A S C - 8 C O B O L

for the PDP-8

by

Dr. Rick Jamieson
All Saints' College,
Bathurst 2795.

SECOND EDITION - JUNE 1978

CONTENTS

6. The CALL verb

The CALL verb has been implemented to allow program overlays and program chaining. However, the automatic generation of compiled program files has not as yet been implemented, so that programs to be CALLed must have been SAVEd using the OS/8 commands as described below:-

(i) PROCEDURE DIVISION overlays

Very long programs may have their PROCEDURE DIVISION segmented, and written as two (or more) programs with identical DATA DIVISIONs. The first segment may CALL the second, provided the compiled PROCEDURE DIVISION, and Literals for the latter have been saved on the system device. To SAVE the PROCEDURE DIVISION and Literals of a program named PROG proceed as follows:-

```
.R COBOL↓
*PROG↓
.SAVE SYS PROG 14000-16777,20000-25377,3xxxx-33777↓
```

To calculate the value to use for xxxx (the beginning of the Literals storage), find the top limit of the DATA DIVISION from the program listing, then take the next multiple of 400 (octal).

eg if the top of the DATA DIVISION is 561L, xxxx would be 1000. (Multiples of 400 octal are 400, 1000, 1400, 2000, 2400, 3000, 3400 etc)

To call this second segment of the program, the statement CALL PROG would be the last statement executed in the calling program.

(ii) Program Chaining

A program may CALL a completely different program (with a different DATA DIVISION and PROCEDURE DIVISION), if the compiled core image for the latter has been SAVEd on the system device. To compile and SAVE a complete program called PROG proceed as follows:-

```
.R COBOL↓
*PROG↓
.SAVE SYS PROG 12000-17577,20000-25377,30000-33777↓
```

To chain to this program, the statement CALL PROG is used as the last statement executed in the calling program.

ACKNOWLEDGMENT

The presentation of COBOL in this booklet is based on the COBOL standard developed by the American National Standards Institution (ANSI). In response to their request the following acknowledgment is reproduced in its entirety:-

Any organisation interested in reproducing the COBOL report and specifications 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 organisations are requested to reproduce this section as part of the introduction to the document. Those using a short passage, as in abbook review, are requested to mention "COBOL" in acknowledgment 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 organisation or group of organisations.

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. Inquiries 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 materials used herein: FLOWMATIC (Trademark of the Sperry Rand Corporation), Programming for the UNIVAC 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 authorised the use of this material in whole or in part, in the COBOL specifications. Such authorisation extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

INTRODUCTION

This booklet describes how to use the COBOL compiler and associated programs developed at ALL Saints' College Bathurst for the PDP-8 computer. Hardware requirements for this compiler are:-

Any PDP-8 family computer (except PDP-8/S)
16K words of memory (or more)
Console keyboard/printer or VDU
Magnetic disk or tape storage (eg floppy disks)

The software pre-requisite is the OS/8 Operating System and utility programs (Editor, ODT). Most input-output operations are performed via OS/8, so that all standard PDP-8 peripherals are supported (printers, cassettes, cartridge disks, DECtape, etc.).

This booklet describes only the single-user COBOL system. A multi-terminal version of the system is at present under development.

The compiler has been designed assuming that programs will be typed in directly onto disk or tape using one of the OS/8 editors, so that "card sequence" numbers (columns 1 to 6) and "continuation characters" (column 7) are not required. "Column A" (as defined on COBOL programming sheets) is taken to be the first character on the line and "Column B" may conveniently be taken as the first tab position (9th character). An option is available, however, to allow for sequence numbers - for instance when input is through a card reader.

Error messages are printed in a direct, rather than a coded form, and memory addresses printed on program listings allow for quick location of errors. We have found that beginners can type in short programs and compile, correct and run them in a matter of a few minutes.

SYSTEM OVERVIEW

To use the system, the first step is to write a Source Program. This consists of a sequence of instructions which the computer will be required to execute. Examples of COBOL instructions are:-

```
ADD 2 TO TOTAL.
IF STUDENT-NUMBER IS GREATER THAN 400, GO TO FINISH.
```

Source programs are typed into the computer and stored on a disk (or magnetic tape). When this has been done, a program known as the COBOL Compiler is called upon to translate these English-like statements into a series of coded numbers which can be understood by the electronics of the computer.

If the Compiler translates the program without finding any errors, another program known as the Run-Time System is called upon to actually perform the required operations. The Run-Time system includes program subroutines which can add, subtract, multiply and divide numbers, perform operations to file data on disks, etc. The Run-Time system uses the compiled coded numbers (which had been stored in the memory by the Compiler) to determine the locations of data and the sequence of operations which are to be carried out.

OPERATING THE COMPILER

The following example shows how to enter a short program into a disk (or tape) file, and then compile the program and run it.

It is assumed that a COBOL system disk (or tape) has been obtained; that the OS/8 Operating System has been started up in the normal way; and that the monitor dot has been printed on the console.

First, create a disk file called (say) PROG using the OS/8 Editor program, by typing:-

.CREATE PROG↓

- here the underlined symbol (.) has been printed by OS/8, and "↓" represents typing the RETURN key.

When the Editor's command sign # appears, show that you are going to add text into the memory buffer by typing:-

#A↓

Now type in the following simple program. Start the first three lines at the left-hand-side of the page, but indent the last line by typing the TAB key (or CNTRL/TAB if there is no separate TAB key), at the start of the line:-

```
FILE-CONTROL.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
        DISPLAY "HELLO".
```

(This program when executed will display the word HELLO on the console).

Note carefully the punctuation in this program - in particular:-

(i) The hyphen (but no spaces) in FILE-CONTROL.
(ii) The full-stop at the end of each line.
(iii) The spelling of WORKING-STORAGE SECTION and PROCEDURE DIVISION (a single space in each).
(iv) The RETURN character is used to end each line - including the last line.

Mistakes in typing can be fixed using the Editor commands described in the OS/8 Handbook.

Having entered the program, type CNTRL/L (hold down the CONTROL key while typing L), in order to get back the Editor's # sign. Then type:-

#E↓

to end the editing session. This will create the source program file PROG on the system disk.

Now run the COBOL compiler by typing:-

.R COBOL↓

A listing of the program, showing memory addresses, can now be obtained by typing:-

*,LPT:<PROG↓

(Note the comma). (If a separate printer is not available, use TTY: instead of LPT:). If the compiler detects any errors in the program, these will be indicated on the listing. Errors may be corrected using the Editor.

If desired instead, the errors only (no listing) may be obtained on the printer by typing:-

```
.R COBOL
*LPT:<PROG/E
```

If the program compiles without errors, it may then be compiled and run in the one operation by typing:-

```
.R COBOL
*PROG/G
```

Here, the "/G" tells the compiler to load the Run-Time system after compiling, and then execute the program.

Compilation or execution of a program can be interrupted at any time by typing CNTRL/C. The exception to this is when random-access (I-O) files have been written but not closed - then, typing CNTRL/C will result in the error message CONTROL/C NOT PERMITTED and the program execution will continue (this is necessary since updated disk blocks may remain in core memory until the file is CLOSEd).

Core-Image Program Files.

When a program has been successfully tested, it can be compiled and saved as a core-image (SV) file. To create a core-image file PROG.SV from a source program PROG type:-

```
.R COBOL
*PROG<PROG
```

Then to run the compiled program, just type:-

```
.R PROG
```

- this will load the core-image program and also the Run-Time system program COBRT.SV (which should be present on the system device), and start the program.

Overlays.

A PROCEDURE DIVISION overlay may be created by typing:-

```
.R COBOL
*PROG<PROG/O
```

- this will create a core-image file PROG.SV consisting only of the PROCEDURE DIVISION (and literals) of the program. This overlay may be CALLed from another program and will retain the data in the DATA DIVISION of the first program (c.f. CALL verb). Note that the /O option may require up to 512 characters of DATA DIVISION storage area in order to separate the literals from the data-items, for overlay purposes.

Combining Command Options.

The Command Decoder options described above may be combined as is usual with OS/8. For example:-

```
.R COBOL
*PROG,LPT:<RXA1:PROG/G
```

- will give a core-image output file and line-printer listing and will execute the program (if no errors are detected). The source program is on RXA1 (the right-hand floppy disk).

Listings.

When the command string to the compiler includes a listing device (which must be either LPT: or TTY:), the compiler prints a listing of the program with memory location numbers at the beginning of each line to assist in locating errors. These location numbers are 4-digit octal numbers representing memory word addresses. Locations 0000 to 0007 are not used, so that addresses start at 0010.

In the DATA DIVISION the location numbers have an "L" or "R" added to them to indicate whether the left- or right-hand byte (ie character) of the particular memory word is being referred to - for instance, the address of the first data byte is 0010L, and of the second data byte 0010R.

In the PROCEDURE DIVISION the location numbers give the address of the beginning of the compiled code for each line. For example:-

```
0010        MOVE A TO B.
0013        ADD B TO C.
```

here the line "MOVE A TO B" has been compiled into code which takes up locations 0010, 0011 and 0012. If during execution the error message:-

```
RUN-TIME ERROR:-
DATA NOT NUMERIC AT PROCEDURE LOCN. 0012
```

- were obtained, this would indicate an error occurring during the MOVE statement above.

FILES

COBOL is a "Data-Processing" language - that is, it is used to process data. The data is normally stored on disks or magnetic tape. Each major unit of data on a disk or tape is known as a file - for instance, a school address-list file may contain the name and address of every person at a school. Each file on a disk (or blocked magnetic tape) has a File-Name so that it can be distinguished from other files on the disk. A list of File-Names and details of the area of disk which each occupies is kept in a directory which is also stored on the disk.

Files are divided into records. Each record normally contains all the information relating to a particular class of data in the file - eg a record in a school address-list file would contain the name and address of a particular person; a record in a debtors' file may consist of details of debts owing from a particular company, etc.

Each record in a file is terminated by an end-of-record mark. The end of a file is indicated by an end-of-file mark, which consists of two consecutive end-of-record marks.

File Devices.

This system performs all input-output operations (except for console operations) using the standard OS/8 device handlers. The Run-Time system has available four PDP-8 memory pages for use by device handlers (as well as the system device handler, which is always resident). The system does not support system devices with two-page handlers (eg TD8E dectape without ROM). However, devices which are not the system device may have either one or two-page handlers. Note also that parity terminals and terminals generating 7-bit ASCII codes are supported.

The system supports two main types of file devices:-

(i) File-structured devices.

These devices (eg disk or DECtape) have the property that they may be randomly addressed - that is, a particular block of the device may be read, without having to start at the beginning of the device. Such devices may hold more than one file, and they always contain a directory which stores the names of the files and their locations on the device. File-structured devices support both sequential and random-access files. Data on these devices is stored in 6-bit code - two characters per PDP-8 word. OS/8 limits the length of files on these devices to just over one million words (two million COBOL characters). Apart from this restriction, there is no restriction on the number of COBOL records allowed in any file.

(ii) Non-file-structured devices.

These devices (eg 7-or 9-track Mag-tape, cassettes, paper-tape, line-printers, terminals) may not be randomly addressed. Hence they support only sequential files. Data is stored on them in 8-bit ASCII in standard OS/8 format. The length of files on these devices is limited only by the amount of storage medium available.

Types of Files.

This system supports two main types of file organisation:-

(i) Sequential files.

Sequential files are written to, or read from the device one record after another, starting at the first record in the file. In a COBOL program, a WRITE statement applied to a sequential file transfers one record from the computer memory onto the end of whatever has been previously written onto that file. Similarly, a sequence of READ statements applied to a sequential file will read consecutive records from the file into the computer memory.

There are two types of sequential files:-

(a) Those with fixed-length records - ie each and every record in the file is the same length

(b) Those with variable-length records - ie the records need not be all the same length. For example, an address-list file may contain records having two, three, four or five lines for each address - depending on how many are needed for each. In this way wasted space in a file can be minimised. When reading a file with variable-length records, the Run-Time system determines the length of each record by noting the position of the end-of-record marks.

(ii) Random-access files.

Random-access files allow reading or writing of a particular record in a file, without needing to start right at the beginning of the file. Random-access files must have fixed-length records, so that the Run-Time system can compute where a particular record is in the file. In order to do this it must be supplied with a "record number" (eg to be told whether the 1st, 3rd, 100th etc record is required). This "record number" is given by the value of a data-item known as the ACTUAL KEY for the file.

Random-access files speed up the location of particular records if ...se are required in a random order. For instance in a student-account file the status of J. SMITH'S account could be found very quickly by specifying his "student number" (say 234), so that the computer can locate the 234th record in the file directly. Note that random-access files are normally created, in the first place, as fixed-length-record sequntial files.

Also note that it is nearly always best to store any type of file with the records in a certain order - eg in name alphabetical order, or in order of "student number" etc. Features of the COBOL language make this quite easy to do. Also, if it is desired to change the order of records in a file (eg from alphabetical to number order), the sort program CSORT (c.f) may be used.

Indexed-Sequential Files.

The processing of random-access files by this system has been made ...ible and efficient enough to permit the implementation of some Indexed-Sequential file structures.

When the access key for a random-access file (eg student number, policy number, debtor number, etc) ranges from 1 upwards consecutively (eg 1 to 1000), then individual records can be accessed directly by specifying the value of the ACTUAL KEY. However, when the access key is not distributed so simply - eg when the access key consists of peoples' names - it is common practice to use some type of Indexed-Sequential file structure if it is desired to access records in a random manner.

Problems commonly encountered with Indexed-Sequential files include:-

(i) either the index, when held in core, takes up too much space (no room for programs), or

(ii) if the index is kept small then the average time required to access a particular record is too long, or

(iii) if most of the index (or sub-levels of index) are kept on disk then access times also tend to be too long;

(iv) systems are often inflexible with regard to "stretching out" parts of the file (where access keys are crowded);

(v)

(v) programmers often have little control over the file structure, and find difficulty optimising storage space in a particular file;

(vi) re-structuring a file is often difficult - for example when a file expands beyond its designed capacity.

This system allows implementation of Indexed-Sequential files in a straightforward way, while minimising most of these problems. The technique suggested here employs:-

(i) a main index which is small enough to be stored within the DATA DIVISION of a program;

(ii) a cross-reference file with records that are small enough that they can be shuffled around to a reasonable extent within a single disk block of 512 characters; and

(iii) a master-file which is a random-access file.

As an example, consider the following arrangement for a file where the access keys are to consist of peoples' names:-

index stored in DATA DIVISION		name-to-number cross-reference file			master file		
name	record no. in cross-ref file	record number	name	file number	record number	name	data
A	1	1	AARDVARK A.	632	1	SMITH P.	$25
B	25	2	ABLE B.C.	257	2	BAKER B.	$71
C	41		(empty records)		3	FRANKS M	$62
(etc.)		25	BADCOX M.	749		(etc.)	
		26	BAKER B.	2			
			(empty records)				
			(etc.)				

Here, each person is allocated a file number, which is equal to his record-number (ACTUAL KEY value) in the Master File. If his number is known, his record in the Master File can be accessed directly.

To locate a person's file number from his name, the cross-reference file is utilised. But in order to save looking through the whole of the cross-reference file to find Mr. Zeback (for example), the DATA DIVISION index is utilised, which shows the position in the cross-reference file of names starting with each letter of the alphabet. The program then just needs to search through a fairly small part of the cross-reference file in order to find the required name. The latter operation is efficient and fast so long as the records in the cross-reference file are small compared with the OS/8 block length. (Of course, the index could have been made longer and more accurate than shown here).

To insert a new name into the cross-reference file, the program just needs to begin at the correct letter; bi-pass the records before the new one; insert the new one; and shuffle the remainder on in the file until an empty record is located.

Again, such operations on cross-reference files have been speeded up in this system by means of efficient processing of random-access files - viz. a disk (or tape) block is not written to the device with each WRITE statement, but the data is retained in core in case further transactions involving that block of 512 characters are required. The READ statement operates in a similar way.

Provided records in the cross-reference file are kept small (eg 20-50 characters) and the accuracy of the DATA DIVISION index is adequate (eg one entry for approximately one OS/8 block of the cross-reference file), the accessing of indexed files has been found to be very fast - even with floppy disks (typical records can be located with only two or three disk accesses).

Simple COBOL programs to update or retrieve data from indexed files require only 20 to 50 extra statements, and the programmer has complete control over the structure of the index and the distribution of the files. Expanding or contracting a file, or changing the distribution of the keys, can be readily done by designing a new index and writing a short conversion program.

The DATA DIVISION index may conveniently be set up using a subscripted Data-Name, and the VALUE clause. A PERFORM...VARYING verb can then be used to search through the index.

THE COBOL LANGUAGE

All COBOL compilers are different. This compiler is highly compatible with most ANSI COBOL compilers found on smaller commercial computers. It lacks some of the more sophisticated features found on compilers on very large machines, but has some features which may not be found even on large machines (eg listings showing data and procedure addresses; flexible facilities for processing random-access files and files with variable-length records; powerful error-locating facilities; simple interactive operation). The compiler capacities, and a list of features which have not been implemented, are set out at the end of this booklet.

Character Set.

Source programs may consist of the following symbols:-

0 1 2 3 4 5 6 7 8 9 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
+ - $. , ; ( ) * and also space (or blank - written b in the examples). Tab characters may represent multiple spaces.

In addition, alphanumeric data-items, and literals, may contain the following additional symbols:-

[" ' : = ? square brackets, number sign, percent sign, &, /, back slash, less than, greater than, and circumflex (or up-arrow). The underline (or back-arrow) symbol is excluded as its code is used as the end-of-record and end-of-file character.

Source Program Format.

Division, Section and Paragraph-names must start at the first character position of a line. PROCEDURE DIVISION statements may not start at the first character position - they must be preceded by space(s) or, more conveniently, by tab characters. The compiler replaced each leading tab character by 8 spaces.

Divisions and Sections.

COBOL programs are divided into certain standard Divisions and Sections. This compiler requires three headings to be present even if there are no entries in the respective sections:-

```
FILE-CONTROL.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
```

However, it is advisable to include some program-identification and description at the beginning of a program, and a recommended format is shown in this example:-

```
PROGRAM-ID.
        ADDTWO.
REMARKS.
        THIS PROGRAM ACCEPTS A NUMBER FROM THE CONSOLE,
        ADDS TWO AND DISPLAYS THE RESULT.
AUTHOR.
        J. DOE,  ALL SAINTS' COLLEGE,  MAY 1978.
FILE-CONTROL.
        (File SELECT statements may appear here).
DATA DIVISION.
FILE SECTION.
        (File Descriptions may appear here).
WORKING-STORAGE SECTION.
        77AA    PICTURE 999.
PROCEDURE DIVISION.
        ACCEPT A.
        ADD 2 TO A.
        DISPLAY A.
```

Note that the two main Divisions are the DATA DIVISION and the PROCEDURE DIVISION. The DATA DIVISION consists of two Sections - the FILE SECTION and the WORKING-STORAGE SECTION.

Comment Lines.

Lines commencing with an asterisk (*) in the first (left-hand) print position are not processed by the compiler, except for being listed where applicable. Example:-

```
*PROCESSING FOR END-OF-MONTH:-
```

Data-Names.

Names such as "A" in the statement "ADD 2 TO A" are called Data-Names, since they specify areas of memory where data is stored. Data-Names must:-

- begin with a letter (A to Z)
- include only letters, numbers and hyphens
- not include any spaces
- not consist of COBOL Reserved Words (c.f.). Where a data item will not be referred to by name, it may be given the dummy name FILLER.

Examples of Data-Names:-

A, TOTAL, CURRENT-ACOUNT, TOTAL-FOR-GROUP-2.

Literals.

Data constants defined in the PROCEDURE DIVISION (such as "2" in the above program example) are known as Literals. There are two types of literals:-

(i) Numeric Literals.

These consist of numeric digits with an optional decimal point and an optional minus sign. There may be up to 10 digits before the decimal point, and up to 6 after.

Examples:- 2, 12345.6, -50.234

```
ADD 2 TO A.
```

(ii) Alphanumeric Literals.

An alphanumeric literal consists of a string of any of the Legal Characters (c.f.). Such literals may not be used for performing arithmetic operations, and may not be longer than 150 characters. They must be enclosed either in single or double quotation marks (' or "). Whichever quotation mark is not used to define the literal may appear inside the literal. Long alphanumeric literals may be written on two lines, the two parts being separated only by the carriage-return character.

Examples:- "SMITH", "SMITH'S", 'THE LETTER "R"', "2"

MOVE "THE QUICK BRO
WN DOG" TO TITLE.

Figurative Constants.

Certain constants are frequently used, and have special names which need not be defined as Data-Names:-

ZERO, ZEROS or ZEROES.
SPACE or SPACES.
HIGH-VALUE or HIGH-VALUES. This constant consists of characters with an internal computer code (77 octal) which is greater than that of any other character. A data-item consisting of these characters would appear last in an alphabetically-ordered list.

LOW-VALUE or LOW-VALUES. Null or @ characters (code 00).

Example:- MOVE SPACES TO NAME, MOVE ZERO TO A.

The CURRENT-DATE Data-Field.

Due to difficulties with OS/8, the CURRENT-DATE data-field has not been implemented on this compiler. If it is required for programs to refer to the CURRENT-DATE, it is suggested that a short file containing the date be created, updated each morning and read by those programs requiring it.

File-Names.

Each file in a program (including print files) must be given a File-Name in OS/8 format, which is as follows:-

- the first character must be a letter (A to Z)
- the only characters allowed are letters and numbers (0 to 9)
- there may be up to 6 characters for the main name, followed (optionally) by a full stop and a 2-character extension.

Examples:- DATA2 COSTS.DA PRFILE

READ COSTS.DA, AT END GO TO FINISH.

Word Separators.

Spaces are very important in COBOL programs. All words and other items must be separated by blank(s), tab or the return (end-of-line) character. Commas and semi-colons may be inserted additionally to improve readability.

Examples:- OPEN INPUT FILE1, FILE2.

- the comma here is optional, but the space (or tab) following the comma is required.

ADD 34.10 TO TOTAL (2).

- here the space following TOTAL (and before the subscript) is necessary.

THE PARTS OF A COBOL PROGRAM

In the following descriptions of the clauses and statements which can make up a program, the following conventions are used:-

(i) COBOL Reserved Words are printed in capital letters.

(ii) Underlined Reserved Words may not be omitted in a particular clause.

(iii) Reserved Words not underlined may be omitted, if desired.

(iv) Square brackets are used to enclose optional clauses.

(v) Curly brackets are used where there is a choice of words or phrases.

Example:-

READ File-Name RECORD [{ AT END / INVALID KEY } any statement(s)]

- here the COBOL Reserved Words are in capitals; the words "RECORD", "AT" and "KEY" may be omitted; the clause in square brackets is optional; when this clause is included, there is a choice of either "AT END" or "INVALID KEY".

Statements may take up more than one line. But

FILE-CONTROL.

This is normally the first part of a program, after the program identification and remarks. Its purpose is to define the File-Names and the types of files, and to specify the devices (disks, printer etc) which are to be involved.

The FILE-CONTROL consists of one SELECT sentence for each file. The format of the SELECT sentence is:-

SELECT File-Name [ASSIGN TO Hardware-Name]
[ACCESS MODE IS { SEQUENTIAL / RANDOM }]
[ACTUAL KEY IS Data-Name].

Notes:-

(i) The File-Name (c.f.) must be in OS/8 format.

(ii) The Hardware-Name must be an OS/8 device-name such as SYS, LPT, RXA1, RKA1, DTA1 etc.

(iii) If the ASSIGN TO clause is omitted, the system device SYS is assumed to be the Hardware-Name.

(iv) If the ACCESS MODE is not defined, SEQUENTIAL is assumed.

(v) The ACTUAL KEY clause is used for random-access files only. The Data-Name (key) must be defined in the DATA DIVISION.

(vi) A period is required at the end of a SELECT sentence.

(vii) Up to 7 files may be SELECTED in any program.

Examples:-

```
SELECT DFILE.DA.
```

- selects a sequential file called DFILE.DA on the system device.

```
SELECT PRFILE ASSIGN TO LPT.
```

- selects a file called PRFILE which will be printed on the printer.

```
SELECT NFILE.SC, ASSIGN TO RXA1;
        ACCESS MODE IS RANDOM,
        ACTUAL KEY IS NFILE-KEY.
```

- selects a file called NFILE.SC which will be on RXA1 (the right-hand floppy disk); it is a random-access file whose key (ie the number of the record which is to be accessed) is specified by the contents of the Data-Name NFILE-KEY.

FILE SECTION.

This is the first Section of the DATA DIVISION, and is where the layout for the records for each file is specified.

The description of the record layout for each file begins with an FD (File-Description) sentence in the format:-

```
FD File-Name [ RECORDING MODE IS { F } ]
             [                   { V } ]
  [ BLOCK CONTAINS  integer-1 TO  int-2 { RECORDS    } ]
  [                                     { CHARACTERS } ]
  [ RECORD CONTAINS  int-3 TO  int-4 CHARACTERS ]
  [ LABEL { RECORD IS  } { STANDARD } ]
  [       { RECORDS ARE} { OMITTED  } ]
  [ DATA { RECORD IS  }  Data-Name-1  [, Data-Name-2, ....] ]
  [      { RECORDS ARE}                                     ]
```

Notes:-

(i) The File-Name must be the same as that previously specified in an ASSIGN sentence in the FILE-CONTROL paragraph.

(ii) If the RECORDING MODE clause is omitted, the mode is assumed to be F (fixed-length records). When this clause is included, it must be the first clause following the File-Name.

(iii) Files with variable-length records are often convenient for non-file-structured devices (eg printer, where different line formats may not have the same number of characters).

(iv) Apart from the RECORDING MODE clause, all other clauses may be omitted with this compiler. Blocks are always 256 PDP-8 words (the OS/8 standard block length); the lengths and names of the records are always obtained from the 01-level Record Descriptions following the FD statement; and LABEL RECORDS are not used except where organised by the program. If the BLOCK, RECORD, LABEL or DATA RECORD clauses are included in a program, they are regarded by the compiler as comments only (ie for program documentation purposes).

Examples:- FD NFILE.SC.

- here the file NFILE.SC is assumed to contain fixed-length records.

FD PRFILE, RECORDING MODE IS V.

- here the file PRFILE is to have variable-length records.

THE RECORD DESCRIPTION

Following the FD statement comes the layout for the records in the file. Each record normally consists of a number of items of data, each referred to by means of a Data-Name. The Data-Names are arranged in groups - the group organisation being specified by means of a Level-Number for each Data-Name. The Data-Name following the FD statement is called the Record-Name. The level number for the Record-Name is always 01. A record designed to store a person's name and address could be set out as f follows:-

```
01 DFILE-RECORD.
       02 D-NAME.
              04 D-SURNAME      PICTURE X(20).
              04 D-INITIALS     PICTURE XXXX.
       02 D-ADDRESS.
              04 D-FIRST-LINE   PICTURE X(20).
              04 D-SECOND-LINE  PICTURE X(20).
              04 D-THIRD-LINE   PICTURE X(20).
```

Here the 01-level name DFILE-RECORD is the Record-Name. Note that this would be a different name from the File-Name (each file may have more than one Record-Name). The Record is broken up into two 02-level data-items, D-NAME and D-ADDRESS. Both of these names are in fact called Group-Names, since they are further broken down into 04-level data-items (we could have used 03 level here, or in fact any level up to the maximum of 49). These 04-level Data-Names could also have been Group-Names, containing other items with still higher level-numbers. On the other hand, a complete record could consist of just a single Data-Name, the Record-Name, at 01 level.

DATA-NAME DESCRIPTION CLAUSES.

Following the level-number and Data-Name may come one or more of the clauses:-

BLANK WHEN ZERO	PICTURE
VALUE	REDEFINES
OCCURS	JUSTIFIED

If present, the REDEFINES clause must come before any others. BLANK WHEN ZERO must immediately precede the PICTURE clause. The other clauses may appear in any order.

BLANK WHEN ZERO clause.

Format:- [BLANK WHEN ZERO] PICTURE .....

When present, this clause must immediately precede the PICTURE clause. This clause changes the data to blanks when the value of a numeric item placed in it is equal to zero.

Example:-

```
04 TOTAL    BLANK WHEN ZERO    PICTURE 99999.
```

The BLANK WHEN ZERO clause may only be used with Edited items (c.f.). So the presence of the clause in the above example implies that the data item TOTAL is an Edited item, and so may not be used in arithmetic operations.

PICTURE clause (may be abbreviated PIC)

This clause describes the type of data and the number of characters in the data item.

There are three types of data items:-

(i) Alphanumeric Data Items ("strings").

These items may contain strings of any of the legal characters. They may not be used in arithmetic operations. Each character in the item is represented in the PICTURE by a letter X. Multiple characters may be indicated by a number in brackets immediately following the X. These items may be of any length.

Examples:-

```
04 D-SURNAME          PICTURE X(20).
```

- this item consists of 20 alphanumeric characters.

```
04 D-INITIALS         PICTURE XXXX.
```

- alternately, this PICTURE could be written X(4).

(ii) Numeric Data Items.

Numeric items may contain numbers only. Numeric digits are represented in the PICTURE by the number 9. Brackets may be used to indicate multiple 9's.

Example:- 04 STUDENT-NUMBER PICTURE 9(4).

- represents a 4-digit number (some digits of which may at any time be zero).

The position of the decimal point may be indicated in the PICTURE of a numeric data item by the letter V. Although the COBOL system remembers this information, the "V" is not stored within the data item, and it does not require a data character for its storage. Where the position of the decimal point would lie outside the range of the digits stored, the symbol P is used to denote the missing digits. This system allows up to 10 digits before the decimal point, and up to 6 after (arithmetic operations are performed to an accuracy of 16 decimal digits).

Examples:- 04 ITEM-COST PICTURE 9(5)V99.

- this item has a total data storage of 7 characters, 5 being before and 2 after the position of the decimal point.

04 GROSS-INCOME PICTURE 9(6)PPPV.

- this item has only 6 numeric digits stored, but the least significant digit represents "thousands".

When the sign of a numeric data item is important, the letter S ("signed") is written at the beginning of the PICTURE. When S is omitted, the system always takes whatever number is contained in the item to be positive. Note also that S does not take up a character of storage.

Example:- 02 FINAL-BALANCE PICTURE S9(6)V99.

- this is an 8-digit signed item, with two digits coming after the position of the decimal point.

(iii) Edited Data Items.

Numeric data items are normally not in a suitable format for printing. For example, an item with PICTURE 9(6)V99 and containing the actual value 2.5 would be printed (or DISPLAYed on the console) in the form "00000250". To edit such a data item, it may be MOVEd to a data item having an Edited PICTURE, before being printed. Special symbols, beside the symbol 9, which may be used in Edited PICTUREs are set out below. Note that every symbol in an Edited PICTURE must be counted when calculating the number of characters of storage which that item occupies.

Z. "Suppress a leading zero". A Z written in place of a 9 changes a leading zero to a blank. Example:- The number 2 MOVEd to a data item with:-

PICTURE Z999 would become b002
PICTURE ZZZZ would become bbb2

- where "b" denotes a blank (or space).

+ (plus sign). A plus sign written at the beginning or end of an Edited PICTURE results in either:-

(i) a plus sign being placed in that position if the data moved to the item is positive (or unsigned), or

(ii) a minus sign being placed in that position if the data is negative.

Multiple plus signs at the beginning of a PICTURE have the effect of replacing leading zeroes by blanks, except that a single plus or minus sign is placed in the position of the last leading zero.

Example:- -2 MOVEd to a PICTURE +++9 would become bb-2.

- (minus sign). A minus sign at the beginning or end of a PICTURE results in either:-

(i) a minus sign being placed in this position if theddata is negative, or

(ii) a space being placed in this position if the data is positive, or unsigned.

Multiple leading minus signs give a "floating" minus sign, as for the plus sign.

$ (dollar sign). Dollar signs written at the left of a PICTURE result in a single dollar sign being placed in the position of the right-most dollar sign which replaces a leading zero. Example:- The number 2 MOVEd to a:-

PICTURE $99 would become $02
PICTURE $$9 would become b$2

* (asterisk) ("check character"). Asterisks replace leading zeroes. Example:- 23 MOVEd to a PICTURE **9 would become **23.

. (decimal point). A number MOVEd to an Edited PICTURE which contains a decimal point will be aligned according to the position of the decimal point, and the decimal point will remain in its position.

Example:- A data item containing the digits 1234 in a PICTURE 99V99 when MOVEd to an item with PICTURE ZZZ.999 would become bb2.340.

, (comma). A comma will be inserted in the Edited item at the position indicated by the comma in the PICTURE, unless all the digits to the left of the comma have been zero-suppressed. Example:- The number 1234 MOVEd to

PICTURE ZZZ,999 would become bb1,234
PICTURE ZZZ,ZZZ,999 would become bbbbbb1,234
PICTURE **,***,999 would become ****1,234

0 (zero). A character 0 will be inserted in the Edited data at the indicated position. Example:- The number 123 MOVEd to a

PICTURE 999000 would become 123000

B. The character B in an Edited PICTURE will result in a blank being inserted in the edited data at that point.

Example:- The number 1234 MOVEd to a PICTURE ZZZBZZ9 would become bb1b234

CR and DB (credit and debit). The letter-pairs CR or DB, if present, must appear last in an Edited PICTURE. If the number MOVEd to the item is negative, then the two letters (either CR or DB) will appear in that position. If the number is positive or unsigned, the two letters will be replaced by two blanks. Example:- -4.20 MOVEd to a

PICTURE 99.99DB would become 04.20DB

VALUE CLAUSE.

Data-items may be given an initial VALUE (before the program commences execution) by means of the VALUE clause. Alphanumeric data items must have an alphanumeric VALUE (in quotes), and numeric data items must have a numeric VALUE. (Edited or group items may no be given an initial VALUE). Also, the VALUE which is given to an item must be exactly the same length and format as the data-item's PICTURE (or else the VALUE may be a Figurative Constant).

Examples:-

```
04 ITEM-PRICE      PICTURE 999V99    VALUE 001.50.
04 ITEM-NAME       PICTURE X(6)      VALUE " NAILS".
04 UNIT-PRICE      PICTURE 999V99    VALUE ZERO.
04 INITIALS        PICTURE XX        VALUE SPACES.
```

This compiler allows a VALUE to be specified in the FILE-SECTION of the DATA DIVISION as well as in the WORKING-STORAGE SECTION. Note that numeric items which are not given an initial VALUE may not be used in arithmetic operations until a numeric data item is MOVEd to them.

This compiler does not offer the "condition-name" (level 88) VALUE tion.

REDEFINES clause.

This clause is used to indicate that an area of data storage is to be referred to by means of two (or more) different names. This clause, when present, must immediately follow the Data-Name. The level-number of the REDEFINES data item must be equal to the level-number of the data item that is being REDEFINEd, and the redefinition must be made at the first opportunity in the DATA DIVISION (ie, other data items may not be inserted before the REDEFINES item).

Example:- If a data item called STUDENT-NUMBER is n some cases to contain numeric data (for use in arithmetic statements), but in other cases to contain alphanumeric information, it could be defined twice:-

```
04 STUDENT-NO-NUM     PIC 9999.
04 STUDENT-NO-ALPH    REDEFINES STUDENT-NO-NUM     PIC XXXX.
```

Then the PROCEDURE DIVISION could contain statements such as:-

```
ADD 2 TO STUDENT-NO-NUM          (numeric data item)
MOVE SPACES TO STUDENT-NO-ALPH   (alphanumeric data item)
```

- even though both these statements refer to the same four characters in the memory.

Notes:-

(i) Some items in a REDEFINES data-group may not need to be referenced by a Data-Name. In such cases the Data-Name may be replaced by the word FILLER (see example following).

(ii) This compiler allows REDEFINEd items to be given a VALUE. But if the REDEFINing item also has a VALUE, then the previous VALUE would be over-written by the second one.

(iii) A REDEFINES clause is not permitted with a data item which is within a group-name that has an OCCURS clause. For example:-

```
01 WEEK  OCCURS 4 TIMES.
     02 WEEK-A     PIC XX.
     02 WEEK-B     REDEFINES WEEK-A    PIC 99.
```

- is illegal, and would give a compiler error message. Note, however, that the fundamental (01-level) group-name may be REDEFINEd as many times as is desired.

(iv) A REDEFINES clause is not legal when associated with a data item that is already within a REDEFINES group-name. For example:-

```
01 A      PIC X(6).
01 B  REDEFINES A.
     02 C          PIC 999.
     02 D  REDEFINES C    PIC XXX.
```

- is illegal, and would give a compiler error message. Note, however, that data items (that are not within a REDEFINES group) may be REDEFINEd in as many different ways as is desired.

Redefining Record-Names.

It is often desirable for different records in a file to have different data layouts. This is essential, for example, in a variable-length-record file. In such cases, the different record layouts must follow one another consecutively. Each Record-Name will have an 01-level number, and the different record layouts will effectively REDEFINE one another (ie their data items will be stored in a common data area)

The following record layout is for a variable-length-record file called ADDR. The Record-Names ADDR-2 and ADDR-3 have respectively 2 and 3 lines in their addresses:- The item called NO-OF-ADDR-LINES would contain the number 2 or 3, indicating the number of address lines in each record of the file. Such information would be required by a program which reads the file. Also note that the 02-level items in the second record layout need not be re-named, so that these items are simply given the name FILLER.

```
FD ADDR RECORDING MODE IS V.
01 ADDR-2.
       02 NO-OF-ADDR-LINES          PIC 9.
       02 NAME                      PIC X(20).
       02 ADDRESS-LINE-1            PIC X(20).
       02 ADDRESS-LINE-2            PIC X(20).
01 ADDR-3.
       02 FILLER                    PIC X(61).
       02 ADDRESS-LINE-3            PIC X(20).
```

OCCURS clause.

Data-Names may be subscripted - for example, instead of referring to HOURS-WORKED-1, HOURS-WORKED-2 etc for 5 different days of the week, we could define:-

```
04 HOURS-WORKED   OCCURS 5 TIMES   PICTURE 99.
```

This would set aside 5 different data areas for 5 different data items, and would allow the PROCEDURE DIVISION to refer to HOURS-WORKED (1), HOURS-WORKED (2) etc (the numberin brackets is called a "subscript"). The value of a subscript may be specified by means of a Data-Name as well as by a literal - eg:-

```
MOVE 5 TO DAY.
DISPLAY HOURS-WORKED (DAY).
```

The OCCURS clause may also be specified for group names. For example:-

```
02 HOURS-DETAILS   OCCURS 5 TIMES.
       04 HOURS-WORKED          PICTURE 99.
       04 OVERTIME              PICTURE 99.
```

- in this case the PROCEDURE DIVISION could refer to HOURS-WORKED (1) etc; OVERTIME (1) etc; or to one of the group items: HOURS-DETAILS (1) etc.

Up to three subscripts are allowed for a particular data item. For example, the DATA DIVISION could specify:-

```
02 HOURS-WORKED-IN-WEEK        OCCURS 4 TIMES.
       04 HOURS-WORKED-IN-DAY  OCCURS 5 TIMES   PIC 99.
```

- and the compiler would allocate room for 20 items of 2 characters each. The Data-Name HOURS-WORKED-IN-DAY (3, 5) (for example) would then refer to the hours worked in the 3rd week on the 5th day.

This compiler allows data items with different numbers of subscripts to be defined at the same level within the same group item. However, those data items with the greater numbers of subscripts must be written after those with less subscripts. Hence:-

```
01 WEEK   OCCURS 4 TIMES.
       02 WEEKEND          PIC 999.
       02 WEEK-DAY  OCCURS 5 TIMES     PIC 999.
```

- is legal (here WEEK and WEEKEND would have one subscript, WEEK-DAY two). But not:-

```
01 WEEK   OCCURS 4 TIMES.
       02 WEEK-DAY  OCCURS 5 TIMES     PIC 999.
       02 WEEKEND          PIC 999.
```

JUSTIFIED clause.

If the literal "ABC" were MOVEd to a data item with PICTURE X(6), the resulting data in the item would be ABCbbb. If the numeric literal 123 were MOVEd to a data item with PICTURE 9(6), the resulting data would be 000123. In other words, alphanumeric items are normally JUSTIFIED LEFT and blank-filled; numeric items are always JUSTIFIED RIGHT and zero-filled. If desired, alphanumeric items can be defined to be JUSTIFIED RIGHT. For example:-

```
04 SURNAME    PIC X(20)    JUSTIFIED RIGHT.
```

USAGE and SYNCHRONIZED clauses.

Unlike some COBOL compilers, this compiler has the simplifying advantage that all data within memory and on disk files is stored as "6-bit" code - eg the letter "a" is stored as octal 01 and the number 1 as octal 61. Hence there is no requirement for either USAGE or SYNCHRONIZED clauses.

WORKING-STORAGE SECTION.

Data items which do not form part of a file record must be placed in the WORKING-STORAGE SECTION of the DATA DIVISION.

The layout of the data in the WORKING-STORAGE SECTION is identical to that in the FILE SECTION, except that independent items (items which are not group-names and are not part of a group) should be given the level-number 77. Example:-

```
77 ACCOUNT-TOTAL      PICTURE 99999V99.
```

PROCEDURE DIVISION

The PROCEDURE DIVISION is where the procedural part of the program is written - ie the sequence of instructions which define what the program is to do. The PROCEDURE DIVISION is made up of Sections, Paragraphs, Sentences and Statements.

A Statement is the basic unit of the PROCEDURE DIVISION. It is a single instruction to perform a single operation (or a small group of similar operations). Examples of statements are:-

```
ADD 2 TO TOTAL
GO TO START
```

Each word of a statement must be separated from the other words by at least one space or tab character.

A Sentence consists of either a single statement, or else a group of statements, terminated by a period (.). Sentences should be indented by a tab spacing from the left-hand-side of the page, and they may extend for any number of lines. However, it is good practice to restrict sentences to only one or two statements - this simplifies error detection and correction. Statements within a sentence may end with a comma (,) semicolon (;) or the word THEN. Examples of sentences:-

```
ADD 2 TO TOTAL; SUBTRACT 2.60 FROM COST.
IF A = B, THEN ADD 2 TO TOTAL, SUBTRACT 2.60 FROM COST.
```

A Paragraph is a group of sentences starting with a Paragraph-Name - which must be written at the beginning (left-hand-side) of the page. Paragraph-Names are constructed according to the same rules as Data-Names (c.f.).

A Section is a group of paragraphs, beginning with a Section-Name and the word SECTION. Section-Names are constructed according to the same rules as Data-Names (c.f.).

Example:-

```
PROCEDURE DIVISION.          (Division Name)
INITIAL SECTION.             (Section-Name)
START.                       (Paragraph-Name)
        MOVE ZERO TO TOTAL.  (sentence)
```

A program may be written without any Section-Names (many programs are). A very short program may be written without any Paragraph-Names.

PROCEDURE DIVISION STATEMENTS ("Verbs")

The following pages describe the various forms of PROCEDURE DIVISION verbs. Note that, wherever literals are specified, Figurative Constants (SPACES, ZERO etc) may be used instead.

MOVE

```
MOVE  { Data-Name-1 }  TO  Data-Name-2  [, Data-Name-3, .....]
      { Literal     }
```

Examples:-

```
MOVE 2 TO TOTAL
MOVE "JONES" TO SURNAME
MOVE ZEROES TO TOTAL, SUB-TOTAL
```

Notes:

(i) When there is more than one destination-item (as in the last example above), the same source-item is MOVEd to each of the destinations.

(ii) There are restrictions on the types of data which can be MOVEd to the various types of destination-items. Illegal MOVEs are:-

(a) Alphanumeric data-item being MOVEd to a numeric or edited item.

(b) Edited data-item being MOVEd to a numeric data-item.

(iii) A data-item that is to be MOVEd to a numeric or edited data-item must have been previously assigned a numeric value (data-items without a VALUE clause are initially given the 6-bit code 00, which is not numeric).

(iv) Group data-items have no MOVE restrictions, since their data type is considered to be unspecified. Hence MOVEs of group items can involve no processing of the data (such as editing or aligning with decimal points).

(v) When the source and destination data-items are not the same length, the following rules apply:-

(a) Alphanumeric items are LEFT JUSTIFIED (unless otherwise specified) blank-filled (if necessary), or else truncated on the right (if necessary).

(b) Numeric items are aligned according to the position of the decimal points (if any); otherwise right-justified and zero-filled (if necessary); and truncated on the left (if the numberis too large) or on the right (if there are insufficient decimal places allowed in the destination). Truncation on the left results in the Run-Time message:- WARNING - DATA TRUNCATED - and program execution continues using the truncated data.

ADD

```
ADD  { Literal-1 }  [, { Literal-2 }  ; ........]
     { Data-Name-1 }   { Data-Name-2 }

     { TO     Data-Name-3 }  [, Data-Name-4, .....]
     { GIVING Data-Name-5 }

     [ROUNDED]  [ON SIZE ERROR  any statement(s)]
```

Examples:-

```
(i)    ADD SUM-1, SUM-2 GIVING TOTAL
(ii)   ADD 2 TO TOTAL
(iii)  ADD SUM-1, SUM-2 TO TOTAL
(iv)   ADD 2 TO SUM-1, SUM-2
(v)    ADD SUM-1 TO TOTAL ROUNDED
(vi)   ADD SUM-1 TO TOTAL; ON SIZE ERROR GO TO END.
```

The source-items (there is one source item in examples (ii), (iv), (v) and (vi); two in (i) and (iii)) are first added together. With GIVING, the result is placed in the destination-item (in example (i), the sum of SUM-1 and SUM-2 is placed in TOTAL). With TO, the result is added to the destination item (and the result placed in the destination item) (eg in example (iii), the final value of TOTAL will be the sum of SUM-1, SUM-2, and the original value of TOTAL).

With the GIVING option, the destination-item may be an edited item.

When the ROUNDED option is used, the data is ROUNDED (instead of truncated) if necessary to fit into the destination. For example, if the numeric result 2.6 was to be placed into a data-item with PICTURE 99, the result would normally be 02. But if the ROUNDED option had been specified, the number 03 would result (ROUNDED adds a "one" to the least significant digit if the following digit would have been 5 or greater).

The ON SIZE ERROR option is used to detect results which are too large to fit into the destination data-item. For example, if the result 123 were to be placed in a data-item with PICTURE 99, the truncation procedure would result in only the digits 23 being stored. But the ON SIZE ERROR option could be used to detect the fact that truncation had occurred. If data truncation occurs but the ON SIZE ERROR clause has not been specified, the warning error message "WARNING - DATA TRUNCATED" will be printed.

SUBTRACT.

```
SUBTRACT   { Literal-1   }  [ , { Literal-2   }  , ..... ]
           { Data-Name-1 }     { Data-Name-2 }

  FROM   { { Literal-3   }                                    }
         { { Data-Name-3 }   GIVING   Data-Name-4             }
         {   Data-Name-5   [ , Data-Name-6, ..... ]           }

  [ ROUNDED ]   { ON SIZE ERROR   any statement(s) }
```

Examples:-

```
(i)    SUBTRACT 2 FROM TOTAL
(ii)   SUBTRACT 2 FROM TOTAL GIVING PROFIT
(iii)  SUBTRACT PRICE-1, PRICE-2 FROM TOTAL
(iv)   SUBTRACT 2 FROM TOTAL; SUB-TOTAL
(v)    SUBTRACT 2 FROM TOTAL, ROUNDED; ON SIZE ERROR GO TO END.
```

The source data-items (if there are more than one - eg example (iii)) are added, and this result subtracted from each of the destinations. The result is placed in the destination data-items, unless GIVING is specified. (In example (ii), the original number in TOTAL will remain unchanged). The Data-Name following GIVING may be an edited item.

MULTIPLY.

```
MULTIPLY { Literal-1   } BY { Data-Name-2                          }
         { Data-Name-1 }    { Literal-2   } GIVING Data-Name-4     }
                            { Data-Name-3 }

[ ROUNDED ]  [ ON SIZE ERROR any statement(s) ]
```

Examples:-

(i) MULTIPLY 2 BY TOTAL
(ii) MULTIPLY TOTAL BY 2 GIVING G-TOTAL

Without GIVING the answer is placed in Data-Name-2 (eg in TOTAL in example (i)). Note therefore that "MULTIPLY TOTAL BY 2" would be illegal. With GIVING, the answer is placed in the Data-Name after GIVING.

DIVIDE.

```
DIVIDE { Literal-1   }  { INTO { Data-Name-2                          }  }
       { Data-Name-1 }  {      { Literal-2   } GIVING Data-Name-4     }  }
                        {      { Data-Name-3 }                        }  }
                        { BY   { Literal-3   } GIVING Data-Name-6     }  }
                        {      { Data-Name-5 }                        }  }

[ ROUNDED ]  [ ON SIZE ERROR any statement(s) ]
```

Examples:-

(i) DIVIDE 2 INTO TOTAL
(ii) DIVIDE 2 INTO TOTAL GIVING SUB-TOTAL
(iii) DIVIDE TOTAL BY 2 GIVING SUB-TOTAL
(iv) DIVIDE NUMBER INTO 273 GIVING PRICE

Note that the use of BY requires also the use of GIVING. (INTO may use GIVING if desired). Hence "DIVIDE TOTAL BY 2" would be illegal. Care should be taken in cases where the divisor could possibly have the value ZERO (resulting in an infinite answer) - the ON SIZE ERROR option should be used in these cases to detect the over-size answer.

Although usually negligible, the execution times for the DIVIDE operation may be appreciable if there are many of them. As a general rule, it is much faster to MULTIPLY than to DIVIDE. eg:-

MULTIPLY 0.5 BY TOTAL

would execute much more quickly than:-

DIVIDE 2 INTO TOTAL

COMPUTE.

The COMPUT verb is not available with this compiler. Arithmetic operations must be done with ADD, SUBTRACT, MULTIPLY and DIVIDE verbs.

IF.

IF Condition THEN Statement(s)-1 [{ ELSE / OTHERWISE } Statement(s)-2]

Examples:-

(i) IF A = 2 THEN GO TO FINISH.
(ii) IF NAME EQUALS "JONES" MOVE 3 TO SUM.
(iii) IF A = 2 THEN MOVE 3 TO SUM, GO TO FINISH.
(iv) IF A = 2 GO TO FINISH; ELSE MOVE 3 TO SUM.
(v) IF A = 2, THEN IF B = C GO TO FINISH.
(vi) IF A = 2 OR B = C GO TO FINISH.
(vii) IF A IS GREATER THAN B GO TO FINISH;
ELSE IF C = 2 MOVE 4 TO TOTAL, GO TO START;
ELSE MOVE ZERO TO TOTAL.

Conditions.

Conditions may be simple conditions or compound conditions.

Simple Conditions may take any of the following forms:-

{ Literal-1 / Data-Name-1 } { IS NOT EQUAL TO / NOT EQUALS / IS NOT GREATER THAN / IS NOT LESS THAN / IS NOT { POSITIVE / NEGATIVE / ZERO / NUMERIC } } { Literal-2 / Data-Name-2 }

Alternately, the symbols = > or < may be used in place of the words EQUAL TO, GREATER and LESS respectively.

NUMERIC may be used to test an alphanumeric data-item to determine if all the characters have numeric values (0 to 9), or to test a numeric data-item to see if its value has been defined.

GREATER and LESS may be used for comparing alphanumeric data-items, as well as numeric ones. The internal 6-bit codes of the first letters of the alphanumeric items are compared first - hence "A" would be found to be LESS than "B" etc. If the first letters are the same, the second letters are compared, etc. This facility allows alphabetic sorting of data. In such alphanumeric comparisons, SPACES are considered to be nulls (octal code 00). This ensures, for example, that the name ADAMS would be considered to come before ADAMSON.

Compound Conditions take the general form:-

Simple-Condition-1 { OR / AND } Simple-Condition-2 [{ OR / AND } Simple-Condition-3 ... (etc).

It is recommended that AND and OR compounding not be mixed in one Compound Condition. Conditions such as:-

A = B OR C = D OR E = F
or A = B AND C = D AND E = F

are (comparatively) easy to use and check in a program. If AND and OR are mixed, the truth or otherwise of the condition is determined in a serial fashion starting from the left. eg:-

A = B AND C = D OR E = F would be taken to mean:-

(A = B AND C = D) OR E = F.

Care also needs to be taken with negative compound conditions. For example, the opposite (or "complement") condition to:-

A = B OR C = D is actually:-

A IS NOT = B AND C IS NOT = D.

A maximum of eight Simple-Conditions are allowed in a Compound-Conditio

Implied subjects or objects (eg "IF A = B OR C") have not been implemented in this compiler.

Nested IF Statements.

Referring to the general format for the IF statement (at the top of p. 27), it may be seen that "Statement(s)-1" may contain further IF statements. For example:-

IF A = B THEN IF C = D GO TO FINISH; OTHERWISE ADD 1 TO TOTAL.

In such cases the ELSE or OTHERWISE is always taken to refer to the final "IF" preceding the ELSE or OTHERWISE. Use of such constructions is not, however, recommended (Compound Conditions are preferable, to avoid confusion).

"Statement(s)-2" of an IF statement may also contain further IF statements. The problem then is not so serious - for example, in the example (vii) on p. 27, it is clear that the first ELSE refers to the first IF and the second ELSE to the second IF.

GO TO - Form (i).

GO TO { Paragraph-Name / Section-Name }

Example:-

GO TO FINISH

Instead of the Run-Time system executing statements in sequence as they are written in the program, control is transferred to the start of the Paragraph or Section indicated.

GO TO - Form (ii).

```
GO TO { Paragraph-Name-1 } , { Paragraph-Name-2 } [ , ..... ]
      { Section-Name-1   }   { Section-Name-2   }

DEPENDING ON Data-Name-1
```

Example:- GO TO MON, TUES, WED, THURS, FRI DEPENDING ON DAY.

In this example, if the Data-Item DAY contains the value 1, control will pass to the Paragraph MON; if it contains the value 2, to Paragraph TUES etc. If the data-item is outside the relevant range (eg if DAY contains a value greater than 5 in this example), the GO TO statement will be ignored.

The maximum number of destinations allowed is 24.

PERFORM.

```
PERFORM { Par-Name-1  } [ THRU { Par-Name-2  } ]
        { Sect-Name-1 }        { Sect-Name-2 }

[ { Literal-1   } TIMES                                              ]
  { Data-Name-1 }
  { UNTIL Condition-1                                                }
  { VARYING Data-Name-2 FROM { Literal-2   }                         }
  {                          { Data-Name-3 }                        }
  {                    BY { Literal-3   }  UNTIL Condition-2         }
  {                       { Data-Name-4 }                           }
```

Examples:-

```
(i)   PERFORM PAR-1
(ii)  PERFORM PAR-1 THRU PAR-2
(iii) PERFORM PAR-1 7 TIMES
(iv)  PERFORM WRITE-LINE UNTIL LINE-NUMBER = 66
(v)   PERFORM CALC, VARYING NUMBER FROM 1 BY 1
            UNTIL NUMBER IS GREATER THAN 100
```

The PERFORM verb is used for two basic functions:-

(i) to execute a subroutine (examples (i), (ii)).

(ii) to execute a loop - ie to execute a set of statements a number of times (examples (iii), (iv), (v)).

Notes:

(i) The maximum number of TIMES is limited to 4096. If the Run-Time system finds the number of TIMES to be greater than 4096, a warning error message will be printed.

(ii) In example (i), control passes to the statement following the Paragraph- (or Section-) Name PAR-1. When the end of that Paragraph (or Section) has been reached (ie just before the next Paragraph- (or Section-) Name), control returns to the main program - ie to the next statement after the PERFORM statement.

(iii) In example (ii), execution of the subroutine begins after the Paragraph (or Section) Name PAR-1 and continues up to the last statement in the Paragraph (or Section) PAR-2.

(iv) The UNTIL option works rather differently from similar statements in BASIC or FORTRAN compilers. On each occasion before the loop is to be executed, the Run-Time system tests to see whether the condition has been satisfied. If it has, the loop is not executed on that occasion. For instance, in example (v) the loop would be executed only 100 times.

(v) A GO TO statement within a subroutine or loop may pass control to statements outside that subroutine (many COBOL compilers do not allow this). However, control must at some future time be returned to the end of the subroutine so that the main program (after the PERFORM statement) can be continued. (The EXIT statement may be placed at the end of a subroutine to allow this to happen). Similarly, another subroutine may be PERFORMED from within a subroutine. This compiler allows subroutines to be "nested" in this way up to the 13th level.

(vi) This compiler does not offer the AFTER function (this function may be carried out using nested PERFORMs).

EXIT.

This verb is usually used as the only verb in a Paragraph, to provide an end-point for a subroutine that is to be PERFORMed. Such an end-point may be required when there is more than one possible path through a subroutine.

Example:-

```
        PERFORM SUBR THRU SUBR-EXIT.
            .
            .
            .
SUBR.
        IF SUM    100 GO TO SUBR-EXIT.
        ADD 1 TO SUM.
            .
            .
            .
SUBR-EXIT.
        EXIT.
```

Here, the end of the subroutine will be reached (at SUBR-EXIT), whether SUM is greater than 100 or not. It is necessary for this to happen, so that control can be returned correctly to the main program.

STOP (or STOP RUN).

This statement stops execution of the program and returns control to the OS/8 monitor. It is normally the last statement executed in a program (but need not be at the physical end of the program listing). If the STOP verb is omitted in a program, one is automatically executed if the physical end of the program is reached.

EXAMINE.

Form (i):- EXAMINE Data-Name-1 TALLYING { UNTIL FIRST / ALL / LEADING } { Literal-1 / Data-Name-1 }
[REPLACING BY { Literal-2 / Data-Name-3 }]

Form (ii):- EXAMINE Data-Name-1 REPLACING { FIRST / UNTIL FIRST / ALL / LEADING } { Literal-3 / Data-Name-4 }
BY { Literal-4 / Data-Name-5 }

Examples:-

```
EXAMINE NAME TALLYING ALL "E"
EXAMINE NUMBER TALLYING LEADING ZEROES
EXAMINE NAME REPLACING ALL "E" BY N-CHARACTER
EXAMINE NAME TALLYING ALL "E", REPLACING BY "F"
EXAMINE NAME, REPLACING FIRST "E" BY "F".
```

This verb looks at individual characters within a data item. The TALLYING option counts up all specified occurrences of a particular character, and places the sum in an internal numeric data-item named TALLY. The contents of this item may be accessed in subsequent statements - for example:-

```
IF TALLY = 3 ADD 1 TO SUM.
```

The capacity of TALLY is four decimal digits (max: 9999).

DISPLAY.

DISPLAY { Literal-1 / Data-Name-1 } [, { Literal-2 / Data-Name-2 } , ...]

Examples:-

```
DISPLAY          (this prints a carriage-return and line-feed)
DISPLAY SURNAME
DISPLAY INITIALS, SURNAME
DISPLAY SURNAME, "  ", INITIALS
DISPLAY "YES OR NO  " (LINE);  ACCEPT ANSWER
DISPLAY (HOME) "ENTER STUDENT NAMES:-"
```

This verb prints the contents of the specified data item(s) on the console, followed by a carriage-return and line-feed. Multiple items in the same DISPLAY statement are printed on the same line (with no spaces between them). This compiler allows any type of data item to be DISPLAYED.

Since this compiler tends to be used in ways which require a lot of operator interaction, two additional DISPLAY options are provided:-

(LINE) specifies that a carriage-return, line-feed combination are not to be printed at the end of the line.
(HOME) erases the screen (VT50 terminals).

ACCEPT.

ACCEPT Data-Name

Examples:-

```
ACCEPT NAME
ACCEPT STUDENT-NUMBER
```

When this statement is being executed, the Run-Time system prints a question-mark (?) on the console - and then the operator is required to type in the characters or numeric value to be assigned to the Data-Name.

Notes:

(i) Only numeric or alphanumeric Data-Names may be specified (not edited or group names). For numeric Data-Names, only numeric values may be entered. A non-numeric entry will result in the error message "DATA NOT NUMERIC - RE-TYPE:-", and the item will need to be re-entered. Similarly, an alphanumeric entry which is too long will result in the error message "TOO LONG - RE-TYPE:-".

(ii) The usual rules applying to the MOVE verb also apply to ACCEPT in cases where the data entered is not the same length as the data-item (viz: alphanumeric data is left-justified and space-filled; numeric data is aligned with the decimal point and zero-filled).

(iii) Data entered during the execution of an ACCEPT statement may be terminated by typing the RETURN key (which results in a carriage-return and line-feed being printed); or by typing the ESCAPE key (which does not give a carriage-return or line-feed). The DELETE key may also be typed during the entry of data - this has the effect of deleting the previous character typed. When the console is a VT50 screen, the character is actually erased from the screen - otherwise, a back-slash character is typed for each character deleted.

(iv) If the RETURN (or ESCAPE) key is the only key typed during execution of an ACCEPT statement, the data-item is filled with zeroes (for a numeric item) or spaces (for an alphanumeric item).

(v) Characters which do not have unique 6-bit codes are not allowed as input during an ACCEPT statement. Such characters include LINE-FEED, BACK-SPACE, TAB and some control characters. Entry of such characters will give the error message "RE-TYPE:-".

FILE-PROCESSING VERBS

Note:

This compiler allows simultaneous INPUT and OUTPUT operations on two files on the same file-structured device and having the same File-Name. When this facility is used, the INPUT file is deleted when the OUTPUT file is closed (this is a similar mode of operation to that used by the OS/8 Edit program, for example).

However, so that the COBOL program can distinguish between the two files, the extension TM ("temporary") must be added to one of the File-Names. COBOL deletes this extension when it CLOSEs the file.

Example:- This program copies the file DATA onto a new file which will also be called DATA. The original file is then deleted.

```
        FILE-CONTROL.
                SELECT DATA.
                SELECT DATA.TM.
        DATA DIVISION.
        FILE SECTION.
        FD DATA.
        01 DATA-REC      PIC X(50).
        FD DATA.TM.
        01 DATA-TM-REC   PIC X(50).
        WORKING-STORAGE SECTION.
        PROCEDURE DIVISION.
                OPEN INPUT DATA, OUTPUT DATA.TM.
        NEXT-REC.
                READ DATA AT END GO TO FINISH.
                MOVE DATA-REC TO DATA-TM-REC.
                WRITE DATA-TM-REC.
                GO TO NEXT-REC.
        FINISH.
                CLOSE DATA, DATA.TM.
```

OPEN.

```
OPEN { INPUT  }
     { OUTPUT } File-Name-1 [, File-Name-2, ..... , etc.]
     { I-O    }
```

Examples:-

```
OPEN INPUT ADDR
OPEN INPUT ADDR, NAMES
OPEN INPUT ADDR, NAMES; OUTPUT PRFILE
```

Before each file is used in the PROCEDURE DIVISION, it must be OPENed. The functions performed by the Run-Time system during an OPEN operation are:-

(i) the OS/8 device handler for the particular device is read into memory, if necessary;
(ii) the file is located on the device, and the first block of data read into a memory buffer (but not transferred into the Record-Name data area);
(iii) for OPEN OUTPUT, a "tentative" file is created on the device.

Notes:-

(i) An INPUT function is one which READs data from a device into memory. An OUTPUT function is one which WRITEs data from memory onto a device.

(ii) I-O must be specified for Random-Access files (c.f.) only. This allows both INPUT and OUTPUT operations to be made on the file, if desired.

(iii) A file which is already OPEN may not be OPENed again (it must be CLOSEd first).

(iv) Only one OUTPUT file may be OPENed on a particular device at any one time. (This is an OS/8 restriction - OS/8 allocates the largest possible contiguous area on a device for an OUTPUT file). But with two disk drives, for example, one OUTPUT file may be OPENed on each.

(v) The Run-Time system has four PDP-8 memory pages available for device handlers (in addition to the system device handler, which is always resident). Device handlers may be one page or two pages long. If an OPEN is attempted and the handler is not in core, and the handler area is full, a fatal "DEVICE I-O" error will be printed.

CLOSE.

CLOSE File-Name-1 [, File-Name-2, ...]

Examples:- CLOSE ADDR
CLOSE ADDR, PRINT

Close must be the last function performed on every file during execution of a program. OUTPUT files which are not CLOSEd will not be permanently entered on the directory for the device (if file-structured); and OUTPUT to a line-printer, for example, would not be completed.

READ.

READ File-Name RECORD [{ AT END / INVALID KEY } any statement(s)]

Examples:-

```
READ NAMES
READ NAMES AT END GO TO FINISH
READ ADDR INVALID KEY GO TO ERROR-ROUTINE
```

This verb reads a record from the specified file into the Record-Name area in the DATA DIVISION.

For sequential files, the next record is read (ie the one in the file following the previous one read), unless the end-of-file mark is found.

For Random-Access files, the record to be read is found from the value of the ACTUAL KEY Data-Name. For example, if the ACTUAL KEY contains the number 47, the READ statement would read the 47th record in the file, etc. If the value of the key results in the calculated position of the record being outside the file, the INVALID KEY condition is then satisfied.

Notes:-

(i) Execution of a READ statement may not require an actual data transfer from the device. This is because device transfers are made in whole OS/8 blocks (512 characters) into a Primary Data Buffer which is allocated for each file. A READ statement only transfers the data from the file's Primary Data Buffer into the record data area in the program. Only if the Primary Data Buffer is exhausted before the complete record has been transfered is a block transfer from the device required. (Random-Access READs normally do require a device transfer).

(ii) If the number of characters in the record read is not equal to the number of characters in the record data area (or is greater than that area in the case of a variable-length-record file), an error message is printed and the program execution terminated.

(iii) When an end-of-file or INVALID KEY condition is discovered but there is no AT END or INVALID KEY clause in the READ statement, an error message is printed and the program execution terminated.

(iv) A READ from a non-file-structured device operates as follows:-

(a) External 8-bit ASCII codes are changed to internal 6-bit codes.

(b) Null (ASCII 000) and line-feed characters are ignored.

(c) The carriage-return character is used to detect an end-of-record (but is not stored in the Record area).

(d) End-of-file is recognised in the usual OS/8 manner - eg by an end-of-file character on Mag-tape, cassette or card-reader; end-of-tape on paper-tape reader; or a CNTRL/Z character from a terminal.

(e) Data is assumed to be alphanumeric (PIC X) - so that, for example, there is no conversion of numeric data to fit into specified data formats. However, if the record is found to be shorter than the record data area, the latter is blank-filled. If a record is found which is larger than the record data area, a fatal RECORD LENGTH error message will be printed.

WRITE.

```
WRITE Record-Name [ { BEFORE } ADVANCING { Literal   } { LINE  } ]
                  [ { AFTER  }           { Data-Name } { LINES } ]

                  [ INVALID KEY any statement(s) ]
```

Examples:-

```
WRITE ADDR-REC
WRITE PRINT-REC BEFORE ADVANCING 1 LINE
WRITE NAMES-REC, INVALID KEY GO TO ERROR-ROUTINE.
```

This verb writes the specified record (which must be an 01-level Record-Name) onto the file whose FD statement preceeds the Record-Name description in the program.

For sequential files, the record is written onto the device immediately after the record previously written. For Random-Access files, the record is written at a location calculated from the value of the ACTUAl KEY data item.

The INVALID KEY option may be used both for Random-Access files (to detect when the value of the KEY is outside the range of the existing file area); and for sequential files (to detect when the space allocated for the file on the device has been exceeded).

The ADVANCING option is intended for non-file-structured devices (line-printer, paper-tape punch, etc). It causes a carriage-return character and the specified number of line-feed characters (maximum: 4095) to be written. The data from the Record-Name area may be written either BEFORE or AFTER the line-feed characters. However, a record written to a non-file-structured device is always terminated by a carriage-return character, whether the ADVANCING option has been specified or not. Data is written to non-file-structured devices in 8-bit ASCII code in standard OS/8 format.

Note that, in COBOL, a READ statement always specifies the File-Name, whereas a WRITE statement always specifies the Record-Name.

BREAK.

BREAK

This verb is not a standard COBOL verb, but has been implemented to help in the Run-Time de-bugging of COBOL programs. Execution of the statment BREAK results in the word - BREAK - being printed on the console and control being passed to the OS/8 monitor. At this time the OS/8 de-bugging program ODT may be called upon to examine or change data in the DATA DIVISION (which is in memory field 3). (For instructions on using ODT, refer to the OS/8 Handbook). The octal word addresses of data items in the DATA DIVISION are given on the compiled program listing.

To re-start the COBOL program at the statement following the BREAK statement, the charagters ØG (zero and G) are typed - this instructs ODT to re-start the Run-Time system at the re-start address zero.

More than one BREAK may be employed to check the operation of a program. When the program is working correctly, the BREAK statement is normally removed from the source program.

Examples:-

(i)
```
ADD A TO B.
BREAK.
DISPLAY B.
```

(ii)
```
IF A = B THEN BREAK, GO TO NEXT; ELSE GO TO FINISH.
```

Run-Time Example:-

When a program is executing and a BREAK verb is encountered, data in the DATA DIVISION may be examined and/or altered as shown:-

- BREAK -	(printed by the Run-Time system)
.ODT;	(monitor dot printed by OS/8. ODT is called by typing "ODT")
30010/6162 6163;	(location 0010 in field 3 (DATA DIVISION) is examined. ODT prints the contents of this location - viz. the 6-bit codes 61 and 62 (representing the numbers 1 and 2). The required new contents 6163 are then typed in to replace the old contents)
ØG	(now, typing zero followed by G will cause the COBOL program to be resumed at the statement following the BREAK statement).

SAMPLE PROGRAMS

These three short programs demonstrate a few of the features of the compiler, including the use of Random-Access files.

Sample Program No 1:-

```
PROGRAM-ID.
        CREATE.
REMARKS.
        THIS PROGRAM CREATES A FILE, "STUDEN.DA", CONTAINING STUDENT
        NUMBERS, NAME, AND CURRENT BALANCE OF ACCOUNT.
        THE FILE IS ORDERED ACCORDING TO STUDENT NUMBERS, WHICH
        RANGE FROM 1 TO 10.  DATA ENTRY IS VIA THE CONSOLE.
AUTHOR.
        R. BARNES,  ASC,   JULY 1977.
FILE-CONTROL.
        SELECT STUDEN.DA.
DATA DIVISION.
FILE SECTION.
FD STUDEN.DA.
01 STUDENT-REC.
        02 STUDENT-NO        PIC 999    VALUE ZERO.
        02 STUDENT-NAME      PIC X(20).
        02 STUDENT-ACCOUNT   PIC S9(6)V99    VALUE ZERO.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
        OPEN OUTPUT STUDEN.DA.
        DISPLAY "STUDENT ACCOUNT FILE-CREATE PROGRAM.".
        DISPLAY "ENTER TEN STUDENT NAMES AND THEIR CURRENT ACCOUNT:-".
        DISPLAY.
        PERFORM INPUT-DATA 10 TIMES.
        CLOSE STUDEN.DA.
        STOP RUN.
INPUT-DATA.
        ADD 1 TO STUDENT-NO.
        DISPLAY "STUDENT NAME  " (LINE).
        ACCEPT STUDENT-NAME.
        DISPLAY "CURRENT-ACCOUNT  " (LINE).
        ACCEPT STUDENT-ACCOUNT.
        DISPLAY.
        WRITE STUDENT-REC.
```

Notes on the above program:-

(i) Student numbers take the values 1, 2, 3 etc automatically.

(ii) Note that this compiler allows VALUE clauses in the FILE SECTION of the DATA DIVISION.

(iii) Note the powerful use of the PERFORM verb to execute a subroutine a certain number of times.

Sample Program No 2:-

```
PROGRAM-ID.
        PRINT.
REMARKS.
        THIS PROGRAM PRINTS THE DATA WHICH IS CONTAINED IN THE
        STUDENT ACCOUNT FILE "STUDEN.DA" ONTO THE LINE-PRINTER.
AUTHOR.
        R. BARNES,    ASC    JULY 1977.
FILE-CONTROL.
        SELECT STUDEN.DA.
        SELECT REPORT, ASSIGN TO LPT.
DATA DIVISION.
FILE SECTION.
FD STUDEN.DA.
01 STUDENT-REC.
        02 STUDENT-NO          PIC 999.
        02 STUDENT-NAME        PIC X(20).
        02 STUDENT-ACCOUNT     PIC S9(6)V99.
FD REPORT.
01 PRINT-REC.
        02 P-NO                PIC ZZ9.
        02 FILLER              PIC XXX.
        02 P-NAME              PIC X(20).
        02 FILLER              PIC X(6).
        02 P-ACCOUNT           PIC ----,--9.99.
WORKING-STORAGE SECTION.
PROCEDURE DIVISION.
        OPEN INPUT STUDEN.DA, OUTPUT REPORT.
        MOVE " NO.       NAME                              ACCOUNT"
             TO PRINT-REC.
        WRITE PRINT-REC BEFORE ADVANCING 2 LINES.
        MOVE SPACES TO PRINT-REC.
NEXT-STUDENT.
        READ STUDEN.DA, AT END GO TO FINISH.
        MOVE STUDENT-NO TO P-NO.
        MOVE STUDENT-NAME TO P-NAME.
        MOVE STUDENT-ACCOUNT TO P-ACCOUNT.
        WRITE PRINT-REC BEFORE ADVANCING 1 LINE.
        GO TO NEXT-STUDENT.
FINISH.
        CLOSE STUDEN.DA, REPORT.
        STOP RUN.
```

Notes on the above program:-

(i) Note the use of the word FILLER to denote data characters which will not be referred to specifically in the program (these characters will normally contain spaces).

(ii) Note that in order to print the report, the data items must be MOVEd individually so that the editing will be carried out (if group items are MOVEd there is no processing of the data).

(iii) Note the use of the AT END option to detect the end of the file.

(iv) If a separate printer is not available, TTY could be used in place of LPT.

Sample Program No. 3:-

```
PROGRAM-ID.
        UPDATE.
REMARKS.
        THIS PROGRAM ALLOWS UPDATING OF THE STUDENT ACCOUNT FILE.
        BY SPECIFYING THE STUDENT NUMBER, THE OPERATOR CAN OBTAIN THE
        STUDENT NAME (FOR VERIFICATION).
        NEW INFORMATION MAY THEN BE ENTERED FOR THAT STUDENT'S
        CURRENT ACCOUNT BALANCE.   TO STOP THE PROGRAM, ENTER
        ZERO (OR RETURN) FOR THE STUDENT NUMBER.
AUTHOR.
        R. DREW,  ASC     JULY 1977.
FILE-CONTROL.
        SELECT STUDEN.DA, ACCESS MODE IS RANDOM,
                          ACTUAL KEY IS STUDENT-KEY.
DATA DIVISION.
FILE SECTION.
FD STUDEN.DA.
01 STUDENT-REC.
        02 STUDENT-NO              PIC 999.
        02 STUDENT-NAME            PIC X(20).
        02 STUDENT-ACCOUNT         PIC S9(6)V99.
WORKING-STORAGE SECTION.
77 STUDENT-KEY        PIC 999.
77 ANSWER             PIC X.
PROCEDURE DIVISION.
        DISPLAY "STUDENT ACCOUNT UPDATE PROGRAM.".
        OPEN I-O STUDEN.DA.
NEXT-STUDENT.
        DISPLAY.
        DISPLAY "STUDENT NUMBER  " (LINE).
        ACCEPT STUDENT-KEY.
        IF STUDENT-KEY = ZERO GO TO FINISH.
        READ STUDEN.DA;  INVALID KEY DISPLAY "NUMBER INVALID",
                                     GO TO NEXT-STUDENT.
        DISPLAY STUDENT-NAME, " CORRECT - Y OR N  " (LINE).
        ACCEPT ANSWER.
        IF ANSWER NOT = "Y" GO TO NEXT-STUDENT.
        DISPLAY "NEW CURRENT ACCOUNT  " (LINE).
        ACCEPT STUDENT-ACCOUNT.
        WRITE STUDENT-REC.
        GO TO NEXT-STUDENT.
FINISH.
        CLOSE STUDEN.DA.
        STOP RUN.
```

Notes on the above program:-

This program allows changing ("updating") of the information contained in the student account file which was created by Sample Program No. 1. Although the file was created as a fixed-length-record sequential file, it is here updated as a Random-Access file. When the operator enters the student number, this becomes the value of the ACTUAL KEY - ie it indicates which record of the file is to be accessed. When this record has been read into the STUDENT-REC record area, the information (eg name, current-account etc) can be processed or changed, and the record written back onto the file (using the same ACTUAL KEY, so that it goes back in the same position).

ERROR MESSAGES

There are two classes of error messages - Compiler error messages and execution (Run-Time) error messages.

Compiler Error Messages.

Compiler error messages are followed by the line of source code and, where applicable, a circumflex (or up-arrow) underneath the line to indicate just where the error was detected. For example:-

```
BAD PICTURE:-
        77 TOTAL        PICTURE.999Q
                                ^
```

Note that the circumflex is printed in the position where the compiler detects the error. In some cases this is not where the error actually is. For example:-

```
NO SUCH NAME:-
        ADD PRICE TO SUM GIVING TOTAL.
                   ^
```

- here the actual error is the word TO, since TO and GIVING are not allowed in the same ADD statement. The words following TO are assumed by the compiler to be destination Data-Names. The line should have read:-

```
ADD PRICE, SUM GIVING TOTAL.
```

Similarly, a "MISSING PERIOD" error indicated at the beginning of a line may refer to an error at the end of the previous line.

COMPILER ERROR MESSAGES

ACCESS MODE ERROR

The ACCESS MODE specified is not either F or V.

BAD PICTURE.

Illegal character in PICTURE; error with brackets; wrong ordering of symbols; more than ten digits before decimal point, or more than six after.

CHAR. COUNT ERROR IN PREVIOUS ITEM.

(i) REDEFINES error: A Data-Name or group of data-names which REDEFINE a previous Data-Name (or group) do not have the same total number of characters.

(ii) VALUE error: The literal in a VALUE clause does not have the same number of characters (or the same number of characters before the decimal point, etc) as the Data-Name PICTURE.

COBRT.SV NOT FOUND.

The /C option has been specified to the compiler, but the Run-Time system program COBRT.SV is not present on the system device.

DEVICE NOT FOUND.

The device specified following the word ASSIGN in a SELECT sentence does not have a device handler on the OS/8 system. Common device names include SYS, LPT, TTY, RXA0, RXA1, RKA0, DTA0 etc.

ERROR - NO SUBSCRIPT.

A Data-Name defined with an OCCURS clause (or defined within a group which has an OCCURS clause) is not followed by a subscript (in brackets) in a PROCEDURE DIVISION statement.

ERROR - SUBSCRIPTED SUBSCRIPT.

Subscripts in the PROCEDURE DIVISION may not themselves be subscripted.

eg SUM (A) but not SUM (A (2))

ERROR - UNDEFINED ACTUAL KEY.

An ACTUAL KEY has been specified in a SELECT clause, but the Data-Name has not been found (with its PICTURE) in the DATA DIVISION.

FILE NOT ASSIGNED.

A File-Name following the symbol FD has not been ASSIGNed to a device in a SELECT clause.

ILLEGAL CHAR.

A character, other than those in the allowable Character Set (c.f.) has been detected in the source program. Illegal characters (eg backspace, CNTRL characters) may be invisible on a program listing, or they may appear as a legal character. Normally the faulty line should be re-typed.

ILLEGAL LITERAL.

A non-numeric character has been found in a numeric (unquoted) literal.

ILLEGAL MOVE.

An alphanumeric Data-Name is being MOVEd to a numeric or edited Data-Name; an edited Data-Name is being MOVEd to a numeric Data-Name.

ILLEGAL NAME.

A Data-Name (starting with a letter A to Z) was expected but not found; a device name (in an ASSIGN clause) is more than 4 characters; a File-Name (c.f.) is not in OS/8 format.

ITEM NOT NUMERIC.

An arithmetic operation has been specified for a Data-Name with a non-numeric PICTURE.

```
eg      ADD PSUM TO TOTAL
```

where TOTAL has the edited PICTURE 99.99.

LEVEL NO. ERROR.

(i) The level-number following a group name is not greater than the level-number of the group name.

```
eg      02 ADDRESS.                (group name)
          01 LINE-1          PIC X(20).
```

(ii) A level-number is greater than the previous level-number and the previous item was not a group name.

```
eg      02 ADDRESS          PIC X(50).
          04 PHONE-NO       PIC X(10).
```

(iii) A level-77 item is not in the WORKING-STORAGE SECTION.

(iv) Within a group OCCURS item, a data item having a greater number of subscripts has been written before a data-item with less subscripts (c.f. OCCURS).

LINE TOO LONG.

A line of the source program has been found having more than 80 characters.

MISSING KEY.

A SELECT clause has specified "ACCESS MODE IS RANDOM", but there is no ACTUAL KEY clause, or "ACTUAL KEY" is spelt wrongly (there should be a single space between "ACTUAL" and "KEY").

MISSING PERIOD.

(i) Period (.) missing after a PICTURE clause.

(ii) Period missing after a Paragraph-Name, or after a word written right at the beginning of the line (which is assumed to be a Paragraph-Name).

(This error is normally detected by the compiler at the beginning of the line following the faulty line).

NAME PREVIOUSLY DEFINED.

Either a Data-Name encountered in the DATA DIVISION, or a Paragraph-Name (or Section-Name) encountered in the PROCEDURE DIVISION, has been defined previously in the program.

NO FILE-CONTROL SECTION.

Missing FILE-CONTROL heading (this heading is required even if there are no entries), or "FILE-CONTROL." is spelt wrongly (eg leading spaces, missing hyphen, or missing period).

NO ROOM FOR OUTPUT FILE.

A compiled output file has been requested, but there is not a large enough empty space on the specified device (use the SQUISH CCL command or delete some files).

NO SUCH NAME.

A Data-Name or File-Name referred to in the PROCEDURE DIVISION has not been defined in the DATA DIVISION, or has been spelt wrongly.

NO WORKING-STORAGE SECTION.

The WORKING-STORAGE SECTION is missing (the heading is required even if there are no entries), or "WORKING-STORAGE SECTION." is spelt wrongly (eg leading spaces, no hyphen, multiple spaces between "STORAGE" and "SECTION", or no period at the end).

PRECEDING LITERAL NOT TERMINATED.

A quote sign (" or ') was found, indicating the beginning of an alphanumeric literal, but the same closing quote sign has not been found after 150 characters.

PROGRAM TOO LONG.

The PROCEDURE DIVISION is too long. The limit is 5400 (octal) locations - approx 700 (decimal) statements. The program needs to be segmented and overlays (c.f.) utilized.

REDEFINES ERROR.

A REDEFINES clause is associated with a Data-Name which is either:-

(i) already within a REDEFINES group; or

(ii) within a group item which has an OCCURS clause.

SYNTAX ERROR.

The structure of a statement or clause is incorrect. A large number of situations can give rise to this error message, including:- MOVE without TO; GIVING followed by more than one Data-Name; unrecognised or unexpected word found; a VALUE clause specified with an OCCURS clause, etc.

TOO MANY EDITED PICS.

The storage capacity for edited PICTURES has been exceeded. Capacity: approx. 1500 bytes. (Note: Only PICTUREs of edited Data-Names contribute to this total).

TOO MANY DATA ITEMS.

(i) The number of Data-Names defined in the DATA DIVISION is too large (limit: approx. 400); or the names are too long (storage area for names is approx. 3500 (decimal) characters). (Note: FILLER does not require Data-Name storage).

(ii) The data storage area for data items and literals has been exceeded. The limit for data-items and literals combined is approx. 3000 (decimal) bytes - however this may be reduced by up to 512 bytes when the /O (overlay) option is specified.

TOO MANY FILES.

There are more than 7 SELECT clauses specifying distinct files.

TOO MANY ITEMS IN GROUP.

A group Data-Name in the DATA DIVISION contains too many Data-Names within the group. Limit: 128 independent Data-Names within a group (when a group contains other groups, most Data-Names within the sub-groups do not contribute to this total).

TOO MANY PAR. NAMES.

There are too many Paragraph- and/or Section-Names. (Limit: 128 total).

UNDEF. PAR. NAME.

The Paragraph- or Section-Name specified in the error message is referred to in the program (eg in a GO TO or PERFORM statement), but is not defined as a Paragraph or Section heading, or has been spelt wrongly.

UNRECOGNISED STATEMENT.

A verb or clause is expected, but a recognised one has not been found, or has been mis-spelt.

VALUE ERROR.

(i) The VALUE given to a Data-Name is negative although the Data-Name is not signed (no "S" specified in PICTURE).

(ii) A non-numeric character has been found in the VALUE, although the Data-Name has a numeric PICTURE.

WRONG NO. OF FD STATEMENTS.

The number of FD (File-Description) statements in the DATA DIVISION is not equal to the number of SELECT statements in the FILE-CONTROL section; or "FD" is not at the beginning of a line, or has been mis-spelt.

WRONG NUMBER OF SUBSCRIPTS.

The number of subscripts following a Data-Name in the PROCEDURE DIVISION does not agree with the data structure in the DATA DIVISION.

RUN-TIME ERRORS

Errors which are detected by the Run-Time system (when the program is actually executing) are printed on the console preceded by the message:-

RUN-TIME ERROR AT PROCEDURE LOCN. xxxx :-

- where "xxxx" is a 4-digit octal number which indicates where in the PROCEDURE DIVISION the error occurred. The source statement which caused the error can be found using a compiler listing (c.f.) of the program. Compiler listings show the PROCEDURE DIVISION addresses at the beginning of each line of source code.

Most Run-Time errors are fatal - that is, they result in the program being stopped and control returning to the OS/8 monitor. At this time, the status of data stored in the DATA DIVISION may be ascertained (if desired for de-bugging purposes), by running the program CDUMP (c.f.), which prints out the addresses of the DATA DIVISION and their contents.

RUN-TIME ERROR MESSAGES

CONTROL/C NOT PERMITTED.

Termination of program execution by typing CONTROL/C is not allowed since a Random-Acces (I-O) file has been altered (by WRITE statements), but the file has not been closed. (Note: In all other cases a program may be stopped by typing CONTROL/C).

DATA NOT NUMERIC.

This error message includes a specification of the DATA LOCN. of the data item which was found not to be numeric. This error can occur when:-

(i) The contents of a data item specified in an arithmetic operation have been found to be non-numeric. This situation can occur, for example, when the contents of the item have not been defined - ie when there was no VALUE clause for the item, and no arithmetic data has been placed into the item.

(ii) An ACCEPT, READ or MOVE statement has tried to place data into a numeric data item, but has found that the data is non-numeric. (For ACCEPT, the Run-Time system will re-type the question-mark and allow re-entry of the data).

(iii) The contents of an ACTUAL KEY data item have been found to be non-numeric, during a READ or WRITE on a Random-Access file.

DEVICE I-O.

(i) A non-recoverable hardware error has been detected during a READ or WRITE statement.

(ii) A file is being OPENed but the area available for OS/8 device handlers (4 PDP-8 pages) is already full.

FILE NOT CLOSED.

A STOP verb has been encountered (or the physical end of the program reached) and one or more output or I-O files has not been CLOSEd.

FILE NOT FOUND.

During an OPEN INPUT or OPEN I-O operation the File-Name specified has not been found on the directory of the ASSIGNed file-structured device.

FILE NOT OPENED.

A READ or WRITE operation has been requested for a particular file, but an OPEN statement for that file has not been executed.

FILE TOO BIG.

A file on a file-structured device (eg disk or DECtape) has reached the OS/8 limit of 4096 blocks (just over 2 million COBOL characters).

RECORD LENGTH.

(i) While READing a file with fixed-length records (RECORDING MODE specified or assumed to be F), the length of the record read does not agree with the record-length in the Record-Name layout in the DATA DIVISION.

(ii) While READing a file with variable-length records (RECORDING MODE specified as V), the length of the record read is greater than the largest Record-Name layout for that file in the DATA DIVISION.

SUBSCRIPT OUT OF BOUNDS.

The value of a subscript is zero or negative, or greater than the number specified in the relevant OCCURS clause.

TOO LONG.

During execution of an ACCEPT statement, the data entered has more characters than appear in the PICTURE for that data item (or more characters before the decimal point, or after the decimal point). The data must be re-entered.

TOO MANY NESTED PERFORMS

A subroutine has been PERFORMed, and from within the subroutine another PERFORM statement has been made - and this nesting process has continued past the 13th level, which is the maximum allowed.

TOO MANY "TIMES".

A PERFORM.....TIMES statement is being executed and the number of TIMES has been found to exceed 4095. Execution will continue, with the number of TIMES equal to the desired number modulo 4096. (To PERFORM a routine more than 4095 TIMES, use the VARYING option).

TRANSFER PAST END-OF-FILE.

(i) When WRITing a sequential file, the capacity of the largest contiguous area available on the device has been exceeded.

(ii) When READing or WRITing a Random-Access file, the value of the KEY is such as to require a transfer out of the range of the storage area assigned for the file on the device (and an INVALID KEY clause has not been specified in the READ or WRITE statement).

USER ERROR 0 AT xxxxx.

This error message is printed by OS/8, and indicates an error while loading a core image (SV) program. In particular, when a compiled core-image program is being loaded (by the R or RUN command), this message can indicate that the Run-Time system program COBRT.SV is not present on the system device.

WARNING - DATA TRUNCATED.

A MOVE, ACCEPT or GIVING arithmetic operation has resulted in a numeric value which will not fit the destination data area without being truncated on the left. Program execution will continue, using the trucated data.

THE PROGRAM CDUMP (Dump Program)

This program is usually used as a diagnostic aid to check the operation of COBOL programs. CDUMP has several options:-

(i) The contents of either the DATA DIVISION (of a COBOL program which has just been run), or a particular disk (or tape) file, may be DUMPed.

(ii) The data may be printed either in character form or in octal form.

DUMPs may be made either to a line-printer or to the console. All DUMPs are printed with the octal memory addresses at the start of each line.

Operation.

With the OS/8 monitor operating, type:-

.R CDUMP↵

Then type in the Command Decoder line, specifying an optional output device, an optional input file-name, and an optional swithch - O. The output device may be TTY: or LPT: (if no output device is specified the DUMP will be to the line-printer). If no input file-name is specified, the DATA DIVISION (of the COBOL program just run) will be DUMPed. The "O" switch produces a print-out in octal, rather than character form.

Examples:-

*LPT:↵ or *↵

- this will print the DATA DIVISION in character form to the printer.

*TTY:<RXA1:STUDEN.DA/O↵

- this will print the contents of the file STUDEN.DA from the right-hand floppy disk, in octal form, onto the console.

If an input file-name has been specified, the program will ask for a:-

BLOCK NO:

- and requires entry of a decimal number indicating which OS/8 block (of 512 characters) is to be DUMPed. BLOCK NO. 1 is the first block of the file, etc. After DUMPing one block, the program will request another BLOCK NO. The program may be terminated by typing CNTRL/C.

Note that the end-of-record character is printed in character mode as the "underline" character ("back-arrow" on some terminals). An end-of-file is indicated by two consecutive "underline" characters.

Example of a line of CDUMP output (character mode):-

```
0200   AB CD EF GH IJ KL MN OP   QR ST UV WX YZ 01 23 4_
```

Note that two characters occupy one PDP-8 word. A space is printed between each PDP-8 word. Three spaces are printed between the two groups of eight PDP-8 words on a line. So for example, "CD" occupies word no. 0201. "QR" occupies word no. 0210. And the end-or-record mark is at address 0217R (right-hand byte of word no. 0217).

THE PROGRAM CSORT (Sort/Merge Program)

This program is used to copy an existing file or set of files and to create a new file, and in the process to sort the records into a certain order in the new file. The new order will be determined by the values of a string of characters (called the "KEY") within each record - the strings could represent for example, "student number", or "surname and initials", etc. The resulting file is ordered in increasing order of the values of the keys (this implies alphabetic order for alphanumeric keys). When keys consist of a mixture of numbers, letters and other characters, the records will be ordered according to the values of the 6-bit codes of the characters. A record having a key consisting of LOW-VALUES will be placed first in a sorted file; a record whose key is HIGH-VALUES will be placed last. Spaces are always considered to be nulls (LOW-VALUES) for sorting.

To run the program CSORT, type:-

.R CSORT↵

The Command Decoder line is then entered in the form:-

*OUTPUT-FILE<INPUT-FILE(S)/V/D↵

Multiple input files are separated by commas. The optional /V switch indicates that the input file(s) contain variable-length records. The optional /D switch causes the input file(s) to be deleted before the output file is written. This may be useful where there could be insufficient room for the output file. Up to 5 input files may be specified.

Example:-

*RXA1:NEW<RXA1:OLD↵

- this will sort the fixed-length-record file OLD from the right-hand floppy disk, creating the sorted file NEW. The original file OLD will not be deleted, since the /D switch has not been used.

The program will then request:-

SORT KEY:- FROM CHAR. NO:
UP TO AND INCLUDING CHAR. NO:

at which time the position of the key in each record should be indicated. For example, the key may consist of characters no. 9 to 20 (inclusive) in each record (counting from the beginning of the records).

The program will then operate as follows:-

(i) Approximately 20,000 characters of the input file(s) are read into memory (or less if an end-of-file mark is reached).

(ii) A list of the record memory addresses is constructed, and this list sorted in memory according to the values of the respective keys.

(iii) The records are written from memory in the sorted order onto a scratch (temporary) file on the system device (or else direct to the output file if the whole file is 20,000 characters or less).

(iv) This process is repeated until all the input file(s) have been read. If seven scratch files are written before all the data has been read, the seven files are then merged into a secondary scratch file, and this process may be repeated until up to seven secondary scratch files have been created.

(v) If the "/D" option has been specified, the input file(s) are now deleted from the input device(s).

(vi) The scratch files are read together into memory, and the final merged file written to the specified device.

(vii) As the records are written to the final output file, the key values for each record are displayed on the console, for checking purposes. This operation (of displaying keys) may be stopped at any time by typing "S"

(viii) The sort operation may be terminated at any time by typing CNTRL/C.

Notes.

(i) If two (or more) records have identical keys, the ordering of these records will be unspecified.

(ii) The time required for the in-core sorting of record addresses depends critically upon the number of records in core. For files with short records (eg less than 50 characters), and with more than a couple of hundred records, the in-core sorting of record addresses can take a noticeable time. Do not be concerned in such cases if the computer seems to "go dead" for as long as several minutes.

(iii) Unless the whole file can be accommodated in memory (capacity: approx. 20,000 characters), then sufficient scratch space must be available on the system device to hold a copy of the complete (segmented) file(s).

Error messages.

ERROR - SCRATCH AREA INSUFFICIENT.

The scratch area available on the system device is insufficient.

ERROR - OUTPUT AREA INSUFFICIENT.

The output device cannot accommodate the sorted file. If the input and output files are to be on the same device, the original file(s) may be deleted using the /D option to make room on the device for the new file.

ERROR - VARIABLE-LENGTH RECORDS.

Records in the input file(s) have been found to be not all the same length, and the /V option has not been specified.

ERROR - DEVICE I-O.

A non-recoverable hardware read or write error has been encountered.

ERROR - RECORD TOO LARGE.

The input file(s) were specified as having variable-length records (/Voption specified); and the file contains more than approx. 20,000 characters; and a record has been found with more than 2047 characters.

ERROR - INPUT FILE(S) TOO LONG.

The input file(s) have exceeded approx. 4.7 million characters, or 200,000 records. Or for variable-length records, approx. 1.0 million characters, or 45,000 records.

COMPILER CAPACITIES - SUMMARY.

Numerical Accuracy: Ten characters before the decimal point; Six characters after the decimal point (hence up to sixteen decimal places numeric accuracy).

Program Data Storage: Approx. 3,000 characters (DATA DIVISION plus literals).

No. of Data-Items: Approx. 400.

Storage for Data-Names: Approx 3,500 characters.

No. of Paragraph and/or Section Names: 128.

No. of Files in any Program: 7.

Max. File Size on File-structured (Random-addressable) devices: Just over 2 million characters.

File Size on Non-File-structured Devices: No limit.

No. of Records per File: No limit except as above.

Max. No. of Program Statements: Approx 700 to 1000 (but overlays can give effectively unlimited no. of statements).

Device-handler Capacity: 4 PDP-8 pages (plus system device handler). (System device handler must be 1 page only).

No. of OUTPUT files open on a device at one time: One only (but several INPUT and I-O files may be open on any device at any time).

FACILITIES NOT IMPLEMENTED ON THIS COMPILER
(But implemented on some large-machine compilers)

(i) Condition-Name (level-88) VALUE option (use condition statements instead).

(ii) USAGE and SYNCHRONIZED clauses (these are not needed since all internal data is stored as 6-bit characters).

(iii) No implied subjects or objects in IF statements (eg no "IF A = B OR C"). (use instead: "IF A = B OR A = C).

(iv) No AFTER function in the PERFORM verb (use nested PERFORMs instead).

(v) The TIMES option of the PERFORM verb is limited to 4095 TIMES (for larger numbers, use the VARYING option).

(vi) No COMPUTE verb (use a series of arithmetic verbs).

(vii) Only one OUTPUT file open on a device at one time (several INPUT or I-O files may be open on any device at any time).

(viii) Random-Access and Indexed-Sequential files must have fixed-length records.

(ix) No CURRENT-DATE data field (use a small file containing only the date).

TECHNICAL NOTES

These notes may be of interest to technical users of the compiler.

PROCEDURE DIVISION Code.

This sytem has been designed to run in a multi-programming environment ie the Run-Time system, when modified, will be capable of controlling a number of compiled programs. Hence the Run-Time system contains within it all the routines required to execute the various operations. Compiled "code" for PROCEDURE DIVISION statements consists only of links to these routines, together with parameters specifying details of the operations (data addresses, options etc). For example, the statement:-

MOVE A TO B

compiles to just three PDP-8 words - one allowing a link to the Run-Time MOVE routine and one each specifying the location of information describing the data items A and B.

Most COBOL statements take up only two, three or four PDP-8 words. This allows approximately 700 to 1000 statements in a program without using overlays.

Data Formats.

Internal data storage (within the DATA DIVISION and on files on file-structured devices) is in standard 6-bit code, two characters per PDP-8 word. The only exception is that the octal code 37 is used as the end-of-record mark, and two code 37 characters in succession represent an end-of-file mark. The corresponding 8-bit character ("underline" or "back-arrow") is not accepted as input by the compiler or Run-Time system. Note also that in COBOL, the carriage-return and line-feed characters are not represented internally or in file-structured files.

The Run-Time system writes data to non-file-structured devices (line-printer, Mag-tape, paper-tape punch etc) in standard 8-bit ASCII.

All input and output operations (except console operations) are performed by the OS/8 device handlers, in 256-word blocks. For file-structured data, 256 words is equivalent to 512 characters - but for non-file-structured data the OS/8 standard format of three 8-bit characters to two words is used.

Negative Numbers.

Negative numbers are represented in internal 6-bit code by clearing the first bit of the last digit of the number. For example, the number 123 would be stored as the octal codes 61-62-63, but the number -123 would be stored as 61-62-23 (ie the final octal 6, which in binary is 110, has been changed to 010, represented as octal 2). Hence the internal octal codes 20 to 31 represent the numbers 0 to 9 when they are the last digit of a numeric, negative data item.

6-bit and ASCII Codes

Notes: (i) The 6-bit codes 20 to 31 represent the numbers 0 to 9 respectively when they are the last digit of a negative numeric item.

(ii) Spaces are considered to be nulls (code 00) for purposes of sorting or alphanumeric comparison.

symbol	6-bit code (octal)	8-bit code (octal)		symbol	6-bit code (octal)	8-bit code (octal)
LOW-VALUE or @	00	300		blank	40	240
				!	41	241
				"	42	242
				#	43	243
A	01	301		$	44	244
B	02	302		%	45	245
C	03	303		&	46	246
D	04	304		'	47	247
E	05	305		(	50	250
F	06	306		)	51	251
G	07	307		*	52	252
H	10	310		+	53	253
I	11	311		,	54	254
J	12	312		-	55	255
K	13	313		.	56	256
L	14	314		/	57	257
M	15	315		0	60	260
N	16	316		1	61	261
O	17	317		2	62	262
P	20	320		3	63	263
Q	21	321		4	64	264
R	22	322		5	65	265
S	23	323		6	66	266
T	24	324		7	67	267
U	25	325		8	70	270
V	26	326		9	71	271
W	27	327		:	72	272
X	30	330		;	73	273
Y	31	331		<	74	274
Z	32	332		=	75	275
[	33	333		>	76	276
\	34	334				
]	35	335				
^ or ↑	36	336		HIGH-VALUE or ?	77	277
end-of-record or _, ←	37	(none)				

RESERVED WORDS

The COBOL verbs, and other words which have special significance to the compiler, are known as Reserved Words, and should not be used for Data-Names, File-Names, Paragraph-Names, etc. Reserved Words are written in capital letters wherever they appear in this booklet.

INDEX