store should be in a continuous area for allocation to new programs and that all existing programs should be densely packed in store. Therefore Executive ensures that any gap is automatically eliminated by moving later programs and amending the values to which Datum and Limit are set for those programs.

Allocation of peripherals

Every peripheral within a particular data processing system bears an absolute number; within a program, however, each peripheral used in that program is assigned a relative number by the programmer.

When a program is loaded, Executive must first check that sufficient peripherals are available. The correspondence between the relative numbers of the peripherals involved and the absolute numbers is then retained by Executive in the form of a list, which is referred to in all Executive processes involving programs. There is therefore no risk of two or more programs accidentally addressing the same peripheral unit.

When a program is deleted, all the peripherals assigned to it are released and become available to another program.

OPERATOR COMMUNICATION

All operator communication on 1900 Series computers (except basic 1901 and 1901A) takes place through a console typewriter. This automatically gives a permanent record of all communications in the sequence in which they occurred, a feature which is valuable in any installation but particularly so in multiprogramming operation. The typewriter is used both to inform of unusual situations within the machine, peripheral units, and programs, and also as the means whereby the operator directs Executive to load, activate, and delete programs. For the latter purpose, a set of standard messages is used which are recognizable by Executive and also intelligible as part of the printed record.

Input messages

The format of input messages requires that certain elements be present in a set sequence; extra words or characters may be inserted to improve the readability of the printed version, provided that the framework is correct. The following rules determine the framework, and also indicate how the decoding is performed within Executive.

1 The type of message is defined by a key word.

2 The program referred to is identified by a four-character name, preceded by the symbol # .

3 Other information, such as peripheral unit numbers or entry points, is all numeric. If there is more than one character in a message, they may be separated by any non-numeric character or characters. These numbers are assumed to be decimal, unless immediately preceded by the symbol * in which case they are interpreted as octal numbers.

The details of the procedure for inputting a message are slightly different on the smaller processors. However, the same basic steps must be followed.

1 The operator presses the INPUT key to indicate that he wishes to insert a message.

2 Unless Executive requires the typewriter for an output message, it signals that it is ready to accept by outputting a new line and three spaces. The INPUT key is illuminated.

3 The operator then types his message and, if he is satisfied that it is correct, presses the ACCEPT key. At any time prior to this, the operator may cancel the message by pressing the CANCEL key. A message being input will also be cancelled automatically if Executive wants to output a message. The whole procedure must then be repeated.

4 Executive examines and decodes the message. If acceptable, it types OK in reply. If there is an error in the form of the message, or the program name, it types ERROR X (where X defines the type of error).

On multiprogramming processors all messages (except PRINT PRIORITY LIST) include the name of a program, identified by being immediately preceded by the symbol #.

The following list gives the main types of message which may be input to Executive. It is emphasised that these brief definitions do not include the full message formats; they are intended merely to show the communication which can be made between the operator and Executive. For full details of console operating with Executive, the appropriate reference manuals should be consulted.

Each message begins with a key word, followed by a program name and, normally, certain other fields.

LOAD

The LOAD message causes a specified program to be read into store from the peripheral whose unit number is given. Parameters may also indicate peripherals to be assigned to the program, and a special request may be made for storage allocation.

Example

```
LO #JACK 4 *3600
```

JACK is the program to be loaded, 4 is the unit number of the input peripheral and *3600 is the number of words of store required (in octal).

Note: This message is not used by an operator to load a program directly from magnetic file storage. The FIND message is used for this purpose.

FIND

This message is used to load programs held in magnetic file storage (for example, magnetic tape or E.D.S.).

For magnetic tape, Executive loads the first program held on the specified file, or alternatively, it loads a named program from a second named file (in particular, when it is required to search a library tape). For other forms of magnetic storage, Executive automatically loads a search program to carry out the same procedure. As with LOAD, parameters can also specify peripherals to be assigned to the program and/or storage allocation.

Example

```
FI#JACK #TAPE
```

A file named TAPE is searched until a program named JACK is encountered.

ON
OFF

These two messages set to 1 or 0 (respectively) a specified bit of word 30 in store of the named program. Word 30 is reserved for this purpose. Each bit position acts as a switch enabling the operator to affect the running of the program, for example, in monitoring procedures.

Example

ON # JACK 9 10 21

JACK is the program name and bits, 9, 10 and 21 of word 30 are set to 1.

ALTER

This message is used to alter the contents of a specific storage location in a program, so that it contains a given value, which may be either a decimal or octal number. The message is obeyed without suspension of the program running in store.

Example

AL # JACK 3742 * 35631123

JACK is the program name, 3742 is the address of the word to be altered and *35631123 is the new contents of the word.

GO

If a program is suspended this message causes the program to restart from the point at which it stopped. Alternatively, it can be used to activate a program initially, or to restart a program at the storage location specified.

Examples

GO # JACK

JACK is the name of the program to be restarted.

GO # JACK 20

JACK is the program name and 20 the specific location where the program is to be restarted.

SUSPEND

This message causes the program to be suspended until the operator restarts it with a GO message.

Example

SU # JACK

Program JACK is suspended.

DELETE

This causes Executive to delete the program from the processor. A DELETED message is produced. On a multiprogramming machine all programs in highly numbered store locations are moved to fill in the space originally occupied by the deleted program.

Example

DE # JACK

JACK is the program to be deleted.

DUMP

This message has two different meanings. If no unit number of a peripheral is specified Executive searches currently assigned tapes for a scratch tape. When this is found, a specified program area is written to it in binary format. If a unit number of a slow peripheral is specified, the program area is written to this peripheral.

Example

DU # JACK 6

JACK is the program to be dumped and 6 the unit number of a card or paper tape punch.

OUTPUT

This message has two versions. The first causes the contents of a specified word in a program to be output on the console typewriter. (Word 8 may be output if it is suspected that the program is looping.) The second outputs an area of store to a line printer, card or paper tape punch, starting from the location specified.

Examples

OU # JACK 4764

JACK is the program name and 4764 the address of the word to be output.

OU # JACK 9647 500 7

9647 is the start address, and 500 the number of words to be output. 7 is the unit number of the peripheral to which the data is to be output.

GIVE

The allocation of peripherals to a program is often dealt with by Executive when the program is first loaded. This message, however, allows one particular peripheral of a group to be assigned to a program (for example, if a system has two paper tape readers, one of which has been set up to read in non-standard format). Executive is instructed which peripheral to assign.

Example

GI #JACK 4 0

JACK is the program name, 4 is the unit number of the peripheral to be allocated, and 0 is the programmer's symbolic number for the peripheral.

TAKE

This causes the specified peripheral to be removed from the program, and to be made available for general use.

Example

TA #JACK 6

JACK is the program name and 6 the unit number of the peripheral to be removed from JACK.

PRINT PRIORITY LIST

This message causes Executive to type out on the console typewriter in priority sequence the priority of each program in the machine. The program name is typed followed by the priority number. Unit numbers of peripherals currently assigned to the program are also given.

Example

```
PR
0JACK99 6 26 28 29
OALPH90 7 24 26
```

PR is the key word. JACK and ALPH are the two programs running in the processor. 99 and 90 are the priorities of the programs. All other numbers are the unit numbers of the peripherals currently assigned to # JACK and # ALPH.

REVISE PRIORITY

This message revises the priority of the program and changes its position in the priority list. A revised priority list is then printed.

Example

```
RE # JACK 93
```

changes the priority of JACK to 93.

Output messages

Output messages may originate either from Executive or from a program within the machine. In general, they will be found to be self-explanatory and there will be little difficulty in interpreting them. There are some minor differences between the larger and smaller processors, but operating procedures are largely the same.

Output messages are offset to the left compared with messages typed in by the operator; they may be divided into four categories. Some examples of each category are given below, but these are not intended as an exhaustive list.

1 MESSAGES OUTPUT IN REPLY TO INPUT MESSAGES

All input messages are acknowledged by an Executive output message which indicates whether the input message is acceptable.

If the message has the correct format and can be obeyed, OK is typed out. If the message is incorrect, ERROR X is typed, where X is a character indicating the type of error.

2 MESSAGES OUTPUT ON PROGRAM LOADING

At the time of loading a program, the core allocation is output by the message CORE *nnnn*. A loading allocation list containing details of each peripheral is also typed. For a multiprogramming machine, a list of the priorities assigned to each member of the program is produced.

When program loading is complete, the message HALTED : — LD is output giving the program name. The operator then normally activates the program with a GO input message.

If an input message requesting the loading of a program is unacceptable to Executive, the word FAULT is output following the message and indicating the nature of the fault.

3 MESSAGES OUTPUT DURING A PROGRAM RUN

Various messages from Executive can be output to inform the operator of the state of a particular program currently held in the processor. All these messages originated by the program start with the program name, followed by a word indicating the type of instruction which caused the printout.

Some of these messages are as follows:

DELETED is output either as the result of a program instruction for the deletion of a program or in response to a DELETE input message. It can be followed by up to forty characters explaining the situation.

DISPLAY is output as the result of a program instruction to report an event in the specified program.

HALTED is output as the result of a program instruction requesting the suspension of a program. Again, the situation may be explained by a character string.

ILLEGAL reports a programming error, giving the type of error and the address of the rejected instruction.

LOAD is a request from the program to load a specified tape.

4 MESSAGES OUTPUT ON PERIPHERAL OCCURRENCES

Certain messages are output by Executive to report on peripheral occurrences. These start with the program name followed by UNIT, the unit number and a word defining the type of event. Some of the most common are described below.

BUSY informs the operator that a requested action, particularly the deletion of a program, cannot take place immediately due to the activity of a peripheral.

ERR indicates that following a transfer failure on magnetic tape the maximum number of attempts has been made to transfer the data.

FAIL indicates the failure of an automatic check; for example, that after a parity failure the maximum number of attempts has been made to transfer the data.

FIX requests the operator's attention on the specified peripheral (for example, if it has been accidentally disengaged).

FREE is output as the result of a program instruction requesting the release of a specified peripheral from the program's allocation.

USED AS is output as the result of an instruction requesting the allocation of a specified peripheral as a program's unit *n*.

For an example of a console log, see Chapter 8.

Chapter 3 Flowcharting

INTRODUCTION

The actual writing of a program on COBOL coding sheets should always be preceded by the design of *flowcharts*. These are drawn up by the programmer on the basis of the Program Specification, which contains definitions of input and output, definitions of procedures, criteria for choice of procedures, reference tables and other necessary information.

AIMS OF THE FLOWCHART

A 1900 computer follows the method known as *sequential operation*. This means that all the program instructions are obeyed in strict sequence until a branch instruction is reached which may take the program into another sequence. The programmer's flowchart is a method of displaying the logical sequence in graphical form. It has two main functions:

1 To help the programmer develop the logic of the solution to the programming problem.

2 To communicate this solution clearly and concisely to others.

OUTLINE FLOWCHARTS

The outline flowchart is the programmer's first attempt to represent a programming problem in graphical form; it might be the first stage in the process of breaking down a problem or, in the case of a comparatively simple problem, both the first and the final stage. When a team of programmers is engaged on a single programming project the outline flowchart will usually be drawn up by the project leader.

An outline flowchart has the following functions:

1 To give a diagrammatical solution to the problem in a clear and concise manner.

2 To show each distinct logical path to be taken, depending on the type of record or major difference in data condition, which will lead to a significantly different sequence of actions.

3 To show all input and output functions, so that the reading and writing of each record may be traced.

4 To show all entry points, halts and other actions to be provided by the program.

In addition, the outline flowchart usually has a fifth function:

5 To demonstrate breakdown of the program into precisely defined *segments*.

Segmentation

It is the task of the project leader or leading programmer to consider the value of dividing a particular program into segments; however this technique, known as *segmentation*, is generally applied in all but the smallest programs.

Segmentation in 1900 terms as described here should not be confused with segmentation in COBOL terms, which implies overlay. See Chapter 13.

The advantages of segmentation are as follows:

1 The task of programming can be divided amongst a team of programmers.

2 The program can be more easily amended: segments can be expanded or removed completely and new segments can be inserted without interfering with the rest of the program. (This is achieved during compilation by two special routines, supplied by ICL, which are used to 'update' the source program.)

3 Testing can be carried out on separate segments of a program independently and/or simultaneously, thus considerably reducing the elapsed time for testing the complete program.

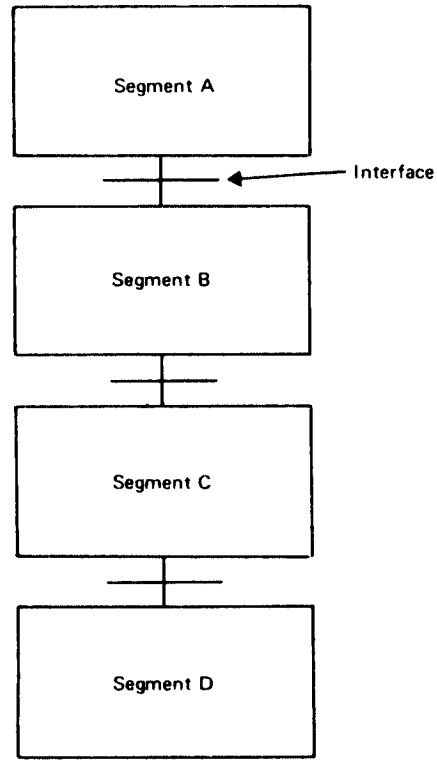
4 The technique of *overlaying* can be used. (This will be explained more fully in Chapter 13.)

If a program is to be segmented two methods are possible, the *Direct Method* and the *Tree Method.*

THE DIRECT METHOD

In this method the segments are in series with an interface between each pair, so that control is passed from the end of one directly to the beginning of the next.

Such a series of segments is illustrated by the diagram below:

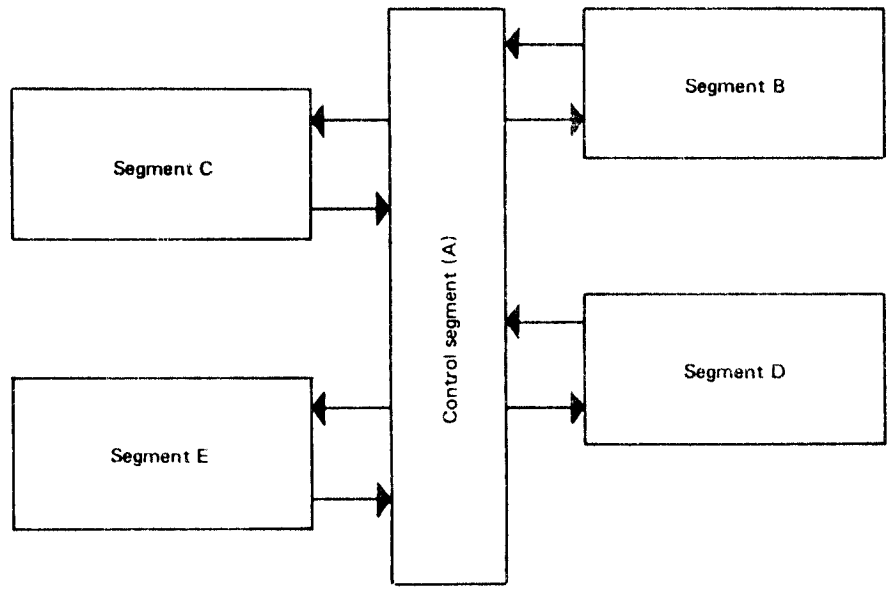


The Direct Method

THE TREE METHOD

This method is of general application but is particularly necessary for overlay programs; it involves the use of a *control segment* or *steering segment*, which transfers control to each of the other segments in turn and has control returned to it after each segment has been used.

The method is illustrated by the diagram below which can be linked to a tree in that it has a control segment for a 'trunk' and segments 'branching' from it.



The Tree Method

The following points should be borne in mind once it has been decided to segment the program:

1 Too many small segments will absorb unnecessary effort in compilation and the arrangements of test data.

2 If there are only a few large segments, the whole purpose of segmentation (that is to break the program down into manageable units) will be defeated.

3 A record should be maintained of what each segment is supposed to do.

DETAIL FLOWCHARTS

After the completion of the outline flowchart it is usually necessary to draw up a series of flowcharts to show individual segments or routines of the program in greater detail. The purpose of these flowcharts, known as *detail flowcharts*, is as follows:

1 To settle the programming techniques to be used.

2 To provide clear directions for coding.

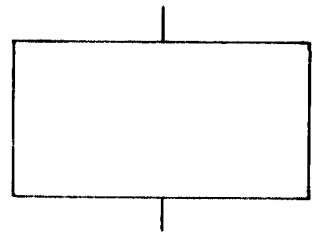
3 To make the coded program readily intelligible to others.

Although only one level of detail flowcharting is normally required, a particularly complicated segment can be broken down still further, flowcharts being prepared of routines or groups of instructions within the segment. Detail flowcharts should differ from outline flowcharts in that while the narrative used in the latter is problem oriented, that is, uses the terminology of the task for which the program is designed, that in the former is machine-oriented, that is, is capable of easy translation into the programming code which is to be used.

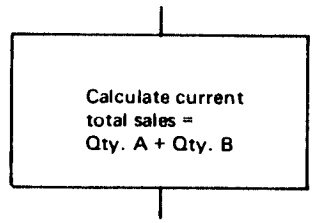
FLOWCHART SYMBOLS

The following symbols which are in accordance with the ECMA standards for program flowcharts, are used when flowcharting. Symbol size may be varied but the shape must be easily recognisable. These symbols are all contained on the ICL flowcharting template.

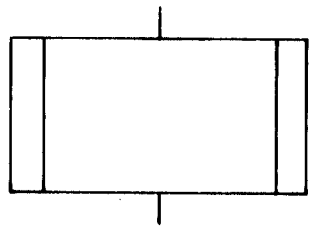
Arithmetic, Transfer or Other Process



The process should be described briefly within the box. This description may cover several connected operations, provided there is no intervening decision or entry of flow from another part of the program. For example

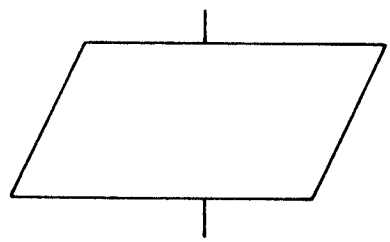


Subroutine



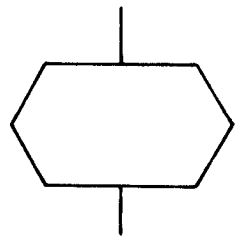
This is regarded in the same way as any other process, except that it is given a name, and is the subject of a separate detailed flowchart (unless it is a standard library routine). The name should be written in the rectangle. Other details are best given in a schedule of subroutine specifications attached to the outline flowchart. Note that subroutines may be used within subroutines.

Input/Output



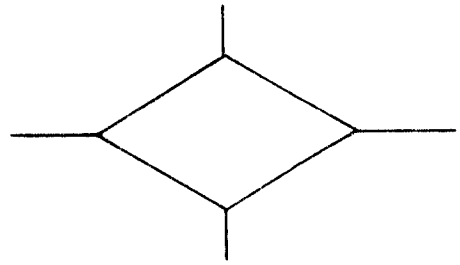
Details of the input or output process required are written within the symbol.

Preparation

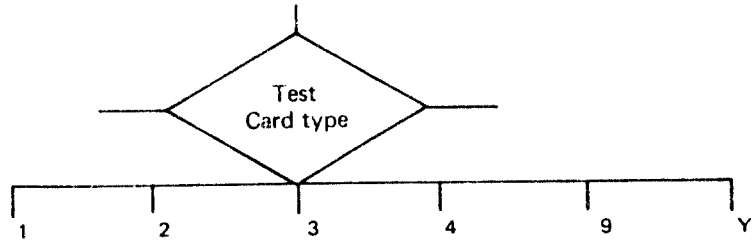


This symbol represents modification of an instruction or group of instructions which changes the program itself, for example, set a switch, modify an index register or initialize a routine.

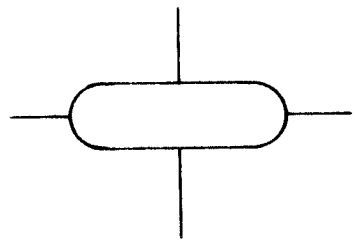
Decision



The normal form of this symbol provides for one entry and two or three exits; if more are required (which may happen, for instance, in determining action according to class-of-record or similar codes), a horizontal line is drawn through the bottom point of the diamond, and the exits spaced along this line (as in the example below).



Stop/Start

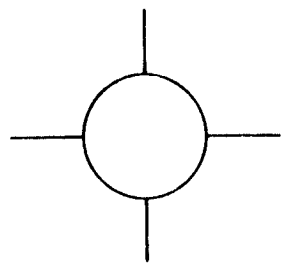


This symbol will usually contain one of the following standard narrations:

ENTRY	beginning of run
E.O.R.	successful end of run
STOP	abandonment
HALT	pause for operator action
BEGIN	entry to subroutine
LINK	exit from subroutine

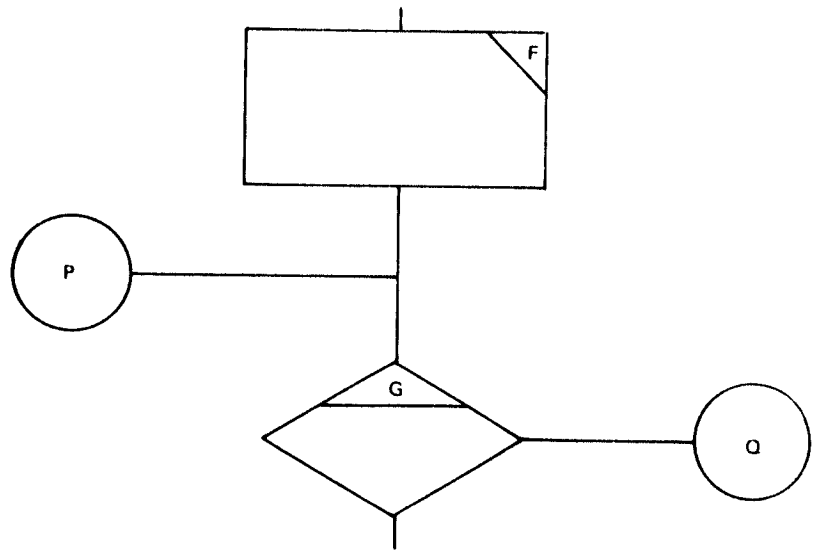
STOP and HALT points should be numbered inside the symbol for reference, and for operator communication.

Connectors

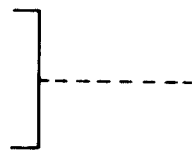


Connectors are used in pairs as substitutes for flow lines that would be awkward to draw or that would pass from one page to another; connectors which indicate that flow is to pass to another part of the chart are known as *tail connectors* and are offset to the right of the main line of flow. Connectors to which the program jumps are called *head connectors* and are offset to the left.

The illustration below shows both kinds of connector as they might appear in an outline flowchart: rules for labelling connectors are given later in this chapter.



Comment



This symbol is used to add descriptive notes or comments to a flowchart. It is particularly useful for identifying segments.

FLOWCHART DESIGN

Flowcharts are drawn on A4 or A3 sized paper, subject to the limitation that no page may contain more than 25 boxes. The general direction of flow is from top to bottom, left to right of the page.

Flowcharts must be clearly titled, dated, labelled and referenced with the author's name.

All text associated with a symbol is written in block capitals. Where possible the text is written within the symbol, but if this is not practical the text may be written in a comment symbol.

FLOWCHART REFERENCING

Referencing method

Each box on a flowchart is given a reference letter or letters. If a box is expanded to a lower level chart, it must also contain an asterisk.

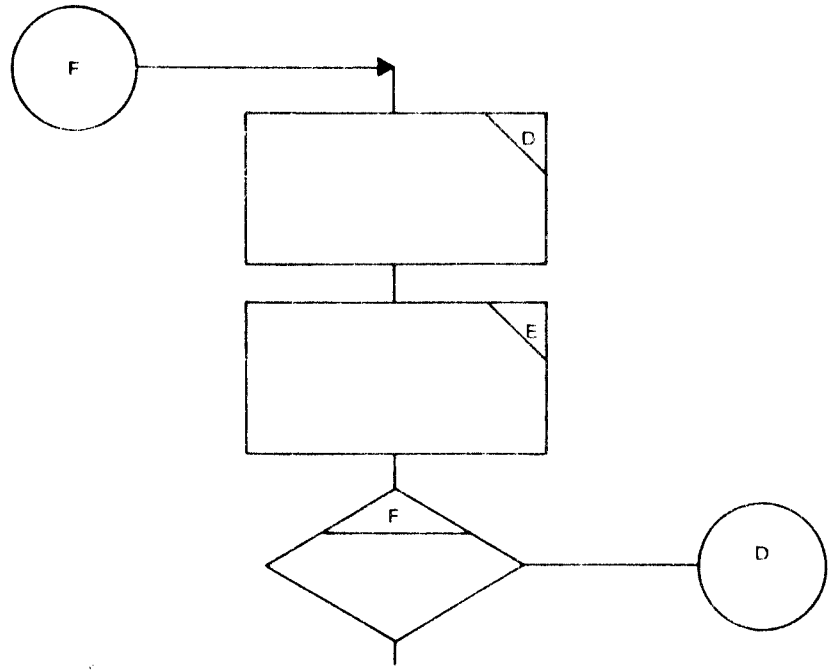
The boxes on the outline flowchart are referenced by a single letter in the range A to Y. The letter Z is reserved for subroutines, which are referenced from the beginning of the program as ZA to ZY, ZZA to ZZY and so on.

At the next level the reference of the first box on the page and the page identity are the same as that of the box which is being expanded. The references of other boxes on the page are formed by suffixing the page identity with the letters A to Y. For example if box D on the outline flowchart is being expanded, the first box of the expansion flowchart is referenced by D and the second box by DA, the third by DB and so on. Again boxes which require further expansion contain an asterisk and the next level flowchart is referenced in the same way. If box DB is being expanded the boxes in the expansion flowchart will be referenced by DB, DBA, DBB and so on.

Referencing connectors

Tail connectors are given the same letter as the symbol to which the branch is made.

Head connectors, however, are given the same letter as the symbol from which the branch was made.



Restrictions

To avoid any two pages having the same identity, the first box on any detailed flowchart page must not be expanded.

The number of boxes per page is limited to 25 as there are only 25 available letters. However, it is recommended that there should be no more than 20 boxes per page in the first instance. This allows alterations to be made without extensive relabelling.

EXAMPLE FLOWCHART

A magnetic tape master file contains customers' names, addresses, balance of payment and other details arranged in rising sequence of customer account number. Details of further transactions are punched into cards which are also in rising sequence of customer account number; there will only be one card (at most) per customer. The main file is to be updated to give a new balance of payment for each customer.

There are no new records, that is, there should always be a main file record for each card and the card file should end before the magnetic tape file. Some main file records will not be amended, that is, will have no corresponding card.

Figure 1 shows the flowchart for this problem. This is an outline flowchart but most of it is sufficiently detailed in a simple problem like this for the programmer to code directly from it. Only two boxes L and O need further expansion and these specify the error procedures.

Box A unsets a switch. This could correspond with setting a field to zero. This switch is set, that is the field is set to non-zero by box Q when the card file ends and is used for two purposes:

1 To control the sequence so that the remainder of the records on the input magnetic tape file, which have no amendments, are copied to the output tape.

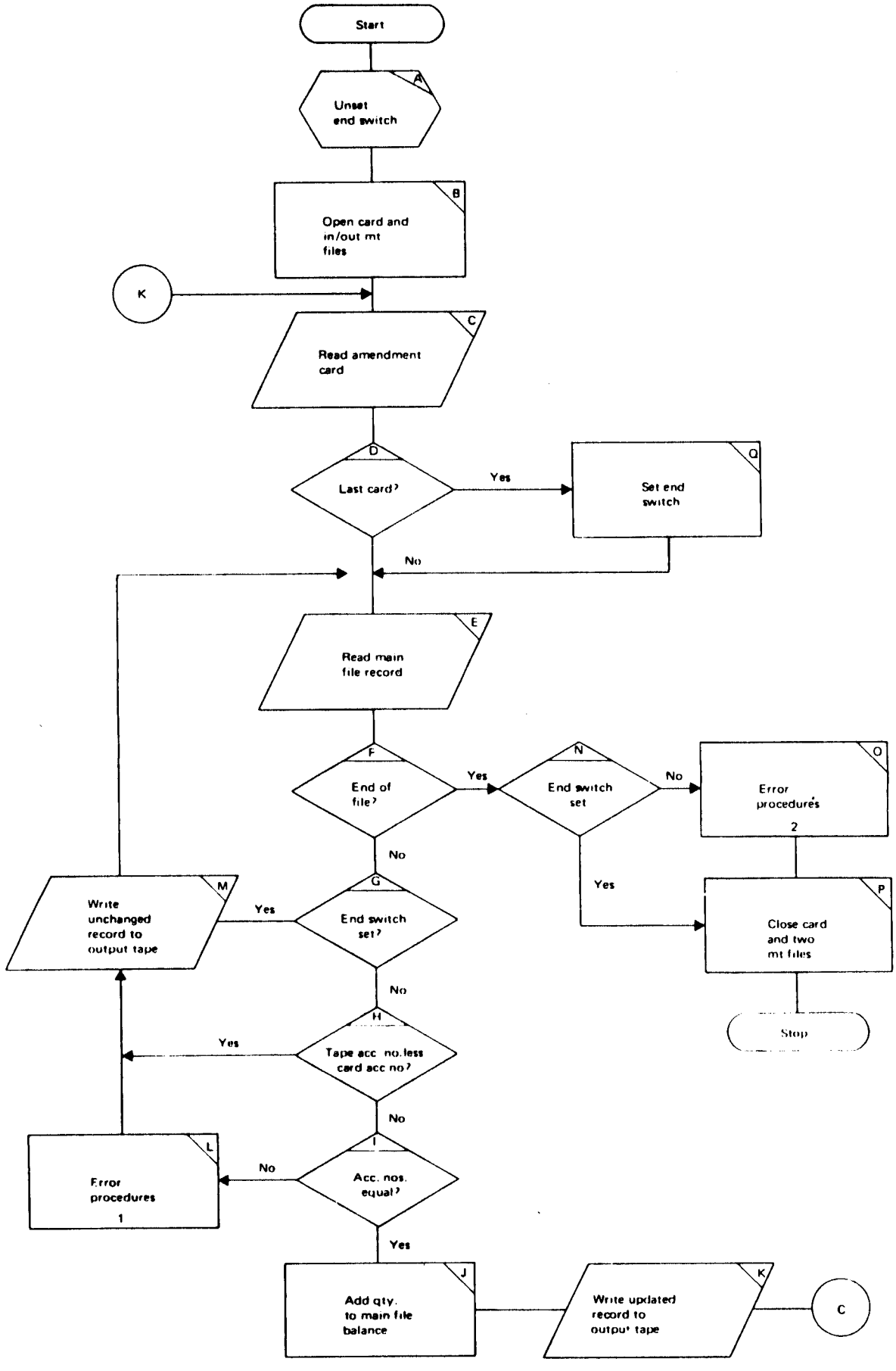


Figure 1 Example flowchart

2 To check that the card file has already ended when the magnetic tape file ends. If it has not, an error condition has occurred and box O handles it.

Box B makes a card reader available, finds a spare output tape and labels it, and finds the old main file tape.

Boxes C to F obtain a record from each of the input files and check each one to see if the end of that file has been reached. The sequence, READ-END? is the same as the COBOL READ verb.

According to the relative state of the two account number fields and the end switch, appropriate processing occurs.

If the tape account number is greater than the card account number, that is the answer box I is NO, then there is no main file record corresponding to the card account number. This error is handled by box L.

At the end box P releases the card reader, writes an end label on the output tape and rewinds it. The input tape will be rewound automatically when the end label is read.

DECISION TABLES

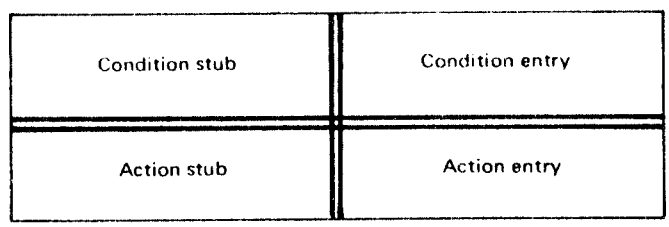
Decision tables can be of value to the programmer before he produces a flowchart, to help him plan its layout, and also after he has written a flowchart to enable him to check that every case has been catered for. It has been argued that a decision table approach rather than a flowchart should be used to formulate a solution before coding a program.

The format of a decision table

A decision table consists of four sections:

The condition stub (or condition statement).
The action stub (or action statement).
The condition entry.
The action entry.

These four sections are arranged in the table as follows:



It is conventional, as shown above, to divide the table into four quadrants, divided by double vertical and horizontal lines.

The condition stub lists all the variables in a problem and the condition entry is split into columns, one column for each combination of conditions (or rules). The action stub lists the actions that may be taken, and the action entry is split into columns which line up with the columns of the condition entry and show the actions to be taken for a given set of conditions.

Consider the following example:

In a credit control application it is decided to approve an order if credit is O.K. or if pay experience is favourable or if special clearance has been obtained. Otherwise the order is not approved.

The decision table for this problem is as follows:

ORDER APPROVAL	RULE 1	RULE 2	RULE 3	RULE 4
Credit O.K.?	Y	N	N	N
Pay experience favourable?	–	Y	N	N
Special clearance obtained?	–	–	Y	N
Approve order	X	X	X	–
Do not approve order	–	–	–	X

Notice that not all possible combinations are shown in this table, because if one condition holds others are irrelevant. When using a decision table to check a flowchart it is advisable to produce a table with columns for each and every combination of conditions and then to eliminate the impossible or irrelevant entries and produce a minimum size table. If this is done, it is advisable to add another condition entry column headed ELSE and specify the action to be taken if an impossible combination does occur. An ELSE column may also be used to mean 'If none of these conditions occur do this'.

The maximum number of columns in a table is 2^n where n is the number of conditions. To keep tables to a reasonable size for complex cases, the conditions in any one table should be limited to three or four and lead via the action entry to a further table which can quote further combinations of conditions and actions.

Example

A Punter wishes to select one horse to back out of three possibles A, B and C. A can be expected to win if the race is less than 1½ miles and if he won last time out. B is likely to win a longer race but only then if the going is soft. C has a chance if the going is hard and the wind is in the West. If he decides that no horse has a chance, he will buy a beer instead.

A full decision table for the logic used by the Punter is shown below.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Race under 1½ miles	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
A won last time	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
Going soft	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Wind in the west	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Back A	X	X	X	X												
Back B									X	X			X	X		
Back C							X				X				X	
Buy Beer					X	X		X				X				X

This table can be reduced to four columns plus an ELSE column since if some conditions hold, others are irrelevant. To be exact, rules 1, 2, 3 and 4 can be expressed in one column, rules 9, 10, 13 and 14 in another, rules 7, 11 and 15 in two columns and the rest covered by ELSE.

	1	2	3	4	ELSE
Race under 1½ miles	Y	N	N	–	
A won last time	Y	–	–	N	
Going soft	–	Y	N	N	
Wind in the west	–	–	Y	Y	
Back A	X				
Back B		X			
Back C			X	X	
Buy Beer					X

The flowchart for this problem is as shown in Figure 2.

COBOL pre-processor

ICL has produced a piece of software called a decision table *pre-processor* which will accept from most media either a free-standing decision table or one embedded in a COBOL source program, and output a program in a format suitable as output to a COBOL compiler.

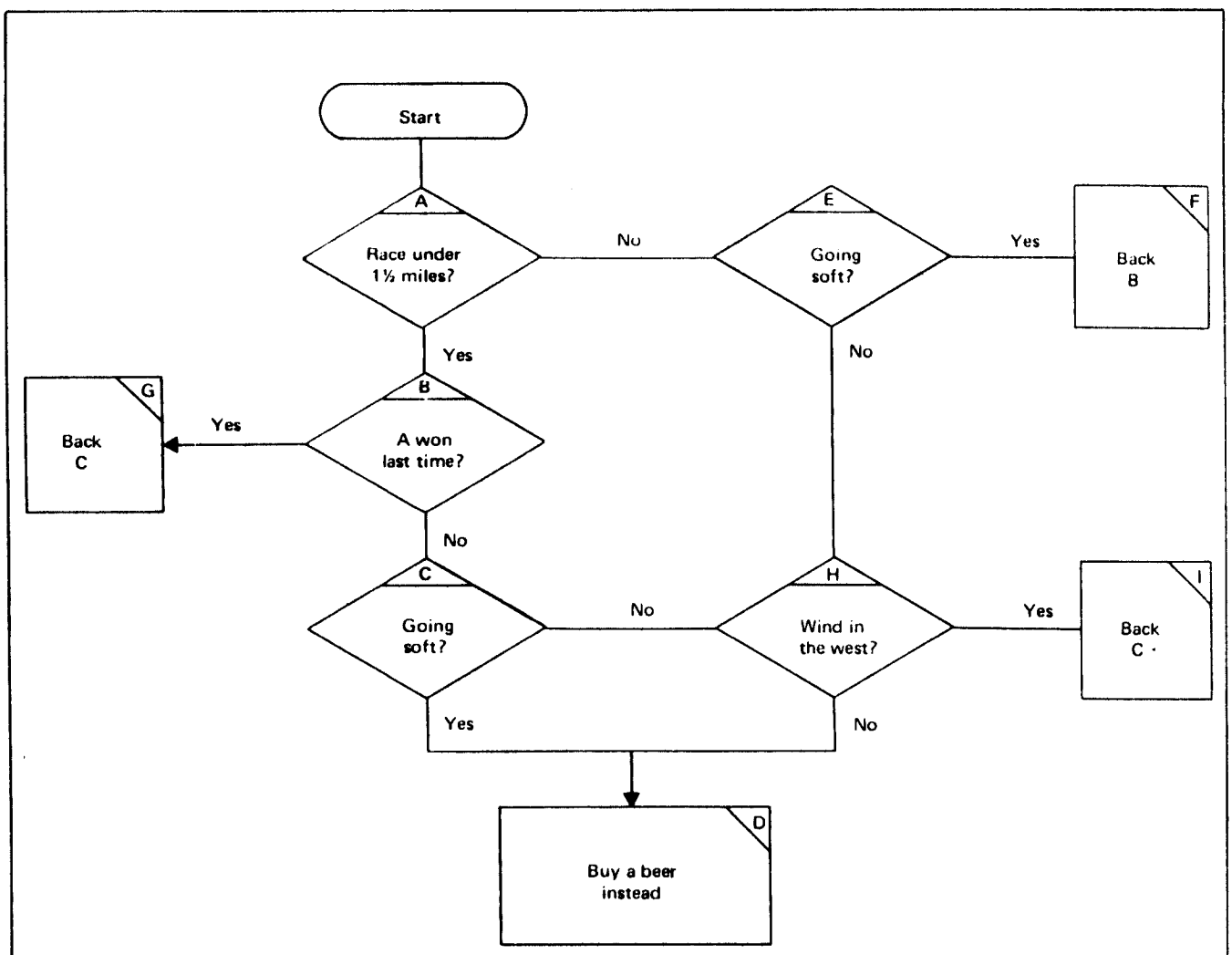


Figure 2 Punter flowchart

Chapter 4 Introduction to COBOL

CODASYL AND COBOL

CODASYL (Conference on Data Systems Languages) and COBOL (Common Business Oriented Language) were conceived at a meeting held in 1959. At that time the following general objective was stated:

> "The current experience of users of electronic data processing equipment indicates that a major problem in the efficient utilization of such equipment lies in the inability to state the data processing application in such a way that computer programs are developed and maintained with a minimum of time and programming effort.
>
> "COmmon Business Oriented Language, independent of any make or model of computer, open ended, and stated in both an English notation and a narrative form, would do much to solve or to ameliorate this problem. Such a language would also simplify and speed up the solution of the related problem of training personnel in the design of data processing systems and in the development of computer programs for such systems.
>
> "In general, the development of a common language would serve to benefit the user in the following situations.
>
> "In developing data processing systems for existing computers, it is important that these systems be capable of processing on future, more powerful computers of any manufacturer with a minimum of conversion costs.
>
> "Full documentation of present systems in a form conducive to making changes and additions with minimum expenditure of time and cost is necessary in order to meet effectively the rapidly changing and expanding requirements of management.
>
> "The need to produce a large number of computer programs in a short period of time often requires the augmentation of programming staffs by the addition of relatively inexperienced programmers."

On May 28 and 29, 1959, a meeting was held in the Pentagon in order to consider whether it would be desirable and feasible to establish a common programming language for commercial applications. Representatives from users both in private industry and in the United States government, computer manufacturers and other interested parties were present.

The result of the meeting was an agreement to go ahead with the project and to establish three committees, short range, intermediate range and long range, to pursue the matter. The short range committee

was given the task of developing an immediate language and was instructed to take the best of three existing language-compiler systems, FLOWMATIC, AIMACO and Commercial Translator and to produce a language superior to any of these.

By September 1959 the short range committee had specified a language which they considered superior to existing language compiler systems. This language specification was further modified and by December 1959 COBOL existed as a language which was not identified with any manufacturer and which presented advantages for both government and private industry users. COBOL as it then existed was specified in a report published by the United States Government Printing Office and has since become known as COBOL-60.

The second official version of COBOL, COBOL-61, was published in 1961. It was not completely compatible with COBOL-60 but it is the version which was generally impiemented and it forms the basis for most COBOL compilers in use today. Other versions of COBOL since then have consisted basically of COBOL-61 with various additional facilities and modifications.

The effort to define a COBOL standard began in January 1963 and currently involves the United States of America Standards Institute (USASI), the European Computer Manufacturers Association (ECMA) and the International Organization for Standards (ISO). The USASI issues the COBOL Information Bulletin at regular intervals and has just produced a new COBOL Standard.

THE PHILOSOPHY OF COBOL

The task of specifying COBOL was undertaken because of the computer user's need for a problem-oriented, machine-independent language for commercial applications. To be successful such a language should have certain characteristics. Programs written in the language should be capable of easy conversion from one machine to another; the language should be easily understandable in format and notation; it should provide good program documentation; and it should be capable of specifying data processing problems in such a way that an implementation of the language by means of a compiler can produce efficient object code.

COBOL is a language for programmers. It is not intended for use by people unfamiliar with computers. The attributes of the language make it more feasible for use by applications-oriented personnel and make it easier to teach to novices, but an understanding of programming and data processing concepts is a pre-requisite to the writing of efficient COBOL programs.

The principal benefit of COBOL is the ability to write programs quickly and efficiently. Since COBOL has a vocabulary and syntax which is close to English, its programs are rather easier to write than other programming languages. In view of this, the speed of writing and testing can be increased by up to 40%. As regards efficiency, COBOL programs should not use significantly more core or take much longer to run than the same program written in a low level language. In fact, a skilfully written COBOL version is likely to require about 10% more core space than a PLAN version using conventional software, and may take slightly longer to run. However, for the commercial user this is outweighed by the increased speed of writing and testing and by other major benefits which improve the overall efficiency of processing; these occur in the fields of documentation, comprehensibility and compatibility.

The completeness of program documentation produced by a COBOL compilation is dependent upon the particular compiler and upon the amount of thought and effort expended by the programmer. Because

of the narrative nature and Englishlike syntax of the language, however, a certain amount of documentation is inherent. If the user establishes programmer guidelines so that data names and procedure names are consistent and meaningful, the listings produced by a compilation, together with the source program, will constitute a good basis for documentation. Generally, the quality and quantity of documentation produced by the use of COBOL is superior to that of other languages. This superiority is very important for the maintenance and revision of production programs.

The organization of the language, both as to data hierarchies and procedure statements, results in COBOL being much easier to teach and understand than machine-oriented languages or assembly systems. This attribute makes COBOL very useful for man-to-man as well as man-to-machine communications. However, it is necessary to keep in mind the importance of comprehensive training, including basic computer logic and programming fundamentals as well as COBOL training itself.

While COBOL is not a completely machine-independent language, it closely approaches this goal. Any computer manufacturer can take the COBOL specification, and write a compiler program to translate programs written to this specification into the machine code of his particular machine. Any user, therefore, who has access to a COBOL compiler can use COBOL source programs. However, some degree of care is necessary to produce highly compatible programs.

The changes which need to be made to a source program compiled on a different machine from that on which it was written vary considerably. They depend upon such factors as store layouts and above all, the features implemented by the compilers. The manufacturer may select those features of the COBOL language which he considers most useful, to produce a compiler. The features included in compilers may vary, and this may call for re-coding of programs written for a different machine. However, re-writing will in any case be far less than complete rewriting at assembly language level.

The degree of machine-independence in the different parts of a COBOL program is considered later in this chapter.

ICL Data processing

COBOL program sheet

title
programmer

sheet number
date

Sequence No. | Identification

1 6 7 8 11 12 15 20 25 30 35 40 45 50 55 60 65 70 72 73 75 80

FORM C14/41(7.68)

Figure 3 A COBOL program sheet

THE COBOL PROGRAM SHEET

COBOL programs are written on standard program sheets, which are designed to accommodate the different elements of the program. It is of first importance to become familiar with the layout of a program sheet, and a sample is given in Figure 3. As shown, headings including title, programmer, sheet number and date are printed at the top of the page. These details will have no effect on the actual program but the programmer may enter them on the sheet if he wishes.

First of all, it may be seen that the program area on the sheet is divided vertically into 80 columns, and horizontally into a number of lines, so that each line of the program may be punched into an 80-column card or a block on paper tape. Each of the 80 columns in each line is used for the insertion of a separate character or symbol. Alphanumeric characters are always entered in printed capitals.

The program sheet is further divided vertically into five distinct areas, which have the following contents:

Columns	
1 to 6	Sequence number
7	Continuation indicator
8 to 11	Headings
12 to 72	Program sentences or statements
73 to 80	Identification columns

The *sequence number* in columns 1 to 6 is optional. It contains a number given to the corresponding program line, which must be part of an ascending sequence though not necessarily consecutive. If used, the numbers should advance in steps of 100 to leave room for the insertion of other entries, should this prove necessary. If a sequence number is punched in the first line of program (on cards or paper tape) the compiler checks that each subsequent entry is in ascending order. If there is no sequence number against the first entry, the compiler inserts numbers in steps of 100 and ignores any numbers against subsequent entries.

The *continuation indicator* in column 7 takes the form of a hyphen. It is used to indicate that a word or literal appearing first on the line is continued from the previous line. Continuation lines normally begin in column 12, and no spaces may occur at the right-hand end of the previous line. Unless there is a continuation indicator beside a line, the end of the previous line is treated as the end of a word. Column 7 is also used for comment, see Chapter 5.

Columns 8 to 72 are occupied by the COBOL source statements which must be written according to the rules given in the following chapters. *Headings* (Section and Division headings, file and paragraph names) and one or two other entries must begin in column 8. COBOL *sentences* or *statements*, the main body of the program, begin in column 12. These define data to be processed, and specify action to be taken.

The *identification columns*, columns 73 to 80, are again of most value on cards, and are even then optional. They are used primarily as a means of identifying the source program on cards, by writing the same characters on each line in the same positions. Subsequently, the compiler checks for these characters on each card, to ensure that all cards belong in the pack. In this way it is possible to detect stray cards from other programs. If the first card of the source program is left blank in columns 73 to 80, no check is made, so that these positions can be used for comments. In both cases the contents of the columns are included on the source program printout, but do not affect the object program.

STRUCTURE OF THE PROGRAM

A COBOL source program is written in four parts, each of which serves a specific purpose and must always be present in the order stated. These parts of the program are known as *divisions*, and are briefly defined below. At this point it is also stated to what extent each division is machine-independent.

The *Identification Division* is usually the shortest division, and merely identifies the program. It supplies an identifying name which can then be used to refer to the compiled version, and related details if required.

This division is totally machine-independent. However, since it does not produce object code, this independence is not of great consequence to the user.

The *Environment Division* describes the hardware used by the program, in other words, the actual data processing equipment. It specifies the computer used to compile the source program and execute the object program, as well as the peripheral devices needed. It also sets an upper limit to the size of the program in that it prints an error message if the figure is exceeded.

Since the contents of this division depend entirely on the equipment used, it is completely machine-dependent. It must therefore be amended each time the computer assigned to the source or object program is changed, but as the Environment Division is not extensive, rewriting should not be great.

The *Data Division* describes the format of all the data used during processing. For each input and output file, it indicates the exact nature of each record, and of each field or group of fields within the record, defining the number of characters; whether an item is numeric, alphabetic or alphanumeric; where a decimal point is located; how an item is edited and similar information. In short, the Data Division contains a precise description of all the data input to or output from the object program. It may also be used to define working areas which are used by the program for processing.

This division is compatible between machines in so far as the data described is compatible. For example, changes will be required in the Data Division if the user is converting from a character machine to a word machine.

The *Procedure Division* contains the actual program sequence, the operations performed upon data described in the Data Division. The interaction of these two most important divisions, involving a logical sequence of actions and calculations, produces the results required. Instructions in the Procedure Division are expressed in meaningful English words, statements, sentences and paragraphs, which are translated by the compiler into machine code program steps.

This division is almost entirely independent of the type of computer used, but it does depend on other divisions in the source program, since it refers to information contained in them. However, certain changes may need to be made in so far as the language is dependent upon the hardware; if, for example, the collating sequence of a new machine is different from that of the machine currently in use, certain statements in the Procedure Division will have to be rewritten.

EXAMPLE PROGRAM

In order to familiarise the beginner with the nature of a COBOL program, and to clarify the functions of the four divisions, a simple example program is examined in this section. This program is shown in full, in Figure 4. At this stage, it is sufficient to understand the basic structure of a COBOL program, but a knowledge of individual operations is unnecessary. Each division is fully dealt with in a later chapter.

The example program involves a pack of cards, each of which contains the following data:

Columns	
1—3	Numeric reference number (maximum size 999)
4—5	A (maximum size 99)
7—8	B (maximum size 99)

The last card of the pack is punched with four asterisks in columns 1—4. It is required to read the cards, and to write a record to magnetic tape for each card, consisting of the following fields:

Numeric reference number
C (equal to A + B, maximum size 999)

The output magnetic tape is to have a file name of JOB HOURS in the header label, and a retention period of 21 days. There are to be 50 records per block.

The program is described division by division, with reference to the sequence numbers in columns 1 to 6.

Identification Division

100—200 These entries give the division heading and then assign a name to the program. The name given is EXAM69. The first four characters identify the program during the run, and the last two comprise a priority number if the program is run on a multiprogramming machine.

Environment Division

This division has two sections, the *Configuration Section* which deals with the source and object computers used, and the *Input-Output Section* which selects peripheral devices for input and output and relates them to specific files of data.

300 Gives the division heading.

400 Heads the Configuration Section.

500—600 These lines specify the source computer used for compilation of the program, and the object computer used for running it; here, as in most cases, the two are the same, ICL 1904s.

700 Specifies a maximum size for the final program, in words. This entry is of value for documentation. An error message is printed out if the size is exceeded.

800 Heads the Input-Output Section.

900 Gives the heading FILE-CONTROL to the paragraph which selects files to hold data and allocates appropriate peripherals to them.

1000—1100 These lines name an input file, called CDIN, and an output file, called MTΘ; the names are chosen by the programmer and are used for subsequent reference in the program. CDIN holds the data in the pack of cards on a card reader which is associated with it; this is allocated in line 1000. The RESERVE 1 clause arranges for the file CDIN to be read using double buffering which will speed up reading considerably. MTΘ is associated with a magnetic tape deck, and is used to hold the data which is written to it, after being read from the cards and processed.

ICL

COBOL program sheet

title COBOL EXAMPLE

sheet number 1

programmer R. F JUKES

date

Sequence No. 1 6 7 8 11 12 15 20 25 30 35 40 45 50 55 60 65 70 72 73 75 80 Identification

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID. EXAM69.

000300 ENVIRONMENT DIVISION.
000400 CONFIGURATION SECTION.
000600 SOURCE-COMPUTER. ICL-1904.
000600 OBJECT-COMPUTER. ICL-1904.
000700     MEMORY 4000 WORDS.
000800 INPUT-OUTPUT SECTION.
000900 FILE-CONTROL.
001000     SELECT CDIN ASSIGN CARD-READER 1 RESERVE 1.
001100     SELECT MTO ASSIGN TAPES 1.

001200 DATA DIVISION.
001300 FILE SECTION.
001400 FD  CDIN.
001500 01  IN-DATA.
001600     02 REF-IN       PIC A99.
001700     02 FACTOR-A     PIC A99.
001800     02 FILLER       PIC X.
001900     02 FACTOR-B     PIC 99.
```

STOCK CODE 8053F FORM 14/41 (12.73)

ICL

COBOL program sheet

title COBOL EXAMPLE   sheet number 2

programmer R. F. JUKES   date

```
Sequence No.                                                        Identification
1      6 7 8   11 12                                            72 73         80
002000 FD  MTO
002100     RECORDING MODE F
002200     BLOCK 60 RECORDS
002300     LABEL RECORDS STANDARD WITH GENERATION-NO
002400     VALUE OF ID IS "JOB HOURS"
002500     ACTIVE-TIME IS 2'
002600 01  OUT-DATA.
002700     02 REF-OUT       PIC 999.
002800     02 FACTOR-C      PIC 999.
002900 WORKING-STORAGE SECTION.
003000 01  WORK-AREA        PIC 999.

003100 PROCEDURE DIVISION.
003200 PARA-ONE.
003300     OPEN INPUT CDIN OUTPUT MTO.
003400 CALC.
003500     READ CDIN AT END GO TO LAST-PARA.
003600     MOVE REF-IN TO REF-OUT.
003700     MOVE FACTOR-A TO WORK-AREA.
003800     ADD FACTOR-B WORK-AREA GIVING FACTOR-C.
003900     WRITE OUT-DATA.
004000     GO TO CALC.
```

STOCK CODE 8053F FORM 14/41 (12.73)

ICL

COBOL program sheet

title COBOL EXAMPLES

sheet number 3

programmer R. F. JUKES

date

Sequence No. 1–6 | 7 | 8 | 11 12 | 15 … 72 | 73–80 Identification

```
004100 LAST-PARA.
004200     CLOSE CDIN MTO.
004300     STOP RUN.
```

Data Division

This division again has two sections, the *File Section*, which gives full information about each input and output file, and the *Working-Storage Section*, which defines working areas in the computer store, for processing of input data.

As explained in Chapter 1, input and output data is held on files, which are divided into records, and fields within records. The Data Division is laid out in terms of files, as assigned in the Environment Division, and its purpose is to define the details of each file. This involves describing each type of record within the file (records are grouped according to type; for example, in a file consisting of issues and receipts, there would be two types of record, an issues record and a receipts record). In the case of the example program, there is only one type of record for each file. Within each record, the Data Division defines each field, which can be further subdivided into sub-fields. A field which is itself subdivided is called a *group field*, and a field which is not subdivided is called an *elementary field*. A field can be only one character long, for example, a sales area number, or many characters long, for example, an employee's name. The programmer must produce an exact definition of each item of data at its lowest level, and this is essential if the Procedure Division statements are to be correctly translated by the compiler.

1200	Heads the Data Division.
1300—2800	These entries comprise the File Section of the division. 1300 is the section heading. For each file there is a File Description entry, which gives the name of the file, and, in the case of disc and magnetic tape files other information. After this, one record description is given for each type of record in the file. This gives a name to the record, and names each of the fields and sub-fields within the record. Each field is described by what is called a PICTURE clause (abbreviated to PIC) which is written following the field and which gives a shorthand description of its size and nature. The hierarchy of the record is denoted by *level numbers*; record names are given the number 01, fields 02, subfields 03, and so on, until the lowest elementary field has been defined.
1400	This is the File Description entry of the card file CDIN. It begins with the abbreviation FD, and then names the file.
1500	This gives the name IN-DATA to the record in CDIN and assigns it an 01 level number.
1600—1900	These entries describe all the fields within the record; as they are all primary divisions of the record, they have 02 level numbers. Each field is written on a separate line. Although the contents of a field may vary in actual value, the format of the field must be identical in each record. That is to say, it will always have the same number of characters of the same type. Line 1600 defines a reference number with the name REF-IN. This has three numeric characters specified by PIC 999. Each 9 represents a numeric character. Line 1700 defines a number called FACTOR-A, whose Picture, PIC 99, shows that it has two numeric characters. FACTOR-B is another number with an identical Picture. The name FILLER given in line 1800 does not refer to a normal field, but indicates that there is a space on the file between FACTOR-A and FACTOR-B. PIC X would usually indicate one alphanumeric character, but used with FILLER it signifies that the program is not interested in the contents of this position. It will not be referred to in the Procedure Division.

2000—2800 These lines give a similar file description of the output magnetic tape file and also supply additional information about the tape. 2000 is the File Description entry.

2100 Indicates that all the records in the file have the same fixed length.

2200 Indicates that a block on the tape can hold 50 records (see Chapter 6).

2300 Indicates that the tape contains standard header and trailer labels.

2400 VALUE OF ID gives the name on the header label which is to be written on the tape, in this case "JOB HOURS".

2500 ACTIVE-TIME IS 21 specifies the number of days for which the magnetic tape is to be retained before being used for something else.

2600—2800 The record description for MTΘ is similar in structure to that of CDIN. Only one type of record, OUT-DATA, is to be written to the tape file.
The rest of the record consists of the reference number which was part of CDIN, and a number called FACTOR-C. The reference number must be distinguished from the identical field in the input file, and so it is given a different-name, REF-OUT, though it has the same Picture. FACTOR-C is a quantity with the Picture 999, denoting three numeric characters. This quantity is produced as a result of processing, as will be seen in the Procedure Division.
None of the fields in OUT-DATA are subdivided, and they therefore all have the level number 02.

2900—3000 These lines comprise the Working-Storage Section, which may be used to allocate an area or areas for the accumulation of results during the program, or the storing of values which remain constant. In fact, this section is not always necessary as sometimes calculations can be done in the input or output record areas. 2900 gives the section heading, and 3000 allocates an area called WORK-AREA of three numeric characters, to be set aside for processing. Single fields in the Working-Storage Section may have the special level number of 01. The way in which WORK-AREA is used will be shown in the Procedure Division. In this example the WORK-AREA is not strictly necessary, since the calculation could have been done in the output record area.

Procedure Division

The Procedure Division specifies operations to be performed on the data described in the Data Division. As the Data Division has a hierarchial structure consisting of files, records and fields, so the Procedure Division has a similar structure.

The most elementary unit in the division is a *statement*, which is one operation calling for some kind of action; for example, ADD A to B. A statement always begins with a COBOL *verb* which has the form of an imperative verb in English. Each verb is followed by other words which are the operands; these are sometimes records or fields defined in the Data Division, or they may be literal quantities. A verb and its operands form what is known as a *specific verb format*.

Statements may be combined to make COBOL *sentences*; a sentence may consist of just one or a number of statements terminated by a full stop. In theory, any number of statements can be included in a sentence, but single-sentence statements are recommended, as this leads to clearer layouts, and assists error detection.

In COBOL, a group of sentences can be combined to form a *paragraph*. Sentences will be grouped according to the logical structure of the program, and each paragraph must be given a *paragraph name*. It is often required to jump from one point in the program to an earlier one, as a result of some test, or in order to use the same statements several times. According to the rules of COBOL, it is only possible to branch within a program to the first statement of a paragraph. This therefore determines where one paragraph ends and another begins.

When segmentation is used, as explained in Chapter 13, paragraphs are grouped together in *sections*. However, these will not be considered here.

3100—4300 These lines contain the Procedure Division statements of the program starting with a division heading. There are three paragraphs, each introduced by a paragraph name.

3200—3300 PARA-ONE opens the input and output files, CDIN and MTO. All files must always be opened before processing, and closed afterwards. Opening the files involves allocating to the program the peripheral devices specified in the Environment Division, on which the files are held, labelling procedures are also carried out for magnetic tape files.

3400—4000 CALC is the chief processing paragraph, which deals with input and output records, and performs intermediate calculation. The statement
READ CDIN AT END GO TO LAST-PARA
reads each record in the file consecutively, until all have been read. At the end, the program branches to LAST-PARA. Although the statement instructs the program to read the file, one record is read at a time and the procedures in the paragraph performed. Then the program returns to CALC and reads the next record. As each record is read, the program tests to see if it is the end card, (punched with four asterisks) in which case it goes to LAST-PARA. If the record is not the last card, an output record is built up from it. REF-IN is moved straight to the output area REF-OUT. FACTOR-A is then moved to WORK-AREA which has been set up as a working area. Then FACTOR-B is added to the contents of WORK-AREA (in other words FACTOR-A) to give a third quantity, FACTOR-C, which has been defined under MTΘ. The output record is then written to MTΘ, the magnetic tape file. Note that whereas the file name CDIN was read, the record name OUT-DATA is written. GO TO CALC, finally returns the program to to read the next record.

4100—4300 After all the records are read and the end card is detected, the program branches to a final paragraph, LAST-PARA. The statement CLΘSE CDIN MTΘ closes the files. This involves writing a trailer label and rewinding on magnetic tape, and releasing both the peripherals from the program STOP RUN then stops the program, and END OF RUN appears on the console typewriter.

DATA NAMES

As explained each input and output file, record and field defined in the Data Division is given a name and this is fundamental to programming in COBOL. The name is known as a *data name*, and is a mnemonic name devised by the programmer to describe a variable item of data.

First, a data name must be distinguished from a *data item*. A data item consists of one or more characters, which may be numeric digits, letters of the alphabet, or any symbols allowed on the 1900 Series. A space or blank is also regarded as a character. The item is characterised by its class and length; it may be numeric, alphabetic or alphanumeric, and it may have any number of characters, but once its length has been defined in the program, this length cannot be changed during processing. Each data item must also be given a name to distinguish it in the program, and this is the purpose of the data name. It is therefore important that the name should be unique within the program; another item with the same name would cause confusion.

Whereas a data item consists of an actual value of quantity, a data name is a symbol by which the item is referenced. For example, if an item of data called CODE is to be processed, action can be taken on the contents of CODE without knowing its actual value at the time. The contents of CODE may also be different, say, for each record. The advantage of referring to the data by a name is that general procedures can be set up to handle any data within reasonable limits without its precise nature being known. An item which has the name GROSS-PAY can be referenced for each employee, even though its contents are different every time an employee's wages are calculated.

While data names comprise file, record, field and certain other names, paragraph and section names must also be invented for each paragraph and section within the program. These are known as *procedure names*. They can either follow the rules for data names given below, or they can be a string of up to 30 numeric characters. They must be unique within the program, and must not be the same as any data name or condition name (see Chapter 6).

Rules for data names

A data name must conform to the following rules:

1 It must not be a *reserved word*. Reserved words, a complete list of which is given in Appendix 5, are words which have a precise meaning in the COBOL language and can only be used in that context. However, since only a few reserved words are hyphenated it is suggested that the programmer should use hyphenated data names as far as possible.

2 It must not exceed 30 characters chosen from the following range:
0 to 9
A to Z
– (hyphen)
A data name must contain at least one alphabetic character and must not begin or end with a hyphen. Spaces are not permitted and separate elements *must* be connected by a hyphen.

3 It must normally be unique within the program. (However, if the same name refers to more than one data item, the name must be qualified. See Chapter 14.)

Examples
The following are examples of valid data names:

```
GROSS
LIST–14
9010B
TOTAL–UNITS–ISSUED
```

The following are examples of invalid data names:

TOTAL UNITS ISSUED	(Spaces not permitted: this would not be interpreted as one data name)
–LIST 14	(must not begin with a hyphen)
ACCESS	(COBOL reserved word)
7764–1	(must contain at least one alphabetic character)

COBOL CONVENTIONS

Since the syntax and format of a COBOL program is of the utmost importance, it is essential for CODASYL to define precisely how the language is to be written. Given such a definition, the compiler is able to translate correct statements and detect erroneous ones. A set of conventions which define the format of each COBOL element is therefore in use, and this is also used throughout the manual.

This section lists the conventions in the language specification, and also draws attention to some other rules which should be followed in writing a COBOL program. The reader will gradually realise the importance of the items listed.

1 All words in capitals underlined must be written exactly as shown, and must always be present when the function of which they are a part is used. These are *key words* or *reserved words*, which have a special meaning to the compiler.

For example,
MEMORY integer WORDS

An error message will appear if the underlined words are absent or incorrectly spelt in a statement.

Underlining is not necessary when writing in actual COBOL source program.

2 All words in capitals not underlined are used for readability only, and may be written or not as the programmer wishes. These are known as *optional words*.

For example,
RECORDING MODE IS V

3 All words in lower case letters represent generic terms for material to be supplied by the programmer. These may be identifiers which include data names and procedure names or *literals* which are values directly quoted in a statement (see Chapter 7).

For example,

ADD data–name–1	TO data–name–2
SUBTRACT literal–1	FROM data–name–1

4 When material is enclosed in braces, thus

this is an indication that a choice must be made from the possibilities given.

For example,

ADD { data–name–1 / literal–1 } TO data–name–2

5 When material is enclosed in square brackets [ ], this means that it is an option which may be included or omitted as required.

For example,
MULTIPLY data—name—1 BY data—name—2 [GIVING data—name—3]

Brackets can be nested, and are evaluated by starting at the outer pair and working inwards. If material is enclosed in braces within square brackets, this means that the whole entry is optional, but that if it is used, a choice must be made between the possibilities given.

6 Three full stops, thus, ..., indicate that any number of items may be included in a format as shown.

For example,

$$\text{ADD} \left\{\begin{matrix}\text{data—name—1}\\ \text{literal—1}\end{matrix}\right\} \left\{\begin{matrix}\text{data—name—2}\\ \text{literal—2}\end{matrix}\right\} \ldots \text{TO data—name—n}$$

The following conventions should be followed when writing a COBOL program:

1 The letter O should be distinguished from the digit zero, by writing O as

ø
and zero as
0

2 The letter I should be distinguished from the digit one, by writing I as
I
and one as
1

3 Care should also be taken in writing 2 and Z, 6 and G, which may easily be confused. The use of continental sevens (7̸) and Zs (Z̸) may also avoid ambiguity.

4 The use of full stops is very important in a COBOL program, and they must always be written as shown in a format, and must be followed by at least one space. In general, a full stop must follow all division and section headings, and paragraph names; each file description, and each record description entry; each COBOL sentence.

5 Semi-colons and commas may be written at any point in the source program for the sake of readability. They must always be followed by a space, and are ignored by the compiler.

6 Hyphens must always be included in key-words as shown.

It is important to realise that the rules concerning punctuation and the writing of key-words are essential to the success of a COBOL program. With a high level language, where one source instruction generates many object code instructions, a small mistake may lead to a useless object program. Common errors include missing full stops, missing hyphens, and key-words written wrongly.

Chapter 5 Identification and environment divisions

This chapter deals with the identification and Environment Divisions of a COBOL program. Several of the Environment Division options will be further explained in later chapters. The use of the Environment Division for files held on direct access devices is described in outline in Chapter 11. Further options available in the division, but less commonly used, are given in Chapter 14.

IDENTIFICATION DIVISION

This brief division exists principally to give the program a name to identify it at run-time, and a priority number for multiprogramming purposes. Extra facilities are available for adding documentary comments if required.

The basic format of the division is as follows:

IDENTIFICATION DIVISION.
PROGRAM-ID. Program name.

As with all division names, the heading must be written exactly as shown on one line of coding sheet, beginning in column 8. It must be followed by a full stop. (Note that a space must always follow a full stop.)

PROGRAM-ID entry

The heading must be followed on the next line by the PROGRAM-ID entry. PROGRAM-ID should again start in column 8, and must include the hyphen and full stop as shown. On the same line is written a program name, devised by the programmer according to the rules below:

1 It must have exactly six characters.

2 The first character must be alphabetic, the next three alphanumeric (alphabetic or numeric), and the last two numeric.

3 It must be followed by a full stop.

The first four characters are allocated as the name of the object program at compilation; thus they allow the program to be located and loaded into store at run-time, by means of operator and Executive action.

The last two characters comprise a priority number, which is used if the program is run on a multiprogramming machine. This must be numeric, and in the range 01 to 99. It is essential that a priority is given if compiling a program for use on multiprogramming machines. Should the priority number be omitted, the compilation printout will show message number 800, and a priority of 01 will be given to the program.

Even if multiprogramming is not used, two numeric characters must always be present, although in this case any number will suffice.

Examples
The following are examples of valid program names:
COST71.
X78943.
T3SR02.

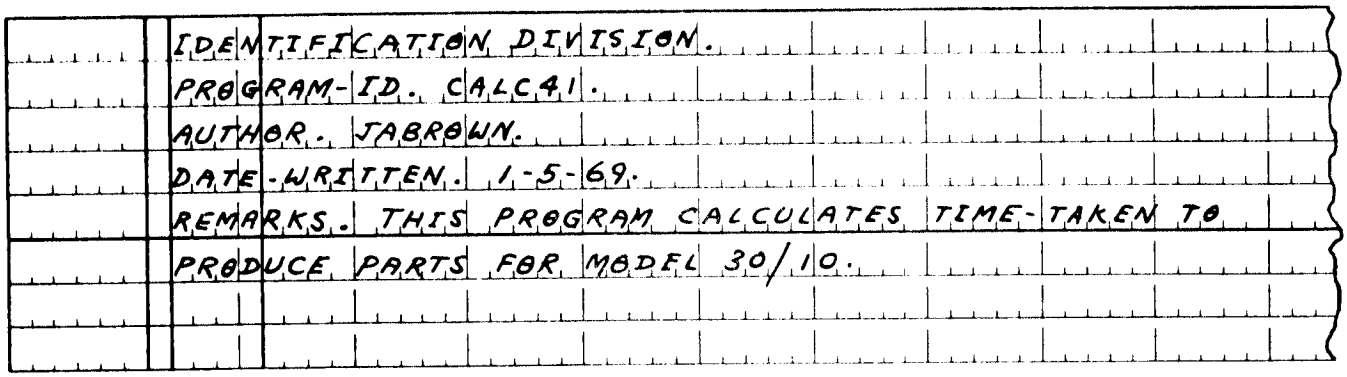
The following are examples of invalid program names:
COST71 (full stop omitted)
345109. (first character not alphabetic)
GAMEPQ. (last two characters not numeric)

Documentation facilities

It is possible to supply information concerning the program. This must appear in the correct format, with no spelling mistakes, after the priority number and before the next division heading.
Some useful comments might be the author of the program, the date the program was written or the name of the installation.
The information will be listed on the compilation listi but have no affect on the object program.

The date of actual compilation is always supplied by the compiler and written at the head of the compilation printout.

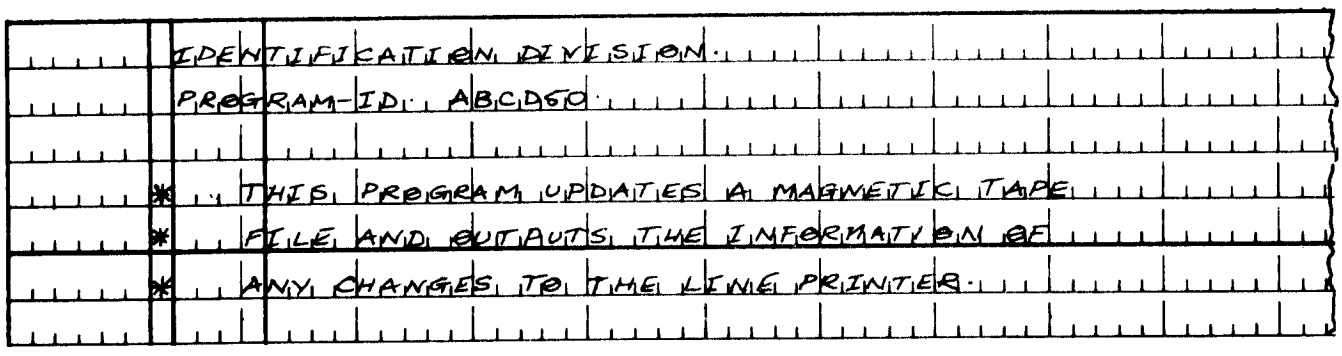
Example
The following example shows a typical Identification Division:



Comment lines

An asterisk in column 7 denotes a comment line. The contents of the line will appear on printed output but are otherwise ignored by the compiler. Continuation indicators cannot be used to describe continuation comment lines. Each comment line must be denoted by an asterisk in column 7. Comment lines can appear in any division after the Identification Division header.

Example



ENVIRONMENT DIVISION

This division provides a standard means of expressing information about the hardware required for a particular program. It always contains two sections: the Configuration Section specifies the details regarding the source and object computers used with the program; the Input/Output Section specifies the names of all files to be used with the object program, and the peripherals to be assigned to the files.

The format of the division is as follows:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
```

Division, section and paragraph headings, as shown above, must be written exactly as shown, each on a new line starting in column 8. Particular care should be exercised in the placing of full stops and hyphens.

Configuration Section

This section may consist of three paragraphs, the Source-Computer Paragraph, the Object-Computer Paragraph and the Special-Names Paragraph. The first two must always be present, but the third is optional.

SOURCE-COMPUTER PARAGRAPH

This paragraph has the format

```
SOURCE-COMPUTER. ICL-190n.
```

Its purpose is to tell the compiler which type of computer is being used to translate the source program. The line must start in column 8 and at least one space must precede the ICL entry. ICL-190n specifies a member of the 1900 Series, where n is a numeric character, as for example, ICL 1903. The A series should not be specified. The ICL entry can alternatively be written on a new line, starting in column 12.

ICL-2903 or ICL-2904 should be specified, if appropriate.

OBJECT-COMPUTER PARAGRAPH

This paragraph can be larger than the previous one and has the format

```
OBJECT-COMPUTER. ICL-190n
    MEMORY integer WORDS
    [SEGMENT-LIMIT IS integer]
    [ASSIGN OBJECT-PROGRAM TO { CARD-READER
                              { PAPER-READER
                              { TAPES
                              { CASSETTES
                              { EDS
                              { CORE-EDS
```

A full stop must be written at the end of the last entry in the paragraph.

The OBJECT-COMPUTER entry tells the compiler which computer is to be used to run the object program. This entry begins in column 8, but all other entries begin in column 12. The rules given under *SOURCE-COMPUTER PARAGRAPH* also apply here, including the fact that the A Series should not be specified.

The compiler examines the last digit of the computer number to see if the machine is a 1904 (or above) or a 1903 (or below). On earlier 1900 machines this was the point at which some functions which were Extracodes on smaller machines were carried out by hardware on the larger processors. Examples included movement of large amounts of data, and multiplication. The compiler can produce a more efficient object code program if it knows whether some functions are performed

by Extracodes or by hardware. On current 1900's some machines below the 1904 have certain of these Extracode functions done by hardware. If the program is to be run on such a machine; it is worth specifying 1904 even if it is, say, a 1901S.

The next statement, the MEMORY clause, is used to state the amount of storage in words available to the object program. The number of words given must be a whole number. It cannot be greater than store size minus the amount taken up by Executive.

Note that the amount of store allocated to the object program may be larger than that used to compile the source program. For example, a program may be compiled on a 16K machine for running on a 32K machine; the MEMORY SIZE would then be about 2700 in the OBJECT—COMPUTER clause.

If the object program requires more store than is stated, a message
* 645 COMPUTER SIZE EXCEEDED
is printed out. However, the program will still compile, and can be run so long as sufficient store is available.

The SEGMENT—LIMIT clause is used only when a program is segmented. Such a program is divided into segments consisting of one or more sections; each section is given a *priority number*, sections in the same segment having the same number. The clause specifies that sections with priority numbers less than the integer specified are held in the permanent area of storage. The integer must be in the range 0 to 49. Sections with priority numbers equal to or greater than that stated are overlaid, that is they are copied into core each time they are called by the program.

If this clause is not specified in the Environment Division, the segment limit is taken as 50; sections with priority numbers below 50 then become permanent sections.

A full description of segmentation is given in Chapter 13.

The ASSIGN OBJECT—PROGRAM clause is again used only for segmented object programs. It specifies to the compiler the type of peripheral on which the object program is to be held and from which it is to be overlaid. A special overlay package is incorporated in the object program.

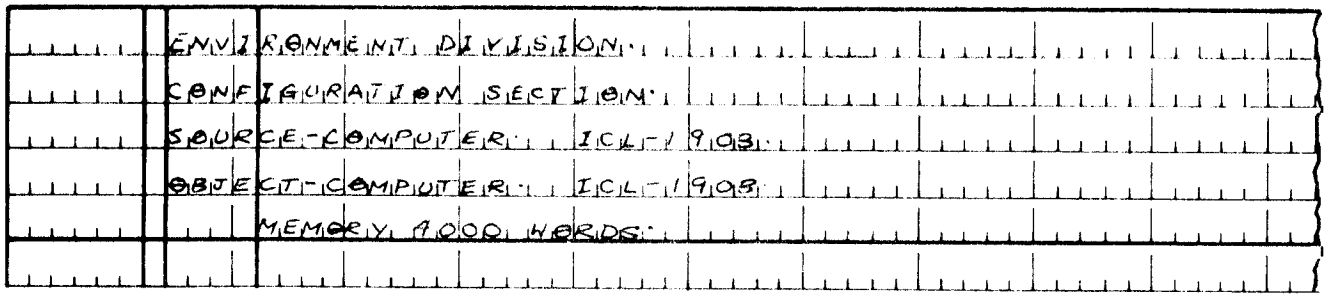
This clause is normally relevant only when the medium from which the program is overlaid is different from the medium on which the compiler is held. If the clause is omitted, the same backing store is assumed; for example, an object program compiled by an E.D.S. compiler will be overlaid from E.D.S. The overlay package used is that appropriate to the medium on which the compiler is held.

The effect produced by the clause depends on the compiler, and on the peripheral specified. CARD—READER and PAPER—READER are for documentation purposes only. TAPES and EDS each produces its own overlay package, and results in the program being held on the medium specified.

CORE—EDS can be specified. On loading the object program, the overlay segments normally held on E.D.S. will be held in store instead (in Upper data). This is a method of running programs which occupy more than 32K in store. These would otherwise require the use of a 1900 feature called 'Extended mode'.

For further details of this clause, see the COBOL reference manual.

Example
A typical portion of the Environment Division, showing Source and Object—Computer paragraphs, is shown below



SPECIAL—NAMES PARAGRAPH

This provides a means of giving mnemonic names to hardware units, so that the name can be used, for example, in ACCEPT and DISPLAY statements, to allow input to the program from the card reader or output to the printer. Names can also be given to channels of the printer control loop, and to switches on the console typewriter. Some detail of this paragraph is given here, but cross-reference should be made to other specified parts of the manual.

The paragraph has the format

[SPECIAL—NAMES.]
  [hardware—name integer [TYPE type—number]
    IS mnemonic—name] ...
  [CHANNEL—n IS mnemonic—name] ...
  [hardware—name { ON STATUS IS condition—name / OFF STATUS IS condition—name } ...].

It consists of a heading which must be written as shown, followed by one or more clauses with the above formats. The three clauses shown correspond to the three options available. Of course, the same clause may occur more than once in a paragraph. The Special—Name entry must begin in column 8, and all other entries must begin in column 12. A full stop must be written at the end of the last entry in the paragraph.

A *mnemonic name* is a name defined by the programmer using the rules for data names (see Chapter 4). It must be unique within the program and in this context is used to identify a hardware device. The relevant device is then referred to by this name in the Procedure Division.

The first clause assigns mnemonic names to peripheral devices, where the hardware name must be one of the following:

CARD—READER
CARD—PUNCH
PRINTER
PAPER—READER
PAPER—PUNCH

The integer written after the hardware name indicates that the peripheral is the *n*th of that type assigned to the program. (This enables the programmer to ensure that the device is or is not the device assigned to a specified file.) The mnemonic name is a pseudonym for the peripheral identified in the same clause. This name refers to the device in the Procedure Division, when it is required to accept into the program a small amount of data held on a peripheral, or to display similar data on an output peripheral.

Data can be accepted from a card reader or paper tape reader, and can be output on a card punch, paper tape punch or the line printer. Details of these operations, and of how the mnemonic name is used are given under *ACCEPT*, and *DISPLAY*, in Chapter 7.

The TYPE option allows a particular model of the peripheral device to be specified by means of the type number. For example, type number 110 denotes a card reader with binary image facility, operating at 900 cards a minute; 001 denotes a line printer with 96 print positions, operating at 300 lines a minute. A complete list of type numbers acceptable to the compiler is given in the COBOL reference manual. If the TYPE option is omitted and the computer has more than one type of the peripheral named, the type is assigned arbitrarily. Further details of this option are given at the end of this chapter.

The second clause assigns mnemonic names to the channels of the paper tape loop used to control line spacing on the printer. See Chapter 1, for a description of the loop. CHANNEL is followed by n which is an integer in the range 1 to 7, specifying a channel number. The mnemonic name is a pseudonym for the channel, and is used with the WRITE verb to control spacing on a printing form. See *WRITE*. This facility may be of value when compiling COBOL programs written for other machines.

In the third clause, the hardware device must be a switch in word 30 of the program area. These switches can be set by operator action on the console typewriter. The program may need to test whether one of these switches is in the ON or OFF state, and branch accordingly. The two states of the specified switch can be given condition names, so that it can be tested using the statement.

IF condition name

The Special—Names paragraph gives a name to the switch in either of the two states. If the ON state is named, the value of the condition is true when the switch is on, and false when it is off. If the OFF state is named, the reverse is true.

The condition name can be used in a switch status condition, wherever a condition is permitted in the Procedure Division. See Chapter 7, "CONDITION NAME CONDITIONS".

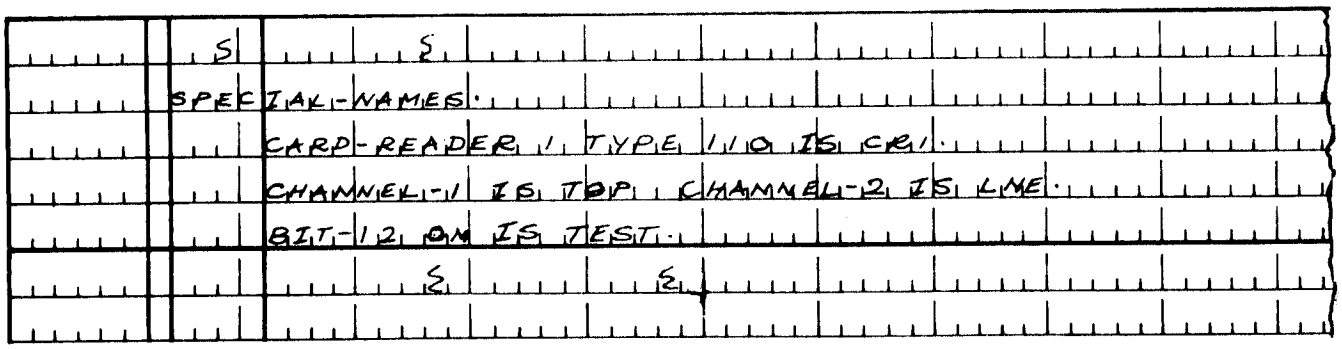
In this clause, the hardware name must have the format

BIT—nn

where nn is in the range 0 to 23. Each bit represents a switch in word 30 of the object program, and a 1 bit value corresponds to ON, while an 0 bit value corresponds to OFF. The condition name follows the rules for data names (see Chapter 4).

Example

The following example shows a Special—Names paragraph including all the possible options.



Input-output Section

The Input—Output Section normally consists of a single paragraph, the File—Control paragraph, which is written immediately after the section heading

INPUT—OUTPUT SECTION.

FILE—CONTROL PARAGRAPH

This paragraph describes the peripheral units used to hold input and output data, and allocates them to files. Each file referred to in the Procedure Division must be named here and associated with a specific peripheral. For example, if a pack of cards is to be read, the paragraph allocates the cards to a card reader.

The paragraph has the following format:

```
FILE—CONTROL.
    SELECT file—name—1
    ASSIGN TO
                                  { [integer—1] TAPES
                                  { TAPES integer—2
                                  { hardware—name integer—3
                                  {   [TYPE type—number]
    [RESERVE 1].
    [SELECT file—name—2 ...].
```

After the paragraph heading, which must be written as shown starting in column 8, a SELECT...ASSIGN statement must be given for each file used by the program, starting in column 12. A full stop must follow each select assign statement.

Each file used must be given a name, the file name is a data name devised by the programmer according to the rules.

Each file name is included in a statement which assigns it to the appropriate peripheral. If the file is to be written to or read from magnetic tape, the TAPES option is used. The preceding integer indicates the number of tape units to be assigned to a file. Within the 1900 2903/4 series it is not necessary to specify more than one tape unit, as any continuation reels needed are automatically allocated. The integer is therefore always 1, and is included only for documentation and compatibility.

Allocation of magnetic tape units to files takes place dynamically in 1900 COBOL; that is, those units which are next available are allocated as they are needed. There is no means of allocating a particular unit to a file.

If the file is on a slow peripheral or on direct access, the hardware name given in the statement must be one of the following:

```
CARD—PUNCH integer
CARD—READER integer
PRINTER integer
PAPER—READER integer
PAPER—PUNCH integer
EDS integer                    (Exchangeable Disc Store)
```

The integer after the hardware name must be a whole number; this must start at 1 for each type of peripheral; it identifies the unit as the first, second (and so on) of the type assigned to a program. When a file is opened in the object program, the allocation is automatic in that it is assigned to the next available peripheral device (although the programmer can override the automatic choice; see below).

The device chosen in this way becomes the program's integer device of this type, where integer is the number given in the SELECT...ASSIGN statement. For example,

SELECT LPØ ASSIGN PRINTER 4

does not mean that the file is assigned to the fourth printer, but merely that it is to be assigned to a printer referred to as 4.

As mentioned above, the programmer can select a specific peripheral to be assigned to a file. The TYPE option enables a particular model of a given peripheral to be used, if there is more than one kind available to the program.

The phrase

TYPE type-number

is written immediately following the hardware name in the appropriate statement. The number after the word TYPE is equivalent to the property code. This indicates that the unit assigned must have at least the properties defined by the property code, even though the unit may in fact be capable of more than the stated action. For example, a minimum speed of peripheral transfer may be required, a card reader with binary image, or a printer with a minimum number of print positions.

The property code for each device consists of nine bits which are interpreted by Executive at run time. The number written by the COBOL programmer is the octal equivalent of this, in other words a three digit number. For example, the statement

SELECT ISSUES ASSIGN CARD—READER 1 TYPE 000

allocates to the file ISSUES a 300 cards per minute card reader without binary image. A full list of type numbers and the devices they represent is given in the COBOL reference manual in Chapter 4.

It should be noted that the TYPE option can only be used for character peripheral devices, and not for magnetic tape or direct access devices.

For efficient programming it is often desirable to create double buffering for certain files. In COBOL this is achieved by simply adding RESERVE 1 at the end of the SELECT...ASSIGN statement. This is all that is necessary to obtain double buffering; RESERVE 1 causes two buffer areas to be assigned to the file. While the data of one buffer area is being processed, data can be transferred to or from the other area, thus increasing overall speed.

The decision to double buffer is dictated by the program. Generally, slow peripherals used throughout a program will be double buffered. There is no advantage in double buffering a fast peripheral if the speed of the program is controlled by input/output of slow peripherals; in addition, the fast peripherals usually have large block sizes and the storage required to double these is uneconomic.

Example

A typical Input-Output Section is given below.

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PTIN ASSIGN PAPER-READER 1
        TYPE 000 RESERVE 1.
    SELECT MAG-OUT ASSIGN TAPES 1.
```

Chapter 6 The data division

As explained in Chapter 4, the Data Division describes the format of all the data used during processing. It was discussed how data items are given data names, and how data is organised into files, records, group fields and elementary fields. In this way all the internal storage available to the program is divided up and names assigned to each area. This is achieved by making an exact description of all external files held on peripherals, presented to or produced by the program. This chapter deals fully with the elements of the Data Division. Further detail of less common features is given in Chapter 14. For direct access devices, additions to the rules given in this chapter are given in Chapter 11.

The purpose of the Data Division is to give the compiler program sufficient information about files, records and fields to allow it to translate the Procedure Division statements correctly. It determines the coding generated by a verb such as ADD, MOVE or READ; this coding will vary according to the nature and size of the fields being acted upon. For example, in the simple statement

```
ADD A-IN TO A-OUT
```

the machine code required to carry out this operation will depend on whether A-IN and A-OUT are numeric, whether they are in character or binary form and so on.

If A-IN and A-OUT are each only one word long, and their contents are in binary notation, then the addition could be performed in two machine code steps. On the other hand, if they are both long fields in character form, and if the decimal points do not line up, then the addition could take thirty steps. Therefore the size and nature of each field to be processed determines the program steps which are inserted in the object program by the compiler.

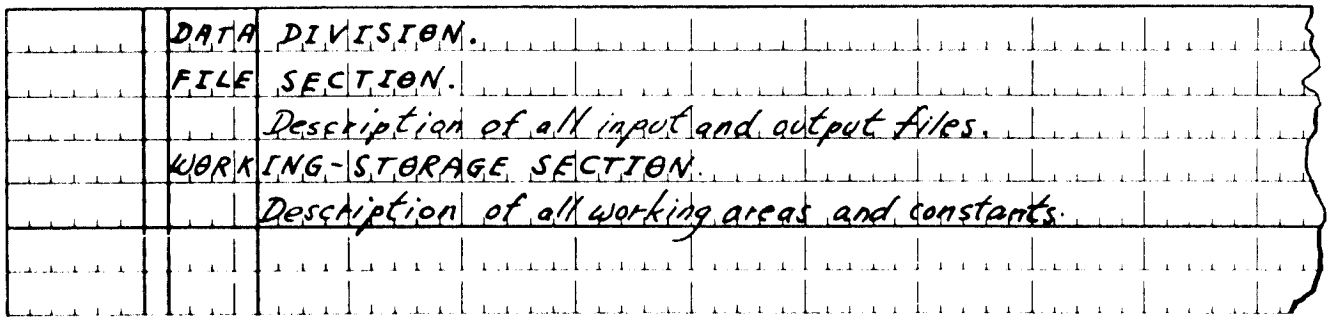
The compiler will build up a table from the Data Division entries showing each data name and the various characteristics of each field. Then, as it compiles the Procedure Division, it will look up this table for each data name, in order to generate the coding to handle the data in the field.

Because of this, it is essential to remember that all names referred to in the Procedure Division must have been defined in the Data Division. The one exception is a name given to an item of hardware in the Special-Names Paragraph of the Environment Division. As a general rule, however, no names should be used which have not had a definition given to them in the Data Division. It is also advisable, particularly on small machines, to keep data names as short as possible, so that the table of entries is kept down to a reasonable size and does not occupy too much store.

STRUCTURE

The Data Division begins with a division heading, and is then divided into two main sections. The *File Section* gives all the detail required by the compiler about input or output files held on peripheral units and handled by the program. It describes each file in terms of the records which it comprises. Every field within a record is defined at its lowest level. The *Working-Storage Section* describes working areas to be set aside for processing, and defines constants, items of data whose values are the same throughout the program. There is also a *Linkage Section* which describes subroutine parameters, but this will not be considered until Chapter 10.

The outline structure of the division is shown below.



RULES FOR WRITING THE DATA DIVISION

In addition to the general rules for writing COBOL there are specific rules for writing the Data Division, as follows:

1 Division and section headings must begin in column 8 and must be written exactly as shown in the diagram above, including hyphens and full stops.

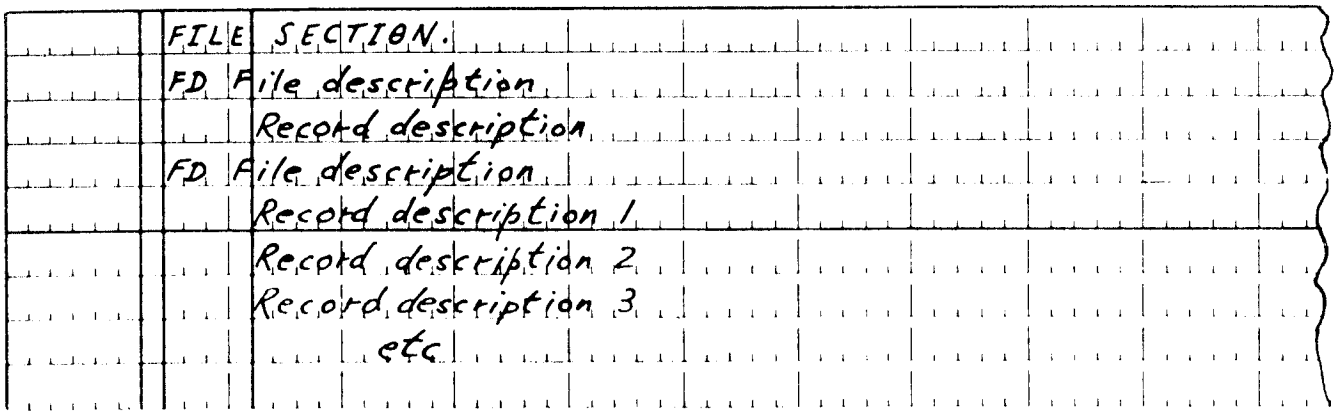
2 The FD of a file description must be written beginning in column 8. The level numbers of record descriptions can begin anywhere after column 7 and can be inset. All other file and record description entries must begin in column 12 or later.

3 A file description must be terminated by a full stop after the last entry.

4 Each record description entry (record name or field description) must begin on a new line and must be terminated by a full stop.

THE FILE SECTION

As stated above, the File Section defines input and output files and records processed by the program. It has one entry for each file, and this consists of a file description entry (FD) giving the name of the file and information about it, followed by one record description for each type of record in the file. If a file contains several record types, this means that there are different structures in which data is organised, which are read or written as the file is handled. The computer must be programmed to deal with each separately, as the varying nature of the records will necessitate different coding by the object program. It should however be noted that there is only one record area per file; thus different record types are merely views of the same area. The order in which files and records within the files are defined is arbitrary, but it is usual to define input files before output files. The structure of the File Section is shown diagrammatically below.



File description

A file description is a COBOL sentence which gives information about a data file and precedes record descriptions for that file. A number of options may be included but there are basically two types of file description.

1 for files held on slow peripherals.

2 for files held on magnetic peripherals.

These two categories will be described separately.

SLOW PERIPHERALS

The format of this file description is as follows:

FD file–name
[DATA RECORDS data–name–1 [data–name–2...]].

FD must begin in column 8, but other clauses must start in column 12 or later.

Since a file description is always in the form of one COBOL sentence, a full stop must follow only the final clause.

FD (File Description)

This clause introduces each file description. FD is followed by a file name devised by the programmer, which will already have appeared in the Environment Division, in a SELECT...ASSIGN clause. Thus a specified peripheral will have been assigned to the file.

The DATA RECORDS clause

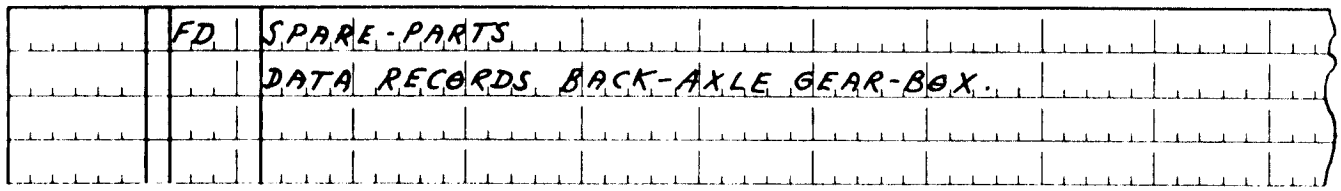
There is another extra clause which may be added as useful documentation, particularly for a file with several record types. It is written as

DATA RECORDS data—name—1 [data—name—2 ...]

Its purpose is to list the names of the records, which must be the same as those given in the record description, and are devised according to the rules for data names. All records in the file share the same area of store, so that the record type has to be determined by test before fields within it can be located.

The DATA RECORDS clause is recommended for all files with more than one record type. The order in which records appear is unimportant. The record names need be separated only by spaces, but note that the omission of the hyphen in a name will cause it to be interpreted as two records.

An example of a file description for a slow peripheral file is given below.



MAGNETIC TAPE

The file description for magnetic peripherals also comprises a COBOL sentence followed by a full stop. It has the following format:

FD file—name

[RECORDING MODE is {F / V}]

[BLOCK CONTAINS integer {RECORDS / CHARACTERS}]

[LABEL RECORDS STANDARD [WITH GENERATION—NO]]

[VALUE OF {ID / IDENTIFICATION} IS literal—1]

[ACTIVE—TIME IS literal—2]

[DATA RECORDS data—name—1 [data—name—2 ...]] .

FD (File Description)

This clause introduces each file description. See above for details.

The RECORDING MODE clause

This entry causes the compiler to allocate an extra word of storage at the beginning of the record area for the file, and the object program, on obeying an input statement, will insert a count word at the beginning of each record and set the appropriate value in it. This count word is not accessible to the user program and is not set to zero after the record has been written. RECORDING MODE F means that all the records in the file have the same fixed length. RECORDING MODE V means either that the records in the file may have different fixed length records or that one or more records may contain an OCCURS ... DEPENDING ON clause.

THE BLOCK CONTAINS clause

This entry indicates the size of a block on a magnetic tape file, and therefore the size of the buffer required in store, as the file is read or written block by block.

A block normally contains a number of records depending on the length of the block and the record size. The block size is normally expressed in terms of the number of records:

BLOCK CONTAINS integer RECORDS

For example:

BLOCK CONTAINS 3 RECORDS

would result in three records being grouped together as shown below:

The BLOCK clause can only be written in a file description whenever a block contains more than one record. The amount of storage allocated by the compiler for the buffer is the number of records specified times the size of the largest record. For example, if a file contained three record types of 20, 30 and 40 words respectively, 360 words would be allocated for the buffer area to handle the file. However, in certain circumstances there may be more than the specified number of records in a block. If the file were an output file and a series of 40-word records occurred in sequence, then nine records would be assembled in the buffer. In this case, there would be nine records in the block written to tape.

The input and output of magnetic tape files in blocks is described under *READ and WRITE*, Chapter 7.

As an alternative to the RECORDS option, the size of the block can be expressed in terms of characters, as:

BLOCK CONTAINS integer CHARACTERS

can be written. This option, however, is normally used only for direct access files to specify the hardware equivalent of the bucket size. This will be either 1 024, 2 048 or 4 096 characters. For further details see Chapter 11.

The LABEL RECORDS clause

As explained in Chapter 1, a magnetic tape file must always have a header label and a trailer label. These labels are special identifying records. For input files they make it possible to check that the correct files have been loaded for the job on hand; for output files, they are created as required (see *OPEN*, Chapter 7). Labels will also be present at the beginning and end of each reel of magnetic tape if a file consists of several reels.

The entry

LABEL RECORDS IS STANDARD

must be included in order that the documentation indicates that the labels conform to ISO/ANSI standards. They can then be read from the tape and checked for an input file, or written to the tape for an output file.

The GENERATION NO entry

This entry is optional and

WITH GENERATION NO

is written, the file's generation number will be checked for input files, or written to the new label for output files.

A normal part of data processing is updating a master file to form a new master file, for example, a stock file which is updated each week or month from sales and supply statistics. The new master file will usually have the same name as the old one, in which case it is necessary that the file's generation number should be qualified. This qualification is provided

by the generation number, which varies from run to run and is supplied by the operator at run-time. It may be given in one of two ways.

If ACCEPT FILE KEYS is used in the Procedure Division (see Chapter 7), and WITH GENERATION-NO specified in the file description, the generation number is read from cards or paper tape when ACCEPT is obeyed.

The message

SET FILE nn GENERATION NO IN WORD 0

is typed out on the console; files are numbered for reference in messages of this kind. The operator must then use the ALTER typewriter directive to alter word 0 of the program so that it contains the value of the generation number specified on the job instruction sheet. He then types GO, the number is extracted from word 0 by the object program, and is used to check or write the tape. There is a halt in the program for each file requiring a generation number, if the operator has to type the number on the console.

The VALUE OF ID clause

This entry must be present for a magnetic tape file when the LABEL RECORDS clause is included in the file description. It is used to check the identification on the header label of an input file, and is written to the new label of an output file.

The clause has the format

VALUE OF {ID / IDENTIFICATION} IS literal—1

where literal—1 must be a non-numeric literal of up to twelve characters. This value is the same as the name on the file's header label, but is not necessarily the same as that chosen by the programmer to refer to the file in the program.
For example,

VALUE OF ID IS "PAY—WEEK "

For an input file, the VALUE OF ID is compared with the actual value of the label identification read into the computer at run-time. For an output file, a header label is written to the tape containing the identification given.

Since the identification is a literal, it must remain the same every time the program is run. Different versions of a file are distinguished by means of the generation number.

The ACTIVE-TIME entry

This entry is needed only for output files, and specifies the number of days for which the magnetic peripheral is to be retained before being used for something else. Its format is

ACTIVE—TIME IS literal—2

where literal—2 is an unsigned numeric literal not greater than 999 (see Chapter 7 literals). For example,

VALUE OF ID IS "CONTROL—3010" ACTIVE—TIME IS 7

The Active—Time is then written to the file's new header label, and will ensure that the file is not overwritten before the time has expired.

If the value of the Active—Time is not specified, an Active—Time of zero is assumed, implying that the file is not needed after the program has been run.

Although it is not usual to specify this entry for input files, its absence will cause the message.

FILE MAY BE LOST ON CLOSING

to appear on the compiler listing. This message can be ignored.

Note: Both VALUE OF ID and ACTIVE—TIME can now specify a data-name instead of a literal. This enables the user to vary the name of a magnetic tape each time the program is run. Data-name must be defined within the Data Division and have a value inserted in it before the file is opened.

Record description

The first record description for a file starts on the line following the file description. As shown in Chapter 4, records consist of group fields and elementary fields. The function of the record description is to give a name to each part of the record and to describe each in turn so that the compiler can, as it were, form a picture of the data in the record. The hierarchy of fields must be seen in relation to *level structure* which is the basis of the record description. The purpose of this system is to show the relative importance of fields within a record.

It must above all be emphasised that there is only one record area for each file. However many records there are within a file, these all occupy the same area of store in turn. Different record descriptions are therefore different views of this one area of store. The record area is allocated according to the size of the longest record, and each record occupies the area in turn as it is processed.

LEVEL STRUCTURE

Level structure is used to show the organisation of group fields and elementary fields within a record. In order to set up the table of data names in store, the compiler must be told what level of importance each name has; whether it is a record, group field or elementary field. The programmer must also be able to see the structure of each record.

This is done by writing a *level number* beside each name, indicating its importance. Record names are given the lowest number 01, and each subdivision in the record has its own level number. The record is normally split into a number of fields which are given the number 02. If a field is subdivided, the names of these subfields have the number 03. If a subfield is further divided, the divisions will have the number 04, and so on, until the lowest elementary field has been defined. Level numbers can go as far as 49.

Suppose that a record called CLOCK—CARD gives the date, a man's name, his employee number and the number of hours he worked. The following fields are present in the record:

DATE
DAY
MONTH
YEAR
NAME
SURNAME
FIRST—INITIAL
MIDDLE—INITIAL
EMPLOYEE—NO
HOURS

This list of names is meaningless to the compiler unless the way it is organised is made evident, as shown below.

CLOCK-CARD							
DATE			NAME			EMPLOYEE-NO	HOURS
DAY	MONTH	YEAR	SURNAME	FIRST-INITIAL	MIDDLE-INITIAL		

It can now be seen that within the record CLOCK—CARD there are four first-level fields. DATE and NAME are group fields in that they each contain three sub-fields. DAY, MONTH and YEAR are part of the group DATE and SURNAME, FIRST—INITIAL and MIDDLE—INITIAL are part of the group NAME. EMPLOYEE—NO and HOURS are both elementary fields as they are not subdivided. Similarly all the sub-fields here are elementary fields. (If they were further divided they would themselves be group fields.)

On the coding sheet the example would therefore be written as follows. It is common but not essential to indent level numbers according to importance, to convey the data structure more clearly to the eye.

```
01  CLOCK-CARD
    02  DATE
        03  DAY
        03  MONTH
        03  YEAR
    02  NAME
        03  SURNAME
        03  FIRST-INITIAL
        03  MIDDLE-INITIAL
    02  EMPLOYEE-NO
    02  HOURS
```

The record must always be defined in this fashion, with all of a group field's subdivisions given in sequence down to the lowest level, before the next group field is specified. A group includes all elementary fields or smaller groups beneath it, until a level number numerically less than or equal to the number of that group is reached. Thus a reference to CLOCK—CARD will involve all the listed items. A reference to DATE will involve DAY, MONTH and YEAR, and so on.

It should [illegible] example is incorrect:

```
01  CLOCK CARD
    03 DATE
       05 DAY
       05 MONTH
       05 YEAR
    02 NAME
       03 S[illegible]
       04 FIRST-[illegible]
       04 MIDD[illegible]
    03 EMPLOYEE-[illegible]
    03 HOURS
```

Th[illegible] same level as the other group [illegible] invalid as it is different from the other [illegible] example, level numbers need not [illegible] they are in ascending sequence and [illegible] record.

The [illegible] and elementary fields is very important [illegible] fields with different level numbers [illegible] different levels. However, [illegible] elements [illegible] in arithmetic and the effect of moving [illegible] depending on whether movement is from [illegible] to elementary.

Control of data structure

A [illegible] whose type and sequence [illegible] the grouping of these characters [illegible] control of the programmer. This means [illegible] record should be determined by the [illegible] the record in the Procedure Division [illegible] should be regarded as a field if it [illegible] Division statements. The [illegible] it is required to handle a part of the [illegible]

The [illegible] organised in several other ways. If [illegible] elementary field separately, but not the [illegible] NAME, the record could have been structured [illegible] Alternatively, only DATE, NAME, EMPLOY[illegible] might have been used and no subfields [illegible]

```
01  CLOCK CARD
    02 DAY
    02 MONTH
    02 YEAR
    02 SURNAME
    02 FIRST-INIT[illegible]
    02 MIDDLE-INI[illegible]
    02 EMPLOYEE-N[illegible]
    02 HOURS
```

The fact that the precise structure of a record is determined by the processing carried out on it may mean that parts of the Data Division cannot be completed until after the Procedure Division has been written. It is wise to leave room on the COBOL sheets to allow the Data Division to be changed.

It may be that the view of the data required by one part of the Procedure Division is quite different from that required by another. In this case two or more record types must be declared, each providing a different breakdown of the record area. This operation is described under *MULTIPLE RECORD TYPES* later in this chapter.

THE PICTURE CLAUSE

In addition to naming each part of the record, and giving it a level number, its size and nature must also be conveyed to the compiler. A record description consists of a series of COBOL sentences, each comprising a data name and a level number, some of which are followed by a clause which enables the compiler to form a logical picture of the entry. This clause is known as the *PICTURE* clause and has the format

{PIC / PICTURE} IS any allowable combination of up to thirty characters and symbols

The abbreviated form, PIC, can conveniently be used in 1900 COBOL. The character string following PIC defines the amount of storage the field will occupy and gives it characteristics. Basically, it consists of a number of characters which give information about the class, size and decimal point position of an item. The number of characters in the string indicates the size of the field, and the type of character written denotes the nature of the data. These characters are considered below. Other editing characters may also be included and these are described later in this chapter.

9

9 represents a numeric character chosen from the numerals 0 to 9. For example,

```
PIC 99999.
```

shows that the field has five numeric characters. Note that there must always be a space between PIC and the next character, and that the full stop is necessary unless there are further clauses which relate to this field.

Alternatively,

```
PIC 9(5).
```

could have been written. The number of characters can be abbreviated (particularly when there are an unwieldly number in the Picture) by writing the number of times the character appears enclosed in brackets after the character itself. There should be no spaces within the bracketed entry.

X

X represents any alphanumeric character chosen from the 1900 Series Internal Character Code given in Appendix 2. For example,

```
PIC XXXXXX.
```

or

```
PIC X(6)
```

shows a six-character alphanumeric field.

A

A represents an alphabetic character chosen from the letters A to Z and space. For example,

```
PIC AAA.
```

or

```
PIC A(3).
```

denotes a three-character alphabetic field. A is less commonly used than 9 and X.

Certain operational characters given below are also used to denote the presence of a decimal point and the signed value of a field. It should be noted that these do not normally affect the number of characters in an item in store by taking up a character position.

V

V indicates the location of an *assumed* decimal point. For example, the Picture below informs the compiler that a five-digit numeric item has two places of decimal:

PIC 999V99.

When the field is involved in arithmetic, the machine code treats the first three characters as whole numbers, and the last two as decimal places.

V can appear as the last character in a Picture but it is unnecessary, since a decimal point is assumed at the right of the Picture when the item is an integer.

S

S must form part of the Picture of any field which may contain a negative value. It is used to indicate that the value is signed and is written in the most significant position. For example,

PIC S9999.

denotes a four-digit numeric field.

In the example, the sign may or may not be held as a separate character, according to the *Usage* of the item (see pages 92 to 93).

It must be emphasised that if a negative value is put into a field which does not have S in its Picture, the sign will be ignored, and the item held as a positive value. S is therefore of the greatest importance in a Picture.

P

P indicates the position of an assumed decimal point when it lies outside the item in storage. It acts as a scaling character, which indicates that the characters held in store are to be adjusted by a power of 10. The number of Ps to the right or left of the Picture show how many positions to right or left the decimal point is located. In this way store can be used more economically. If for example a value is always to be an exact number of hundreds, there is no need to hold the zeros in store with a Picture of

PIC 9999.

Instead the field can be held as

PIC 99PP.

which takes up only two character positions in store. Any value in this field will always be multiplied by 100 before being used in arithmetic.

P is of limited value from an economic point of view since the storage saved in holding the factor will be lost later in the extra steps needed to adjust the factor before arithmetic.

Examples

Some examples of PICTURE clauses using the characters described are given below. It is important to remember that while the number of characters and symbols in a Picture is limited to 30, the clause must not be used to describe an item whose size is more than 120 characters.

Picture	*Number of characters and symbols in Picture*	*Size of item*	*Class*	*Characters in storage*	*Interpreted by the compiler*
AAAAAA	6	6	Alphabetic	MNOPQR	MNOPQR
A(6)	4	6	Alphabetic	MNOPQR	MNOPQR
XXXXX	5	5	Alphanumeric	AB284	AB284
X(5)	4	5	Alphanumeric	AB284	AB284
999	3	3	Numeric	284	284
9(3)	4	3	Numeric	284	284
S999	4	3	Numeric	473 (plus)	473
S999	4	3	Numeric	473 (minus)	-473
99V999	6	5	Numeric	28447	28.447
V999	4	3	Numeric	284	.284
99P	3	2	Numeric	71	710

Not all clauses in a record description require PICTURE clauses. Only elementary fields should be defined in a Picture, group fields which are subdivided (including the 01 record name) do not have Pictures attached to them. Thus the items in a record are defined by Pictures at their lowest level. Every field which is not subdivided must have its Picture, even an 01 level if this has no fields within it.

This rule is illustrated by the record description given below. Note that each clause, whether or not it contains a Picture, must be followed by a full stop.

```
01 CLOCK-CARD.
   02 DATE.
      03 DAY PIC 99.
      03 MONTH PIC 99.
      03 YEAR PIC 9999.
   02 NAME.
      03 SURNAME PIC A(24).
      03 FIRST-INITIAL PIC A.
      03 MIDDLE-INITIAL PIC A.
   02 EMPLOYEE-NO PIC XXXX.
   02 HOURS PIC 99V9.
```

All the rules for using the PICTURE clause are summarised later.

As described earlier, each record or field is assigned a level number which is followed by a data name. Sometimes, however, what is known as a *Filler* can be used instead of a data name. This is a special field name written as FILLER, which indicates to the compiler that a field will not be addressed in the Procedure Division. It is important to remember that FILLER is not a data name, and cannot appear in the Procedure Division.

FILLER

FILLER is used to denote fields which are to be ignored or are to be left blank. On input files, fields defined by FILLER will be ignored (perhaps because they contain information not required by the current program). On output files, Filler fields are also ignored but there are two particular cases:

1 On the card punch, FILLER will be used for all card columns which are to be blank, to pad out the record size to exactly 80 characters. Each group of blank columns will be indicated by FILLER in the record description.

2 On the printer, the number of print positions left blank between fields will be shown by FILLER. The purpose of this is to provide spacing and to pad out the record size to 120 characters or the size of the print line. (Alternatively, on buffered printers, the record size need only be the length of the actual data.)

In both these cases, there must be Procedure Division statements to move spaces into these positions before the record is written. Since the Filler fields cannot be addressed directly, spaces are usually moved to group fields in the record or to the record name itself. The correct number of spaces as specified by FILLER will then be inserted in the record.

The following example shows a record ITEM—TOTAL which is to be output on a line printer. Data is only to be printed in certain positions, and the rest are occupied by spaces. Note that the name FILLER must be followed by a PICTURE clause, indicating the number of characters left blank. By convention X is used to indicate a character position. No editing symbols will of course be included.

```
01  ITEM-TOTAL.
    02  FILLER  PIC  XXX.
    02  CODE-OUT  PIC  X(5).
    02  FILLER  PIC  XXX.
    02  QTY-OUT  PIC  99V9.
    02  FILLER  PIC  X(106).
```

MULTIPLE RECORD TYPES

As may be seen, FILLER has the same level number as the items with which it is grouped in the record.

Record descriptions in a file are merely different views of the same area of store, since there is only one record area. Sometimes these different views are known as a *remapping* of the record area. This section considers the concept in a little more detail.

The following example from a Data Division shows parts of two record descriptions in a slow peripheral file called CDIN.

```
FD  CDIN.
01  FIRST-REC.
    02 FIELD-1 PIC 99.
    02 FIELD-2 PIC X(3).
     }
     }
01  SEC-REC.
    02 A-FIELD PIC 999.
    02 B-FIELD PIC 999.
```

The first record gives an address, FIELD—1, to the first two characters of the record area, which are regarded as numeric, and an address, FIELD—2, to the next three characters, which are regarded as alphanumeric.

The second record *remaps* the same five characters, and includes the sixth, quite differently. A—FIELD is the address of the first three characters which are numeric, and B—FIELD refers to the next three, which are also numeric.

Whenever the programmer wishes to address any of this part of the record area, he will use the field name which will give him the correct characters viewed in the way he requires. If he discovers that he needs to address these five characters as, say, one numeric field, he must insert a third record description in the Data Division, starting with an entry such as

```
01  THIRD-REC.
    02 FIELD-ONE PIC 9(5).
```

If an input file has several record types, there should be at least one field in the same position in each record to indicate the type. After reading a record, the program will then examine this field and branch to appropriate Procedure Division statements which refer to the fields in the correct record description.

In output files, the most obvious example of multiple record types is to build up different formats for output on the line printer. When producing an invoice, for example, the layout of fields in

- name and address lines
- heading lines
- detail lines
- total lines

will all differ. Within the output file, each of these kinds of data will be given a separate record description. The programmer will build up the correct data in the output area by using the field names appropriate to the record he is creating. The record can then be printed by a WRITE order; in the case of the printer (and punches) it does not matter which record name is specified.

THE WORKING-STORAGE SECTION

Once files and records have been defined in the File Section, it is possible by means of Procedure Division statements to move data from an input to an output file, to add two items of an input file and record the result in an output file, and so on. In most programs it is also necessary to accumulate intermediate results and create constants for use by the program. Storage for this purpose is allocated in the Working--Storage Section, and will be available to all sections of the program while it is being run.

Working-storage areas are described in exactly the same way as in the File Section; records and fields are named, and characterised by the Picture clause. Other symbols and clauses used in the File Section, described under Editing Symbols later in this chapter are also applicable to the Working-Storage Section.

Following the section heading, WORKING-STORAGE SECTION, there are two types of storage which may be allocated in this section, these are known as *contiguous storage* and *non-contiguous storage*.

Contiguous Storage

Contiguous storage is the name given to areas reserved for fields which are grouped together in a level structure to form records. This method may be used, for example, to set up tables of constants (see OCCURS later in this chapter), or to duplicate the format of an input or output record which requires working-storage.

As many records as are necessary can be defined. Records are defined exactly as in the File Section. However, if a record is identical to one in the File Section, the data names must be distinguished from those in the original record. This can be done without changing the names, by adding a hyphenated suffix to the original names. For example, a record defined in the File Section as shown below:

```
01  DETAIL.
    02 SIZE PIC 99.
    02 COLOUR PIC A.
    02 FITTING PIC 9V9.
```

could appear in the Working-Storage Section in the following form.

```
WORKING-STORAGE SECTION.
01  DETAIL-1.
    02 SIZE-1 PIC 99 COMP SYNC RIGHT.
    02 COLOUR-1 PIC A.
    02 FITTING-1 PIC 9V9 COMP SYNC RIGHT.
```

It should be noted that numeric items used to hold subtotals or intermediate results can usefully be described as COMP SYNC RIGHT (see later in this chapter).

Non-contiguous storage

Non-contiguous storage is allocated to elementary fields which are independent of any other field and are used in isolation.

It is conventional to write any non-contiguous entries first in the Working-Storage Section, followed by contiguous record entries, as shown below.

```
WORKING-STORAGE SECTION.
01  WORK-AREA PIC 999.
01  LINE-COUNT PIC 99 COMP SYNC RIGHT.
01  DETAIL-1.
    02 SIZE-1 PIC XX.
```

VALUE clause

This clause allows fields in the Working-Storage Section to be set to particular values at the start of the program; in other words, it allows *constants* to be set up. It has the format

VALUE IS literal

where the literal can be numeric, as for example,

VALUE 7521

or, non-numeric, as for example,

VALUE "TOTAL"

The rules for literals are given in Chapter 7 "LITERALS" but it is sufficient to say here that non-numeric literals must be bounded by quotation marks. A figurative constant (see page 112) can be substituted for a literal.

While VALUE clauses containing numeric literals are used during processing and in arithmetic operations, non-numeric literals are often used to output headings on printed documents. See *WRITE,* Chapter 7.

```
01  INITIAL-TOTAL PIC 999 VALUE 373.
01  FACTOR PIC 9V9 VALUE 1.5.
01  HEADING PIC X(15) VALUE
      "TOTAL-UNITS ISSUED".
```

The last example shows an 01 level used to specify a heading.

EDITING AND USAGE

Conversion of data from one form to another is often required in a COBOL program, especially before output. The Picture of an output or working-storage field can change the way in which the data read into it is stored, although the actual value remains the same. Such conversion is carried out by steps generated by the compiler, which ensure that when data is moved to a field it takes on the form dictated by the PICTURE clause and usage clauses of the receiving field. This applies when data is MOVED into a field, or when a new field is used to hold the result after the arithmetic operation. (It does not apply when data is READ into a field.)

There are two ways in which data can be converted from the form it takes in the source field, before the operation, to the form it takes in the result or receiving field, after the operation.

1 EDITING this provides a means of inserting in a Picture certain symbols which clarify the data and show the form it takes in output. For example, they can indicate where the decimal point is located and whether it has a negative value. Editing symbols make data meaningful to the human user, and are used especially in output via the line printer.

2 USAGE this determines the way in which data is stored in the receiving or result field, by certain clauses placed after the Picture of the item. Data is usually stored in a standard way, but for program efficiency it may be convenient to specify, for example, whether an operational sign occupies a separate character.

When the programmer determines the form of an output or working-storage field, he must select those editing symbols and usage clauses which force data to take on the form he requires when it is moved into the field. The distinction between the Picture of this type of field and an input field is that the latter gives an accurate description of data as it is held. Thus data is not changed in any way, and when the input file is read by a READ instruction, data is simply transferred into store. No conversion takes place, so that the Picture symbols and usage clauses must define the item exactly as it stands.

As with the PICTURE clause, editing and usage must only be applied to elementary fields. The editing described here operates when data is placed in an elementary field, or when it is moved from elementary field to elementary field. Different effects are achieved when a group field is moved to a group field, and editing does not take place. Instead, each group field is regarded as an unedited alphanumeric elementary field and the rules for moving between fields of this type are obeyed. This rule applies when the elementary fields in the group have editing symbols or usage clauses in their descriptions. See *MOVE*, Chapter 7 for details of moving fields.

Editing symbols

The PICTURE clause has been considered up to now as a means of specifying the size and nature of the field. Further symbols can also be added to the character string to specify editing requirements. When data is moved to a result or receiving field it is often useful to make the result more meaningful. This can be achieved by zero suppression, insertion of characters and report signs. These, for example, suppress leading zeros, and insert punctuation, the decimal point and a positive or negative sign.

When the compiler sees these requirements in the Picture of a field, they are recorded in the Data Division tables. Later, when the field is the receiving field of a Procedure Division statement, the conversion to machine codes takes editing into account and inserts steps to achieve the required result.

The various symbols which may be used are classified as follows:

Zero suppression	Z
	*
Insertion characters	, comma
	. full stop
	£
	$
	B
	0
Report signs	+
	−
	CR
	DB

Each symbol is considered separately in this section. It should be noted that only numeric fields may use all the above symbols. Alphabetic and alphanumeric fields may use only the insertion characters 0 and B. It should also be noted that all fields including numeric fields, which contain editing symbols, are classified as alphanumeric.

ZERO SUPPRESSION

When the Picture of a result field contains Z in certain positions, leading non-significant zeros in these positions are altered to space characters. Z is written instead of 9 for each digit which is to be zero-suppressed.

For example, if a Picture were written as
PIC ZZZ99.
then zeros would be suppressed down to the last two digits, and the result 00004 would appear as 04.

If the field contained zeros, two zeros would be printed.

Examples

	Result field	
Source data	*Picture*	*Edited result*
01234	Z9999	1234
00012	Z9999	0012
01234	ZZZ99	1234
00000	ZZ999	000
00000	ZZZZZ	(blank)
10001	ZZ999	10001

* *Asterisk*

The insertion of asterisks follows exactly the same procedure as zero suppression, except that leading zeros are replaced by asterisks which appear in the result field. This operation is known as *asterisk fill* or *cheque protect*.

Examples

		Result field
Source data	*Picture*	*Edited result*
12345	***99	12345
00123	**999	**123
00012	**999	**012

INSERTION CHARACTERS

, Comma

The main use of the comma in a field is to improve the legibility of printed output. A comma will be inserted in the result field in the position shown in the Picture, but it may not appear as the right hand character. For example,

PIC 99,999

would result in a value of 12345 being printed as 12,345.

Commas can also be combined with Zs. For instance,

PIC ZZ,ZZ9.

If there are no significant digits to the left of the comma, the comma itself is suppressed and replaced by a space character.

On the Continent, a comma often serves as a decimal point.

Examples

		Result field
Source data	*Picture*	*Edited result*
12345	99,999	12,345
00012	99,999	00,012
00012	Z9,999	0,012
00012	ZZ,999	012

. Full Stop

The main use of the full stop is to represent a decimal point on printed output. The decimal point is not normally represented in store and must be inserted before output, occupying a separate character position. The stop as a decimal point must always be accompanied by V in the Picture, to indicate to the compiler the position of the point. This is because a full stop has no special significance to the compiler. V can be placed before or after the stop. For example,

PIC 99V.9

will interpret the value 606 as 60.6.

If a full stop is used without V, the result will not represent a decimal point, but will merely improve legibility. This use is common on the Continent.

Examples

		Result field
Source data	*Picture*	*Edited result*
150325	9,999V.99	1,503.25
160668	99.99.99	16.06.68
123456	9.999V,99	1.234,56
00456	ZZZV.99	4.56

The second example shows the full stop being used to denote a date. The third shows the continental use, where the full stop is merely for clarity and the comma represents the decimal point. As seen, the compiler positions the decimal point wherever there is a V, and the stop or comma is incidental.

£

A single pound sign can be placed in front of a sterling field to produce a pound sign in that position of the result field. For example,

PIC £9,999.

will output the value 0162 as

£0,162

The £ sign can also be *floated*, to print in front of the most significant non-zero character. £ is placed in each position where non-significant zeros are to be suppressed.

PIC ££999.

would print the field 0012 as

£012

Examples

		Result field
Source data	*Picture*	*Edited result*
12345	£99999	£12345
00012	£99999	£00012
00012	£££££	£12
00012	£££££9	£12
00012	£££999	£012

$

The dollar sign placed in front of a monetary field operates in exactly the same way as the pound sign. For example,

PIC $999

will print 370 as

$370

The dollar sign can also be floated.

Examples

		Result field
Source	*Picture*	*Edited result*
37025	$999.99	$370.25
01250	$$$9.99	$12.50

B

Space characters can be inserted in a field for clarity by using the special character B. If B is used in a Picture, a blank will be inserted in the field in that position. For example, to place a space in a sterling field,

PIC £££B99.

would result in 01309 being printed as

£13 09

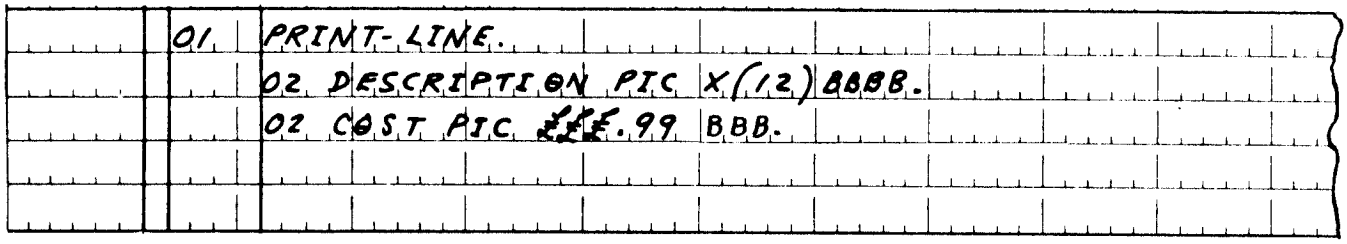
Examples

		Result field
Source data	*Picture*	*Edited result*
010769	XXBXXBXX	01 07 69
1325	9BB9B9BB9	1 32 5
JOHNBROWN	AAAABAAAAA	JOHN BROWN

B can also be used to insert spaces between fields in a print record, instead of using Filler. The part of a record description given below requires spaces to be moved to Filler fields in the Procedure Division.

```
01  PRINT-LINE.
    02  DESCRIPTION PIC X(12).
    02  FILLER PIC XXXX.
    02  COST PIC £ZZZ.99.
    02  FILLER PIC XXX.
```

However, this could be written by the following statements, which require no Procedure Division action at all.



This way of representing spaces could take longer at run time, since the spaces are inserted each time the fields are receiving fields. Using Filler, it may be possible to move spaces into the Filler fields only once.

0

If this character is used in a Picture, a zero will be inserted in the position shown. This operates in the same way as does B.

Examples

		Result field
Source data	*Picture*	*Edited result*
12345	9909099	1203045
12345	0999990	0123450
00012	ZZZZZ00	1200

Note: The use of B and O is very inefficient, it would be better to describe the record as shown in the first example.

REPORT SIGNS

Report signs are printed to indicate the plus or minus value of an item. They are used to replace the S which would normally be necessary for a field which might contain a negative value.

+ *Plus*

A plus sign placed in the first or last position of a Picture will cause a plus sign to be printed if the item is positive, or a minus sign if the item is negative. For example,

PIC +999.

will print 203 as

+203

or −203

Examples

		Result field
Source data	*Picture*	*Edited result*
78901	+99999	+78901
78901 (minus)	+99999	−78901
68351 (minus)	99999+	68351−
00000	+9(5)	+00000

− *Minus*

A minus sign is used in a similar way to the plus sign. A minus is inserted if the field is negative, but plus is replaced by a space if the field is positive. For example,

PIC −999.

would print 203 as

203

or −203

Floating + and −

If plus and minus signs are placed before an item, they can be floated in the same way as the £ sign described on page 00. Leading non-significant zeros will be suppressed in each position where + or − is placed.

Examples

		Result field
Source data	*Picture*	*Edited result*
12345	+ + + 999	+ 12345
00123	+ + + 999	+ 123
00012 (minus)	+ + + 999	− 012
00123	− − − 999	123
00123 (minus)	− − − 999	− 123
00001	− − − − − −	1

CR and DB

Instead of denoting a negative field by a minus sign, a field can be followed by the characters CR or DB, which must appear as the last two characters in the Picture. If the value of the item in the source field is negative, CR or DB will appear in the result field. If the value is positive, they will be replaced by spaces. For example,

PIC 999DB.

or PIC 999CR.

would cause 175 to print as

175

if positive,

175DB

or 175CR

if negative.

Examples

		Result field
Source data	*Picture*	*Edited result*
12345 (minus)	99999CR	12345CR
12345	99999DB	12345
00001 (minus)	ZZZZZDB	1DB

Rules for using the PICTURE clause

There are certain rules which must be observed when the PICTURE clause and editing symbols are used. These rules are summarised below.

1 The number of characters and symbols in a Picture is limited to a maximum of 30.

2 The PICTURE clause must not be used to describe a field with a size of more than 120 characters.

3 All characters other than operational characters (V S and P) are counted in determining the size of an item.

4 The Picture of a field must always be large enough to contain the longest possible answer plus any insertion characters or report signs. One character should be allowed for

£, . + −

and two characters for CR and DB.

5 A field is alphanumeric if its picture contains any of the editing characters.

6 Once data has been edited it should not be involved in further arithmetic, and should be moved only to a field containing Xs. The data should still be available in its unedited form in some other field.

The table given below shows the combinations of symbols which can be used to form Pictures. To find out whether a pair of symbols can be combined, locate the preceding symbol in the left-hand column, and the following symbol along the top of the table. The entry in the table corresponding to the pair will indicate whether it is a valid or invalid combination. This information applies not only if the symbols are adjacent but also if they appear in the same Picture.

It should be noted that $ and O are not included; $ follows exactly the same rules as £, and O as B.

The key to the table is as follows:

Blank square	Combination is invalid.
✓	Combination is valid.
L	Combination is valid provided the second of the symbols is the last character in a Picture (except for a P).
9	Combination is valid provided a 9 does not follow the second of the two symbols.
1	Combination is valid provided the first symbol occurs only once.

	A	X	9	V	S	P	Z	*	£	,	.	B	+	-	CR	DB
A	✓											✓				
X		✓										✓				
9			✓	✓		✓				✓	✓	✓	L	L	L	L
V			✓				9	9		✓	✓	✓	9	9	L	L
S			✓	✓												
P			✓			✓	✓	✓		✓	✓	✓	✓	✓	L	L
Z			✓	✓		✓	✓			✓	✓	✓	L	L	L	L
*			✓	✓		✓	✓			✓	✓	✓	L	L	L	L
£			✓	✓		✓	1	1	✓	✓	✓	✓	✓	✓	L	L
,			✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	L	L
.			✓	✓		✓	✓	✓	✓	✓	✓		✓	✓	L	L
B	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	L	L
+			✓	✓		✓	1	1	1	✓	✓	✓	✓			
-			✓	✓		✓	1	1	1	✓	✓	✓		✓		
CR																
DB																

Usage clause

As mentioned earlier, the compiler can be told about the nature of the data in a field by means of a usage clause added to the description of an elementary field. Usage can be used to determine the way data is to be held, in an output or working—storage field, but may only describe data as it is already held, in an input field. For example, COMP should be used to define an input field held in binary on magnetic tape.

The usage clause is largely machine-orientated, as it depends on the format of data as held within the store of the particular machine. Usage as applied to the 1900 Series and handled by its COBOL compilers is described in this section. It should be noted that the clauses listed may have different meanings if they are used on other computers.

The following types of usage corresponding to different forms of data are recognised by 1900 compilers:

DISPLAY
DISPLAY–3
COMPUTATIONAL (abbreviated to COMP)
SYNCHRONIZED (abbreviated to SYNC)

In general, DISPLAY indicates character fields of various kinds and COMP indicates binary fields. A usage clause is sometimes followed by the SYNC clause.

These clauses are usually written immediately following the PICTURE clause of the appropriate field, preceded by a space.

DISPLAY

This denotes a field containing numeric or alphanumeric characters, each of which is represented in store by a 1900 six-bit character.

With any field whose Picture is X or 9, DISPLAY is assumed if no usage clause appears. There is therefore no need to use DISPLAY, as this is the standard mode of storage.

If any DISPLAY field could contain a negative value (represented by S in the Picture) the negative sign would be held by *overpunching* of the left hand character. If this character in a negative field is in the range 1 to 9 it will be held, combined with the sign, as a letter from the range J to R. If it is zero, it will be held with the sign as the quotes character "

Thus, 3041 would be held as

3041

if positive, or

L041

if negative.

DISPLAY–3

This is used to describe a field when the negative sign is always held as a separate character in the most significant character position. This does not, however, affect the size of the Picture. –3041 would be held as

L	0	4	1

with DISPLAY, or

–	3	0	4	1

with DISPLAY–3, but in either case the Picture is

PIC S9999.

COMP

This clause indicates a field held in binary, which must have a Picture made up of 9's. Arithmetic on the 1900 Series is performed on binary values. When calculations are performed on DISPLAY items the compiler converts the data to binary form for processing and back to character form afterwards. However, it is convenient to keep the number in binary form if, for example, output results on magnetic tape are to be used as input for subsequent runs. In this case the field is declared as COMP and conversion is only performed once, before the field is used for processing, instead of before and after each calculation.

A COMP field must be a numeric field. When a field could contain a fraction, a COMP field scales up the value of the fractional places to hold it as an integer. The value 12.34 would have a Picture

PIC 99V99 COMP.

and would be held as 1234. Decimal points are aligned when COMP fields are involved in arithmetic.

A figure held in characters can be converted to binary by moving it to a field described as COMP. If it is to be used for calculation, it is useful to set up a field in the Working—Storage Section. For example, an input field called DAYS—LATE is shown below.

```
03 DAYS-LATE PIC 999.
```

To convert DAYS—LATE to binary and perform arithmetic on it, a COMP field called PAY—DEDUCTION is declared in Working—storage. An instruction which moves DAYS—LATE to PAY—DEDUCTION will make the necessary conversion. Note that COMP cannot be used to describe an input field which is not already held in binary, but only a result or receiving field.

```
01 PAY-DEDUCTION PIC 999 COMP.
```

SYNC

The SYNCHRONIZED clause (abbreviated to SYNC) provides a convenient way of allocating storage to binary fields described as COMP.

In a word machine like the 1900 Series the compiler allocates consecutive storage for Data Division items, without regard to the beginning and end of words; that is, an item may begin in the middle of one word and end in another. This method of storage is economical, but if a field is referred to in the Procedure Division, the compiler must generate object program instructions to extract and combine the two parts of the field. When a field is referenced several times it is useful to allocate storage more conveniently. This can be done by declaring a field as

SYNC RIGHT

following COMP. These clauses specify that an item is to be *synchronized*, that is, to be stored in one or more complete computer words.

SYNC RIGHT is normally used for numeric binary fields. Each item is stored within one or more complete words, starting at the right hand end of the least significant word. This means that three fields described as would be stored so that A1, A2 and A3 are each in a separate word.

```
02 A1 PIC 9(5) COMP SYNC RIGHT.
02 A2 PIC 9(4) COMP SYNC RIGHT.
02 A3 PIC 9(3) COMP SYNC RIGHT.
```

If an output item, such as a field output to magnetic tape, is described as SYNC RIGHT it will be released to the peripheral in synchronized form. For example, if a field in character form occupied five characters, that is, over one word, moving it to this field before output

```
02 ITEM-A PIC 9(5) COMP SYNC RIGHT.
```

would cause it to be released for output in binary form right justified in one word.

If this output item were to be used subsequently for input, COMP SYNC RIGHT would again have to be included in the input field description, otherwise the compiler would not be aware of the format of the data.

COMP SYNC RIGHT

As a general rule for efficiency, an input data field which is involved in arithmetic more than once should be converted to binary in a COMP SYNC RIGHT working—storage field with the same Picture as the item. The item is then moved to this field once, before it is used in calculations.

If, for example, an input field were defined as

```
02 PERCENTAGE PIC 99V9.
```

the following working—storage field would be set up:

```
01 WORK-STORE PIC 99V9 COMP SYNC RIGHT.
```

This would ensure that the field were synchronized in one word of store. As a result, the minimum number of machine code steps would be required to translate statements handling WORK—STORE.

Any answers in binary can easily be converted into character form for output, by being moved to an output DISPLAY field.

Note: When arithmetic is performed on the 1900 Series it is always on complete words. Describing fields as SYNC thus enables the compiler to perform arithmetic without worrying whether the field occupies the whole word or not. No check is made at run time to see whether the result field is large enough unless the ON SIZE ERROR clause (see Chapter 7) has been included. If a field is negative the sign is propagated up to bit 0 of the word(s).

OCCURS

It is often necessary in the Data Division to describe a series of items which are exactly the same in size and format. The only difference between any two items is that the actual values they contain at run-time are different. These items can be defined by giving a field description to each one. This method is quite practicable for a few fields but cumbersome for a large number, and this is where the OCCURS clause can be used, enabling identical items to be defined in one line of the Data Division. OCCURS can be used either in the File Section or Working—Storage Section.

For example, a card consists of six different quantities, one for each day of the week. The following record description might be written for these items (assuming that there were no spaces between fields).

```
01 WEEK-CARD.
   02 MON-AMOUNT PIC 9999.
   02 TUES-AMOUNT PIC 9999.
   02 WEDS-AMOUNT PIC 9999.
   02 THURS-AMOUNT PIC 9999.
   02 FRI-AMOUNT PIC 9999.
   02 SAT-AMOUNT PIC 9999.
   02 WEEK-TOTAL.
      03 GAINS PIC 9(5).
      03 LOSSES PIC 9(5).
```

Using the OCCURS clause, however, one field entry is written, followed by the number of times the item appears in contiguous storage. This clause has the format

OCCURS integer TIMES

Notice that TIMES is optional. The above example could now be expressed as shown.

```
01  WEEK-CARD.
    02 WEEK-AMOUNT PIC 9999 OCCURS 6 TIMES.
    02 WEEK-TOTAL.
       03 GAINS PIC 9(5).
       03 LOSSES PIC 9(5).
```

One collective name is given to all the fields, and this causes storage to be reserved for six quantities, one for each day, of four digits each. Notice that the rest of the record is unchanged.

Data laid out with an OCCURS clause is often described as a *table*.

Subscripts

Since all fields now have the same name, they must somehow be distinguished when they are addressed in the Procedure Division. This is done by qualifying the name by what is known as a *subscript*. The subscript may be either a literal (see Chapter 7 "LITERALS") or another data name, and is written in brackets immediately following the general data name for the fields. Thus the format of a subscript is

data-name-1 ({ literal / data-name-2 })

The rules for spacing of the brackets are given below, under rule 3.

Using this method, any one of the fields included in the OCCURS clause can be identified in one of two ways.

First, the subscript may be a numeric literal. Each field is therefore assumed to have a number, so that the most significant field is 1, the second 2 and so on. If it is required to refer to the fifth item in the example above, WEEK—AMOUNT(5) would be written. The compiler then locates the correct item in store. The Procedure Division statement

ADD WEEK—AMOUNT(5) TO QUANTITY.

would therefore cause the quantity for day 5 (FRI—AMOUNT in the first example above), to be added to the contents of QUANTITY.

Alternatively, the name of another field may be used as a subscript, if the item required is indeterminate. This name will be a data name, denoting a field which is used to hold a reference to the item. For example,

ADD WEEK—AMOUNT(DAY) TO QUANTITY.

names a field, DAY, whose current value must be in the range 1 to 6. If the value of DAY is 1, WEEK—AMOUNT (DAY) means WEEK—AMOUNT(1), and so on. With this second method, the same procedure statement can be used to refer to any of the six fields, depending on the value of DAY. This is particularly useful when the field wanted is unknown at the start of a program, for example, when it depends on the fulfilling of conditions.

RULES FOR SUBSCRIPTS

The rules for using subscripts are summarised here:

1 A literal or the contents of a data name used as a subscript must be a positive whole number. The compiler does not check that the subscript is within range; the programmer must do this.

2 When a subscript is represented by a data name, this name must be defined in the Data Division.

3 A subscript or set of subscripts must be enclosed in brackets immediately following the item referred to. There can be a space (or not) before the first bracket, but there must be a space after the second bracket. Inside the brackets, there should be no space between the first bracket and first character, and the last character and second bracket, but there must be a space between subscripts if there is more than one.

OCCURS with Group fields

So far only elementary fields used with OCCURS have been mentioned. OCCURS can however also be used to describe group fields.

Taking another example of a six-day week, imagine a clock-card, containing hours worked, and total earnings for each day. Instead of writing out the fields for each day, the following description could be given:

```
01  WEEK-CARD.
    02 FILLER OCCURS 6.
       03 HOURS-WORKED PIC 99V9.
       03 TOTAL-EARNED PIC 9(6).
```

The name FILLER is given to the group field because it will not be addressed in the Procedure Division. If this was necessary, the field would have to be given a name as shown below.

```
01  WEEK-CARD.
    02 DAY-DATA OCCURS 6.
       03 HOURS-WORKED PIC 99V9.
       03 TOTAL-EARNED PIC 9(6).
```

In the first example, FILLER cannot be addressed by a statement at all, but there will be six pairs of fields, HOURS—WORKED and TOTAL—EARNED held sequentially in store. These items can be subscripted, so that

HOURS—WORKED(2)

refers to the second field containing daily totals, and

TOTAL—EARNED(6)

to the sixth field containing earnings. Alternatively, a data name could be used, as for example,

HOURS—WORKED(REFERENCE)

where REFERENCE contains a number from 1 to 6.

In the second example, the group field can be addressed in a statement such as

MOVE DAY—DATA(6) TO ASSESSMENT.

This has the effect of moving the sixth group field, consisting of HOURS—WORKED (6) and TOTAL—EARNED(6), to an area called ASSESSMENT.

Two and three-level tables

All the examples given above are of one-dimensional tables. A two-dimensional table can be defined by using OCCURS at two different levels, that is, by describing a table within another table.

For example, suppose a quantity field were required for each day of a five day week in a four week month. The following Data Division shows how such a table could be set up, defining four fields called WEEK, each containing five fields called DAY—QTY.

```
01  TABLE.
    02  WEEK  OCCURS  4.
        03  DAY-QTY  PIC  999  OCCURS  5.
```

This causes storage space to be reserved for twenty three-digit quantities. It should be noted that only the lowest level requires a PICTURE clause. Any quantity can now be addressed in the Procedure Division by means of two subscripts, the first giving the week, and the second the day. Therefore, the fifth day of the second week can be addressed as

DAY—QTY (2, 5)

Notice that the subscript of the group field is addressed first. Similarly,

DAY—QTY(SUB—1, SUB—2)

could be used, assuming that SUB—1 and SUB—2 are fields containing the week number and day number respectively.

WEEK can be addressed as a group name in the Procedure Division, but must always have a single subscript. Thus,

WEEK(2)

refers to the five quantity fields of the second week.

Similarly, OCCURS can be used with a three-dimensional table. Consider the above example, if a quantity for the day and week were required for a thirteen month year. A table to handle this could be set up by the following statements:

```
01  TABLE.
    02  MONTH  OCCURS  13.
        03  WEEK  OCCURS  4.
            04  DAY-QTY  PIC  999  OCCURS  5.
```

This lays out 260 three-digit fields sequentially in store.

To address a DAY–QTY field, a three-level subscript must be used. The field for the fifth day of the second week of the tenth month would be addressed as

DAY–QTY(10, 2, 5)

or

DAY–QTY(SUB–1, SUB–2, SUB–3)

assuming that three subscripts have been set up to refer to the month, week and day.

A three-dimensional table is the maximum that may be used in COBOL, but there is no need for the lowest level to be an elementary field. The following example shows an example where there are three fields for each day of a year.

```
01  TABLE.
    02  MONTH  OCCURS  13.
        03  WEEK  OCCURS  4.
            04  DAY-ENTRY  OCCURS  5.
                05  CODE-D  PIC X(5).
                05  QTY-D  PIC 999.
                05  PRICE-D  PIC 99V9.
```

The processing of multi-level tables is considered under complex PERFORM, Chapter 7. It is important to remember that when a multi-level table is involved, two or three subscripts must be set up and if a subscript is involved, this must be advanced.

Uses of OCCURS

So far, the rules for OCCURS have been illustrated. This section gives some common applications of OCCURS. Further uses with REDEFINES are given at the end of this chapter.

Most of the examples in this chapter have considered OCCURS used with input records. However, exactly the same procedure is followed if OCCURS is used with a working–storage or output field.

It should be noted that OCCURS cannot be used at 01 level, nor can it be combined with the VALUE clause. This second restriction applies to elementary fields within a group field defined by OCCURS, as well as to the group field itself.

TABLE LOOK—UP

Table look-up is the selection by means of a subscript, of one of a series of constants laid out in a table. A table of constants can only be set up in the Working—Storage Section, as described earlier in this chapter under "VALUE clause", and is used to hold values. These values, placed in the record area at the beginning of the program, can form a basis for calculation in other parts of the program.

The procedure for locating a constant within a table is exactly the same as that for variable items. Consider the part of a Working—Storage Section and the part of a Procedure Division given below.

```
WORKING-STORAGE SECTION.
01  FACTORY PIC 9.
01  FILLER.
    02 MEN-EMP PIC 9(6) COMP SYNC RIGHT
       OCCURS 6.
PROCEDURE DIVISION.
{                 }
ADD WOMEN-EMP TO MEN-EMP (FACTORY).
```

An analysis is to be made of employment figures at six factories. Assuming that suitable constants have been placed in the six areas of store called MEN—EMP, a figure in the range 1 to 6 is held in another field, FACTORY. The Procedure Division statement adds a variable quantity, WOMEN—EMP, into one of the six fields, according to the current contents of FACTORY.

SUBSCRIPT LOOPS

In addition to the use of OCCURS shown above, subscripting may also be used to save steps in the object program. If the same process is to be carried out on a series of identical items, all of which are available at the same time, it may be possible to use the same procedure for each item. In this case, subscripting in COBOL is equivalent to modification in assembly languages it enables one statement or set of statements to process each item in turn.

The factors which are required to be processed, and any result fields, must be laid out as a table using the OCCURS clause. Each factor is then processed by means of a *subscript loop*. This is a group of Procedure Division statements which deals with the elements of the table one by one, storing the answers in the result table. As the processing statements refer to all the items, procedures must also be included to advance the subscripts, that is, to move on to the next item, and to count the number of times the loop has been obeyed, so that an exit is made at the right moment.

Take a simple example, where an input record has twelve identical fields, each containing a price and quantity. Assuming that the group field is not to be addressed, the following definition can be written.

```
01  SALES.
    02  FILLER  OCCURS  12.
        03  PRICE  PIC  9(5).
        03  QUANTITY  PIC  99V9.
```

For each field, the price is to be multiplied by the quantity and the result stored in a further field. Result fields are set up in the Working–Storage Section, also in the form of a table. Storage is also set aside for a count of the number of times the loop is performed, as shown below.

```
WORKING-STORAGE  SECTION.
01  COUNT  PIC  99  COMP  SYNC  RIGHT.
01  FILLER.
    02  TOTAL-COST  PIC  9(6).
```

The loop consists of a number of Procedure Division statements headed by a paragraph name.

```
    MOVE  1  TO  COUNT.
LOOP.
    MULTIPLY  PRICE(COUNT)  BY  QUANTITY(COUNT)
      GIVING  TOTAL-COST(COUNT).
    ADD  1  TO  COUNT.
    IF  COUNT  <  13  GO  TO  LOOP.
```

All these procedures are fully explained in Chapter 7, but their general functions are self-explanatory. Before the loop is performed for the first time, 1 is moved to the area called COUNT. This means that COUNT can be used to contain the subscript; when the first item is processed, it is located by referring to the 1 contained in COUNT. Similarly, 1 is added to COUNT each time the loop has been run through, so that it holds the number of the next subscripted items. Each pair of items is multiplied and stored in the correct TOTAL–COST field. At the end of the loop, a test determines whether COUNT is less than 13. If it is less, there are still items to be processed, but if it contains 13, all the items have been processed; the program then passes out of the loop and continues to the next statement in the program.

REDEFINES

It is often useful to have several different descriptions of the same area of store. This is a concept which has already been met in connection with records; the different types of record in a file are different views of the same record area. Similarly, this principle can be applied at the field level by means of the REDEFINES clause. This enables fields or groups of fields to be viewed in more than one way. Procedure Division statements can then perform different operations on the separate views of an item.

The REDEFINES clause has the format

level—number data—name—1 REDEFINES data—name—2

which indicates that the field or fields described by data—name—1 are an alternative view of those under data—name—2. Both definitions must have the same level number and must occupy precisely the same amount of store. REDEFINES may be used to define either group fields or elementary fields, but it may not be used at 01 level in the File Section (different records automatically redefine an area of store).

For example, in the following clauses, REDEFINES is used with elementary fields.

```
02 A-FD PIC XXXXX.
02 QTY REDEFINES A-FD PIC 99V999.
```

In this case, the second Picture of the field can be written on the same line after the REDEFINES clause. A—FD, a five-character alphanumeric field is redefined by QTY, which has five numeric digits (note that the V does not take up a character position). The two views may be different in their nature, but the items must be the same size. Whenever the area is referred to as A—FD in the Procedure Division, the first view will be taken by the compiler; whenever it is referred to as QTY the second view.

A whole group of fields can be redefined, as shown in the example below.

```
02 BANK-ACC.
   03 ACC-NO PIC 9999P.
   03 ACC-NAME.
      04 INITIALS PIC AAA.
      04 SURNAME PIC X(20).
02 GIRO-ACC REDEFINES BANK-ACC.
   03 GIRO-NO PIC 9(12).
   03 PO-NAME PIC X(15).
```

In this example, the group field GIRO—ACC redefines the area BANK—ACC, and the whole structure of sub-fields is reorganised. The area of store is allocated quite differently in the second description, and these definitions will be taken when the fields are addressed as GIRO—ACC. REDEFINES is used in this way either when the same data is to be organised and referenced in two different ways, or when part of a record can contain more than one type of data.

It must be emphasised that VALUE must not be used in the same entry as REDEFINES, nor in a sub-field of an entry containing REDEFINES.

Rules for REDEFINES

The rules for REDEFINES are summarised as follows:

1 This clause may not be used to redefine records (01 level) in the File Section. It may however be used for records in the Working—Storage Section and the Linkage Section.

2 The level number of data—name—1 must be the same as that of data—name—2.

3 The REDEFINES clause must immediately follow data—name—1.

4 The entries giving the new description of the area must immediately follow the REDEFINES entry.

5 There must be no other field with the same, or a numerically less level number between data—name—1 and data—name—2.

6 When redefining 02 levels or below, data—name—1 and data—name—2 must be of the same size. Only 01 levels (in the Working—Storage Section) may be of a different size.

7 A VALUE clause may not be used with a REDEFINES entry, or with any of the sub-fields redefined.

Uses of REDEFINES

There are many applications of REDEFINES in addition to those given above, and two of the most common are considered here.

When two or more types of record in a file are almost identical, it is often unnecessary to declare several record types. It may well be possible to declare one record type and cover the differences with REDEFINES. The following example shows a record which includes two layouts of data, where only one field needs to be redefined.

```
01  DOMICILE.
    02  STREET-NO  PIC 999.
    02  STREET-NAME  PIC X(15).
    02  CITY  PIC X(24).
    02  DISTRICT REDEFINES CITY.
        03  TOWN  PIC X(12).
        03  POSTCODE  PIC X(6).
        03  COUNTY  PIC X(6).
    02  COUNTRY  PIC X(20).
```

REDEFINES also enables an operand to be built up from two or more parts, or to be dissected down to the character level. In the first case, two or more values which were separate fields in a record could be redefined as a complete field. In the second instance, it might be required to examine each character of a numeric field to ensure that there is no alpha punching, but also to treat the data as one field for arithmetic.

An example of this is given below.

```
02 QTY PIC 99V9.
02 CHARS REDEFINES QTY.
   03 A PIC 9.
   03 B PIC 9.
   03 C PIC 9.
```

The description of CHARS could be abbreviated by using the OCCURS clause, and thus the examination of each character could be done by a subscript loop (see the end of this chapter).

REDEFINES USED WITH OCCURS

Finally, three major applications of REDEFINES also require OCCURS, and these are considered in this section.

Filling of tables

As explained earlier, it is often necessary to set up tables of constants using OCCURS, and use them as a basis for calculation.

These constants can only be given in the Working—Storage Section. Data can be set up in separate fields with the VALUE clause, and then, since the VALUE clause cannot be associated directly with OCCURS or REDEFINES, the fields can then be redefined in the form of a table. This table can then be addressed by a subscript in order to locate any of the constants held in it.

Consider the entries given below.

```
01  REC-ONE.
    02 SEP.
       03 ONE PIC 999 VALUE 100.
       03 TWO PIC 999 VALUE 150.
       03 THREE PIC 999 VALUE 200.
       03 FOUR PIC 999 VALUE 275.
       03 FIVE PIC 999 VALUE 750.
    02 TABLE REDEFINES SEP PIC 999 OCCURS 5.
```

In this example, five identical fields are filled by the VALUE clause; the whole group field SEP is then redefined as TABLE, and can be addressed by a subscript such as

TABLE(3)

This is the address of the field THREE which contains 200.

Note: In the example the data names ONE to FIVE could have been FILLER, as they are never referenced in the Procedure Division.

Data validation

It may be required to hold the characters of a single operand separately in order to check their validity. This situation is common when a field must contain, say, only numeric characters, or characters within a certain range.

For example, a six-character code number must consist of any combination of the letters A to F and the digits 0 to 9. The code is required as one field, but each character must also be tested to see if it is valid. The code can therefore be defined as follows:

```
    02 CODE-NO PIC X(6).
    02 CHARS REDEFINES CODE-NO PIC X
       OCCURS 6.
```

It can now be treated as one six-character field, CODE—NO, or six identical fields, CHARS, of one following statements check the individual characters, following statements check the individual characters.

```
           MOVE 1 TO SUB.
      TEST.
           MOVE CHARS(SUB) TO WORK.
           IF WORK NOT > 9 GO TO NEXT-1.
           IF WORK < "A" GO TO ERROR-C.
           IF WORK > "F" GO TO ERROR-C.
      NEXT-1.
           ADD 1 TO SUB.
           IF SUB < 7 GO TO TEST.
```

Each CHARS field is referenced by SUB and is moved into WORK for testing. A check is made to see that WORK is not greater than 9 in the collating sequence; if it is, a further test is carried out to ensure that it is in the range A to F. Note the use of relation conditions (Chapter 7) and numeric literals (Chapter 7). SUB is advanced each time a character has been checked.

In cases like this, where the same entry must be extracted from a table on several occasions, it is most efficient to move the factor once into a work field and to use this field for processing.

Although this is a simple example, the principle is an extremely valuable one which can save a large number of steps.

Scanning data for an end marker

REDEFINES can often be used with OCCURS to scan variable length paper tape records for an end marker. Non-significant zeros and spaces are frequently omitted when data is punched on paper tape, the items being separated instead by special characters.

For example, a paper tape record might consist of the following fields:

A code	:	6 characters
A quantity	:	1 to 6 characters
Newline	:	1 character

This record will therefore occupy between 9 and 14 characters in store (since newline is held as the two characters †*). The quantity digits in any record must be moved, right-justified, to a six-character numeric field for processing.

The program entries given below show how this task can be accomplished. The record is defined in the File Section, with a field CODE, and the remainder of the record held in REST. The paragraph COUNT—SIZE counts the number of digits in REST until it reaches the newline. The next paragraph moves the digits from REST one by one to the working—storage field DIGIT, by means of the two subscripts COUNT and SUB. When COUNT is exhausted, the program passes out of the loop. An alternative view of the working—storage field is QTY PIC 9(6) which can be used in arithmetic.

```
01  VAR-REC.
    02 CODE PIC 9(6).
    02 REST PIC X OCCURS 8.
       }                 }
WORKING-STORAGE SECTION.
01  COUNT PIC 9.
01  SUB PIC 9.
01  FILLER.
    02 DIGIT PIC 9 OCCURS 6.
    02 QTY REDEFINES DIGIT PIC 9(6).
       }                 }
PROCEDURE DIVISION.
       }                 }
    MOVE ZEROS TO QTY.
    MOVE 2 TO COUNT MOVE 6 TO SUB.
COUNT-SIZE.
    IF REST(COUNT) = "↑" GO TO SIZE-KNOWN.
    ADD 1 TO COUNT GO TO COUNT-SIZE.
SIZE-KNOWN.
    SUBTRACT 1 FROM COUNT.
    MOVE REST(COUNT) TO DIGIT(SUB).
    SUBTRACT 1 FROM SUB
    IF COUNT NOT = 1 GO TO SIZE-KNOWN.
```

This is a simple example. However, records containing more than one variable length field may not require many more statements since one routine could be used for all numeric items by means of the PERFORM statement or a subscript loop. A second routine would be needed for variable length alphanumeric items.

In the last two examples COBOL is forced to work at the character level. PLAN, however, is a more suitable language for such processing. ICL provides standard subroutines written in PLAN to handle records containing variable length items. See Chapter 10, "variable length data handling".

Chapter 7 The procedure division

As explained in Chapter 4, the Procedure Division contains the actual steps which inform the computer what operations or procedures are to be executed on data described in the Data Division. It has a hierarchial structure similar to the Data Division, in that it is organised into sentences, statements and paragraphs. The interaction of these operations with files, records and fields produces the required results.

STRUCTURE

The most elementary unit of a procedure is a statement introduced by a COBOL verb. This consists of a single logical operation, which the compiler performs upon with the help of data names, literals or other operands. For example,

```
ADD A TO B
GO TO START
```

One or more statements terminated by a full stop forms a COBOL sentence. These statements could be separated by commas or semi-colons, as for example,

```
ADD A TO B; GO TO START.
```

This constitutes a sentence, and is written starting in column 12. In general, for simplicity of program testing and correction, single-statement sentences are advisable where logically possible. For the same reason, each sentence should start on a new line.

Sentences are grouped together to form paragraphs, and the Procedure Division must contain at least one paragraph. Each paragraph begins with a paragraph name, which is either a data name or is numeric and is written beginning in column 8. The number of paragraphs in a program, and the way in which they are grouped depends upon the logical structure of the program. In COBOL it is only possible to branch to a paragraph name; therefore any sentence to which a branch is made must come first in a paragraph. A paragraph, then, usually consists of a number of sentences which are connected and performed as a whole.

If the program is segmented, the Procedure Division must be further divided into sections according to the rules given in Chapter 13.

This chapter deals with the fundamental processing verbs of the Procedure Division, that is, those which allow basic arithmetic operations and movement of data within the store. Only the most useful facilities are described here; more complex procedures are given in Chapter 13. The verbs are described under the following headings:

Basic verbs

These verbs undertake primary functions of arithmetic such as ADD and DIVIDE, and basic operations in the program, such as the movement of data and stopping of the program.

Sequence control

These verbs include conditions, where the course of action depends on the result of the condition, and other verbs which control the sequence of program steps, such as PERFORM and GO TO.

Peripheral verbs

These regulate input and output data to and from peripheral devices. They also enable files of data to be opened for processing or closed after processing.

Miscellaneous verbs

These perform miscellaneous functions not included under the headings.

Before the verbs are described, a section is included on literals and figurative constants, which rarely occur in the Data Division, but are frequently used as operands in the Procedure Division.

RULES FOR WRITING THE PROCEDURE DIVISION

In addition to the general rules for writing COBOL given at the end of Chapter 4, there are specific rules for writing the Procedure Division as follows:

1 The heading PROCEDURE DIVISION must be written beginning in column 8 and terminated by a full stop.

2 The next entry must be the first paragraph name (or section name if the program is segmented).

3 All paragraph names must be written on a new line starting in column 8 and termined by a full stop. Paragraph names must be unique, and must be either numeric or data names devised according to the rules for data names in Chapter 4.

4 The first sentence of a paragraph may be either written on a new line starting in column 12, or on the same line as the paragraph name. In this case, at least one space must follow the full stop at the end of the paragraph name.

5 Sentences must be written between columns 12 and 72, and may if necessary extend to a second line. If a word is split over two lines, a hyphen must be written in column 7. Each sentence must be followed by a full stop.

6 Each sentence may begin on a new line in column 12, or may follow the previous sentence on the same line. At least one space must follow the full stop at the end of the previous sentence. However, it is recommended that each sentence should begin on a new line.

7 Statements grouped within a sentence may optionally be separated by commas or semi-colons, for readability. There must be at least one space after a comma or semi-colon. Sentences consisting of one statement only are recommended.

LITERALS

Because data in COBOL is constantly changing, it is generally referred to by data names. However, it is also possible to refer directly to data which remains the same throughout the program, by creating a constant of the required value. For example, in a payroll application, a factor of 1.5 used in calculating overtime pay need not be given a data name but can be referred to directly. In the statement

MULTIPLY 1.5 BY EXCESS-HOURS.

1.5 is an example of a *literal*. Literals can be written directly into the Procedure Division, or alternatively, they may be used with the VALUE clause in the Data Division. Three types of literal exist in COBOL. *Numeric literals* consist of numeric values and *non-numeric literals* consist of any character string; *figurative constants* are a third type of literal, where certain symbols are given fixed data names.

Numeric literals

A numeric literal must consist of characters taken from the digits 0 to 9, the decimal point and the minus sign. It must contain at least one digit, but must not exceed 120 digits in length. Where a numeric literal is involved in arithmetic, it must not exceed 18 digits.

If no sign is given the literal is assumed to be positive. The negative sign, if used, must appear in the left hand position, with no spaces between the sign and the first digit. A decimal point may appear anywhere except as the right hand character and is treated as an implied decimal point. This means that the decimal point is not stored as a character, but is taken into account when the literal is used in calculation.

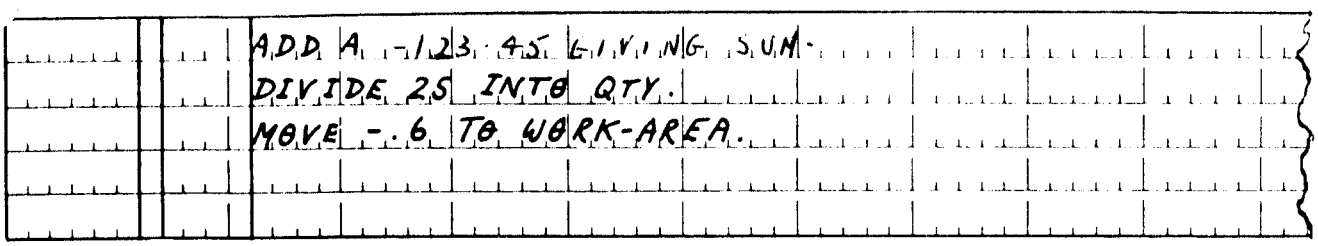
The following are examples of valid numeric literals:

```
  4019
  1.25
     7
 0.620
-9.55
```

The following are examples of invalid numeric literals:

```
123.   (must not end with decimal point)
- 10   (no space allowed after sign)
£370   (£ sign not allowed)
```

Example



Non-numeric literals

A non-numeric literal is a character string which must be enclosed in quotation marks. The character string may consist of any 1900 character, including numeric and alphabetic characters, spaces and symbols, with the exception of quotation marks. The maximum length for non-numeric literals is 120 characters, excluding the surrounding quotation marks.

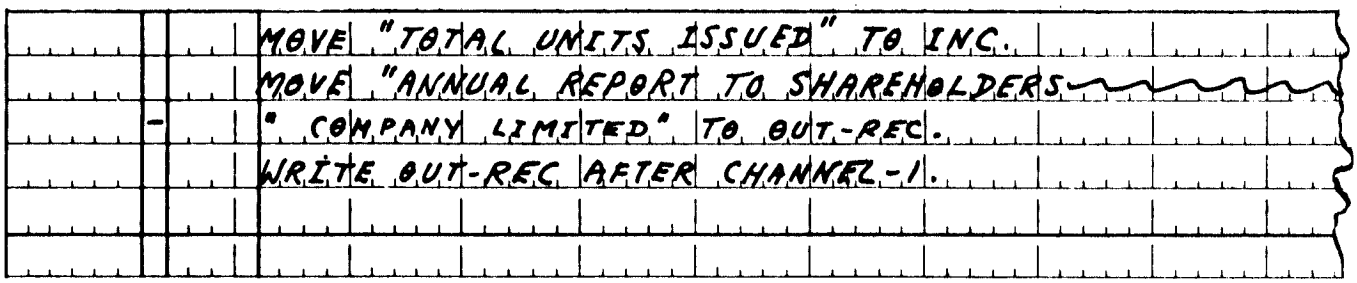
When a literal extends over more than one line of the coding sheet, a continuation indicator must be entered in column 7 of the second, and any subsequent, line. In this case, quotation marks must appear in column 12 of the second line, and the first character following marks the start of the continuation. No quotation mark is needed at the end of the previous line.

The following are examples of non-numeric literals:

"INCOME TAX CALCULATION"
"MAN—HOURS 24"
"—2.73"
"(SUB—TOTAL)"

It can be seen from the above examples that numeric and non-numeric literals can have the same appearance except for the quotation marks. It is essential not to confuse the two; —12.5 would be stored in a totally different way from "—12.5".

Examples



Non—numeric literals are often used to output headings on the line printer, as shown in the second example. See *WRITE,* later in this chapter.

Figurative constants

Certain constants used frequently in programs have been assigned fixed data names. Such constants are called *figurative constants*. The most commonly used of these are ZEROS and SPACES.

ZEROS represents one or more of the character 0, depending on the context, and is used to replace a numeric constant. The statement

MOVE ZEROS TO TOTAL.

where TOTAL is a five-character field, is equivalent to

MOVE 00000 TO TOTAL.

SPACES represents one or more spaces depending on the context; it replaces a non-numeric literal " " of any length. The statement

MOVE SPACES TO OUT—FIELD.

is the same as

MOVE "▽▽" TO OUT—FIELD.

where OUT—FIELD is two characters long.

These constants are used to zeroise the contents of a field, or to space-fill a field. When ZEROS or SPACES are moved to or compared with an item, the character is always repeated so as to be equal to the size of the item. However, when they are not associated with another item, as for example when used with DISPLAY a single character is generated.

BASIC VERBS

Basic verbs in COBOL include arithmetic verbs, and verbs performing other primary functions.

The following verbs provide the means of performing arithmetic in COBOL:

ADD
SUBTRACT
MULTIPLY
DIVIDE
COMPUTE

Some general rules should be noted. Arithmetic can be performed in COBOL only on elementary fields. Each data name must refer to an unedited numeric field, except that the result field may be edited. Each literal must be a numeric literal. In general, if a field receives the result of an arithmetic (or MOVE) operation, its previous contents are lost; all other operands are unchanged.

The ADD verb

There are three forms of the ADD statement, as follows:

1 ADD {data-name-1 / literal} TO data-name-2.

This format adds the contents of data-name-1 or the literal to the contents of data-name-2 and stores the result in data-name-2. The previous contents of data-name-2 are overwritten. The second operand may not be a literal as it will store the result.

Examples

ADD OVERTIME TO TOTAL-HOURS.

	OVERTIME	TOTAL HOURS
Before	1 3	4 0
After	1 3	5 3

ADD 14 TO STOCK.

	STOCK
Before	1 2 5
After	1 3 9

2 ADD {data-name-1 / literal-1} {data-name-2 / literal-2} GIVING data-name-3.

This format adds two data names or literals and stores the result in a further field, specified after GIVING. Data-name-1 and data-name-2 remain unchanged, but the contents of data-name-3 are overwritten. The GIVING option is used when it is required to form the sum of two items yet retain each item in store.

Examples

ADD OVERTIME HOURS GIVING TOTAL–HOURS.

	OVERTIME	HOURS	TOTAL–HOURS
Before	1 3	4 0	1 1 1
After	1 3	4 0	0 5 3

ADD COSTS 330 GIVING EXPENSES.

	COSTS	EXPENSES
Before	4 0 0	0 0 9
After	4 0 0	7 3 0

Both the above formats can be used to add more than two operands.

ADD $\left\{\begin{matrix}\text{data–name–1}\\\text{literal–1}\end{matrix}\right\}\left\{\begin{matrix}\text{data–name–2}\\\text{literal–2}\end{matrix}\right\}\cdots\left\{\begin{matrix}\text{data–name–n}\\\text{literal–n}\end{matrix}\right\}$

TO data–name–x.

This adds all the given data names or literals, storing the result in data–name–x.

ADD $\left\{\begin{matrix}\text{data–name–1}\\\text{literal–1}\end{matrix}\right\}\left\{\begin{matrix}\text{data–name–2}\\\text{literal–2}\end{matrix}\right\}\cdots\left\{\begin{matrix}\text{data–name–n}\\\text{literal–n}\end{matrix}\right\}$

GIVING data–name–x.

This adds data names and literals (except data–name–x) which are stored in data–name–x.

The maximum number of multiple operands which can be handled in one ADD (or SUBTRACT) statement is nine. Although the use of multiple operands will give shorter coding in the source program, it may not save steps in the object program. This point also applies to many other COBOL facilities.

Examples

ADD EXPENSES FEES 25 TO WAGES.

	EXPENSES	FEES	WAGES
Before	3 0 0 0	1 2 0	5 0 0 0
After	3 0 0 0	1 2 0	8 1 4 5

ADD 10 COSTS FEES GIVING EXPENSES.

	COSTS	FEES
Before	0 6 1 2 0 1	2 0 0 0 0 0
After	0 6 1 2 0 1	2 0 0 0 0 0

	EXPENSES
Before	1 2 3 4 5 6
After	2 6 1 2 1 1

3 *Less commonly, a third format of ADD can be used.*

ADD $\left\{\begin{matrix}\text{data-name-1}\\\text{literal-1}\end{matrix}\right\}\left\{\begin{matrix}\text{data-name-2}\\\text{literal-2}\end{matrix}\right\}$ ... data-name-n.

This adds together all the operands and stores them in the last operand. It operates in exactly the same way as format 1.

For example,
ADD COST-PRICE PURCHASE-TAX 30 PROFIT.
All the items are added and the result stored in PROFIT.

The SUBTRACT verb

There are two forms of the SUBTRACT statement, as follows:

1 SUBTRACT $\left\{\begin{matrix}\text{data-name-1}\\\text{literal-1}\end{matrix}\right\}$ FROM data-name-2.

This format subtracts the contents of data-name-1 or the literal from data-name-2, and stores the result in data-name-2. The second operand may not be a literal since it will store the result.

Examples
SUBTRACT 1 FROM COUNT.

	COUNT
Before	3 0
After	2 9

SUBTRACT OVERTIME FROM TOTAL-HOURS.

	OVERTIME	TOTAL-HOURS
Before	1 3	5 0
After	1 3	3 7

2 SUBTRACT $\left\{\begin{matrix}\text{data-name-1}\\\text{literal-1}\end{matrix}\right\}$ FROM $\left\{\begin{matrix}\text{data-name-2}\\\text{literal-2}\end{matrix}\right\}$

GIVING data-name-3.

This format subtracts one operand from another and stores the result in a third field, whose contents are overwritten. Data-name-1 and data-name-2 remain the same.

Examples

SUBTRACT EXCISE FROM PROFIT GIVING FINAL–PROFIT

	EXCISE	PROFIT	FINAL–PROFIT
Before	2 0 0	9 0 0	0 2 0
After	2 0 0	9 0 0	7 0 0

SUBTRACT COSTS FROM 313 GIVING PROFITS.

	COSTS	PROFITS
Before	4 0 1	9 9 9
After	4 0 1	–0 8 8

Note that in the final example the result (–088) is stored as a negative quantity. In this case the result field should have had an S in its Picture.

Like the ADD statement, SUBTRACT can be used to handle multiple operands.

SUBTRACT {data–name–1 / literal–1} {data–name–2 / literal–2} ... {data–name–n / literal–n}

FROM data–name–x.

This adds together all the data–names and literals before FROM, and subtracts them from data–name–x, in which the result is stored.

SUBTRACT {data–name–1 / literal–1} {data–name–2 / literal–2} ... {data–name–n / literal–n}
FROM data–name–x GIVING data–name–y.

This adds together all the data names and literals before FROM, and subtracts them from data–name–x. The result is stored in data–name–y and data–name–x remains the same.

Examples

SUBTRACT 15 TAX GRAD–PENS NAT–INS FROM GROSS–PAY.

	TAX	GRAD–PENS	NAT–INS	GROSS–PAY
Before	3 0 0	1 2 0	1 5 0	3 0 0 0
After	3 0 0	1 2 0	1 5 0	2 4 1 5

SUBTRACT 15 TAX GRAD–PENS NAT–INS FROM GROSS–PAY GIVING NET–PAY.

	TAX	GRAD–PENS	NAT–INS	GROSS–PAY	NET–PAY
Before	3 0 0	1 2 0	1 5 0	3 0 0 0	1 2 3 4
After	3 0 0	1 2 0	1 5 0	3 0 0 0	2 4 1 5

The MULTIPLY verb

The MULTIPLY statement has two formats.

1 MULTIPLY {data-name-1 / literal-1} BY {data-name-2 / literal-2}

GIVING data-name-3.

This format multiplies two data names or literals and stores the result in data-name-3. The contents of data-name-3 are overwritten, but data-name-1 and data-name-2 remain the same.

Examples

MULTIPLY 12 BY CODE GIVING NUMBER.

	CODE	NUMBER
Before	3	5 1
After	3	3 6

MULTIPLY VAL BY QUANTITY GIVING TOTAL.

	VAL	QUANTITY	TOTAL
Before	1 1	1 0	3 2 1
After	1 1	1 0	1 1 0

2 MULTIPLY {data-name-1 / literal-1} BY data-name-2.

Two data names are multiplied, and the result stored in the second.

Examples

MULTIPLY LENGTH BY BREADTH.

	LENGTH	BREADTH
Before	1 2 0	0 1 6
After	1 2 0	1 9 2 0

MULTIPLY cannot handle multiple operands.

The DIVIDE verb

The DIVIDE statement has four formats.

1 DIVIDE {data-name-1 / literal-1} INTO {data-name-2 / literal-2}

GIVING data-name-3.

One data name or literal is divided into another, and the result stored in data-name-3. Data-name-1 and data-name-2 remain the same, but the contents of data-name-3 are overwritten.

Examples
DIVIDE SPEED INTO DISTANCE GIVING TIME.

	SPEED	DISTANCE	TIME
Before	1 3	3 9	8 0
After	1 3	3 9	0 3

DIVIDE LENGTH INTO AREA–1 GIVING BREADTH.

	LENGTH	AREA–1	BREADTH
Before	1 2	1 2 5	9 7
After	1 2	1 2 5	1 0

Note in the second example that the remainder is discarded, and can only be obtained by calculating AREA–1–(LENGTH X BREADTH).

2 DIVIDE {data–name–1 / literal–1} INTO data–name–2.

Data–name–1 or literal–1 is divided into data–name–2, and the result stored in data–name–2.

Example
DIVIDE 12 INTO NUMBER.

	NUMBER
Before	1 5 2
After	0 1 2

DIVIDE cannot handle multiple operands.

DIVIDE ... REMAINDER

3 DIVIDE {data–name–1 / literal–1} INTO {data–name–2 / literal–2} GIVING

data–name–3 [REMAINDER data–name–4].

Data–name–1 or literal–1 is divided into data–name–2 or literal–2. The result is stored in data–name–3 and the remainder in data–name–4.

Example
DIVIDE 2 INTO B GIVING RES REMAINDER REM

	B	RES	REM
Before	1 3	0 0	1 2
After	1 3	0 6	0 1

also:–

4 DIVIDE {data–name–1 / literal–1} BY {data–name–2 / literal–2} GIVING

data–name–3 [REMAINDER data name–4].

The COMPUTE verb

The compute verb combines several arithmetic statements together.

COMPUTE data-name-1 = { data-name-2 / literal-1 / arithmetic expression }

For example
COMPUTE A = B + C
would be the same as
ADD B C GIVING A.

The arithmetic expression may be made up of:—

() brackets
** exponentiation
* multiplication
/ division
+ addition
— subtraction
and are evaluated in that order.

It is most useful when calculating square roots

Example
COMPUTE A = B ** .5

	A	B
Before	1 5	2 5
After	0 5	2 5

Compiler action with arithmetic verbs

All arithmetic verbs can be used on fields in either character or binary form. The maximum length of an operand is eighteen decimal digits.

Because the 1900 Series has a binary adder, the compiler must include in the object program steps to convert character operands to binary before processing; further steps must convert a binary result to characters if the receiving field is in character form.

If a field is to be used for arithmetic several times, it is uneconomical to convert it each time it is processed. The number of conversions can be reduced by holding in binary form each factor which is used in more than one arithmetic statement. See Chapter 9 "SYNCHRONIZATION" for details.

Any numeric field can have one or more decimal places, where the position of the decimal point (or binary point for binary fields) is specified in the Data Division (See Chapter 6 "THE PICTURE CLAUSE"). This information enables the compiler to generate an object program in which fields are aligned at the decimal point. The decimal point does not, however, take up space in store.

A negative number is usually held in absolute form with a sign. This sign may either be in a separate character position on the left, or it may be combined with the most significant digit, according to the usage of the item. See Chapter 6 "Usage clause".

Two further options are available for use with all arithmetic verbs, as described below.

The ROUNDED clause

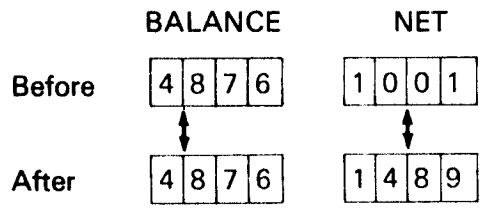
ROUNDED can be used after any arithmetic verb to prevent the loss of excess digits in a result field. If a result field includes more decimal places than its Picture in the Data Division, excess digits are lost or truncated. When the ROUNDED option is used, the result is rounded up to the nearest integer or to a required decimal place.

Rounding is accomplished by adding 5 to the right hand digit which will be lost and then truncating the digit. For example if 618.89 had been stored in an area described as having only one decimal place, 5 would be added to the item to make it 618.94. The 4 would then be lost and the item stored as 618.9.

Examples

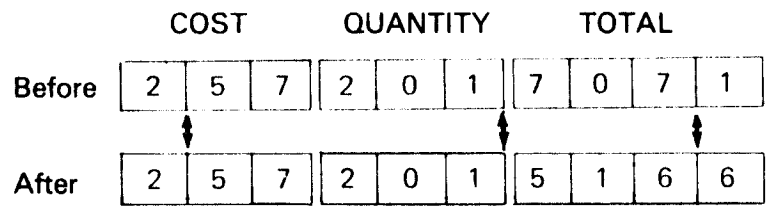
In these examples, the assumed decimal point is indicated by an arrow.

ADD BALANCE TO NET ROUNDED.



If the ROUNDED option were not used, the result would be 14.886 which would be truncated to 14.88.

MULTIPLY COST BY QUANTITY GIVING TOTAL ROUNDED



Without the ROUNDED option, the result would be 516.57, truncated to 516.5.

The SIZE ERROR clause

The SIZE ERROR option has a function similar to the ROUNDED option. It allows the user to specify action if the result of an arithmetic statement contains more digits to the left of the decimal point than are specified in the area assigned to store it. If this happens, a *size error* occurs and the most significant digits may be lost or the program mutilated.

The clause has the format

ON SIZE ERROR

which is written after the last operand of an arithmetic verb (or after ROUNDED if this is present). It is followed by an imperative statement, normally a GO TO statement, see later in this chapter, leading to a routine to deal with a size error if this occurs. When the arithmetic statement has been performed, SIZE ERROR causes the number of

characters in the result to be checked against the area designed to hold it. If the area is too small, the imperative statement is obeyed and a branch is made to an error routine. In this case, no answer will be obtained in the result field, and the original contents of the operands will remain unchanged.

It is advisable to use a SIZE ERROR clause if the size of the result field is undetermined. However, it is better practice to ensure that the result field is large enough.

Note that SIZE ERROR only checks for truncation on the left; if ROUNDED is not used as well, any loss of decimal places will be ignored.

Examples

```
MULTIPLY COST BY NUMBER SIZE ERROR GO TO CORRECT.
```

	COST			NUMBER	
Before	2	5		5	0
After	2	5	1	2	5

Here two quantities are multiplied, and two characters of store are allocated to the result. However, as the product exceeds two digits, the most significant digit of 125 cannot get into NUMBER. When this situation occurs, SIZE ERROR causes a branch to a routine called CORRECT which will attempt to deal with the situation.

The following verbs, MOVE, STOP and DISPLAY, are also fundamental to COBOL. They move data from one area to another, stop the program and display small amounts of data on the console.

The MOVE verb

The movement of data from one place in to another is one of the most common operations in a program. The MOVE verb is used specifically for this purpose.

The simplest form of the MOVE statement is as follows:

$$\underline{\text{MOVE}} \quad \left\{\begin{matrix}\text{data-name-1}\\ \text{literal}\end{matrix}\right\} \quad \underline{\text{TO}} \ \text{data-name-2.}$$

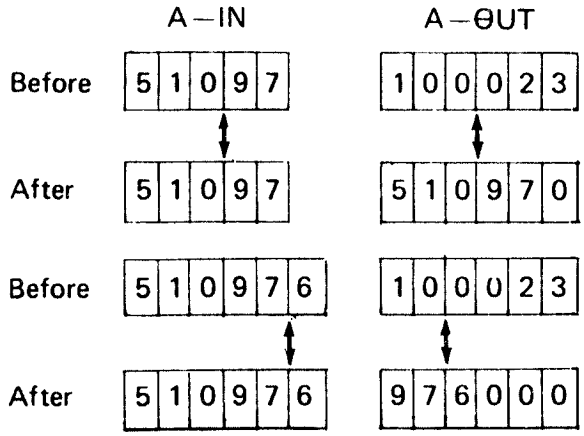
This moves the contents of data—name—1 or the literal to a receiving area, data—name—2. After MOVE is executed the contents of the receiving area is overwritten, but the source field remains as before.

As shown in Chapter 6, data transferred from a source field attempts to take on the form specified in the receiving field. The contents of the receiving field are determined by the Picture of the source field, and by the editing and usage of the receiving field.

In its simplest form, the MOVE statement transfers data between elementary fields of the same class and usage.

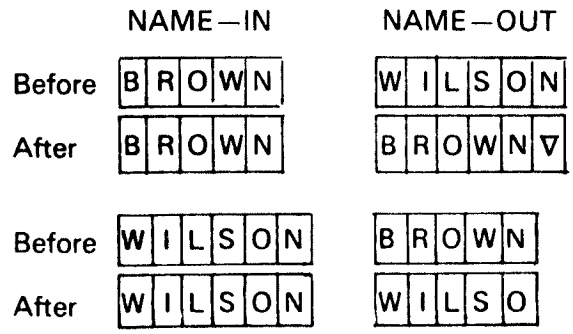
When a transfer is made between *numeric* fields, the decimal points of the source and receiving fields are aligned. The character immediately before the point in the source field is aligned with the character before the point in the receiving field. If the receiving field is too small, excess digits to left or right are truncated. If the receiving field is larger than the source field, the spare positions are zero-filled. If both fields are signed, the sign is placed in the leading character position of the receiving field.

Examples
In the following examples, the assumed decimal point is indicated by an arrow.
MOVE A–IN TO A–0UT.



When a transfer is made between *alphanumeric* or *alphabetic* fields, the characters of the source field are moved into the corresponding positions in the receiving field, starting with the left hand position. If the source field is shorter than the receiving field, spare character positions are space-filled. If it is longer than the receiving field, excess characters in the source field are ignored.

Examples
MOVE NAME–IN TO NAME–OUT.



In addition to moving an item of data from one area of storage to another, MOVE can also be used with multiple destinations. Its format is as follows:

$$\underline{\text{MOVE}} \left\{\begin{matrix}\text{data–name–1}\\ \text{literal}\end{matrix}\right\} \underline{\text{TO}} \text{ data–name–2 data–name–3...} \text{ data–name–n.}$$

Data–name–1 or the literal is moved to two or more areas, data–name–2, data–name–3 and so on. The item can then be accessed in any of these areas. For example,

MOVE SPACES TO LP–QTY LP–PRICE LP–RATE.

will fill LP–QTY, LP–PRICE and LP–RATE with spaces.

The most common use of MOVE, as described here, is between elementary fields. It may however be necessary to move data between group fields, or from group fields to elementary fields. Similarly, data is normally moved between fields of the same type, but it is also possible to move, say, a numeric to a binary field, or a field with one type of usage to another. The rules for all these types of movement are complicated, and are set out in detail in Chapter 10 of *COBOL*.

The STOP verb

This verb causes the program to halt. Its format is

$$\underline{\text{STOP}} \left\{\begin{matrix}\underline{\text{RUN}} \\ \text{literal}\end{matrix}\right\}$$

STOP RUN should always be the last obeyed statement in the object program, and brings the program to a permanent halt. A check is made that all files are closed before the program is suspended, and the message

HALTED: — END OF RUN

appears on the console. The program can then be restarted only at the beginning.

STOP literal causes a temporary halt of the program, while waiting for operator action. The literal is communicated to the operator on the console. It may be numeric or non-numeric and may have a maximum of forty characters (including spaces). For example,

STOP "ERROR DETECTED".

When the operator types GO, the program continues with the statement following STOP.

Messages should be kept as short as possible. When a long pause is necessary, the program should print a short message referring the operator to the operating instructions where the message is given in full. Undue use of the console can make the speed of the computer operator—dependent.

The DISPLAY verb

The DISPLAY verb is used to output small amounts of data. It allows literals or the contents of fields to be typed out on the console without halting the program, and is of particular use in program testing.

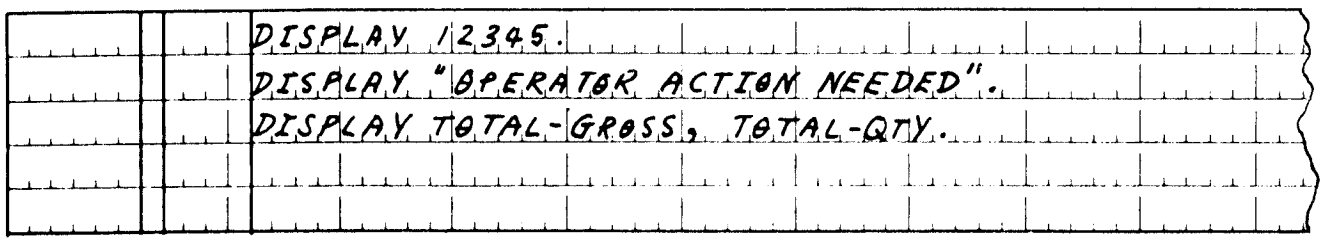
DISPLAY can take two forms, as follows:

1 $\underline{\text{DISPLAY}} \left\{\begin{matrix}\text{data-name-1} \\ \text{literal-1}\end{matrix}\right\} \left\{\begin{matrix}\text{data-name-2} \\ \text{literal-2}\end{matrix}\right\} \cdots \left\{\begin{matrix}\text{data-name-n} \\ \text{literal-n}\end{matrix}\right\}$

This is the simplest format of DISPLAY, which is followed either by field names, literals or a combination of both. Literals may be numeric or non-numeric. This information is then displayed on the console. The maximum number of characters (including spaces) which can be printed by one DISPLAY statement is forty.

Fields are displayed in the same order as in the DISPLAY statement, on the same line and without spacing, unless this has been inserted by a figurative constant or non-numeric literal.

Examples



If TOTAL—GROSS and TOTAL—QTY contained 999 and 111 respectively, the following message would appear:
999111

```
DISPLAY TOTAL-GROSS " " TOTAL-QTY.
DISPLAY TOTAL-GROSS, SPACES, TOTAL-QTY.
```

would both display
999 111

```
DISPLAY "TOTAL IS " TOTAL-QTY.
```

would display
TOTAL IS 111.

2 This is an extended form of the DISPLAY statement.

DISPLAY data—name UPON mnemonic—name.

The data name, whose contents may exceed one word in length, is displayed upon mnemonic—name; this identifies a peripheral defined in the Special—Names Paragraph of the Environment Division. As described in Chapter 5 "SPECIAL—NAMES PARAGRAPH", peripherals can be given names by the statement

PRINTER
CARD—PUNCH
PAPER—PUNCH } integer [TYPE type—number]

IS mnemonic—name.

When mnemonic—name is specified after DISPLAY, data is output on the appropriate peripheral. Unlike ACCEPT, DISPLAY does not release the peripheral. Consequently mnemonic—name can refer to a peripheral which has been assigned to a file. DISPLAY can then be used before the file is opened but not after it has been closed, since the CLOSE statement releases the peripheral.

Example

```
SPECIAL-NAMES.
    PRINTER 1 IS TEST-PRINTER.
FILE-CONTROL.
    SELECT PRINTOUT ASSIGN TO PRINTER 1.
    {                         }
    {                         }
    DISPLAY RESULT UPON TEST-PRINTER.
```

The above example prints test results on a printer which is also allocated to a file. A file PRINTOUT is allocated to PRINTER 1, which is given the name TEST—PRINTER in the Special—Names Paragraph. Before the file is closed the contents of RESULT are displayed upon TEST—PRINTER.

DISPLAY WITH USAGE

The following rules apply when DISPLAY is used on fields for which usage clauses have been specified.

A COMP field is not displayed in binary form, but as if it had been defined by DISPLAY—3, as a decimal number with a preceding sign if required.

Group fields to be displayed must start on a 1900 Series character boundary and must be a whole number of characters long. Each character is displayed separately and is treated as alphanumeric.

The ACCEPT verb

The ACCEPT verb provides a method of inputting a small amount of data which varies from one run to another, such as a week number or period number.

There are two formats of the ACCEPT statement:

1 ACCEPT data—name.

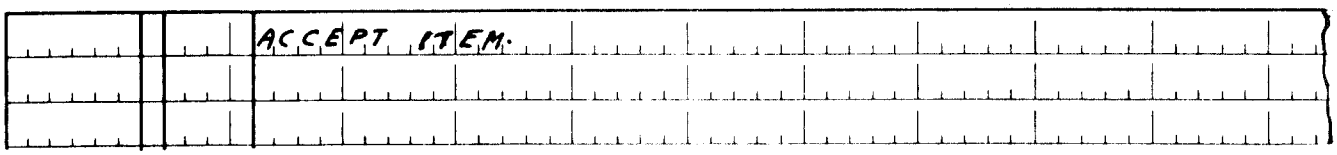
This format causes data to be read from the console and stored in the area defined by data—name. This area cannot be greater than one word in length. The data accepted can either be four alphabetic characters or a decimal integer in the range 0 to 8,388,607, depending on whether the receiving field is defined as alphabetic or numeric. It is moved according to the rules for alphanumeric items or numeric literals.

When an ACCEPT statement is encountered in a program, the message

COBOL ACCEPT (SEE JOB SHEET)

is typed out on the console. The program halts and the operator types in the information requested followed by GO. The data is then stored in data—name.

For example,



2 ACCEPT data—name FROM mnemonic—name.

Receives data from a specified peripheral identified by mnemonic—name. The peripheral is then released by the program. Data may exceed one word in length. If the data is shorter than the area defined by data—name, it is padded out to the required length with spaces in the usual way.

The mnemonic name is associated with the peripheral in the Environment Division, and will have been specified in the Special—Names paragraph by the entry

$$\left\{\begin{matrix}\text{CARD READER}\\ \text{PAPER—READER}\end{matrix}\right\} \text{ integer [TYPE type—number] IS mnemonic—name.}$$

Mnemonic—name is invented using the rules for data names, and must be unique within the program. TYPE allows a particular model of the peripheral to be specified.

Since the card reader cr paper tape reader is released after the data has been accepted, the integer given to it should not be the same as the number given to a device of that type in a SELECT ... ASSIGN statement. When there is only one card reader or paper tape reader, it should be given one number in the SELECT ... ASSIGN clause and a different number in the Special—Names paragraph. This will prevent the ACCEPT statement from closing the file.

Example
The following example gives the mnemonic name CARD—X to a card reader. Data is then accepted from CARD—X and is stored in the area defined by ITEMS. Note that the card reader in the Special—Names paragraph is given a different number from that assigned to a file CARD—IN in the file—Control paragraph.

```
SPECIAL-NAMES. CARD-READER 2 IS CARD-X.
FILE-CONTROL.
    SELECT CARD-IN ASSIGN CARD-READER 1.
PROCEDURE DIVISION.
    ACCEPT ITEMS FROM CARD-X.
```

This is a relic from George II days. Although it can still be used, it is no longer needed, so ignore

3 A variant of format 2 is
ACCEPT FILE KEYS FROM mnemonic—name.
This is used to read disc or magnetic file generation numbers from a card reader (or paper tape reader) punched in a format described in *COBOL,* Chapter 12. This is an alternative to the operator typing in the data from the console.

SEQUENCE CONTROL

It is often necessary in a program to depart from the sequence of events as written. This process can either depend on the result of some condition, or it may constitute a branch to a procedure or series of procedures in another part of the program, before returning to the normal sequence. The principal verbs which enable the user to do these things are the IF statement and the PERFORM verb. Both of these are available with a number of options.

The IF statement

The IF statement provides the most common means of departing from the normal sequence of program operations. It should be noted that in COBOL, IF is treated as a verb, as some kind of action is always called for when it appears in the program.

A vital feature of any programming language is the ability to test items of data and to select, as a result of this test, one of several routes through the program. This facility is provided by conditional statements introduced by IF.

An IF statement has the format
IF condition imperative statement(s).

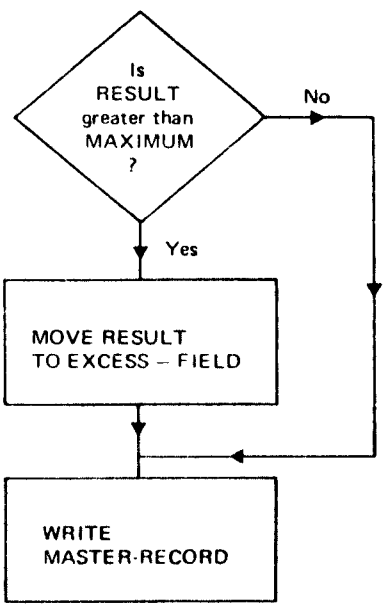
When the statement is encountered in the program, the next step or steps are determined by whether the condition is satisfied. If it is satisfied, control passes to the imperative statement, which is then performed. Afterwards, control is returned to the statement following the IF statement in the program. However, if the condition is not satisfied, the imperative statement is ignored and control passes straight to the sentence following the IF statement.

Conditions and imperative statements may take a variety of forms, as discussed in the next section. However, a simple example of the use of IF is given below.

Example
Consider the following Procedure Division statements:

```
ADD FIELD-A TO RESULT.
IF RESULT GREATER THAN MAXIMUM MOVE RESULT TO
   EXCESS-FIELD.
WRITE MASTER-RECORD.
```

After FIELD—A is added to RESULT, the condition
IF RESULT GREATER THAN MAXIMUM
has to be tested. The flowcharting for this condition would be shown thus:



If RESULT is greater than MAXIMUM, the imperative statement
MOVE RESULT TO EXCESS–FIELD
is executed. The program then executes the WRITE statement. If RESULT is equal to or less than MAXIMUM, the program ignores the imperative statement and passes directly to the WRITE statement.

Notice the importance of full stops in the example. When the condition is true, every statement up to the first full stop is obeyed. However, if the full stop after EXCESS–FIELD were omitted, WRITE MASTER–RECORD would also be interpreted as part of the conditions statement, and would be ignored if the condition were false.

Conditions

As stated above, several types of condition can be used after an IF statement to test the nature of a field of data. Some of the most common are considered here.

RELATION CONDITIONS

A relation condition makes a comparison between two given values. The format of the IF statement condition containing all possible relations is as follows:

```
IF { data-name-1 }  IS [NOT] { GREATER THAN }  { data-name-2 }
   { literal-1   }           { LESS THAN    }  { literal-2   }
   { Arithmetic  }           { EQUAL TO     }  { arithmetic  }
   { Expression  }           { >=           }  { expression  }
                             { <=           }
                             { =            }

   imperative
   statement
```

One value is tested to see if it is greater than, less than or equal to another. Relations may be written in full, but the relational operators ><and = are more often used as substitutes. There must be at least one space on either side of the operator; that is, = counts as a word. Both the values compared must be numeric or alphanumeric, but both cannot be literals.

A relation test of *numeric values* makes an algebraic comparison. The length of the item in terms of the number of digits is not significant; for example, −7 is greater than −43, and 24 is equal to 0024.

When a test is made of *alphanumeric values*, it is necessary to know whether the letter A is greater or less than the letter E, whether the digit 1 is greater or less than the letter A, and so on. This is determined by the fact that each character has an arithmetic equivalent, which is used to form a *collating sequence*. In 1900 COBOL, a character has a value greater than another if it appears later in the character code given in Appendix 2.

When one alphanumeric field is compared with another, the contents of the two fields are compared character by character, starting with the left hand character. If the first character of one field has a value greater than that of the other, the field to which it belongs has a greater value. If the two characters are the same, the next character in each field is compared and so on. Where one field is shorter than another and the other characters are equal, the shortest will be less in value.

For example
FRED: is greater than FRED

The following examples illustrate these rules:

JONES is greater than JOHNS
JAMES▽▽is less than JAMESON
ABCDEF is greater than ABC123
ABC123 is less than ABC456
123ABC is less than ABC123
1ABCDE is less than 2ABCDE
1ABCDE is less than A99999

If one of the operands is a literal, it is considered to be of the same class as the field with which it is compared, and the preceding rules therefore apply.

The values of conditions using relational operators are summarised in the table below.

	Value of condition		
Operator	*First operand greater*	*Second operand greater*	*Operands equal*
GREATER THAN >	true	false	false
NOT GREATER NOT >	false	true	true
LESS <	false	true	false
NOT LESS NOT <	true	false	true
EQUAL TO = EQUALS	false	false	true
NOT EQUAL TO NOT =	true	true	false

Examples
The following are examples of relation conditions used with IF in the Procedure Division.

```
IF EXPENSES < PROFITS SUBTRACT NET FROM
  GROSS.
IF DAYS LESS 28 MOVE PAY-IN TO PAY-OUT.
IF ITEM = REF-NO ADD BALANCE TO QTY
  MULTIPLY QTY BY PRICE GIVING VAL.
```

SIGN CONDITIONS

These are special cases of relation conditions, which determine whether a numeric item is positive, negative or zero. The format of a sign condition is as follows:

IF data—name IS [NOT] {POSITIVE / NEGATIVE / ZERO}

POSITIVE, NEGATIVE and ZERO are more economical ways of writing GREATER THAN, LESS THAN and EQUAL TO ZERO.

Examples

```
IF BALANCE NOT POSITIVE ADD ISSUES LOANS
   GIVING NEW-CASH.
IF A NEGATIVE MOVE ZEROS TO COUNT.
```

CLASS CONDITIONS

These are used to test whether an item contains a certain class of character only. This test may be carried out for alphabetic or numeric characters. A class condition has the following format.

IF data—name IS [NOT] {ALPHABETIC / NUMERIC}

The ALPHABETIC test is available for alphabetic, alphanumeric or group fields. The field is examined to see if it contains characters from the set A to Z and space. If it does, the value of the condition is true. Not alphabetic has the opposite effect; the condition is true only if any of the characters include A to Z and space. Group fields can only be tested if they start on a character boundary and occupy a whole number of characters.
For example, the following statement tests NAME to see if it is alphabetic.

```
IF NAME ALPHABETIC MOVE NAME TO ADVICE.
```

It could have been written if NAME had been defined in any of the ways shown below, in other words if it were an alphanumeric, alphabetic or group field.

```
02 NAME PIC X(25).
02 NAME PIC A(25).
02 NAME.
   03 INITIALS PIC X(5).
   03 SURNAME PIC X(25).
```

The NUMERIC test can be applied to a numeric or alphanumeric field. The field is examined to see if it contains only characters from the set 0 to 9, together with a possible sign. If it does, the value of the condition is true. NOT NUMERIC has the opposite effect; the condition is true if any characters are not in the range 0 to 9 and possible sign. If a field is defined as numeric, leading spaces are counted as zeros. If a field LENGTH contains

V	V	V	5

and is defined as

PIC 9(4)

it is taken as numeric. However, if it is defined as

PIC X(4)

then it would not pass the numeric test, as it does not contain all characters from the range 0 to 9. The statement

```
IF LENGTH NUMERIC MOVE LENGTH TO DETAILS.
```

can be applied to LENGTH, whether it is defined as numeric or alphanumeric.

Uses of class conditions

Class conditions are most commonly used for vetting data.

CONDITION NAME CONDITIONS

As described in Chapter 6 "condition names", a field called a *condition name* can be declared in the Data Division using an 88 level. This allows a name to be given to the required condition, that is, that the field contains one of a specified number of values. By means of the name, the truth of the condition can be tested in the Procedure Division using a normal IF statement.

The condition name is set up by defining the field to be tested and then using an 88 level immediately after the definition. The 88 level entry specifies the value or values which the field must contain if the condition is true. This is the only place where a VALUE clause may appear in the File Section. Several entries may be included if several ranges of values are required for the same field, so long as each entry is given is own name. The condition name written after the 88 level is used with the IF statement to test the condition. If one of the values appears in the field previously defined, the condition is true.

Example
A field, AREA—1 must contain one of a number of values at a certain point in the program, if a branch in the Procedure Division is to be made. The following statements set up a condition name, TEST, under AREA—1 in the Data Division, giving the required range of values. The statement

IF TEST GO TO CALC.

then tests the contents of TEST: if it contains one of the values specified, the condition is satisfied.

```
02 AREA-1 PIC 99.
   88 TEST VALUE 3 8 92 THRU 97.
02 RADIUS PIC 9V99.
     ≷       ≷
IF TEST GO TO CALC.
```

Note that separate values can be combined with a range specified by THRU. 92 THRU 97 indicates a consecutive range of numbers, from 92 to 97 inclusive.

The use of condition names saves program writing, but does not economise on object code.

SWITCH STATUS CONDITIONS

As explained in Chapter 5 "SPECIAL—NAMES .PARAGRAPH", switches in word 30 of the program area can be tested to see if they are in an ON or OFF state, by means of condition names. The format of a switch status condition is

IF condition name imperative statement

where the switch has been named in the Special—Names paragraph of the Environment Division. For example, suppose switch-x has been assigned the name FLAP for ON status. The Statement

IF FLAP GO TO ROUTINE—1.

will transfer control to ROUTINE—1 if the switch has been set to ON. If the switch is OFF the program continues in sequence.

GO TO is frequently used after a SIZE ERROR clause in an arithmetic statement (see earlier in this chapter); in this case it gives a branch to a routine set up to deal with the error condition, if the result of a calculation is too large to fit the field. For example,

```
MULTIPLY A BY B GIVING C SIZE ERROR
    GO TO CORRECT.
```

2 GO TO paragraph—name—1 paragraph—name—2... paragraph—name—n DEPENDING ON data—name.

This is used to transfer control to a paragraph depending on the value of a given item stored in data—name. Data—name must be an elementary item defined in the Data Division as an integer; it must contain a number in the range 1 to n, where n is the number of paragraphs referred to in the statement. GO TO will then cause a branch to paragraph—name—1 when the value in data—name is 1, to paragraph—name—2 when the value is 2, and so on. If the value is zero or exceeds n, the GO TO statement is ignored, and control passes to the next statement.

Paragraphs need not be referred to in the same order as they appear in the Procedure Division.

Example
Each record in a file contains a field giving a tax code in the range 1 to 5. According to the contents of the field in each record, the program will branch to one of five different paragraphs at a certain point. If the field contains any other value the program branches to an error routine. This is referred to in a second GO TO statement, which is executed if GO TO ... DEPENDING is ignored.

```
01  DAY-DATA.
    {   {
02  TAX-CODE PIC 9.
    {   {
PROCEDURE DIVISION.
    {   {
    GO TO TAX-1 TAX-2 TAX-3 TAX-4 TAX-5
      DEPENDING TAX-CODE.
    GO TO ERROR-1.
TAX-1.
    {
TAX-2.
    {
```

The IF ... ELSE statement

This is the IF ... ELSE format, which specifies two courses of action, one when the condition is satisfied and the other when it is not. In effect it works in much the same way as IF with GO TO, where the program either branches to another paragraph, or continues in sequence. However, IF ... ELSE gives two positive options, one of which must be followed before the program sequence is resumed.

The repetition of this sequence is inefficient, not only in terms of program writing, but also because identical machine code is incorporated into the object program each time the instructions are encountered.

The PERFORM verb eliminates these difficulties. A statement introduced by PERFORM enables the program to depart from its normal sequence, to execute one or more procedures written elsewhere, and then to return to the original sequence. A set of procedures need therefore be written only once, and referred to each time by means of PERFORM.

The PERFORM statement has three basic forms, which are as follows. A further more complex version of PERFORM is described under *processing multi-level tables*, later in this chapter.

1 PERFORM paragraph—name—1 [THRU paragraph—name—2].

This is the basic format of PERFORM, which permits execution of a single paragraph or set of paragraphs and returns control to the statement following the PERFORM statement. If a single paragraph is named, the program performs all the statements under paragraph—name—1. If, however, the THRU option is used, the program performs paragraph—name—1 and all paragraphs up to and including paragraph—name—2. This means that any number of paragraphs may be performed provided that they are sequential and that the last to be executed is named. Following the last statement in the last paragraph, the program returns the original statement following PERFORM. For example,

PERFORM PARA—A.

will execute all the statements in PARA—A, and

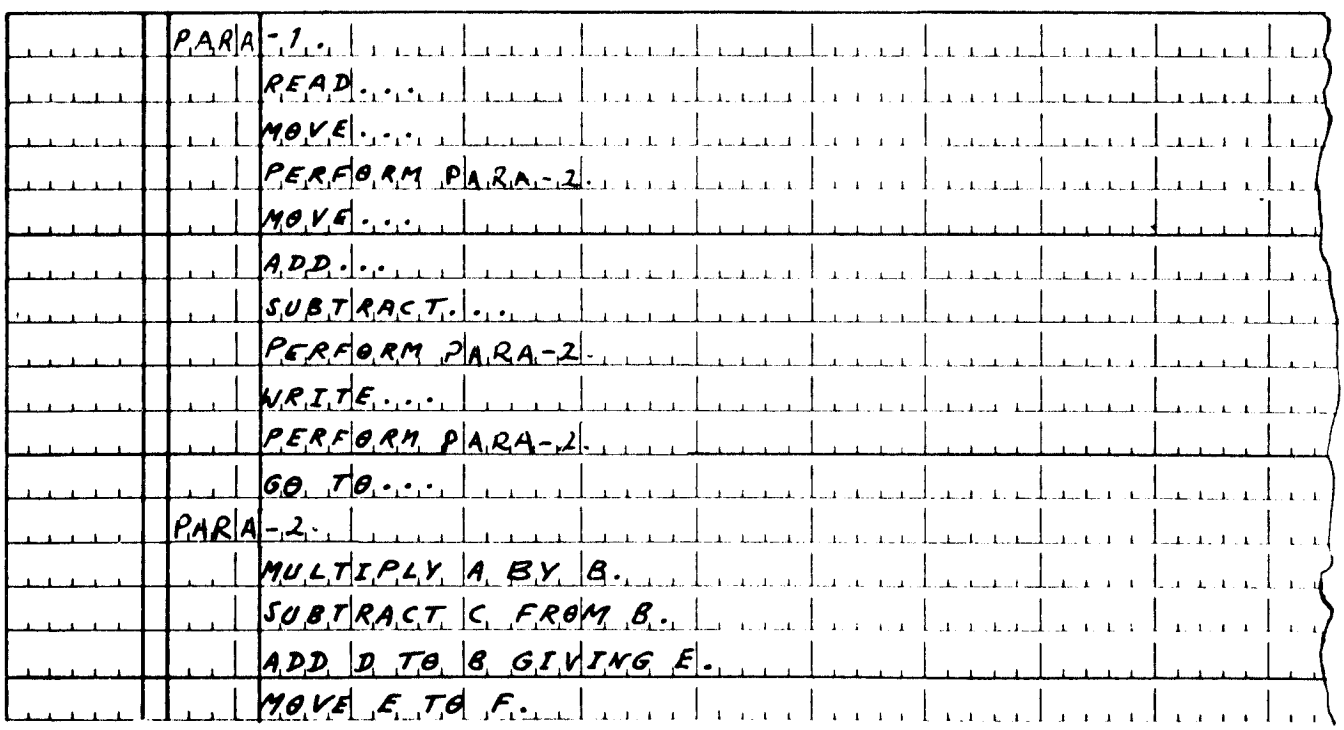
PERFORM PARA—A THRU PARA—G.

will execute all paragraphs up to and including PARA—G.

If necessary, the paragraphs referred to by PERFORM may be isolated at the end of the program; they need not form part of the normal program sequence.

Examples

In the following paragraph named PARA—1 it is required to perform a sequence three times.



This sequence is therefore placed in a separate paragraph called PARA—2 which is executed each time PERFORM PARA—2 is encountered. The program will first execute READ and MOVE, and will then perform PARA—2, before returning to the following MOVE instruction, and so on.

In the next example, the THRU option is used to perform several paragraphs.

```
           PERFORM ORDER-ROUTINE THRU ISSUE-ROUTINE.
           MOVE QUANTITY TO NEW-BALANCE.
               ⋮               ⋮
       ORDER-ROUTINE.
           ADD...
           MOVE...
       TRANSACTION.
           MULTIPLY...
           MOVE...
           WRITE...
       RECEIPT-ROUTINE.
           ADD...
           SUBTRACT...
           MOVE...
       ISSUE-ROUTINE.
           MOVE...
           WRITE...
```

When the PERFORM statement is encountered, the program executes the paragraph ORDER—ROUTINE, and the paragraphs TRANSACTION, RECEIPT—ROUTINE and ISSUE—ROUTINE, until it reaches the last statement in ISSUE—ROUTINE. When this has been performed, a return is made to the statement

MOVE QUANTITY TO NEW—BALANCE

2 <u>PERFORM</u> paragraph—name—1 [<u>THRU</u> paragraph—name—2]

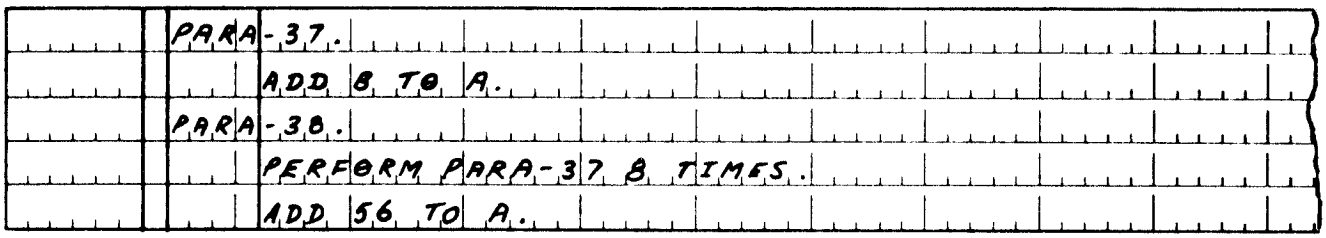
$\left\{\begin{array}{l}\text{integer}\\ \text{data—name}\end{array}\right\}$ <u>TIMES.</u>

This enables one or more paragraphs to be performed a given number of times, by use of the TIMES option. The statement executes paragraph—name—1 and all paragraphs up to and including paragraph—name—2, as many times as are specified by TIMES. An integer, which may be positive or zero, may be given, or alternatively, the number of times may be contained in an area referred to by data—name. This must contain either a positive or zero value. In either case, the value zero causes the PERFORM instruction to be ignored.

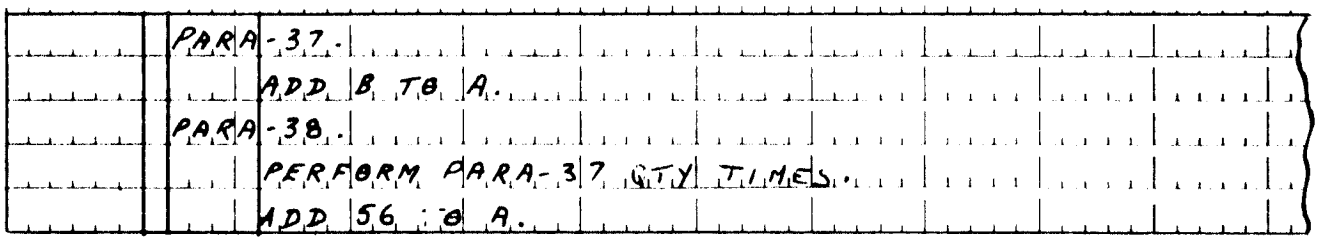
If the data—name option is used, a count is set up and the number of times held in the data name is tested when PERFORM is executed for the first time. This count is decreased by one each time the paragraph or paragraphs are performed. When it equals zero, control returns to the statement following PERFORM.

With this format it is the initial value of the field which is important. If the paragraph(s) performed change the field subsequently, the new contents will not affect the number of times the PERFORM occurs. In the following format, the contents of the field as PERFORM is obeyed will affect the number of times the loop is gone through.

Examples
The following example calculates A + 9B + 56. As it is necessary to add B to A nine times, a PERFORM statement is used. PARA—37 is executed once in the normal program sequence, and then eight times by the PERFORM. Thus PERFORM PARA—37 8 TIMES is used, as the paragraph will have already been gone through once.



Alternatively, similar steps could be used to calculate A + *n*B + 56, where *n* is variable. If the data name QTY, as shown below, contained 3, PARA—38 would calculate A + 4B + 56.



3 PERFORM paragraph—name—1 [THRU paragraph—name—2] UNTIL condition.

The statement performs one or more paragraphs several times, where the number of times is unknown. In this case PERFORM is executed until the condition is satisfied, and control is then transferred to the statement following PERFORM. If the condition is already true, the PERFORM statement will not be executed.

The UNTIL option is therefore equivalent to the conditional statement

P.
IF NOT condition PERFORM paragraph—name—1
[THRU paragraph—name—2] GO TO P.

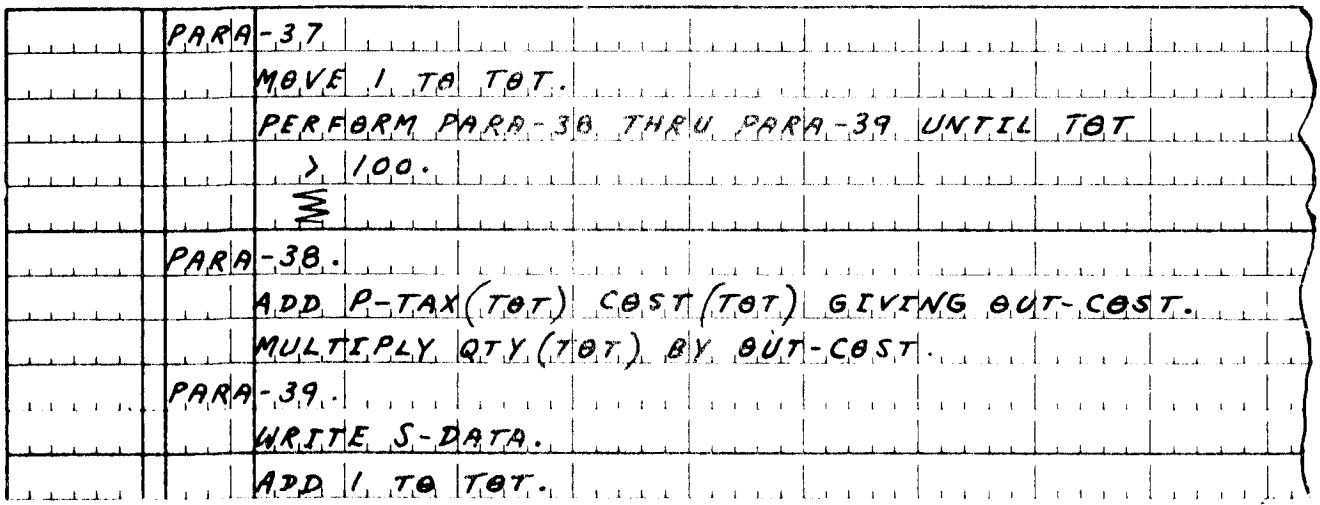
The condition after UNTIL will usually refer to a field containing the number of times the sequence is to be performed. For example, the condition

UNTIL COUNT EQUALS ZERO

will cause paragraphs to be performed according to the number currently held in a field called COUNT. The value of COUNT must be reduced by one each time the sequence is performed, and when COUNT equals zero, control is transferred to the statement following PERFORM.

Example

The following example performs two paragraphs a number of times, until a field called TOT contains more than 100. TOT is used as a subscript to reference three fields, P—TAX, COST and QTY. 1 is added to TOT each time the paragraphs are executed.



4 PERFORM paragraph—name—1 [THRU paragraph—name—2]

VARYING data—name—1 FROM { literal—1 / data—name—2 }

BY { literal—2 / data—name—3 } UNTIL condition.

This performs the same operations on a series of identical fields which are addressed by means of a subscript (see Chapter 6 "RULES FOR SUBSCRIPTS"). The subscript is a field already set up in working—storage which contains a reference to the current field to be processed. Program steps are executed for each data item by means of a subscript loop, which refers to the correct item by the value held in the subscript. An example of this was given under "SUBSCRIPT LOOPS" Chapter 6.

Using PERFORM ... VARYING, the subscript given in data—name—1 will start at the value of literal—1 (or the contents of data—name—2) and will be incremented by literal—2 (or the contents of data—name—3) each time the paragraphs are performed, until the condition is satisfied. The condition will usually specify an upper limit for the PERFORM, the number the subscript must contain for the condition to be satisfied. For example,

UNTIL COUNT = 10

indicates that PERFORM must be executed until COUNT contains 10, when control is returned to the statement following PERFORM.

Since the subscript is automatically incremented by a given value, it is unnecessary to add the value to the subscript explicitly in the program.

Example

Consider the example of a [illegible] loop given on page 102, which multiplies a [illegible] by a quantity [illegible] twelve times. This is achieved by [illegible] using a subscript COUNT, which is increased by 1 each time the loop is performed, and is then tested to see if it contains less than 13.

Using the same Data Division and working-storage entries, twelve identical fields are set up in a table using OCCURS. A table of result fields is defined in working-storage, together with the subscript COUNT. Using PERFORM ... VARYING, the Procedure Division statements would be written as shown below. LOOP is a paragraph placed at some point in the program where it cannot be reached accidentally.

```
       PERFORM LOOP VARYING COUNT FROM 1 BY 1
           UNTIL COUNT = 13.
           ...
   LOOP.
       MULTIPLY PRICE(COUNT) BY QUANTITY
           (COUNT) GIVING TOTAL-COST(COUNT).
```

As shown, there is no need for any statements to increment or test the value of COUNT: all this is done by the PERFORM. Note that COUNT must contain 13 before the loop is stopped, if it is to be performed 12 times. COUNT need not start from 1 or be increased by 1 each time: it could, for example, be increased by 2, if it were required to process every other subscripted item.

As an alternative to the example above, data names can be used to hold subscripted values if the number of times the loop is to be performed is unknown. In this case the following statement might be written, where A-FIELD and B-FIELD hold the starting value of COUNT and the incrementation value, respectively.

```
       PERFORM LOOP VARYING COUNT FROM A-FIELD
           BY B-FIELD UNTIL COUNT = 13.
```

Compiler action with PERFORM

Although a PERFORM statement may be written in one of the ways described above, and requires no further action by the programmer, it is useful to know how the compiler deals with the statement in order to execute a given sequence in the program. Certain steps are inserted into the object program by the compiler, which enable it to perform given paragraphs and then return to the normal program order.

A simple example of PERFORM is shown below.

```
       PERFORM P-X.
       MOVE TOTAL TO P-TOTAL.
           ...         ...
   P-X.
       DIVIDE 17 INTO GROSS.
       MOVE GROSS TO TOTAL.
   P-Y.
```

The machine code generated by the compiler is equivalent to the steps shown below.

```
  Place GO TO STEP-3 at the end of P-X.
  GO TO P-X.

STEP-3.
  Clear GO TO STEP-3 from the end of P-X.
  MOVE TOTAL TO P-TOTAL.

P-X.
  DIVIDE 17 INTO GROSS.
  MOVE GROSS TO TOTAL.
  [GO TO STEP-3.]

P-Y.
```

Nested PERFORM

So far, only single PERFORM statements have been discussed. However, a paragraph referred to by a PERFORM statement may include another PERFORM, given certain conditions. The range of paragraphs covered by the second PERFORM must be either

1 completely outside the range specified by the first PERFORM.
or
2 completely within the range specified by the first PERFORM.

PERFORMS enclosed one within another in this way are known as *nested PERFORMS*.

The following skeleton examples show how the principles are applied:

```
  PERFORM A THRU B.
  MOVE ANSWER TO RESULT.
   ⋮              ⋮
A.
   ⋮              ⋮
B.PERFORM C THRU E.
   ⋮              ⋮
C.

D.

E.
```

In this example, PERFORM C THRU E is within the range of PERFORM A THRU B, but the paragraphs C to E it refers to are outside the range of the first PERFORM. A and B are performed until the second PERFORM is reached. The program then executes C, D and E before returning to the next statement in B. When B is completed, control is passed to MOVE ANSWER TO RESULT.

```
  PERFORM A THRU E.
  MOVE ANSWER TO RESULT.
   ⋮              ⋮
A.
   ⋮              ⋮
B.PERFORM C THRU D.
   ⋮              ⋮
C.

D.

E.
```

In this example, C THRU D is totally contained within the range of the first PERFORM, A THRU E. A and B are executed until the second Perform is reached. The program then executes C and D, returns to the next statement in B, and continues again with C, D and E. Finally, control is passed to MOVE ANSWER TO RESULT.

If the ranges of two PERFORM statements overlap, the second performed sequence cannot be completed. The following example is incorrect because C THRU E is neither contained within, nor outside A THRU D.

```
  PERFORM A THRU D.
  MOVE ANSWER TO RESULT.
   ⋮              ⋮
A.
B.PERFORM C THRU E.
   ⋮              ⋮
C.
D.
E.
```

Here, the first PERFORM statement inserts a GO TO at the end of paragraph D, to return to MOVE ANSWER TO RESULT. When the second PERFORM is encountered, a second GO TO is inserted at the end of paragraph E, to return to the next statement after the PERFORM in paragraph B. The program executes paragraphs C and D, but then encounters the GO TO at the end of D. Instead of continuing with paragraph E and then back to B, it returns immediately to the MOVE statement. In this case, the GO TO will also remain at the end of E, and will be obeyed inappropriately if E is reached in normal sequence.

Paragraphs associated with two different PERFORMS may overlap only if one PERFORM statement is not contained within the range of the other. The following example is therefore correct.

```
  PERFORM A THRU C
   ⋮              ⋮
A.
B.
C.
D.
   ⋮              ⋮
E.PERFORM B THRU C.
```

The EXIT verb

The EXIT statement provides a common end point for a series of paragraphs. Its format is simply

```
EXIT.
```

EXIT must appear in a sentence by itself. It must be preceded by a paragraph name and must be the only sentence in the paragraph. The EXIT verb is always used in conjunction with a PERFORM statement; hence if it is encountered without a PERFORM being in effect, it is ignored.

EXIT is used chiefly when there are several paths to the end of a PERFORMed sequence.

Example
The following example shows two otherwise identical routines which both contain a sequence change to the next paragraph.

```
D-PARA.
    SUBTRACT DELTA FROM ALPHA.
    IF ALPHA = ZERO GO TO E-PARA.
    DIVIDE ALPHA INTO BETA GIVING RESULT.
E-PARA.
    ...
F-PARA.
    ...
G-PARA.
    SUBTRACT DELTA FROM ALPHA.
    IF ALPHA = ZERO GO TO H-PARA.
    DIVIDE ALPHA INTO BETA GIVING RESULT.
H-PARA.
```

In this case, PERFORM cannot be used on its own; for example, the statement PERFORM D—PARA under G—PARA would produce an unwanted branch to E—PARA, where a branch to H—PARA was required.

PERFORM can, however, be used in conjunction with EXIT as shown below.

```
D-PARA
    SUBTRACT DELTA FROM ALPHA.
    IF ALPHA = ZERO GO TO D-END.
    DIVIDE ALPHA INTO BETA GIVING RESULT.
D-END.
    EXIT.
E-PARA.
    ...
F-PARA.
    ...
G-PARA.
    PERFORM D-PARA THRU D-END.
H-PARA.
```

A paragraph, D—END is inserted after D—PARA, and this gives a common EXIT whenever the paragraph is performed. If D—PARA is performed in its normal sequence, the program will automatically continue to E—PARA after EXIT. When PERFORM D—PARA THRU D—END is encountered, the PERFORM statement will automatically insert the statement GO TO H—PARA at the end of D—END, and this will return the program to the next paragraph, H—PARA.

Processing multi-level tables

As described in [illegible] data can be set up in the form of a table of up to three levels, by using an OCCURS clause. A three-level table would be processed by setting [illegible] subscripts and extracting each field in turn by means of [illegible] a loop mechanism would be involved, and each subscript would have to be incremented each time the loop was gone through.

Taking the example given under [illegible] and three level tables" in Chapter 6, suppose that a paragraph called DAY-PROCESS had to extract each DAY-QTY field in turn from the table. The coding given below shows the necessary Data Division and Procedure Division entries.

```
       WORKING-STORAGE SECTION.
       01  MONTH-SUB PIC 99 COMP SYNC RIGHT.
       01  WEEK-SUB PIC 9 COMP SYNC RIGHT.
       01  DAY-SUB PIC 9 COMP SYNC RIGHT.
       01  TABLE.
           02 MONTH OCCURS 13.
              03 WEEK OCCURS 4.
                 04 DAY-QTY PIC 9(5) OCCURS 6.

           MOVE 1 TO MONTH-SUB MOVE 1 TO WEEK-SUB
               MOVE 1 TO DAY-SUB.
       DAY-PROCESS.
           MOVE DAY-QTY (MONTH-SUB, WEEK-SUB, DAY-SUB)
           TO WORK.
              (PROCESS WORK AS REQUIRED)
           ADD 1 TO DAY-SUB. IF DAY-SUB NOT > 6
              GO TO DAY-PROCESS.
           MOVE 1 TO DAY-SUB ADD 1 TO WEEK-SUB.
           IF WEEK-SUB NOT > 4 GO TO DAY-PROCESS.
           MOVE 1 TO WEEK-SUB ADD 1 TO MONTH-SUB.
           IF MONTH-SUB NOT > 13 GO TO DAY-PROCESS.
```

The most complex version of PERFORM could be used as an alternative to the loop. This has the format

PERFORM para – name – 1 [THRU para – name – 2] VARYING

data – name – 1 FROM {literal – 1 / data – name – 2} BY {literal – 2 / data – name – 3}

UNTIL condition – 1 [AFTER data – name – 4 FROM

{literal – 3 / data – name – 5} BY {literal – 4 / data – name – 6} UNTIL condition – 2]

[AFTER data – name – 7 FROM {literal – 5 / data – name – 8} BY

{literal – 6 / data – name – 9} UNTIL condition – 3].

This format allows the subscripts of one, two or three level tables to be initialised and advanced, and the contents of the table to be processed by means of a single statement. However, as might be expected, this statement is of very considerable complexity. In the example given above, the paragraph DAY—PROCESS can be placed in a corner of the Procedure Division (without the loop mechanism at the end) and can be PERFORMed 260 times as shown below.

```
PROCEDURE DIVISION.
    PERFORM DAY-PROCESS VARYING MONTH-SUB FROM
        1 BY 1 UNTIL MONTH-SUB = 14 AFTER WEEK-SUB
        FROM 1 BY 1 UNTIL WEEK-SUB = 5 AFTER
        DAY-SUB FROM 1 BY 1 UNTIL DAY-SUB = 6.
    .
    .
DAY-PROCESS.
    MOVE DAY-QTY (MONTH-SUB, WEEK-SUB, DAY-SUB)
        TO WORK.
    (Process work as required)
NEXT PARA.
```

Notice that the subscripts must be referred to in the PERFORM statement, in the sequence shown. The conditions given after UNTIL must be equal to 14, 5 and 6 respectively if the paragraph is to be PERFORMed until the subscripts reach 13, 4 and 5.

It has been seen that multi-level tables involve great complexity in the Data Division in defining the table, and in the Procedure Division in setting up and advancing two or three subscripts. Therefore, there is no value in using multi-level tables unless some feature of the problem in hand makes their use advantageous. In the majority of cases, one level tables are sufficient.

It should be emphasised that multi-level tables are merely viewpoints of store. If only the lowest level of field was to be addressed in the above example, the 260 DAY—QTY fields could be defined as

```
01 TABLE.
   02 DAY-QTY PIC 999 OCCURS 260.
```

This requires only one level of subscript varying from 1 to 260. In either case the actual layout of store is the same, 260 consecutive three-digit fields, but a one level table is normally easier to process.

PERIPHERAL VERBS

In COBOL, input and output of data to and from peripheral units is achieved by the operation of certain verbs. These verbs are as follows:

OPEN
READ
WRITE
CLOSE

An input file must first be OPENed by the program. It is then READ and each record is processed in turn. For an output file, each record is written separately by a WRITE verb. When a file is no longer needed it is CLOSEd by the program.

For a given file, the peripheral verbs must be used in the order given above. Only one OPEN and CLOSE instruction usually occurs in a program for each file, but any number of READ or WRITE instructions may be given. (A magnetic tape file can be re-opened once it has been closed, but the operator may have to put the tape deck back on line.)

This section does not deal with peripheral verbs as used with direct access devices. A description of this is given in Chapter 11.

Units of input and output

General points about peripheral devices were made in Chapter 1, under "peripherals". Certain rules also apply, regarding the length of records in 1900 COBOL, which varies with the type of peripheral used. The amount of data handled by a READ or WRITE statement is affected by the record length, by the nature of the file and by the peripheral.

CARD PERIPHERALS

For an *input* card file, the record length is defined according to the longest record declared for the file, which may have a maximum of 80 characters. The data read always begins at column 1 of the card, and consists of the number of characters defined in that longest record.

For an *output* card file, the record length declared in the Data Division must be exactly 80 characters. All 80 card columns must be filled with characters; blank columns are space-filled.

PAPER TAPE PERIPHERALS

A paper tape file may contain either fixed length or variable length records. Variable length records must be terminated by a newline (see Chapter 1).

For an *input* file, the program will always attempt to read the number of characters (maximum) defined in the largest record in that file. Reading stops when this number has been read, or when a newline is detected, whichever occurs first. Note that two character positions must be allowed for newline in the record description.

For a record containing variable length fields (where non-significant zeros and spaces are omitted), these fields must be expanded explicitly by program steps.

For an *output* file, the record area is normally the size of the longest record, and the number of characters punched will be the same for every record. A subroutine, which allows variable length records to be punched, is available and is described in Chapter 10, under "variable length data handling".

It should be noted that newline is never punched unless a working storage field is set up with the VALUE clause, giving the decimal or binary equivalent of the newline. This must then be moved to an output record area before punching.

LINE PRINTERS

The record length for files output on the line printer depends on whether the printer is buffered or unbuffered.

For buffered printers, the record description need only specify the number of print positions actually required, starting at column 1, and the record area is allocated accordingly.

For unbuffered printers, the record length must be the number of print positions on the printer (normally 96 or 120 positions). Each record description must therefore specify this number of characters, and spare positions in a record must be filled with spaces. If a record is defined with a length other than 120 characters, the compiler listing may contain the message.

BLOCK SIZE WRONG

However, the object program will handle the data correctly unless a genuine error has occurred.

If the type of printer used is not known, the record length declared in the Data Division should be 120 characters.

MAGNETIC TAPE

As described in Chapter 1, each record area for a magnetic tape file is associated with a buffer; in this, data is held before it is transferred to the record area for an input file, and after being moved from the record area for an output file. The length of a record is held in words by the word count word. The word count word indicates how many words are to be moved between the record area and the buffer, and this total includes the word count word itself.

By specifying the RECORDING MODE clause in the file description the correct value of the word count word is normally inserted automatically by steps inserted by the compiler according to the record name quoted in the WRITE. For example, if there are two record types in a magnetic tape file, REC–A, 50 words long and REC–B, 25 words long, then the statement.

WRITE REC–A

inserts a count of 50 in the word count word, while WRITE REC–B inserts a count of 25.

The word count word in the record area is always set to zero after a WRITE statement. When the next WRITE is encountered, the word is tested, and, if still zero, the automatic value is entered.

The OPEN verb

The SELECT ... ASSIGN statements in the Environment Division allocate specific peripherals to input and output files. The OPEN verb makes these files available to the program, and also performs certain procedures involving magnetic tape labels.

The OPEN statement has the following format:

```
OPEN {INPUT file-name-1 [file-name-2] ...}
     {OUTPUT file-name-1 [file-name-2] ...}
```

Before a file can be read or written it must be opened in the program. Any number of files may use a single OPEN verb, provided that they are specified as input or output. The files can be listed in any order. Each file-name must be identical to that given in a SELECT assign statement. A second OPEN for a file cannot be executed unless a CLOSE statement has been executed. An example of an OPEN statement is given below.

```
OPEN INPUT STOCKS ORDERS OUTPUT SALES-DATA.
```

For files on punched cards or paper tape units, the statement OPEN reserves a peripheral of the required type (as specified in the Environment Division) and produces a LOAD FILE message on the console.

When an input file is held on magnetic tape, OPEN causes Executive to search for a tape with the required identification and, if specified, the correct generation number. The header label of the tape is searched, the identification of the file being given by the VALUE of ID (see Chapter 6 "The VALUE of ID clause"). The generation number, if present, will have been specified in the LABEL RECORDS clause for the file, and will also appear in the header label (see Chapter 6 "The LABEL RECORDS clause").

When an output file is held on magnetic tape, OPEN causes Executive to search for a scratch tape, that is, one where the ACTIVE TIME entry in the header label indicates that the retention period has elapsed (see Chapter 6 "The ACTIVE TIME clause"). This file is erased, and a new header label written containing the name, retention period and generation number, if required, of the file.

In both cases, if a generation number is specified, the message
SET FILE GENERATION NO IN WORD 0
appears on the console.

In the case of magnetic tape files, the file-name specified in the OPEN statement should not be confused with the VALUE OF ID, the name in the header label of the file.

The READ verb

The READ verb gives the program access to the next record on an input file. It also specifies action to be taken when the end of the file is reached.

The READ statement has the following format:
READ file-name AT END imperative-statement.

This statement makes the next record from the specified input file available in the record area for processing. Each record in the file is READ consecutively; the program checks for the end of file marker and when this signifies that the end of the file has been reached, the imperative statement following AT END is obeyed. It should be noted that this is almost always a GO TO statement containing a branch to a closing routine.

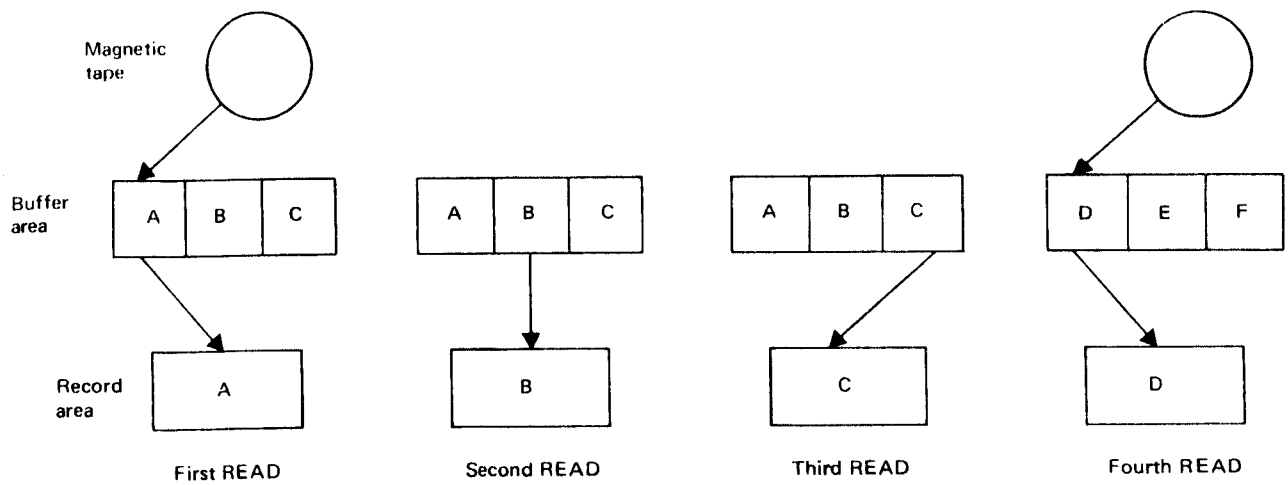
For example,

```
READ STOCKS AT END GO TO FINISH.
```

The *end of file marker* varies with the nature of the file. For a card file, it is a card punched with asterisks in columns 1 to 4. If double buffering is used, several blank cards should occur after the end card. For a paper tape file it is a block punched with four asterisks and a newline (with double buffering, two newlines). For a magnetic tape file, an end of file trailer label (see Chapter 1 "magnetic tape") appears. If a magnetic tape file occupies more than one reel, READ automatically closes one reel and makes available the first record of the next.

The procedure carried out by READ for magnetic tape files is rather more complex than that for slow peripheral files, as it is determined by the operation of the magnetic tape block (see Chapter 6 "The BLOCK CONTAINS clause"). Records are grouped in blocks, and when a block has been read into store, each record must be *unpacked* and processed in turn. If a block contains only one record, each READ statement reads a single record block into store and makes the record available to the programmer in the record area. If, however, a block contains several records, the first READ transfers the whole block to an input file area, unpacks the first record and reads it into the record area. Subsequent READ statements move each record to the record area and overwrite the previous record. When the whole block has been read, the next READ moves the next block into the file area, and so on.

This sequence of events is illustrated below. A block of records is read collectively from the tape. If the block contains three records, A, B and C, the first READ transfers the whole block to the file area, and A to the record area. When all the records have been read, the fourth READ will once again read a whole block.



It should be emphasised that the programmer is not directly concerned with blocks, but need only issue normal READ instructions.

The WRITE verb

The WRITE verb releases a record to an output file. It is important to realise that whereas READ reads a *file name*, WRITE writes a *record name*. A file often contains more than one record type. On an input file, there is no way of distinguishing the type of record until it has been read, and then it must be determined by test. READ can therefore demand a record only from a particular file. On the other hand, when a record is output it has already been named and built up by the program, and can thus be specified by WRITE.

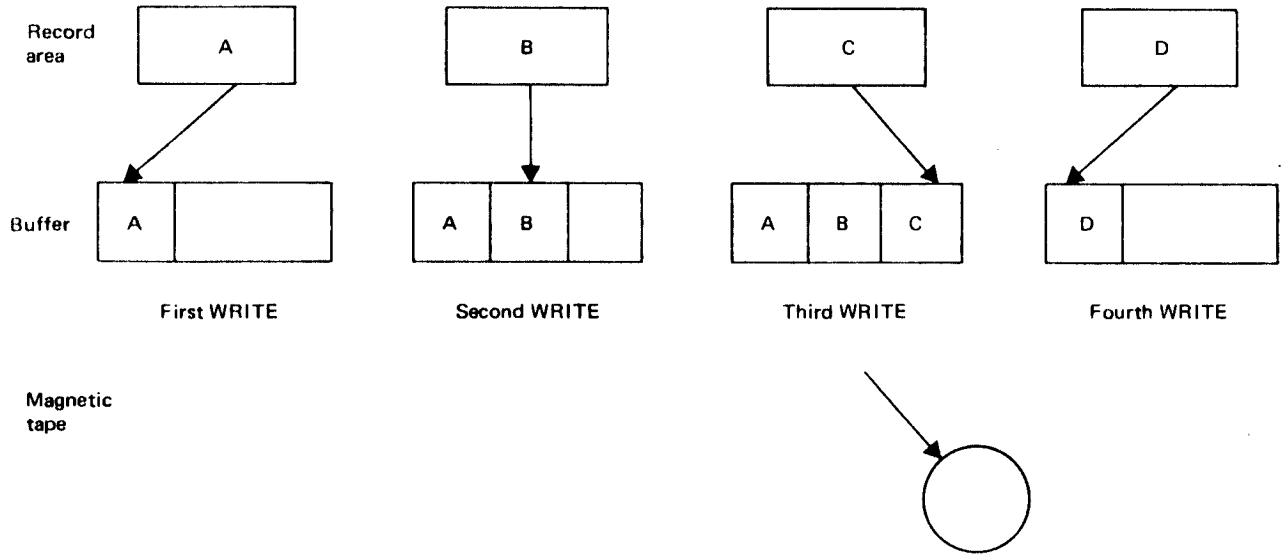
The WRITE has two formats, as follows:

1 WRITE record—name.

This is the simple form of WRITE which is used for all records but those output on the line printer. For example,

```
WRITE A-REC.
```

As records on magnetic tape are unpacked by READ, so magnetic tape records are packed into blocks by WRITE. This process is the exact opposite of that described above. The first WRITE instruction moves the first record to the output file area; subsequent WRITEs transfer records until the block has been filled. The last WRITE instruction writes the whole block to the magnetic tape file, and the packing of the next block begins. This sequence of events is illustrated below.



2 WRITE record—name [AFTER {1 / 2 / CHANNEL—n / mnemonic name}]

This form of WRITE is used for files output on the line printer. The AFTER clause controls line spacing of the printed record.

If AFTER 1 is specified, a line will be spaced before the record is printed. This record will therefore be printed out on the line below the previous record in the file. Similarly, AFTER 2 specifies double-spacing; an extra line will be left blank between the two records. For example,

```
WRITE A-REC AFTER 2.
```

AFTER CHANNEL—n causes the paper to be moved until a punching is detected in the specified channel of the paper tape loop which controls the printer (see Chapter 1). CHANNEL—1 is head of form position, and the remaining channels 2 to 7 can be used to indicate stopping positions on the form. Mnemonic names specifies a name given to one of the channels of the loop in the Special—Names paragraph (see Chapter 5 "SPECIAL—NAMES PARAGRAPH").

It is the programmer's responsibility to ensure that his WRITE statements produce the required spacing for printed output. A line-count is normally used to keep track of the current position on the form. A detailed description of the use of the control loop, and the statements which should be used is given in the next section.

SPACING ON THE PRINTER

It has already been stated that the program can specify paper spacing directly, by means of AFTER 1 or 2. Alternatively, WRITE can initiate paper throw and specify the channel in the paper tape loop where a punching is to halt the throw. This is done by using WRITE AFTER CHANNEL—1 to CHANNEL—7.

It has been mentioned that CHANNEL—1 is used only once to establish the *head of form* position. Although any of the remaining channels can be used to halt paper throw within the body of the form, it is better practice in COBOL to use the same channel in the range 2 to 7 consistently.

The programmer must keep a line count to indicate the current position on the form. According to the type of line he is building up, he will specify field names from the appropriate record description, and will then give a WRITE order which specifies the correct paper throw before this line is printed.

These procedures are not difficult, but can easily cause errors in program writing. There is only one record area for the print file and any data to be printed must be placed in that area. It is not uncommon to find that when checking a program's logic, a line of detail has been built up in the record area when in fact a heading is required next. Such errors are due to testing the line count at the wrong point; if not corrected, they would mean that the detail line data would have to be built up again in the record area, after the heading data had been moved in and printed.

The flowchart given in Figure 5 shows the logic of a program printing one line from a card, and thirty-five lines to a page, with a heading at the top of the page.

A heading is usually laid out as a non-numeric literal. This can be associated with a 01 level field in the Data Division, by a VALUE clause, and the field is then moved to the print record area. Alternatively, the literal itself can be moved to the record area. The literal does not have to be 120 characters long, but need only be as long as the number of print positions, from the first to the last which is not a space.

Notice the movement of spaces to the print record area after printing the heading. Whenever lines of different formats are printed sequentially, the layout should be checked to ensure that remnants of one line are not left in space fields of the next line. The simplest way of doing this is to space-fill the entire record area after printing, as is done in the example.

A full example of the paper tape loop in relation to the printer is given below.

Example
This example shows how a simple statement is printed out. The printing form and the paper tape loop are shown diagrammatically, in Figure 6, and the statements necessary to achieve the required spacing are given.

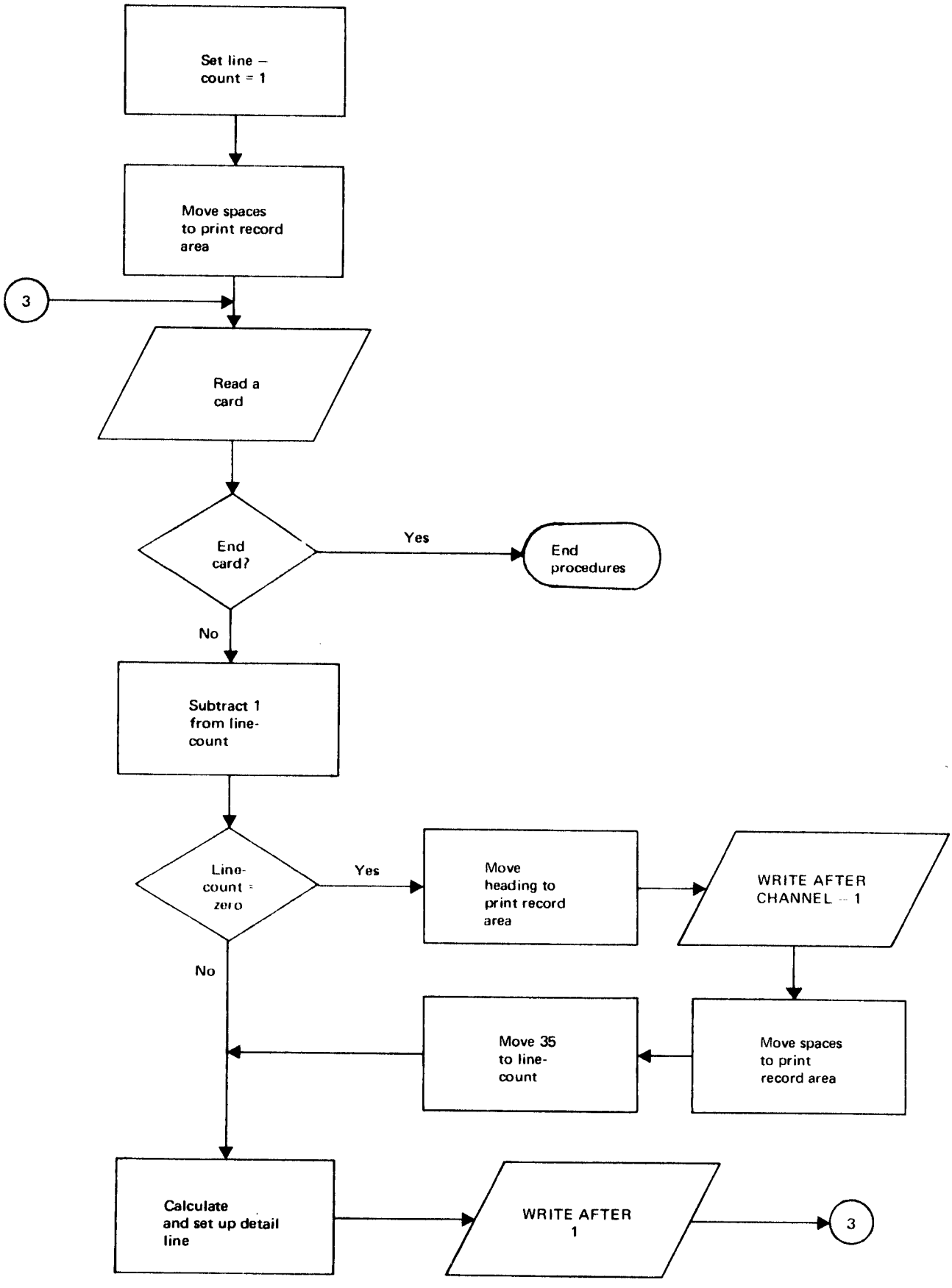


Figure 5 Line printer output

On the document shown, a name and address, a heading, a number of detail lines and a total are printed on each sheet. The necessary WRITE statements would be as follows:

1 WRITE ... AFTER CHANNEL—1. This throws to the hole punched in channel 1 of the loop, representing head of form, and then prints the first line of name and address.

2 WRITE ... AFTER 1. This spaces one line and prints the next line of name and address. Identical statements print subsequent name and address lines.

3 WRITE ... AFTER CHANNEL—3. This gives a throw to the first hole punched in channel 3 of the loop, and prints the heading line. See note 4 below.

4 WRITE ... AFTER CHANNEL—3. This throws to the first line of detail.

5 WRITE ... AFTER 1. This spaces one line and prints the next detail line. Other detail lines are printed in the same way.

6 WRITE ... AFTER CHANNEL—3. Finally, a throw is made to the last hole punched in channel 3, and the total line is printed.

Notes:

1 The control loop is not to scale, since the diagram is intended to illustrate the relation of the loop to the paper only.

  The loop has a 6 : 10 ratio in relation to the form. One sprocket hole is equivalent to one line on the form. The form has 6 lines to the inch, while the loop has 10 sprocket holes to the inch. An 11" form would therefore require a 6.6" length of paper tape.

2 Channel 3 is used consistently for WRITE statements in the body of the form. This means that the same WRITE ... AFTER CHANNEL—3 order could be used with PERFORM.

3 This example ignores the possibility of continuation sheets. If such a possibility could occur, extra statements would be required, for example, to print "Page 2" or any other continuation information at the top of the next sheet, before continuing to print detail lines.

4 Headings may be laid out either by means of the VALUE clause in a 01 level of the Working—Storage Section, or as a non-numeric literal in the Procedure Division. In the first case, the field containing the heading would be moved to the output record area before being written. In the second, the literal itself would be moved to the record area. See "LITERALS" earlier in this chapter.

The CLOSE verb

The CLOSE verb terminates the processing of input and output files, and releases peripherals from the program.

The CLOSE statement has the following format:
CLOSE file—name—1 [file—name—2....].

When an input or output file has been processed, it must be associated with a CLOSE verb in the program; the file name must be the same as that given in the OPEN statement.

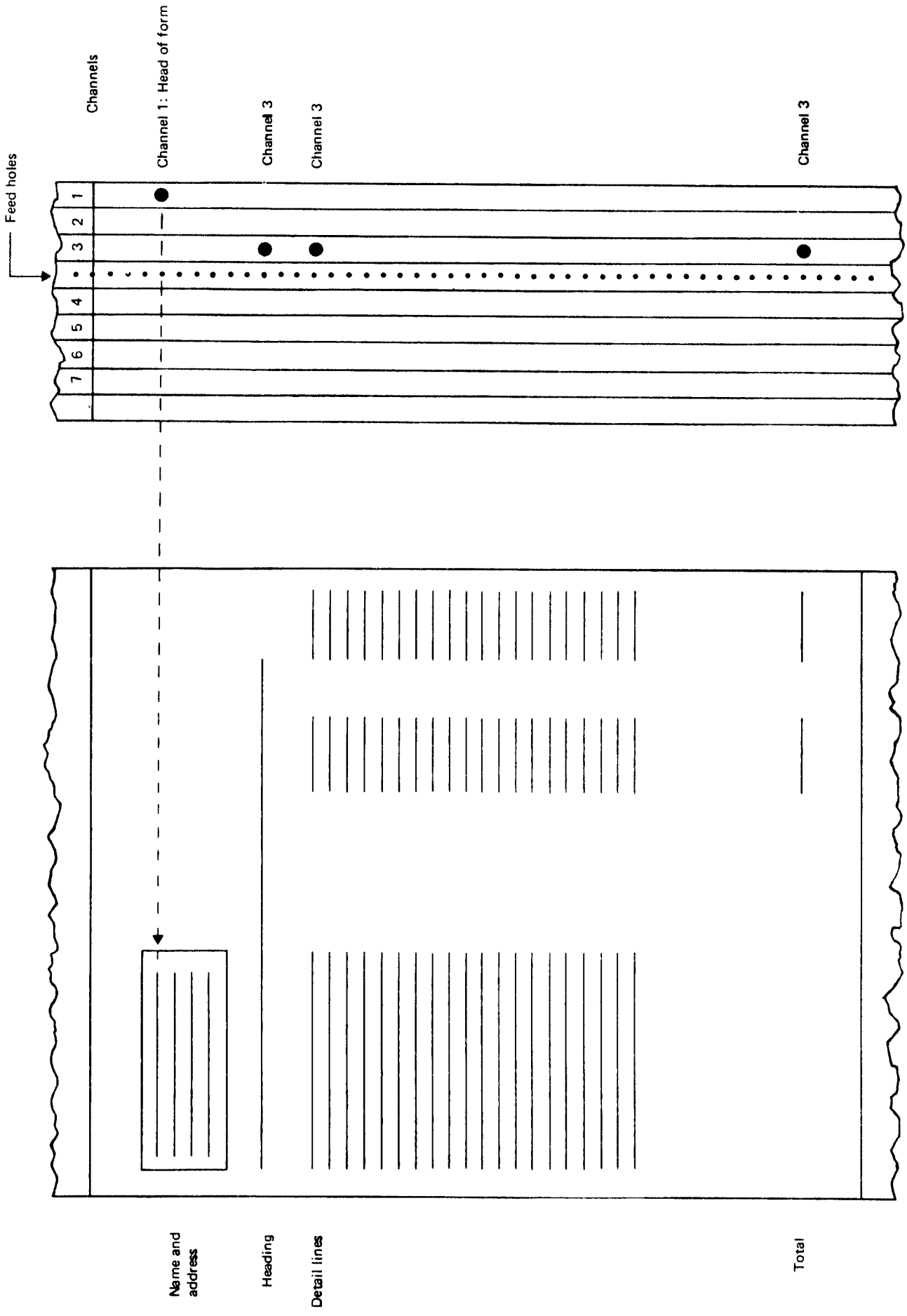
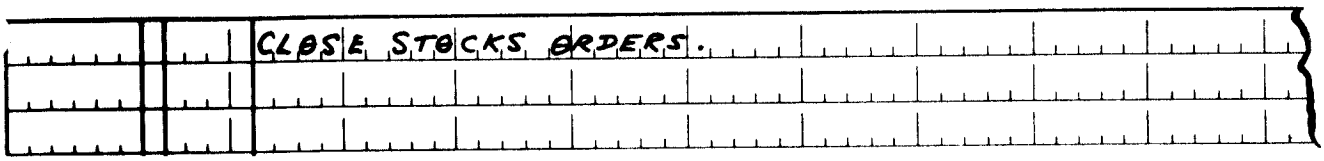


Figure 6 Operation of the loop

For example,



For a file held on cards, paper tape or the line printer, CLOSE releases the appropriate peripheral from the program.

A magnetic tape input file will have already had its trailer label detected by the last READ statement.

For a magnetic tape output file, CLOSE causes any remaining data to be written, and a trailer label to be created and written. The tape is rewound and the tape deck released.

Chapter 8 Compilation and the object program

THE COMPILATION PROCESS

Introduction

The compiler program reads source COBOL which is usually on paper tape or cards, translates this and outputs an object program in 1900 machine code which can be loaded into a 1900 machine and executed. This chapter considers the detail of compilation procedures.

Compilation consists of a number of phases, the main ones being:

1 The *analysis* phase

2 The *generation* phase

3 The *consolidation* phase

The analysis phase

The compiler scans the COBOL source program word by word looking for format errors.

The Data Division is converted into tables which give full details of each field, that is size, editing, usage and so on.

Tables may be held in store or written to backing store. This is decided by the compiler and depends how much store it has available.

The Procedure Division is converted to simple expressions or simplexes from which the next phase will produce object code.

The generation phase

Object machine code is generated using the tables and simplexes from the first phase. This is in semicompiled form rather than pure machine code. Semicompiled formats are considered in more detail below.

The consolidation phase

Standard subroutines, for example, MTH and slow peripheral handling routines, are held on libraries in semicompiled form and are incorporated in the semicompiled program at this point. Coding requested by the programmer by ENTER statements is also incorporated at this point.

In both cases, the coding is obtained by searching a library on backing store or, more rarely, by request to the operator who can feed in routines from paper tape or cards.

The consolidated leader, which is a summary of the storage requirements of the whole program, is output with the object program as is a special loader program called the General Purpose Loader (G.P.L.) which will be used to put the object program into store at some later stage.

Semicompiled form

Each step is in machine code form except for the second address. This consists of a relativiser and a number, instead of being a word address relative to word 0 (X0) of the program.

These relativisers of LV, LP, UP and UV were considered in Chapter 1. If SUM were the thirteenth Lower variable, it would be addressed as LV 12. This address must be converted to a word address eventually. For example, if it is decided that the first lower variable, LV0, is to be word 45 relative to word 0, then SUM must eventually become word 57.

The conversion is not carried out during compilation. The compiler only has a complete picture of the object program after consolidation when all sub-routines have been incorporated. It would not be efficient to process all the object program again at this point. Instead the processing is left until the whole program has to be handled again, during loading. Then the G.P.L. uses the consolidated leader information to convert the second address of each step as it loads it into store.

An object program of this type that is semicompiled with a consolidated leader and G.P.L., is referred to as being in *consolidated semicompiled* form.

After loading and conversion of the second address, the program is referred to as being in *binary* form.

After the program has been loaded the operator can output a binary copy to cards, paper tape or magnetic tape. Such binary programs can be reloaded into store at any time by a simple copying process which is included in Executive.

Programs held in libraries on a magnetic backing store are usually in binary form, being easier to load and more compact to store than the consolidated semicompiled form.

COBOL COMPILERS

A wide range of COBOL compilers is available for use on the 1900 Series machines, differing in core requirements, backing storage medium and COBOL facilities that can be translated. A short description of each compiler is given in Appendix 6. Compiler facilities, however, are liable to change. Full details may be found in the COBOL reference manual and current User Notices.

STEERING LINES

The purpose of steering lines

All the COBOL compilers have a number of optional facilities. Output of the object program may be to cards, paper tape or to magnetic tape or disc, depending on the compiler. Correction of a source program written to backing storage during a previous compilation may be required. Consolidation may or may not be required and so on.

Steering lines are the programmer's orders to the compiler telling it exactly what it is to do during the translation process. These orders are punched into cards or paper tape and fed to the compiler before the source program is read in.

The format of steering lines

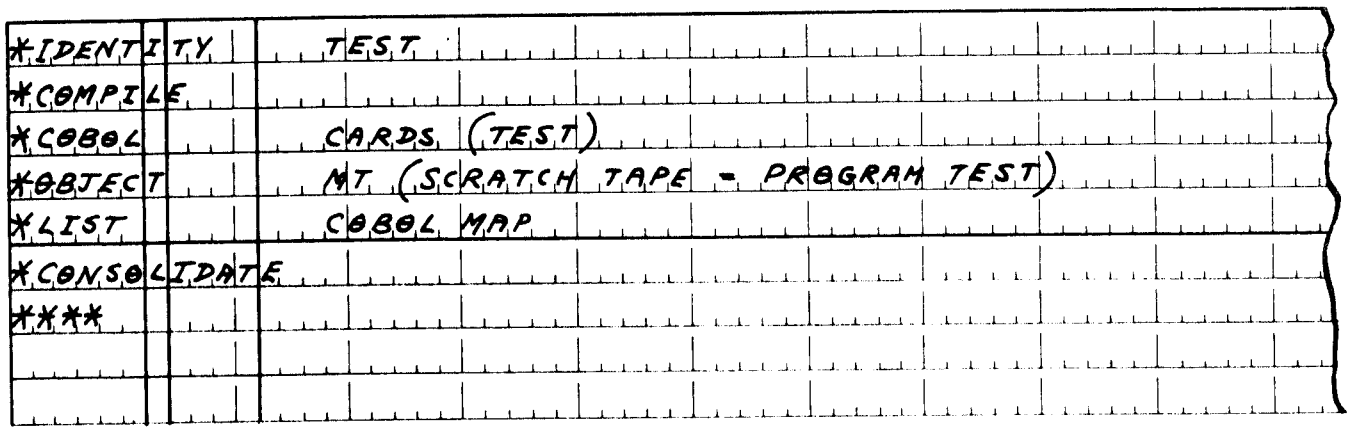
Each steering line is punched from column 1 onwards into a card or paper tape block. It consists of a *directive*, preceded by an asterisk, a *medium* entry and then an *operand*. The medium and operand are not always required.

There are many different types of directive and a number of different entries may occur in the medium and operand. This section will not attempt a complete survey of all the different possibilities. It will concentrate on a simple set of steering lines for a magnetic tape COBOL compiler. Full details of steering lines can be found in the COBOL Compilers manual.

Writing the steering lines

The simplest way of writing COBOL steering lines is to use a COBOL coding sheet which can be punched with the COBOL source program. The punching must start in column one. Entries for medium and operand must be separated from each other and the preceding directive by at least one space.

This is a typical set of steering lines punched on cards.



The first point to be made is the importance of getting the steering lines correct. The slightest error is likely to halt the compilation immediately, even before the source program has been read. There are three common errors which will be pointed out as this example is examined.

The first line is the *IDENTITY directive. This gives the name of the program, in this case TEST. This is the same name as that quoted in the Identification Division but without the priority number, that is the most significant four characters only. The first common error is to put the priority in as well.

The *COMPILE directive specifies that compilation is required.

The *COBOL directive specifies the medium from which the source program is to be read, in this case the card reader. The name in brackets should be the program name that is TEST but it is of little importance. It is only used by the compiler in a message output on the console typewriter:

PUT TEST ON UNIT *n*

However there must be at least one space between the medium and the left hand bracket. The second common error is to omit this space.

The *OBJECT directive specifies where the semi-compiled program is to go, in this case to magnetic tape. The entry in brackets states the current name in the label of the tape which is to receive the object program and the new name to be inserted in the label. So in this case a scratch tape may be used and it is to be re-labelled with a new name PROGRAM TEST. This entry can be much more complex. A named tape can be requested rather than a scratch tape, a specific tape serial number can be specified and sub-files can be handled.

The third common error is to omit the space between MT and the left hand bracket. It does not matter whether there are spaces on either side of the equals or not.

The directive *LIST asks for a full COBOL listing. The format of this listing is considered under "THE COMPILATION LISTING" later in this chapter.

The directive *CONSOLIDATE specifies that the consolidation phase is to be entered, that is a loadable program is to be produced either held in store or on a disc file.
The card punched with four asterisks marks the end of the steering lines.

Other directives

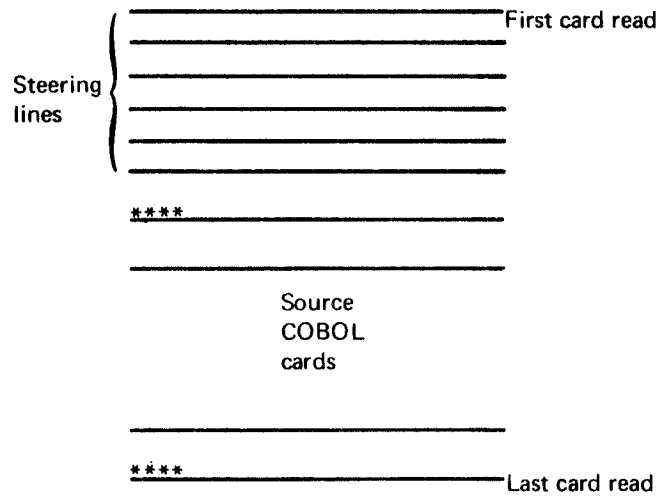
After the CONSOLIDATE directive three other directives, LOAD, BINARY and SUBROUTINES might appear.

The directive LOAD indicates that the object program is to be loaded into store after compilation, ready to be run. The BINARY directive is used to hold the object program on a disc file and is actually loaded by using LOAD.

The directive SUBROUTINES will be used if special subroutines are to be incorporated into the object program during consolidation because of ENTER statements in the original source program.

Order of input

The diagram below shows the sequence in which cards must be fed to the compiler. It should be noted that there is a second card punched with four asterisks at the end of the COBOL source pack.



Batch compiling

Batch compilers can handle a string of source programs, without operator intervention between each compilation. Each source program is preceded by a set of steering lines which is terminated by a record containing four asterisks in the first four character position. The batch is terminated by four slash marks (////). Identical directives need not be repeated in different sets of steering lines since they are carried forward until replaced.

Example
The four programs, FSA1, FSA2, FSA3 and FSA4 are to be batch compiled. The steering lines and source packs are input as shown opposite.

ICL Data processing

COBOL program sheet

title
programmer

sheet number
date

Sequence No. | Identification

```
1      6 7 8      11 12     15        20        25        30        35        40        45        50        55        60        65        70  72 73 75        80
*IDENTITY          FSA1
*COMPILE
*COBOL             CARDS (FSA1)
*OBJECT            MT (SCRATCH TAPE=PROGRAM MODS.PROGRAM FSA1)
*LIST              COBOL MAP
*CONSOLIDATE
****
                   Source pack for FSA1
****
*IDENTITY          FSA2
****
                   Source pack for FSA2
****
*IDENTITY          FSA3
****
                   Source pack for FSA3
****
*IDENTITY          FSA4
****
                   Source pack for FSA4
****
/*/*
```

FORM C14/41(7.68)

1900 Series
Program operating instruction sheet

ICL

Section A

Program control letter

Programmer: J. BROWN — Program type: COBOL

Estimated running time: 5 minutes — Program name: XCXB

Maximum running time: 7 minutes — Job code: 142

Section B **Special instructions**

Section C **Peripheral requirements** (enter number required)

Card reader	1	Paper tape reader		Magnetic tape units	4	Cassette stations		E.D.S. transports	
Card punch		Paper tape punch		Line printer	1	Minimum number of scratch tapes	3		

Section D **Input/Output requirements**

Input file details and order of input

Seq.	File	Cards (labelled)	P.t (labelled)	Magnetic media (M.S.N./Filename)
1	SYSTEM			#0000027/PROGRAMTAPE
2	STEERING	Supplied		
	RAPIDWRITE SOURCE			
	MASTER SOURCE			
	AMENDMENTS			
3	COBOL SOURCE	#EXAM		
	SUBROUTINES			

Output file details

	TRANSLATED SOURCE			
	CORRECTED SOURCE			
1	OBJECT PROGRAM			SCRATCH TAPE

Section E **Program running**

1 FI # X E KB #DISC in 13000 (CORE REQUIRED) from COBOL SYSTEM LIBRARY

2 ON 4 total of magnetic media units required for run, minimum of 4, maximum of 8

3 ✓

GO 20 ☐ STEERING ON PAPER TAPE — If S/R's are on a slow peripheral and all those which have been supplied have been read and exec. Types "UNIT nn FIX", then SUspend and GO ☐

GO 21 ✓ STEERING ON CARDS

4 **Expected end of run messages**

HALTED: COMPILED: # XXXX nnnn ERRORS	
HALTED: CONSOLIDATED: # XXXX nnnn ERRORS	See below ✓
HALTED: SERVICED: # XXXX nnnn ERRORS	ABANDON
DELETED LO # XXXX	

Run to follow if consolidated with less than 5 errors. Cards P.T.

Give binary dump if consolidated with less than 5 errors, on ☐ ✓ P.T.O.

Figure 7 Operating instruction sheet

OPERATING INSTRUCTIONS

Introduction

Most installations nowadays have operating systems to simplify the task of the operator and control the whole running of jobs on the machine. The current operating systems on 1900s are GEORGE 1, 2, 2+, 3, 3+ and 4.

The COBOL programmer has to supply the operating instructions to the operating system which are similar to the instructions given to the operator under a manual system.

Operating instructions for COBOL compilations

The main information to be passed to the operator is the compiler to be used and the amount of store that it is expected the compiler will have to operate in.

The compiler name must be the appropriate one for the installation hardware and the COBOL facilities used in the source program.

The store must be at least equal to the minimum for that compiler.

Figure 7 shows a completed COBOL instruction sheet which supplies the operator with the required information.

Operating instructions for the object program

The most important information required by the operator is the messages that the object program may output and the operator action expected.

Other information includes the various peripherals and the amount of store required by the program.

The end message expected, for example,

HALTED:— END OF RUN

must be given and any operator action required at the end of the run, clearly laid out. Since the program will be under test initially, operator action whether at the final or at some unexpected halt will often be to take a core dump and tape or disc prints on the line printer. In this case, the requirements must be explicitly stated.

A program which was produced by a 1900 COBOL computer will output certain standard messages, a full list of which can be found in Chapter 17 of the COBOL reference manual.

When files are opened on fast peripherals, a SET FGN message may appear, if WITH GENERATION—NO was used in the Data Division. The operator must be told what he must alter word 0 to become, that is what the generation-number is for this file on this run.

Program Instruction Sheet

PROGRAM NAME	#EXAM		
PROGRAM DESCRIPTION	CARDS TO TAPE		
CORE	2000	PRIORITY	69

Estimated Running Time _____ Mins. 10

SPECIAL INSTRUCTIONS On any unexpected halt dump first 500 words of core and print tape produced (JOB HOURS) using # XRME

PAPER TAPE	—		
PRINTER STATIONERY	—	CONTROL LOOP	—

PERIPHERAL REQUIREMENTS	At Beginning	Running	At End Only
PAPER TAPE READER			
PAPER TAPE PUNCH			
CARD READER	✓	✓	✓
CARD PUNCH			
LINE PRINTER			
MAG. TAPE DECKS	1	1	1
E.D.S.			

LIBRARY REQUIREMENTS — LIBRARY TAPE —

DATA TAPES USED	WPR

No. OF SCRATCH TAPES NEEDED 1

E.D.S. CARTRIDGES

OPERATING INSTRUCTIONS

STEP No.		MESSAGE	NEXT STEP
1	FI #EXAM	EXAM loaded	2
2	GO #EXAM 20	LOAD FILE 0 ON N	3
3	Put cards in reader		
	GO #EXAM	SET FILE 1 GEN. NO	4
4	ALTER #EXAM 0 1	OK	5
5	GO #EXAM	Invalid card seq. nos. displayed. EXPECTED EOR MESSAGE "HALTED – END OF RUN."	6
6	Dump first 500 words of core. Then print tape produced (JOB HOURS) using #RME		

FORM 14/35 (4.67)

Figure 8 Program instruction sheet

These messages appear on the console log in the order in which the files are opened in the Procedure Division. The files are numbered in these messages from 0 in the order in which they appear in the Data Division.

Figure 8 shows a typical program instruction sheet for a job writing cards to magnetic tape.

The console log

When a programmer is testing a program he should always receive a copy of the console log. If running under an operating system he may also receive a document monitor which gives a clear account of what happened while the program was running.

Figure 9 is a typical console log showing a COBOL compilation. In this case # XEKB was used operating in 16,000 words of store. GO 21 indicates that the steering lines were on cards.

A run followed writing cards to magnetic tape, with a label name JOB HOURS.

Finally after the END OF RUN message the first 500 words of the program's area were output on unit 3, the printer, that is a limited core dump consisting of 500 words of lower data was taken.

Then the tape that had been written was printed using the utility program # XRME.

File numbering in the object program

Three sets of numbers are involved and should not be confused. For files assigned to slow peripherals, a typical COBOL Environment Division statement would be

SELECT CARD—INPUT ASSIGN CARD—READER *n*

The compiler takes *n* as the program unit number of the card reader to which this file is assigned, and it will be quoted in such run-time Executive messages as

UNIT *p* USED AS *n*

where *p* is the absolute unit number of the peripheral in the installation.

The SELECT ... ASSIGN ... statement does not allow a program unit number for magnetic tape files. The compiler therefore assigns numbers to all magnetic tape files in the order in which they are selected in the Environment Division, commencing the numbering at 1. These numbers are then quoted in run time Executive messages, exactly as for slow peripherals. The number 0 is reserved and when the object program is segmented into overlays, it is copied in binary by the loader, to tape number 0. This tape should be preserved for later object runs. If there has been a RERUN statement in the COBOL source program and the dump is to a magnetic tape, this tape is assigned a number by the compiler, one greater than the last COBOL magnetic tape file.

Both these points should be remembered when assigning numbers to magnetic tape files which are defined and referenced in entered routines.

For the purpose of run time messages such as

LOAD FILE *nn* ON

all files in the COBOL source program are also numbered from zero up, in the order in which they are declared in the Data Division.

COBOL compilation

```
FI  # XEKB    DISC 16000 O.K.
    # XEKB P
    # XEKB HALT LD
GΘ  # XEKB 21
    # XEKB CR4 FIX
    # XEKB LP3 FIX
    # XEKB DISP: COMPILED    EXAM
    # XEKB DELTED FI   XPCK
    # XPCK P
    # EXAM HALT LD
```

Object program run

```
GΘ  # EXAM 20 O.K.
    # EXAM HALT LOAD FILE 00 ON 04
GΘ  # EXAM O.K.
    # EXAM CR4 FIX
    # EXAM HALT SET FILE 01 GEN
AL  # EXAM 0 1
GΘ  # EXAM O.K.
    # EXAM MT21 JOB HΘURS *00005021 CLOSED
    # EXAM HALT END OF RUN
```

Core dump

```
ΘU  # EXAM 0 500 3 O.K.
    # EXAM HALT AE
DE  # EXAM O.K.
    # EXAM DLTD
```

Disc print

```
FI  # XRME    DISC O.K.
    # XRME P
    # XRME HALT LD
GI  # XRME 21 0 O.K.
GΘ  # XRME 25 O.K.
    # XRME LP 3 FIX
    # XRME MT21 JOB HOURS *00005021 CLOSED
    # XRME LP3 FR
    # XRME HALT HH
DE  # XRME O.K.
    # XRME DLTD
```

THE COMPILATION LISTING

A general survey

Figure 10 is a listing of the example program in Chapter 4. A number of deliberate errors have been incorporated. This is the format that will be output by the compiler if MAP is specified in the steering lines.

The listing consists of a number of main sections:

1 The steering lines and source COBOL statements.

2 Error messages.

3 Data map.

4 Program map.

5 Certain statistics.

6 Consolidation information.

Steering lines and source COBOL

These are listed and if not already sequenced, the source COBOL lines are given six digit numbers rising in steps of 100. The words COMPILER SEQUENCING will be printed, as in this example, if the compiler does have to number the lines. The sequence numbers are used thereafter to refer to any given line in the source.

Error messages

Most but not all error messages follow immediately after the source COBOL listing. Exactly where an error is printed depends on the nature of the error and how much core the compiler has to work in. Error messages may occur before or after this point in the listing.

The errors are errors of *format* which means the rules of COBOL have been broken. An error free compilation does not mean the program is correct as there may still be logic errors.

All error messages have the same format. Consider for example the ones referring to SEQ 003600 in Figure 10. The error here is a missing hyphen in REF—OUT which the compiler therefore sees as two names, REF and OUT, which have not been defined in the Data Division.

The REF numbers, in this case 003, refer to the COBOL reference manual where further details of each error type are given in Chapter 16. In most cases, however, the information given on the compiler listing together with an examination of the erroneous line will be enough to find the error.

Since the error messages refer only to line numbers, it may be difficult to find an error in a line containing complex statements or multi-statement sentences. In general, the simpler the COBOL the easier it will be to find errors.

An error at one point of a program can cause error messages not only referring to that point, but also referring to later lines which may have nothing wrong with them. For example the error message referring to line 000800 states that the I/O section entry is wrong in some way. In fact line 000800 is quite valid. What is wrong is line 000600 where the hyphen is missing from OBJECT—COMPUTER. Line 000600 also contains another error which causes an error message referring to line 001200. This error is the presence of 'A' after ICL-1904.

It can be seen that the error messages must be treated with some care. They indicate that the compiler has not accepted a given line either because there is a format error or because an earlier error causes the

```
       DATE 07/02/77                                  1900 COBOL COMPILER XEKB# 559
*IDENITY        EXAM
*COMPILE
*COBOL           CARDS (EXAM)
*OBJECT       MT (SCRATCH TAPE = PROGRAM EXAM)
*LIST            COBOL MAP
*CONSOLIDATE
****
=←"↑</() $(!>(|=<|^
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID. EXAM69.
00300 ENVIRONMENT DIVISION.
00400 CONFIGURATION SECTION.
—†— SOURCE-COMPUTER. ICL-1904A.
000600 OBJECT-COMPUTER. ICL-1904A.
000700     MEMORY 4000 WORDS.
000800 INPUT-OUTPUT SECTION.
000900 FILE-CONTROL.
001000     SELECTCDIN ASSIGN CARD-READER 1 RESERVE 1.
001100     SELECT MTO ASSIGN TAPES 1.
001200 DATA DIVISION.
001300 FILE SECTION.
001400 FD   CDIN.
001500 01   IN-DATA.
001600     02 REF-IN PIC 999.
—[©—     -] FACTOR-A PIC 99.
001800     02 FILLER PIC X.
001900     02 FACTOR-B PIC 99.
002000 FD   MTO.
002100     RECORDIMG MODE-F.
002200     BLOCK 50 RECORDS.
002300     LABEL RECORDS STANDARD WITH GENERATION-NO.
002400     VALUE OF ID "JOB HOURS".
002500     ACTIVE-TIME IS 21.
002600 01   OUT-DATA.
002700     02 REF-OUT PIC 999.
002800     02 FACTOR-C PIC 999.
002900 WORKING-STORAGE SECTION.
003000 01   AREA PIC 999.
003100 PROCEDURE DIVISION.
003200 PARA-ONE.
003300     OPEN INPUT CDIN OUTPUT MTO.
003400 CALC.
003500     READ CDIN AT END GO TO LAST-PARA.
003600     MOVE REF-IN TO REF-OUT.
—*©—    MOVE FACTOR-A TO AREA.
003800     ADD FACTOR-B AREA GIVING FACTOR-C.
003900     WRITE OUT-DATA.
004000     GO TO CALC.
004100 LAST-PARA.
004200     CLOSE CDIN MTO.
004300     STOP RUN.
****
ERROR    SEQ 000600  REF 078 INVALID WORD IN COL. 8.
ERROR    SEQ 000600  REF 079 SENTENCE INVALID FOR THIS DIV.
CAUTION  SEQ 002400  REF 805 LEVEL NUMBER MISSING.
ERROR    SEQ 002400  REF 567 INVALID DATA DECL.
ERROR    SEQ 002700  REF 567 INVALID DATA DECL.
CAUTION  SEQ 003000  REF 837 NO DATA NAME.
ERROR    SEQ 003000  REF 079 SENTENCE INVALID FOR THIS DIV.
ERROR    SEQ 003600  REF 003 UNDEFINED NAME - REF
ERROR    SEQ 003600  REF 003 UNDEFINED NAME - OUT
CAUTION  SEQ 003600  REF 006 EMPTY LIST  0686
CAUTION  SEQ 003700  REF 006 EMPTY LIST  0686
ERROR    SEQ 003800  REF 003 UNDEFINED NAME - FACTOR-C
ERROR    SEQ 003800  REF 012 INVALID SYNTAX  0139
ERROR    SEQ 000800  REF 734 I/O SECTION MISPLACED
ERROR    SEQ 001200  REF 717 OBJECT COMPUTER MEMORY INVALID/MISSING
                                              MAP FD CDIN
                                              MAP 01 IN-DATA                  0.0.0
                                              MAP 02 REF-IN                   0.0.0
                                              MAP 02 FACTOR-A                 0.3.0
                                              MAP 02 FILLER                   1.1.0
                                              MAP 02 FACTOR-B                 1.2.0
CAUTION  SEQ 002000  REF 703 FILE MAY LOST ON CLOSING
                                              MAP FD MTO
```

```
ERROR   SEQ 002400  REF 706 FILE UNASSIGNED
CAUTION SEQ 002000  REF 743 RECORD<4 CHARS
CAUTION SEQ 002400  REF 702 FILE INFORMATION CLASH
CAUTION SEQ 002400  REF 703 FILE MAY BE LOST ON CLOSING
                              MAP FD ACTIVE
ERROR   SEQ 002700  REF 745 NO SIZE/PICTURE
                              MAP 01 OUT-DATA                  0.0.0
                              MAP 02 REF-OUT                   1.0.0
                        FILE-LIST    CDIN              LV    43
                        FILE-LIST    MTO               LV    45
                        FILE-LIST    ACTIVE            LV    43


MAP  PROGRAM  NAME  EXAM

MAP SECTION COBOL                                PERMANENT
MAP      SEQ 003200        ADDRESS 000002
MAP      SEQ 003300        ADDRESS 000002
MAP      SEQ 003400        ADDRESS 000033
MAP      SEQ 003500        ADDRESS 000033
MAP      SEQ 003600        ADDRESS 000052
"@↑      $(! 003700        ADDRESS 000052
MAP      SEQ 003900        ADDRESS 000052
MAP      SEQ 004000        ADDRESS 000063
MAP      SEQ 004100        ADDRESS 000064
MAP      SEQ 004200        ADDRESS 000064
MAP      SEQ 004300        ADDRESS 000075
"@↑ $(=[<←| =←◊←/                          ↑()"@|(|[     ↑)←^)@"     82
STATISTICS: *DATA      =     12 X  6.3
STATISTICS: *PROCEDURE=      3 X  7.0
STATISTICS: *LOWER = 75 WORDS

CORL    1728
         LV 45          LP 92          LT 174          UP 962          UV 1096
"@↑ ↑()"@|(|[ ↑)←^)@"(^<|$ @[     245
41          861  COBL
41          837  COBM
41          348  HMTWR
41          842  COBO
41          327  HMTEND
41          611  HMTDEF
41          785  COBS
41          597  HMTMOD
41          402  HMTP
41          483  HMTF
41          390  HMTN
41          453  HMTD
41          441  HMTQ
41          425  HMTO
41          553  HMTAA
CAUTION SEQ 004300  REF 619 TOTAL OF   13 ERROR(S) SIGNALLED
```

On the compilation listing, entries headed FILE—LIST give the start address of each record area expressed as a LV number. Record areas could also start at LP or UV/UP words. In general File Section entries will be LV until a large number of words have been used and then further entries will be UV. Working—storage entries will be LP and later ones may be UP 01 levels always start on a word boundary.

In Figure 10 it can be seen that CDIN starts at LV 43. Since LV0 is usually word 45 in LOWER, CDIN starts at word 88. Therefore it can be seen that for example FACTOR—B is in word 89.

The FILE—LIST entry ACTIVE is caused by the error in the source when the hyphen was left out of ACTIVE—TIME and a full stop appeared after the ID entry.

Figure 11 is a section of a core dump of the corrected program. The columns of the core dump are as described in Chapter 9. This dump was produced by a small processor and therefore only has three columns; the word address in octal, the word as a machine code step and the contents of the word in octal.

The contents of FACTOR—B can now be found from this dump since the word address has been found from the data map. The word address of FACTOR—B is 89 which in octal is 131. It can be seen from Figure 11 that word *131 contains *20202020.

```
*00001: 000 4 0/0553   *40000553
*00002: 064 0 0/2054   *03202054
*00003: 000 0 0/0220   *00000220
*00004: 000 0 0/0002   *00000002
*00006: 000 0 0/0001   *00000001
*00007: 000 0 0/0001   *00000001
*00010: 000 4 0/0555   *40000555
*00011: 000 0 0/0004   *00000004
*00012: 000 0 0/0047   *00000047
*00021: 126 6 0/0014   *65300014
*00022: 126 0 0/0007   *05300007
*00023: 161 0 0/5444   *07045444
*00024: 074 0 0/0400   *03600400
*00025: 161 0 0/0201   *07040201
*00026: 161 0 0/0202   *07040202
*00027: 161 0 0/0203   *07040203
*00030: 161 0 0/0204   *07040204
*00031: 161 0 0/0205   *07040205
*00032: 161 0 0/0206   *07040206
*00033: 161 0 0/0207   *07040207
*00034: 161 0 0/0210   *07040210
*00035: 161 0 0/0211   *07040211
*00130: 046 3 2/3232   *32323232
*00131: 004 2 0/2020   *20202020
*00133: 060 0 1/0400   *03010400
*00134: 101 0 0/0000   *04040000
*00137: 000 0 0/0006   *00000006
*00140: 000 0 0/0030   *00000030
*00141: 000 0 1/0000   *00010000
*00142: 000 0 0/0014   *00000014
```

Figure 11 Core dump

Program map

This section allows the number of machine code steps generated by each line to be calculated. For this purpose the words within the program step area are numbered from 0. Words 0 and 1 are reserved for use during the running of the object program, and the steps generated from the first statement in the Procedure Division will start at word 2.

From these it can be calculated that, for example, the READ at SEQ 003500 generated 19 steps because the READ starts at word 33 of the program step area and the next statement, MOVE, starts at word 52.

Paragraph names and sequence numbers generate no code, nor do statements with serious errors.

Statistics

This section is printed during the consolidation phase.

DATA gives the number of data names in the source and their average length in characters.

PROCEDURE gives the number of paragraph names in the Procedure Division and their average length.

LOWER gives the storage requirements in Lower Data for this program less the amount already specified in the data map.

Consolidation information

This section gives information which is only available when the whole program has been compiled. It contains core requirements, where each category of store starts and subroutine detail.

CORE

This is the total requirement for the program, in this case 1728 words. This means that the first word of LOWER is, as usual, word 0, while the last word of UPPER is word 1727.

STORAGE CATEGORIES

The word numbers (relative to X0) on which LV, LP, UP and UV start in this program, are printed on one line.

LT is short for literal which is another category of store. The program steps start at word 245 after the LT words and before the LP words at 962.

SUBROUTINE DETAIL

This is a list of all the subroutines incorporated in the program, quoting the name and word number for the first step. The number 41 is a code indicating that the rest of the entry relates to a program segment (in 1900 PLAN terms).

For example CKBL is a standard subroutine which is entered by coding generated by any peripheral verb. It checks that the peripheral action is valid, that is, if the program tries to READ, is the file OPEN?

If an error occurs, the program must be abandoned and a message appears on the console.

HMTP is one of the MTH subroutines (whose names usually start HMT). It has been added to this program because it handles a magnetic tape file.

After this list comes a line specifying the TOTAL number of errors. This number also appears on the console.

Chapter 9 Program efficiency and testing

PROGRAM EFFICIENCY

Introduction

COBOL is often looked on as a high level language which can be learnt and written with little regard to the machine upon which the object program is to be run. In practice, such an attitude is likely to produce programs which are too large to fit the available core, especially when the machine used is an 8K machine.

This section is concerned with certain methods which should keep COBOL programs written for the 1900 Series down to a reasonable size. A competent COBOL programmer should be able to produce programs which take only a little longer to run than a comparable PLAN program and require no more than 10% excess core. On the other hand, the COBOL version should be up to 40% quicker to write and test.

The use of PERFORM

PERFORM is probably the most powerful aid to economy of generated coding. Wherever the same operations occur at more than one place in the program, the PERFORM verb should enable one set of coding to handle the operations on all the occasions that they are required. When designing flowcharts and when writing coding, possibilities of using PERFORM should always be looked for.

The advantages of using PERFORM (which only generates four steps in the object program) are obvious when repeated sets of COBOL statements are lengthy. There are cases however, when it is well worth using PERFORM even though the repetition consists of only one statement.

EDITING STATEMENTS

These are a case when it is worth using PERFORM for repetitions of only one statement.

Consider the following PICTURE clause:

```
OUT-SUM PIC ££99B99CR.
```

The statement:

```
MOVE WORK-FIELD TO OUT-SUM.
```

could generate up to 50 machine code steps, because of the complex conversion and editing that might be required. If there are several such editing operations required to build up one line of print, the 60 or 70 steps will occur for each one. In these circumstances, it may be possible to do the editing and conversion in a working—storage field and use these steps each time the operations are required.

PERIPHERAL VERBS

This is another case where a single statement can generate many steps. Any peripheral verb generates between ten and twenty steps. For example, more than twenty steps can be involved when a READ has complex AT END procedures. A peripheral verb which occurs in identical form more than once, should always be put into a paragraph by itself and on second and subsequent occasions that paragraph should be PERFORMed.

THE PROGRAM MAP

The program map indicates how many machine code steps each line of coding has generated (see end of Chapter 8). On a small machine or in a multi programming environment where there is little store available, the program map of the compilation listing gives a valuable indication of those parts of the source program where PERFORM's use might save many steps.

Subscript loops

Sometimes subscript loops can give the same sort of saving in machine code steps as PERFORM. For this to be possible, the factors to be processed must all be available at the same time and the problem must allow the factors to be laid out as a table.

Example

This is the same example as that used to illustrate the saving of machine code steps by using PERFORM (see below). The coding for this problem, using the subscript loop, is as follows:

```
DATA DIVISION.
FILE SECTION.
=
01  PRINT-REC.
    02 FILLER PIC XXX.
    02 FILLER OCCURS 3.
       03 OUT-FIELD PIC ££99B99CR.
       03 FILLER PIC XXX.
    02 etc
=
WORKING-STORAGE SECTION.
01  COUNTER PIC 9 COMP SYNC RIGHT.
01  FILLER.
    02 SUMS PIC S9999 COMP SYNC RIGHT OCCURS 3.
PROCEDURE DIVISION.
=
    MOVE 1 TO COUNTER.
LOOP-OUT.
    MOVE SUMS(COUNTER) TO OUT-FIELD(COUNTER).
    ADD 1 TO COUNTER.
    IF COUNTER < 4 GO TO LOOP-OUT.
```

The major limitation on the use of the technique is this essential requirement that it must be possible to describe the data as a table.

The PERFORM VARYING option already described in Chapter 7, could be used instead of the loop. This would involve having a paragraph:

```
EDIT-PARA.
    MOVE SUMS(COUNTER) TO OUT-FIELD(COUNTER).
```

in the Procedure Division where it could not be reached by accident, and at the point where the loop was required:

```
    PERFORM EDIT-PARA VARYING COUNTER
        FROM 1 BY 1 UNTIL COUNTER > 3.
```

Binary conversion

As has already been seen, it is advisable to take account of the fact that the 1900 has a binary adder and a word-organised store.

ARITHMETIC OPERATIONS

If a factor is involved in arithmetic more than once, it should be moved to a COMP SYNC RIGHT working—storage field and this field used in the arithmetic statements. This will ensure that binary conversion of the factor occurs only once, instead of each time it is involved in arithmetic.

Some types of IF statement involve binary conversion and the factors involved should also be moved to a COMP SYNC RIGHT working—storage field.

The saving in generated coding can be very considerable if this procedure is followed.

SUBSCRIPTS AND CONSTANTS

Working—storage fields which are to be used as subscripts, or which contain constants by use of the VALUE clause, should usually be described as COMP SYNC RIGHT. If this is done, the coding generated whenever these fields are handled will be considerably shorter.

Subscripting should be limited to the minimum logically possible. If a factor has to be extracted from a table several times, it should be extracted once, moved to a working—storage field and that field used on later occasions.

Literals

On some of the 1900 COBOL compilers, a literal is created as a constant each time it is mentioned in the Procedure Division. If a literal is used several times therefore, it is better to create it once as a constant in working storage, using the VALUE clause. For example consider

```
01  UNITY PIC 9 COMP SYNC RIGHT VALUE 1.
```

Whenever a constant of 1 is required, the coding would be:

```
MOVE UNITY TO X.
ADD UNITY TO Y.
```

Further aids to program efficiency

PAPER TAPE INTER BLOCK GAPS

The coding generated by WRITE statements for paper tape files will normally create data blocks without gaps between them. If it is required to create blocks with gaps between them the following steps should be carried out:

1 Define and name an extra DISPLAY field at the end of the output record, of a multiple of two characters in size.

2 MOVE into this field a literal consisting of the pair of characters ↑▽ repeated as many times as there are to be run-out characters in the gap. For example, for a gap of three characters, the coding is as follows:

MOVE " ↑▽↑▽↑▽ " TO FIELD.

where ▽ is a space character and where the field is defined as six characters in length.

FILLERS

It is fairly important for object code efficiency that fields should not be described as being longer than necessary. For example if the first ten columns on a card contain a number of some kind, then it may be described as:

```
01 CARD-RECORD.
   02 NUMBER PIC 9(10).
```

However if the number cannot exceed 999,999 it is better described as

```
01 CARD-RECORD.
   02 FILLER PIC X(4).
   02 NUMBER PIC 9(6).
      etc.
```

This saves code if arithmetic operations are to be performed on NUMBER, since the converted value will be single not double length, and the code generated to access and convert it will be considerably less. If there is no shortage of space, this is not of primary importance; in general, however, it should be remembered for this and similar situations that fields should not be described as longer than necessary.

SYNCHRONIZATION

The SYNC [illegible]
code. How[illegible]
areas as [illegible]
understood. [illegible]

```
02  A  OCCURS  10
    03  B  PIC  9.
    03  C  PIC  9  SYNC
    03  D  PIC  99.
```

Suppose [illegible]
a word, then [illegible]
as shown [illegible]

The compiler [illegible]
certain FILLER [illegible]
accounted for [illegible]
inserted between [illegible]
the second [illegible]
characters [illegible]
inserted it will [illegible]
characters and [illegible]
after D, the [illegible]
as shown in [illegible]
correctly processed.

DATA VALIDATION

At object run time, if a non-numeric character is read into a numeric field, the character will be ignored (on conversion by MOVE and so on) and no error will be flagged. Because of this it is recommended that data is validated by using the IF . . . NUMERIC test.

THE FULL STOP AS AN INSERTION CHARACTER

The full stop character in a Picture is used as an insertion symbol and the implied decimal point is always symbolised by V. Thus the full stop in

```
PIC 9V.99
```

is used as a simple insertion to represent the decimal point in the printed field.

A full stop can be inserted at the end of a Picture clause either.

1 by adding a semi-colon or comma immediately after the insertion full stop at the end of the PIC, if PIC is not the last clause describing the field, thus:

```
PIC 999.;
```

or

2 if it is the last clause, by writing an additional full stop after the insertion full stop at the end, thus:

```
PIC 999..
```

USE OF THE VERB CLOSE

When a program is to be run on multiprogramming machines, all slow peripheral files should be CLOSEd as soon as possible to release the units for other programs.

In a non-multiprogramming installation it is better to put all CLOSE statements in a single segment, usually the end of run segment, to conserve storage. The same applies to OPEN statements, which go into a Housekeeping segment.

Magnetic tape files are not put off-line by the CLOSE statement. Input files are released by MTH as soon as the end-of-file is read, and in this case the tape is put off-line. The CLOSE referring to such files causes the fact that the files are closed to be noted in the MTH tables and the run time file table. If it is required to read a tape several times during the running of an object program, the tape will have to be put on-line by the operator each time the file is re-opened.

This operator action can be avoided by writing a dummy marker block before the trailer label when the tape is created. Then when the file is read, the program can test for the marker block after each READ and when it is detected give a COBOL CLOSE. This will prevent the trailer label being read and the tape being put off-line although it will be rewound.

USE OF UPPER DATA

As already described, some data areas may be in Upper data, the set of words after the program steps which will be used when Lower is not sufficient. Such areas will be addressed by modification in the object code which is both time-consuming and wasteful of core.

In cases where store is short, it will be worthwhile for the COBOL programmer to try to ensure that frequently-used areas are in Lower rather than Upper. This can be done by defining such areas in the part of the Data Division which is most likely to be in Lower. This is the early part of the Working—Storage Section.

Frequently used working areas should be defined early in the Working—Storage Section and input/output data which is frequently accessed should also be processed in such working areas rather than in the record area.

If not concerned with efficiency, the whole question of Lower and Upper can be ignored by the COBOL programmer.

PROGRAM TESTING

Types of error

Almost every program contains errors when first written. These fall into two categories, format errors and logic errors.

Format errors mean that the rules of COBOL have been broken. These errors will be detected by the compiler.

Logic errors mean that the program does not do what the programmer expected. Such errors can be detected by feeding test data to the program and examining the results.

The importance of checking

Both types of error can be detected without running the program on the computer.

Flowcharts should be rigorously checked for logic, preferably by someone other than the programmer who wrote them.

Source COBOL should be checked for missing full stops and hyphens, and for the use of reserved words.

This deck checking can be carried further by dry running which involves taking sample sets of data and running them through the logic of the program, obeying the various tests and branches as the computer would.

Such checking procedures can save a great deal of computer time, and in general are much easier and quicker to do in COBOL than in a low level language like PLAN.

The mechanics of correction

If errors of any kind are discovered once the program has been compiled, the corrections must be incorporated.

It is not normal to attempt to 'patch' the machine code object program. Instead corrections are incorporated into the source program and this is then recompiled.

The process can be speeded up by putting the source program on to disc or magnetic tape during the initial compilation. This involves the use of the *CORRECT steering line. During any subsequent recompilation, this disc or tape can be used to provide the source program and only the corrections need to be read from a slow peripheral. Again the steering line *CORRECT must be used. See page 188 of the COBOL reference manual.

The detection of logic errors

Any logic errors which elude the desk checking and dry running procedures can only be detected by running the object program and examining the results. The test data chosen for this purpose must cover all extreme cases as well as all normal ones.

The output from such test runs comprises the normal results of the program plus a core dump showing the state of the store when the program halted. If this is not sufficient to pinpoint particular errors, it is possible for the contents of important fields to be printed as the program runs.

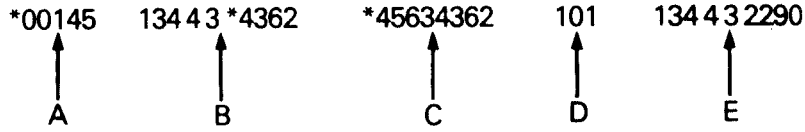
NORMAL RESULTS

Where output is to magnetic tape or disc, utilise routines exist to print out the contents of the file in various formats.

CORE-DUMPS

A core dump is usually printed on the line printer and is initiated by an OU<TPUT> message on the console. It is normally sufficient and quicker to output only part of the user's area of store.

The format of the output is shown below with one line per word and various interpretations of the binary pattern in that word printed across the line. Zero words are not printed.



Section A is the word number in octal. This is relative to the first word of Lower data the first accumulator.

Section B is the word as a machine code step. 134 is the function, 4 is the accumulator, 3 is the modifier and *4362 is the operand address in octal.

Section C is the contents of the word as 8 octal digits. In this particular case, it is a more sensible view than that taken by Section B. The word contains the internal code for ESCR, part of the word, DESCRIPTION in a heading constant.

Section D is the word number in decimal.

Section E is the same as section B except that the operand address is in decimal.

Sections D and E do not appear on core dumps produced by smaller processors. In this case the word number from Section A must be converted from octal to decimal. Word numbers on compiler listings are always in decimal.

The procedure for establishing the word address of a field from the compiler listing, is considered in Chapter 8.

PRINTING DURING THE RUN

If the contents of selected fields are to be output during the running of the program, it can be done by putting extra DISPLAY statements at appropriate points in the source COBOL.

Testing large programs

Large programs may be divided into sections for both writing and testing purposes.

As far as testing is concerned, this can be done by treating each section as a subroutine and compiling it separately using the COBOL subroutine compiler.

It is necessary to write a control routine either in PLAN or COBOL which will call in the section to be tested.

This routine must be compiled and during its consolidating phase, the section to be tested is incorporated as a subroutine. This section will have already been compiled by the COBOL subroutine compiler whose output is in suitable form to allow such incorporation.

The control routine may supply test data to the section under test and print out results. In addition it can supply any further coding that may be required because other sections of the program are missing.

Chapter 10 Subroutines

This chapter considers in some detail the use of subroutines in COBOL programs. These enable the programmer to incorporate in his program routines dealing with difficult or unwieldy procedures, and standard operations; he can also provide routines containing coding required a number of times, or may test his program divided into separate subroutines.

TYPES OF SUBROUTINE

There are three main types of subroutine which can be called in to COBOL programs.

1 Standard ICL subroutines: these are routines which cover most standard tasks in scientific and commercial fields. They are written by ICL for incorporation into any program written in PLAN or certain other languages. Some of these are of use in COBOL programs.

  General subroutines can also be written by installations for their own use, and can be used in COBOL programs.

2 Special subroutines: these are routines provided by ICL especially for the use of the COBOL programmer. They are written in PLAN and are designed to undertake operations not easily performed by the COBOL language or by specific COBOL compilers.

Both these classes of subroutine are incorporated in a COBOL program by means of the ENTER verb, whose operation is fully described in the first section of this chapter.

3 COBOL subroutines: the user may write his own subroutines in COBOL, to be called in by the main program. This facility is useful for routines which are common to several programs and need be written and compiled only once. It also assists testing of large programs, which can be split into a number of subroutines.

  These subroutines are incorporated in the main program by means of the CALL verb, as described later in this chapter.

This chapter describes how subroutines can be used and written and gives a full example of each type. However, for ICL subroutines, it does not attempt to list those available.

Full details of standard ICL subroutines will be found in the appropriate reference manuals; specifications of subroutines written for COBOL are given in *COBOL*.

THE ENTER VERB

The ENTER verb allows a COBOL programmer to incorporate subroutines into his own programs. These may be standard routines written by ICL, or the user's own routines written at installation level. The subroutines may be written originally in PLAN or FORTRAN, and will be held in semicompiled form.

Purpose of ENTER

There are two main reasons for using other language routines when writing a COBOL program.

1 Subroutines may ease programming effort. If a program requirement entails any procedures which would be unwieldy if written in COBOL, it is more efficient to write these procedures in PLAN and ENTER the routine at the appropriate point in the COBOL program. Examples of this application are where data is to be handled at bit level, or to be scanned for End Markers.

2 It is obviously sensible to utilise existing subroutines as far as possible. Many large programs written by installations will have one or more sections performing tasks common to most companies, for instance, a payroll requirement for handling tax. These widely used procedures are written as subroutines by ICL, and are always available on the Library Tape or Disc. The user, when writing his program, need only supply parameters regarding input, output and data areas, and ENTER the routine at the necessary point in the program.

   It is also common for installations to write their own general subroutines and incorporate these on the Library Tape or Disc.

Format of ENTER

The ENTER statement has the following format:

ENTER language–name subroutine–name USING parameter–1 parameter–2 ... parameter–n

Language–name can be any word written in conjunction with rules for data names, but for documentation purposes it should be the name of the language in which the routine was originally written. It must always be present following ENTER.

Subroutine–name is the name given to the routine at the time of writing. It may have a maximum size of eleven characters.

Parameter–1, and so on, are parameters required for the operation of the routine. These normally take the form of field names to be processed by the subroutine, or information in literal form. Parameters may be separated by commas if required, provided that each comma is followed by a space. Further details of the parameters are given in the next section.

ENTER parameters

A subroutine will normally require information about the program into which it is called. The necessary items are specified by parameters following the USING clause in the ENTER statement, and will in general include:

1 addresses of areas containing factors to be used in processing by the subroutine,

2 addresses of areas in which results generated by the subroutine are to be stored,

3 sizes of factors and work areas in the main program, types of End Markers, and so on,

4 addresses of areas into which error markers can be moved.

Subroutine parameters can thus be regarded as a guide for the routine to locate the data to be processed, and to return results to the main program. If necessary they also define field lengths and working–storage areas. The use of these parameters is best seen in relation to the example of ENTER which follows.

For full details of legal parameters, see under *ENTER* in Chapter 6 of *COBOL*.

Action of ENTER

This section gives a brief account of the way in which ENTER works. Further information depends upon the type of subroutine being used, and is given under the appropriate specification.

During the consolidation of the COBOL source program the compiler will incorporate into the object program all subroutines called in by ENTER. If several entries call in the same routine, only one copy of the routine is included in the object program. The necessary entry and exit instructions are incorporated for the second and for each subsequent time ENTER is encountered for that routine.

The subroutine will already have been compiled by some other language compiler, and will be held in semicompiled form. If the compiler is on magnetic tape and if the subroutine is not in the subroutine group on this tape, a *SUBROUTINES steering line denotes the name and medium where the subroutine can be found. If the compiler is on disc, there must be only one subroutine file also held on disc. If this file is not named SUBROUTINES, then a steering line must be inserted giving the name of the file.

During the running of the object program, where the machine code steps generated by an ENTER statement are encountered, a branch to the subroutine steps will occur. When the subroutine is completed, an automatic return to the main program takes place. This return is always to the statement following ENTER.

Example of ENTER

Figure 12 shows an example of the use of ENTER. This problem cannot be coded in COBOL because it involves bit manipulation with logical instructions, although it is a simple problem in PLAN. The information given should enable a COBOL programmer to ENTER a PLAN subroutine, or a competent PLAN programmer to write a subroutine for incorporation in a COBOL program.

```
ENTER PLAN MATCH USING WANTED DATA-EMP INDIC.
IF INDIC. = ZEROS GO TO PRINT-PARA.

#PROGRAM                    /MATCH
         0BEY          0(1)    [ADDRESS OF WANTED IN X3
         LDX      4    0(3)    [CONTENTS OF WANTED IN X4
         0BEY          1(1)    [ADDRESS OF DATA-EMP IN X3
         ANDX     4    0(3)    [X4 = 0 IF ALL REQ'S MET
         0BEY          2(1)    [ADDRESS OF INDIC IN X3
         STO      4    0(3)    [INDIC SET APPROPRIATELY
         EXIT     1    3

         CALL     1    MATCH
         LDX      3    'WANTED'
         LDX      3    'DATA-EMP'
         LDX      3    'INDIC'
```

A file of personnel data is held on magnetic tape, one record for each employee. Part of this data is held in a compressed form, where 18 characteristics are represented by a string of 18 bits, 0 meaning that the characteristic is present and 1 that it is absent.

A COBOL program interrogates this file, printing out the names of employees who have required characteristics. The particular pattern required is input as a parameter and held in a field called

```
WANTED PIC 9(6) COMP SYNC RIGHT.
```

A 1 is punched in the pattern if the appropriate characteristic must be present, a 0 if it is to be absent or is irrelevant. The characteristics actually present in a given employee will be held in

```
DATA-EMP PIC 9(6) COMP SYNC RIGHT
```

The COBOL programmer reads an employee record and then ENTERS a PLAN subroutine called MATCH. This compares the contents of WANTED and DATA-EMP and if the required characteristics are all present sets a COMP SYNC RIGHT field INDIC to zero. If the required characteristics are not all present the subroutine sets INDIC non-zero. There is then a return to the COBOL program which branches to PRINT-PARA if INDIC is zero and prints out the employee's name as one who meets this particular set of requirements.

The first part of Figure 13 shows the COBOL statements required to call the subroutine. The language name PLAN and the subroutine name MATCH are given. The parameters needed are WANTED, DATA-EMP and INDIC. When writing the subroutine the PLAN programmer would state that he requires three parameters:

1 The address of the area holding the required pattern of characteristics; this has been defined in the program as WANTED.

2 The address of the area which holds the characteristics of a particular employee; this has been defined as DATA-EMP.

3 The address of the area to receive the result; this has been defined as INDIC.

The COBOL programmer makes sure that the areas are available, and that the first two will contain the relevant patterns prior to entering the routine. When the subroutine has been run through, the program branches back to

```
IF INDIC = ZEROS GO TO PRINT-PARA.
```

INDIC will then contain the required value.

The second part of Figure 13 shows the PLAN statements required in this subroutine.

As may be seen at the bottom of the figure, the ENTER verb compiles into four machine code steps, expressed here in PLAN. This is known as *standard linkage* and is used in 1900 software to connect sections of program written in different languages.

The step

```
CALL 1 MATCH
```

branches to the subroutine after storing the address of the next word in X1. This address is often referred to as *the link*. In that word is the next step which will have the general form

```
LDX 3 parameter-1
```

In this case it is a step loading the address of WANTED into X3. Each parameter specified after ENTER generates a step of this kind. Here there are three LDX steps, the second and third loading the addresses of DATA-EMP and INDIC into X3.

The subroutine's first step will be reached, as described above, with the address of the first parameter stored in X1. The subroutine can now obtain the parameters whenever it needs them, by preserving the contents of X1. All that is required is to specify

```
OBEY    0(1)
```

to load the first parameter into X3,

```
OBEY    1(1)
```

to load the second parameter, and so on. In the example, the address of WANTED is obtained at once by OBEY 0(1). The contents of WANTED are then extracted by modifying by X3. The address of DATA—EMP is now obtained by OBEY 1(1) and then the two patterns are merged together using a logical AND instruction. This will result in a zero answer in X4 if the required characteristics are all present. The result zero or non-zero is then stored in INDIC whose address has been obtained by the OBEY 2(2) instruction.

Finally, note the step

```
EXIT 1 3
```

3 is necessary in order to pass over the three parameter words when returning to the main COBOL program.

USING STANDARD ICL SUBROUTINES

A large number of standard ICL subroutines has been written, and these may be used by COBOL programmers. However, since the specifications are written for the use of PLAN programmers, a programmer knowing only COBOL may find some difficulty in using them. In general, the COBOL program must allocate the correct data areas, and these must contain the required starting values before entering the subroutine. An example of how this is done is given below.

Example: GRADPENSWEX

Consider the following specification of a routine to calculate graduated pension contributions for weekly paid employees. The name of the subroutine is GRADPENSWEX.

Note: The subroutine specification which follows is typical of the layout used for PLAN. This particular subroutine calculates an answer which was required during payroll calculations in the U.K. at one point in the evolution of the Tax Laws. The detail of the processing is not important, but it is necessary to appreciate the way in which the storage and parameter requirements stated here must be met in the COBOL program.

ISSUE 1

GRADPENSWEX

Title

Graduated Pension Scheme Subroutine (Weekly)

Description

The subroutine calculates the graduated contributions payable by an employee on weekly pay on an exact percentage basis, including the earnings-related supplemented contributions for employees in the scheme and those contracted out.

Input Parameters

The three locations after the CALL instruction in the main program should be set up as shown below:

```
CALL   1   GRADPENSWEX
LDN    3   SYMBOL1
LDN    3   SYMBOL2
LDN    3   SYMBOL3
```

or LDX 3 'SYMBOL1' etc. if the parameters are in upper storage.

SYMBOL1 will name the location which contains the current pay for this week in binary pence. Where holiday pay is paid in advance, the amounts for each week's holiday pay must be contained separately in locations SYMBOL1+1, SYMBOL1+2 and so on.

SYMBOL2 will name the location which contains the number of weeks' holiday pay contained in locations following SYMBOL1 (in binary).

SYMBOL3 will name the first location of three consecutive locations which are to contain the results on exit from the subroutine.

Before the subroutine is called, the correct information must be preset in the two locations named by SYMBOL1 and SYMBOL2. In particular, if an employee's current pay does not include any holiday pay paid in advance, then the location named by SYMBOL2 must be clear when the subroutine is entered. (SYMBOL1+1 etc. need not however be cleared if SYMBOL2 is zero). For employees contracted out, the sign bit (B0) of the location referred to by SYMBOL2 should be set to one and the subroutine then entered as before. The remainder of the location named by SYMBOL2 should contain the number of weeks' holiday pay or be clear, as for employees included in the scheme.

Results

SYMBOL3 will contain—for not contracted out entries—4¼% of the amount (up to £9) by which the gross pay in the income tax week exceeds £9, in binary pence (½d rounded down).

SYMBOL3 for contracted out entries will be zero.

SYMBOL3+1 will contain ½% of the amount (up to £21) by which the gross pay in the income tax week exceeds £9, in binary pence (½d rounded up).

SYMBOL3+2 will contain the rounded total of the 4¼% (if any) and ½% amounts before rounding, in binary pence (½d rounded down). For contracted out entries, the total comprises the ½% amount only (½d rounded down).

Use of Overflow

Overflow remains unchanged.

Entry Point

GRADPENSWEX

Link Accumulator

X1

It can be seen that three areas required in store are named in the specification as follows:

SYMBOL1 four words binary (assuming three weeks holiday pay)
SYMBOL2 one word binary
SYMBOL3 three words binary

These areas can be set up by the following entries in the Working—Storage Section of the main program.

```
01  SYMBOL2 PIC S9(4) COMP SYNC RIGHT.
01  FILLER.
    02 SYMBOL1 PIC 9(4) COMP SYNC RIGHT
       OCCURS 4.
01  FILLER.
    02 SYMBOL3 PIC 999 COMP SYNC RIGHT
       OCCURS 3.
```

The names used as ENTER parameters must be the same as the names of the areas defined in the Data Division. They will probably not be the same as those used in the specification, as standard names are used for all standard ICL subroutines.

The important point is that the storage should have the layout expected by the subroutine. This must be duplicated exactly, including starting on word boundaries as specified.

In the Procedure Division, after statements which put the current pay in SYMBOL1 (in binary pence) and set SYMBOL2 and SYMBOL3 as appropriate, the routine can be entered using the statement.

```
ENTER PLAN GRADPENSWEX USING SYMBOL1(1)
    SYMBOL2 SYMBOL3(1).
```

The next statement in the Procedure Division can be written assuming that the results defined in the specification will now be in the correct fields.

ICL has produced some subroutines especially for the use of the COBOL programmer, which are designed to overcome some limitations of the language or of the current compilers. These routines are written in PLAN, but their specifications are such that they can be understood by programmers knowing only COBOL.

For example
Subroutines to handle variable length data.

Variable length data handling This comprises a series of subroutines which perform two main tasks. Half move variable length input fields from paper tape or magnetic tape into fixed length fields for subsequent processing. The other half work in reverse, packing variable length output fields into a fixed length format for subsequent WRITing to paper or magnetic tape.

There is also a further subroutine, PUNCHN which allows the output of variable length records to paper tape. The normal WRITE order for paper tape results in the output of a fixed number of characters equal to the size of the largest record in the file, irrespective of the record type used.

The specifications for these subroutines are given in Appendix 7 of this manual. Some effort is needed to master them, since the problems handled by the subroutines are complex, with many variables. However, this effort is nothing to the effort required for a user to write his own subroutine dealing with these problems.

Example
As an example, consider the subroutine COBDIST (pages 295 to 303).

This requires five parameters, as follows:

1 The current address of the data.

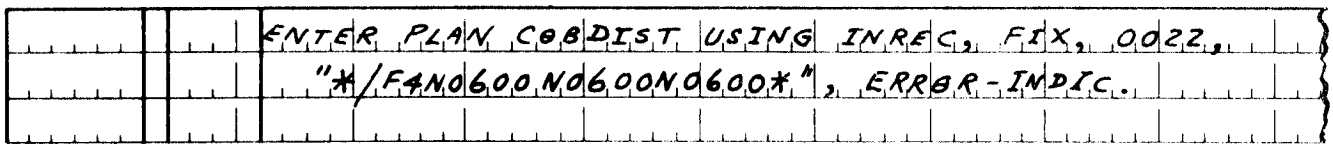
2 The name of the group field containing fixed length fields which are to receive the data.

3 The size of the receiving area.

4 A coded "map" of the format of the data, usually given in the form of a non-numeric literal.

5 The address of an error indicator field.

A typical entry point of COBDIST might be



This informs the subroutine that it is to locate the data in INREC and move it to FIX which is a group of fields with a total length of 22 characters. The data in INREC consists of four fields; the most significant is fixed length with four characters, and the remaining three are variable length integers of up to six characters each, with an asterisk marking the end of each field.
If an error is found, the subroutine will set markers in ERROR–INDIC. This group field should be tested by the statement immediately after the ENTER, to see if any errors have been detected.

COBOL SUBROUTINES

It is possible to write subroutines in COBOL for incorporation in a program. These subroutines are compiled by a special COBOL compiler whose output is in semicompiled form. In this form, a subroutine can be incorporated at consolidation time into a PLAN or COBOL program.

COBOL subroutines are compiled in the same way as full programs, using one of the current compilers.

Value of COBOL structures

There are two main uses of COBOL subroutines.

1 Sections of coding required by several programs need be written only once, and compiled once as a subroutine. Afterwards, the subroutines can be stored in a library and can be called into any program requiring it in semicompiled form. For instance, a subroutine to output data on the line printer might be written.

2 Large programs can be split into logical sections and each section written and tested as a separate subroutine. As explained in Chapter 9 "Testing usage programs", the testing may also require a controlling routine. Such an approach to large programs can speed up writing and testing considerably. The overall structure of the original flowcart can be planned so that a main routine calls in other portions of the program as required.

The next two sections consider how to write a COBOL subroutine, and how to call such a subroutine from a main program. An example is then given to illustrate this material.

Writing a COBOL subroutine

This section describes the way in which a COBOL subroutine must be written, where this differs from the writing of a normal program. Each division is considered in turn.

IDENTIFICATION DIVISION

The program name after PROGRAM—ID is not devised according to normal rules. It can be up to eleven characters long, of which the first must be alphabetic and rest alphabetic or numeric. (These are the rules for PLAN segment names.) It is advisable to keep this name as short as possible, so long as duplication is avoided. In particular, all subroutines must have different names.

ENVIRONMENT DIVISION

SELECT.....ASSIGN clauses are permitted; in other words, a subroutine can have its own files. However, it cannot use files belonging to the main program.

DATA DIVISION

The heading FILE SECTION must appear followed by the file description and record description entries for the subroutine's files. Record areas of subroutine files cannot be handled by the main program.

The Working—Storage Section can be used normally to define storage which the subroutine requires. These storage areas cannot be handled by the main program.

The Working—Storage Section must be followed by a further section required only by a subroutine. This is known as the *Linkage Section*, and defines storage areas which are used by both the main program and the subroutine. These areas are those defined by the main program which the subroutine can access and whose address is passed to the subroutine. Full details of each field must appear, as if it were being defined in a normal Data Division.

The main program branches into the subroutine using the CALL verb which will be considered shortly. The CALL statement includes parameters in the same way as does ENTER. These parameters must be 01 level fields, and are those fields which are defined in the Linkage Section.

Whenever these fields are referred to in the subroutine's Procedure Division, the coding generated must address the fields in a particular way. Since the storage areas have already been defined in the main program, no storage will be reserved for Linkage Section entries. Instead, when compiling subroutine statements handling a field, the compiler generates code which picks up the address of the field from the standard 1900 linkage between the main program and the subroutine. This address is obtained by the same method used for ENTER (see above). The field must be defined in the Linkage section so that the detail of the area can be handled correctly by the subroutine steps. This detail may be quite different from the definition in the main program or in any other subroutine.

In writing the Data Division of a subroutine, therefore, the programmer must define all data fields common to the main program in the Linkage Section. These fields will also be defined in the main program. Record areas and working storage required only by the subroutine are described in the File and Working—Storage Section.

PROCEDURE DIVISION

The Procedure Division of a subroutine is written in the normal way but certain points should be noted.

The subroutine will be entered from the main program at the first statement of the first paragraph. No assumptions about initial conditions should be made in the logic of the program. Conditions set last time the subroutine was used may or may not still hold. This depends upon whether the main program uses overlay, and whether the subroutine is held in an overlay unit. If it is, the subroutine's data areas are also overlaid and are brought in their original form each time the overlay unit is called. The simplest approach is to initialise all areas each time the subroutine is entered. Any markers, and so on, can be set in common storage.

Peripheral verbs which occur in the subroutine relate to the subroutine's files. The full sequence of OPEN, READ/WRITE and CLOSE must appear.

At the end of the subroutine, a return to the main program steps is achieved by a new statement,

EXIT PROGRAM.

The return is made to the statement following the CALL statement.

Calling a COBOL subroutine

To call a COBOL subroutine from a main program, the CALL verb is used.

The CALL statement has the format

CALL subroutine—name USING parameter—1 parameter—2 ... parameter—n.

Parameter—1, and so on, are parameters which must be the names of fields aligned on a word boundary in the Data Division of the main program.

As explained above these fields are common to main program and subroutine and will also appear in the Linkage Section of the subroutine. They must occur in the Linkage Section in the same sequence in which they occur in the CALL statement, although the names need not be the same. This last point is very important.

CALL and ENTER are basically the same except that CALL is more limited in types of parameter. The rule is that while CALL is used for subroutines written in COBOL, ENTER is used for subroutines written in any other language. Two verbs are provided only for reasons of compatability, and to allow better validity checking of parameters.

Example of a COBOL subroutine

The COBOL subroutine given in Figure 13 calculates square roots.

The relevant parts of the main program calling the subroutine are given second.

Two common areas are defined in working—storage, one WS—NUM which contains the original number to be worked on, and the other WS—SQRD will hold the result of the calculation. Note that these fields are also given in the Linkage Section of the subroutine.

A CALL statement is then given in the Procedure Division, to the subroutine SQRT. The common areas WS—NUM and WS—SQRD are used as parameters.

When the subroutine has been completed, a return is made to the statement.

If WS—SQRD < 15 GO TO PARA—18.

ICL

COBOL program sheet

title

programmer

sheet number

date

Sequence No. 1 6 7 8 11 12 15 20 25 30 35 40 45 50 55 60 65 70 72 73 75 80 Identification

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SORT.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE COMPUTER. ICL-1904.
OBJECT-COMPUTER. ICL-1904.
    MEMORY 2000 WORDS.
INPUT-OUTPUT SECTION.
FILE CONTROL.
DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
01  DIF                    PIC 9999V9(5) COMP SYNC RIGHT.
01  NUM                    PIC 9(4) COMP SYNC RIGHT.
01  ROOT                   PIC 999V9(5) COMP SYNC RIGHT.
01  RECIP                  PIC 999V9(5) COMP SYNC RIGHT.
LINKAGE SECTION
01  NUM-IN                 PIC 9(4).
01  NUM-OUT                PIC 999V9(5).
```

STOCK CODE 8053F FORM 14/41 (12.73)

ICL

COBOL
program sheet

title

programmer

sheet number

date

Sequence No. 1 6 7 8 11 12 15 20 25 30 35 40 45 50 55 60 65 70 72 73 75 80 Identification

```
PROCEDURE DIVISION.
PARA-1.
    MOVE NUM-IN TO NUM.
    MOVE 1 TO DIF.
PARA-2.

    DIVIDE NUM BY 2 GIVING ROOT.
    PERFORM LOOP UNTIL DIF < 0.00001.
    MOVE ROOT TO NUM-OUT.
    EXIT PROGRAM.

LOOP.
    DIVIDE NUM BY ROOT GIVING RECIP.
    ADD RECIP TO ROOT.
    DIVIDE 2 INTO ROOT.
    SUBTRACT RECIP FROM ROOT
             GIVING DIF
```


STOCK CODE 8053F FORM 14/41 (12.73)

Chapter 11 Direct Access

Introduction

Updating direct access files

Magnetic tape files are always updated *by copy*. That is, the amendments and the master file are held on separate tapes, while the amended records are copied onto a third tape, creating an updated master file. Updating by copy gives good *file security*, since if one of the three files is spoiled or destroyed it can always be re-created from the other two.

Direct access files can also be updated by copy. But when they are, the particular advantages of direct access will be lost—if the whole file is copied to another storage area, then every record will have to be processed, and the hit rate will be 100%.

So direct access files are usually updated *by overlay*. That is, each record that has to be amended is read from the master file, updated, then written back to the same place on the same master file. When overlay processing is being used extra attention has to be paid to file security. If the master file is destroyed, there will be no way of reproducing it, unless arrangements have been made for copies to be kept.

Real time systems

However, the overlay processing facility has the important advantage of making *real-time* processing possible.

With magnetic tape processing, amendments to a master file have to be *batch* processed. The amendments are collected over a period of time, until there are enough to make a processing run worthwhile. They then have to be *sorted* into an appropriate order, and used to update the master file serially, by copy.

But on direct access devices it is possible to keep a file more or less permanently on-line to the central processor, updating records whenever amendments are presented to the system. This is what is meant by 'real-time' processing. An example of this type of system would be an airline bookings system, which has to deal with a constant stream of amendments and enquiries from terminals in booking offices, all of which must be dealt with immediately. Without the direct access devices, real-time systems would not be possible.

In order to take full advantage of the direct access devices, direct access files must be organised in a rather more sophisticated way than is necessary for magnetic tape processing. Most of this chapter is taken up by descriptions of the different file structures in use. This means that programming for direct access can be a little more complex than programming for magnetic tape.

What do we mean by

Strictly speaking, the term *direct access* describes a method of accessing stored information. But in practice it is often used to refer to particular systems of backing storage in which this method of access is possible—usually systems based on *magnetic discs*. The term 'direct access' is frequently treated as though it were equivalent to 'magnetic discs' and in the rest of this chapter discs will be our main concern.

The direct access devices share some of the features of the magnetic tape devices, which are the other main form of backing storage. For example, in both cases information is recorded as patterns of magnetized spots in a thin film of a magnetizable oxide.

But the two types of backing storage differ fundamentally in that storage areas on direct access devices are divided up into units that can be addressed individually by the hardware, and consequently can be given a kind of symbolic address by the software. In this respect direct access storage is more like core store than magnetic tape. On magnetic tape, the only way of locating a specific piece of information is to search through the whole tape until you find it—access to the information is 'serial'. But on a direct access device, the information can be retrieved 'directly', as long as it is possible to work out the address of the area in which it is stored. This is the origin of the term 'direct access'.

The advantages of direct access

In applications where only a few records on a file need to be read during each processing run, direct access will be much faster than the serial access used with magnetic tape—less time will be wasted searching through unwanted information. In addition, direct access devices are designed so that the amount of mechanical movement involved in accessing data is much less than for magnetic tape. So an important advantage of the direct access devices is that access to information can be very fast.

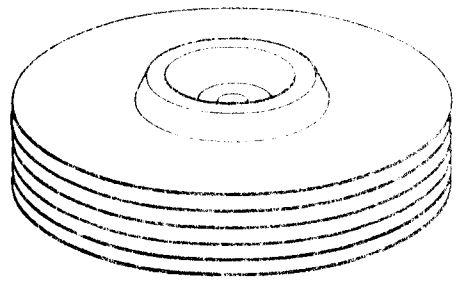
However, where a large proportion of the items on the file need to be processed on each run (that is, where the *hit rate* is high), direct access may not be so very much faster than serial access—if most of the records have to be read anyway, the time taken up searching through the few unwanted records will not be significant. Direct access storage is more expensive than magnetic tape, so in applications where direct access has no particular advantage over serial access (where the hit rate is high) magnetic tape is generally used.

EXCHANGEABLE DISC STORES

Cartridges

Of the direct-access peripherals, Exchangeable Disc Stores are by far the most commonly used. In these devices, data is recorded on the surfaces of metal discs, which are coated with magnetizable oxide. The discs are assembled in *cartridges*, which may be stored in a library when not in use.

Both surfaces of each disc are used for recording data, except the top and bottom surface of each cartridge. A cartridge of 6 discs looks something like this:



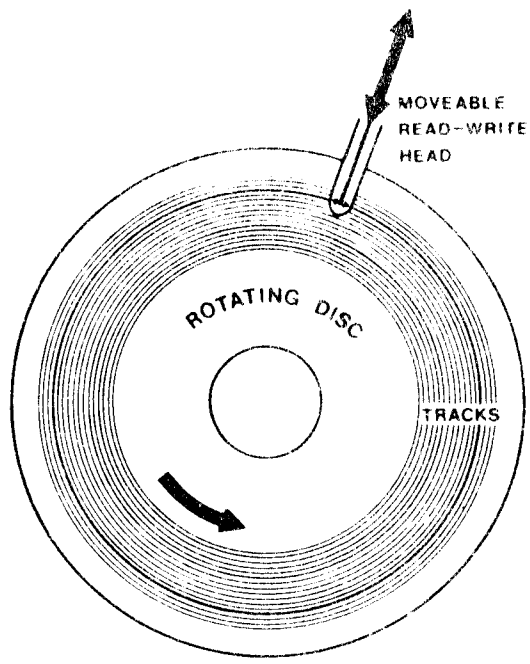
Disc transport units

When a cartridge is to be used for reading or recording data, it is placed on a *transport unit* (or *disc handler*). This unit spins the cartridge at a steady speed, and operates the moving *read-write heads*. As their name suggests, these heads write, read and erase magnetic marks on the disc surfaces.

The transport unit has one read-write head for each recording surface on the cartridge. The head can be moved to and fro across the rotating disc surfaces, permitting rapid access to any part of the stored data.

Tracks

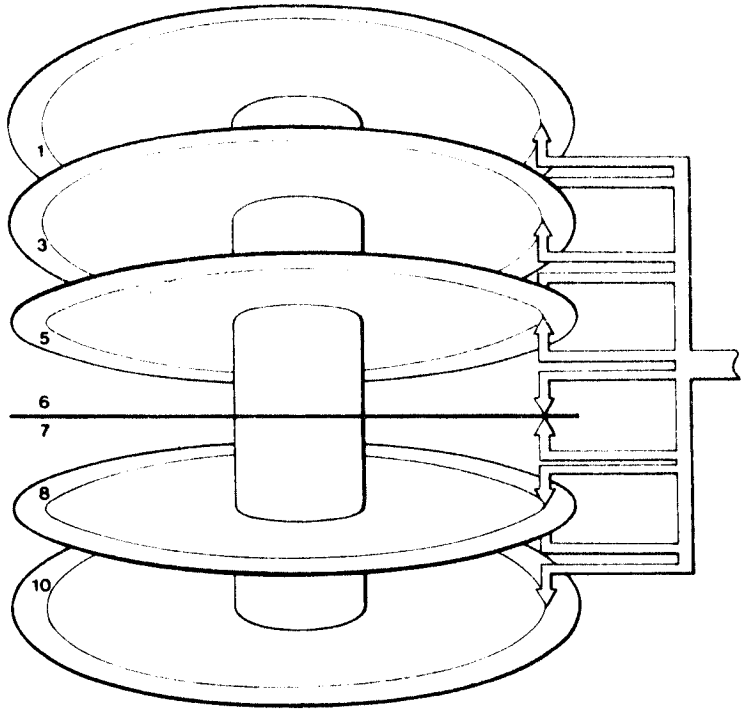
The positions to which the read-write heads can move are controlled by the transport unit mechanism. The magnetizable oxide forms a continuous film on the disc surfaces, but the data is only recorded around predetermined *tracks*. Each track is a circle and its radius corresponds to one of the possible head positions.



Cylinders

On a transport unit all the read-write heads move in unison. So at a given moment the heads can access one track on each disc surface, and each of these tracks has the same radius. The set of tracks which can be accessed on a cartridge at a particular head position is known as a *cylinder*.

For example, on a six-disc cartridge, which has 10 recording surfaces, a cylinder consists of 10 tracks:



The importance of the cylinder in direct access will become clear later in this chapter.

Access times

Using direct access it is possible to read or write data at any point on the disc. The time it takes to do this will depend mainly on:

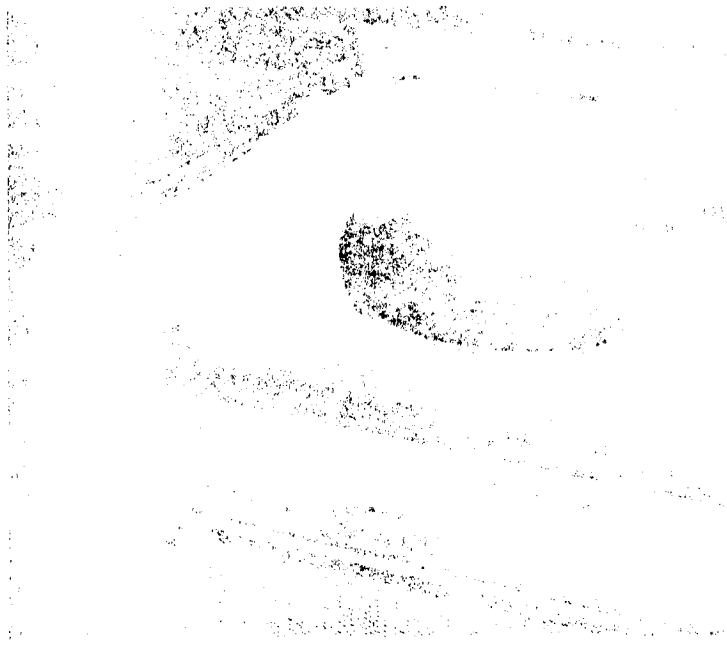
1. The time it takes for the read/write heads to move to the correct seek area known as the seek time.

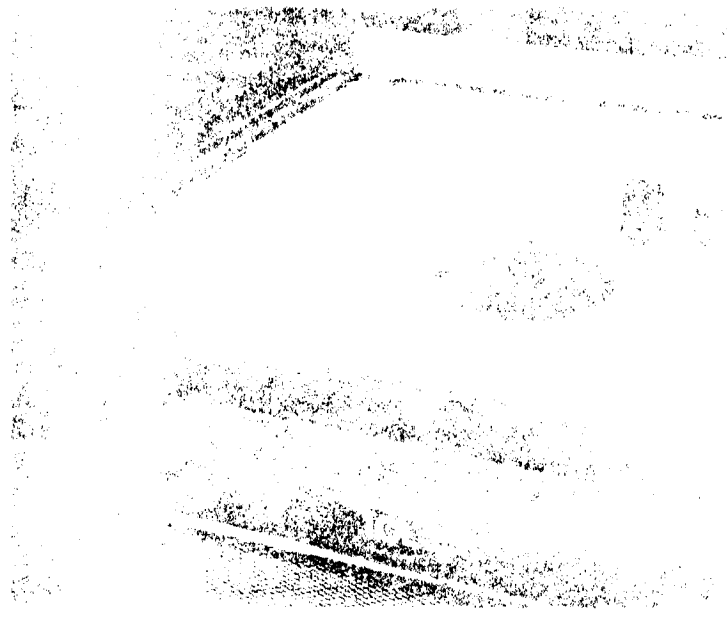
2. The time time disc takes to revolve to the required block.

The seek time varies with the number of tracks to be 'jumped' but an average seek time can be calculated. The average time for step 2 (called the latency) will be half a revolution The average seek time and the latency can be added together to give the average access time.

The total time needed for reading or writing will also depend on the *transfer rate* of the peripheral and of course, on the number of characters to be transferred.

Writing takes longer than reading. This is because disc peripherals automatically check what they write by reading it back again. This means that each writing operation takes one extra revolution of the disc.





TYPES OF EXCHANGEABLE DISC STORE

EDS 8

EDS 8 cartridges consist of six discs which provide ten recording surfaces. Each surface has 203 tracks and the total storage capacity is over eight million characters. Each transport unit can hold one cartridge and is linked via a control unit to the central processor. One control unit can handle up to eight transports. The average seek time is 85 milliseconds and the average latency 12.5 milliseconds, giving an average access time of 97.5 milliseconds. The transfer rate is 208,000 characters per second.

EDS 30, EDS 60

EDS 30 and EDS 60 can be used with any processor from 1902A upwards. The cartridges consist of eleven discs and provide twenty disc surfaces. EDS 30 has 203 tracks and a storage capacity of over thirty million characters; EDS 60 has 406 tracks and a storage capacity of over 60 million characters. The cartridge is mounted on a transport unit. The EDS 30 transports are housed in cabinets in clusters of 1, 3, 5, 7 or 9 units depending on the version. Each EDS 60 transport, however, is a separate free-standing cabinet. The control units for both systems can each handle up to nine transports. The average access time for EDS 30 is 72.5 milliseconds, and for EDS 60, 47.5 milliseconds. The transfer rate is 416,000 characters per second.

EDS 200

This cartridge also consists of eleven discs but one surface is pre-recorded with information as to which tracks are defective; this leaves 19 usable surfaces. It has 822 tracks and a storage capacity of approximately 200 million characters. The average access time is 38.3 milliseconds and the transfer rate is the same as for EDS 30s and EDS 60s.

Twin Exchangeable Disc Store

The Twin Exchangeable Disc Store (TEDS) is for use with the 1901A processor. The transport handles two cartridges simultaneously, each cartridge consisting of 2 discs. There are two versions, one giving 100 tracks per disc surface, the other 200 or 203. The average access time is a little slower than EDS, but the transfer rate is the same. The cartridge capacity is much smaller since there are only 2 recording surfaces per cartridge.

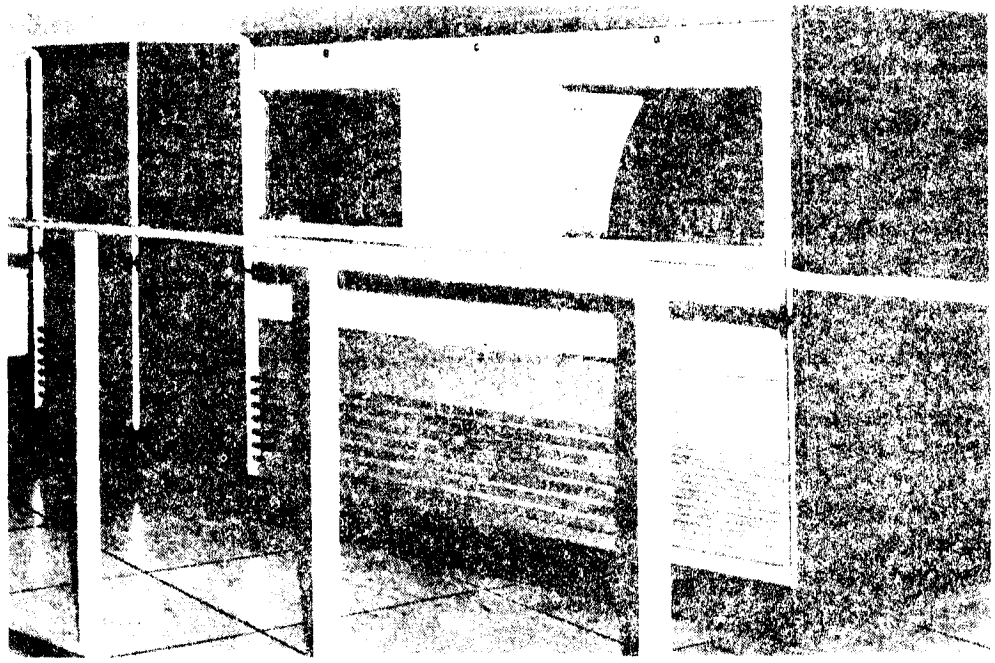
TEDS peripherals have now been replaced by EDS, which can also be used with 1901A and 1902A machines.

FEDS

The FEDS are for use with the 2903/2904. The cartridge consists of one disc only, but both surfaces are used for recording. Each surface has 406 tracks and a storage capacity of approximately five million characters.

FEDS cartridges are always used in pairs i.e. 2 cartridges (2 discs) on a common spindle, but treated as a separate unit. The lower disc is fixed and is supplied as part of the device, only the upper disc is exchangeable.

FIXED DISC STORES (FDS)

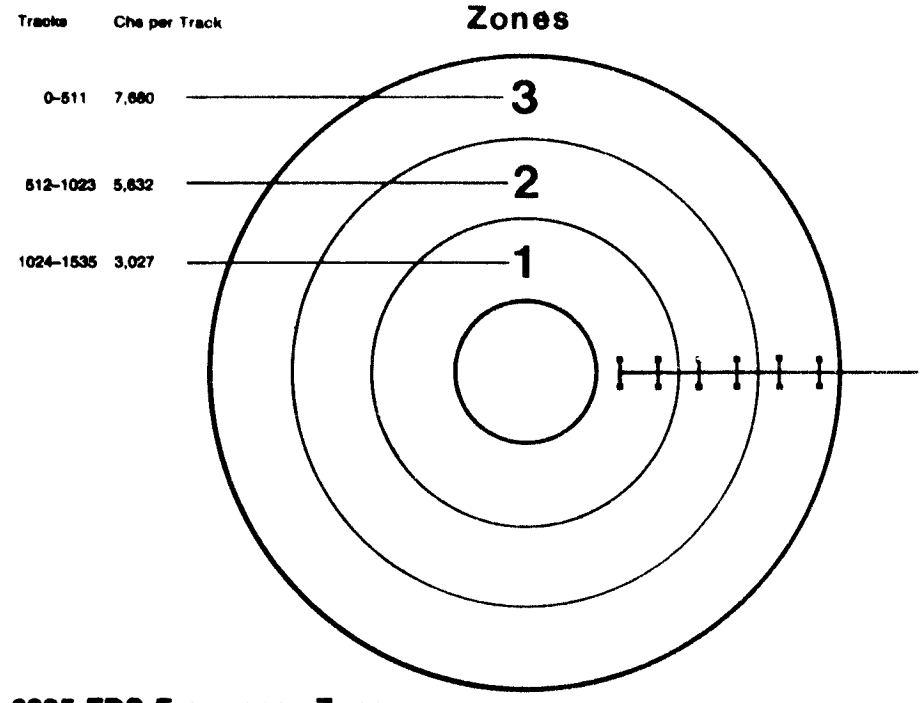


The Fixed Disc storage device

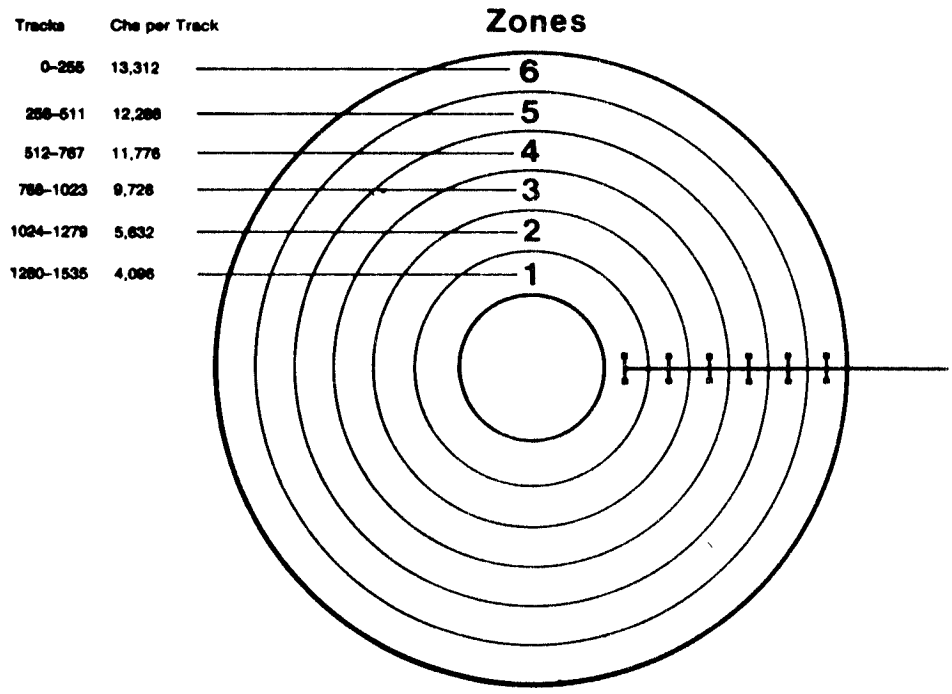
As the name implies, Fixed Disc peripherals use discs which are permanently installed in the machine instead of in exchangeable cartridges. They provide a much larger on line storage capacity than EDS devices. A fixed disc store may be used with 1904 (32K) or bigger processors.

There are several versions of the Fixed Disc Store, the largest containing 26 discs and capable of recording 741,343,232 characters. The discs themselves are bigger than exchangeable discs. There are six read-write heads for each recording surface, each head accessing 256 tracks.

The recording surface is divided into *frequency zones*, each zone having a different storage capacity per track (and hence a different transfer rate). On one type of Fixed Disc peripheral (the 2805 Fixed Disc Store) there are three zones of 512 tracks, with two heads accessing each zone. On others (2806 Fixed Disc Stores) there are six zones of 256 tracks, with one read-write head for each zone.



2805 FDS Frequency Zones



2806 FDS Frequency Zones

MAGNETIC DRUM STORES

The basis of a Magnetic Drum Store is a rotating cylindrical drum whose outer surface is coated with magnetizable oxide. Data is recorded in parallel tracks around the curved surface, and there is one read-write head for each track.

Access to data on the drum is very fast, since no head movement is necessary. In one version the latency – and hence the average access time – is only 6.3 milliseconds.

The applications of Magnetic Drum Stores are rather specialised. They may be used, for example, for Executive and Operating System overlays – typically in an installation using George III.

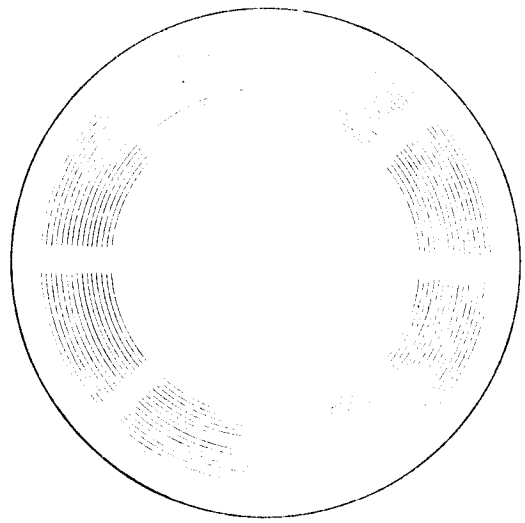
STORAGE AREAS ON DISCS

On page 199 we described how information is stored in *tracks* on a disc surface. Taken together, all the tracks make up a cartridge's total 'storage area'. This area is thought of as beginning at the outermost track (track 0) on the top recording surface and finishing at the innermost track on the lowest recording surface.

Blocks

The amount of data that can be stored on a track depends on the type of disc unit. For fixed discs, it also varies from one zone to another. To simplify matters, each track is divided into *blocks* of 128 words capacity.

On EDS 8 there are eight blocks to each track, arranged like this:



As the physical size of the blocks decreases towards the centre of the disc, the packing density increases.

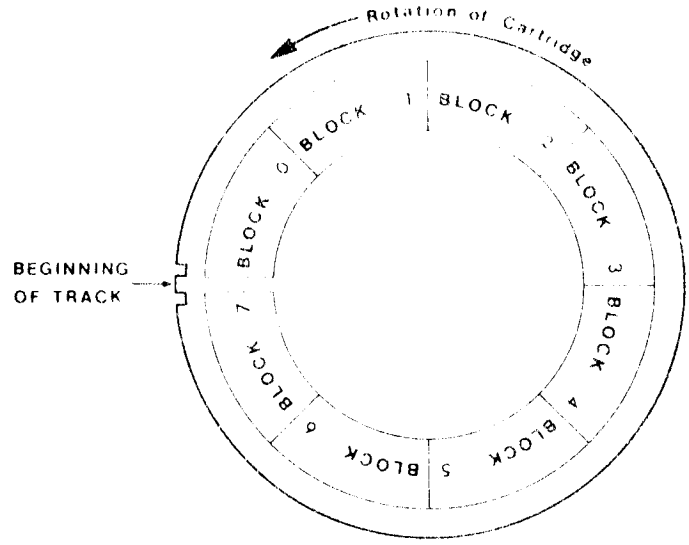
EDS 30 and EDS 60 have 15 blocks to each track.

FEDS have 12 blocks to each track.

Note that the direct access block is a different concept from a 'block' as used in magnetic tape processing. In direct access, the block is simply a constant unit of storage space, fixed by the hardware. For magnetic tape the block size is chosen (within limits) by the user; (it is more akin to the 'bucket' described later in this chapter).

The *inter-block gaps* contain information used by the hardware and by Executive. This information includes a 'hardware address' for each block. This is necessary because direct-access processing usually involves jumping from one block to another in a non-adjacent part of the cartridge. The hardware system also identifies the beginning of each track by a double notch on the base of the cartridge, which is detected photo-electrically.

We can represent the blocks on one track and their hardware addresses like this:

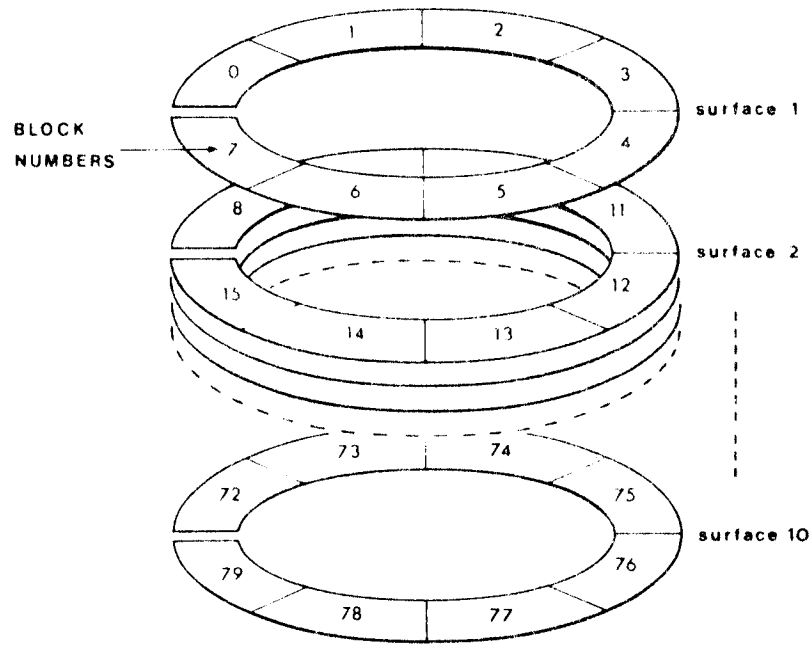


Seek areas

So far we have considered blocks in a single track. In practice, blocks are grouped into larger units known as *seek areas*. A seek area consists of all the blocks in one cylinder.

In fact the two terms—cylinder and seek area—are virtually interchangeable. Strictly speaking, the cylinder is a hardware concept: the physical area defined on a cartridge by a particular position of a read-write head. The seek area is a software concept—a unit of available storage area.

An EDS 8 seek area consists of 80 blocks arranged like this:



Block numbers and seek area numbers

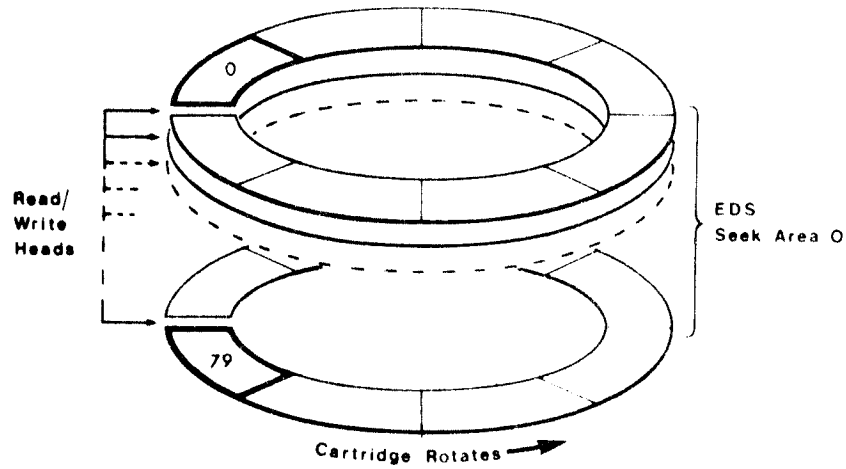
The blocks are numbered from 0 to 79 in the first EDS seek area; then, starting at the top of the next cylinder inwards, from 80 to 159 in the second area; then from 160 to 239 and so on.

The seek areas themselves are identified by numbers; an EDS 8 these run from 0 to 202, going upwards.

The reason for using seek areas

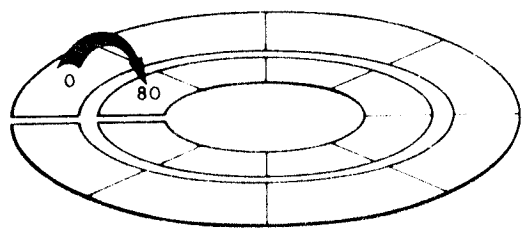
The seek area is chosen as a subdivision of storage area because all the blocks in one seek area can be accessed without head movement.

Suppose we needed to jump from block 0 to block 79:



There is no head movement, and the time taken to switch from one read-write head to another is negligible, so all we need is the time taken for the disc to rotate: 25 milliseconds. (The *average* time to reach another block in the same seek area—the latency—is 12.5 milliseconds).

Now consider a jump from block 0 to block 80:



In this case, the heads have to move mechanically to the next track, which takes 30 milliseconds. During this time the disc has been rotating, and it will take another 20 milliseconds for the start of block 80 to come under the head, giving a total of 50 milliseconds. (The *average* time to access a particular block on an adjacent track is 42.5 milliseconds; it takes even longer to reach non-adjacent tracks.)

Thus although the tracks in a cylinder are far apart in terms of physical distance, they are close in terms of access time.

EDS 30 and EDS 60 seek areas

EDS 30 and EDS 60 have 20 recording surfaces with 15 blocks to a track, giving a seek area of 300 blocks. Only 296 of these are usable since seek areas are a multiple of 8 blocks. The 4 blocks at the "right hand end" of each seek area cannot be used.

FEDS Seek Areas

The two discs are treated as separate cartridges, each cartridge having two surfaces with 12 blocks per track and, therefore, 24 blocks per seek area.

EDS 200

Although the cartridge has 11 discs and therefore 20 usable surfaces, one surface is pre-recorded with information about flaw areas and is not considered to be part of any seek area as far as the user is concerned.

Therefore each seek area comprises 19 tracks of 25 blocks each, giving a total of 475 blocks. However 475 is not a multiple of 2, 4 or 8 so that the last 3 blocks are not used in order to give a seek area of 472 blocks.

HOW FILES ARE ARRANGED IN THE STORAGE AREA

An important feature of direct access storage is its flexibility—the ease with which a file of data can be expanded or contracted during processing. When a file is written on to a disc, it is usual to leave room for expansion. A specific storage area is *allocated* to the file, but the actual data may only occupy a certain proportion of that area.

The allocation of disc storage space is carried out by ICL software. The software always allocates storage in multiples of 8 blocks.

Storage space on a disc can be allocated either to a *scratch file* or to a *permanent file*. A scratch file is an area available for the temporary storage of data during the running of a program. Permanent files may consist of ordinary data records, tables of constants, program instructions etc.

File areas

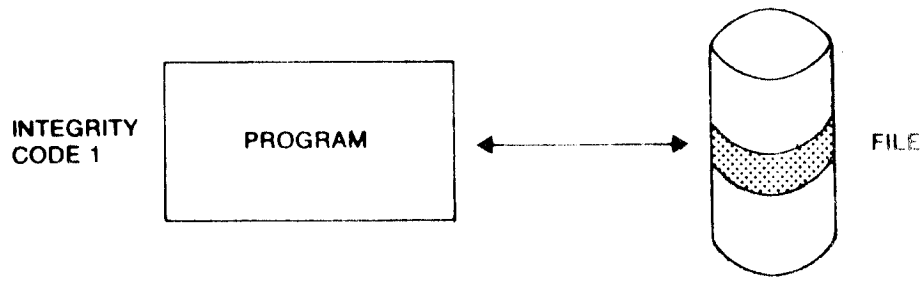
It is rare for a file to conveniently fill one cartridge. A large file may spread onto several cartridges, and there may be room for a number of small files on a single cartridge.

When there is more than one file stored on a cartridge, each file is allocated one or more separate areas. Each of these areas is known as a *file area*. A file area cannot extend from one cartridge to another, so if a file spreads across several cartridges, the file must be divided into at least a corresponding number of file areas. If a file has more than one file area it is known as a fragmented file.

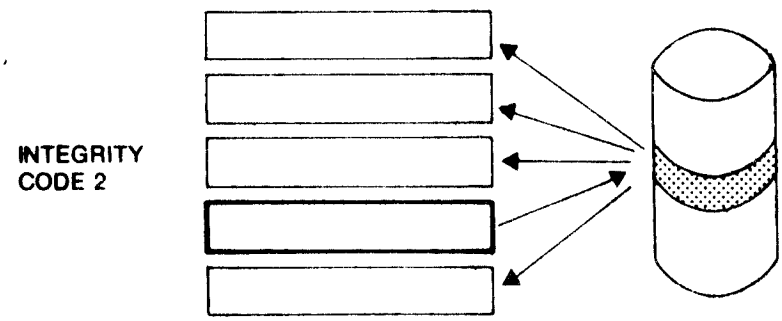
Integrity codes

It is possible for several programs to have the same file open for processing at the same time. This multiaccess is controlled by executive using a system of codes called integrity codes.

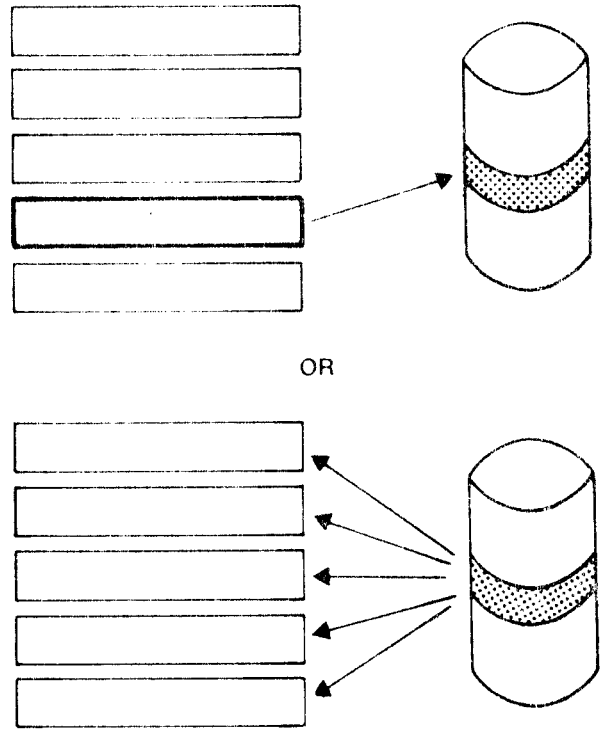
Code 1 is used for general purpose processing of data files by user programs. It permits one program at a time to open the file; until that program closes the file, no other program can open it.



Code 2 is generally used for files which are used in real-time systems (e.g. a booking system). Any number of program can open the file, but only one program at a time can write to it. While one program is writing to the file, others may read from it.



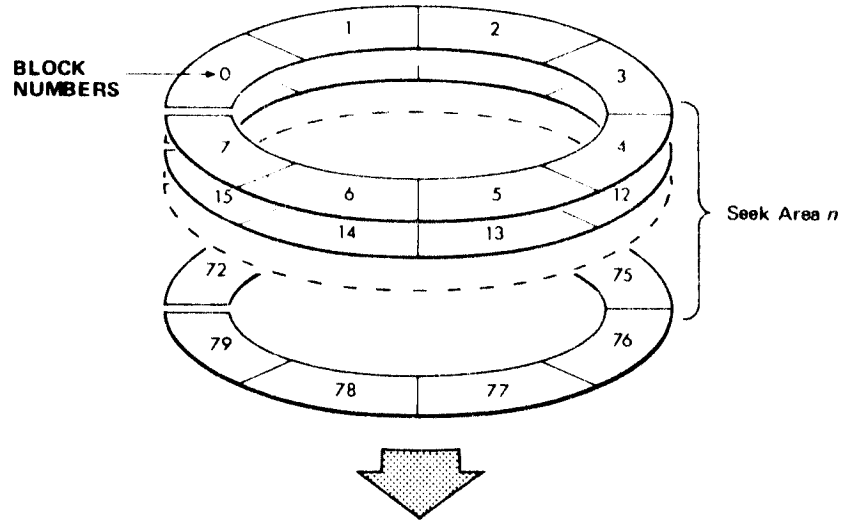
Code 3 is normally used for library files. It allows you to process the file in one of two ways: either one program can write to the file, or any number of programs can read from the file. You can change from one way to the other, but in between you must close the file.

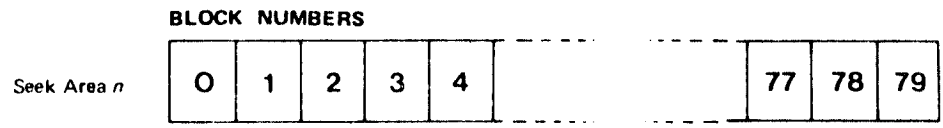


FILE MAPS

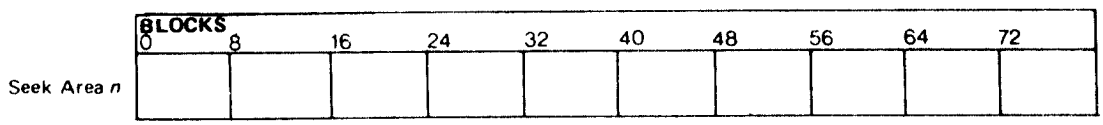
A *file map* is a diagram showing how files are on a cartridge and what areas they occupy.

For programming and systems design, we are not interested in the physical shape of the storage areas or in inter-block gaps. So a seek area can be 'opened out' into rectangular form:

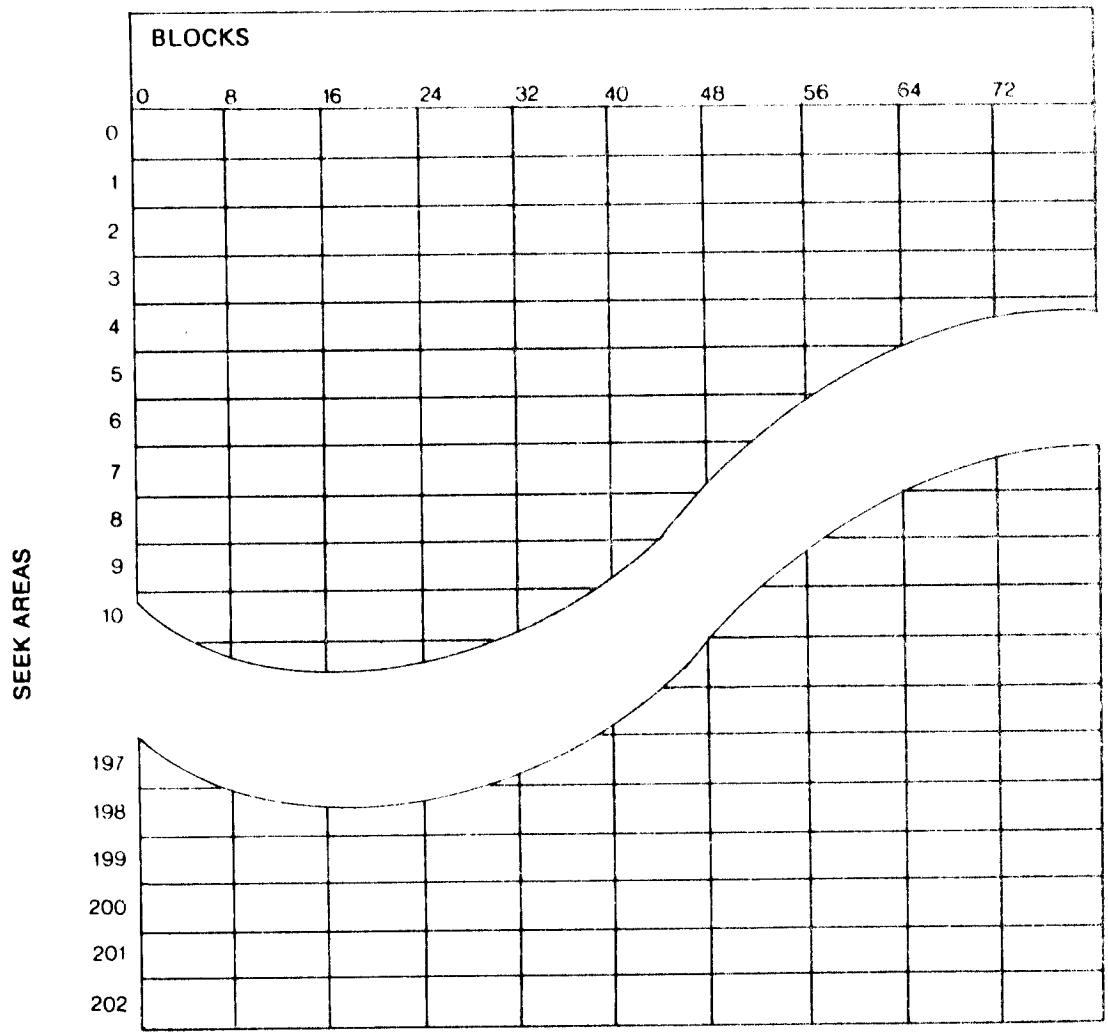




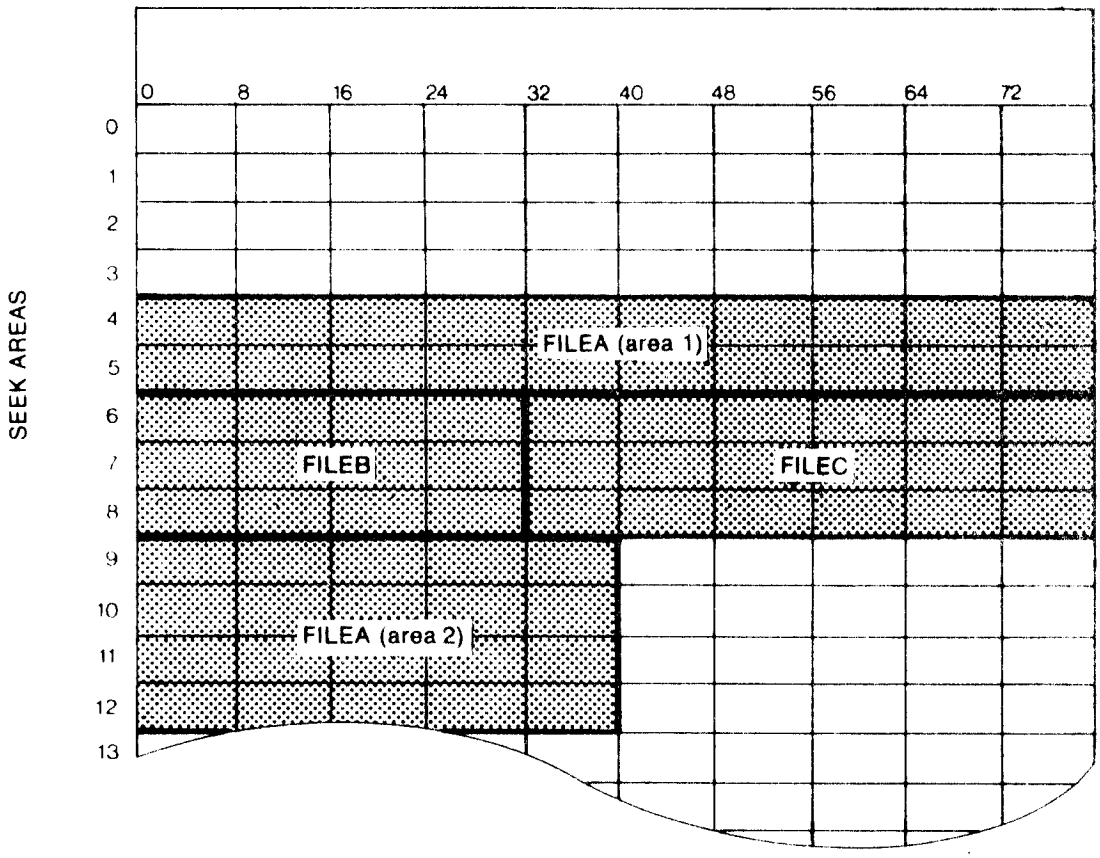
Since the storage area may only be allocated in multiples of 8 blocks, we can condense our 'seek area map' to this:



We can now show subsequent seek areas as further lines, and the file map for a whole EDS 8 cartridge will take this form:



Allocated space can be shown on the file map by shaded rectangles:



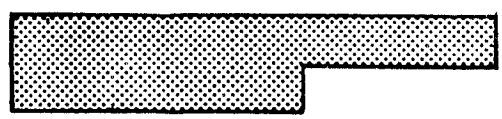
This example can serve to illustrate several further points about file areas:

A particular file may occupy more than one separate area on the same disc: for example, FILEA(area 1) and FILEA(area 2).

Seek areas may be shared by different file areas. For example seek areas 6, 7 and 8 are shared by FILEB and FILEC. These are known as *part-width* file areas.

It is more usual to allocate *whole-width* file areas such as FILEA(area 1).

A file area must be the same width throughout. In other words, it must be rectangular on the file map, not shaped like this:



A file area can be defined by a start and finishing seek area and a start and finishing block number. For example, FILEA(area 2) is specified by:
Start seek area 9; finishing seek area 12; start block 0; finishing block 39.

RESERVED STORAGE AREAS

One or more seek areas at the beginning of every cartridge will contain a System Control Area. This area is *reserved*, which means that it cannot be allocated for user files. The System Control Area contains:

details of all files which begin on that cartridge;

descriptions of allocated and unallocated areas on the cartridge;

a *flaw area* to which the contents of any flawed block on the cartridge can be written;

a *flaw index* which indicates the addresses of flawed blocks and their replacements in the flaw area.

The System Control Area is itself a file; it is allocated when the cartridge is initialised and its filename is always ICLSCAFILE.

The size of ICLSCAFILE depends on the type of storage unit and on whether or not the unit is to be used to hold Executive or operating system overlays. (When the overlaid Executive is loaded, it will request a storage unit to write its overlays to. The operator can indicate which unit is to receive the overlays by means of a console directive.)

ORGANISATION OF DATA FOR TRANSFER AND PROCESSING

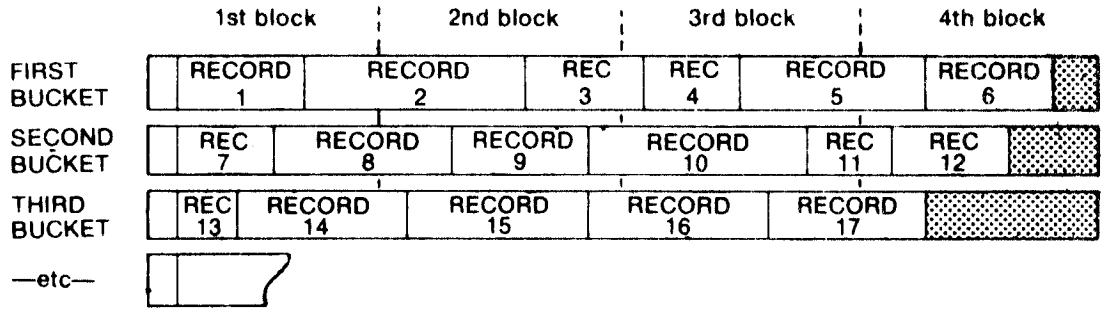
Buckets

A *bucket* is the basic *unit of transfer* in direct access processing. In other words, data is transferred to and from core store one bucket at a time.

A bucket may consist of 1, 2, 4 or 8 blocks, and this size is constant throughout a file. The systems designer chooses the most appropriate bucket size when the structure of the file is first decided. The choice is governed by a number of considerations, one of which is the most economic use of disc store.

Since the unit of transfer is one bucket, it is normal to think of a file divided into so many buckets, rather than so many blocks, for the purpose of processing.

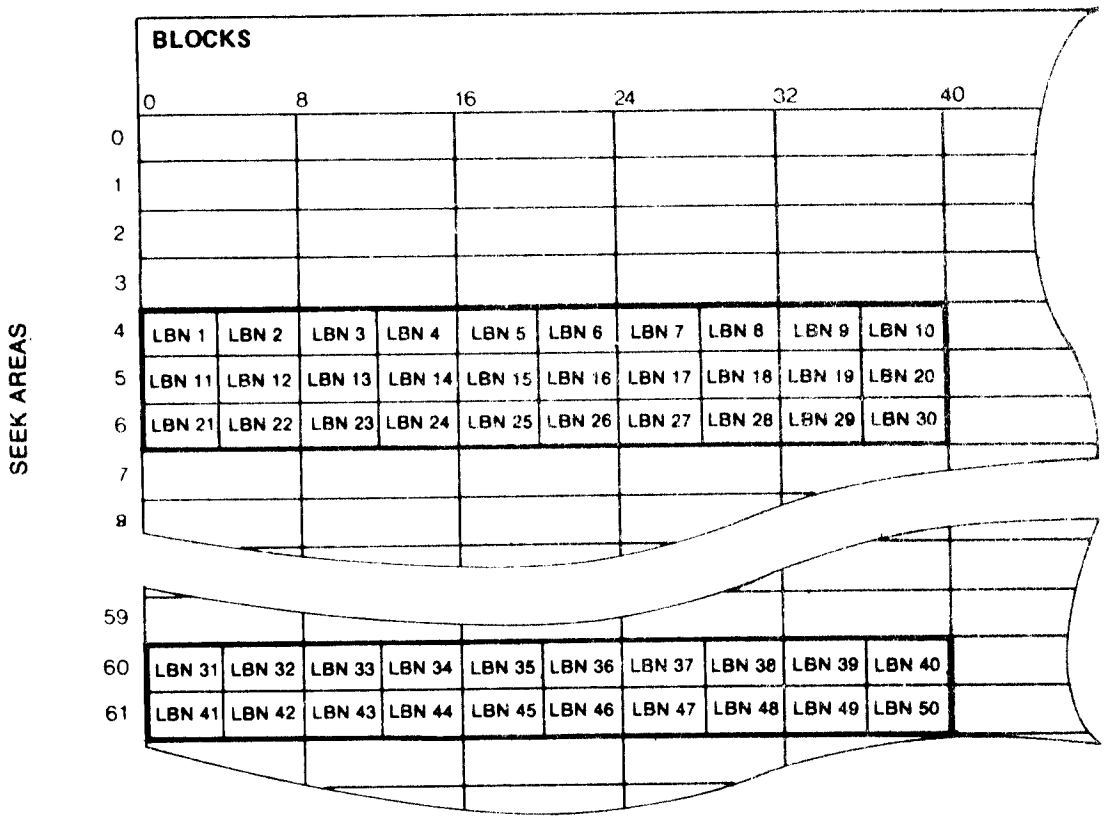
A bucket may contain a number of data records. The records may spread from one block to another, but not between buckets. This diagram shows records stored in 4-block buckets:



Logical Bucket numbers

Within a file, each bucket is assigned a serial number, called the *logical bucket number*. Numbering starts at 1 within a file and continues in ascending sequence from the first bucket in the first seek area of the first file area to the last bucket of the last seek area of the last file area (if there is more than one).

For example, FILEA(area 1) occupies seek areas 4, 5 and 6, blocks 0 to 39; FILEA(area 2) occupies seek areas 60 and 61, blocks 0 to 39. The bucket size is 4 blocks. This file map shows the logical bucket numbers (LBN's):



The logical bucket numbers still go in ascending order through subsequent file areas on separate cartridges.

Addressing buckets

Once a file has been allocated to specific areas, we are no longer concerned with block numbers. To access a particular bucket on a cartridge, the programmer need never specify more than the *file name* and the logical bucket number. Executive can then find the correct bucket, via the system control area, which indexes file names.

Housekeeping

Direct Access Housekeeping is the software package used for processing direct-access files. To make it possible to use Housekeeping, the contents of the buckets must comply with certain information standards. What follows is a very brief outline of these standards.

The current (1978) version of this package is DAHK Mark III. There are major differences between this and earlier marks, which are dealt with in a separate section at the end of this chapter.

Bucket headers

Every bucket must begin with a *bucket header* which is 2 or 3 words long. The remainder of a bucket after the bucket header is available for storing ordinary data records.

Record keys

Each record normally has a *record key* by which it can be identified. The key must be different for every record in the file, but its length and its position in the record must be standard throughout the file. The length of the key for a given file may be anything up to 64 characters.

Often a field such as a part number (in a stock file) or a staff number (in a payroll file) is used or adapted as a record key.

First word of record

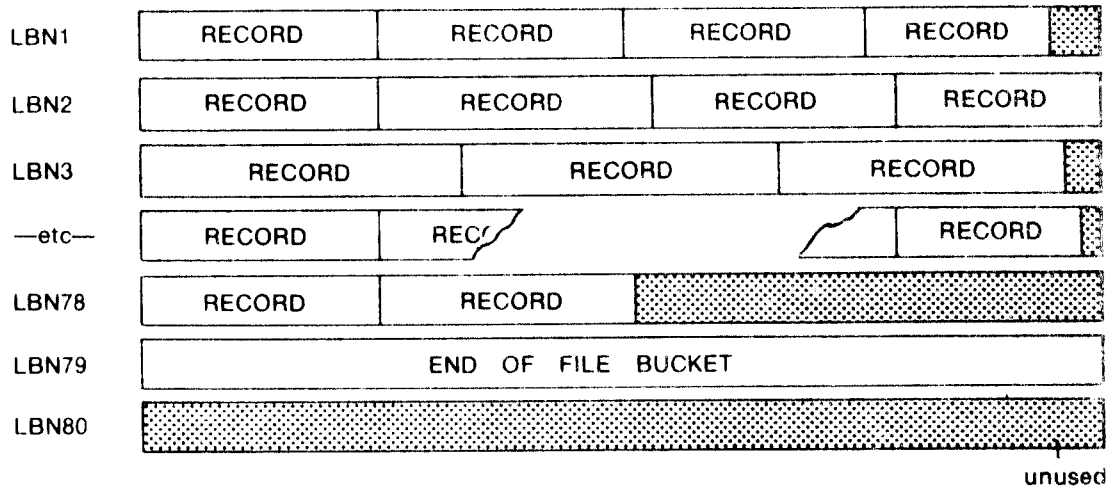
The importance of record keys will become clear in subsequent chapters of this manual.

The first word of every record is reserved. It must contain a *word count* indicating the length of the record in words (including the first word).

THE CHARACTERISTICS OF A SERIAL FILE

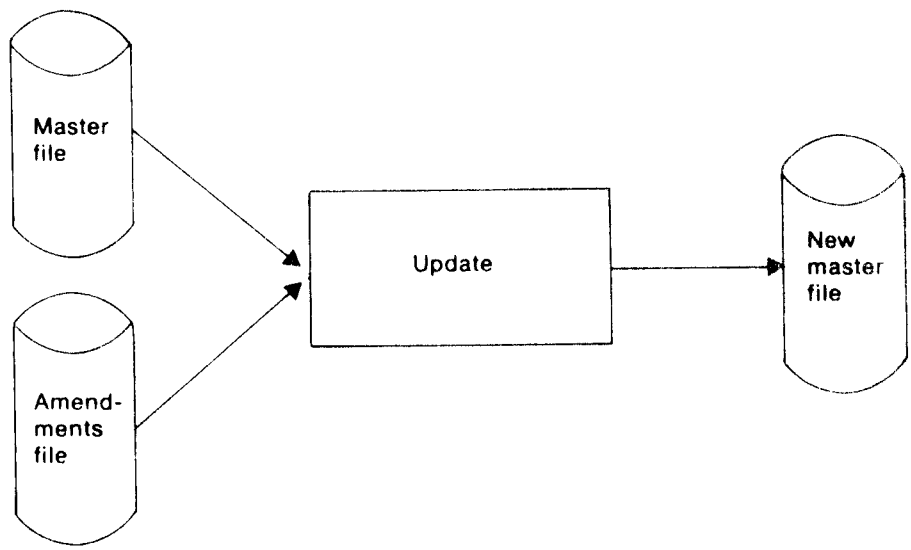
Serial files are the simplest type of disc file. Records are stored in the order in which they were transferred to the file, and are usually read back in the same order. Each bucket is filled with as many records as will go into it, so *bucket packing density* may be up to 100% (i.e. no spare space in the bucket). The end of the file will be marked by a special *end-of-file bucket*.

Here is a diagram showing how a serial file occupying 78 buckets of an 80 bucket area might be laid out:



Updating serial files

Serial files can usually only be updated 'by copy'—the master file and amendments file are processed together, starting at the beginning and working through to the end. The updated records are written to a third file, the new master file:



However, it is possible to update records by *overlay* (reading them into store, amending them, and writing them back to the same place on the master file) as long as the amendment does not involve changing the size of the record.

Records can only be added at the end of a serial file. The end-of-file bucket is overwritten by the first of the new records, and is rewritten after the last.

The uses of serial files

A serial file is generally accessed *serially*, that is each record is retrieved in turn according to its physical order on the file. For data files a serial structure may be preferred to other file structures where the hit rate is going to be very high.

A serial structure is essential for most system files: program and subroutine libraries, workfiles, files associated with GEORGE or with a compilation system. A program file should have a four character filename, 1-block buckets and an integrity code of 3.

Programming for direct access devices

The COBOL verbs used in the context of direct access programming are known as mass storage verbs.

USING MASS STORAGE VERBS

Defining a serial file

Handling a serial file with mass storage verbs is very similar to programming magnetic tape.

In the INPUT-OUTPUT section a serial file is allocated to a direct access unit with this format:

SELECT file-name ASSIGN TO EDS integer-1

The hardware name EDS is used for both Exchangeable Disc Store and FEDS. As with MT the integer must be in the range 1—63 for each direct access device. (1—15 for DAHK Mark I).

The RESERVE 1 clause does not double buffer for a file on a disc device. It has a different meaning and must never be specified for a serial file on disc.

In the I-O-CONTROL paragraph you may identify a field to be used for dealing with exceptions:

APPLY data-name TO TRANSFER-REPLY ON file-name.

Data-name is a PIC 9(6) COMP SYNC RIGHT field in the DATA DIVISION. This must be tested by the program if an exception could occur after any processing verb e.g. OPEN, READ. The field will contain zero if no exception has occurred. Different values of the exception code and the verbs to which each applies are on page 224.

The FD paragraph of the DATA DIVISION is similar to magnetic tape.

For disc files the usual format of the BLOCK clause is:

BLOCK CONTAINS integer CHARACTERS

BLOCK here does not refer to the hardware block on the disc device. The clause specifies the size of the bucket transferred to the input buffer. It uses the character option because the bucket contains a fixed number of characters rather than records (it contains 1, 2, 4 or 8 hardware blocks of 512 characters).

For example the COBOL coding for a disc file with 4-block bucket size would be:

BLOCK 2048

The ACTIVE-TIME clause may be used for an output file to specify a *data retention period*, that is the number of days that must elapse before the file may be overwritten.

The WITH GENERATION—NO clause is used if a generation number is to be checked, in the same way as for magnetic tape.

Below is an example of how to define a serial file.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT AAA-SERIALIN ASSIGN EDS 1.
I-O-CONTROL.
    APPLY MAA-REPLY TO TRANSFER-REPLY ON AAA-SERIALIN.
DATA DIVISION.
FILE SECTION.
FD  AAA-SERIALIN
    RECORDING MODE F
    BLOCK 1024
    LABEL RECORDS STANDARD WITH GENERATION-NO
    VALUE OF ID "MASTER  "
    GENERATION-NO IS MAB-GEN.
01  AAA-SERIALIN                    PIC X(72).
WORKING-STORAGE SECTION.
01  MAA-REPLY                       PIC 9(6) COMP SYNC RIGHT.
01  MAB-GEN                         PIC 9(6) COMP SYNC RIGHT
                                    VALUE 0.
```

Processing a serial file

Input and output files are opened by the OPEN verb with the same form as for other peripherals i.e.

OPEN INPUT file—name—1 OUTPUT file—name—2 . . . .

Also, for adding to existing serial files, you can open a file at end with the clause:

OPEN OUTPUT file—name—1 AT END.

To read records on a serial file, the READ verb has exactly the same form as for files on other media.

READ file—name AT END imperative statement.

The first READ in the program causes a bucket of records to be transferred to store and then unpacks the first record from the bucket and places it in the user's record area. Each subsequent READ causes housekeeping to unpack one record into the user's record area. When the first bucket has been completely unpacked, the next physical bucket will automatically be brought into store and so on.

Records are written serially by the WRITE verb, i.e.

WRITE record—name.

The CLOSE verb is used to close serial files, with the same form as for files on other media, i.e.

CLOSE file—name—1 file—name—2 . . . .

When used with an output file it will also write a short bucket and the end-of-file bucket.

If you want the file retained for further use by the program use the WITH RETAIN option. To ensure that the file is not accessed again you may write WITH LOCK.

EXCEPTION CONDITIONS

The exception codes which are possible for each mass storage verb when accessing serial files are these. Their meanings are explained below.

OPEN INPUT and OPEN OUTPUT . . . AT END	2, 5
OPEN OUTPUT	2, 3, 5
READ	None
WRITE and CLOSE	4

2 Integrity code failure

3 Purge date not exceeded.

4 Logical bucket number out of range. (This will happen if you try to write a record and there is not enough space on the file to take it. It can happen when attempting to close a file because, if it is an output file, a short bucket and an end of bucket are written.)

5 File not in system.

EXAMPLE OF SERIAL FILE PROCESSING

The problem

Write a program to create a serial file on disc from card input.

CARD FILE:	Record size	32 characters
	Standard terminator	
DA FILE:	Name	STOCK RECORDS
	Bucket size	4 blocks
	Record size	8 words + word-count word
	Retention period	3 days
	Generation-no	1

No validation is needed and the input records are in the required format for writing to the disc.

In the event of 'file not in system' exception condition on attempting to open the disc file, the open should be reattempted. All other exceptions should be displayed and the run abandoned.

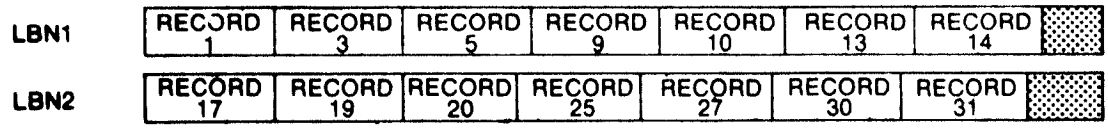
```
IDENTIFICATION DIVISION.
PROGRAM-ID. GOAX50.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. ICL 1904.
OBJECT-COMPUTER. ICL 1904
    MEMORY ... WORDS.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT AAA-CDSIN ASSIGN CARD READER.
    SELECT AAA-SERIALOUT ASSIGN EDS 1.
I-O-CONTROL.
    APPLY WAA-REPLY TO TRANSFER REPLY ON AAA-SERIALOUT.
*
DATA DIVISION.
FILE SECTION.
FD  AAA-CDSIN.
01  AAA-CDREC                     PIC X(80).
*
FD  AAA-SERIALOUT
    RECORDING MODE F
    BLOCK 8 RECS
    LABEL RECORDS STANDARD WITH GENERATION NO
    VALUE OF ID "STOCK RECORDS"
    ACTIVE-TIME 3
    GENERATION NO IS WAA-GEN.
01  AAA-CDSREC                    PIC X(80).
*
WORKING-STORAGE SECTION.
77  WAA-REPLY                     PIC 9(6) COMP SYNC RIGHT.
77  WAA-GEN                       PIC 9(6) COMP SYNC RIGHT
                VALUE 1.

PROCEDURE DIVISION.
PARA-START.
    OPEN OUTPUT AAA-SERIALOUT.
    IF WAA-REPLY = 0
        STOP "ONLINE REPLY AND GO"
        GO TO PARA-START.
    IF WAA-REPLY NOT = ZERO
        STOP "ABANDON RUN".
PARA-OPEN.
    OPEN INPUT AAA-CDSIN.
PARA-READ.
    READ AAA-CDSIN AT END GO TO PARA-END.
    MOVE AAA-CDREC TO AAA-CDSREC.
    WRITE AAA-CDSREC.
    IF WAA-REPLY = 4 DISPLAY "FILE EXCEEDED"
        STOP "ABANDON RUN".
    GO TO PARA-READ.
PARA-END.
    CLOSE AAA-CDSIN AAA-SERIALOUT.
    IF WAA-REPLY = 4 DISPLAY "FILE EXCEEDED"
        STOP "ABANDON RUN".
    STOP RUN.
```

THE CHARACTERISTICS OF A SEQUENTIAL FILE

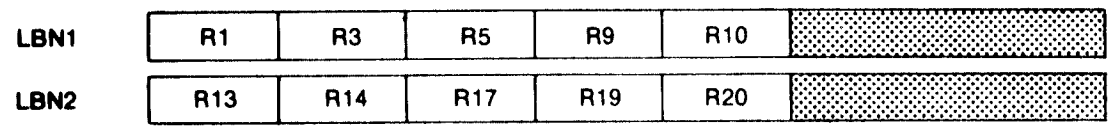
The distinguishing characteristic of a sequential file is that its records are held in *key sequence*. On a serial file there is no relationship between the record key (if any) and the position of the record in the file. But on a sequential file the key number determines the position of the record on the file, relative to the other records. Records with low key numbers are held at the beginning of the file. Records with higher key numbers are held nearer the end.

For example, this diagram shows the first two buckets of a file, with the records arranged in sequential order:

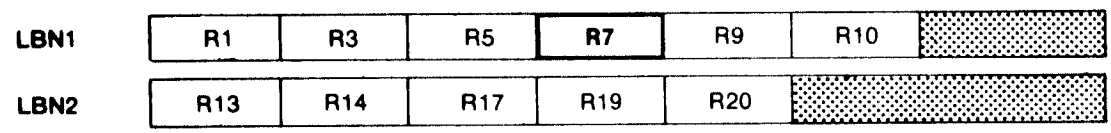


Adding records to the file

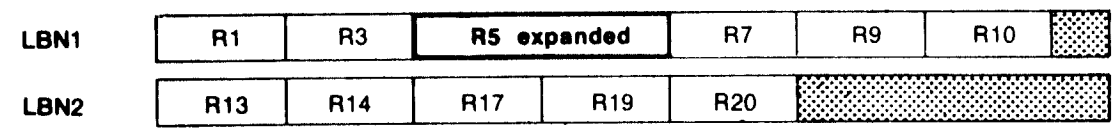
When new records are added to a serial file they are simply tacked on at the end after all the other records. But with a sequential file each new record has to be inserted at a particular point in the body of the file, so that the sequential order is maintained. To make this possible, the buckets on a sequential file are not usually filled with records. A proportion of the space is left free so that new records can be inserted. So the two buckets in the last diagram would more probably be organised like this:



New records can now be inserted without upsetting the organisation of the whole file. For example, if you want to add a new record with the Key Number 7, only records 9 and 10 need to be moved:



In the same way, if one of the records is expanded, only the records between that record and the end of the bucket will be displaced:

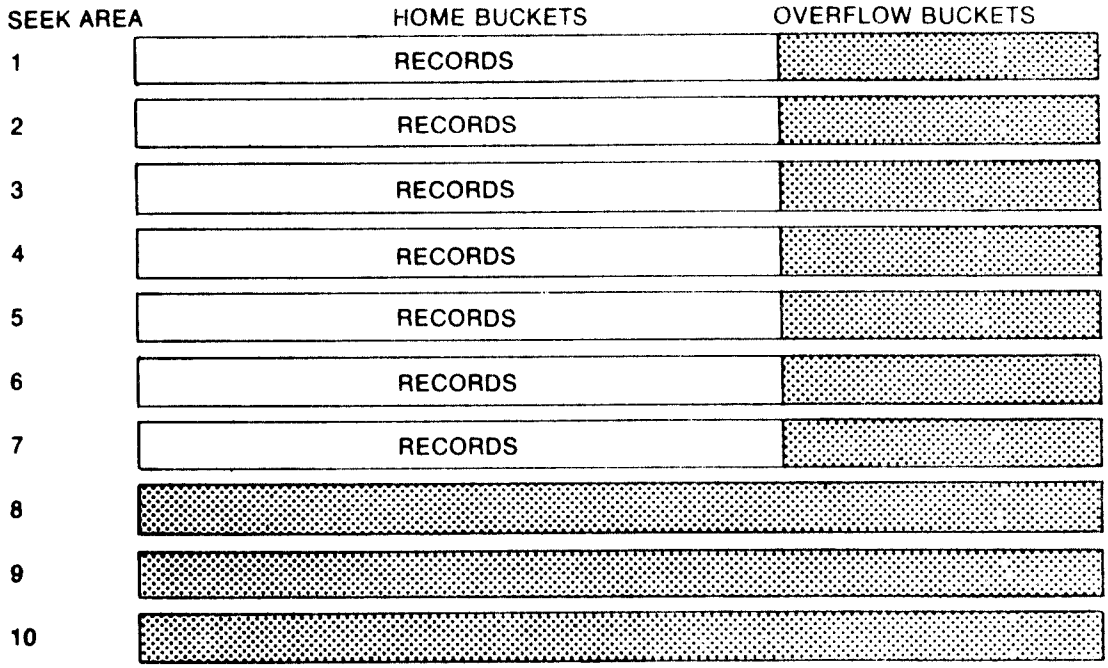


Overflow

As new records are added, and existing records expanded, some of the buckets on the file will eventually fill up. If you try to write a new record to a bucket that is already full, *overflow* will occur.

To cope with this, a number of buckets at the end of each seek area are usually left free when the file is loaded—just as an area within each bucket is left free. These buckets are called *overflow buckets*. (Buckets which hold records when the file is loaded are called *home buckets*.)

For example, a file with 10 seek areas might have this layout:



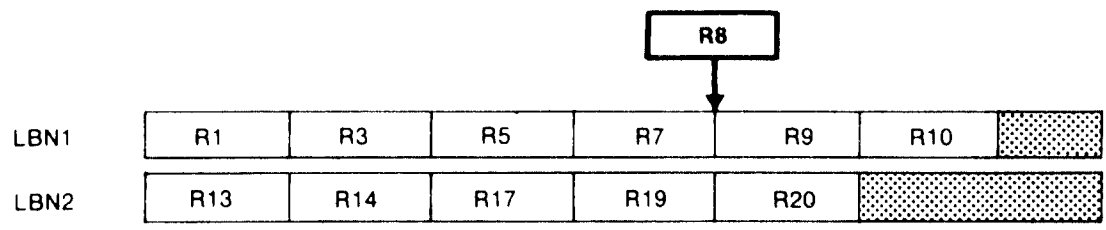
The end-of-file bucket on a sequential file will follow the last *home* bucket containing data (not the last *data* bucket, since this may be an overflow bucket).

Tags

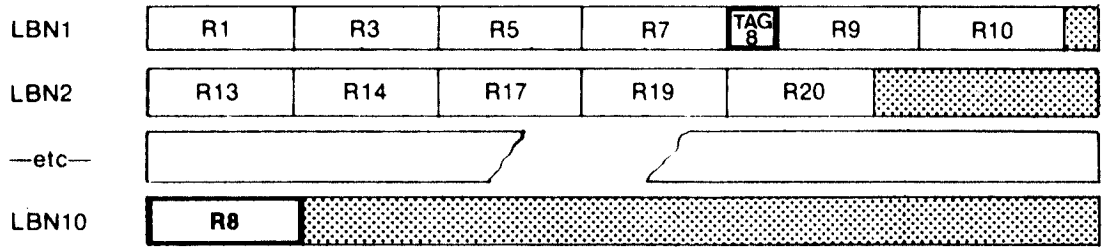
When overflow first occurs, Housekeeping will copy the record that has been overflowed to the beginning of the last overflow bucket on the seek area. In the home bucket, it will put a *tag* in the position that the record would have occupied if there had been room.

A tag consists of (*a*) a 'pointer' word, and (*b*) the key of the record it replaces. The pointer word will contain a number which, when added to the LBN of the home bucket, gives the LBN of the overflow bucket holding the record itself.

For example, continuing the illustration we used above, suppose you wanted to insert Record 8 into LBN 1, and there was not enough room in the bucket to take the whole record:



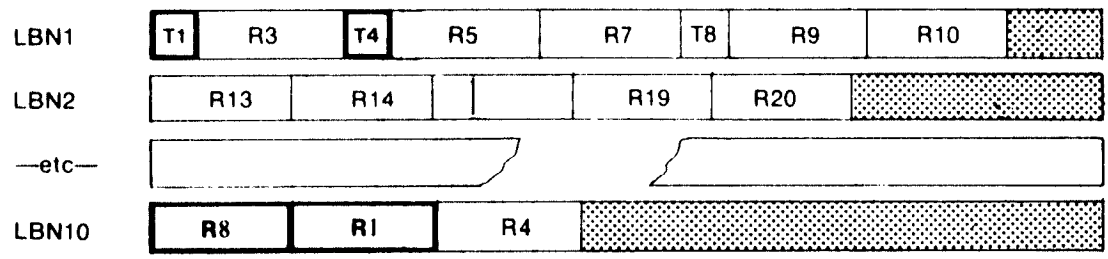
Housekeeping would store Record 8 in the last overflow bucket that has room for it on the seek area and put a tag in the home bucket:



Tag 8 will consist of the record key (8) and a word containing the number 9. When the time comes to retrieve the record, this number can be added to the home LBN (1) to give the overflow LBN (10).

If there is not enough room in the bucket even to take the tag, the first record will be moved to the overflow bucket, and a tag inserted for both this record and the new record:

For example, add Record 4:



If records are later deleted from the home bucket so that space becomes available, Housekeeping will *not* move records back from the overflow bucket. The only way to clear the overflow buckets is to reorganise the file, using the appropriate software.

If records are deleted from an overflow bucket and the overflow bucket pointer has been updated, the free space cannot be used to receive further overflow records.

Second level overflow

If all the overflow buckets on a seek area become full, or if a home bucket is completely filled with tags, it will not be possible to deal with new records or expansions using the normal overflow procedure. Instead, *second level overflow* will have to be used.

Second level overflow means storing records and tags in a specially reserved area at the end of the file. When second level overflow occurs, the home bucket that has overflowed will be assigned an *extension bucket* in this second level area, and any records or tags that will not fit into the home bucket will be stored in the extension bucket. As its name implies, the extension bucket will be treated as an extension of the home bucket itself, not as an overflow bucket. When a record is read, the home bucket and its extensions will be scanned as though they made up one bucket.

Second level overflow involves a lot of extra processing, and (since the home and extension buckets will be on different cylinders) time-consuming head movement. So second level overflow should be regarded as an emergency measure. When it begins to occur, you should reorganise the file in a larger area, using the appropriate software.

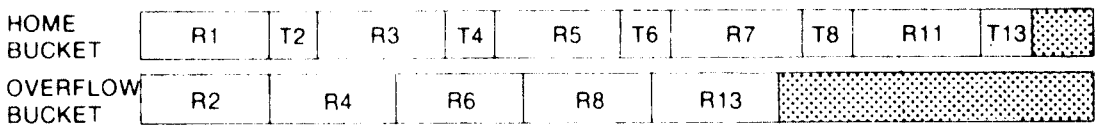
Defining a sequential file

We have the following entry in the FILE-CONTROL paragraph:

SELECT file—name ASSIGN EDS integer—1 [RESERVE 1]
[ACCESS MODE IS SEQUENTIAL]
[ORGANISATION IS FIRST-LEVEL [SECOND-LEVEL]].

RESERVE 1, when referring to a sequential disc file, has nothing to do with double buffering as it does with files on other peripherals. Instead, it requests a special buffer in store to hold an overflow bucket. If you use this option, you will increase the efficiency of a program by reducing the number of physical transfers needed to access records in overflow.

For example, suppose the first few records in a file were distributed between the home and overflow bucket like this:



If no overflow buffer was being used, each time a tagged record was read, the overflow bucket would have to be transferred to the file's home buffer, overwriting the home bucket. The home bucket would then have to be transferred from the file once again, so that the next record could be accessed.

Reading these few records sequentially would involve several physical transfers. However, if an overflow buffer had been reserved, the home and overflow buckets could both be held in store until their records had been read.

The clause defining the type of access is optional—sequential access will be assumed if you omit it.

The ORGANISATION clause is necessary for handling the creation of new overflow on the file (existing overflow will be handled automatically).

FIRST-LEVEL provides for handling new first level overflow only.

SECOND-LEVEL provides for handling new second level overflow.

If you want both levels dealt with FIRST-LEVEL and SECOND-LEVEL must both be written.

The ACCESS and ORGANISATION clauses are part of the SELECT. . . .sentence so the full stop comes only after the last one written.

The I-O-CONTROL paragraph is the same as for serial processing, i.e.

APPLY data—name TO TRANSFER—REPLY ON file—name.

In the FD paragraph of the DATA DIVISION we can have WITH GENERATION—NO.

Processing a sequential file

You may want to open a sequential file for input only, or for overlay processing.

The form of the OPEN verb is:

OPEN INPUT file—name—1 I—O file—name—2. . . .

You should test the reply field after OPEN and every other processing verb which can cause an exception.

To read the next record in sequence the normal READ verb is used. Tags will be followed if the next record in Key sequence is not in its home bucket.

The format is:

READ file—name AT END imperative—statement.

There are two new COBOL verbs for handling sequential files opened for overlay.

REWRITE record—name.

This writes back onto the file the last record you have read. If you have not updated the record, it is not necessary to rewrite it. It is already on the file and you can go on to the next read.

The DELETE verb can be used to delete the record currently in the record area in store. The form is:

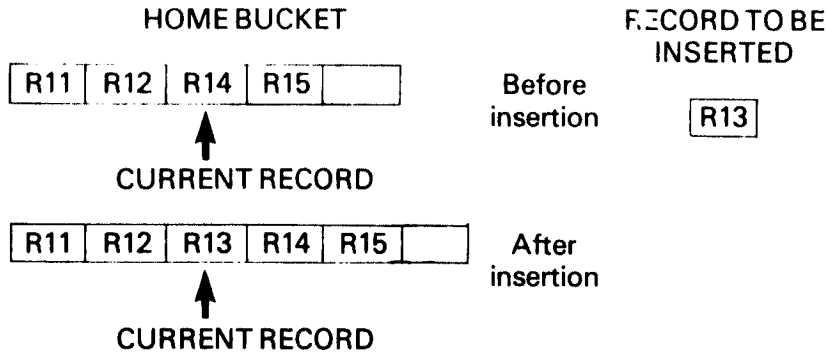
DELETE record—name.

The WRITE verb is used to insert new records in the file.

The format is:

WRITE record—name.

A new record will be inserted in front of the current record (i.e. the record just read). Housekeeping will then move a pointer from the record read back to the record inserted so that it becomes the current record.



So, to use this example, the next READ statement would cause record 14 to become the current record again.

To close a sequential file, use the normal COBOL CLOSE verb, i.e.

CLOSE file—name—1, file—name—2. . . .

If you want the file to remain available to the program, the WITH RETAIN option can be used. You may also use WITH LOCK if you do not want to access the file again.

For processing of sequential files using earlier marks of housekeeping, see Appendix A.

Exception conditions

The exception codes which are possible for each mass storage verb when working with sequential files are these. Their meanings are explained below.

OPEN INPUT:	2, 5
OPEN I—O:	2, 3, 5
READ:	None
REWRITE and WRITE:	4
CLOSE:	None

2 Integrity code failure (explained on page 210).

3 Purge date not exceeded.

4 Overflow error. (Existing overflow is handled automatically. But if new overflow arises which has not been catered for by the ORGANISATION clause, this error will occur.)

5 File not in system.

An example of sequential file processing

The problem

Write a program to amend records in a sequential file called MASTERFILE from card input. The amendment cards are sorted in Key sequence and the end of the card file is the end of the run.

DA FILE:

Name	MASTERFILE
Bucket size	4 blocks
Record size	11 words

Record format

Word 1:	word-count
2:	4-character numeric Key
3—9:	Detail-1
10:	Detail-2
11:	Total (binary)

CARD FILE: One record per card, standard terminator.

Record format

Col 1:	Amendment type
2—5:	Key
6—7:	Quantity (validated numeric)
8—11:	Detail (alphanumeric)

amendment type:

1 = addition of quantity to total
2 = delete this record
3 = insert this record (move card detail field to second detail field on disc record)

Provide for first-level overflow. If further overflows occurs, display card key and reply field and abandon the run. If the open causes a "file not in system" error condition, attempt the open again. For other exceptions display the reply field and stop run.

No generation number check should be performed on the DA file.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. BBRAGG.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. ICL 1904.
OBJECT-COMPUTER. ICL 1904A
    MEMORY 3000 WORDS
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT AAA-CDIN ASSIGN CARD READER.
    SELECT BAA-SEQLAY ASSIGN EDS 1 RESERVE 1
        ORGANISATION FIRST LEVEL.
I-O-CONTROL.
    APPLY MAA-REPLY TO TRANSFER-REPLY ON BAA-SEQLAY.
DATA DIVISION.
FILE SECTION.
FD  AAA-CDIN
01  AAA-CDREC.
    03 AAA-TYPE          PIC 9.
    03 AAA-KEY           PIC 9(4).
    03 AAA-QTY           PIC 99.
    03 AAA-DETAIL        PIC X(4).

FD  BAA-SEQLAY
    RECORDING MODE F
    BLOCK 2048
    LABEL RECORDS STANDARD
    VALUE OF ID "MASTERFILE".
01  BAA-DISCREC.
    03 BAA-KEY           PIC 9(4).
    03 FILLER            PIC X(28).
    03 BAA-DETAIL        PIC X(4).
    03 BAA-TOT           PIC 9(6) COMP SYNC RIGHT.
*
WORKING-STORAGE SECTION.
01  MAA-REPLY            PIC 9(6) COMP SYNC RIGHT.


PROCEDURE DIVISION.
PARA-1.
    OPEN I-O BAA-SEQLAY.
    IF MAA-REPLY = 0
        STOP "ONLINE STUFF AND GO"
        GO TO PARA-1.
    IF MAA-REPLY NOT EQUAL DISPLAY MAA-REPLY
        STOP "OPEN ERROR-ABANDON".
PARA-2.
    OPEN INPUT AAA-CDIN.
PARA-3.
    READ AAA-CDIN AT END GO TO PARA-END.
PARA-4.
    READ BAA-SEQLAY AT END DISPLAY "HIGH VALUES MISSING" AAA-KEY
        GO TO PARA-END.
PARA-5.
    IF AAA-KEY > BAA-KEY GO TO PARA-4.
    IF AAA-KEY < BAA-KEY GO TO PARA-INS.
    IF AAA-TYPE = 1 ADD AAA-QTY TO BAA-TOT
                    REWRITE BAA-DISCREC
                    GO TO PARA-5.

    IF AAA-TYPE = 2 DELETE BAA-DISCREC
        ELSE DISPLAY AAA-KEY "TYPE ERROR".
        GO TO PARA-3.
PARA-INS.
    MOVE AAA-DETAIL TO BAA-DETAIL.
    MOVE AAA-QTY TO BAA-TOT.
    MOVE AAA-KEY TO BAA-KEY.
    WRITE BAA-DISCREC.
    IF MAA-REPLY = 1 DISPLAY "OVERFLOW" BAA-KEY
        STOP "ABANDON RUN".
    GO TO PARA-3.
PARA-END.
    CLOSE AAA-CDIN BAA-SEQLAY.
    STOP RUN.
```

SELECTIVE SEQUENTIAL PROCESSING

When processing the two types of file we have already dealt with, every record must be accessed. On a serial file each record in physical sequence is read, while on a sequential file every record in logical sequence is read, if necessary by following tags. This type of processing is called *automatic processing* since Housekeeping automatically presents the next record when the COBOL READ verb is used.

But the reason for having index tables on a sequential file is to make it possible for *selected* records to be read by a *selective sequential* process—the user will specify which record is to be read next, rather than having the next record in logical sequence automatically presented. Housekeeping will search the indexes in the file to find the position of that record. This type of processing is called *non-automatic processing*. It is what we really mean by *direct access.*

THE CHARACTERISTICS OF AN INDEXED SEQUENTIAL FILE

When a serial or sequential file is processed, every record on the file has to be read. It is not normally possible to pick out the records you want, and skip over the rest. So serial and ordinary sequential files are used when the hit rate is high—i.e. when a large proportion of the records on the file need to be processed on each run.

However, in certain cases you may want to be able to select only a few records without having to read through the whole file. This is done by holding the records on an *indexed sequential file*, which is a sequential file with *index tables* added to it.

INDEX TABLES

An index table is a specially reserved area on a file that holds information about where particular records are stored. It consists of a bucket (or buckets) containing a number of *cells*.

INDEX TABLE

Each cell is made up of:

1 LBN of either a home bucket or another index table

2 the key number of the record with the highest key in that home bucket or area addressed by the index table

CELL

Word 0	1	etc.
LBNx	XXXX	XXX — — — — XXX

Highest key of all the records in LBNx.

Bucket indexes

The basic type of index table is the *bucket index*, which gives the highest key in each *bucket* in a particular area. For example, suppose LBN2 holds records up to R10, LBN3 holds records up to R20, LBN4 holds records up to R30, and so on:

LBN1	
LBN2	Records 1-10
LBN3	Records 11-20
LBN4	Records 21-30
LBN5	Records 31-40

If LBN1 was a bucket index, its first few cells would contain this information:

LBN1	LBN 2	Max Key =10	LBN 3	Max Key =20	LBN 4	Max Key =30	LBN 5	Max Key =40	etc.

This index table would be held in store while the file was being processed. When a particular record was read, Housekeeping would be able to work out from the table which bucket the record was in.

Seek Area and File Area indexes

A bucket index has one cell for each home bucket in the area it indexes. So if you only have one index table for the whole file, the table may be very large—particularly if the file itself is a long one. A correspondingly large area in store will have to be reserved to hold the index table.

You can cut down the size of index tables by using a system of *multi-level indexing*. This involves putting a separate bucket index at the beginning of each cylinder or file area, and indexing these areas in turn with a *seek area index* or *file area index*.

For example, a file might have a bucket index before the first data bucket on each cylinder, and a seek area index at the beginning of the whole file:

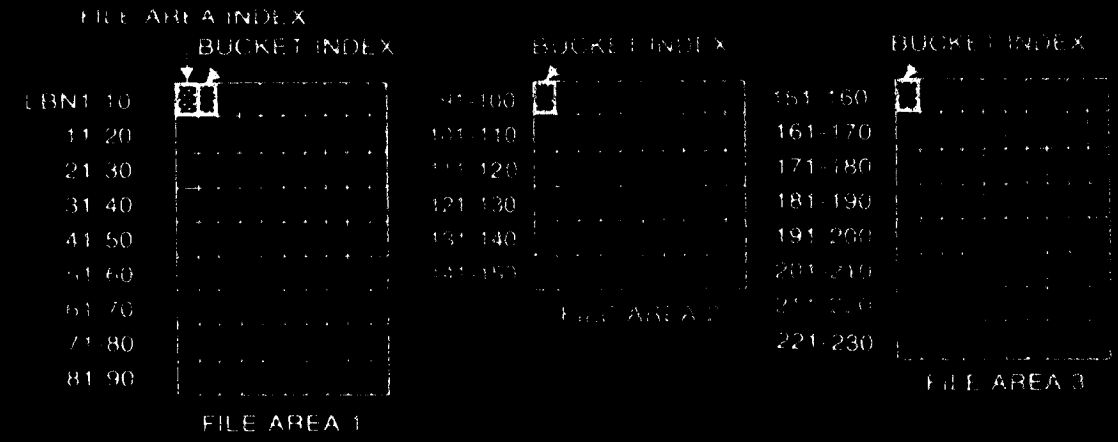
	SEEK AREA 1		SEEK AREA 2		SEEK AREA 3
LBN1	**SEEK AREA INDEX**	11	**BUCKET INDEX**	21	**BUCKET INDEX**
LBN2	**BUCKET INDEX**	12	Records up to R104	22	Records up to R203
LBN3	Records up to R10	13	Records up to R113	23	Records up to R212
LBN4	Records up to R23	14	Records up to R122	24	Records up to R224
LBN5	Records up to R35	15	Records up to R134	25	Records up to R233
LBN6	Records up to R48	16	Records up to R145	26	Records up to R245
LBN7	Records up to R59	17	Records up to R157	27	Records up to R256
LBN8	Records up to R71	18	Records up to R166	28	Records up to R267
LBN9	Records up to R83	19	Records up to R179	29	Records up to R278
LBN10	Records up to R92	20	Records up to R188	30	Records up to R289

The seek area index would have a cell for each cylinder, giving the maximum record key in each of the cylinders:

SEEK AREA INDEX

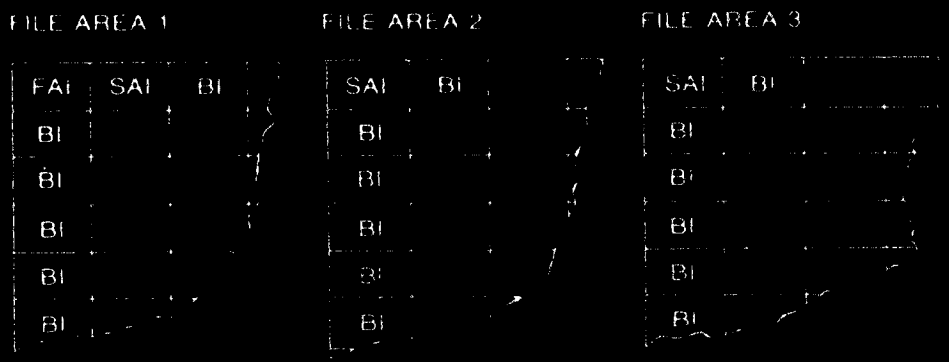
LBN1	SA 1	Max Rec =92	SA 2	Max Rec =188	SA 3	Max Rec =289	etc.

Alternatively, you could have a bucket index at the beginning of each file area, and file area index at the beginning of the whole file:



The file area index (FAI) in LBN1 would give the maximum record key in each file area. If FA1 contained records up to 625, FA2 up to 753, FA3 up to 987, LBN1 would contain

It is also possible to combine all three types of index – a bucket index at the beginning of each seek area, a seek area index at the beginning of each file area, and a file area index at the beginning of the whole file.



Software terminology

Index tables are set up automatically by the software programs that allocate and reorganise files, but you have to specify which of the possible indexing structures you want used. The software will recognise three *levels* of index:

Level 1 — Index table at start of File.

Level 2 — Index table at start of each File Area.

Level 3 — Index table at start of each Seek Area.

The permissible combinations of these levels are:

Level 1 only — Bucket Index at start of file.

Levels 1 and 2 — File area index at start of file, and bucket indexes at start of file areas.

Levels 1 and 3 — Seek area index at start of file, and bucket indexes at start of seek areas.

Levels 1, 2 and 3 — File area index at start of file, seek area indexes at start of file areas, and bucket indexes at start of seek areas.

The choice of indexing structures will depend on a number of considerations. The simplest arrangement is to have a Level 1 index only, but this may mean having to use an excessively large area of core store as an index buffer. Multi-level indexes require less core, since the individual index tables will be smaller, but they will occupy more space on the disc, and will increase search times, since index tables will have to be transferred back and forth between file and core store. (With single level indexing, the index table can remain in core store all the time that processing is going on.)

USING MASS STORAGE VERBS

Defining an indexed sequential file

Here is an example of how you can code the FILE—CONTROL and I—O—CONTROL paragraphs.

```
FILE—CONTROL.

  SELECT file—name—1 ASSIGN TO EDS integer—1 [RESERVE 1]

  ACCESS MODE IS RANDOM
  ORGANISATION IS INDEXED [FIRST—LEVEL] [SECOND—
  LEVEL]
  SYMBOLIC KEY IS data—name—1.

I—O—CONTOL.
  APPLY data—name—1 TO TRANSFER—REPLY ON file—name—1
  APPLY data—name—2[data—name—3] [data—name—4] TO
  INDEX—BUFFER ON file—name—2.
```

RESERVE 1 may be used to set aside a buffer for an overflow bucket.

ACCESS MODE IS RANDOM means that we are reading records selectively rather than each one in sequence.

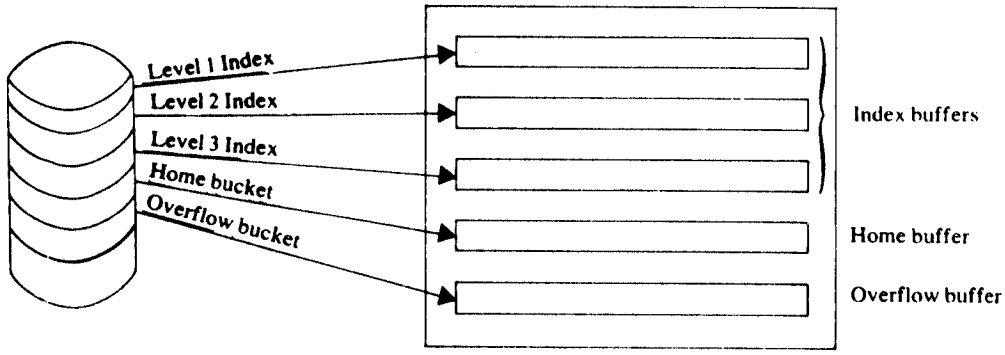
ORGANISATION IS INDEXED specifies that selection of records is to be by index tables. FIRST—LEVEL and SECOND—LEVEL have the same meaning as for sequential files.

The SYMBOLIC KEY clause defines the field which holds the key of the record you wish to select. These clauses are all part of the SELECT. . . sentence, so the full stop should only follow the last one written. The first sentence in the I—O—CONTROL paragraph relates to transferring error codes.

When handling a selective sequential file, up to three buffers can be set up in store for holding indexes. If you do not allow this space, every time a record is accessed, each level of index must be brought into the file's home buffer in turn until the record's home bucket is found.

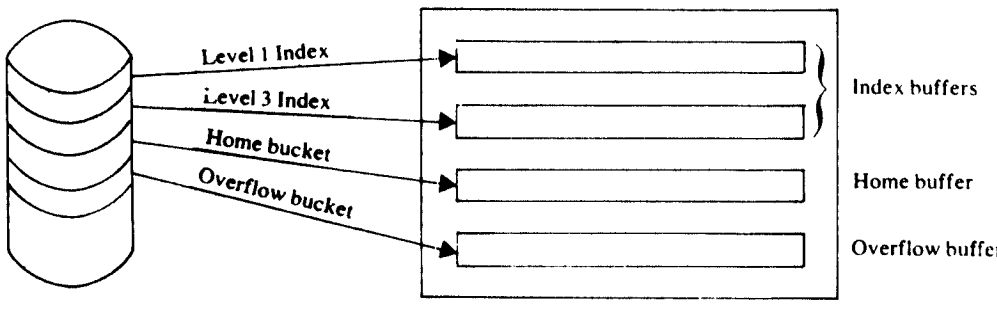
The following diagrams show how buffers will be used for different combinations of index levels.

1. A file has indexes at Levels 1, 2 and 3 and you set aside three buffers.



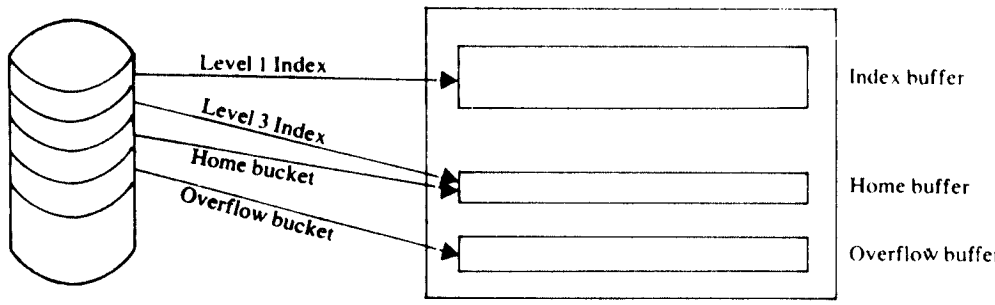
In this case the home buffer is being used by home buckets only.

2. A file has indexes at Levels 1 and 3 and you set aside two buffers in store:



Again the home buffer is always being used by home buckets.

3. A file has Levels 1 and 3 indexes and only one buffer is set aside in store:



The Level 3 index and home buckets share the file's home buffer.

If index buffers are set aside, access time to selected records is reduced and the program's efficiency increased.

If you use special index buffers, areas for these must be set up in WORKING—STORAGE SECTION. They are defined by this sentence:

APPLY data—name—1 [data—name—2][data—name—3] TO INDEX—BUFFER on file—name

Index buffers should be defined in the Working-Storage Section like this:

```
01  FIRST BUFFER.
    02  FILLER                    PIC X(4) OCCURS n.
```

where n is the number of words required for the index buffer

(02 levels must define the size of buffers because OCCURS cannot be used at the 01 level.)

Here is an example of all the coding to define an Indexed Sequential file using mass storage verbs.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
    MEMORY 10000 WORDS.
SPECIAL-NAMES.
    CARD-READER 1 IS PERIPH.
INPUT OUTPUT SECTION.
FILE CONTROL.
    SELECT INDEX-FILE ASSIGN EDS 1 RESERVE 1
        ACCESS MODE RANDOM
        ORGANISATION INDEXED FIRST-LEVEL
                     SECOND-LEVEL
        SYMBOLIC KEY AMENDREC-KEY.
```

```
I-O-CONTROL.
    APPLY REPLY-FIELD TO TRANSFER-REPLY
                      ON INDEX-FILE.
    APPLY BUFFER-1 BUFFER-2 TO INDEX-BUFFER
                            ON INDEX-FILE.
DATA DIVISION.
FILE SECTION.
FD  INDEX-FILE
    BLOCK 1024
    LABEL RECORDS STANDARD WITH GENERATION-NO
    VALUE OF ID 'MASTER'.
```

Processing an indexed sequential file

You can open this type of file for INPUT or OVERLAY processing in exactly the same way as a sequential file.

i.e. OPEN INPUT file-name I-O file-name-2.

For indexed sequential files there is a special new verb-SEEK. This looks up index tables, and sets a pointer to the entry in the index which will locate the record's home bucket.

The format is:

SEEK file-name RECORD

The SEEK verb takes the contents of the SYMBOLIC KEY field you have defined, and uses it to search index tables.

Before attempting to read a file you are accessing with index tables, a SEEK must be executed.

The format of the read verb is:

READ file-name INVALID KEY imperative-statement

There is no AT END clause because we are reading records selectively.

It is necessary to specify the INVALID KEY clause. If the SEEK has produced an LBN outside the file's bucket range the imperative statement, usually a GO TO, will be followed. Control will also tranfer to the imperative statement if the SEEK cannot find an index entry because the indexes are wrongly set up, but this would be very unusual.

The REWRITE verb replaces the last record read from the file after updating it.

Its form is:

REWRITE record-name

To eliminate a record from the file, use a SEEK to point to the relevant bucket and then the DELETE verb.

DELETE record-name INVALID KEY imperative-statement

If you want to insert a new record in the file, again use a SEEK to find the correct position and then the WRITE verb.

WRITE record-name INVALID KEY imperative-statement

When writing and deleting records, the INVALID KEY clause will serve the same purpose as with READ.

An indexed sequential file is closed by the conventional CLOSE verb. If you want the file to remain available to the program, you may use the WITH RETAIN option.

Exception conditions

The exception codes which are possible for each mass storage verb when accessing indexed sequential files are these. Their meanings are explained below.

OPEN INPUT:	2, 5
OPEN I—O:	2, 3, 5
SEEK:	None
READ:	1
DELETE:	1
REWRITE:	4
WRITE:	4, 6

1 Record missing.

2 Integrity code failure.

3 Purge date not exceeded.

4 Overflow error.

5 File not in system.

6 Duplicate key (when inserting a new record with a specific key number, a record with the same key is already on the file).

An example of selective sequential processing

The problem

Write a program to update an indexed sequential file called INSEQFILE, from amendments on cards. There is a maximum of one card per disc record, and the hit rate is low. Three index buffers should be set up, and first level overflow handled.

DA FILE:	Name	INSEQFILE
	Bucket-size	2 blocks
	File areas	3
	Index tables	Levels 1, 2 and 3
	Key	4 characters
	Record length	6 words
	Cylinder packing density	75%

There are 3 cells in the Level 1 index.

There is a maximum of 15 entries in a Level 2 index.

There is a maximum of 29 entries in a Level 3 index.

Record format

Word 1: word count

2: key

3: total quantity (binary)

4—6: other detail

CARD FILE: One record per card, standard terminator

Record format

Col. 1—4: key
5—8: new quantity—validated numeric

9: amendment type (1—3)

10—21: other detail

Amendment type

1 = Simple addition of new to total quantity.

2 = Delete record with this key.

3 = Insert record with this key.

Columns 4—15 contain data which will complete the record. This is to be stored in character form straight from the card.

The end of the card file is the end of the run. If exception code for "file not in system" occurs, attempts to reopen the file. If "duplicate key" or "record missing" errors come up display the key and continue the run. However, for any other exception, abandon the run and print a message on the console.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ISEX50.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. ICL-1904.
OBJECT-COMPUTER. ICL-1904.
    MEMORY 5000 WORDS.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT AAA-CAIN ASSIGN CARD-READER 1.
    SELECT BAA-ISOLAY ASSIGN DISC 1 RESERVE 1
        ACCESS RANDOM
        ORGANISATION INDEXED FIRST-LEVEL
        SYMBOLIC KEY AAA-KEY.
I-O-CONTROL.
    APPLY WAA-REPLY TO TRANSFER-REPLY ON BAA-ISOLAY.
    APPLY WAB-FIRST WAB-SECOND WAB-THIRD
        TO INDEX-BUFFER ON BAA-ISOLAY.
DATA DIVISION.
FILE SECTION.
FD  AAA-CAIN.
01  AAA-CAREC.

    03 AAA-KEY          PIC X(4).
    03 AAA-QTY          PIC 9(4).
    03 AAA-TYPE         PIC 9.
    03 AAA-DETAIL       PIC X(12).
FD  BAA-ISOLAY
    RECORDING MODE F
    BLOCK 1024
    LABEL RECORDS STANDARD
    VALUE OF ID "INSEQFILE".
01  BAA-REC.
    03 BAA-KEY          PIC X(4).
    03 BAA-QTY          PIC 9(6) COMP SYNC RIGHT.
    03 BAA-DETAIL       PIC X(12).
WORKING-STORAGE SECTION.
01  WAA-REPLY           PIC 9(6) COMP SYNC RIGHT.
01  WAB-INDIC           PIC 9(6) COMP SYNC RIGHT.
01  WAB-FIRST.
    03 FILLER           PIC X(4) OCCURS 18.
01  WAB-SECOND.
    03 FILLER           PIC X(4) OCCURS 42.
01  WAB-THIRD.
    03 FILLER           PIC X(4) OCCURS 70.
```

```
PROCEDURE DIVISION.
OPEN-DISC.
    OPEN I-O BAA-ISOLAY.
    IF WAA-REPLY = "STOP ONLINE *7774 AND GO"
        GO TO OPEN-DISC.
    IF WAA-REPLY NOT ZERO DISPLAY WAA-REPLY
        STOP "OPEN ERROR".
PARA-2.
    OPEN INPUT AAA-CAIN.
    MOVE ZERO TO WAB-INDIC.
CHECK-PARA.
    IF WAA-REPLY = 1 OR 6
        DISPLAY AAA-KEY SPACE WAA-REPLY SPACE WAB-INDIC
        GO TO READ-PARA.
    IF WAA-REPLY = 4
        DISPLAY AAA-KEY "OVERFLOW" INDIC
        GO TO END-PARA.
READ-PARA.
    READ AAA-CAIN AT END GO TO END-PARA.
    SEEK BAA-ISOLAY.
    IF AAA-TYPE = 3
        GO TO INSERT-PARA.
    IF AAA-TYPE = 2
        GO TO DELETE-PARA.
    IF AAA-TYPE = 1
        GO TO AMEND-PARA.
    DISPLAY "TYPE ERROR " AAA-TYPE.
        GO TO READ-PARA.
DELETE-PARA.
    MOVE "D" TO WAB-INDIC.
    DELETE BAA-REC INVALID KEY
        GO TO INV-PARA.
    GO TO CHECK-PARA.
INSERT-PARA.
    MOVE "I" TO WAB-INDIC.
    MOVE AAA-KEY TO BAA-KEY.
    MOVE AAA-DETAIL TO BAA-DETAIL.
    MOVE AAA-QTY TO BAA-QTY.
    WRITE BAA-REC INVALID KEY
        GO TO INV-PARA.
    GO TO CHECK-PARA.
AMEND-PARA.
    MOVE "A" TO WAB-INDIC.
    READ BAA-ISOLAY INVALID KEY
        GO TO INV-PARA.
    PERFORM CHECK-PARA.
    ADD AAA-QTY TO BAA-QTY.
    REWRITE BAA-REC.
    GO TO READ-PARA.
INV-PARA.
    DISPLAY "INVALID KEY " AAA-KEY.
    GO TO READ-PARA.
END-PARA.
    CLOSE BAA-ISOLAY AAA-CAIN.
    STOP RUN.
```

THE CHARACTERISTICS OF A RANDOM FILE

With random files the position of each record on the file is worked out by applying a formula (or algorithm) to the key to give the LBN.

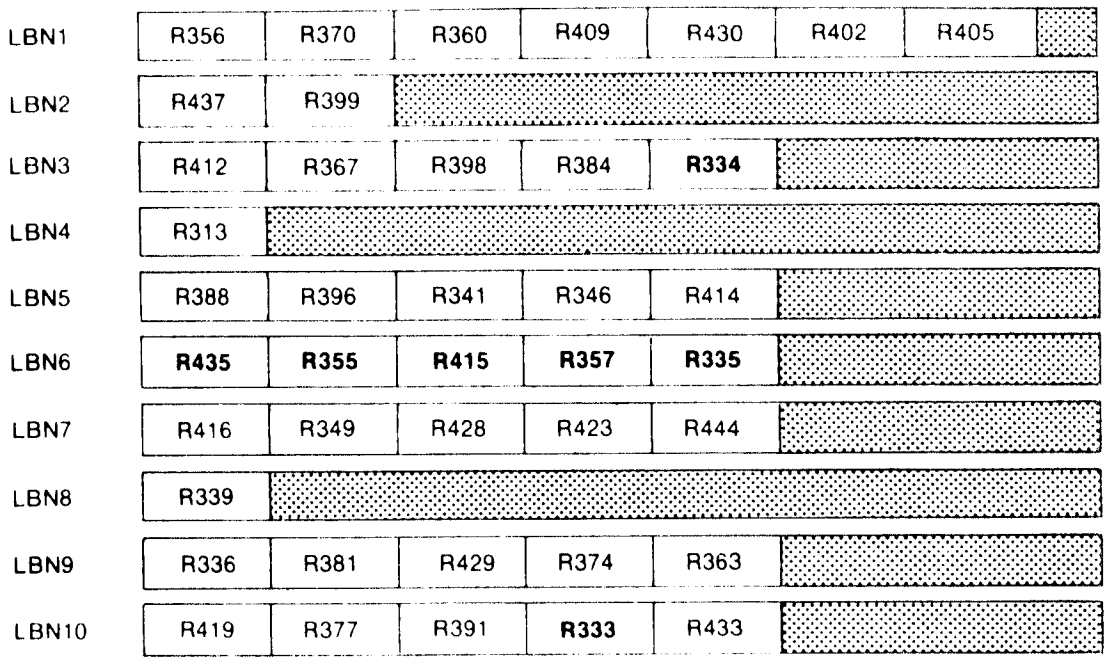
But the random files differ from all other types of sequential file in that the records on the file are not arranged in key order. The address-generating algorithm used on a random file will distribute records in an apparently 'random' order (that is, non-sequential order).

To take a simple example; suppose records with keys in the range 333—444 are to be stored in a 10-bucket random file; the LBN of each record is to be worked out like this:

1 Multiply together the last two digits of the key.

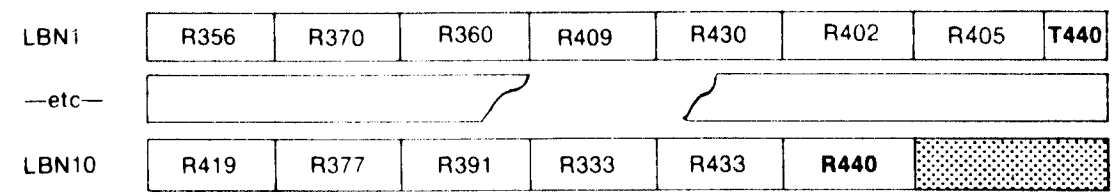
2 Take the last digit only of the product, and add 1.

So Record 333 will be stored in LBN10; Record 334 in LBN3; Record 335 in LBN6, which might also hold Records 355, 357, 415 and 435;



There are normally no overflow buckets on a random file. All buckets are treated as home buckets. If a bucket overflows, the overflowing record will be stored in the last bucket that has room for it on the same cylinder.

For instance, suppose Record 440 was to be added to the file in the examples above. According to the algorithm, Record 440 belongs in LBN1. But LBN1 is full. So Record 440 will be stored in LBN10 (assuming that LBN10 is on the same cylinder as LBN1):



If the last bucket on the cylinder is full, the overflow record will be stored in the next to last (in this case LBN9). If that is full it will be stored on the one before that, and so on. This means that it is possible for a record on a random file to overflow into an LBN *below* the one it belongs in. (This will happen when all higher buckets are full.) When that happens, the pointer in the tag for the record will be a negative number.

Random files do not have end-of-file buckets since they are intended for non-automatic processing.

Defining a random file

Here is an example of how you can code the FILE—CONTROL and I—O—CONTROL paragraphs.

FILE—CONTROL.

SELECT file—name—1 ASSIGN TO EDS integer—1 [RESERVE 1]

ACCESS MODE IS RANDOM

ORGANISATION IS DIRECT [FIRST—LEVEL] [SECOND—LEVEL]

ACTUAL KEY IS data—name—1

SYMBOLIC KEY IS data—name—2.

I—O—CONTROL.

APPLY data—name—3 TO TRANSFER—REPLY ON file—name—1.

We now have ORGANISATION IS DIRECT which indicates that the method of selection is to be by address generation. ACTUAL KEY refers to a field in which you must store the result of the address generating technique (i.e. the required LBN) after you have calculated it. Data—name—1 must be that of an integral numeric field, of no more than six characters if DISPLAY or COMP.

The meaning of the other clauses is the same as with indexed sequential files.

i.e. ACCESS MODE IS RANDOM means we are processing records selectively, instead of accessing every one.

SYMBOLIC KEY defines the field which holds the key of the record you wish to select.

The I—O—CONTROL paragraph has its usual entry about the reply location to be tested after any processing verb.

Processing a random file

This type of file can be opened for input or overlay processing. The format of the OPEN statement does not change from other file types.

i.e. OPEN INPUT file-name-1 I-O file-name-2

In order to select a particular record you wish to access, the first thing to be done is to take the contents of the SYMBOLIC KEY, perform the address generating technique for the file on it, and store the result in the ACTUAL KEY field. This now points to the actual LBN in which the record is to be found.

After this a READ will obtain the required record, DELETE will eliminate the specified record and WRITE will insert a new record at the end of the correct bucket.

The format of these verbs is:

READ file—name INVALID KEY imperative—statement.

DELETE record—name INVALID KEY imperative—statement.

WRITE record—name INVALID KEY imperative—statement.

As with the Index Table method of selection, no preceding READ is needed before DELETE or WRITE.

The use of INVALID KEY is obligatory. It is possible that the AGT could produce an LBN out of range in which case the "imperative—statement" will be followed.

If a record you have just read is to be replaced then REWRITE is used.

REWRITE record—name

The procedures described above, apart from READ, can be used on files opened for overlay processing only.

A random file is closed by the conventional CLOSE verb. If it is needed again in the program, WITH RETAIN may be written.

Exception conditions

The exception codes which are possible for each mass storage verb when accessing random files are these. Their meanings are explained below:

OPEN INPUT:	2, 5
OPEN INPUT-OUTPUT:	2, 3, 5
READ:	1
DELETE:	1
REWRITE:	4
WRITE:	4, 6

1 Record missing

2 Integrity code failure

3 Purge date not exceeded

4 Overflow error

5 File not in system

6 Duplicate key (when inserting a new record, a record with the same key is already on the file)

An example of random file processing

The problem

Write a program to update a random file, RANDOMFILE, from amendments on cards. The amendment cards are in random sequence and the hit rate is very low.

DA FILE:	Name	RANDOMFILE
	Bucket size	1 block
	2nd Level overflow	80 blocks
	Key	4 characters
	Record length	12 words

The LBN for each record can be calculated by subtracting 1 from the key, dividing the result by 8 and taking the quotient + 1 as the LBN.

Record Format

Word 0:	word count
1:	key (range 0001-n)
2:	code (4 characters)
3—11:	other detail

CARD FILE: One record per card, standard terminator

Record Format

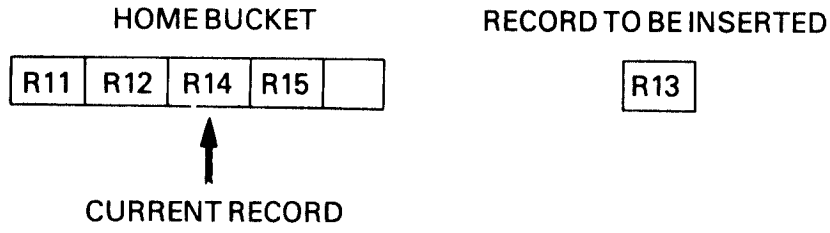
Col. 1—4:	key (validated numeric)
5—8:	new code

The amendment is a replacement of the code and the end of the card file is the end of the run.

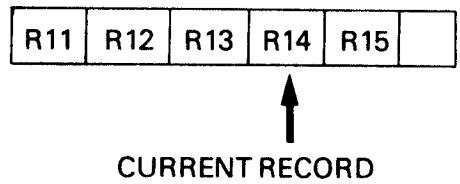
On any exception in the DA file abandon the run, displaying the exception on the console.

MAJOR DIFFERENCES BETWEEN MK. I AND MK. III HOUSEKEEPING

As far as the programmer is concerned, the major differences are in the way sequential files are handled. The following example shows what happens when a record is to be inserted: —



Record 14 on the sequential file has just been read and is therefore the current record. If record 13 is then written the position will be: —



The current record pointer is still at record 14, so that record must be processed before the next READ statement is issued. (With Mk. III Housekeeping the current record pointer would be at the record just inserted, in this case record 13).

So the programmer must ensure that the following action takes place when processing sequential files with Mk. I Housekeeping:

If a record is to be inserted, the current record on the sequential file i.e. the record just read, must be copied into an area which has been set aside in the Working-Storage Section.

The record to be inserted is then moved into the sequential file record area and written away. Before processing can continue the current record is moved back into the sequential file record area from Working-Storage to allow further insertions at this point and amendments to the current record itself.

Chapter 12 Sorting

SORTING

Introduction

For certain processes such as updating serial files it is necessary that records are held in sequence of a key field, or perhaps several key fields.

This can be done by sorting input documents manually but as this is sometimes impossible or undesirable, sorting is normally carried out by software as part of a computer process.

For example, if a master file containing customer orders is updated once a week, order received during the week are input to a file as they are received, validated and then sorted immediately prior to the weekly update run.

It is possible to sort data held on magnetic tape or disc files in either of 2 ways:

1 using a free-standing sort utility such as XSDA, XSDC, XSEG, etc.

2 using the COBOL SORT verb as part of a program carrying out other processes.

Each method has particular advantages; a free-standing sort is generally more efficient and quicker to use when a straightforward sort is required, however a sort as part of a COBOL program may be preferable for a number of reasons. For example the user is able to process the file immediately before and after the sort; also it is possible to transfer data from tape to disc for sorting, this is an important advantage as disc-based sorts are much faster than tape-based sorts.

Setting up a COBOL SORT

In any program containing a COBOL SORT, certain information about the sorting process must be supplied in the following ways:

```
MEMORY 5888 WORDS.
```

MEMORY clause

This entry is important because 5888 words is the minimum store requirement for the SORT software. The effect of the SORT verb is to dump the COBOL program to a workfile, load the appropriate sorting program (XSXA for disc sorting, XSXM for tape sorting) and finally to load the COBOL program back into store on completion of the sort. The SORT programs can work more efficiently if allowed more store.

```
FILE-CONTROL.
  SELECT file-name ASSIGN TO {EDS / TAPES} integer-1
```

In addition to SELECT ... ASSIGN statements for files to be used in the normal way, this statement must be used to specify a dummy file, called the sort-file. It is assigned to EDS for a disc-based sort, or TAPES for a tape-based sort. For example:

```
SELECT SORT-FILE ASSIGN EDS 1.
```

I-O-CONTROL paragraph

Further information about this dummy sort-file is supplied in the file section of the data division.

The next entry:

```
I-O-CONTROL .
APPLY data-name TO SORT-* ON file-name.
```

is optional and, if omitted, certain default values about the sorting environment will be assumed by the sort program (see specifications for XSXA and XSXM). If present the entry might take this form:

```
APPLY SORT-DATA TO SORT-* ON SORT-FILE.
```

Where SORT-DATA is an area in the working-storage section which contains certain values such as the names and generation numbers of files to be used as work files (details of these working-storage entries are given further on in this chapter). SORT-FILE is the name given to the dummy sort file previously assigned in FILE-CONTROL.

SD entries

```
DATA DIVISION .
FILE SECTION .
SD file-name
  [RECORDING MODE { F } ]
                  { V }

  BLOCK{ integer-1 CHARACTERS }
       { integer-2 RECORDS    }

01 record description.
```

The entry, beginning with SD, is for the dummy sort file and appears in addition to the NORMAL FD entries for files to be processed as usual. The RECORDING MODE clause is only necessary if the files containing the unsorted and sorted data are described with this clause. The BLOCK CONTAINS clause is for specifying the block or bucket size of the work files, no other clauses are required. The record description is of the records which are to be sorted, although it is only necessary to name the key fields, all other parts of the record can be described using FILLER.

For example:

A file on disc containing 20 word records (excluding the word count) with the key-filed in the second data word, four numeric characters, is to be sorted. The data file and the work files have 1 block buckets. The sort-file is called SORT-FILE. The File Section entries would be:

```
SD  SORT-FILE
    RECORDING F
    BLOCK 512.
01  SORT-REC.
    03 FILLER PIC X(4).
    03 KEY-FIELD PIC 9(4).
    03 FILLER PIC X(72).
```

WORKING–STORAGE entries

If the APPLY data–name TO SORT–* on file-name entry has been used, entries must be made in the WORKING–STORAGE SECTION as follows:

1 If sorting is to take place on tape.

3 words = Name of library file containg sort program.

1 word = reel sequence number of this file

1 word = generation number

1 word = device type; for tapes = 4

1 word = number of work tapes to be used

1 word = reel sequence number of file containing the data to be sorted.

1 word = reply word for sort program.

For example:
Data held on a magnetic tape file, reel sequence number 0, is to be sorted using XSXM which is held on a tape called PROGRAM XSXM, reel sequence number 0, generation number 12. 4 work tapes are to be used.

The I–O–CONTROL and WORKING–STORAGE SECTION entries would be:

```
I–O–CONTROL.
  APPLY SORT–DATA TO SORT–* ON SORT–FILE.

WORKING–STORAGE SECTION.
01 SORT–DATA
   03 FILLER PIC X(12) VALUE "PROGRAM XSXM".
   03 FILLER PIC 9(6) COMP SYNC RIGHT VALUE ZERO.
   03 FILLER PIC 9(6) COMP SYNC RIGHT VALUE 12.
   03 FILLER PIC 9(6) COMP SYNC RIGHT VALUE 4.
   03 FILLER PIC 9(6) COMP SYNC RIGHT VALUE 4.
   03 FILLER PIC 9(6) COMP SYNC RIGHT VALUE ZERO.
   03 S–REPLY PIC 9(6) COMP SYNC RIGHT.
```

2 If sorting is to take place on disc.

3 words = name of library file containing sort program.

1 word = generation number.

1 word = device type; for disc = 1

3 words = name of first workfile.

1 word = generation number of first workfile.

3 words = name of second workfile.

1 word = generation number of second workfile.

1 word = reply word for sort program.

For example:

Data held on a disc file is to be sorted using XSXA which is held on a file called PROGRAM XSXA, generation number 25. The workfiles are called SORTWORKFILE, generation numbers 0 and 1. The I-O-CONTROL and WORKING-STORAGE SECTION entries would be:

```
I-O-CONTROL.
  APPLY SORT-DATA TO SORT-* ON SORT-FILE.

WORKING-STORAGE SECTION.
01  SORT-DATA.
  03 FILLER PIC X(12) VALUE "PROGRAM XSXA".
  03 FILLER PIC 9(6) COMP SYNC RIGHT VALUE 25.
  03 FILLER PIC 9(6) COMP SYNC RIGHT VALUE 1.
  03 FILLER PIC X(12) VALUE "SORTWORKFILE".
  03 FILLER PIC 9(6) COMP SYNC RIGHT VALUE ZERO.
  03 FILLER PIC X(12) VALUE "SORTWORKFILE".
  03 FILLER PIC 9(6) COMP SYNC RIGHT VALUE 1.
  03 S-REPLY PIC 9(6) COMP SYNC RIGHT.
```

Processing a COBOL SORT

It is possible to write a COBOL program which will sort data without any other processing at all; in effect this would be a "free-standing" sort. To do this the only entry in the PROCEDURE DIVISION would be:

```
SORT sort-file-name ON { ASCENDING } KEY
                       { DESCENDING }
        key-1                  key-2....
                [ON { ASCENDING } KEY
                   { DESCENDING }
        key-3                  key-4.... ]
  USING file-name-1 GIVING file-name-2.
```

The SORT can handle up to 10 key-fields (which must be declared in the SD file record description) which can be sorted into ascending or descending order. The most significant key is specified first. The USING clause specifies the name given to the input (unsorted) data file which must have been described within the program in the usual way. The GIVING clause specifies the file which will contain the output (sorted) data which is also described in the usual way. It is important that the input and output files are not opened; all opening and closing of files is carried out by the sort program.

Examples of using the SORT

Example

Input data	*Output data*
"UNSORTED" 1 BLOCK BUCKETS 10 WORD RECORDS (EXCLUDING WORD COUNT) 1 WORD BINARY KEY IN 3RD DATA WORD SORTED TO ASCENDING SEQUENCE.	"SORTED DATA" REST AS FOR INPUT FILE

The default values are to be assumed for the sort environment so No I—O—CONTROL or WORKING—STORAGE SECTION entities are required.

```
.
.
.
  MEMORY 5888 WORDS.
.
.
.
  SELECT EDS—IN ASSIGN EDS 1.
  SELECT EDS—OUT ASSIGN EDS 2.
  SELECT SORT—FILE ASSIGN EDS 3.
.
.
.
FILE▽SECTION.
FD EDS—IN
  RECORDING F
  BLOCK 512
  LABEL RECORDS STANDARD
  VALUE OF ID "UNSORTED".
01 IN—REC PIC X(40).
  FD  EDS—OUT
      RECORDING F
      BLOCK 512
      LABEL RECORDS STANDARD
      VALUE OF ID "SORTED DATA".
  01  OUT—REC PIC X(40).
  SD  SORT—FILE
      RECORDING F
      BLOCK 512.
  01  SORT—REC.
      03 FILLER PIC X(8).
      03 KEY—NO PIC 9(6) COMP SYNC RIGHT.
      03 FILLER PIC X(28).
    .
    .
    .
    .
    .
    .
  PROCEDURE DIVISION.
  PARA—1.
    SORT SORT—FILE ON ASCENDING KEY—NO USING EDS—IN
    GIVING EDS—OUT.
    STOP RUN.

****
```

This very simply sort program is only possible if input, output and work files are on the same medium and have the same block or bucket size. However, it is important to note that this type of sort process would normally be carried out with a free-standing utility.

Usually a COBOL program which includes the SORT verb carries out other processes as well. Input procedures can be carried out immediately before the records are sorted and output procedures immediately after the SORT. For example, a program could read and validate records as an input procedure, sort the valid records and use the sorted data in an update process as the output procedure. If input and output procedures are to be included, the SORT verb takes the following format:

```
SORT sort-file-name ON { ASCENDING } KEY key-1 key-2..
                       { DESCENDING }

    ON { ASCENDING } KEY key-1 key-2
       { DESCENDING }

    INPUT PROCEDURE section-name-1
                                  [THRU section-name-2]

    OUTPUT PROCEDURE section-name-3
                                  [THRU section-name-4].
```

The Procedure Division must be sectionalised and an example of the layout might be:

```
PROCEDURE DIVISION.
SORTING SECTION.
PAR-SI.

  SORT SORT-FILE ON ASCENDING STOCK-NUMBER
  AMEND-TYPE INPUT PROCEDURE VETTING OUTPUT PRO-
  CEDURE UPDATING.
  STOP RUN.

VETTING SECTION.
PAR-VI.
   .
   .
   .
UPDATING SECTION.
PAR-UI.
   .
   .
   .
****
```

The SORT verb has the effect of "PERFORMing" the input and output procedures, so both procedures must have a common exit point.

Example

An input procedure is to read the unsorted file, perform validation checks and pass only valid records to the SORT.

```
VETTING SECTION.
P-V1.
  OPEN INPUT EDS-IN.
P-V2.
  READ EDS-IN AT END GO TO P-V10.
   .
   .
   .
   .
   .
  validation checks
   .
   .
   .
  MOVE IN-REC TO SORT-REC.
  RELEASE SORT-REC GO TO P-V2.
P-V10.
  CLOSE EDS-IN.
NEXT SECTION.
   .
   .
   .
```

Important points to notice here are:

1 The verb RELEASE; this has the effect of "WRITE-ing" the record to the sort file. The format is: RELEASE record-name.

2 The file must be closed at the end of the input procedure, this statement is also the exit point for the input procedure.

Example

An output procedure is to read the sorted file, use the record to update another file and so complete the program.

```
UPDATING SECTION.
P-U1.
  OPEN OUTPUT EDS-OUT.
   .
   .
   .
P-U2.
  RETURN SORT-FILE AT END GO TO P-U10.
  MOVE SORT-REC TO OUT-REC.
   .
   .
   .
   .
   .
  updating routine
   .
   .
   .
   .
   .
  WRITE OUT-REC.
  GO TO P-U2.
   .
   .
   .
P-U10.
  CLOSE EDS-OUT
   .
   .
****
```

Important points to notice here are:

1 The verb RETURN; this has the effect of "READ-ing" the records from the sort file. The format is: RETURN file-name AT END imperative statement.

2 The file is closed at the end of the output procedure, which is also the common exit point.

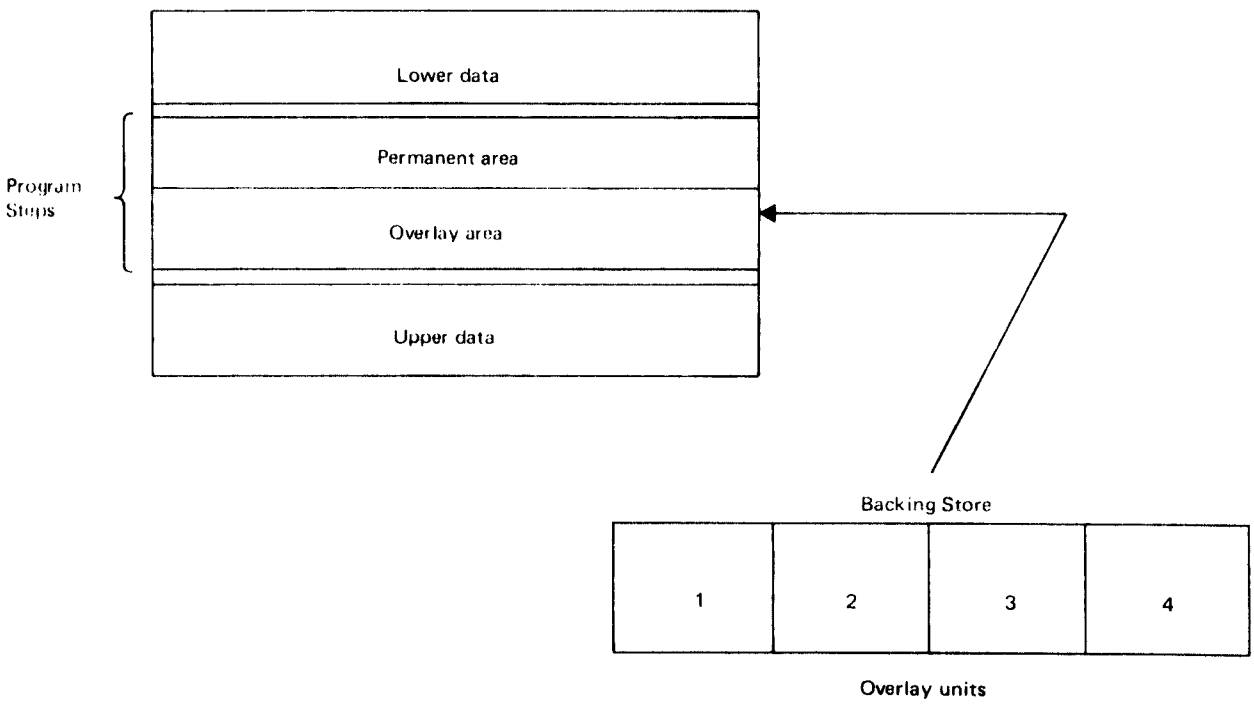
Chapter 13 Segmentation

THE THEORY OF OVERLAY

For every program, there will always be a limit to the amount of store it can occupy. This limit may be set by the size of the store on the central processor, or may be dictated by multiprogramming considerations; that is, if several programs are to be held in store simultaneously, the system must impose a limit on the space occupied by each program.

Segmentation is used in COBOL when, despite the most efficient object program being produced, the program exceeds the amount of store allocated to it. In this case, the program is allocated a smaller amount of store than it needs, which may be regarded as being split into two areas:

1 A permanent area.
2 An overlay area.



The permanent area consists of Lower Data, permanent program steps and Upper Data. The overlay area can logically be considered to be between the permanent program steps and Upper Data.

The theory of *overlay* is fundamental to segmentation in COBOL. The question of terminology should be mentioned here. In general the term "segmentation" in 1900 usage has no connection with overlay; in PLAN it refers only to splitting of the program for ease of writing and testing. It is only in COBOL that segmentation has a much more precise meaning and is synonymous with overlay; similarly, a segment refers to an overlay unit. To avoid confusion with the general 1900 terms, "overlay" will be used in this chapter for "segmentation", and "overlay unit" for "segment".

The overlay area of store will be large enough to contain one *overlay unit*. This term introduces the way in which program steps are divided. Those which are permanent will always be held in the permanent program step area. The remainder are split into overlay units, which are held on a backing store (normally magnetic tape or discs). One of these units at a time may be held in the overlay area of store, and the overlay area is thus allocated according to the longest overlay unit.
This arrangement allows a program to be as large as is necessary, no matter how little core store is available. Only a small portion of the program is held permanently in store, while the rest is held on backing store. When a part of the program on backing store is to be obeyed, it is read in as an overlay unit to the overlay area, overwriting the unit already there. This reading is carried out by a special subroutine added to the program during compilation and held in permanent. The subroutine operates automatically and need not concern the programmer.

It should be noted that only parts of the Procedure Division, that is, program steps, can be overlaid; all Data Division entries are held separately in store.

The next section considers in detail the structure of overlay, and how the program is divided between permanent and overlay.

STRUCTURE OF OVERLAY

The important difference between a segmented and a non-segmented program is that the Procedure Division of a segmented program must be written as a series of *sections*. Just as a paragraph is a series of sentences, so a section is a series of paragraphs. If overlay is to be used, paragraphs must be grouped into sections. Sections are identified by *priority numbers*, which determine whether a section is in permanent or overlay.

Sections

Sections are allocated by writing a *section name* in front of the paragraph name of the first paragraph in the section. This is followed by the word SECTION and a priority number (see the next section). The section entry therefore has the format

section-name SECTION [priority-number].

The section name must be devised according to the rules for paragraph names (see Chapter 1 "Rules for data names"). (The term "procedure name" is a general one covering both paragraph and section names.) The priority number must be in the range 00 to 99, as described below. For example,

```
FRED SECTION 56.
FIRST-PARA.
```

Section names can be used as the destinations of GO TO or PERFORM, in place of paragraph names. When they are used, the first statement of the first paragraph in the section is obeyed.

Priority numbers

As already stated, the priority number allocated to a section determines whether it is held in permanent or overlay. It should be noted that in COBOL the concept of priority has no connection with multiprogramming. In general, sections with priority numbers in the range 00 to 49 go into permanent, while those with priority numbers in the range 50 to 99 go into overlay. This rule does not apply if the SEGMENT–LIMIT clause (see the next section) is used. If no priority number is given to a section, it is assumed to be 00, and the section is held in permanent.

Overlay sections given the same priority number go into the same overlay unit. However, while a unit can contain several sections, one section will normally correspond to one overlay unit. In either case, the number of different priority numbers over 49 determines the number of overlay units in the program. Consecutive sections need not have a consecutive sequence of priority numbers. For example, a series 50, 56, 75 is permissible for three consecutive sections.

SEGMENT–LIMIT clause

This optional clause may be specified in the Environment Division, to change the limit below which segments are held in permanent. It has the format

SEGMENT–LIMIT IS priority–number.

and is written in the Object–Computer paragraph, directly after the MEMORY clause.

The purpose of this clause is to lower the priority number which divides permanent and overlay sections. As stated above, this number is normally assumed to be 50. However, an entry such as

SEGMENT–LIMIT IS 36

will indicate that sections 00 to 35 are permanent, and 36 to 99 are overlay units.

This clause may be used for one of two reasons. It may first be used to provide more overlay units if the normal range is not sufficient.

COMPILATION

When a program has been divided into sections, and the priority numbers have been established, it will then be necessary to compile the program. The sections must be introduced to the compiler in the following sequence:

1 Overlay sections in priority number sequence.
2 Permanent sections in any sequence.

The source program must therefore be arranged with sections in this order.

The program will be entered at run-time at the first overlay unit, that is, at the first statement after the Procedure Division heading.

With every program, an initial compilation is advisable. Even if it is clear that overlay must be used, the program should first be compiled ignoring this fact, that is, with no section headings.

This first compilation listing will confirm the fact that overlay is required, and will provide information as to the amount of coding generated by each paragraph. It now remains only to insert section headings and priority numbers into the source, make any sequence changes that seem necessary, and recompile to produce an overlaid program. This procedure assists the programmer in seeing the structure into which the program will fit.

PROGRAMMING FOR OVERLAY

When writing an overlaid program; certain changes may be advisable, but no changes to the logic of the program are essential.

GO TO and PERFORM can be written as the logic of the program demands, irrespective of whether the destination is in store or not. If it is not, the GO TO or PERFORM causes a branch to the overlay subroutine in permanent which reads the required unit from the backing store and enters it at the required point. If this was caused by PERFORM, a return will occur to the statement following PERFORM.

The medium used for holding overlay units on backing store determines the overlay subroutine which is incorporated in permanent during completion. The medium depends upon the medium on which the compiler is held. For example, a magnetic tape compiler, would assume magnetic tape as backing store. This, however, can be varied using the ASSIGN OBJECT—PROGRAM clause in the Environment Division. See Chapter 5.

Certain other techniques and considerations should be noted when allocating permanent and overlay sections.

Some parts of the program will always be held in permanent, namely, the overlay subroutine and other subroutines called in by the main program, as for example, MTH and ENTERed subroutines. With some COBOL compilers, it is possible for certain subroutines to be held in overlay units. This requires the use of the steering line directive *OVERLAY. If it is used with a non-integral consolidator a subroutine goes into overlay if called by one overlay unit; otherwise it goes into permanent. This course of action may require major logic changes to the program.

It may be decided to design the logic of the program so that a central control routine is in permanent, calling in the overlay units one by one in the required sequence. Note that the first overlay unit is entered immediately after loading, and this would then have to branch to the controlling routine.

The division of the rest of the program into permanent and overlay should be approached with care. The equation

N — (P + L + U)

calculates the size of the overlay area in store, where

N = maximum size for whole operation area
P = permanent program steps
L = Lower Data
U = Upper Data

The way in which paragraphs are grouped into sections depends upon two main factors:

1 The size of an overlay unit must clearly be allocated so as not to exceed the size of the overlay area.

2 Paragraphs must be grouped into sections in such a way that each overlay unit is likely to remain in core store for as long a time as possible. This is because overlay involves non-productive reading from the backing store, which should be kept to a minimum, especially from magnetic tape.

   To take an extreme example, a loop of steps repeated many times should not be written so that half is in one overlay unit and half is in another.

   Routines that are rarely used, like error procedures, should be grouped together in one overlay unit.

   These considerations may require logic changes in order to preserve the sequence of the program.

Chapter 14 Additional facilities in COBOL

This chapter describes facilities currently available in COBOL, in addition to those dealt with in the rest of the manual. Some of these are new verbs, procedures and so on, but the majority are existing features extended to allow more complex operations. While some of these extra powers are useful, enabling data to be manipulated in a more concise way in terms of source program statements, it should be appreciated that there is little or no saving in the final object program. Some of the options are in fact of no practical use except to give compatibility with other COBOL compilers. It should be remembered that it is important to keep the whole program as simple as possible, and particularly the Procedure Division by minimising the use of complex, nested or compounded sentences. Extensive use of the additional facilities described brings no benefits in either comprehensibility or shorter programs, and is liable to increase programmer errors.

First, some general points are made about program writing; then, facilities are dealt with by division. Finally, the COPY option, which enables program entries to be copied from the COBOL library, is considered.

GENERAL POINTS

The following are general facilities which concern the writing of the program as a whole.

Optional words

Optional words have no value in a COBOL program, except to improve the sense. They are words which may be written in the program text to make it closer to normal English. For example,

PIC IS character–string
WRITE record–name AFTER ADVANCING integer LINES.

The optional words are those written in capitals not underlined. Most statements can include words like these, which the compiler checks but does not use. They are printed out on the compiler listing.

It is not intended to give a list of optional words here, but they will all be found in *COBOL*.

Extra figurative constants

Figurative constants are defined in Chapter 7 as constants used in the program by referring to names assigned to them. Certain others can be used in full COBOL, and a list of these is given below.

ZERO) ZEROES)	Alternatives to ZEROS, representing one or more of the value 0.
SPACE	Alternative to SPACES, representing one or more spaces.
QUOTE) QUOTES)	These provide a means of filling a field with quotation marks. Since the quotation mark is the only character which cannot be included in a non-numeric literal, these two constants are occasionally useful.
HIGH—VALUE) HIGH—VALUES)	These constants fill a field with the highest character value in the 1900 collating sequence, # 77.
LOW—VALUE) LOW—VALUES)	These constants fill a field with the lowest character value in the 1900 collating sequence, 0.

The above constants are handled in the same way as SPACES and ZEROS but are rarely used; they apply chiefly in relation tests, where from a particular part in the program it is required to ensure that a test which up to then could give two results, will now always show the same result. This can be achieved by placing the highest or lowest value in a field, so that the field compared with it will yield a predictable result.

ALL non-numeric literal — This constant must be associated with a data item, and fills the field with repetitions of the specified literal. For example,

```
IF N = ALL "9"
```

will generate a string of 9's of the same length as N.

When the field is not an exact multiple of the non-numeric literal, the string of characters is truncated at the right hand end. If NUMBER is defined as PIC 9(12) the statement

```
MOVE ALL "0123456789" TO NUMBER.
```

will place the value

```
012345678901
```

in NUMBER.

Examples

```
MOVE SPACE TO OUTPUT-AREA.
DISPLAY QUOTE, CODE, QUOTE.
```

If the value of CODE were SOLD—OUT, the message would be displayed as

```
"SOLD—OUT"
```

```
MOVE ALL "A" TO EMPTY-AREA.
MOVE ALL "ABC" TO EMPTY-AREA.
```

If EMPTY—AREA were eight characters long, AAAAAAAA and ABCABCAB respectively would be moved into the field.

Qualification

As explained in Chapter 4, data names should normally be unique within a program. However, it happens frequently that an item is common to several files in the program, sometimes complete records have the same format on three or four files.

An item may have the same data name each time it is described in different parts of the Data Division. This data name is then *qualified* when it is used in a procedure statement. In other words, a distinction must be made between the two different fields.

The format for qualification of a data name is as follows:

data—name $\left\{\begin{matrix}\text{IN}\\\text{OF}\end{matrix}\right\}$ qualifier

For example, suppose that a program has four files, M—IN, CURRENT, M—OUT and PRINT—OUT, each containing one type of record, and that the item QUANTITY appears in each. The four fields can be addressed as

QUANTITY IN M—IN
QUANTITY IN CURRENT
QUANTITY IN M—OUT
QUANTITY IN PRINT—OUT

The qualifier can be any name which is higher in the data structure, provided that the result is unique. File names have been used as qualifiers above. However, if M—IN contains the following record, IN—REC,

```
01  IN-REC
    ≡
02  ORDER-DETAILS.
    03  REA-NO  PIC X(5).
    03  QUANTITY  PIC 9(3).
```

then QUANTITY IN M—IN can also be referred to as
QUANTITY IN IN—REC
or
QUANTITY IN ORDER—DETAILS.

Qualification, above all, reduces the need for invention on the part of the programmer, at the expense of easier writing. It can sometimes make the coding easier to follow.

ENVIRONMENT DIVISION

The Input—Output Control paragraph

This optional paragraph can be written in the Environment Division and allows files to share input/output areas. It can also be used to specify the points at which rerun is to be established (see *COBOL*).

The format of the I–O–Control paragraph is as follows:

I–O–CONTROL.
  [SAME [RECORD]AREA FOR file–name–1 file–name–2
                    [file–name–3...]].

The heading is written in column 12. The SAME AREA clause specifies that input or output areas (according to whether the files named are input or output) are to be shared between file–name–1, file–name–2 and so on.

If the RECORDoption is included, only the record area is shared by the files. If the word RECORD is omitted, the files share the same buffer area as well. In this case only one of the files named in the SAME AREA clause can be open at any one time.

The SAME AREA clause is useful in that it saves store, and also reduces physical transfer between record areas.

Other options which can be included in the I–O–Control paragraph are given in the COBOL reference manuals.

The File–Control paragraph

A number of extra clauses can be written in this paragraph to describe files held on direct access devices. They may specify, for instance, the access mode, the processing mode and organisation of the file.

For full details of these clauses, reference should be made to Chapter 4 of *COBOL*.

BLANK WHEN ZERO

A special case is the editing clause
BLANK WHEN ZERO

It is used in addition to the Picture clause to suppress zeros in a numeric field containing decimal places.

In the following entry,

```
02 COST PIC ZZZV.999 BLANK ZERO.
```

the field will contain nothing but spaces when its value is zero.

BLANK WHEN ZERO is ignored by the compiler if it appears in the same sentence as a Picture clause containing the cheque protection symbol.

The RENAMES clause

The RENAMES clause may be used to give a new name to a field or a set of consecutive fields. It does not allow the size or nature of the fields to be changed, but merely permits alternative and/or overlapping grouping of fields in a record.

The special level number 66 is assigned to each RENAMES entry. The clause has the following format:

66 data—name—1 RENAMES data—name—2
[THRU data—name—3].

Data—name—1 is the new name given to a single item, data—name—2 or to a number of items, from data—name—2 to data—name—3.

It should be noted that the entry, unlike a REDEFINES clause, does not immediately follow the original definition. 66 level entries must be written at the end of the record description.

RENAMES cannot be used for fields at levels 01 or 88. Neither data—name—2 nor data—name—3 can have an OCCURS clause associated with it.

The renaming of items usually occurs when the programmer wants to refer to several items in the Procedure Division, in a different grouping from that in which they were defined in the record.

Example
It is sometimes convenient to have group fields which overlap. The following example defines three groups, the third of which overlaps the others. GROUP—3 consists of fields B, C and D.

```
01 IN-REC.
   02 GROUP-1.
      03 A PIC 99.
      03 B PIC XX.
   02 GROUP-2.
      03 C PIC 9(6).
      03 D PIC X(5).
   02 FILLER.
   66 GROUP-3 RENAMES B THRU D.
```

PROCEDURE DIVISION

This section describes compound conditions and more complex forms of IF, in addition to those dealt with in Chapter 7. As regards IF, the extra features do not as a rule produce greater efficiency of the object program. In general, it is wise to avoid complex, nested or compounded sentences. These sentences may in fact lead to program testing problems. The same point applies to PERFORM, whose complex forms were given in Chapter 7.

Compound conditions

In Chapter 7, only simple conditions were described, that is, those which had one condition following each IF statement.

Compound conditions allow two or more simple conditions to be combined by means of *logical AND and OR*, so as to produce one result. These conditions follow the normal rules of logic, so that

A AND B

is true if both A and B are true, and

A OR B

is true if either A or B (or both) is true.

A compound condition therefore takes the following form:

IF { [NOT] condition AND condition) / [NOT] condition OR condition) } imperative statements

AND linking a pair of conditions means that both must be true for the whole condition to be satisfied. OR linking a pair of conditions means that one, or both of the conditions must be true for the whole condition to be satisfied.

NOT may precede either condition of a pair. The results obtained by evaluating the pair (with and without NOT) are shown in the table below.

If A is	If B is	A AND B	A OR B	NOT A AND B	NOT A OR B
True	True	True	True	False	True
False	True	False	True	True	True
True	False	False	True	False	False
False	False	False	False	False	True

By working on the same principles, the results of A AND NOT B and A OR NOT B can easily be obtained.

For example,

```
IF A > B AND A = C GO TO D.
IF COUNT < 100 OR CODE NOT NUMERIC
   GO TO ERROR.
```

Compound conditions can be linked, so that three or more simple conditions are strung together by AND or OR. In such expressions, pairs of simple conditions are taken and each pair evaluated to give a result. If all ANDs or all ORs are involved, then each pair is evaluated as it occurs. Otherwise, ANDs are tested before ORs. It should be noted, however, that linked compound conditions do not produce a shorter object program, so that, if there is any danger of confusion, several IF statements should be used instead.

Consider the following examples.

```
IF A > B AND C = D AND E < F GO TO CALC.
IF A > B OR C = D OR E < F GO TO CALC.
```

In the first, A > B AND C = D is evaluated, and then C = D AND E < F. Each condition must be true for the whole condition to be satisfied.

In the second, conditions are tested in the sequence A > B OR C = D, C = D OR E < F. Only one of the pairs of conditions need be true for the whole condition to be true.

In the example below, which contains both AND and OR, the effect is as if the conditions linked by AND were enclosed in brackets.

```
IF A = B AND CHECK-1 NOT > 74 OR SWITCH
   NOT = 1 GO TO PROCESS-A.
```

The bracketed conditions are evaluated first.

```
IF (A = B AND CHECK-1 NOT > 74) OR SWITCH
   NOT = 1 GO TO PROCESS-A.
```

The branch to PROCESS—A occurs if A is equal to B *and* if CHECK—1 is in the range 0 to 74. If either of these conditions is not true, the branch may still occur provided SWITCH is not set to 1.

Brackets can be used to alter the interpretation of linked conditions. When parts of compound conditions are enclosed in brackets in the program, these parts will be evaluated first. For example, the condition given above could be written

```
IF A = B AND (CHECK-1 NOT > 74 OR SWITCH
   NOT = 1) GO TO PROCESS-A.
```

The branch will now occur if A is equal to B, and if one or both of the bracketed conditions is true.

The brackets in compound conditions can also be nested, the contents of the inner brackets being evaluated first. For example,

```
IF A = B AND (((C = D OR E = A) AND G > H
   AND K NOT = ZERO) GO TO PARA2.
```

When writing linked compound conditions, the simplest tests should be placed first, as a means of efficiency. In the example below the test on SWITCH should have been placed first, as shown below

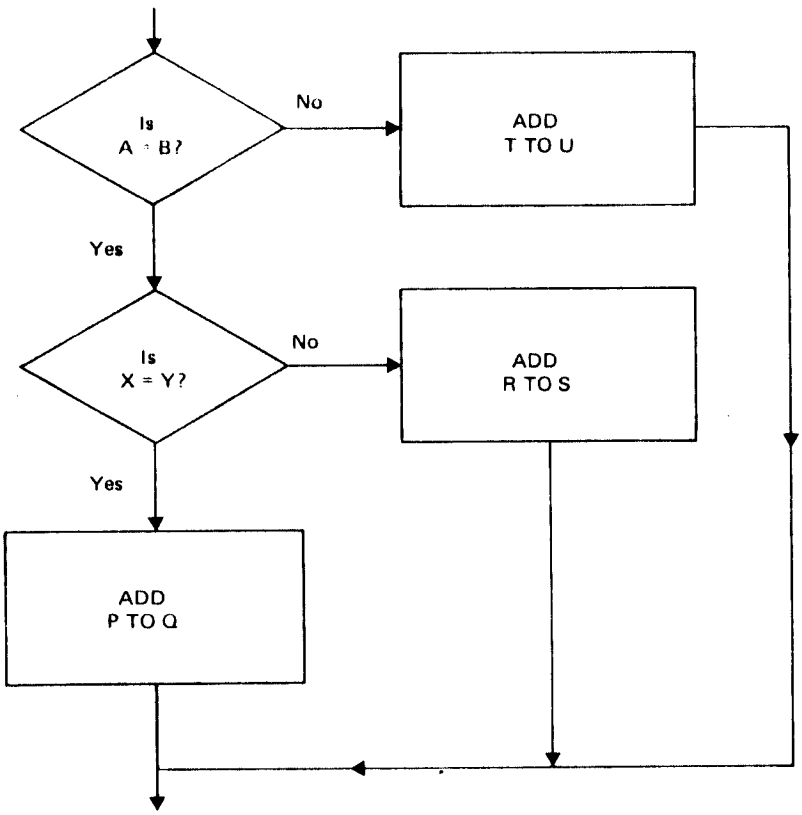
```
IF SWITCH NOT = 1 OR A = B AND CHECK NOT
> 74 GO TO PROCESS-A.
```

Nested IF statements

The IF statement can itself have conditional statements dependent on it, as shown in the example below,

```
IF A = B IF X = Y ADD P TO Q ELSE ADD R TO S
ELSE ADD T TO U.
```

In sentences such as this, each ELSE clause is considered to belong to the nearest preceding IF which has not already been paired with an ELSE clause. The flowchart corresponding to the condition given above shows the sequence of events.



The following is an alternative way of nesting the conditional statements.

```
IF A NOT = B ADD T TO U ELSE IF X = Y
ADD P TO Q ELSE ADD R TO S.
```

Both these approaches are prone to error and, moreover, nesting IF statements does not save object code. The more straight-forward approach as shown below is therefore preferable.

```
IF A NOT = B ADD T TO U GO TO NEXT-1.
IF X = Y ADD P TO Q GO TO NEXT-1.
ADD R TO S.
NEXT-1.
```

Peripheral verbs

A number of extra options are available with the peripheral verbs, OPEN, READ, WRITE and CLOSE. Most of these, however, involve only direct access files. For full details, reference should be made to Chapter 6 of *COBOL*.

A few other formats of READ and WRITE can be used and these are considered below.

READ

The INTO option can be used with READ to move data straight into a named area. It has the format

READ file–name [INTO data–name] AT END
imperative statement.

The option can be used only if the file contains records of one type, that is, if an 01 level entry appears only once in the file definition. The effect is as if the statement

MOVE record–name TO identifier

immediately followed a normal READ.

READ A INTO B.

has the same meaning as

READ A MOVE A TO B.

The record read is then available in both the input record area and the area specified by data–name.

WRITE

The FROM option can be used with WRITE in a similar way to READ...INTO. It has the format

WRITE record–name [FROM data–name].

and has the effect as if the statement

MOVE data–name TO record–name

had immediately preceded a normal WRITE.

WRITE A FROM B.

has the same meaning as

move b to a write a.

In addition to WRITE...AFTER, which may be used to space lines on the printer, a further option is available.

$$\text{WRITE record–name [BEFORE} \begin{Bmatrix} 1 \\ 2 \\ \text{CHANNEL–n} \end{Bmatrix}]$$

can be written so that 1 or 2 lines are spaced, or alternatively, the paper is moved to a specified channel of the paper tape loop, *after* the record is printed.

It should be noted that more than two lines could in theory be specified, In practice however, this is not recommended as a number greater than 2 would generate a series of hardware machine code instructions.

COPY

The COPY facility is designed to reduce the work involved in writing a COBOL source program. Instead of writing an entry or series of entries out in full, the programmer can specify that the compiler is to search the *COBOL Library* for those particular source lines. Examples of entries that can be held on the library and extracted in this way include complete file descriptions, record descriptions or complete sections (that is, sets of paragraphs) from the Procedure Division. The saving of programmer effort can therefore be very great.

The COBOL Library

The library will be held on a backing store so that it can be searched during compilation. It is divided into three sections, corresponding to the three program divisions where COPY can be used (Environment, Data and Procedure Divisions).

Each section will hold a series of library entries for the division in question. An entry consists of two or more items, each item being given a six-digit sequence number, starting at 000000 and rising in steps of 100.

The first item in an entry (line 000000) is the *library name*, which will be quoted in the COPY statement by the program which is consulting the library. The remaining items in the entry are the lines of source COBOL which are to be inserted into the program at the point where the COPY statement is written. (The items cannot themselves include COPY statements.)

The following entry might appear in the Procedure Division section of the library:

```
000000    ENTRY-A.
000100    IF A = B GO TO X.
000200    IF A < B GO TO Y.
000300    GO TO Z.
```

Such an entry will allow any COBOL source program to include these statements in its Procedure Division, merely by writing

```
  COPY ENTRY-A
```

after a paragraph name.

COPY is handled by all current COBOL compilers. The library is set up and amended on tape or disc as backing store, using a special program, #XE99.

Format of COPY

The format of the COPY statement is as follows:

```
COPY library-name [REPLACING word-1 BY word-2
  [word-3 BY word-4] ... ]
```

In the Environment and Procedure Division the COPY entry will be a complete sentence starting with the word COPY and this entire sentence will be replaced by the library entry.

In the Data Division the word COPY must be preceded by

```
  FD file-name
```

or

```
  level-number data-name
```

where level-number is 01.

In these cases the word COPY and the sentence following it will be replaced by the library entry; however, FD or level-number will remain in the source program and the name which follows it will replace the equivalent name in the library entry (unless the two names are the same). An example of this is considered in the next section.

The REPLACING option allows the library entry to be modified. For example, if the following statement appeared in the source,

```
COPY ENTRY-A REPLACING X BY PARA-74.
```

the first line copied would become

IF A = B GO TO PARA–74.

This modification is not a permanent one, that is, the library on the backing store remains unchanged.

Use of COPY

ENVIRONMENT DIVISION

COPY can be used after any paragraph name, but not after a section name.

Example
The following entries are written in the source program:

```
FILE-CONTROL.
COPY MASTER-LAYOUT.
```

A library entry, as follows, might appear in the COBOL library.

```
000000    MASTER–LAYOUT
000100    SELECT IN–ONE ASSIGN TAPES 1.
000200    SELECT IN–TWO ASSIGN TAPES 1.
000300    SELECT AMENDED–MASTER ASSIGN TAPES 1.
000400    SELECT LISTING ASSIGN PRINTER 1.
```

The appropriate SELECT ... ASSIGN clauses will now be copied into the source.

DATA DIVISION

COPY can be used after an FD or 01 level entry.

Example
The source program entry

```
FD MTIN COPY MASTER.
```

could result in the following library entry being copied.

```
000000    MASTER
000100    FD MASTER
000200    RECORDING F
000300    BLOCK 50 RECORDS
000400    LABEL RECORDS STANDARD
000500    VALUE OF ID "UPDATE FILE".
```

The appropriate file description entries will be compiled, but with the name MTIN being taken as the file name. In other words, the entry will start

FD MTIN
RECORDING F and so on

It is possible for the one library entry to cover the whole file description and record descriptions for a file.

PROCEDURE DIVISION

An example of this has already been considered above. This comprised one paragraph; however, if the COPY statement is written after a section name then a series of paragraphs can be copied.

Program conversion
Word-count word

As explained in Chapter 7—Peripheral verbs, the length of a record on magnetic media is held in words by the word-count word.

By specifying the Recording Mode clause the Housekeeping will insert the correct values automatically.

However, it is possible for the programmer to insert his own values for writing variable length records.

In this case the RECORDING MODE clause should be omitted and an area allocated at the beginning of the record, of one word of binary

e.g.

```
01  MT-REC.
    03 WD-CT PIC 9(6) COMP SYNC RIGHT.
      ≀    ≀
    MOVE 6 TO WD-CT.
    WRITE MT-REC.
      ≀    ≀
    MOVE 10 TO WD-CT.
    WRITE MT-REC.
```

The PICTURE clause

In addition to the characters, mentioned in Chapter 6, allowable in the picture clause is PIC 1.

1 represents a bit position containing a binary digit where the field is held in binary. For example

```
PIC 1(24).
```

denotes a binary field one word long.

Non-contiguous storage

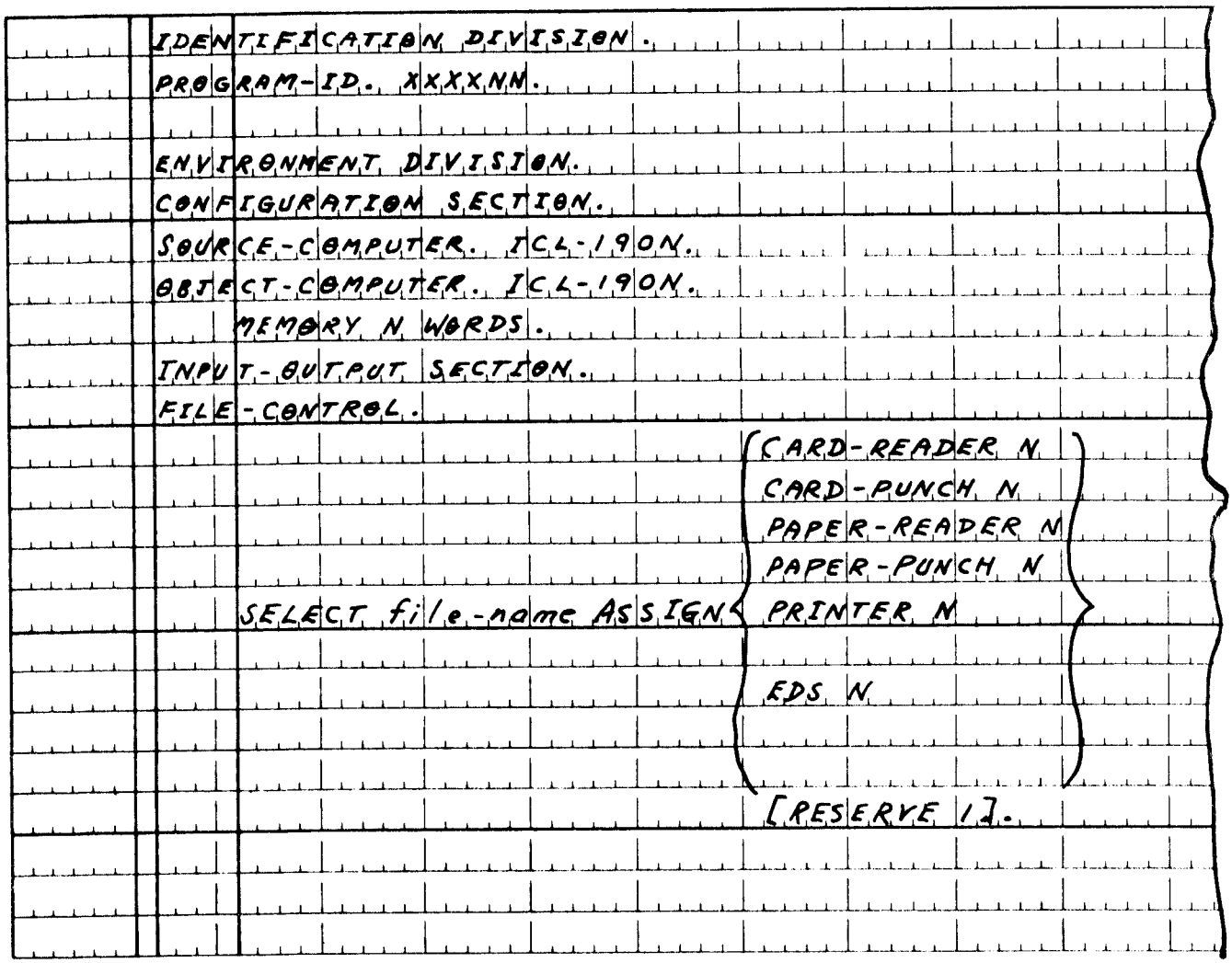
Elementary fields which are independent of any other fields and are used in isolation can be given the special level number 77

e.g.

```
WORKING-STORAGE SECTION.
77 W-LNCT PIC 9(6) COMP SYNC RIGHT.
```

However the 77 level has no advantage over the 01 level and should no longer be used.

Appendix 1 Simplified COBOL format guide



DATA DIVISION.

FILE SECTION.

One entry per input and output file consisting of
A file description
One record description for each type of
record within the file.

File description for slow peripherals

FD file-name
LABEL RECORDS rec-name-1, rec-name-2, ... rec-name-n

File description for magnetic tape or disc

FD file-name
BLOCK n { RECORDS / CHARACTERS }
LABEL RECORDS STANDARD [WITH GENERATION-NO]
VALUE OF FD IS literal-1
[ACTIVE-TIME IS literal-2]
[DATA RECORDS rec-name-1, rec-name-2, ... rec-name-n]

Example record description

01 rec-name-1.
02 field-name-1 PIC 99.
02 field-name-2.
03 sub-field-name-1 PIC XX.
03 sub-field-name-2 PIC 9.
02 field-name-3 PIC 9(6).

Note:

1. Only elementary items (fields which are not further subdivided) may have a PICTURE clause.

2. FILLER may be used for input fields which are to be ignored and for output fields which are to be space-filled by Procedure Division statements.

02 FILLER PIC XXX.

3. The following clauses may also be used:

a) COMP and SYNC RIGHT

02 field-name-1 PIC 99 COMP SYNC RIGHT.

b) REDEFINES

02 field-name-2 REDEFINES field-name-1...

c) OCCURS

02 field-name-1 PIC XX OCCURS 6.

d) Other usage clauses

DISPLAY	Characters	Sign combined with top data character
DISPLAY-3	Characters	Sign separate in most significant position

WORKING-STORAGE SECTION.

This may contain:

a) Non-contiguous fields each field has no connection with any other.

b) Contiguous fields a record and field level structure as in the File Section using the same notation.

In either case the VALUE clause can be used:

02 field-name-1 PIC 999 VALUE 120.

PROCEDURE DIVISION.

Note In the formats which follow:
A and B = data-names or literals
C = a data-name
PN1 PN2 ... = paragraph-names
F1 F2 ... = file-names
R = a record-name

Basic verbs

ADD A B GIVING C.
ADD A TO C.

SUBTRACT A FROM B GIVING C.
SUBTRACT A FROM C.

MULTIPLY A BY B GIVING C.

[ROUNDED and ON SIZE ERROR any imperative statement] can be used with any of the above verbs.

STOP {literal / RUN}.
DISPLAY A [B] .....
MOVE A TO C.

Sequence control

GO TO PN1.
GO TO PN1 PN2 PN3 ... PNN DEPENDING ON C.

IF A [NOT] { > / < / = / POSITIVE / NEGATIVE / ZERO / NUMERIC / ALPHABETIC } B any sentence.

PERFORM PN1 [THRU PN2].
EXIT.

```
ENTER language-name subroutine-name [USING
      parameter-1 ... parameter-N].
CALL subroutine-name [USING
      parameter-1 ... parameter-N].
EXIT PROGRAM.

Peripheral handling

OPEN [INPUT F1 F2 ... FX] [OUTPUT F3 F4 ... FY].

READ F1 AT END any imperative statement.

                  { 1          }
WRITE R [AFTER    { 2          } ].
                  { CHANNEL-1  }
                  {    =       }
                  { CHANNEL-7  }

CLOSE F1 F2 ... FN.
```

Appendix 2 The 1900 Series internal character code

The code (excluding shift characters) is shown below in octal form

0	#00	space	#20	@	#40	P	#60
1	#01	!	#21	A	#41	Q	#61
2	#02	"	#22	B	#42	R	#62
3	#03	#	#23	C	#43	S	#63
4	#04	£	#24	D	#44	T	#64
5	#05	%	#25	E	#45	U	#65
6	#06	&	#26	F	#46	V	#66
7	#07	'	#27	G	#47	W	#67
8	#10	(	#30	H	#50	X	#70
9	#11	)	#31	I	#51	Y	#71
:	#12	*	#32	J	#52	Z	#72
;	#13	+	#33	K	#53	[	#73
<	#14	,	#34	L	#54	$	#74
=	#15	-	#35	M	#55	]	#75
>	#16	.	#36	N	#56	↑	#76
?	#17	/	#37	O	#57	←	#77

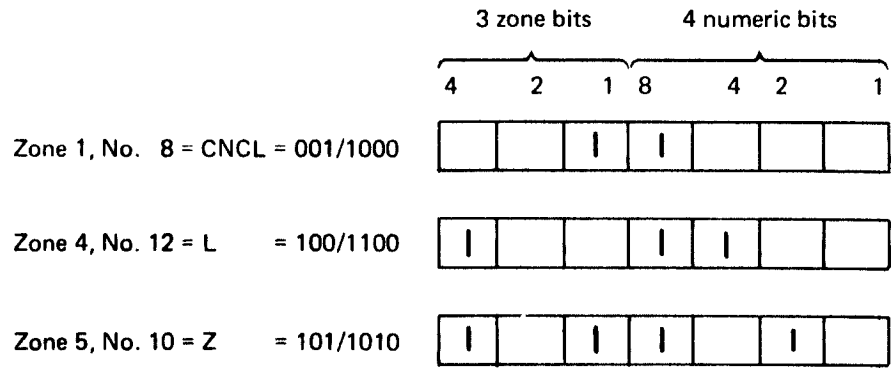
Appendix 3 The ICL 64 - character card code

Symbol	*Card Punching*	*Symbol*	*Card Punching*	*Symbol*	*Card Punching*
0	0	F	10/6	–(minus/hyphen)	11
1	1	G	10/7	" (quotes)	11/0
2	2	H	10/8	/ (solidus)	0/1
3	3	I	10/9	+ (plus)	10/2/8
4	4	J	11/1	. (stop)	10/3/8
5	5	K	11/2	; (semi-colon)	10/4/8
6	6	L	11/3	: (colon)	10/5/8
7	7	M	11/4	' (apostrophe)	10/6/8
8	8	N	11/5	! (exclamation)	10/7/8
9	9	O	11/6	[(left bracket)	11/2/8
space	NONE	P	11/7	$ (dollar)	11/3/8
& (ampersand)	10 or 10/0	Q	11/8	* (asterisk)	11/4/8
# (number)	3/8	R	11/9	> (greater than)	11/5/8
@	4/8	S	0/2	< (less than)	11/6/8
((left parenthesis)	5/8	T	0/3	↑	11/7/8
) (right parenthesis)	6/8	U	0/4	£ (pound)	0/2/8
] (right bracket)	7/8	V	0/5	, (comma)	0/3/8
A	10/1	W	0/6	% (percentage)	0/4/8
B	10/2	X	0/7	? (question)	0/5/8
C	10/3	Y	0/8	= (equals)	0/6/8
D	10/4	Z	0/9	←	0/7/8
E	10/5				

Appendix 4 1900 Paper tape codes

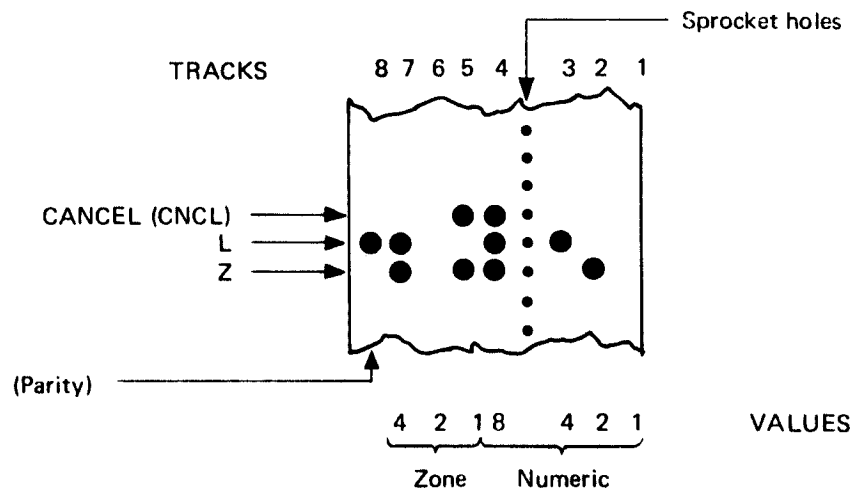
The 8-Track (7 Data Bit) 1900 Paper Tape Code

The 8-track paper tape code employs 7 data bits and 1 parity bit. Of the 7 data bits, three are used for zone identification (0—7) and 4 for numeric identification 0—15. The following three examples illustrate the use of the two groups of bits. A punching is taken to represent binary 1 and no punching to represent binary 0.



N.B. No parity is indicated above

The actual representation on tape would be as follows:



For full 1900 code see table overleaf.

8-Track 1900 Paper Code, using 7 Data Bits (The eighth track is employed only as a parity track, if required)

Zone / Numeric	0 (000)	1 (001)	2 (010)
0000	TC_0 (NULL)	TC_7 Data Link Escape	space
0001	TC_1 Start of Heading	DC_1 Device Control	! (exclamation)
0010	TC_2 Start of Text	DC_2	" (quotes)
0011	TC_3 End of Text	DC_3	# (number)
0100	TC_4 End of Transmission	DC_4 (STOP)	£ (pound)
0101	TC_5 Enquiry	TC_8 Negative Acknowledge	% (per cent)
0110	TC_6 Acknowledge	TC_9 Synchronous Idle	& (ampersand)
0111	BEL Bell, alarm	TC_{10} End of Transmission Block	' (apostrophe)
1000	FE_0 Backspace	CNCL Cancel	((l. parenth.)
1001	FE_1 Horizontal Tabulation	EM End of Medium	) (r. parenth.)
1010	FE_2 New Line	SS Start of Special Sequence	* (asterisk)
1011	FE_3 Line Feed (Start new line)	ESC Escape	+ (plus)
1100	FE_4 Form Feed	IS_4 File Separator	, (comma)
1101	FE_5 Carriage Return	IS_3 Group Separator	– (hyphen/minus)
1110	SO Shift Out	IS_2 Record Separator	. (period)
1111	SI Shift In	IS_1 Unit Separator	/ (solidus)

NOTE 1 Synchronous Idle: A transmission control character used by a synchronous transmission system in the absence of any other character (Idle condition) to provide a signal from which synchronism may be achieved or retained.

2 The code as shown above is that used on the 1900 series. It is the Standard I.S.O. 8-track code, with the following minor exceptions:

@ (at) and _ (underline) have been interchanged,
∧ and \ have been replaced by ↑ and ← respectively.

3 (011)	4 (100)	5 (101)	6 (110)	7 (111)	Zone / Numeric
0	@ (at)	P	_ (underline)	p	0000
1	A	Q	a	q	0001
2	B	R	b	r	0010
3	C	S	c	s	0011
4	D	T	d	t	0100
5	E	U	e	u	0101
6	F	V	f	v	0110
7	G	W	g	w	0111
8	H	X	h	x	1000
9	I	Y	i	y	1001
: (colon)	J	Z	j	z	1010
; (semi colon)	K	[(l.h. brkt)	k	} Reserved for	1011
< (less than)	L	$ (dollar)	l	} National	1100
= (equals)	M	] (r.h. brkt)	m	} Characters	1101
> (greater than)	N	↑	n		1110
? (question)	O	←	o	\\\ (delete)	1111

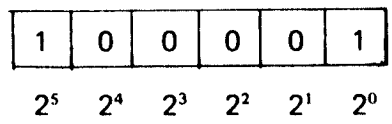
6-Bit, 3 Shift Internal Machine Form

When using the seven data bit paper-tape code it will be necessary, on input, to condense the seven data bit code into six data bits and also to expand the internal six data bits to seven on output from the store. This is achieved by condensing the three-bit zone identification into a two-bit zone identification as follows:

(a) The middle bit of the zone is dropped and the two remaining bits are compared.

(b) If the bits are equal, a 1 bit is substituted for the right-hand bit and is inserted in position 2^4 of the character; the left-hand bit enters the position 2^5 unaltered.

(c) If the bits are unequal, a 0 bit is substituted for the right-hand bit and is inserted in position 2^4 of the character location; the left-hand bit enters position 2^5 unaltered. For example, the zone component for tape character A is condensed as below:
1 Ø 0 The middle bit is discarded.
1 0 The remaining bits are compared and are found to be unequal.
Therefore, 0 is generated and inserted in 2^4; the left-hand bit (1) enters 2^5.



Problems are thus immediately introduced. For example both zones 010 and 000 would have to be condensed to 00. Similarly 101 and 111 would both be represented as 11. The problem, then, clearly is one of identification, and for this purpose three identifying characters known as the shift characters have been introduced.

Appendix 5 Reserved words

In the following list, optional words are marked with a single asterisk [*]. Words which are sometimes optional and sometimes key words depending on the context are marked with a double asterisk [**].

All other members of the list are COBOL keywords.

* ABOUT
ACCEPT
ACCESS
ACTIVE-TIME
ACTUAL
ADD
* ADVANCING
AFTER
ALL
ALPHABETIC
ALPHANUMERIC
ALTER
* ALTERNATE
AN
AND
APPLY
* ARE
* AREA
* AREAS
ASCENDING
ASSIGN
AT
* AUTHOR
BEFORE
BEGINNING
BIT-nn
(where 00 nn 23)
BITS
BLANK
BLOCK
BY
CALL
CARD-PUNCH
CARD-READER
CASSETTES
CASSETTE-*
CHANNEL-1
CHANNEL-2
CHANNEL-3
CHANNEL-4
CHANNEL-5
CHANNEL-6
CHANNEL-7
* CHARACTERS
CHECK
CHEQUE
* CLASS
CLOCK-UNITS
CLOSE
COMMA
COMP
COMPUTATIONAL
COMP-1
COMPUTATIONAL-1
COMPUTE
CONFIGURATION
CONSTANT
* CONTAINS
* CONTROL
COPY
CORE-MT
CORE-EDS
CORR
CORRESPONDING
CURRENCY
DATA
*DATE
* DATE-COMPILED
* DATE-WRITTEN
DECIMAL-POINT
DECLARATIVES
DELETE
DEPENDING
DESCENDING
DIGITS
DIRECT
DISPLAY
DISPLAY-1
DISPLAY-2
DISPLAY-3
DISPLAY-4
DISPLAY-5
DIVIDE
DIVISION
DOWN
EDS
EDS-*
ELSE
* END
ENDING
ENTER
ENVIRONMENT
EQUAL
EQUALS
* ERROR
* EVERY
EXAMINE
EXCEEDS
EXIT
FD
FDS
FDS-*
FILE
FILE-CONTROL
FILE-LIMIT
FILE-MESSAGES
FILLER
FIRST
FIRST-LEVEL
FLOAT
FOR
FROM
GENERATION-NO
GIVING
GO
GREATER
HAND-KEYS
HAND-VALUE
HIGH-VALUE
HIGH-VALUES
ID
IDENTIFICATION
IF
IN
INCLUDE
INDEX

INDEX-BUFFER
INDEXED
INPUT
INPUT-OUTPUT
* INSTALLATION
INTO
INVALID
I-O
I-O-CONTROL
** IS
JUST
JUSTIFIED
KEY
KEYS
LABEL
LEADING
LEAVING
LEFT
LESS
* LINES
LINKAGE
* LOCATION
LOCK
LOW-VALUE
LOW-VALUES
LOWER-BOUND
LOWER-BOUNDS
MCF
MCF-*
MEMORY
* MODE
MOVE
MULTIPLY
* MULTIPLE
NEGATIVE
NEXT
NO
NOT
NOTE
NUMERIC
OBJECT-COMPUTER
OBJECT-PROGRAM
OCCURS
** OF
OFF
OMITTED
** ON
OPEN
OPTIONAL
OR
ORGANISATION
ORGANIZATION
OTHERWISE
OUTPUT
PAPER-PUNCH
PAPER-READER
PERFORM
PIC
PICTURE
* PLACES
POINT
POSITION
POSITIVE
PRINTER
PROCEDURE
PROCEED
PROCESSING
PROGRAM
PROGRAM-ID
* PROTECT
QUOTE
QUOTES
RANDOM
RANGE
READ
** RECORD
RECORDING
RECORDS
REDEFINES
REEL
RELEASE
REMAINDER
* REMARKS
RENAMES
REPLACING
RERUN
RESERVE
RETAIN
RETURN
REVERSED
REWIND
REWRITE
RIGHT
ROUNDED
RUN
SAME
SD
SECOND-LEVEL
SECTION
* SECURITY
SEEK
SEGMENT-LIMIT
SELECT
* SENTENCE
SENTINEL—PROCESSING
SEQUENCED
SEQUENTIAL
SET
SIGN
SIGNED
SIZE
SORT
SORT*
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
* STATUS
STOP
SUBTRACT
SUPPRESS
SYMBOLIC
SYNC
SYNCHRONISED
SYNCHRONIZED
TALLYING
TAPE
TAPES
TAPE-*
* THAN
* THEN
THROUGH
THRU
*TIME
TIMES
TO
TRANSFER-REPLY
TYPE
TYPEWRITER
UNEQUAL
UNIT
UNTIL
UP
UPON
UPPER-BOUND
UPPER-BOUNDS
* USAGE
USE
USER-SENTINEL
USING
VALUE
VALUES
VARYING
* WHEN
WITH
WORDS
WORKING-STORAGE
WRITE
ZERO
ZEROES
ZEROS

Appendix 6 1900 COBOL compilers

COBOL compilers XEKA/B and XEKD

XEKA and XEKB are COBOL compilers on magnetic tape or disc for 1900 machines of 16K and above words of store, XEKD being the corresponding compiler for 2903 and 2904.

Source programs are input from punched cards, paper tape, magnetic tape or direct access, and semicompiled object programs may be output to direct access, magnetic tape, paper tape or punched cards. The compiler will produce object programs which can handle files held on cards, paper tape, magnetic tape, direct access and line printer.

In addition to providing compilation facilities, all three compilers are capable of creating and amending source programs held on magnetic media.

If 'found' in more than the minimum requirements of 13K of core XEKA and XEKB employ different methods of optimising their own operations, XEKB by increasing the size of its overlays and XEKA by increasing the space allocated to tables, although the object program generated will be the same in both cases.

The necessary minimum environment for these compilers is:

One central processor with at least 13K of available words.
Four magnetic tape decks or at least one exchangeable disc transport.
One paper tape or card reader if the source is held on that medium.
One console typewriter.
One line printer.
One paper tape or card punch if the object program is required on that medium.

Further details of the compilers are given in the manual *1900 COBOL Compilers*.

Appendix 7 Variable length subroutines

PROCESSING VARIABLE LENGTH FIELDS BY SUBROUTINE

The four subroutines COBDIST, COBDISTM, COBPAK and COBPAKM provide an easy method of handling variable length input and output data fields held on paper and magnetic tape. Users can deal with variable length fields without using the subroutines but this involves a great deal of explicit coding which is eliminated by using the subroutines. Information relating to the use of these subroutines is given under the following headings:

Introduction
Calling the Subroutines
Key formats
Error Reports

The explanation of the subroutines concludes with examples of programs using them.

Introduction

COBDIST and COBDISTM are used to unpack a string of variable length items demarcated by field-terminators and possibly interspersed with fixed length fields into a series of consecutive fixed length fields in store. In the process the field terminators are omitted; the items are justified and space-filled according to keys specified by the user which generally correspond to the description of the receiving fields given in the Data Division.

COBPAK and COBPAKM have the opposite function. They convert a string of fixed length input data into a string of variable length output data.

Note that although it is stated that the subroutines handle input and output data, it should be clear that they do not perform any peripheral transfers. Their function is to unpack data immediately after input and to pack data immediately before output.

The two unpacking subroutines, COBDIST and COBDISTM, are identical except that COBDIST, which is primarily intended for data input on paper tape, requires a newline character (upwards arrow, asterisk ↑*) to terminate a record of variable length data. COBDISTM is primarily intended for use with magnetic tape on which the size of the record is normally determined by a word count of the number of words in the record. Correspondingly COBPAK moves a newline character into the receiving area after the last field packed, whereas COBPAKM leaves the total number of words occupied by the packed record as an output parameter.

Other differences between the subroutines which are a consequence of this major difference are given in the detailed specifications of the subroutines.

Calling the subroutines

To call one of the subroutines the user writes:

```
ENTER PLAN  { COBDIST  }  USING identifier-1, identifier-2
            { COBDISTM }
            { COBPAK   }
            { COBPAKM  }

{identifier-3} {identifier-4}
{literal-1   } {literal-2   }  , identifier-5, [identifier-6]
```

The meaning of the parameters is as follows. Note that any data-name used as a parameter may be subscripted.

1 Identifier-1. This parameter is the name of the source area which should begin and end on a character boundary.

2 Identifier-2. This parameter is the name of the receiving area which should begin and end on a character boundary and must not overlap with the source area.

3 Identifier-3 or literal 1. This parameter gives the size of the receiving area, in characters for COBPAK, COBDIST and COBDISTM and in words for COBPAKM. It must be either the name of an unsigned DISPLAY field four characters in length (identifier-3) or a four character numeric literal (literal-1), for example 0009.

4 Identifier-4 or literal 2. This parameter provides information about the key specified by the user and in the form described below. It can be either the name of a character field (identifier-4) which contains the key information or an alphanumeric literal (literal-2) which is itself the key information.

5 Identifier-5. This is the name of a four character field used by the subroutines as a reply-word for information about errors in the data or keys as described on page 260. The field must begin on a word boundary.

6 Identifier-6. This parameter is the name of a four character field described as COMPUTATIONAL SYNCHRONIZED RIGHT (identifier-6). Its use is as follows for each of the subroutines:
(a) COBDIST does not use this parameter.
(b) COBDISTM uses it as an input parameter giving the number of words in the source area.
(c) COBPAK uses it as an output parameter giving the number of characters, including field-terminators and the newline characters in the output area.
(d) COBPAKM uses it as an output parameter giving the number of *words* in the output area. Since magnetic tape output has no block terminators any spare characters in the last word of the output area are zero-filled by the subroutine.

In the case of the input or output area not starting on a word boundary the word count starts from the word boundary immediately preceding the start of the input or output area. Similarly, the last word containing any data even if not completely filled is counted as a whole word.

Key formats

The keys which constitute the fourth parameter are a means of indicating to the subroutine the type and length of the variable and fixed length fields of a record. They consist of a string of characters whose meaning is described below and are terminated by an asterisk (*).

At the beginning of the string of characters is *the item or field-terminator* which the user is using to indicate the end of each item or field of input or output data. It consists of any single character or pair of characters except upwards arrow (↑) and the newline character (upwards arrow, askerisk ↑*) and is followed by an oblique stroke (/) which separates it from the other characters of the key.

The item—terminator is omitted by the unpacking subroutines on input and inserted by the packing subroutines on output.

Both unpacking subroutines will accept input records in which the number of item-terminators in the data is less than the number implied by the key-description. In this case they will assume that the missing item-terminators represent a series of zero or blank fields at the end of the record.

Conversely, both packing subroutines will omit item-terminators representing zero or blank fields at the end of a block.

COBDIST, but not COBDISTM, will accept data even if the last non-blank field of a record does not have an item-terminator, since it is always terminated by a newline character. COBPAK and COBPAKM, however, will always insert an item-terminator after the last non-blank field of a record.

The following items describe each of the fields in turn and begin with one of the following characters: A, F, N, I, D, R whose meanings are as follows. (The lower case characters n, m, j and k represent decimal characters.)

A nnn... indicates that the next field is alphanumeric and contains *nnn...* characters.

The unpacking subroutines will move data character by character from the next position of the string of input data to the next position of the receiving area until an item-terminator is met. If the number of characters transferred is less than *nnn...* and consequently the receiving area has not been filled, the subroutines will space-fill the remaining character positions of the field.

The packing subroutines will transfer up to *nnn...* characters to the output area, inserting an item-terminator after *nnn...* characters or after the last non-space character on the right if less than *nnn...* characters have been transferred.

F nnn... indicates that the next field is a fixed length field and does not have an item-terminator.

Both packing and unpacking subroutines will move nnn... characters from the input string to the output string.

N nn mm indicates that the next field is numeric with *nn* decimal places before the decimal point and *mm* places after it.

The packing subroutines deal with fields of this type as follows:

1 If there is no leading sign character: (For the purposes of these subroutines only + and - count as sign characters, i.e. overpunched signs, spaces and quote as in DISPLAY fields are not counted as sign characters.) The subroutines move across up to *nn* characters omitting all leading zeros or spaces; a decimal point, i.e. an explicit '.', is then inserted into the output areas; up to mm characters are then moved across after the decimal point, omitting all trailing zeros and, finally, the item terminator is inserted.

2 The action for signed fields (DISPLAY—3) is very similar except that the suppression of leading zeros begins after the sign character. In other words, the sign is floated up to the most significant non-zero or non-space character before the decimal point. It is not floated beyond the decimal point. The count nn includes the sign character.

The subroutine does not check that the characters in a field described by N are numeric. Hence, N can be used to describe fields with an overpunched sign.

EXAMPLES OF PACKING (The item terminator is assumed to be*)

Key	*Contents of source field*	*Contents of receiving area*
N 03 03	123456	123.456*
N 03 03	003400	3.4*
N 03 03	∇03400	3.4*
N 03 03	∇∇∇100	.1*
N 03 03	—∇0001	—.001*
N 03 03	∇∇∇000	*
N 03 03	123	123*
N 03 03	010	01*
N 03 00	—∇1	—1*
N 03 00	+∇∇	*

Notes:

(a) If on packing there are no characters left in the item except possibly a sign and a decimal point, these characters also are omitted.

(b) If the key is N 00 *mm* or N *nn* 00 an explicit '.' is not inserted.

D nn mm kk has the same meaning as I except that in this case the fractional part of the variable length item is considered to be fixed-length, kk characters long. The integral part is treated as for N. *kk* is normally equal to *mm*.

EXAMPLES (The item-terminator is assumed to be *)

Key	*Contents of source field*	*Contents of receiving area*
(a) Packing		
D 03 02 02	−∇010	−10*
D 03 02 02	+∇;00	+;00*
(b) Unpacking		
D 04 03 02	R00000*	R000000
D 04 03 02	+01*	+∇∇∇010
D 04 03 02	∇−111*	∇∇−1110

R nn mm indicates that the following *nn* keys are repeated *mm* times. For example, A4F5A4F5A4F5 is equivalent to R0203A4F5. Note that R may not control R.

Error reports

As mentioned earlier, identifier −5 in the parameter list is the word into which the subroutines put information about errors detected in the data or the keys. It must have the following description or some equivalent in the Data Division.

```
01 ERR−IND.
03 A PIC 9.
03 B PIC 9(3) COMP.
```

If on exit from the subroutine no error has been detected, A will be zero. If an error has been detected, however A will have one of the following values:

1 means that the receiving area has been filled before all the items in the source area have been packed or unpacked.

2 means that the key information is invalid.

3 means that on unpacking a packed data item contains more characters than allowed for by the corresponding key.

For example:	*Key*	*Item*
	N 02 02	.323*

This error occurs only with COBDIST and COBDISTM. It may also be flagged if the packed data item on D and I keys contains fewer than kk characters

For example:	*Key*	*Item*
	D 02 04 04	123*

4 means that on unpacking the end of the keys has been reached before the end of the packed source area.

For example:	*Key*	*Source area*
	↑U/A1A2*	Z↑UBB↑UA↑*

This error occurs only with COBDIST and COBDISTM.

The action of the unpacking subroutines is the [illegible] that of the packing subroutines. The subroutines examine the first character of the input item to see if it is a [illegible]. If it is, it is moved to the first character position of the receiving area. The remaining characters up to the decimal point (".") are space filled on the left to make a total of *nn* characters (or *nn* + 1 if there is a sign) and moved to the receiving area. The decimal point is omitted and the remaining characters up to the item terminator are moved to the next positions of the receiving area. If the number of characters is less than *nn*, the remaining receiving positions are zero filled.

EXAMPLES (The item-terminator is assumed [illegible]):

Key	*Contents of source field*	[illegible]
N 03 03	"	[illegible]
N 03 03	" 1."	[illegible]
N 03 03	".001"	[illegible]
N 03 03	V " ."	[illegible]
N 00 03	001*	00*

I nn mm kk has the same meaning as N except in the following respects:

(a) On packing, no explicit decimal point is moved into the output string. In addition, exactly *kk* characters of the data before the implied decimal point must be moved to the output string; even if some of the *kk* characters are leading zeros or spaces. *kk* must not be greater than *nn* and *kk* is normally equal to *nn*.

EXAMPLES (The item-terminator is assumed [illegible] to be "):

Key	*Contents of source field*	[illegible]
I 03 03 02	V22200	22"
I 03 03 02	VVV010	10"
I 03 03 02	23000	30"
I 03 03 03	23000	023"

(b) On unpacking, the action is identical to that for an N *nn mm* key except that, instead of searching for an explicit decimal point, the decimal point is assumed to be *kk* characters from the beginning of the item (i.e. the integral part of the item is considered to be fixed length *kk* characters long).

Key	*Contents of source field*	[illegible]
I 02 02 02	" V2"	[illegible]
I 03 03 03	V01"	[illegible]
I 03 03 02	V 23"	[illegible]

5 means that the receiving area has not been filled. This error occurs only with COBDIST and COBDISTM.

6 means that identifier—3, which must be numeric, contains an alphabetic character.

If the error is type 4 or 6, field C will be zero on exit from the subroutine. If the error is type 5, field C will contain the number of unfilled characters in the receiving field. If, however, the error is any other type, field C will contain the number of the key within the key area on which the error has been detected.

Example programs using the subroutines

COBPAK

Suppose an output area is described as

```
01 OUTPT SIZE 120.
```

This area is to contain a string of variable-length items each terminated by the characters ↑U. The items are to be packed after processing from a Working-Storage record with the following description:

```
01 WORK—STORE
02 NUM1 PICTURE 9(4)V9(3).
02 NUM2 PICTURE 9(7)V9(2).
02 ALPH1 PICTURE X(4)
02 COM—1 PICTURE 9(12) COMP SYNC RIGHT.
02 NUM3 PICTURE S9(4) DISPLAY—3.
02 NUM4 PICTURE S9(4).
```

The user could pack this sequence of fields into the area OUTPUT by writing the procedural statement:

```
ENTER PLAN COBPAK USING WORK—STORE OUTPT 0120
"↑U/N0403D070202A4F8N0500N0400*" ERROR—REPLY
PACKED—CHARS.
```

where ERROR—REPLY and PACKED—CHARS conform to the specifications for identifier—5 and identifier—6.

If the contents of the source fields were as follows:

```
NUM1    ▽▽99990
NUM2    001311050
ALPH1   AND▽
COM1    #0000000 00 05 00 44
NUM3    —0009
NUM4    J123(i.e. —1123)
```

The contents of OUTPT on exit from COBPAK would be
99.99↑U 1311 0.50↑U AND↑U 0000050D - 9↑UJ123↑U↑*

The contents of PACKED—CHARS would be 42 and of ERROR—REPLY zero. The remaining characters of OUTPT would be unchanged by the subroutine.

Note that in this case the size of the 'packed' output area is greater than that of the 'unpacked' input area; this is because only a small amount of suppression was possible.

The field COM1 is of particular interest in this example since it is not a character field, whereas the subroutine deals only with characters.

However, as is shown, any block of characters containing non-character fields can be dealt with by writing an F key, in which case nnn... is the total number of characters in the block.

Since the COBOL WRITE statement punches a fixed number of characters when applied to a record to be punched on paper tape, difficulty can be encountered when punching variable length records. This is because the character positions of the punch record area not filled by a record shorter than the record area are filled with unpredictable characters. To avoid this the user can fill his punch record area with run-out characters before moving the data into the area, but this is inconvenient as well as slow since run-out is a double character in store. A more satisfactory solution of the problem is to use the subroutine PUNCHN which will cause the exact number of characters in a record to be punched. It is called by the statement.

ENTER PLAN PUNCHN USING record-name field-name

in which the parameters have the following meaning:

Record-name is the name of a record of a file to be punched on paper tape.

Field-name is a single word integral field which is described as COMPUTATIONAL or COMPUTATIONAL-1 and SYNCHRONIZED RIGHT and whose value is the number of characters the user wishes to be punched.

The user should therefore follow the ENTER PLAN COBPAK statement in the example with the statement

ENTER PLAN PUNCHN USING OUTPT PACKED-CHARS

The following points should be noted when using the subroutine PUNCHN:

1 In general, COMPUTATIONAL and COMPUTATIONAL-1 fields should not be presented as input to PUNCHN, since it punches in 'normal mode'. In general, the contents of the data fields in the record-area should be restricted to the ALPHANUMERIC character set excluding $.(+-

2 The file to be punched must be OPENed and CLOSEd by COBOL statements.

3 PUNCHN will not punch more than 4095 characters, the maximum size permissible for a record on paper tape. If requested to punch more than 4095 characters, the subroutine will stop and display the message:

PUNCHN:- BLOCK GREATER THAN 4095 CHARS

If restarted, it will punch only the first 4095 characters and then exit.

If requested to punch more characters than the number of characters allocated to the file-buffer the subroutine will stop and display the message:

PUNCHN:- BLOCK GREATER THAN BUFFER SIZE

If restarted it will punch only the number of characters in the buffer.

COBDIST

The above example for COBPAK can be reversed to give an example of unpacking variable length fields using COBDIST. If OUTPT is regarded as the source area and WORK-STORE as the receiving area the ENTER function and its parameters are identical except that parameter 3 (0120) is replaced by 0037 and parameter 6 is omitted. Assuming OUTPT contains the same values as before, the contents of WORK STORE after calling COBDIST will be the same as the contents

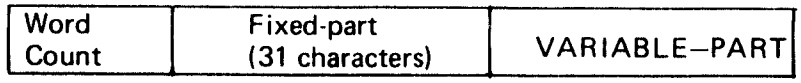
assumed for the COBPAK example except that the items described with an N key will have leading spaces wherever leading zeros were given in the previous example.

COBDISTM and COBPAKM

Detailed examples are unnecessary since the method of packing and unpacking is the same as for COBDIST and COBPAK with the exceptions mentioned. However, it will be useful to show how the subroutines fit in with the magnetic tape READ and WRITE instructions.

For example, suppose the user wished to read a magnetic tape record with variable length fields and distribute it entirely into a Working-Storage record.

If the input record had the following form:



the user would write

```
ENTER PLAN COBDISTM USING WORD–COUNT WORK–
STORE–RECORD (length of destination) "(item-terminator)/F35
(keys for variable part)" ERROR–REPLY WORD–COUNT.
```

WORD–COUNT is used twice, as the address of the beginning of the source area and as the count of the total number of words. If the user wished to distribute only the variable part he would replace the first WORD–COUNT with VARIABLE–PART and precede the ENTER statement with the statement:

```
SUBTRACT 8 FROM WORD–COUNT.
```

Similarly, if the fixed part of the record was already in the output record area and the user wished to condense WORK–STORE–RECORD into a variable length string in VARIABLE–PART and then write the complete record, he would write:

```
ENTER PLAN COBPAKM USING WORK–STORE–RECORD
VARIABLE–PART
(maximum length of VARIABLE–PART) (keys) ERROR–REPLY
WORD–COUNT
ADD 8 TO WORD–COUNT
WRITE MT–RECORD
```

Index